CITHA | Inovação e Sustentabilidade na Amazônia

2025

INTELIGÊNCIA ARTIFICIAL

REDES NEURAIS

(dos fundamentos às arquiteturas CNN e RNN)

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA

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

Inteligência artificial [livro eletrônico] : redes neurais : (dos fundamentos às arquiteturas CNN e RNN) / Eduardo Palhares Jr....[et al.] ; prefácio por Nivaldo Rodrigues e Silva. -- Manaus, AM : Ed. dos Autores, 2025.
PDF

Outros autores: Wenndisson da Silva Souza, Alyson de Jesus dos Santos, Alexandre Lopes Martiniano, Nivaldo Rodrigues e Silva.
Vários colaboradores.
Bibliografia.
ISBN 978-65-01-59137-7

1. Ciência da computação 2. Informação - Sistema de armazenagem e recuperação 3. Inteligência artificial - Inovações tecnológicas 4. Redes neurais (Ciência da computação) I. Palhares Jr., Eduardo. II. Souza, Wenndisson da Silva. III. Santos, Alyson de Jesus dos. IV. Martiniano, Alexandre Lopes. V. Silva, Nivaldo Rodrigues e.

25-287138 CDD-006.3

Índices para catálogo sistemático:

1. Inteligência artificial : Ciência da computação 006.3

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

DOI: 10.5281/zenodo.16411289

Expediente do IFAM

Reitor
Jaime Cavalcante Alves

Pró-Reitor de Administração
Fábio Teixeira Lima

Pró-Reitor de Gestão de Pessoas
Leandro Amorim Damasceno

Pró-Reitora de Ensino
Rosângela Santos da Silva

Pró-Reitora de Extensão
Maria Francisca Morais de Lima

Pró-Reitor de Pesquisa, Pós-Graduação e Inovação
Paulo Henrique Rocha Aride

Diretor Geral do Campus Manaus Distrito Industrial
Nivaldo Rodrigues e Silva

Expediente do Projeto CITHA

Gestores
Nivaldo Rodrigues e Silva
Samirames da Silva Fleury
Alyson de Jesus dos Santos
Maria Cassiana Andrade Braga
Adanilton Rabelo de Andrade

Coordenadores
Tiago Francisco Andrade Diocesano
Jaidson Brandão da Costa
Elcivan dos Santos Silva
Martinho Correia Barros
Adelino Maia Galvão Filho

Expediente de Produção

Autores
Eduardo Palhares Jr.
Wenndisson da Silva Souza
Alyson de Jesus dos Santos
Alexandre Lopes Martiniano
Nivaldo Rodrigues e Silva

Avaliação Pedagógica
Samirames da Silva Fleury

Diagramadores
Wenndisson da Silva Souza
Eduardo Palhares Jr.
Fabio Serra Ribeiro Couto

Revisores de Texto
Alexandre Lopes Martiniano
Alyson de Jesus dos Santos

Inteligência Artificial — Redes Neurais
(dos fundamentos às arquiteturas CNN e RNN)

Autores
Eduardo Palhares Jr.
Wenndisson da Silva Souza
Alyson de Jesus dos Santos
Alexandre Lopes Martiniano
Nivaldo Rodrigues e Silva

Prefácio por Nivaldo Rodrigues e Silva

Revisão
Alexandre Lopes Martiniano
Alyson de Jesus dos Santos

1ª Edição

Manaus - AM
2025

Lista de Expressões para Enriquecimento de Conteúdo

Este material foi cuidadosamente estruturado para apoiar sua jornada de aprendizado. Ao longo dos capítulos, você encontrará diversas chamadas sinalizadas por ícones especiais, que ajudarão a destacar pontos-chave e enriquecer sua compreensão. Durante a diagramação, esses ícones serão inseridos conforme as indicações dos autores, guiando você para diferentes tipos de conteúdo e atividades que potencializam seu estudo.

Fique Alerta!

Destaque para conceitos, expressões e trechos fundamentais que merecem sua atenção especial para a compreensão do conteúdo.

Iniciando o diálogo...

Espaço para reflexão crítica. Aqui você será convidado(a) a problematizar os temas abordados, relacionando-os com sua experiência e buscando conexões relevantes para aprofundar seu aprendizado.

Conhecendo um pouco mais!

Indicação de fontes complementares, como livros, entrevistas, vídeos, aplicativos, links e outros recursos para ampliar seu conhecimento sobre o tema.

Caso Prático

Aplicação direta do conteúdo em exemplos concretos, para facilitar a fixação e demonstrar a utilidade do que foi aprendido.

Copie e Teste!

Trechos de código prontos para serem copiados e executados, para que você possa experimentar, validar e explorar na prática os conceitos estudados.

Prefácio

Vivemos em uma era definida pela complexidade dos dados e pela busca por uma inteligência artificial cada vez mais avançada. No epicentro desta revolução está o Deep Learning, ou Aprendizagem Profunda, um subcampo do Machine Learning que permite aos sistemas emular o raciocínio humano por meio de redes neurais profundas. Esta tecnologia tem sido a força motriz por trás de avanços extraordinários, da visão computacional ao processamento de linguagem natural. Este livro foi desenvolvido para guiar o leitor por uma jornada imersiva e acessível no universo do Deep Learning, desde os fundamentos das redes neurais até a construção de modelos de ponta.

O conteúdo foi cuidadosamente planejado para atender tanto iniciantes que dão seus primeiros passos em redes neurais quanto profissionais que desejam dominar arquiteturas complexas. No primeiro módulo, são apresentados os conceitos essenciais do Deep Learning, incluindo o funcionamento dos neurônios artificiais, as funções de ativação e o papel de frameworks como TensorFlow e PyTorch. O segundo módulo aprofunda-se em arquiteturas fundamentais, como Redes Neurais Convolucionais (CNNs) para análise de imagens e Redes Neurais Recorrentes (RNNs) para a compreensão de sequências e textos.

No terceiro módulo, abordamos os desafios do treinamento de modelos profundos, a otimização de hiperparâmetros, técnicas para evitar o overfitting e a avaliação de performance. Já o módulo final propõe um estudo de caso aplicado à área da saúde, demonstrando como modelos de Deep Learning podem ser usados para o diagnóstico de doenças a partir de imagens médicas, transformando a precisão e a velocidade da medicina moderna.

Mais do que um guia técnico, este livro busca despertar no leitor a capacidade de inovar e a curiosidade para explorar as fronteiras da inteligência artificial. Ao aplicar os conhecimentos adquiridos, o leitor será capaz de construir soluções que não apenas interpretam dados, mas que também percebem, criam e raciocinam. Que esta leitura inspire descobertas, estimule o pensamento criativo e fortaleça sua jornada no fascinante campo da Aprendizagem Profunda.

Projeto de Capacitação e Interiorização em Tecnologias Habilitadoras na Amazônia - CITHA

O projeto CITHA surge com o objetivo de fortalecer a economia da Amazônia por meio do incentivo ao empreendedorismo local e do desenvolvimento sustentável. Sua proposta é capacitar profissionais e impulsionar a criação de startups voltadas para a bioeconomia, além de apoiar cooperativas locais na melhoria de seus processos produtivos. A implementação de tecnologias inovadoras é uma das estratégias centrais do projeto, visando oferecer soluções eficientes que atendam às necessidades regionais, como a otimização dos recursos naturais e a melhoria da infraestrutura local.

Ao longo de sua execução, o projeto se compromete a integrar os diversos stakeholders, como governos, empresas, ONGs e comunidades, por meio da capacitação da mão de obra local. O objetivo é formar um capital intelectual qualificado, capaz de apoiar uma governança eficiente, promover a inovação e assegurar a sustentabilidade. O CITHA dedica-se à criação de processos internos que incentivem o desenvolvimento de novos métodos e tecnologias, adaptáveis às particularidades do território amazônico.

Em síntese, o projeto CITHA visa criar um ciclo de desenvolvimento que não só incentive o empreendedorismo, mas também promova a modernização das estruturas locais, elevando a qualidade de vida das populações da Amazônia. Focado em áreas como bioeconomia, inovação e transferência de tecnologia, o projeto busca estabelecer um ecossistema mais forte e autossustentável, capaz de responder eficientemente às demandas do mercado e da sociedade.

Sumário

1 Redes Neurais Densas **14**

1.1 Introdução ao Deep Learning . . . 14
1.1.1 Histórico e motivação . . . 14
1.1.2 Comparação com aprendizado tradicional . . . 15
1.1.3 Exemplos de aplicação em contextos amazônicos . . . 15
1.1.4 Aplicando seus conhecimentos . . . 16
1.2 Perceptron para o problema lógico AND . . . 17
1.3 Neurônio Artificial e Funções de Ativação . . . 20
1.3.1 Modelo matemático do neurônio . . . 20
1.3.2 Funções de ativação . . . 20
1.3.3 Interpretação gráfica . . . 23
1.3.4 Testando as Funções de Ativação . . . 23
1.3.5 Aplicando seus conhecimentos . . . 24
1.4 Propagação Direta e Cálculo da Saída . . . 26
1.4.1 Operações Vetoriais . . . 26
1.4.2 Cálculo da Saída de uma Rede com Múltiplas Camadas . . . 26
1.4.3 Ativação em Lote (Batch Activation) . . . 27
1.4.4 Propagação Direta em Camadas Densas . . . 28
1.4.5 Complexidade Computacional da Propagação Direta . . . 30
1.5 Função de Custo e Retropropagação . . . 32
1.5.1 Funções de Custo . . . 32
1.5.2 A Lógica por Trás da Retropropagação . . . 33
1.5.3 Fluxo do Gradiente . . . 33
1.5.4 Cálculo do Gradiente com Entropia Cruzada . . . 34
1.6 Treinamento e Implementação com Keras . . . 37
1.6.1 Montagem de uma rede densa . . . 37
1.6.2 Separação de dados . . . 37
1.6.3 Treinamento do modelo . . . 38
1.6.4 Avaliação e métricas . . . 38
1.6.5 Aplicando seus conhecimentos . . . 40
1.7 Regularização e Aprimoramentos . . . 41
1.7.1 Overfitting e Underfitting . . . 41
1.7.2 Dropout: desligando neurônios aleatoriamente . . . 41
1.7.3 Regularização L2 (ou *weight decay*) . . . 42
1.7.4 Taxa de aprendizado (*learning rate*) . . . 42
1.7.5 Otimizadores: SGD vs Adam . . . 43
1.7.6 Aplicando seus conhecimentos . . . 43
1.7.7 Por que os resultados do treinamento mudam? . . . 53

2 Redes Convolucionais (CNN) 55
2.1 Introdução à Visão Computacional . . . 55
2.1.1 Diferença entre imagens e dados tabulares . . . 56
2.1.2 Estrutura de dados de imagem . . . 57
2.1.3 Aplicando seus conhecimentos . . . 58
2.2 Operações de Convolução e Filtros . . . 61
2.2.1 Conceito de Kernel . . . 61
2.2.2 Convolução manual . . . 61
2.2.3 Interpretação de bordas e padrões . . . 63
2.2.4 Quadro Comparativo de Filtros Clássicos . . . 63
2.3 Padding, Stride e Pooling . . . 66
2.3.1 Redução de dimensão . . . 66
2.3.2 Detecção de padrões invariantes . . . 67
2.3.3 Max pooling e average pooling . . . 67
2.3.4 Aplicando seus conhecimentos . . . 67
2.3.5 Caso prático com imagem real . . . 69
2.4 Arquiteturas Típicas e Camadas CNN . . . 73
2.4.1 Camadas fundamentais: Conv2D, Pooling, Flatten e Dense . . . 73
2.4.2 Arquitetura LeNet simplificada . . . 73
2.4.3 Construindo sua própria CNN . . . 74
2.4.4 Aplicando seus conhecimentos . . . 76
2.5 O que a rede está vendo? Visualização de Filtros e Camadas . . . 81
2.5.1 O que são mapas de ativação? . . . 81
2.5.2 Como podemos extrair os mapas de ativação? . . . 81
2.5.3 Visualizando os filtros . . . 81
2.5.4 O que a rede está vendo? . . . 82
2.5.5 Caso Prático: Visualizando o MNIST . . . 82
2.5.6 O que aprendemos com as ativações? . . . 87

3 Redes Recorrentes (RNN) 88
3.1 Sequências e o Eixo Temporal . . . 88
3.1.1 Como assim, aprender com o tempo? . . . 88
3.1.2 A Diferença entre Dados Estáticos e Sequenciais . . . 89
3.2 RNN Básica: Estrutura e Intuição . . . 91
3.2.1 Representação desdobrada no tempo . . . 91
3.2.2 Compartilhamento de pesos . . . 91
3.2.3 Memória de curto prazo . . . 92
3.3 LSTM e GRU: Células com Memória Longa . . . 93
3.3.1 O que são LSTM e GRU? . . . 93
3.3.2 Intuição: como funcionam as portas? . . . 93
3.3.3 Comparando RNN, LSTM e GRU . . . 94
3.3.4 Memória no tempo: Umidade e Previsão do Clima . . . 94
3.4 Aplicações Simples de RNN . . . 96
3.4.1 Caso prático: Previsão de Temperatura . . . 96
3.5 Implementação Leve em Keras com Dados Meteorológicos do INMET . . . 100
3.5.1 Preparação dos dados . . . 100
3.5.2 Construção da sequência . . . 101
3.5.3 Treinando modelos com RNN, LSTM e GRU . . . 102

3.5.4 Comparando os resultados . . . 103
3.5.5 Previsão com o modelo GRU . . . 104
3.5.6 Aplicando seus conhecimentos . . . 107

4 Experimento Prático 108
4.1 Aplicação 01: Reconhecimento de Padrões em Solo . . . 108
4.1.1 Geração de Dados Simulados para Características do Solo . . . 108
4.1.2 Pré-processamento dos Dados . . . 111
4.1.3 Definição da Arquitetura da Rede Neural (Keras) . . . 112
4.1.4 Treinamento do Modelo . . . 114
4.1.5 Avaliação do Desempenho do Modelo e Visualização . . . 115
4.1.6 Interpretação dos Resultados e Discussão . . . 118
4.2 Aplicação 02: Previsão de Surtos de Pragas com Redes Neurais Recorrentes . . . 120
4.2.1 Geração de Dados Simulados para Previsão de Surtos de Pragas . . . 120
4.2.2 Pré-processamento dos Dados para Séries Temporais . . . 123
4.2.3 Definição da Arquitetura da Rede Neural Recorrente (LSTM) . . . 124
4.2.4 Treinamento do Modelo LSTM . . . 125
4.2.5 Avaliação do Desempenho do Modelo LSTM e Visualização . . . 126
4.2.6 Interpretação dos Resultados e Discussão Prática . . . 130
4.3 Aplicação 03: Diagnóstico de Doenças em Plantas com Redes Neurais Convolucionais . . . 131
4.3.1 Organização dos dados . . . 131
4.3.2 Carregar e preparar os dados (Imagens) . . . 134
4.3.3 Criar e Treinar a CNN . . . 136
4.3.4 Visualizar os Resultados do Treinamento . . . 138
4.3.5 Testando o Modelo em Tempo Real com a Câmera . . . 141
4.3.6 Interpretação dos Resultados e Discussão Prática . . . 144

Apresentação dos Autores

Eduardo Palhares Júnior
Currículo Lattes
Professor de Matemática – IFAM *campus* CMDI

Meu nome é Eduardo Palhares Júnior. Iniciei minha formação como Técnico em Eletrônica pela ETE Jorge Street (2005), e ao longo da minha trajetória acadêmica concluí graduações em Ciência e Tecnologia (2010), Engenharia Aeroespacial (2014) e Matemática (2017), além do Mestrado em Engenharia Mecânica (2015), todos pela Universidade Federal do ABC (UFABC), com foco em aprendizado de máquina. Atuei como técnico de TI, investidor e professor nas áreas de engenharia, educação financeira, investimentos, computação e inteligência artificial. Atualmente, sou professor de Matemática no IFAM – Campus Manaus Distrito Industrial (CMDI). Nesta obra, convido você a mergulhar no universo da Inteligência Artificial e do Deep Learning, explorando conceitos e ferramentas que transformam dados em decisões inteligentes. Juntos, vamos desvendar algoritmos, experimentar modelos e aplicar técnicas avançadas com potencial para impactar áreas como finanças, engenharia e educação. Que cada capítulo desta jornada seja um passo em direção à formação de um profissional confiante, criativo e preparado para inovar no século XXI.
"Sem saber que era impossível, foi lá e fez."

Wenndisson da Silva Souza
Currículo Lattes
Professor de Informática – IFAM *campus* CMDI

Olá, estudantes e entusiastas da tecnologia!

Sou Prof. Wenndisson do IFAM. Desde o início da minha carreira, a curiosidade sobre como a tecnologia pode moldar nosso mundo tem sido minha principal motivação. Essa busca me levou da graduação em Sistemas de Informação a um mestrado que integra tecnologia com a sustentabilidade das ciências ambientais na Amazônia.

Acredito que aprender sobre Inteligência Artificial, e especialmente sobre Deep Learning, é mais do que adquirir uma habilidade técnica; é ganhar o poder de resolver problemas complexos, desde a otimização de redes até a criação de um futuro mais sustentável. O conhecimento que estamos trazendo neste livro é uma porta de entrada para esse universo.

Peço que mergulhem neste material com a mesma curiosidade que me move. Apaixonem-se pelo potencial transformador do Deep Learning e descubram como vocês podem fazer a diferença.

Contem comigo e bons estudos!

Alyson de Jesus dos Santos
Currículo Lattes
Professor de Informática – IFAM *campus* CMDI

Olá, queridos alunos e alunas do Curso de Inteligência Artificial: Deep Learning, Espero que todos estejam com saúde e entusiasmo para avançarmos juntos neste importante aprendizado. Sou Alyson de Jesus dos Santos, professor do Instituto Federal do Amazonas (IFAM), Campus Manaus Distrito Industrial. Possuo Pós-doutorado pelo Instituto de Ciências Matemáticas e de Computação da USP (ICMC/USP), doutorado em Engenharia Elétrica pela COPPE/UFRJ, além de mestrado e graduação em Engenharia. Atuo na área de Ciência da Computação com foco em comunicações móveis, Inteligência Artificial, Visão Computacional, Aprendizado de Máquina e Internet das Coisas. Minha missão é compartilhar o conhecimento de forma clara e acessível, conectando a teoria com a prática, para que vocês desenvolvam uma base sólida em Deep Learning e suas aplicações reais. Desejo a todos um excelente curso e que este seja um passo decisivo na trajetória de cada um de vocês!

Alexandre Lopes Martiniano
Currículo Lattes
Professor de Engenharia Elétrica – IFAM *campus* CMDI

Olá a todos e todas! Sou Alexandre Lopes Martiniano, graduado em Engenharia Elétrica com habilitação em Eletrônica (2008), mestre em Engenharia Elétrica (2010) e atualmente doutorando na mesma área, com ênfase em Controle e Automação de Sistemas, pela Universidade Federal do Amazonas (UFAM). Atuo como professor no Instituto Federal de Educação, Ciência e Tecnologia do Amazonas (IFAM), no Campus Manaus Distrito Industrial (CMDI), desenvolvendo atividades nas áreas de Inteligência Computacional, Aprendizado de Máquina e Sistemas Embarcados. Neste curso, meu desejo é que você se sinta inspirado a explorar e aplicar os conhecimentos em Ciência de Dados e Deep Learning em diversos contextos, contribuindo para soluções mais inteligentes, eficientes e inovadoras. Desejo uma excelente jornada de aprendizado a todos!

Nivaldo Rodrigues e Silva
Currículo Lattes
Diretor do *campus* Manaus Distrito Industrial – IFAM

Queridos leitores, sejam muito bem-vindos a esta jornada pelo aprendizado de Inteligência Artificial e Deep Learning. Espero sinceramente que este material os inspire a pensar de forma criativa e a aplicar todo o conhecimento adquirido para gerar um impacto positivo e duradouro em diversas áreas, construindo juntos um futuro mais inteligente e sustentável. Sou Nivaldo Rodrigues e Silva, doutorando em Engenharia de Recursos Naturais da Amazônia, mestre em Ciências e Meio Ambiente, com formação em Engenharia Elétrica e Licenciatura em Matemática. Como professor no Instituto Federal de Educação, Ciência e Tecnologia do Amazonas (IFAM), busco integrar tecnologia e educação para oferecer soluções inovadoras, alinhadas às demandas e potencialidades da região amazônica. Espero que este material capacite e inspire vocês a aplicar a Inteligência Artificial e Deep Learning de forma criativa e impactante, contribuindo para a construção de um futuro mais sustentável e promissor.

Capítulo 1

Redes Neurais Densas

Iniciando o diálogo...

Você já se perguntou como o cérebro humano aprende a reconhecer rostos, entender frases ou resolver problemas? E se disséssemos que é possível ensinar um computador a fazer algo parecido? Neste capítulo, você vai conhecer as **Redes Neurais Densas**, um dos pilares da inteligência artificial moderna. Vamos entender como esses modelos inspirados no cérebro funcionam, aprendem com dados e tomam decisões. Prepare-se para construir sua primeira rede neural e dar os primeiros passos rumo ao aprendizado de máquina!

1.1 Introdução ao Deep Learning

O *Deep Learning*, ou aprendizado profundo, é uma subárea do aprendizado de máquina que tem revolucionado o campo da inteligência artificial nos últimos anos [12]. Ele se destaca não apenas por sua capacidade de resolver problemas complexos e variados, como reconhecimento de voz, tradução automática, análise de imagens e jogos, mas também por seu diferencial em aprender diretamente a partir dos dados brutos, dispensando a necessidade de regras e características feitas manualmente.

Essa autonomia para extrair padrões e representações internas dos dados em múltiplos níveis hierárquicos faz do *Deep Learning* uma tecnologia especialmente poderosa para lidar com a enorme variedade e volume de dados que hoje estão disponíveis em diversas áreas. Além disso, seu desenvolvimento tem impulsionado avanços em setores antes inimagináveis, como medicina, agricultura de precisão, veículos autônomos e sistemas inteligentes de recomendação, modificando profundamente a forma como interagimos com máquinas e automatizamos processos complexos.

1.1.1 Histórico e motivação

As origens do *Deep Learning* estão profundamente ligadas às redes neurais artificiais, que foram inspiradas pelo funcionamento do cérebro humano. Nas décadas de 1940 e 1950, pesquisadores como *Warren McCulloch* e *Walter Pitts* propuseram os primeiros modelos matemáticos de neurônios artificiais [3]. Posteriormente, o Perceptron, criado por *Frank Rosenblatt* em 1958, representou uma das primeiras implementações práticas de uma rede neural simples [6].

Apesar do entusiasmo inicial, os primeiros modelos de redes neurais enfrentaram limitações significativas, principalmente pela falta de capacidade computacional e de grandes

volumes de dados para treinamento. Além disso, problemas como o ”desvanecimento do gradiente”dificultavam o treinamento de redes mais profundas, limitando seu uso a problemas relativamente simples [3].

Foi somente no início dos anos 2000, com o avanço do poder computacional, a disponibilidade massiva de dados digitais e o desenvolvimento de algoritmos eficientes de treinamento, que o *Deep Learning* começou a mostrar resultados superiores. A introdução de arquiteturas profundas com múltiplas camadas, como as redes neurais convolucionais (CNNs) e recorrentes (RNNs), possibilitou o aprendizado de representações hierárquicas dos dados, capturando padrões cada vez mais abstratos e complexos [2, 3].

Atualmente, o *Deep Learning* está no centro de diversas inovações tecnológicas, sendo aplicado em áreas como processamento de linguagem natural, visão computacional, jogos e veículos autônomos, entre outras [1].

1.1.2 Comparação com aprendizado tradicional

O aprendizado tradicional de máquina, também conhecido como aprendizado superficial, geralmente depende de técnicas onde o engenheiro de dados ou cientista precisa definir manualmente as características (features) relevantes que descrevem o problema. Esse processo, chamado de engenharia de características, exige conhecimento profundo do domínio e pode ser bastante trabalhoso [3].

Por exemplo, para uma tarefa de classificação de imagens usando aprendizado tradicional, especialistas precisam extrair manualmente características como bordas, texturas e formas, utilizando técnicas de processamento de imagens clássicas [2]. Essas características são então utilizadas como entrada para algoritmos como máquinas de vetores de suporte (SVM), árvores de decisão ou regressão logística.

Em contrapartida, o *Deep Learning* automatiza a extração de características, trabalhando diretamente com os dados brutos. Redes neurais profundas são capazes de aprender múltiplos níveis de representação, das características mais simples nas camadas iniciais até conceitos abstratos nas camadas profundas. Isso elimina a necessidade da engenharia manual intensiva e, muitas vezes, resulta em desempenho superior, especialmente quando há grande quantidade de dados disponíveis [1].

Além disso, o *Deep Learning* possui a capacidade de generalizar melhor para dados não vistos, desde que bem treinado, e consegue adaptar-se a diversos tipos de dados, incluindo imagens, texto, áudio e séries temporais, tornando-se uma ferramenta versátil para uma ampla gama de aplicações [2, 3].

1.1.3 Exemplos de aplicação em contextos amazônicos

A região amazônica é um ambiente de extrema importância ecológica e social, com vastos recursos naturais e desafios complexos. O *Deep Learning* tem sido empregado para apoiar o desenvolvimento sustentável e a conservação ambiental nessa região por meio de diversas aplicações inovadoras:

- **Monitoramento do desmatamento:** Através do processamento de imagens de satélite, modelos de *Deep Learning* são treinados para identificar áreas de desmatamento ilegal em tempo quase real. Isso permite que órgãos ambientais tomem medidas rápidas para conter atividades ilegais, contribuindo para a preservação da floresta.

- **Previsão climática:** Modelos baseados em redes neurais recorrentes analisam dados históricos meteorológicos para prever eventos climáticos extremos, como enchentes e secas. Essas previsões auxiliam na preparação da população e na tomada de decisões governamentais e agrícolas.
- **Agricultura de precisão:** Sistemas inteligentes que utilizam *Deep Learning* ajudam pequenos agricultores a monitorar a saúde das plantações, detectar doenças precocemente e otimizar o uso de recursos como água e fertilizantes, aumentando a produtividade e minimizando impactos ambientais.
- **Reconhecimento de espécies e biodiversidade:** Aplicações de visão computacional auxiliam na identificação automática de espécies animais e vegetais a partir de imagens e sons, facilitando pesquisas científicas e o monitoramento da biodiversidade local.

Essas aplicações demonstram como o *Deep Learning* pode ser uma poderosa ferramenta para promover o equilíbrio entre desenvolvimento econômico, inclusão social e preservação ambiental na Amazônia.

Fique Alerta!

Apesar do enorme potencial, o sucesso do *Deep Learning* depende fortemente da qualidade e quantidade dos dados disponíveis, do poder computacional e do conhecimento técnico para a modelagem e interpretação dos resultados.

Conhecendo um pouco mais!

Para aprofundar seus conhecimentos, recomendamos a leitura do livro *Deep Learning*, de Ian Goodfellow, Yoshua Bengio e Aaron Courville, além dos recursos disponíveis na biblioteca do IFAM e cursos especializados em plataformas como Coursera e edX.

1.1.4 Aplicando seus conhecimentos

1. Explique com suas palavras o que diferencia o *Deep Learning* do aprendizado tradicional. Cite exemplos práticos.
2. Pesquise e descreva três aplicações do *Deep Learning* fora da Amazônia que você considera impactantes.
3. Analise um artigo científico que use *Deep Learning* para monitorar o meio ambiente. Quais dados foram usados? Quais foram os resultados?
4. Discuta os principais desafios que a região amazônica enfrenta para aplicar *Deep Learning* em larga escala.
5. Implemente um perceptron simples em Python para resolver um problema de classificação binária (por exemplo, AND lógico).

1.2 Perceptron para o problema lógico AND

Caso Prático

Nesta seção, implementamos um perceptron simples em Python para resolver o problema do AND lógico, um exemplo clássico para ilustrar redes neurais básicas.

O problema consiste em treinar uma rede neural com duas entradas binárias (0 ou 1), de forma que ela aprenda a reproduzir a função lógica AND, cuja saída é 1 apenas quando ambas as entradas forem 1. Este é um ótimo ponto de partida para introduzir conceitos como pesos, bias, função de ativação e atualização por erro.

Além de didático, este caso prático mostra na prática como um modelo simples pode aprender padrões por meio de iteração e ajuste de parâmetros, o mesmo princípio que será expandido em redes mais complexas ao longo do livro.

Copie e Teste!

```
import numpy as np

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        self.weights = np.zeros(input_size + 1) # +1 para o bias
        self.lr = learning_rate

    def activation(self, x):
        return 1 if x >= 0 else 0

    def predict(self, x):
        x_with_bias = np.insert(x, 0, 1) # Adiciona bias na
   entrada
        weighted_sum = np.dot(self.weights, x_with_bias)
        return self.activation(weighted_sum)

    def train(self, training_inputs, labels, epochs=10):
        for _ in range(epochs):
            for x_val, label in zip(training_inputs, labels):
                prediction = self.predict(x_val)
                error = label - prediction
                x_with_bias = np.insert(x_val, 0, 1)
                self.weights += self.lr * error * x_with_bias

if __name__ == '__main__':
    inputs = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
```

```
    labels = np.array([0, 0, 0, 1])
    perceptron = Perceptron(input_size=2)
    perceptron.train(inputs, labels, epochs=10)

    print("Pesos finais:", perceptron.weights)
    print("Testes:")
    for x_val in inputs:
        print(f"Entrada: {x_val}, Saída: {perceptron.predict(x_val
)}")
```

Fique Alerta!

Observe que o perceptron utiliza uma função de ativação simples (degrau) e ajusta os pesos iterativamente para minimizar erros.

Saída Esperada

```
Pesos finais: [-0.2 0.2 0.1]
Testes:
Entrada: [0 0], Saída: 0
Entrada: [0 1], Saída: 0
Entrada: [1 0], Saída: 0
Entrada: [1 1], Saída: 1
```

Explicação detalhada do código

- **Inicialização:** O perceptron é criado com pesos zerados, incluindo um peso extra para o *bias*, que permite deslocar a função de ativação.
- **Função de ativação:** A função degrau retorna 1 se a soma ponderada das entradas for maior ou igual a zero, caso contrário, retorna 0. É o critério de decisão do perceptron.
- **Previsão (`predict`):** Recebe uma entrada, adiciona o bias, calcula o produto escalar entre pesos e entradas, e aplica a função de ativação para gerar a saída.
- **Treinamento:** Para cada época (ciclo de treinamento), o perceptron avalia cada entrada, calcula o erro (diferença entre saída desejada e saída prevista) e atualiza os pesos proporcionalmente ao erro e à taxa de aprendizado.
- **Dados de treinamento:** O problema lógico AND tem quatro combinações de entradas binárias e as saídas esperadas correspondentes.
- **Resultado:** Após o treinamento, os pesos ajustados são impressos e o perceptron é testado para confirmar que aprendeu a função AND corretamente.

Importância do Perceptron

Este exemplo simples ilustra o princípio básico das redes neurais artificiais: ajustar parâmetros internos para mapear entradas a saídas desejadas. Embora o perceptron tenha limitações (como não resolver problemas não linearmente separáveis), ele é a base para arquiteturas mais complexas que compõem o *Deep Learning*.

Conhecendo um pouco mais!

- Neural Networks and *Deep Learning* - Coursera (Andrew Ng)
- Deep Learning - 3Blue1Brown (YouTube)

1.3 Neurônio Artificial e Funções de Ativação

1.3.1 Modelo matemático do neurônio

Um neurônio artificial é a unidade fundamental das redes neurais. Inspirado nos neurônios biológicos, ele recebe um conjunto de entradas $x_1, x_2, \ldots, x_n$, cada uma com um peso associado $w_1, w_2, \ldots, w_n$, e calcula uma saída baseada em uma função de ativação.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{1.1}$$

$$y = \phi(z) \tag{1.2}$$

Onde:

- z: soma ponderada das entradas (também chamada de entrada líquida)
- b: termo de bias, que permite o deslocamento da função de ativação
- $\phi(z)$: função de ativação que determina a saída do neurônio

Conhecendo um pouco mais!

O termo de bias b funciona como um limiar ajustável. Mesmo que todas as entradas sejam zero, o neurônio ainda pode produzir uma saída diferente de zero, graças ao bias.

1.3.2 Funções de ativação

As funções de ativação determinam a saída de um neurônio com base na entrada líquida. Abaixo, listamos as mais utilizadas:

Função Degrau (Step Function)

$$\phi(z) = \begin{cases} 1, & \text{se } z \geq 0 \\ 0, & \text{caso contrário} \end{cases} \tag{1.3}$$

É usada em classificadores binários simples, como o Perceptron original.

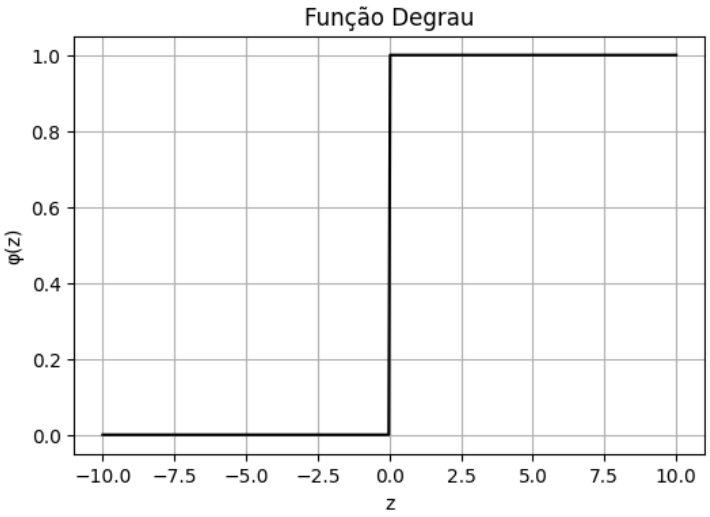

Figura 1.1: Gráfico da função degrau

Fique Alerta!

Apesar de simples, a função degrau não é diferenciável, o que dificulta seu uso em redes neurais treinadas com gradiente descendente [3].

Função Sigmoide

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{1.4}$$

Transforma a saída em um valor contínuo entre 0 e 1. Ideal para problemas de classificação binária.

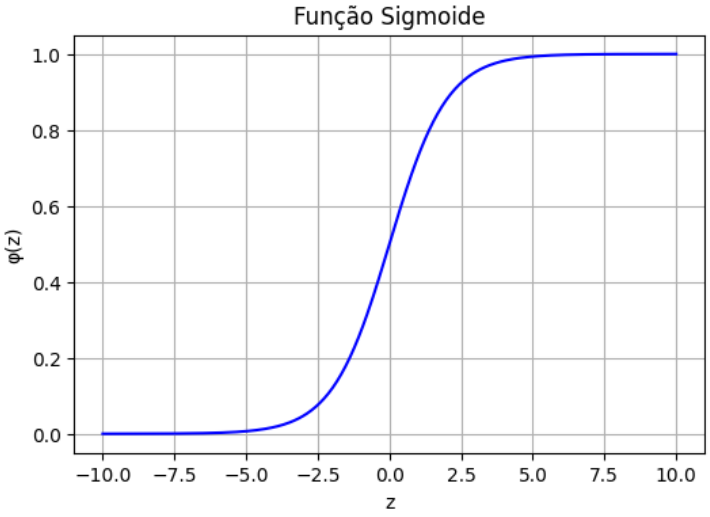

Figura 1.2: Gráfico da função sigmoide

Função Tanh (Tangente Hiperbólica)

$$\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1.5}$$

Semelhante à sigmoide, mas com saída entre -1 e 1. Costuma ser preferida por apresentar saída centrada em zero [2].

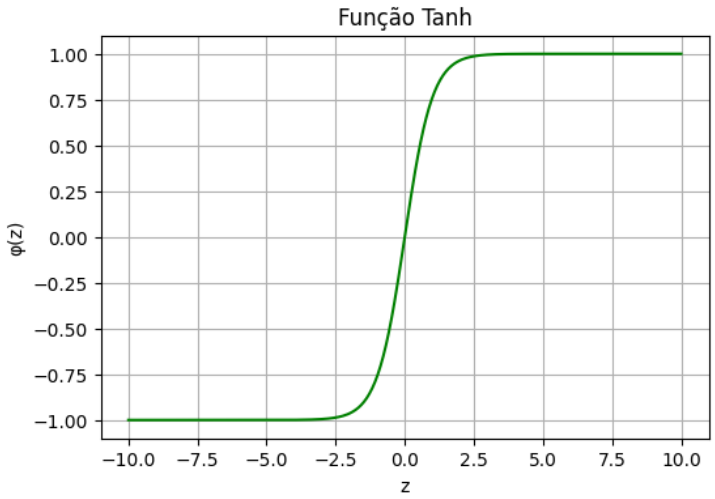

Figura 1.3: Gráfico da função tangente hiperbólica

Função ReLU (Rectified Linear Unit)

$$\phi(z) = \max(0, z) \tag{1.6}$$

Atualmente, é a função mais usada em redes profundas por sua simplicidade e eficiência [1, 2].

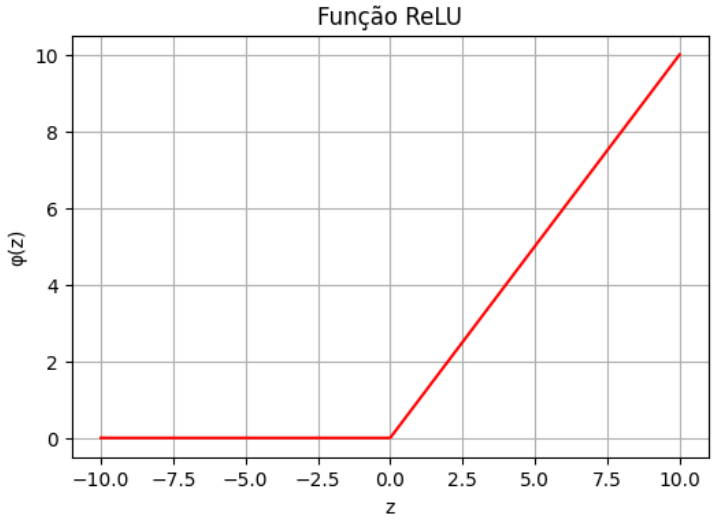

Figura 1.4: Gráfico da função ReLU

Fique Alerta!

A ReLU pode ”morrer”durante o treinamento se os pesos a tornarem negativa para todas as entradas. Uma variação chamada Leaky ReLU resolve isso [2].

1.3.3 Interpretação gráfica

Cada função de ativação possui uma curva característica. As curvas mostram como o neurônio responde a diferentes valores da entrada líquida z.

- A função degrau é abrupta e não contínua.
- A sigmoide e a tanh são suaves, mas saturam em extremos (problema de gradiente).
- A ReLU é linear para valores positivos e zero para negativos.

Essas diferenças impactam diretamente o aprendizado da rede neural. Ao longo do livro, veremos quando e por que escolher cada uma dessas funções.

1.3.4 Testando as Funções de Ativação

Caso Prático

No próximo exemplo, construiremos um neurônio artificial em Python com o objetivo de observar como diferentes funções de ativação influenciam sua saída. A estrutura do neurônio é simples: ele realiza o somatório ponderado das entradas e aplica uma função não linear sobre o resultado, produzindo assim sua saída.

Ao variar apenas a função de ativação, mantendo os mesmos valores de entrada, pesos e bias, poderemos visualizar como cada função responde de maneira distinta ao mesmo estímulo. Esse tipo de comparação é fundamental para entender o papel da não linearidade no aprendizado de redes neurais profundas.

Com esse estudo de caso, você irá:

- Implementar passo a passo um neurônio artificial com Python puro.
- Aplicar as funções Degrau, Sigmoide, Tanh e ReLU no mesmo cenário.
- Analisar os efeitos das diferentes ativações sobre a saída do neurônio.

Esse exercício marca a transição entre a teoria matemática do neurônio e sua implementação prática em código, preparando o terreno para construir redes neurais maiores nos capítulos seguintes.

Copie e Teste!

```
import numpy as np

# Funções de ativação
```

```python
def step(z):
    return 1 if z >= 0 else 0

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

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

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

# Neurônio simples
def neuron(x, w, b, activation_func):
    z = np.dot(w, x) + b
    return activation_func(z)

# Exemplo
x_input = np.array([0.5, 0.8])
weights = np.array([0.4, -0.6])
bias = 0.1

print("Entradas:", x_input, "Pesos:", weights, "Bias:", bias)
print("Soma ponderada (z):", np.dot(weights, x_input) + bias) # z
    = 0.4*0.5 + (-0.6)*0.8 + 0.1 = 0.2 - 0.48 + 0.1 = -0.18
print("-" * 30)
print("Saída com Degrau:", neuron(x_input, weights, bias, step))
print("Saída com Sigmoide:", neuron(x_input, weights, bias,
    sigmoid))
print("Saída com Tanh:", neuron(x_input, weights, bias, tanh))
print("Saída com ReLU:", neuron(x_input, weights, bias, relu))
```

Saída Esperada

```
Entradas: [0.5 0.8]
Pesos: [0.4 -0.6]
Bias: 0.1
Soma ponderada (z): -0.17999999999999997
------------------------------
Saída com Degrau: 0
Saída com Sigmoide: 0.45512110762641994
Saída com Tanh: -0.17808086811733015
Saída com ReLU: 0.0
```

1.3.5 Aplicando seus conhecimentos

1. **Variação de Entradas:** Altere os valores da entrada $x = [0.5, 0.8]$ para outros vetores, como $[1.0, 1.0]$, $[-0.5, 0.3]$, ou $[0, 0]$. Como as saídas do neurônio mudam para cada função de ativação?

2. **Explorando o Bias:** Mude o valor do bias b para 0, -0.5 e 1.0. O que acontece com as saídas? Como o bias afeta o ponto de ativação?

3. **ReLU vs. Sigmoide:** Identifique situações em que a função ReLU gera saída 0, mas a Sigmoide gera um valor próximo de 0.5. O que isso indica sobre a sensibilidade de cada ativação?

4. **Construindo sua própria ativação:** Crie uma nova função chamada `leaky_relu` que retorna:

$$\phi(z) = \begin{cases} z, & \text{se } z > 0 \\ 0.01z, & \text{caso contrário} \end{cases}$$

   Teste-a no mesmo neurônio e compare com a ReLU original.

Copie e Teste!

```python
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha * z, z) # Forma vetorizada
    # Ou: return z if z > 0 else alpha * z # Forma escalar

# Usando os mesmos x_input, weights, bias de antes (z = -0.18)
print("-" * 30)
print("Saída com Leaky ReLU:", neuron(x_input, weights, bias,
   leaky_relu))
```

Saída Esperada

```
------------------------------
Saída com Leaky ReLU: -0.0017999999999999997
```

Fique Alerta!

A vantagem da `leaky_relu` é evitar que o neurônio ”morra”(retorne sempre zero) em regiões negativas, mantendo um pequeno gradiente para continuar aprendendo.

1.4 Propagação Direta e Cálculo da Saída

A propagação direta (forward propagation) é o processo fundamental pelo qual uma rede neural calcula a saída a partir das entradas, passando os dados pelas camadas da rede, aplicando pesos, somas e funções de ativação.

Nesta seção, abordaremos os conceitos essenciais que envolvem operações vetoriais para otimizar esses cálculos, o procedimento para calcular a saída de redes com múltiplas camadas e o conceito de ativação em lote, muito usado para eficiência computacional.

1.4.1 Operações Vetoriais

Para redes neurais, o uso de operações vetoriais e matriciais é crucial para eficiência e clareza na implementação. Vamos considerar uma camada densa típica com n neurônios.

Seja $\mathbf{x} \in \mathbb{R}^m$ o vetor de entrada da camada, $\mathbf{W} \in \mathbb{R}^{n \times m}$ a matriz de pesos, e $\mathbf{b} \in \mathbb{R}^n$ o vetor de vieses (bias). A saída linear $\mathbf{z} \in \mathbb{R}^n$ da camada é dada por:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Em vez de calcular separadamente cada saída para cada neurônio, essa formulação vetorial permite um cálculo simultâneo e eficiente.

- **Vetor de entrada** $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T$
- **Matriz de pesos** $\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix}$
- **Vetor de vieses** $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$

Após o cálculo linear, aplicamos a função de ativação elemento a elemento para obter a saída da camada:

$$\mathbf{a} = f(\mathbf{z}) = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix}$$

Exemplo: Se a função de ativação for a função sigmoide, então

$$a_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}, \quad i = 1, \ldots, n$$

1.4.2 Cálculo da Saída de uma Rede com Múltiplas Camadas

Considere uma rede neural com L camadas, onde a camada l tem pesos $\mathbf{W}^{(l)}$, vieses $\mathbf{b}^{(l)}$ e função de ativação $f^{(l)}$. A entrada para a primeira camada é o vetor de entrada do modelo $\mathbf{x}$. O cálculo da saída se dá de forma iterativa:

$$\begin{cases} \mathbf{a}^{(0)} = \mathbf{x} \\ \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, \ldots, L \\ \mathbf{a}^{(l)} = f^{(l)}\left(\mathbf{z}^{(l)}\right), \quad l = 1, \ldots, L \end{cases}$$

A saída final da rede é $\mathbf{a}^{(L)}$, que pode representar a predição do modelo.
Observações:

- Cada camada transforma a saída da camada anterior.
- As funções de ativação são geralmente não lineares, permitindo que a rede aprenda representações complexas.
- Para tarefas de classificação, a última camada pode usar uma ativação especial, como softmax, para produzir probabilidades.

1.4.3 Ativação em Lote (Batch Activation)

Na prática, o treinamento e a inferência são realizados processando múltiplos exemplos simultaneamente, chamados de *batch* (lote). Isso permite melhor aproveitamento do hardware (como GPUs) e eficiência.

Sejam m exemplos de entrada organizados na matriz

$$\mathbf{X} = \begin{bmatrix} - & \mathbf{x}^{(1)} & - \\ - & \mathbf{x}^{(2)} & - \\ & \vdots & \\ - & \mathbf{x}^{(m)} & - \end{bmatrix} \in \mathbb{R}^{m \times d}$$

onde d é o número de características da entrada.
O cálculo da saída da camada para o lote todo pode ser feito com:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}^T + \mathbf{b}^T$$

onde:

- $\mathbf{X} \in \mathbb{R}^{m \times d}$
- $\mathbf{W} \in \mathbb{R}^{n \times d}$
- $\mathbf{b} \in \mathbb{R}^n$
- Resultado $\mathbf{Z} \in \mathbb{R}^{m \times n}$

Em seguida, aplica-se a função de ativação elemento a elemento à matriz $\mathbf{Z}$ para obter a matriz de ativações $\mathbf{A}$:

$$\mathbf{A} = f(\mathbf{Z})$$

Cada linha de $\mathbf{A}$ corresponde à saída da camada para um exemplo do lote.

Vantagens do processamento em lote:

- Paralelização dos cálculos para maior velocidade.

- Estabilização do processo de otimização (atualizações mais robustas).
- Melhor utilização da memória e caches da CPU/GPU.

1.4.4 Propagação Direta em Camadas Densas

Caso Prático

Neste exemplo, vamos entender como calcular a saída de camadas densas (totalmente conectadas) em uma rede neural, utilizando operações vetoriais e processando várias amostras ao mesmo tempo (em lote). Vamos construir esse raciocínio passo a passo, da propagação em uma única camada até o uso de múltiplas camadas com diferentes funções de ativação.

Imagine que temos um conjunto de amostras com múltiplas características (como altura, peso, idade etc.), e queremos processá-las em uma rede com várias camadas densas.

Cenário inicial:

- Um lote de m amostras (linhas),
- Cada amostra com d características (colunas),
- Uma camada com n neurônios.

Organizamos:

- Entrada: $\mathbf{X} \in \mathbb{R}^{m \times d}$,
- Pesos da camada: $\mathbf{W} \in \mathbb{R}^{n \times d}$,
- Viés (bias): vetor $\mathbf{b} \in \mathbb{R}^{n}$.

O cálculo da propagação direta vetorizada consiste em:

$$\mathbf{Z} = \mathbf{X} \cdot \mathbf{W}^T + \mathbf{b}$$

$$\mathbf{A} = \sigma(\mathbf{Z})$$

onde σ é a função de ativação.

Por que vetorizado? Evitamos múltiplos loops e aproveitamos otimizações de desempenho numérico.

- Entrada: $\mathbf{X}$ $(m \times d)$
- Pesos: $\mathbf{W}$ $(n \times d)$
- Saída (após ativação): $\mathbf{A}$ $(m \times n)$

Exemplo Prático: Rede com Duas Camadas

Objetivo: Calcular a saída de uma rede com duas camadas densas para um lote de 2 amostras. Usaremos a função ReLU na primeira camada e a função sigmoide na segunda (saída).

- Entrada $X \in \mathbb{R}^{2\times 3}$:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{bmatrix}$$

- Camada 1: $W^{(1)} \in \mathbb{R}^{4\times 3}$, $b^{(1)} \in \mathbb{R}^4$

$$W^{(1)} = \begin{bmatrix} 0.2 & 0.4 & 0.6 \\ 0.1 & 0.3 & 0.5 \\ 0.7 & 0.8 & 0.9 \\ 0.5 & 0.4 & 0.3 \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix}$$

- Camada 2: $W^{(2)} \in \mathbb{R}^{2\times 4}$, $b^{(2)} \in \mathbb{R}^2$

$$W^{(2)} = \begin{bmatrix} 0.3 & 0.6 & 0.9 & 0.2 \\ 0.1 & 0.4 & 0.7 & 0.8 \end{bmatrix}, \quad b^{(2)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

Copie e Teste!

```
import numpy as np

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

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Entrada
X = np.array([[1.0, 2.0, 3.0],
[4.0, 5.0, 6.0]])

# Camada 1
W1 = np.array([[0.2, 0.4, 0.6],
[0.1, 0.3, 0.5],
[0.7, 0.8, 0.9],
[0.5, 0.4, 0.3]])
b1 = np.array([0.1, 0.2, 0.3, 0.4])

Z1 = np.dot(X, W1.T) + b1
A1 = relu(Z1)

# Camada 2
W2 = np.array([[0.3, 0.6, 0.9, 0.2],[0.1, 0.4, 0.7, 0.8]])
b2 = np.array([0.1, 0.2])

Z2 = np.dot(A1, W2.T) + b2
```

```
A2 = sigmoid(Z2)

print("Saída da camada 1 (ReLU):\n", A1)
print("Saída final da rede (sigmoide):\n", A2)
```

Saída Esperada

```
Saída da camada 1 (ReLU):
[[ 2.9  2.4  5.3  2.6]
[ 6.5  5.1 12.5  6.2]]

Saída final da rede (sigmoide):
[[0.99954738 0.9992832 ]
[0.99999998 0.99999994]]
```

(Valores aproximados, podem variar conforme a implementação.)

O que foi feito nesse código?

- Entradas com 3 características *(features)* foram passadas por uma camada com 4 neurônios, usando a função de ativação ReLU.
- A saída dessa primeira camada foi enviada para uma segunda camada com 2 neurônios, utilizando a função de ativação sigmoide.
- O resultado final são 2 valores entre 0 e 1 para cada exemplo, interpretados como probabilidades.

Essas probabilidades poderiam representar, por exemplo:

- A chance de uma planta crescer ou não.
- A chance de desmatamento ocorrer em duas regiões.
- Qualquer outro problema com duas saídas binárias.

1.4.5 Complexidade Computacional da Propagação Direta

A propagação direta (ou *forward pass*) em uma rede neural envolve basicamente:

- Produtos matriciais (multiplicações entre matrizes de pesos e vetores de entrada),
- Somas com os vetores de viés (bias),
- Aplicações das funções de ativação (como ReLU ou sigmoide) em cada camada.

Compreender a complexidade dessas operações é importante para estimar o custo computacional, especialmente em tarefas que envolvem grandes conjuntos de dados ou arquiteturas profundas.

Exemplo didático Vamos analisar uma rede simples com as seguintes características:

$$m = 64, \quad n_0 = 100, \quad n_1 = 50, \quad n_2 = 10$$

- m: número de exemplos no lote (batch size),
- n_0: número de neurônios na camada de entrada (100 características),
- n_1: número de neurônios na primeira camada oculta,
- n_2: número de neurônios na camada de saída.

Quantas operações são feitas? A quantidade de operações de multiplicação e soma em cada camada pode ser estimada como:

$$\text{Camada 1: } m \times (n_0 \times n_1) = 64 \times (100 \times 50) = 64 \times 5000$$
$$\text{Camada 2: } m \times (n_1 \times n_2) = 64 \times (50 \times 10) = 64 \times 500$$

Total de operações:

$$64 \times (100 \times 50 + 50 \times 10) = 64 \times (5000 + 500) = 64 \times 5500 = 352000 \text{ operações}$$

Fique Alerta!

Isso significa que, para processar 64 exemplos de entrada, essa rede realiza cerca de 352 mil operações aritméticas (sem contar funções de ativação ou normalização). Isso nos dá uma noção do esforço computacional envolvido, e esse número cresce rapidamente com redes maiores e mais profundas.

1.5 Função de Custo e Retropropagação

Nesta seção, discutiremos como as redes neurais aprendem ajustando seus pesos com base no erro entre a saída prevista e a real. Veremos duas funções de custo comuns e uma introdução intuitiva à retropropagação.

1.5.1 Funções de Custo

Erro Quadrático Médio (MSE)

O Erro Quadrático Médio (MSE, do inglês *Mean Squared Error*) é uma das funções de custo mais tradicionais e amplamente utilizadas, especialmente em problemas de regressão, onde o objetivo é prever valores contínuos.

A fórmula é dada por:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Aqui:

- m é o número total de exemplos de treino.
- $y^{(i)}$ é o valor real (rótulo) do exemplo i.
- $\hat{y}^{(i)}$ é o valor previsto pela rede para o exemplo i.

O MSE calcula a média das diferenças quadradas entre o valor previsto e o real. Por elevar ao quadrado essas diferenças, erros maiores são penalizados com força crescente. Isso ajuda a rede a focar em corrigir grandes discrepâncias.

Fique Alerta!

Quando o valor previsto está longe do valor real, o erro aumenta de forma quadrática, penalizando fortemente previsões ruins.

Exemplo: Suponha que a saída real de uma planta seja 5 cm de crescimento e o modelo preveja 3 cm. O erro para esse exemplo seria $(5-3)^2 = 4$. Agora, se o modelo prever 1 cm, o erro será $(5-1)^2 = 16$, quatro vezes maior, mostrando que erros maiores são mais punidos.

Entropia Cruzada *(Cross-Entropy)*

Para problemas de classificação, especialmente quando queremos decidir entre duas ou mais classes, o erro quadrático pode não ser a melhor escolha. Nesses casos, a função de custo mais comum é a *entropia cruzada*, que mede a distância entre duas distribuições de probabilidade: a verdadeira e a prevista [3].

A fórmula para classificação binária é:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(\hat{y}^{(i)}\right) + (1 - y^{(i)}) \log\left(1 - \hat{y}^{(i)}\right)\right]$$

Aqui, $y^{(i)}$ é 0 ou 1, e $\hat{y}^{(i)}$ é a probabilidade prevista da classe 1.

A entropia cruzada penaliza com muita força as previsões erradas feitas com alta confiança. Por exemplo, se a rede prevê 0.99 para a classe 1, mas o rótulo real é 0, o erro será muito alto. Isso força o modelo a ser cauteloso e ajustar as probabilidades para refletir melhor a incerteza.

Fique Alerta!

Ela é mais sensível a classificações incorretas do que o MSE, ajudando a melhorar a qualidade do modelo em tarefas de classificação.

Exemplo: Se a saída verdadeira é 1 (positivo) e a rede prevê 0.9, o erro será baixo, porque a previsão está correta e confiante. Mas se a rede prevê 0.1, o erro será grande, pois há uma alta confiança na classe errada.

1.5.2 A Lógica por Trás da Retropropagação

O aprendizado da rede neural acontece ajustando os pesos para minimizar a função de custo. Para isso, usamos o algoritmo de retropropagação, que permite calcular eficientemente o gradiente (a taxa de variação) da função de custo em relação a cada peso.

O processo pode ser entendido em três passos principais:

- **Cálculo das derivadas:** Calcula-se como a função de custo muda conforme alteramos cada peso, usando cálculo diferencial.
- **Propagação do erro para trás:** O erro da camada de saída é propagado camada a camada, da saída até a entrada, para atualizar todos os pesos intermediários.
- **Atualização dos pesos:** Usando o gradiente descendente, os pesos são ajustados na direção que diminui o erro, melhorando a performance da rede.

Uma analogia para ajudar fixar: Imagine que você está em uma montanha (função de custo) e quer chegar ao ponto mais baixo (erro mínimo). O gradiente é como o mapa da inclinação do terreno: indica em que direção você deve caminhar para descer. A retropropagação é o processo de calcular essa inclinação em cada passo, camada por camada [9].

1.5.3 Fluxo do Gradiente

Para redes neurais profundas, o cálculo dos gradientes não é trivial, pois depende da composição de várias funções, cada camada aplica uma transformação diferente.

O segredo para calcular o gradiente corretamente está na **regra da cadeia** do cálculo diferencial, que permite decompor a derivada da função de custo em produtos de derivadas das funções de ativação e das transformações lineares de cada camada.

No entanto, durante esse processo, podem ocorrer dois problemas clássicos:

- **Gradientes que desaparecem:** O valor do gradiente fica muito pequeno à medida que volta para as primeiras camadas, impedindo que essas camadas aprendam adequadamente [2].
- **Gradientes explosivos:** O valor do gradiente cresce demais, causando instabilidade e dificultando a convergência do treinamento [3].

Fique Alerta!

Para mitigar esses problemas, são utilizadas funções de ativação adequadas, como a ReLU (que ajuda a evitar gradientes muito pequenos) e técnicas como normalização de dados e inicialização cuidadosa dos pesos.

Resumo visual do fluxo do gradiente:

$$\underbrace{\frac{\partial J}{\partial W^{(L)}}, \frac{\partial J}{\partial b^{(L)}}}_{\text{gradientes da última camada}} \rightarrow \cdots \rightarrow \underbrace{\frac{\partial J}{\partial W^{(1)}}, \frac{\partial J}{\partial b^{(1)}}}_{\text{gradientes da primeira camada}}$$

Onde $W^{(l)}$ e $b^{(l)}$ são os pesos e vieses da camada l.

1.5.4 Cálculo do Gradiente com Entropia Cruzada

Vamos simular uma rede simples com uma única camada de saída com ativação sigmoide. Dado um exemplo de entrada, queremos calcular o valor da função de custo e o gradiente da função de custo em relação à saída da rede.

Seja:

$$y = \text{rótulo real} \in \{0, 1\}$$

$$\hat{y} = \text{saída prevista pela rede (probabilidade)}$$

A função de custo por exemplo é:

$$J = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

O gradiente da função de custo em relação à saída $\hat{y}$ é:

$$\frac{\partial J}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}$$

Este valor indica como a saída da rede deve mudar para reduzir o erro.

Sabendo que $\hat{y} = \text{sigmoid}(z)$ e, portanto, $\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$, podemos usar a regra da cadeia para encontrar o gradiente da função de custo em relação à entrada da função de ativação, z:

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \left(-\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}\right) \cdot \hat{y}(1 - \hat{y}) = -y(1 - \hat{y}) + (1 - y)\hat{y} = \hat{y} - y$$

Este resultado simplificado, $\hat{y} - y$, é crucial para a retropropagação e é o que será calculado no código a seguir para obter `dz`.

Copie e Teste!

```python
import numpy as np

# Entradas e pesos
x = np.array([[0.5, 1.2, -0.3]]) # uma amostra com 3 features
w = np.array([[0.4], [-0.6], [0.1]]) # pesos (3x1)
b = np.array([[0.1]]) # viés
```

```python
# Função sigmoide
def sigmoid(z_val):
    return 1 / (1 + np.exp(-z_val))

# Forward pass
z = np.dot(x, w) + b # saida linear
y_pred = sigmoid(z) # saida da rede (probabilidade)

# Supõe-se que o rótulo verdadeiro seja 1
y_true = 1

# Gradiente da função de custo (entropia cruzada) em relação a z (
   para sigmoide na saída):
# dJ/dz = y_pred - y_true
dz = y_pred - y_true

# Gradiente em relação aos pesos e ao viés
dw = np.dot(x.T, dz)
db = dz # (soma de dz para múltiplas amostras no batch)

print("y_pred =", y_pred)
print("dz =", dz)
print("dw =", dw)
print("db =", db)
```

Saída Esperada

```
y_pred = [[0.38936077]]
dz = [[-0.61063923]]
dw = [[-0.30531962]
      [-0.73276708]
      [ 0.18319177]]
db = [[-0.61063923]]
```

A saída prevista pela rede (`y_pred`) foi aproximadamente **0.3894**. Como o valor real (`y_true`) era **1**, o erro ($dz = \texttt{y\_pred} - \texttt{y\_true}$) foi de aproximadamente **-0.6106**. Este valor negativo indica que a previsão da rede está consideravelmente abaixo do valor esperado (1.0), e os pesos (`w`) e o viés (`b`) precisam ser ajustados para aumentar a saída da rede na próxima iteração.

O gradiente `dw` (com valores aproximados `[[-0.30531962], [-0.73276708], [0.18319177]]`) mostra como cada peso deve ser modificado para reduzir esse erro. A ideia é que, durante a atualização (geralmente $w_{\text{novo}} = w_{\text{antigo}} - \alpha \cdot \texttt{dw}$, onde α é a taxa de aprendizagem), os pesos sejam ajustados na direção que aumenta a previsão:

- Para o **primeiro peso**, `dw[0]` é aproximadamente **-0.3053**. Como a entrada `x[0]` (0.5) é positiva, um gradiente negativo resultará em um aumento deste peso, o que contribuirá para aumentar `y_pred`.
- Para o **segundo peso**, `dw[1]` é aproximadamente **-0.7328**. A entrada `x[1]` (1.2) também é positiva. Assim, este peso também será aumentado, o que também visa elevar

`y_pred`. O texto original mencionava que este peso "contribuiu negativamente para o erro e será corrigido na direção positiva"; de fato, seu valor atual (`w[1]` = -0.6) multiplicado pela entrada (`x[1]` = 1.2) dá **-0.72**, puxando a previsão para baixo. O ajuste aumentará `w[1]`.

- Para o **terceiro peso**, `dw[2]` é aproximadamente **0.1832**. A entrada `x[2]` (-0.3) é negativa. Um gradiente positivo para uma entrada negativa resultará em uma diminuição deste peso. Diminuir `w[2]` (por exemplo, de 0.1 para um valor menor) fará com que o produto `x[2]*w[2]` (que é `-0.3 * w[2]`) se torne menos negativo (ou mais positivo), o que também ajuda a aumentar `y_pred`.

Da mesma forma, o gradiente `db` (aproximadamente **-0.6106**) indica que o viés `b` também será aumentado para ajudar a rede a produzir um valor mais próximo de 1.0.

1.6 Treinamento e Implementação com Keras

Aqui, apresentaremos passo a passo como montar e treinar uma rede neural densa utilizando a biblioteca `Keras`, amplamente usada para projetos de *Deep Learning* em Python. Ao final, o leitor será capaz de compreender a estrutura de uma rede, separar dados para treinamento e avaliar o desempenho do modelo com métricas simples.

1.6.1 Montagem de uma rede densa

Redes densas, também chamadas de *fully connected*, são aquelas em que cada neurônio de uma camada está conectado a todos os neurônios da camada seguinte. Utilizaremos a API sequencial da Keras para construir nossa rede de forma simples.

Copie e Teste!

```
from keras.models import Sequential
from keras.layers import Dense

# Criação do modelo sequencial
modelo = Sequential()
modelo.add(Dense(units=8, activation='relu', input_shape=(4,))) #
    Camada oculta
modelo.add(Dense(units=1, activation='sigmoid')) # Camada de saída
```

Este modelo tem 1 camada oculta (8 neurônios, ReLU) e saída binária (Sigmoide).

A função `Dense` define uma **camada totalmente conectada** (ou *densa*) da rede neural. Cada neurônio dessa camada recebe todas as saídas da camada anterior como entrada. O parâmetro `input_shape=(4,)` indica que cada amostra de entrada possui **4 atributos**, ou seja, o vetor de entrada tem dimensão 4.

Na **camada oculta**, usamos a função de ativação **ReLU** (Unidade Linear Retificada), que introduz não linearidade no modelo e ajuda a rede a aprender padrões complexos. Ela transforma valores negativos em zero e mantém valores positivos inalterados. Já na **camada de saída**, aplicamos a função de ativação **Sigmoide**, que converte o resultado em um valor entre 0 e 1. Isso é especialmente útil em tarefas de **classificação binária**, pois permite interpretar a saída como a probabilidade da classe positiva.

1.6.2 Separação de dados

Para o exemplo, utilizaremos o famoso conjunto de dados `Iris` e o adaptaremos para um problema binário que já vimos no **curso de *Machine Learning* do projeto CITHA** (por exemplo, classificar se a flor é da espécie *setosa* ou não). Assim, usaremos o `scikit-learn` para carregar e separar os dados.

Copie e Teste!

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
import numpy as np

# Carregar dados
iris = load_iris()
X = iris.data
y = np.where(iris.target == 0, 1, 0) # 1 se setosa, 0 caso
    contrário

# Dividir em treino e teste
X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
    test_size=0.3, random_state=42)
print(f"Shape X_treino: {X_treino.shape}, Shape y_treino: {
    y_treino.shape}")
print(f"Shape X_teste: {X_teste.shape}, Shape y_teste: {y_teste.
    shape}")
```

Amostras divididas: 70% para treinamento, 30% para teste.

1.6.3 Treinamento do modelo

Agora que temos o modelo e os dados prontos, compilamos a rede definindo a função de perda, o otimizador e a métrica de avaliação. Em seguida, treinamos com o método `fit`.

Copie e Teste!

```
# Compilar o modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])

# Treinar o modelo
historico = modelo.fit(X_treino, y_treino, epochs=50, batch_size
    =5, verbose=1, validation_data=(X_teste, y_teste))
```

Fique Alerta!

A função de perda `binary_crossentropy` é adequada para classificação binária. O otimizador `adam` combina bons resultados com eficiência computacional.

1.6.4 Avaliação e métricas

Após o treinamento, avaliamos o modelo no conjunto de teste. A função `evaluate` retorna a perda e a acurácia. Também podemos exibir previsões e calcular métricas adicionais.

Copie e Teste!

```
# Avaliação no teste
loss, acc = modelo.evaluate(X_teste, y_teste, verbose=0)
print(f"Acurácia no teste: {acc:.2f}")
```

```
# Visualizar histórico (exemplo)
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(historico.history['accuracy'], label='Acurácia (Treino)')
plt.plot(historico.history['val_accuracy'], label='Acurácia (
    Validação)')
plt.title('Evolução da Acurácia')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(historico.history['loss'], label='Perda (Treino)')
plt.plot(historico.history['val_loss'], label='Perda (Validação)')
plt.title('Evolução da Perda')
plt.xlabel('Épocas')
plt.ylabel('Perda')
plt.legend()

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

Saída Esperada

```
Acurácia no teste: 0.58
```

Discussão

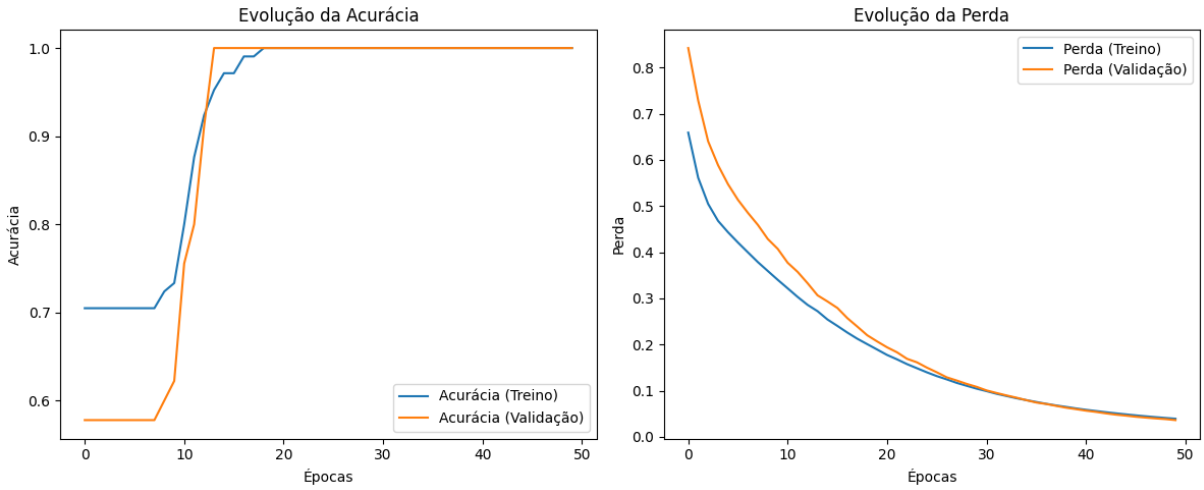

Figura 1.5: Evolução da perda e da acurácia durante o treinamento da rede

O modelo treinado com Keras foi capaz de distinguir eficientemente a classe *setosa* das demais com alta precisão. Apesar da simplicidade, este exemplo introduz os conceitos essenciais para a construção de modelos mais complexos. A escolha da arquitetura, da função de ativação, dos dados e das métricas influencia diretamente o desempenho da rede.

Vendo o resumo da arquitetura

Note que a primeira camada (`Dense(8)`) tem 32 pesos (4 entradas × 8 neurônios) + 8 bias = 40 parâmetros. E a segunda camada tem 8 pesos + 1 bias = 9 parâmetros. Total: 49.

Copie e Teste!

```
modelo.summary()
```

Saída Esperada

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 40
dense_1 (Dense)              (None, 1)                 9
=================================================================
Total params: 149 (600.00 B)
Trainable params: 49 (196.00 B)
Non-trainable params: 0 (0.00 B)
Optimizer params: 100 (404.00 B)
```

1.6.5 Aplicando seus conhecimentos

1. **Modificação da arquitetura:** Altere o número de neurônios da camada oculta para 16. Recompile e treine o modelo. A acurácia final muda? Comente o impacto dessa modificação.

2. **Batch size e epochs:** Execute o treinamento com `epochs=100` e `batch_size=10`. Compare o tempo de execução e os resultados finais com os do exemplo original. O que muda?

3. **Explique com suas palavras:**

   (a) A diferença entre uma camada densa e uma convolucional.

   (b) Por que é importante dividir os dados em conjuntos de treino e teste?

4. **Desafio prático:** Substitua a função de ativação da camada oculta por `tanh` e reexecute o código. Interprete os resultados.

1.7 Regularização e Aprimoramentos

Ao construir redes neurais densas, um dos maiores desafios enfrentados é o equilíbrio entre aprender os padrões reais presentes nos dados e evitar que o modelo memorize detalhes específicos ou ruídos que não representam o problema de forma geral. Esse fenômeno é crítico para que o modelo possa generalizar bem para dados novos e não vistos durante o treinamento. Para atingir esse equilíbrio, aplicamos técnicas de **regularização** e outros aprimoramentos, que atuam como mecanismos que guiam o aprendizado para soluções mais robustas, reduzindo a chance de *overfitting* e melhorando a capacidade de generalização [3, 2].

Além disso, esses aprimoramentos muitas vezes aceleram o processo de treinamento, estabilizam as atualizações dos pesos e tornam os modelos mais resistentes a variações nos dados, o que é essencial para aplicações reais, onde as condições muitas vezes mudam ou os dados são ruidosos e incompletos.

1.7.1 Overfitting e Underfitting

Antes de entrarmos nas técnicas, é fundamental entender os conceitos de **underfitting** e **overfitting**, que são opostos no espectro de desempenho de um modelo.

- **Underfitting** ocorre quando o modelo é simples demais para capturar a complexidade dos dados, ou seja, ele não aprende suficientemente os padrões que descrevem o problema. O resultado é um modelo com baixa performance tanto nos dados de treinamento quanto nos dados novos. Imagine tentar ajustar uma linha reta para dados que claramente seguem uma curva complexa, a linha reta não vai representar bem o comportamento [2, 6].
- **Overfitting** acontece quando o modelo é tão flexível que aprende não apenas os padrões, mas também as peculiaridades e ruídos do conjunto de treinamento. Isso faz com que ele tenha uma performance excelente nos dados usados para treino, porém falhe ao generalizar para novos dados, pois aprendeu detalhes irrelevantes que não se repetem fora da amostra [2, 6].

Fique Alerta!

Um modelo com *overfitting* apresentará alta acurácia no conjunto de treino, mas desempenho fraco no conjunto de teste ou validação. Já o *underfitting* é prejudicial em ambos, não sendo capaz de representar o problema adequadamente.

1.7.2 Dropout: desligando neurônios aleatoriamente

Uma das técnicas mais simples e eficazes para combater o overfitting é o **Dropout**. Essa técnica consiste em desligar aleatoriamente uma fração dos neurônios durante cada iteração do treinamento [3, 2]. Isso impede que o modelo dependa excessivamente de neurônios específicos ou de conexões fixas, promovendo uma espécie de "ensembling interno" onde diferentes subconjuntos da rede são treinados de forma colaborativa.

O efeito prático é que o modelo fica mais robusto, com maior capacidade de generalização, pois precisa aprender representações que funcionem independentemente de neurônios particulares estarem ativos ou não.

```
from keras.layers import Dropout
from keras.models import Sequential
from keras.layers import Dense

modelo = Sequential()
modelo.add(Dense(64, activation='relu', input_shape=(10,)))
modelo.add(Dropout(0.3))  # desativa 30% dos neurônios aleatoriamente
    durante o treino
modelo.add(Dense(1, activation='sigmoid'))
```

Quando o modelo é treinado com `Dropout(0.3)`, durante cada passo do treinamento, 30% dos neurônios da camada oculta são desligados aleatoriamente. Isso força a rede a não depender demais de neurônios específicos, melhorando sua capacidade de generalizar para dados novos e reduzindo o risco de *overfitting*.

1.7.3 Regularização L2 (ou *weight decay*)

Outra estratégia muito usada é a regularização L2, também chamada de *weight decay*. Ela adiciona ao cálculo da função de custo uma penalização proporcional ao quadrado dos pesos da rede. Isso incentiva o modelo a manter os pesos pequenos, evitando que algumas conexões se tornem dominantes e causem *overfitting* [11].

Essa técnica ajuda a simplificar o modelo, controlando sua complexidade, e pode ser combinada com outras técnicas como *Dropout* para resultados ainda melhores.

```
from keras.regularizers import l2
from keras.models import Sequential
from keras.layers import Dense

modelo = Sequential()
modelo.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001))
   )
modelo.add(Dense(1, activation='sigmoid'))
```

Conhecendo um pouco mais!

O parâmetro `l2(0.001)` indica a força da penalização. Quanto maior, maior será a restrição sobre os pesos, levando a modelos mais simples.

Ao usar `kernel_regularizer=l2(0.001)`, adicionamos uma penalidade à função de custo que é proporcional à soma dos quadrados dos pesos. Isso faz com que o modelo prefira pesos menores, evitando que um pequeno grupo de conexões domine a saída da rede, o que ajuda a prevenir *overfitting* e melhora a generalização.

1.7.4 Taxa de aprendizado (*learning rate*)

A **taxa de aprendizado** é um dos hiperparâmetros mais importantes no treinamento de redes neurais. Ela determina o tamanho dos passos dados pelo algoritmo de otimização (como o gradiente descendente) para atualizar os pesos da rede a cada iteração.

- Uma taxa de aprendizado muito alta pode fazer o modelo "pular" o mínimo da função de custo, causando oscilações e até divergência no treinamento.
- Uma taxa muito baixa torna o treinamento muito lento, podendo até travar o aprendizado em mínimos locais subótimos.

Fique Alerta!

Escolher a taxa de aprendizado correta é essencial para o sucesso do treinamento. É comum começar com valores padrão, como 0.001 para o otimizador `Adam`, e ajustar conforme a curva de perda e o comportamento do modelo.

1.7.5 Otimizadores: SGD vs Adam

Os otimizadores são algoritmos responsáveis por atualizar os pesos da rede durante o treinamento, buscando minimizar a função de custo. Entre os mais conhecidos, destacam-se:

- **SGD (Stochastic Gradient Descent):** atualiza os pesos com base em pequenos lotes de dados (mini-batches) (GÉRON, 2023). É um método simples, estável e tradicional, porém pode exigir mais iterações e ajustes finos de hiperparâmetros para convergir bem.
- **Adam:** combina as vantagens do *momentum* com uma taxa de aprendizado adaptativa para cada parâmetro (GÉRON, 2023). Isso torna o Adam mais rápido e eficiente em convergir na maioria dos casos práticos, exigindo menos ajustes manuais.

```
# Otimizador SGD
from keras.optimizers import SGD
modelo.compile(optimizer=SGD(learning_rate=0.01), loss='
    binary_crossentropy')

# Otimizador Adam
from keras.optimizers import Adam
modelo.compile(optimizer=Adam(learning_rate=0.001), loss='
    binary_crossentropy')
```

Conhecendo um pouco mais!

Embora o SGD seja poderoso e fundamentado teoricamente, ele geralmente requer mais cuidado na escolha da taxa de aprendizado e pode demorar mais para convergir. Já o Adam, por ser adaptativo, é muito popular para treinamentos iniciais e protótipos, apresentando desempenho robusto sem necessidade de ajuste fino imediato.

1.7.6 Aplicando seus conhecimentos

Explorando o Dropout: Adicione uma camada `Dropout` entre a camada oculta e a saída de sua rede. Teste com os valores de 0.2, 0.4 e 0.6.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

valor_dropout = 0.4  # altere para 0.2, 0.4 e 0.6

# Dados
iris = load_iris()
X = iris.data
y = np.where(iris.target == 0, 1, 0)

X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
   test_size=0.3, random_state=42)

# Modelo com Dropout
modelo = Sequential()
modelo.add(Dense(16, activation='relu', input_shape=(4,)))
modelo.add(Dropout(valor_dropout))
modelo.add(Dense(1, activation='sigmoid'))

modelo.compile(optimizer='adam', loss='binary_crossentropy',
   metrics=['accuracy'])

historico = modelo.fit(X_treino, y_treino, epochs=25, verbose=1,
   validation_split=0.2)

loss, acc = modelo.evaluate(X_teste, y_teste, verbose=1)

# Plotando a acurácia em treino e validação para observar
   overfitting
plt.plot(historico.history['accuracy'], label='Treino')
plt.plot(historico.history['val_accuracy'], label='Validação')
plt.title(f'Acurácia de {acc:.2f} com Dropout em {valor_dropout}')
plt.xlabel('Épocas')
plt.ylabel('Acurácia (%)')
plt.legend()
plt.grid(True)
plt.show()
```

Saída Esperada

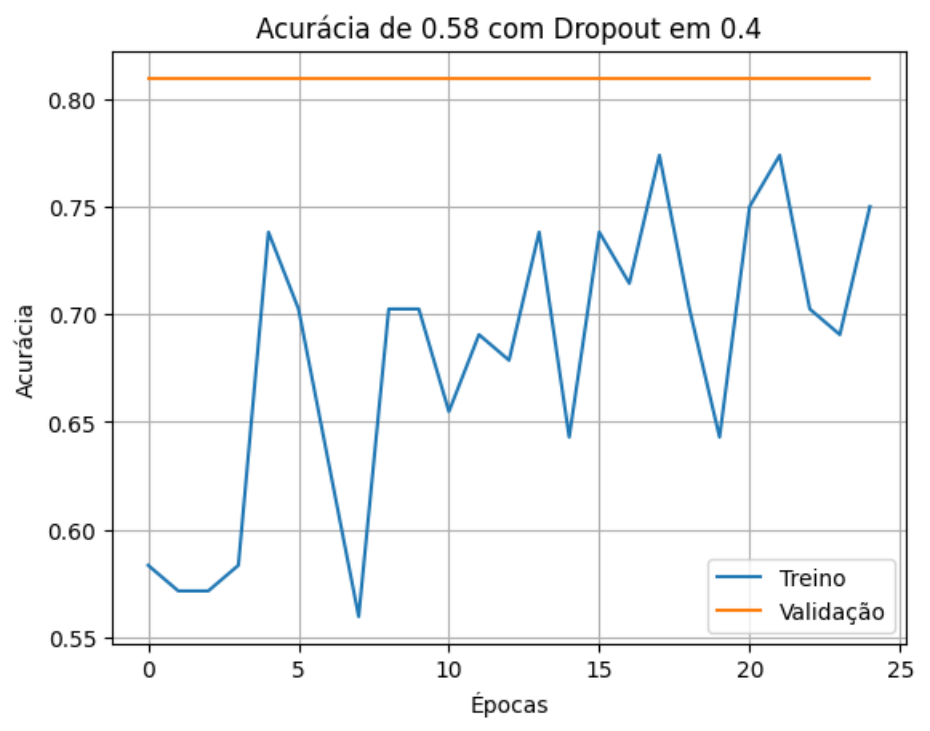

Figura 1.6: Comparação da Acurácia com Dropout em 40%

Agora responda: Como a acurácia no conjunto de teste varia? O comportamento da curva de perda muda com o tempo?

Fique Alerta!

O Dropout atua desligando aleatoriamente uma fração dos neurônios durante o treinamento, o que dificulta que a rede dependa demais de caminhos específicos, isso ajuda a reduzir o overfitting. Com taxas baixas (como 0.2), o modelo ainda aprende de forma eficiente. Com taxas mais altas (como 0.6), o aprendizado pode se tornar instável ou mais lento, e a acurácia no teste pode variar mais. Observe também se as curvas de perda e acurácia ficam mais ”ruidosas”ou demoram mais a estabilizar.

Regularização L2 em ação: Modifique sua camada oculta para aplicar regularização L2 com valores $\lambda = 0.001$, $\lambda = 0.01$ e $\lambda = 0.1$.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l2
```

```
valor_l2 = 0.01  # Altere para 0.001, 0.01 ou 0.1

# Dados
iris = load_iris()
X = iris.data
y = np.where(iris.target == 0, 1, 0)

X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Modelo com regularização L2
modelo = Sequential()
modelo.add(Dense(16, activation='relu', input_shape=(4,),
    kernel_regularizer=l2(valor_l2)))
modelo.add(Dense(1, activation='sigmoid'))

modelo.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])

historico = modelo.fit(X_treino, y_treino, epochs=25, verbose=1,
    validation_split=0.2)

loss, acc = modelo.evaluate(X_teste, y_teste, verbose=0)

plt.plot(historico.history['accuracy'], label='Treino')
plt.plot(historico.history['val_accuracy'], label='Validação')
plt.title(f'Acurácia em {acc:.2f} com L2 = {valor_l2}')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend()
plt.grid(True)
plt.show()
```

Saída Esperada

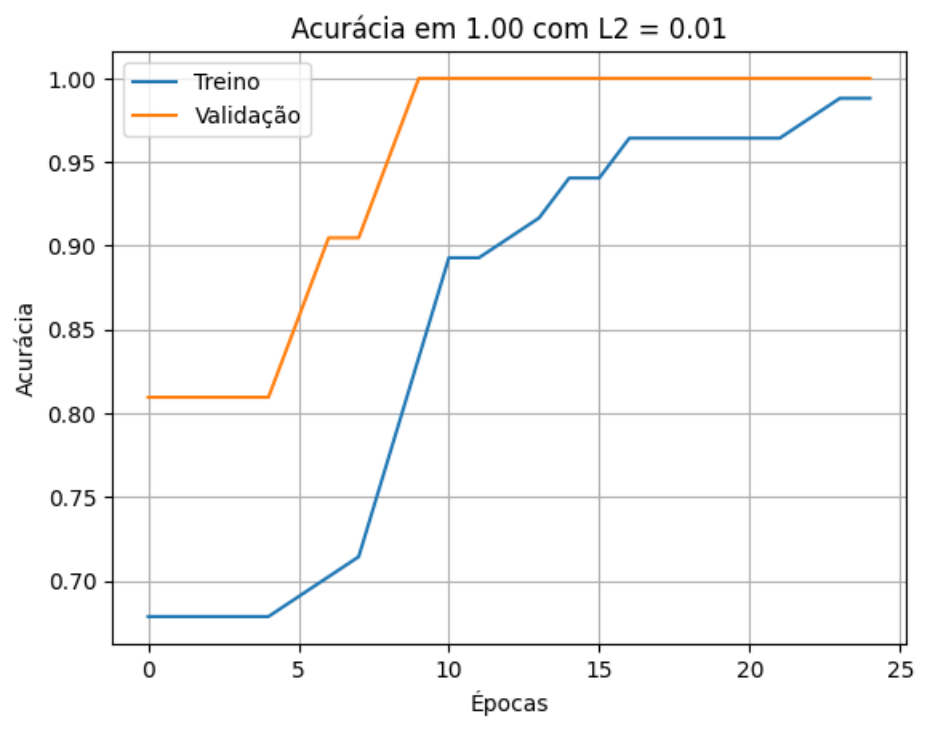

Figura 1.7: Comparação da Acurácia com penalidade L2 = 0.01

Agora responda: O tempo de convergência muda?

Fique Alerta!

A regularização L2 atua como um "freio" nos pesos do modelo, penalizando valores muito altos durante o treinamento. Quanto maior o valor de λ, mais restritivo o modelo se torna, o que pode desacelerar o aprendizado. Isso significa que o tempo de convergência pode sim mudar, especialmente para valores maiores. Observe se a curva de acurácia demora mais para subir ou se estabiliza em um patamar diferente.

Teste de taxa de aprendizado: Treine o modelo com três valores distintos: `0.1`, `0.001` e `0.0001`.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
# Altere para outros valores como 0.1, 0.001 ou 0.0001
taxa_aprendizado = 0.001
```

```
# Dados (classificação binária: setosa ou não)
iris = load_iris()
X = iris.data
y = np.where(iris.target == 0, 1, 0)  # 1 se setosa

X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Modelo com Adam e taxa customizada
modelo = Sequential()
modelo.add(Dense(16, activation='relu', input_shape=(4,)))
modelo.add(Dense(1, activation='sigmoid'))

modelo.compile(optimizer=Adam(learning_rate=taxa_aprendizado),
    loss='binary_crossentropy',metrics=['accuracy'])

# Treinamento com validação para análise
historico = modelo.fit(X_treino, y_treino,epochs=25,verbose=1,
    validation_split=0.2)

# Avaliação no teste
loss, acc = modelo.evaluate(X_teste, y_teste, verbose=1)

# Gráfico da acurácia por época
plt.plot(historico.history['accuracy'], label='Treino')
plt.plot(historico.history['val_accuracy'], label='Validação')
plt.title(f'Acurácia em {acc:.2f} com Learning Rate = {
    taxa_aprendizado}')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend()
plt.grid(True)
plt.show()
```

Saída Esperada

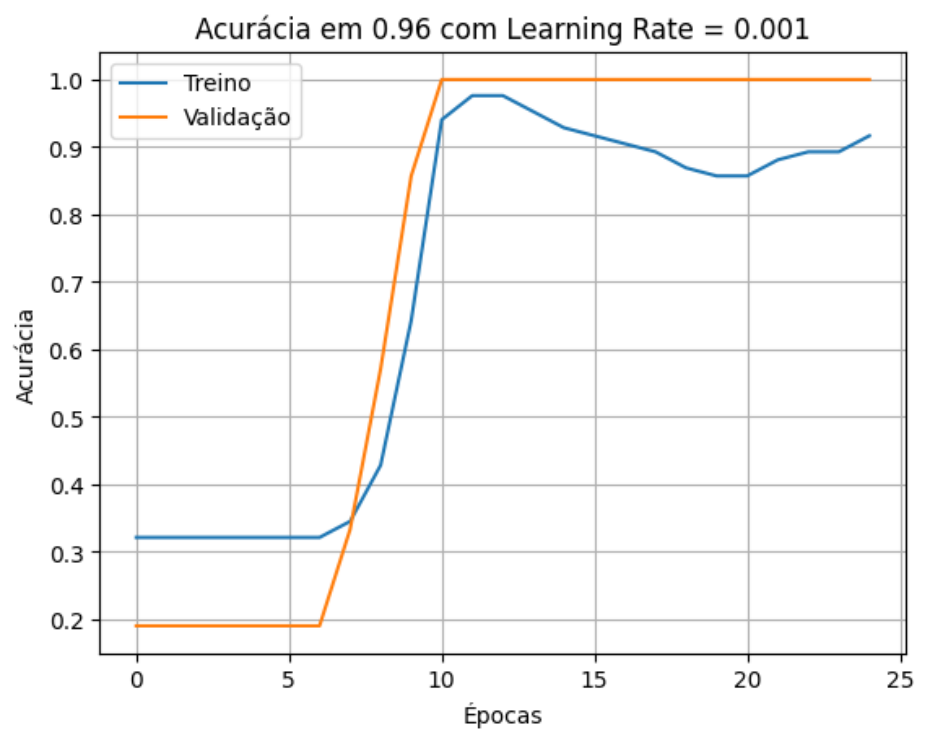

Figura 1.8: Comparação da acurácia com α em 0.001

Agora responda: Qual taxa apresentou melhor curva de perda? Alguma delas impediu o modelo de convergir?

Fique Alerta!

A **taxa de aprendizado** controla o tamanho dos passos dados durante a otimização. Com um valor muito alto (como `0.1`), o modelo pode oscilar e não convergir corretamente. Com um valor muito baixo (como `0.0001`), a convergência será muito lenta, podendo parecer que o modelo "não aprende". Observe as curvas de acurácia e, se possível, a de perda também: a melhor taxa será aquela que apresenta melhora consistente e estável ao longo das épocas, sem grandes oscilações ou estagnação.

SGD vs Adam: Compile e treine sua rede usando os dois otimizadores.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD, Adam
```

```
iris = load_iris()
X = iris.data
y = np.where(iris.target == 0, 1, 0)  # Classificação binária:
   setosa ou não

X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
   test_size=0.3, random_state=42)

def criar_modelo(otimizador):
    modelo = Sequential([Dense(16, activation='relu', input_shape
   =(4,)),Dense(1, activation='sigmoid')])
    modelo.compile(optimizer=otimizador, loss='binary_crossentropy
   ', metrics=['accuracy'])
    return modelo

# Modelo SGD
modelo_sgd = criar_modelo(SGD(learning_rate=0.01))
hist_sgd = modelo_sgd.fit(X_treino, y_treino, epochs=25, verbose
   =1, validation_split=0.2)
loss_sgd, acc_sgd = modelo_sgd.evaluate(X_teste, y_teste, verbose
   =1)

# Modelo Adam
modelo_adam = criar_modelo(Adam(learning_rate=0.001))
hist_adam = modelo_adam.fit(X_treino, y_treino, epochs=25, verbose
   =1, validation_split=0.2)
loss_adam, acc_adam = modelo_adam.evaluate(X_teste, y_teste,
   verbose=1)

# Plotagem do gráfico comparativo
plt.figure(figsize=(10, 7))

plt.plot(hist_sgd.history['val_accuracy'], label='Validação - SGD'
   , color='blue')
plt.plot(hist_adam.history['val_accuracy'], label='Validação -
   Adam', color='green')
plt.plot(hist_sgd.history['accuracy'], '--', label='Treino - SGD',
    color='blue', alpha=0.5)
plt.plot(hist_adam.history['accuracy'], '--', label='Treino - Adam
   ', color='green', alpha=0.5)

plt.title(f'Comparação de Acurácia - SGD ({acc_sgd:.2f}) vs Adam
   ({acc_adam:.2f})')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Saída Esperada

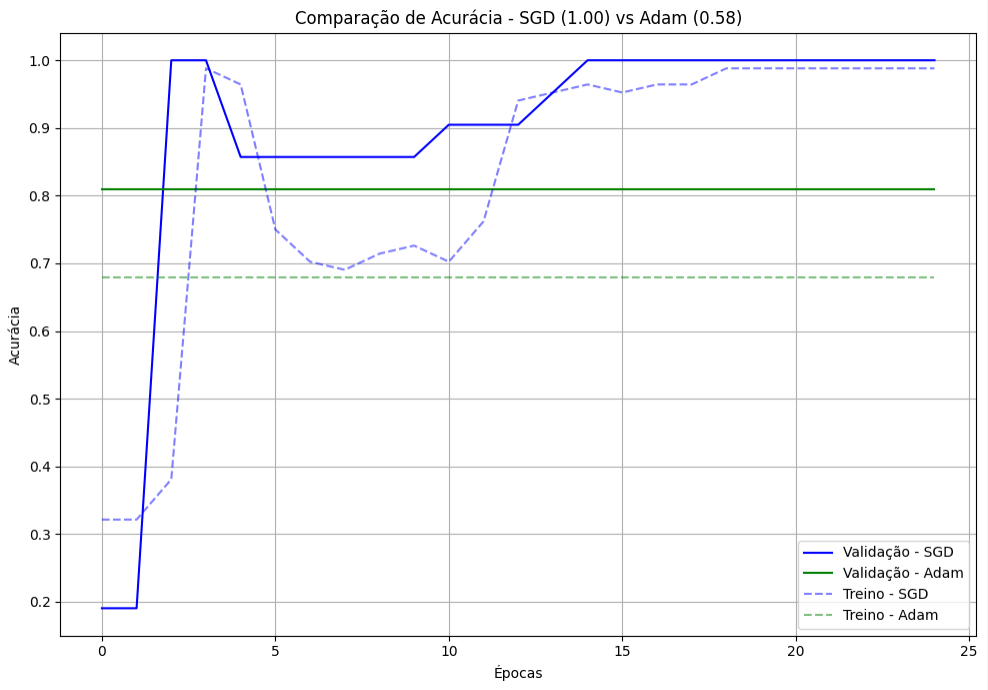

Figura 1.9: Comparação da Acurácia entre SGD e Adam

Agora responda: Qual otimizador levou a uma curva mais estável? Algum deles apresentou oscilações ou desempenho inferior?

Fique Alerta!

Dica para análise: Observe o comportamento das curvas de acurácia ao longo das épocas. O otimizador **Adam** geralmente apresenta uma curva mais suave e ascendente, com convergência mais rápida, pois adapta a taxa de aprendizado durante o treinamento. Já o **SGD** pode gerar curvas mais instáveis, com oscilações maiores, especialmente quando a taxa de aprendizado não está bem ajustada. Compare tanto as curvas de treino quanto de validação para perceber padrões de estabilidade ou flutuações no desempenho.

Diagnóstico de overfitting: Remova qualquer regularização e aumente o número de neurônios da camada oculta para 64.

Copie e Teste!

```
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense

# Construção do modelo: rede densa com 64 neurônios na camada
   oculta (mais complexa) e sem regularização
modelo = Sequential([Dense(64, activation='relu', input_shape=(4,)
   ),Dense(1, activation='sigmoid')])
```

```
# Compilação do modelo com otimizador Adam e função de perda para
    classificação binária
modelo.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])

# Treinamento do modelo com 25 épocas e validação em 20% dos dados
     de treino
historico = modelo.fit(X_treino, y_treino, epochs=25, verbose=1,
    validation_split=0.2)

# Avaliação do modelo no conjunto de treino e teste
acc_treino = modelo.evaluate(X_treino, y_treino, verbose=1)[1]
acc_teste = modelo.evaluate(X_teste, y_teste, verbose=1)[1]

print(f"Acurácia no conjunto de treino: {acc_treino:.2f}")
print(f"Acurácia no conjunto de teste: {acc_teste:.2f}")

# Plot da acurácia de treino e validação
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(historico.history['accuracy'], label='Treino')
plt.plot(historico.history['val_accuracy'], label='Validação')
plt.title('Acurácia durante o treinamento')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend()
plt.grid(True)

# Plot da perda de treino e validação
plt.subplot(1, 2, 2)
plt.plot(historico.history['loss'], label='Perda no Treino')
plt.plot(historico.history['val_loss'], label='Perda na Validação'
    )
plt.title('Perda durante o treinamento')
plt.xlabel('Épocas')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

Saída Esperada

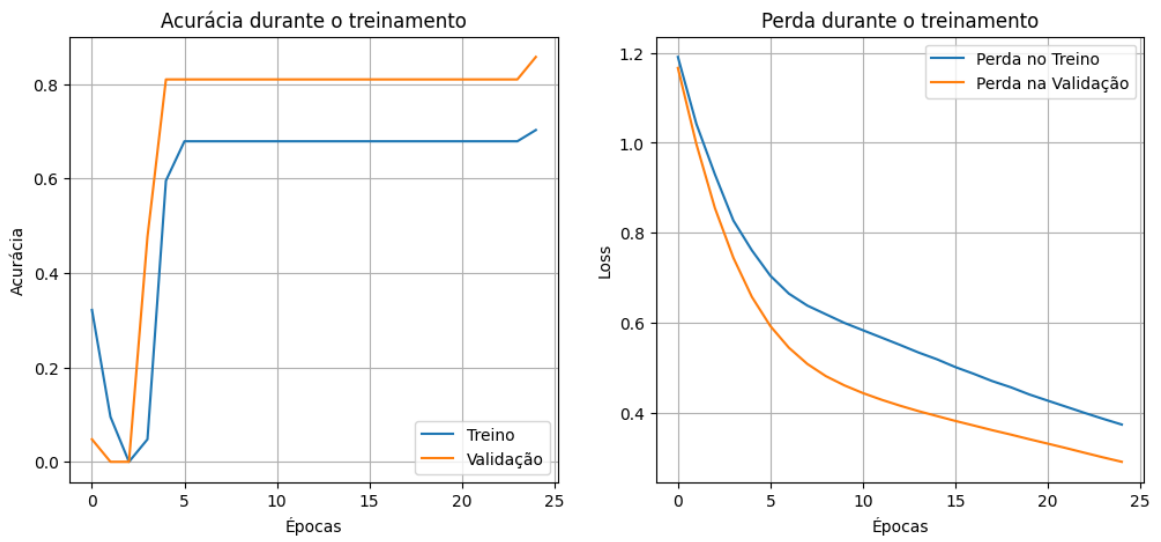

Figura 1.10: Impacto do Aumento de Neurônios sem Regularização na Acurácia de Treino e Teste

Agora responda: O desempenho no treino melhorou? A acurácia no teste piorou? Justifique com base no conceito de *overfitting*.

Fique Alerta!

Ao aumentar o número de neurônios na camada oculta sem aplicar regularização, o modelo tende a melhorar seu desempenho no conjunto de treino, pois pode memorizar melhor os dados. No entanto, essa maior capacidade pode fazer com que o modelo se ajuste demais às características específicas do treino, prejudicando sua capacidade de generalização e, consequentemente, reduzindo a acurácia no conjunto de teste. Esse comportamento é conhecido como **overfitting**, quando o modelo aprende ruídos e detalhes irrelevantes dos dados de treino, comprometendo seu desempenho em dados novos.

Vamos discutir: Com base nos seus testes, escreva um parágrafo comparando as técnicas de regularização estudadas. Qual delas pareceu mais eficiente para seu caso? Em que situações cada uma é mais adequada?

1.7.7 Por que os resultados do treinamento mudam?

Durante o treinamento de redes neurais, os pesos iniciais dos neurônios são definidos com uma semente aleatória (*random seed*). Esse comportamento é **intencional e fundamental** para o bom desempenho dos modelos de aprendizado profundo.

O processo de aprendizado é guiado por um algoritmo de otimização (como o *gradiente descendente*), que tenta encontrar o melhor conjunto de pesos para minimizar a função de custo e como esse processo depende do ponto de partida, isto é, dos valores iniciais dos pesos, diferentes execuções podem seguir caminhos distintos no espaço de soluções. Isso pode resultar em modelos ligeiramente diferentes, com desempenhos variados, mesmo que os dados e o código permaneçam iguais.

Essa aleatoriedade tem uma função estratégica: evita que a rede fique presa em *mínimos locais ruins*, ou seja, soluções subótimas que não representam o melhor possível que o modelo poderia alcançar. Ao iniciar com valores diferentes, o otimizador tem a chance de explorar melhor o espaço de busca e, potencialmente, encontrar soluções mais eficazes.

Portanto, se ao treinar duas vezes o mesmo modelo você notar variações nos valores de perda ou acurácia, saiba que isso é esperado, especialmente em redes pequenas ou com poucos dados.

Capítulo 2

Redes Convolucionais (CNN)

Iniciando o diálogo...

Imagine que você, estudante, faz parte de uma equipe de monitoramento ambiental em sua escola. Um drone foi utilizado para tirar fotos de uma área de floresta próxima à sua comunidade. Seu desafio é analisar essas imagens para identificar locais onde houve queimadas ou desmatamento recente. Mas há dezenas de imagens, e os detalhes nem sempre são fáceis de ver a olho nu. Como ensinar um computador a reconhecer automaticamente esses sinais, como a coloração acinzentada das cinzas ou a ausência de vegetação verde?

2.1 Introdução à Visão Computacional

Este desafio mostra a importância da Visão Computacional, área da inteligência artificial dedicada a permitir que máquinas interpretem imagens e vídeos. As Redes Neurais Convolucionais (CNNs) são ferramentas fundamentais nesse processo, permitindo que algoritmos aprendam automaticamente a reconhecer padrões visuais complexos.

Aliás, você já tentou organizar uma coleção de fotos em seu celular? Como você as categorizaria? Pelo conteúdo visual, pela data, pelo local? Imagine ensinar isso para um computador, quais informações ele precisaria para entender o que está em cada imagem?

As CNNs são amplamente utilizadas para resolver problemas práticos na Amazônia e outras regiões, essas aplicações são exemplos reais e concretos da importância da Visão Computacional, mostrando como o uso das CNNs impacta diretamente a preservação e o estudo do meio ambiente.

- **Detecção e monitoramento de desmatamento:** Imagens de satélite são analisadas continuamente para identificar clareiras novas ou áreas degradadas. CNNs conseguem aprender padrões visuais que indicam desmatamento, mesmo em imagens com diferentes condições de iluminação e clima.
- **Classificação automática de espécies botânicas:** Fotografias de folhas, flores e frutos podem ser usadas para treinar CNNs que identificam automaticamente a espécie da planta, ajudando a catalogar a biodiversidade sem necessidade de um especialista para cada amostra.

- **Reconhecimento de animais em câmeras armadilhadas:** Para estudos da fauna, câmeras instaladas na floresta capturam milhares de imagens que precisam ser classificadas para identificar as espécies presentes. CNNs aceleram esse processo, evitando a análise manual exaustiva.
- **Detecção de focos de incêndio e doenças:** Imagens aéreas e de drones são usadas para detectar sinais visuais de fogo, fumaça ou doenças em árvores, possibilitando ações rápidas de controle e prevenção.
- **Reconhecimento de padrões em rios e hidrografia:** CNNs auxiliam na análise das mudanças no curso dos rios, assoreamento e alterações ambientais, importantes para o planejamento territorial e conservação.

Exercício rápido
Pense em outras possíveis aplicações das CNNs.
Você consegue nos trazer pelo menos três?

2.1.1 Diferença entre imagens e dados tabulares

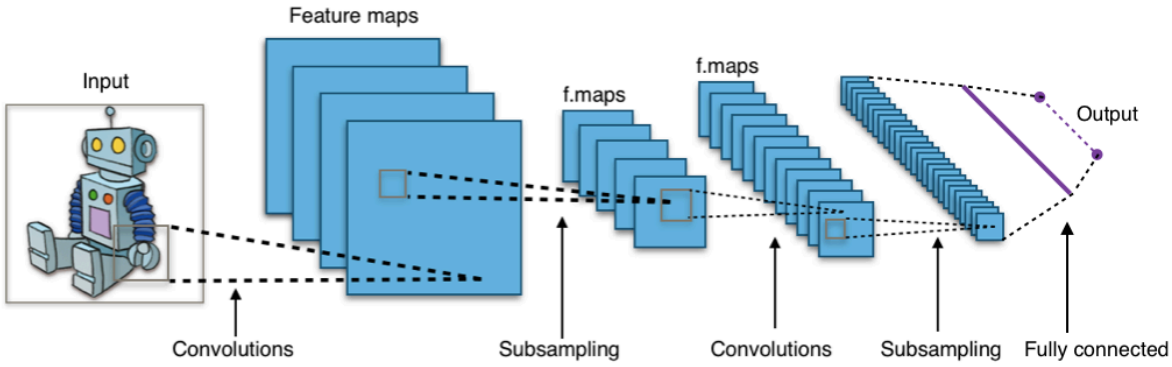

Figura 2.1: Exemplo de uma típica rede neural convolucional totalmente conectada
Fonte: *Wikimedia Commons*, 2025.

Para entender por que as CNNs são especiais para imagens, precisamos compreender as características dos dados visuais em comparação aos dados tabulares tradicionais.

Dados tabulares, a estrutura convencional: Em aprendizado de máquina tradicional, como em regressão ou árvores de decisão, trabalhamos com dados organizados em tabelas. Cada linha representa uma amostra, e cada coluna representa uma variável ou atributo. Esses dados geralmente são armazenados em arquivos `.csv` (Comma-Separated Values), amplamente utilizados por sua simplicidade e compatibilidade com diversas ferramentas. Por exemplo, uma tabela com dados de amostras de solo:

Esses atributos são independentes e não possuem uma ordem espacial entre eles. A coluna ”pH”não está ”próxima”da coluna ”Umidade”, elas são variáveis distintas sem relação posicional.

Amostra	pH	Umidade (%)	Temperatura (°C)
1	5.6	45	26
2	6.1	40	28
3	5.9	50	25

Tabela 2.1: Exemplo de dados tabulares de solo.

Imagens, dados com estrutura espacial: Diferentemente dos dados tabulares, imagens possuem uma estrutura espacial organizada. Em uma imagem colorida (RGB), cada pixel está posicionado em uma grade bidimensional e possui três valores associados, um para cada canal de cor: vermelho (R), verde (G) e azul (B). Essa organização preserva relações de vizinhança, fundamentais para identificar padrões visuais como bordas, texturas e formas.

Uma imagem RGB pode ser representada por um tensor tridimensional:

$$\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$$

onde H é a altura (número de linhas), W a largura (número de colunas), e o número 3 representa os três canais de cor. Cada canal é uma matriz $H \times W$ contendo intensidades de uma das cores primárias.

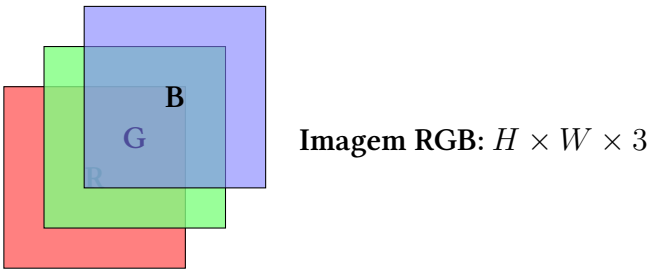

Figura 2.2: Representação esquemática de uma imagem RGB como três camadas: Vermelho (R), Verde (G) e Azul (B).

Exemplo: Para uma imagem de 100×100 pixels, o tensor correspondente tem dimensão $100 \times 100 \times 3$, totalizando 30.000 valores numéricos.

Por que a estrutura espacial importa? Ao contrário dos dados tabulares, a estrutura dos *pixels* em uma imagem define objetos e contextos. Por exemplo, um gato na imagem é reconhecível não só pela cor, mas pela forma e contorno formados pela organização dos *pixels*.

Se embaralhamos os *pixels* aleatoriamente, perderíamos completamente a informação visual, enquanto isso não acontece em dados tabulares, onde a ordem das colunas não importa.

> **Dica para reflexão:** Pense na diferença entre tentar reconhecer um objeto olhando a imagem original e outra onde os *pixels* estão misturados aleatoriamente. Por que essa estrutura espacial faz toda a diferença para o reconhecimento?

2.1.2 Estrutura de dados de imagem

Formalmente, uma imagem digital é um tensor de 2 ou 3 dimensões, dependendo se for escala de cinza ou colorida:

- **Imagem em escala de cinza:**
$$\mathbf{I} \in \mathbb{R}^{H \times W}$$
Cada elemento p_{ij} representa a intensidade do pixel na linha i e coluna j.

- **Imagem colorida RGB:**
$$\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$$
Cada elemento p_{ijk} é a intensidade no pixel (i, j) do canal $k \in \{R, G, B\}$.

Os valores dos *pixels* são normalmente inteiros no intervalo $[0, 255]$ para imagens digitais padrão (8 bits por canal).

Normalização Para modelos de Deep Learning, é prática comum normalizar os *pixels* para o intervalo $[0, 1]$, dividindo por 255:

$$p'_{ijk} = \frac{p_{ijk}}{255}$$

Essa normalização ajuda na estabilidade do treinamento e evita valores muito altos que dificultam a otimização.

2.1.3 Aplicando seus conhecimentos

1. Por que as imagens precisam ser tratadas de forma diferente dos dados tabulares tradicionais?
2. Qual o papel da normalização na preparação de imagens para redes neurais?
3. Que tipos de padrões visuais você acha que uma rede neural pode aprender automaticamente?
4. Carregue uma imagem qualquer (pode ser uma foto do seu ambiente), converta-a para escala de cinza e mostre sua matriz de pixels usando Python. Observe como a dimensão do array mudou.

Dica: use bibliotecas como `PIL` e `numpy` para essa tarefa.

Copie e Teste!

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Certifique-se de ter uma imagem chamada 'imagem.jpg' no mesmo
   diretório
# ou substitua pelo caminho correto da sua imagem.
try:
    img = Image.open('imagem.jpg') # Tente abrir a imagem
    img_array = np.array(img)

    print('Dimensão da imagem original (RGB):', img_array.shape)
```

```
    if len(img_array.shape) == 3: # Verifica se é colorida
        print('Valor do pixel (100, 100) na imagem original:',
   img_array[100, 100])

    plt.figure(figsize=(12,6))
    plt.subplot(1,2,1)
    plt.imshow(img_array)
    plt.title('Imagem Original (RGB)')
    plt.axis('off')

    # Converter para escala de cinza
    img_cinza = img.convert('L')
    img_cinza_array = np.array(img_cinza)
    print('\nDimensão da imagem em escala de cinza:',
   img_cinza_array.shape)
    print('Valor do pixel (100, 100) na imagem cinza:',
   img_cinza_array[100, 100])

    plt.subplot(1,2,2)
    plt.imshow(img_cinza_array, cmap='gray')
    plt.title('Imagem em Escala de Cinza')
    plt.axis('off')

plt.show()

except FileNotFoundError:
    print("Erro: Arquivo 'imagem.jpg' não encontrado. Por favor,
   adicione uma imagem com este nome ou ajuste o caminho.")
except IndexError:
    print("Erro de índice: A imagem pode ser menor que 100x100
   pixels. Tente um pixel dentro das dimensões da imagem.")
```

Saída Esperada

```
Dimensão da imagem original (RGB): (373, 515, 3)
Valor do pixel (100, 100) na imagem original: [45 50 53]

Dimensão da imagem em escala de cinza: (373, 515)
Valor do pixel (100, 100) na imagem cinza: 49
```

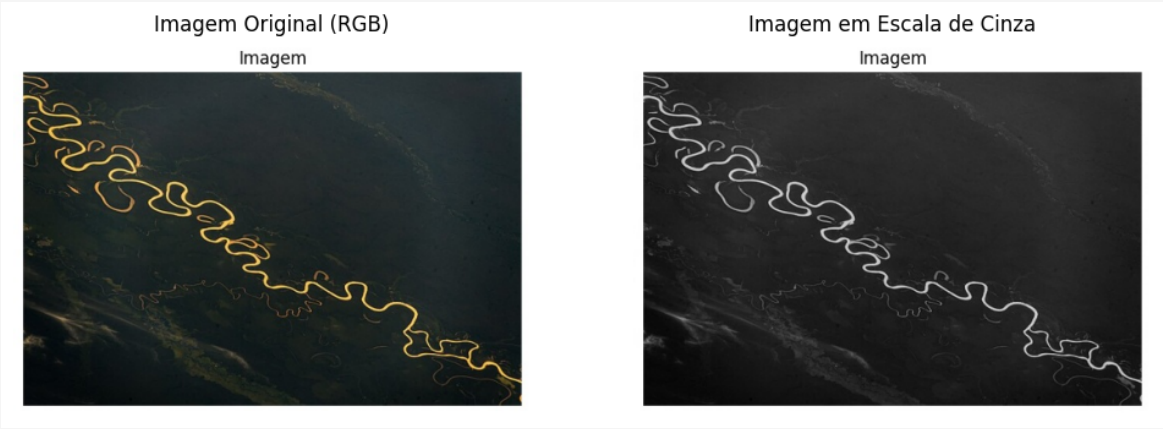

Figura 2.3: Imagem do Rio Juruá sendo exibido atráves do código python acima
Fonte: *Wikimedia Commons*, 2025.

Fique Alerta!

Esse código ajuda a entender que uma imagem é, em essência, um *array* (ou matriz) multidimensional com informações de pixel organizadas espacialmente.

Se, ao verificar a dimensão da imagem original, você encontrar um resultado como (373, 515, 4), isso indica a presença de um quarto canal.

Em arquivos como .png ou formatos semelhantes, esse quarto canal é geralmente o **canal Alpha (A)**, que complementa os canais RGB (Vermelho, Verde e Azul). Nesse caso, a imagem é do tipo RGBA. O canal Alpha é responsável por definir o **nível de opacidade** de cada pixel, controlando o quão transparente ou opaco ele é.

2.2 Operações de Convolução e Filtros

As Redes Neurais Convolucionais, ou CNNs, são muito usadas em reconhecimento de imagens, como por exemplo para identificar rostos, placas de trânsito ou até animais em fotos. Um dos segredos dessas redes está em uma operação chamada **convolução**, que consiste em aplicar um pequeno bloco de números chamado **filtro** ou **kernel** sobre uma imagem.

Essa operação ajuda a "enxergar" detalhes importantes da imagem, como as bordas de um objeto, mudanças de tonalidade, repetições de padrões ou texturas. É como passar uma lupa especial por cima da imagem para destacar certas partes.

2.2.1 Conceito de Kernel

Podemos imaginar um **kernel** como um pequeno quadrado de números (por exemplo, uma matriz 3×3) que percorre a imagem linha por linha, coluna por coluna. Em cada posição, ele multiplica os números da imagem pelos seus próprios valores e soma tudo. O resultado é um novo valor que será colocado na imagem de saída.

Esse processo permite destacar áreas da imagem onde acontecem certas variações. Dependendo dos números escolhidos no *kernel*, podemos ressaltar contornos, detectar formas ou até borrar a imagem.

Conhecendo um pouco mais!

Kernels diferentes revelam padrões diferentes. Alguns funcionam como "detetores" de bordas horizontais, outros de bordas verticais, e há aqueles que suavizam ou realçam regiões específicas.

2.2.2 Convolução manual

Antes de usar redes neurais complexas, é importante entender o que a convolução faz de forma prática. A seguir, mostramos um exemplo com uma imagem simples e artificial, representada por números 0 e 1. Vamos aplicar um filtro que detecta bordas horizontais e observar o resultado visual.

Copie e Teste!

```
#Importe as dependências a seguir
import numpy as np
from scipy.signal import convolve2d
import matplotlib.pyplot as plt

# Imagem artificial (6x6)
imagem_artificial = np.array([
    [0, 0, 1, 1, 0, 0],
    [0, 1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 1, 0],
    [0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0]
])
```

```
# Filtro (detecta borda horizontal)
kernel_borda_horizontal = np.array([
    [-1, -1, -1],
    [ 0,  0,  0],
    [ 1,  1,  1]
])

# Aplicar convolução 2D
resultado_conv = convolve2d(imagem_artificial,
   kernel_borda_horizontal, mode='valid')

# Visualizar resultado
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(imagem_artificial, cmap='gray')
plt.title('Imagem Original Artificial')

plt.subplot(1, 2, 2)
plt.imshow(resultado_conv, cmap='gray')
plt.title('Resultado da Convolução (Borda Horizontal)')
plt.colorbar() # Para ver os valores
plt.show()
```

Saída Esperada

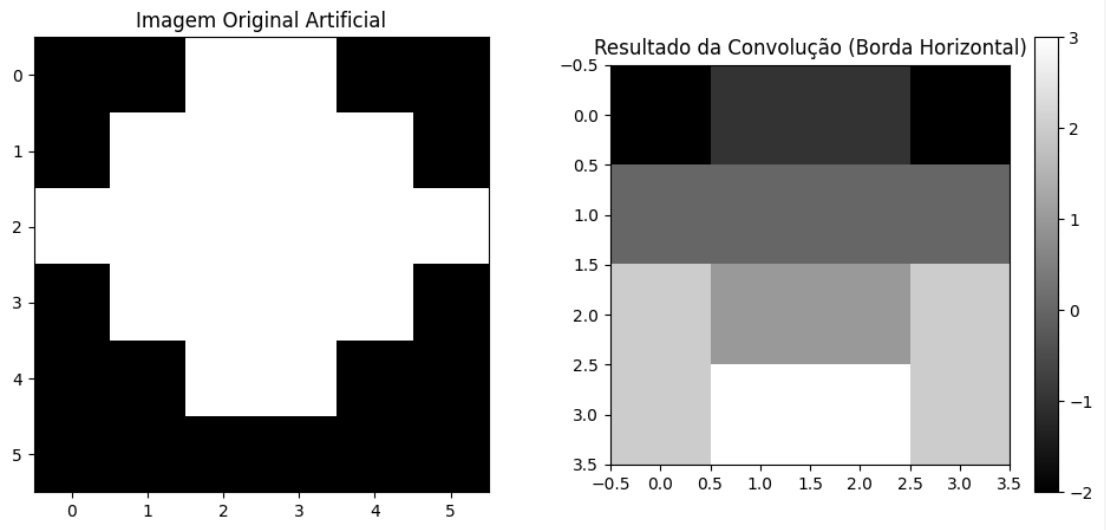

Figura 2.4: Convolução com *kernel* de borda horizontal sobre imagem artificial

Nesse exemplo, a imagem representa algo como uma forma oval em branco sobre fundo preto. Ao aplicar o filtro de borda horizontal, o resultado mostra onde há mudanças de claro para escuro no sentido vertical, destacando as “linhas” que formam o objeto.

2.2.3 Interpretação de bordas e padrões

Quando aplicamos um *kernel* a uma imagem, é como se estivéssemos "fazendo perguntas" à imagem, procurando por partes que combinem com o padrão do filtro. Quanto maior o número gerado, mais forte é a presença desse padrão na imagem original.

Por isso, os mapas de convolução (as imagens de saída da operação) mostram regiões claras onde o padrão foi detectado e regiões escuras onde não há correspondência.

Na prática, esses filtros podem detectar:

- **Contornos** de objetos, como as bordas de uma folha ou de uma letra.
- **Mudanças abruptas** de cor ou textura, como a linha do horizonte ou o contorno de uma estrada.
- **Regiões salientes**, ou seja, partes da imagem com contraste alto que se destacam do fundo.

Fique Alerta!

Em uma CNN real, os filtros não são escolhidos por nós. A rede aprende automaticamente quais são os melhores filtros para resolver o problema, como classificar imagens ou reconhecer objetos.

2.2.4 Quadro Comparativo de Filtros Clássicos

Ao aplicar convoluções sobre uma imagem, diferentes filtros (ou kernels) revelam características distintas. Esses filtros operam como "lentes"que permitem à rede convolucional enxergar aspectos específicos do padrão visual, como contornos, variações suaves ou detalhes intensificados. A escolha ou aprendizado desses filtros é o que torna as CNNs tão poderosas em tarefas de visão computacional.

Na prática, alguns filtros são comumente usados para fins de análise visual e manipulação de imagens. Eles podem ser definidos manualmente para fins didáticos e servem como base para entender o que as redes aprendem automaticamente durante o treinamento.

A seguir, apresentamos uma tabela com os filtros clássicos de convolução 3×3, descrevendo suas matrizes, nomes e efeitos esperados. Esses exemplos ajudam a ilustrar os tipos de padrões que podem ser destacados e como diferentes kernels respondem a variações de intensidade nos pixels vizinhos.

Fique Alerta!

A maioria dos filtros utilizados nas CNNs modernas são aprendidos automaticamente. No entanto, estudar filtros clássicos permite compreender o funcionamento interno e interpretar visualmente o que as redes estão destacando.

Filtro	Kernel (3×3)	Efeito visual
Borda horizontal	$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$	Destaca bordas horizontais, ou seja, regiões da imagem onde há uma transição significativa de claro para escuro na direção vertical. Muito útil para detectar limites superiores ou inferiores de objetos.
Borda vertical	$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$	Evidencia bordas verticais, realçando transições laterais. Frequentemente aplicado na detecção de colunas, bordas de edifícios ou contornos em texto e sinalizações verticais.
Nitidez (Sharpen)	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	Acentua detalhes da imagem ao enfatizar as diferenças entre um pixel e seus vizinhos. Aumenta o contraste local e deixa os contornos mais definidos. Ideal para melhorar a percepção de detalhes.
Desfoque (Blur)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	Promove suavização da imagem ao fazer a média dos pixels vizinhos. Utilizado para reduzir ruído, criar efeitos de fundo ou preparar a imagem para outras operações como segmentação.
Realce de borda (Laplaciano)	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	Destaca regiões de variação abrupta em todas as direções. É um filtro isotrópico, ou seja, não depende de direção específica. Muito útil em tarefas de detecção de contornos gerais.

Tabela 2.2: Filtros clássicos utilizados em convoluções 2D

Esses filtros são amplamente utilizados em processamento de imagem tradicional, mas também representam a base conceitual sobre a qual as CNNs operam. Em redes convolucionais treinadas, os filtros são aprendidos automaticamente para se ajustarem aos padrões relevantes das imagens de entrada. Isso permite, por exemplo, que uma rede aprenda a detectar o contorno de folhas, texturas de pele, formas geométricas ou qualquer outro padrão visual característico dos dados fornecidos.

Compreender como cada tipo de kernel afeta a imagem ajuda o estudante a interpretar as primeiras camadas de uma CNN, onde geralmente predominam filtros parecidos com os mostrados acima. Isso favorece o desenvolvimento de intuições sobre visualizações de ativação, explicabilidade e diagnóstico de modelos.

Caso Prático

Utilize o código apresentado anteriormente em 2.2.2 Convolução Manual como base para realizar novos experimentos. Substitua o *kernel* original por outros filtros clássicos apresentados na Tabela contida no item 2.2.4. Teste, por exemplo:

- **Filtro de borda vertical**: realça contornos laterais;
- **Filtro Laplaciano**: destaca transições abruptas em todas as direções;
- **Filtro de nitidez**: acentua detalhes e contornos finos.

Compare os resultados visuais com a imagem original. Qual filtro melhor evidencia as estruturas presentes? Há diferenças na detecção de bordas horizontais, verticais ou diagonais? Experimente e analise os efeitos gerados por cada matriz de convolução.

2.3 Padding, Stride e Pooling

Nas redes convolucionais, além dos filtros que percorrem a imagem para detectar padrões, existem parâmetros fundamentais que controlam como essas operações são aplicadas: o **padding** e o **stride** (CHOLLET, 2021). Esses dois elementos influenciam diretamente no tamanho da imagem processada e na forma como as informações são mantidas ou descartadas.

Outro componente essencial é o **pooling**, uma etapa que reduz o tamanho das imagens intermediárias, resumindo as regiões mais importantes. Essas técnicas combinadas ajudam as CNNs a funcionarem de maneira mais eficiente, veloz e, ao mesmo tempo, mais robusta frente às variações dos dados de entrada.

2.3.1 Redução de dimensão

Durante o processo de convolução, os filtros ”varrem”a imagem, e em cada passo geram um valor que forma a nova imagem de saída (chamada de mapa de ativação). Entretanto, ao aplicar essa operação sem ajustes, a nova matriz tende a ser menor do que a original, especialmente nas bordas. Isso acontece porque o filtro só pode ser aplicado totalmente onde ele “cabe” dentro da imagem.

Essa redução de tamanho é útil, pois permite que a rede trabalhe com menos dados, economizando memória e processamento. No entanto, é preciso cuidado: se a imagem for reduzida demais logo nas primeiras camadas, podemos perder informações valiosas antes mesmo que a rede aprenda com elas.

Padding

Para evitar perdas precoces de informação, utilizamos o **padding**. Essa técnica adiciona uma borda extra ao redor da imagem original, normalmente com valores zero, o que permite que o filtro seja aplicado até nas bordas, sem diminuir tanto o tamanho da imagem resultante.

Visualmente, é como se colocássemos uma moldura preta em volta da imagem para que o filtro tenha mais espaço para se movimentar sem ”cair”fora.

Conhecendo um pouco mais!

Em bibliotecas como o Keras, o parâmetro `padding='same'` adiciona automaticamente a borda necessária para que a saída tenha o mesmo tamanho da entrada. Já `padding='valid'` não adiciona nada, o que pode ser útil em algumas situações, como quando queremos reduzir a imagem propositalmente.

Stride

O **stride** define o número de posições que o filtro avança a cada movimento. Por padrão, o valor é 1, o que significa que o filtro anda “passo a passo”. Se aumentarmos o `stride` para 2, por exemplo, ele “pula” de dois em dois pixels, tanto na horizontal quanto na vertical.

Esse comportamento é muito parecido com uma lupa: com `stride=1`, examinamos cada cantinho da imagem; com `stride=2`, fazemos uma análise mais rápida, mas menos detalhada.

Fique Alerta!

Usar `stride` maior reduz mais rápido o tamanho da imagem e acelera o processamento. Porém, ao pular muitos pixels, a rede pode deixar de perceber pequenos detalhes ou padrões finos.

2.3.2 Detecção de padrões invariantes

Na prática, as imagens podem variar bastante: um objeto pode aparecer em diferentes posições, tamanhos, ou até com pequenas rotações. Um bom modelo de reconhecimento de imagem precisa ser capaz de identificar um mesmo padrão mesmo que ele esteja em um lugar diferente ou levemente deformado.

É aí que entra o **pooling**. Essa etapa "condensa" os dados, extraindo o valor mais relevante de pequenas regiões da imagem. Isso reduz a dimensão e, ao mesmo tempo, ajuda a rede a focar nas partes mais importantes, mesmo que elas mudem de lugar um pouco.

Exemplo: imagine que temos uma imagem 6×6. Se aplicarmos pooling com blocos 2×2 e stride 2, obteremos uma imagem 3×3, ou seja, com um quarto do tamanho original.

Essa compactação torna o aprendizado mais rápido e ajuda a manter os padrões essenciais, mesmo que o objeto se mova ou se transforme ligeiramente.

2.3.3 Max pooling e average pooling

O **pooling** pode ser feito de diferentes maneiras, dependendo do que queremos extrair de cada bloco da imagem:

- **Max pooling:** seleciona o maior valor do bloco. Isso ajuda a preservar os pontos mais intensos da imagem, como bordas bem definidas ou regiões com forte contraste. Funciona como um "destaque" automático do que mais chama atenção (GÉRON, 2023).
- **Average pooling:** faz a média dos valores do bloco. Isso suaviza a imagem, mantendo uma representação mais generalizada e menos sensível a picos individuais. Pode ser útil para preservar um "panorama geral" da imagem (GÉRON, 2023).

Conhecendo um pouco mais!

Em redes modernas, o max pooling é mais comum, pois ele preserva melhor os traços fortes que as redes usam para tomar decisões.

Essas técnicas combinadas, `padding`, `stride` e `pooling`, ajudam a construir redes convolucionais mais poderosas, capazes de reconhecer padrões em diferentes posições, tamanhos e com maior eficiência.

2.3.4 Aplicando seus conhecimentos

Utilize o código em *Python* a seguir para ajudar na solução dos exercícios abaixo:

Copie e Teste!

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D
```

```
# Imagem 8x8x1 (batch_size, height, width, channels)
imagem_exemplo_sp = tf.random.uniform((1, 8, 8, 1))

# Stride e redução de imagem
print("--- Testando Stride ---")
filtro_stride1 = Conv2D(filters=1, kernel_size=3, strides=1,
   padding='valid')
saida_s1 = filtro_stride1(imagem_exemplo_sp)
print(f"Stride 1, Padding 'valid': Tamanho da saída: {saida_s1.
   shape}")

filtro_stride2 = Conv2D(filters=1, kernel_size=3, strides=2,
   padding='valid')
saida_s2 = filtro_stride2(imagem_exemplo_sp)
print(f"Stride 2, Padding 'valid': Tamanho da saída: {saida_s2.
   shape}")

# Comparando padding SAME e VALID
print("\n--- Testando Padding ---")
filtro_padding_valid = Conv2D(filters=1, kernel_size=3, strides=1,
    padding='valid')
saida_pv = filtro_padding_valid(imagem_exemplo_sp)
print(f"Padding 'valid', Stride 1: Tamanho da saída: {saida_pv.
   shape}")

filtro_padding_same = Conv2D(filters=1, kernel_size=3, strides=1,
   padding='same')
saida_ps = filtro_padding_same(imagem_exemplo_sp)
print(f"Padding 'same', Stride 1: Tamanho da saída: {saida_ps.
   shape}")
```

Stride e redução de imagem: Altere o valor do `stride` de 1 para 2 na aplicação do filtro convolucional. Observe a diferença no tamanho da saída.

Compare padding SAME e VALID: Execute a mesma convolução com `padding='same'` e `padding='valid'`. Explique a diferença no tamanho da saída.

Agora utilize este outro código em *Python* para ajudar a solucionar os exercícios abaixo:

Copie e Teste!

```
import tensorflow as tf
from tensorflow.keras.layers import MaxPooling2D, AveragePooling2D
import numpy as np

# Imagem artificial (batch_size, height, width, channels)
imagem_exemplo_pool = tf.constant([[[[1.0], [2.0], [3.0], [4.0]],
   [[5.0], [6.0], [7.0], [8.0]], [[9.0], [10.0], [11.0], [12.0]],
   [[13.0], [14.0], [15.0], [16.0]]]], dtype=tf.float32)
```

```
print("--- Max Pooling vs Average Pooling (2x2, stride 2) ---")
maxpool_layer = MaxPooling2D(pool_size=(2, 2), strides=2)
maxpool_output = maxpool_layer(imagem_exemplo_pool)
print("Max pooling (2x2, stride 2):\n", tf.squeeze(maxpool_output)
    .numpy())

avgpool_layer = AveragePooling2D(pool_size=(2, 2), strides=2)
avgpool_output = avgpool_layer(imagem_exemplo_pool)
print("\nAverage pooling (2x2, stride 2):\n", tf.squeeze(
    avgpool_output).numpy())

print("\n--- Max Pooling com Stride 1 ---")
maxpool_stride1_layer = MaxPooling2D(pool_size=(2,2), strides=1)
maxpool_stride1_output = maxpool_stride1_layer(imagem_exemplo_pool
    )
print("Max pooling (2x2, stride 1):\n", tf.squeeze(
    maxpool_stride1_output).numpy())
print(f"Tamanho da saída com Stride 1: {maxpool_stride1_output.
    shape}")
```

Max pooling × Average pooling: Rode o código e aplique `MaxPooling2D` e `AveragePooling2D` com janela 2×2. Qual operação mantém os valores máximos? Qual suaviza os dados? Compare os resultados.

Explore stride maior em pooling: Modifique o parâmetro `strides` para 1 no MaxPooling2D e observe como o resultado muda. A saída fica maior? O que isso significa?

2.3.5 Caso prático com imagem real

Vamos agora aplicar tudo o que aprendemos em um exemplo com imagem real. Usaremos uma foto da cidade de Lábrea, localizada no interior do estado do Amazonas. O objetivo é simular como uma rede convolucional pode reduzir a resolução de uma imagem com `MaxPooling`, e em seguida tentar recuperar o formato original com reamostragem.

Caso Prático

Carregue uma imagem em tons de cinza e aplique pooling duas vezes para reduzir a resolução. Em seguida, reamostre a imagem para o tamanho original e observe visualmente a perda de detalhes. Esse processo simula como as redes extraem o essencial e descartam ruídos ou detalhes irrelevantes.

Copie e Teste!

```
import requests
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt
import numpy as np
import cv2 # OpenCV para resize
```

```
from skimage.measure import block_reduce # Para MaxPooling manual

# Baixar a imagem aérea de Lábrea
url = 'https://portalamazonia.com/wp-content/uploads/2021/08/
   b2ap3_large_Labrea.jpeg'
cabecalhos = {'User-Agent': 'Mozilla/5.0'} # Simular um navegador

try:
    resposta = requests.get(url, headers=cabecalhos)
    resposta.raise_for_status() # Levanta erro para códigos HTTP
   ruins (4xx ou 5xx)

    imagem_original_pil = Image.open(BytesIO(resposta.content))
    imagem_cinza_pil = imagem_original_pil.convert('L')
    imagem_cinza_reduzida_pil = imagem_cinza_pil.resize((256, 256)
   )
    imagem_cinza_normalizada = np.array(imagem_cinza_reduzida_pil)
    / 255.0

    # Aplicar MaxPooling duas vezes (reduz pela metade em cada
   etapa)
    # block_size=(2,2) e func=np.max simula MaxPooling2D com
   pool_size=(2,2) e strides=(2,2)
    imagem_pool_1x = block_reduce(imagem_cinza_normalizada,
   block_size=(2,2), func=np.max)
    imagem_pool_2x = block_reduce(imagem_pool_1x, block_size=(2,2)
   , func=np.max)

    # Reamostrar para o tamanho original (256x256)
    imagem_upsample = cv2.resize(imagem_pool_2x, (256, 256),
   interpolation=cv2.INTER_NEAREST)

    # Visualização
    fig, eixos = plt.subplots(2, 2, figsize=(10,10))
    eixos[0][0].imshow(imagem_original_pil)
    eixos[0][0].set_title("Imagem Original (Colorida)")
    eixos[0][1].imshow(imagem_cinza_reduzida_pil, cmap='gray')
    eixos[0][1].set_title("Imagem Cinza Redimensionada (256x256)")
    eixos[1][0].imshow(imagem_pool_2x, cmap='gray')
    eixos[1][0].set_title(f"Após 2x MaxPooling (Tamanho: {
   imagem_pool_2x.shape})")
    eixos[1][1].imshow(imagem_upsample, cmap='gray')
    eixos[1][1].set_title("Reamostrada (Upsample para 256x256)")

    for linha_ax in eixos:
        for eixo_ax in linha_ax:
            eixo_ax.axis('off')
    plt.tight_layout()
    plt.show()
```

```
except requests.exceptions.RequestException as e:
    print(f"Erro ao baixar a imagem: {e}")
except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

Saída Esperada

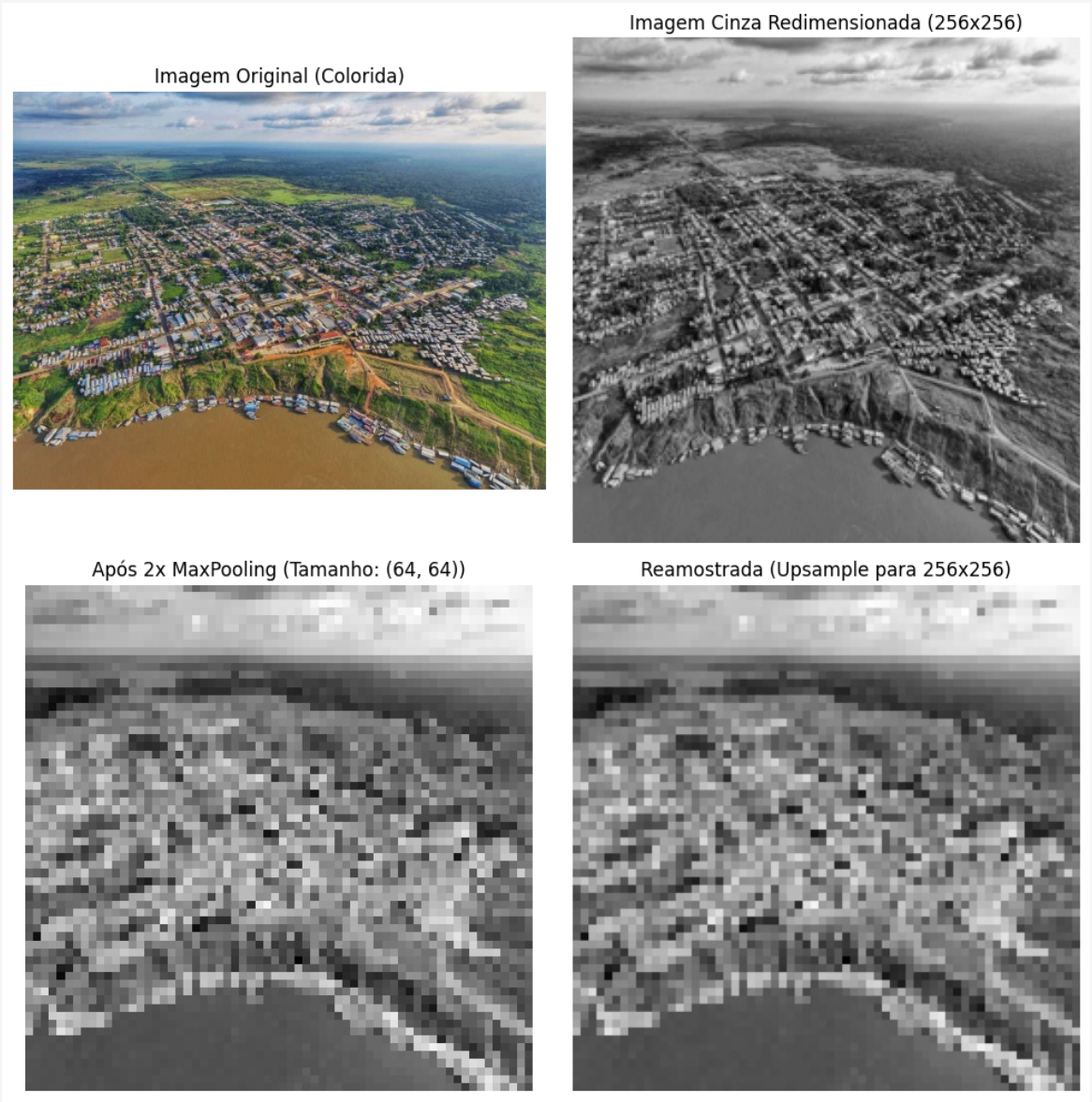

Figura 2.5: Comparação das imagens: original, após pooling e reamostragem - Lábrea-AM
Fonte:*portalamazonia.com*, 2021.

Fique Alerta!

Você percebeu que a imagem após o pooling ainda conserva o formato geral da cidade, mas perdeu muitos detalhes finos como telhados, ruas e vegetação? Esse processo mostra como as redes CNN resumem as informações visuais, mantendo o que é mais importante para a tarefa (como detectar formas gerais), e ignorando o que pode ser ruído.

Conhecendo um pouco mais!

Na prática, redes convolucionais utilizam pooling para acelerar o treinamento, reduzir memória e tornar o modelo mais robusto a pequenas variações de posição e escala.

2.4 Arquiteturas Típicas e Camadas CNN

Agora que já compreendemos o que é uma convolução e como os filtros funcionam, é hora de ver como essas peças se encaixam para formar redes convolucionais completas. Vamos entender como as redes CNNs são organizadas e quais são suas camadas mais comuns.

2.4.1 Camadas fundamentais: Conv2D, Pooling, Flatten e Dense

Podemos imaginar uma rede convolucional como um grande funil que transforma uma imagem cheia de informações em uma única resposta, como dizer se há um gato, um número ou uma árvore na imagem. Para isso, usamos uma combinação de camadas, cada uma com uma função específica:

- **Conv2D (Camada de Convolução):** É como um conjunto de óculos com lentes diferentes. Cada filtro (ou kernel) é uma lente que enxerga uma coisa específica: bordas, linhas, curvas, etc. O Conv2D aplica vários desses filtros e cria uma nova imagem para cada um deles, chamada de mapa de ativação.
- **ReLU (Função de Ativação):** Serve como um ”filtro de energia”. Valores negativos são descartados (zerados), e apenas os positivos passam. Isso ajuda a rede a aprender com mais eficiência e a lidar com padrões não lineares.
- **Pooling (Agrupamento):** Imagine tirar uma foto de longe, você vê menos detalhes, mas ainda reconhece o formato. O Pooling reduz o tamanho das imagens (ou mapas), pegando só o mais importante. O mais usado é o `MaxPooling`, que pega o maior valor de cada região.
- **Flatten:** Transforma a imagem que ainda tem forma de tabela (matriz) em uma linha só de números. Isso é necessário para que a rede possa tomar decisões.
- **Dense (Camada Densa ou Totalmente Conectada):** Aqui é onde as decisões finais são tomadas. Cada neurônio considera todos os dados anteriores e ajuda a rede a dizer, por exemplo: “isso é um número 7” ou “isso é uma folha”.

Conhecendo um pouco mais!

Essas camadas são conectadas como peças de LEGO: primeiro detectamos padrões simples, depois juntamos esses padrões para formar outros maiores, e por fim tomamos uma decisão com base em tudo o que foi visto.

2.4.2 Arquitetura LeNet simplificada

A LeNet foi uma das primeiras redes convolucionais bem-sucedidas. Ela foi criada nos anos 1990 por Yann LeCun para identificar números escritos à mão [3, 2]. Apesar de simples, a LeNet é um excelente exemplo para aprender como uma CNN funciona. Ela possui:

- 2 camadas convolucionais (para extração de padrões)
- 2 camadas de agrupamento (para reduzir o tamanho)
- 3 camadas densas (para tomar decisões)

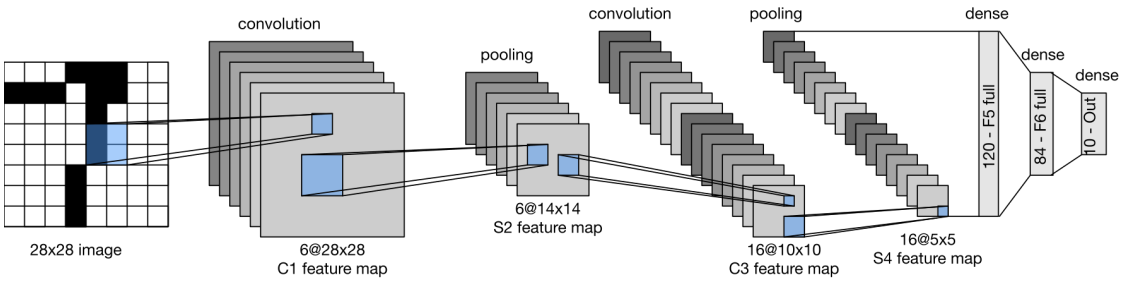

Figura 2.6: Esquema ilustrativo da arquitetura LeNet simplificada
Fonte: *Wikimedia Commons*, 2023.

Fluxo da rede LeNet:

$$\texttt{Imagem } 28 \times 28 \rightarrow \texttt{Conv2D (6 filtros)} \rightarrow \texttt{ReLU} \rightarrow \texttt{Pooling} \rightarrow \texttt{Conv2D (16 filtros)} \rightarrow \texttt{ReLU} \rightarrow \texttt{Pooling} \rightarrow \texttt{Flatten} \rightarrow \texttt{Dense (120)} \rightarrow \texttt{Dense (84)} \rightarrow \texttt{Dense (10)}$$

Cada etapa é como uma estação de análise. No início, a rede detecta pequenas partes da imagem. Depois, combina esses pedaços para identificar formas maiores. Por fim, converte tudo isso em uma decisão.

Fique Alerta!

Mesmo sendo antiga, a LeNet ainda é uma ótima rede para estudar e usar em problemas simples com imagens pequenas, como o conjunto MNIST (números manuscritos).

2.4.3 Construindo sua própria CNN

Caso Prático

Neste exercício, você irá montar uma rede convolucional inspirada na arquitetura LeNet, utilizando a biblioteca `Keras`. O objetivo é reconhecer dígitos escritos à mão com o conjunto de dados `MNIST`. Siga os passos abaixo e registre suas observações. Ao final, você poderá comparar o desempenho de diferentes funções de ativação e entender o papel de cada camada em uma CNN.

Passo 1 – Prepare os dados

1. Carregue o conjunto de dados `MNIST`.
2. Redimensione as imagens para o formato `(28,28,1)`.
3. Normalize os valores dos pixels para ficarem entre 0 e 1.

Copie e Teste!

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

(X_train_mnist, y_train_mnist), (X_test_mnist, y_test_mnist) =
    mnist.load_data()

# Redimensionar e normalizar as imagens
X_train_mnist = X_train_mnist.reshape(-1, 28, 28, 1).astype('
    float32') / 255.0
X_test_mnist = X_test_mnist.reshape(-1, 28, 28, 1).astype('float32
    ') / 255.0

# Converter rótulos para o formato one-hot encoding
y_train_mnist_cat = to_categorical(y_train_mnist, 10)
y_test_mnist_cat = to_categorical(y_test_mnist, 10)

print(f"Shape X_train_mnist: {X_train_mnist.shape}")
print(f"Shape y_train_mnist_cat: {y_train_mnist_cat.shape}")
```

Passo 2 – Criando o modelo

1. Crie uma rede sequencial.
2. E adicione:
   - Duas camadas `Conv2D` com ativação ReLU
   - Duas camadas `MaxPooling2D`
   - Uma camada `Flatten`
   - Três camadas `Dense` com a última usando `softmax`

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense

modelo_lenet = Sequential()
# Camada Convolucional 1 + Pooling
modelo_lenet.add(Conv2D(6, (5,5), activation='relu', input_shape
    =(28,28,1)))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Camada Convolucional 2 + Pooling
modelo_lenet.add(Conv2D(16, (5,5), activation='relu'))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Flatten e Camadas Densas
modelo_lenet.add(Flatten())
modelo_lenet.add(Dense(120, activation='relu'))
```

```
modelo_lenet.add(Dense(84, activation='relu'))
modelo_lenet.add(Dense(10, activation='softmax')) # 10 classes
    para os dígitos 0-9

modelo_lenet.summary()
```

Passo 3 – Treinar e Avaliar:

1. Compile a rede usando `adam` e `categorical_crossentropy`.
2. Treine com 5 épocas e avalie a acurácia.

Copie e Teste!

```
modelo_lenet.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

print("\nIniciando treinamento do modelo LeNet...")
historico_lenet = modelo_lenet.fit(X_train_mnist,
    y_train_mnist_cat, epochs=5, batch_size=128, validation_split
    =0.1, verbose=1)

loss_lenet, acc_lenet = modelo_lenet.evaluate(X_test_mnist,
    y_test_mnist_cat, verbose=0)
print(f"\nAcurácia no teste MNIST: {acc_lenet*100:.2f}%")
```

Saída Esperada

```
Acurácia no teste MNIST: 98.55%
```

** Valor pode variar, mas geralmente é maior que 98%*

Agora responda:

(a) Qual foi a acurácia da sua rede?

(b) O que acontece se você trocar a função `ReLU` por `tanh`? Refaça os testes e compare.

(c) Qual camada transforma o mapa 2D da imagem em vetor 1D? Por que isso é necessário?

Fique Alerta!

Lembre-se de que redes convolucionais aprendem padrões visuais automaticamente. A ordem das camadas e as funções de ativação influenciam muito o desempenho final.

2.4.4 Aplicando seus conhecimentos

Agora que você construiu e treinou uma rede convolucional com sucesso, experimente os desafios a seguir para consolidar sua compreensão. Compare os resultados, registre observações e reflita sobre o que muda. **Modifique o número de filtros:** Utilize como base o exemplo

apresentado no item 2.4.3 (”Construindo sua própria CNN”). Em seguida, altere a primeira camada `Conv2D` para utilizar apenas **16 filtros** e a segunda para **32 filtros**. Essa modificação reduz a complexidade do modelo? Diminui o número total de parâmetros treináveis?

Observe: como essa alteração impacta o tempo de treinamento e a acurácia final do modelo? O desempenho se mantém satisfatório?

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense

modelo_lenet = Sequential()
# Camada Convolucional 1 + Pooling
modelo_lenet.add(Conv2D(16, (5,5), activation='relu', input_shape
   =(28,28,1)))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Camada Convolucional 2 + Pooling
modelo_lenet.add(Conv2D(32, (5,5), activation='relu'))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Flatten e Camadas Densas
modelo_lenet.add(Flatten())
modelo_lenet.add(Dense(120, activation='relu'))
modelo_lenet.add(Dense(84, activation='relu'))
modelo_lenet.add(Dense(10, activation='softmax')) # 10 classes
   para os dígitos 0-9

modelo_lenet.summary()

modelo_lenet.compile(optimizer='adam', loss='
   categorical_crossentropy', metrics=['accuracy'])

print("\nIniciando treinamento do modelo LeNet...")
historico_lenet = modelo_lenet.fit(X_train_mnist,
   y_train_mnist_cat, epochs=5, batch_size=128, validation_split
   =0.1, verbose=1)

loss_lenet, acc_lenet = modelo_lenet.evaluate(X_test_mnist,
   y_test_mnist_cat, verbose=0)
print(f"\nAcurácia no teste MNIST: {acc_lenet*100:.2f}%")
```

Insira uma camada Dropout: Após a camada `Flatten`, insira `Dropout(0.3)` para testar regularização. Essa técnica evita overfitting. A curva de validação melhora?

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense, Dropout
```

```
modelo_lenet = Sequential()
# Camada Convolucional 1 + Pooling
modelo_lenet.add(Conv2D(16, (5,5), activation='relu', input_shape
    =(28,28,1)))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Camada Convolucional 2 + Pooling
modelo_lenet.add(Conv2D(32, (5,5), activation='relu'))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Flatten e Camadas Densas
modelo_lenet.add(Flatten())
modelo_lenet.add(Dropout(0.3)) #Camada de regularização
modelo_lenet.add(Dense(120, activation='relu'))
modelo_lenet.add(Dense(84, activation='relu'))
modelo_lenet.add(Dense(10, activation='softmax')) # 10 classes
    para os dígitos 0-9

modelo_lenet.summary()

modelo_lenet.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

print("\nIniciando treinamento do modelo LeNet...")
historico_lenet = modelo_lenet.fit(X_train_mnist,
    y_train_mnist_cat, epochs=5, batch_size=128, validation_split
    =0.1, verbose=1)

loss_lenet, acc_lenet = modelo_lenet.evaluate(X_test_mnist,
    y_test_mnist_cat, verbose=0)
print(f"\nAcurácia no teste MNIST: {acc_lenet*100:.2f}%")
```

Teste diferentes taxas de aprendizado: Use a função `Adam(learning_rate=...)` com valores como 0.1, 0.001 e 0.0001. Como a taxa afeta a velocidade e a qualidade do aprendizado?

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
     Dense, Dropout
from tensorflow.keras.optimizers import Adam

learning_rate_teste = 0.001 # Teste outros valores como 0.1,
    0.001, 0.0001

modelo_lenet = Sequential()
# Camada Convolucional 1 + Pooling
modelo_lenet.add(Conv2D(16, (5,5), activation='relu', input_shape
    =(28,28,1)))
```

```
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Camada Convolucional 2 + Pooling
modelo_lenet.add(Conv2D(32, (5,5), activation='relu'))
modelo_lenet.add(MaxPooling2D(pool_size=(2,2)))
# Flatten e Camadas Densas
modelo_lenet.add(Flatten())
modelo_lenet.add(Dropout(0.3)) #Camada de regularização
modelo_lenet.add(Dense(120, activation='relu'))
modelo_lenet.add(Dense(84, activation='relu'))
modelo_lenet.add(Dense(10, activation='softmax')) # 10 classes
   para os dígitos 0-9

modelo_lenet.summary()

modelo_lenet.compile(optimizer=Adam(learning_rate=
   learning_rate_teste), loss='categorical_crossentropy', metrics
   =['accuracy'])

print("\nIniciando treinamento do modelo LeNet...")
historico_lenet = modelo_lenet.fit(X_train_mnist,
   y_train_mnist_cat, epochs=5, batch_size=128, validation_split
   =0.1, verbose=1)

loss_lenet, acc_lenet = modelo_lenet.evaluate(X_test_mnist,
   y_test_mnist_cat, verbose=0)
print(f"\nAcurácia no teste MNIST: {acc_lenet*100:.2f}%")
```

Visualize erros de classificação: Após treinar, mostre algumas imagens onde o modelo errou a previsão e analise os padrões.

Copie e Teste!

```
import matplotlib.pyplot as plt

y_pred_probs_lenet = modelo_lenet.predict(X_test_mnist)
y_pred_classes_lenet = np.argmax(y_pred_probs_lenet, axis=1)

# Encontrar os índices dos erros
erros_indices = np.where(y_pred_classes_lenet != y_test_mnist)[0]

# Mostrar alguns erros
plt.figure(figsize=(8, 16))
for i, idx_erro in enumerate(erros_indices[:8]): # Mostra os
   primeiros 8 erros
    plt.subplot(4, 2, i + 1)
    plt.imshow(X_test_mnist[idx_erro].reshape(28, 28), cmap='gray'
   )
    plt.title(f"Previsto: {y_pred_classes_lenet[idx_erro]},
   Correto: {y_test_mnist[idx_erro]}")
```

```
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Fique Alerta!

Essas variações a seguir são comuns em experimentos reais. Ajustar camadas, funções, regularizações e taxas de aprendizado faz parte do trabalho de quem constrói redes neurais.

Saída Esperada

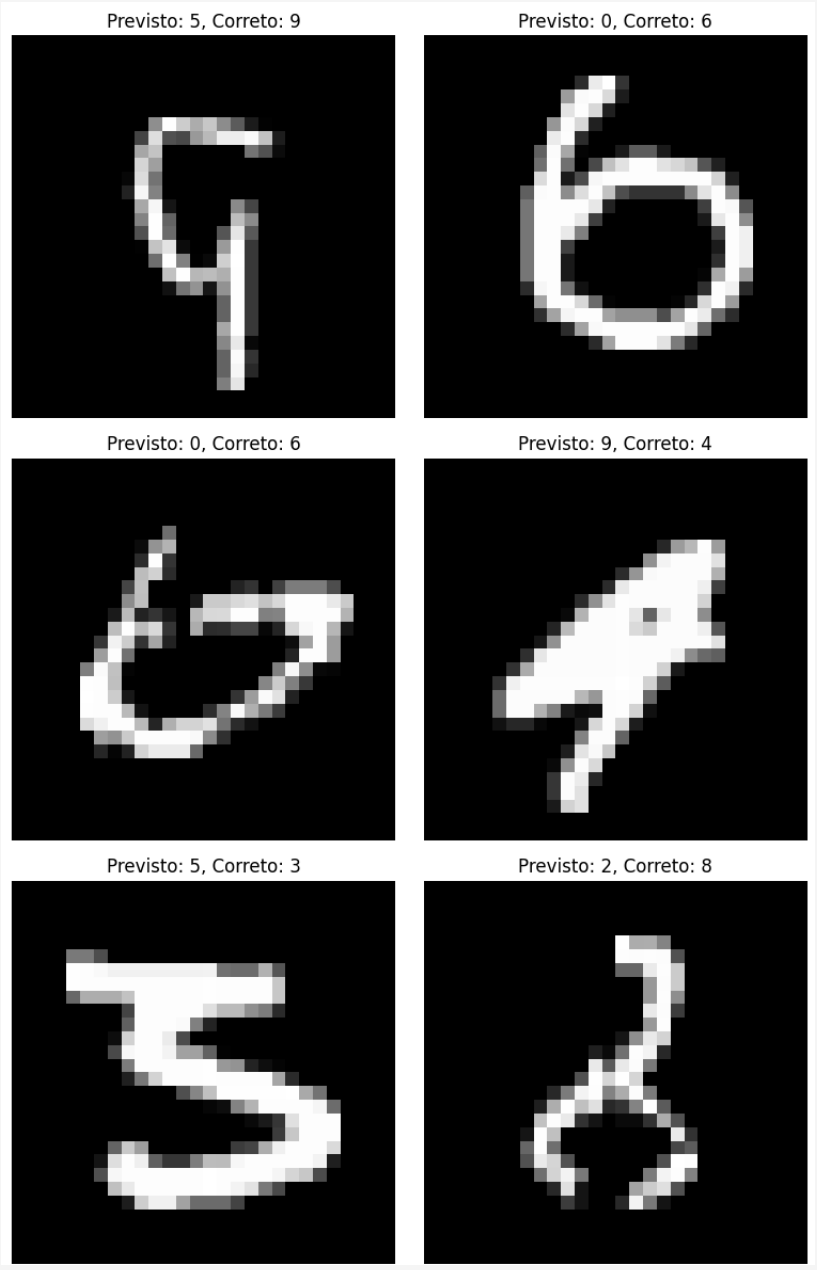

Figura 2.7: Exemplos de Dígitos MNIST Classificados Incorretamente pelo Modelo LeNet

2.5 O que a rede está vendo? Visualização de Filtros e Camadas

As redes convolucionais (CNNs) ganharam destaque devido à sua impressionante capacidade de reconhecer objetos, padrões e até doenças em imagens. Mas surge uma dúvida comum entre iniciantes: **O que exatamente a rede está enxergando?**

Neste item, vamos explorar ferramentas e técnicas que nos permitem "espiar" o interior de uma rede neural convolucional e entender como as informações são processadas em cada camada.

Essa prática, conhecida como **visualização de filtros e camadas**, nos ajuda a entender como uma CNN transforma uma imagem de entrada, como uma folha de planta em uma decisão de "doente" ou "saudável".

2.5.1 O que são mapas de ativação?

Durante o processamento, cada camada convolucional aplica um conjunto de filtros sobre a imagem. Cada filtro tem a missão de detectar padrões específicos, como bordas, curvas ou manchas. O resultado é chamado de **mapa de ativação** (ou *feature map*).

Conhecendo um pouco mais!

Imagine um detector de calor. Ele mostra em vermelho as áreas mais quentes e em azul as frias. De forma semelhante, um mapa de ativação indica as "regiões mais relevantes" da imagem para aquele filtro específico.

Esses mapas são produzidos camada por camada. As primeiras camadas geralmente destacam formas simples, como contornos, enquanto as camadas mais profundas reconhecem padrões mais abstratos, como nervuras de folhas ou formatos de letras.

2.5.2 Como podemos extrair os mapas de ativação?

Para visualizar esses mapas, usamos um recurso do Keras que permite criar um modelo intermediário. Com ele, podemos acessar a saída de uma camada específica ao fornecer uma imagem de entrada, como se estivéssemos "pausando" o cérebro da rede em determinado momento.

2.5.3 Visualizando os filtros

Cada filtro gera um mapa em tons de cinza, que mostra onde ele foi ativado. Por exemplo, ao usar uma imagem do número "5" do MNIST, um filtro pode se ativar mais nas curvas superiores, enquanto outro responde às linhas verticais.

Conhecendo um pouco mais!

Pense nos filtros como diferentes lupas: uma realça as bordas, outra destaca sombras, outra ignora o fundo e foca nos traços centrais. Juntas, elas formam uma representação rica da imagem.

Ao exibir os mapas lado a lado, percebemos como cada filtro "enxerga" aspectos distintos da imagem, mesmo sem qualquer programação explícita.

2.5.4 O que a rede está vendo?

As CNNs processam as imagens de forma hierárquica, em três níveis:

- **Camadas iniciais:** detectam bordas, contrastes e linhas.
- **Camadas intermediárias:** combinam essas formas em estruturas mais complexas.
- **Camadas finais:** extraem os padrões mais significativos e tomam decisões [1, 3].

Essa estrutura reflete a forma como o ser humano enxerga: primeiro percebemos formas básicas e depois reconhecemos o todo.

Fique Alerta!

Se um filtro estiver ativando regiões irrelevantes ou gerando mapas vazios, pode indicar que o modelo está aprendendo padrões errados ou ainda não foi treinado adequadamente.

2.5.5 Caso Prático: Visualizando o MNIST

Para ilustrar na prática, vamos utilizar um exemplo clássico: o conjunto de dados MNIST. Suponha que aplicamos uma imagem do dígito "9" a uma CNN já treinada. Ao observar os mapas de ativação, notamos que:

- Alguns filtros se ativam fortemente em regiões com contornos circulares;
- Outros quase não apresentam ativação;
- A combinação dessas ativações forma uma espécie de "assinatura visual" única para o número.

Caso Prático

Parte 1: Primeiras camadas convolucionais
Escolha uma imagem do MNIST (por exemplo, o dígito 9) e visualize os mapas de ativação das duas primeiras camadas convolucionais. Quais partes do dígito cada filtro destaca? O que muda ao avançar para a segunda camada?
Parte 2: Camadas mais profundas e saída da rede
Agora visualize os mapas de ativação da segunda camada convolucional e também a probabilidade final de classificação da rede para a mesma imagem. Como os padrões evoluem? A rede classificou corretamente? O que podemos aprender sobre o processo de decisão da rede a partir dessas informações?

Copie e Teste!

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
# mnist, to_categorical, Conv2D, MaxPooling2D, Flatten, Dense, np,
```

```
    plt já devem ter sido importados

# 1. Dados (reutilizando X_train_mnist, y_train_mnist_cat,
   X_test_mnist da seção 2.4.3)
if 'X_train_mnist' not in locals(): # Segurança caso não tenha
   rodado a célula anterior
    print("Carregando dados MNIST novamente...")
    (X_train_mnist, y_train_mnist), (X_test_mnist,
   y_test_mnist_orig) = mnist.load_data()
    X_train_mnist = X_train_mnist.reshape(-1, 28, 28, 1).astype('
   float32') / 255.0
    X_test_mnist = X_test_mnist.reshape(-1, 28, 28, 1).astype('
   float32') / 255.0
    y_train_mnist_cat = to_categorical(y_train_mnist, 10)
    y_test_mnist_cat = to_categorical(y_test_mnist_orig, 10)


# 2. Modelo funcional (para facilitar acesso às camadas)
input_img = Input(shape=(28, 28, 1))
x_conv1 = Conv2D(8, (3,3), activation='relu', padding='same', name
   ='conv1')(input_img) # Adicionado padding
x_pool1 = MaxPooling2D((2,2), name='pool1')(x_conv1)
x_conv2 = Conv2D(16, (3,3), activation='relu', padding='same',
   name='conv2')(x_pool1) # Adicionado padding
x_pool2 = MaxPooling2D((2,2), name='pool2')(x_conv2)
x_flat = Flatten(name='flatten')(x_pool2)
output_final = Dense(10, activation='softmax', name='output_dense'
   )(x_flat)

modelo_viz = Model(inputs=input_img, outputs=output_final)
modelo_viz.compile(optimizer='adam', loss='
   categorical_crossentropy', metrics=['accuracy'])
print("\nTreinando modelo para visualização...")
modelo_viz.fit(X_train_mnist, y_train_mnist_cat, epochs=3,
   batch_size=128, verbose=1, validation_split=0.1) # Treino
   rápido

# 3. Escolher imagem de teste (ex: dígito 9, índice pode variar)
# Para encontrar um '9', por exemplo:
# indice_digito_9 = np.where(y_test_mnist_orig == 9)[0]
# if len(indice_digito_9) > 0:
#     indice_img_teste = indice_digito_9[0] # Pega o primeiro '9'
# else:
#     indice_img_teste = 0 # Fallback

indice_img_teste = 9
imagem_teste_viz = X_test_mnist[indice_img_teste].reshape(1, 28,
   28, 1)
label_real_viz = y_test_mnist_orig[indice_img_teste] if '
   y_test_mnist_orig' in locals() else np.argmax(y_test_mnist_cat[
```

```
    indice_img_teste])


print(f"\nVisualizando ativações para a imagem de teste (índice {
    indice_img_teste}, Rótulo Real: {label_real_viz})")

# 4. Modelos para visualizar ativações das camadas Conv2D
modelo_camada1_out = Model(inputs=modelo_viz.inputs, outputs=
    modelo_viz.get_layer('conv1').output)
modelo_camada2_out = Model(inputs=modelo_viz.inputs, outputs=
    modelo_viz.get_layer('conv2').output)

ativacoes1 = modelo_camada1_out.predict(imagem_teste_viz)
ativacoes2 = modelo_camada2_out.predict(imagem_teste_viz)

# 5. Mostrar mapas da 1ª camada convolucional (conv1 - 8 filtros)
plt.figure(figsize=(10, 6.5))
plt.suptitle(f"Ativações da 1ª Camada Convolucional (conv1) para
    Dígito '{label_real_viz}'", fontsize=16)
for i in range(ativacoes1.shape[-1]): # Itera sobre os filtros
    plt.subplot(2, 4, i + 1) # Assumindo 8 filtros
    plt.imshow(ativacoes1[0, :, :, i], cmap='viridis') # 'viridis'
    é uma boa alternativa a 'gray'
    plt.title(f'Filtro {i+1}')
    plt.axis('off')
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()

# 6. Mostrar mapas da 2ª camada convolucional (conv2 - 16 filtros)
plt.figure(figsize=(10.5, 11))
plt.suptitle(f"Ativações da 2ª Camada Convolucional (conv2) para
    Dígito '{label_real_viz}'", fontsize=16)
for i in range(ativacoes2.shape[-1]): # Itera sobre os filtros
    plt.subplot(4, 4, i + 1) # Assumindo 16 filtros
    plt.imshow(ativacoes2[0, :, :, i], cmap='viridis')
    plt.title(f'Filtro {i+1}')
    plt.axis('off')
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()

# 7. Prever a saída final para a imagem e mostrar
saida_final_viz_probs = modelo_viz.predict(imagem_teste_viz)
predito_viz_classe = np.argmax(saida_final_viz_probs[0])

fig_out, ax_out = plt.subplots(1, 2, figsize=(10, 4))
# Imagem de entrada
ax_out[0].imshow(imagem_teste_viz.reshape(28,28), cmap='gray')
ax_out[0].set_title(f"Imagem de Entrada (Real: {label_real_viz})")
ax_out[0].axis('off')
```

```
# Saída final (probabilidades)
barras = ax_out[1].bar(range(10), saida_final_viz_probs[0], color=
    'skyblue')
barras[predito_viz_classe].set_color('salmon')
ax_out[1].set_title(f"Saída Final - Previsto: {predito_viz_classe}
    ")
ax_out[1].set_xlabel("Dígito")
ax_out[1].set_ylabel("Probabilidade")
ax_out[1].set_xticks(range(10))
ax_out[1].grid(axis='y', linestyle='--')
plt.tight_layout()
plt.show()
```

Saída Esperada

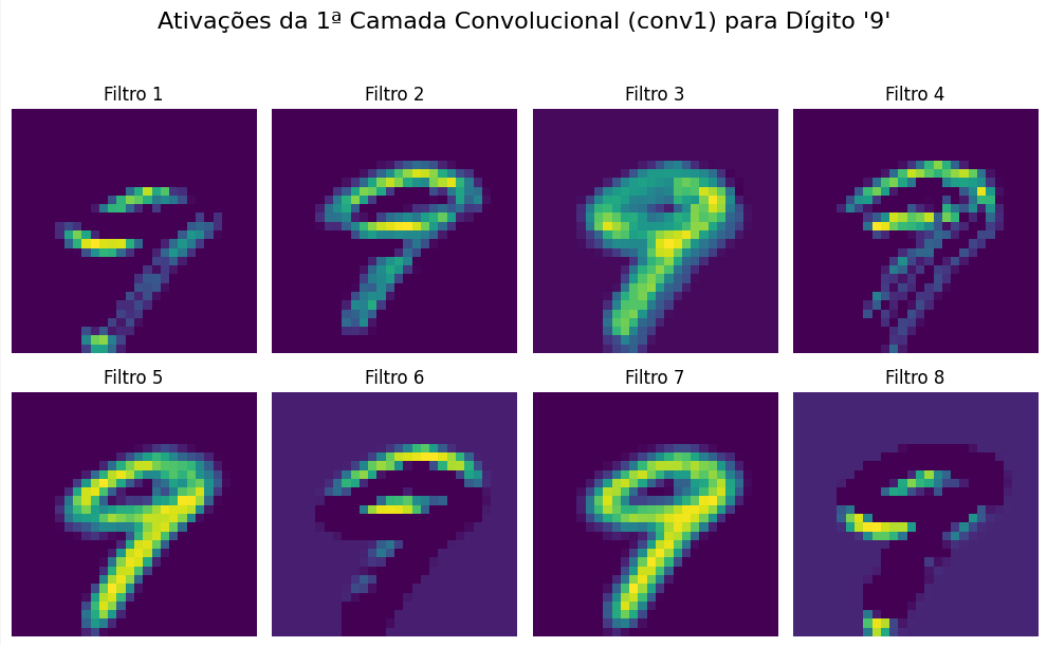

Figura 2.8: Ativações da 1ª camada

Fique Alerta!

As camadas iniciais detectam padrões simples. Camadas posteriores combinam esses padrões e a última camada converte tudo em uma decisão de classificação. Se a rede erra, você pode usar essas visualizações para entender onde o problema começou.

Conhecendo um pouco mais!

Você pode adaptar esse código para visualizar outras imagens e até salvar as ativações em arquivos PNG. Use `indice = ...` para alterar a imagem de entrada.

Saída Esperada

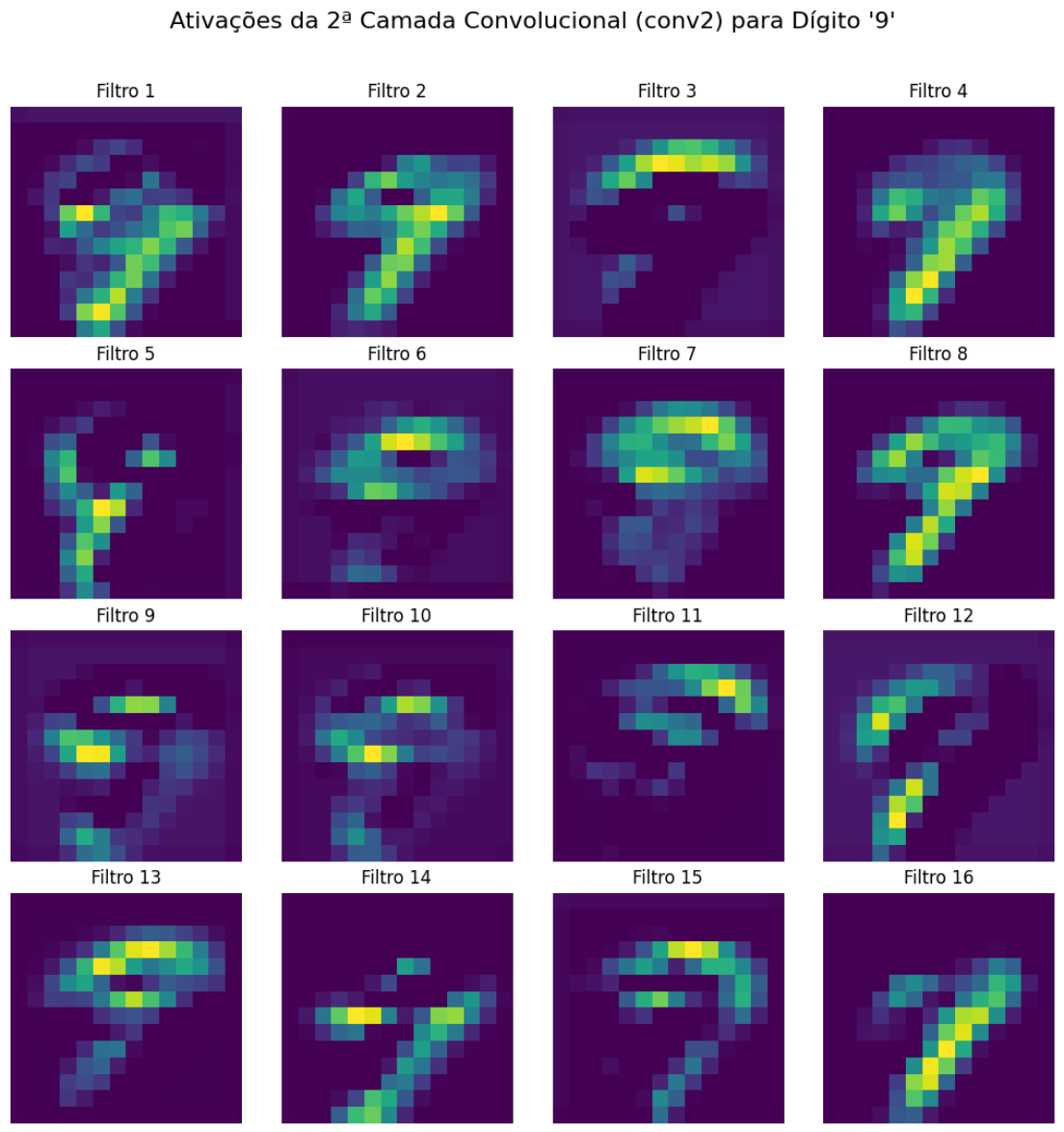

Figura 2.9: Ativações da 2ª camada

Fique Alerta!

Filtros "apagados" podem indicar que eles não estão sendo usados, isso é comum com redes rasas ou com poucos dados. Considere treinar por mais épocas.

Saída Esperada

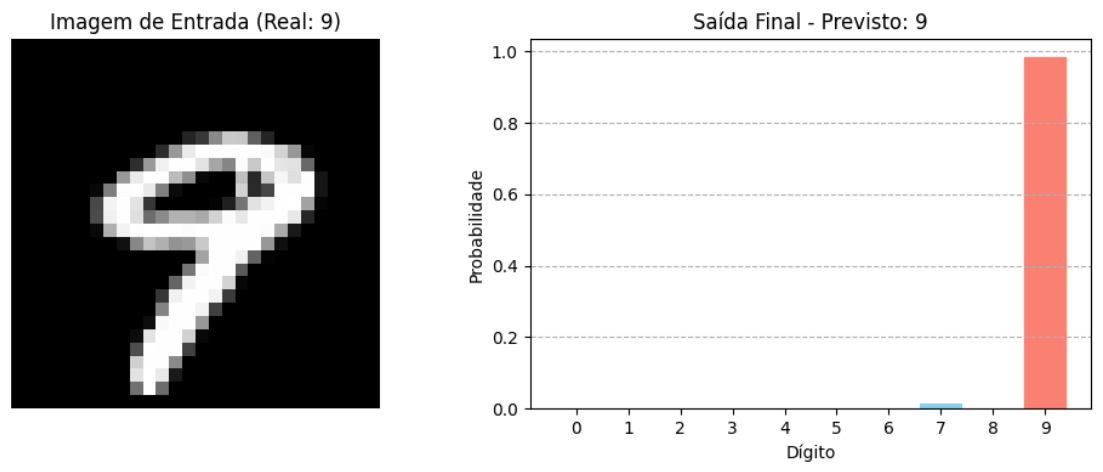

Figura 2.10: Imagem de Entrada e saída final da rede para imagem de teste

2.5.6 O que aprendemos com as ativações?

Ao visualizar as ativações internas de uma rede convolucional, conseguimos entender de forma concreta o que cada camada "vê"e aprende. Isso traz diversos benefícios:

- Permite ajustar a arquitetura da rede (número e tipo de camadas) com mais fundamento;
- Facilita a identificação de gargalos ou excessos na extração de características;
- Ajuda a diagnosticar erros de classificação, revelando o que pode estar confundindo o modelo;
- Torna mais acessível a comunicação dos resultados, especialmente com públicos não técnicos.

Conhecendo um pouco mais!

Ferramentas como `gradCAM`, `keras-vis` e `tf-explain` permitem uma análise ainda mais rica: elas evidenciam visualmente quais regiões da imagem mais influenciaram a decisão final da rede.

Esses recursos reforçam um dos pilares do aprendizado profundo moderno: a interpretabilidade. Em aplicações críticas, como diagnósticos médicos, segurança pública e monitoramento ambiental, entender `como e por que` um modelo toma suas decisões é tão importante quanto alcançar alta acurácia.

Capítulo 3

Redes Recorrentes (RNN)

Iniciando o diálogo...

Você já tentou prever o próximo passo de uma música, entender o sentimento em uma frase, ou antecipar o clima de amanhã? Todos esses exemplos têm algo em comum: são construídos com base em **sequências**. Neste capítulo, vamos explorar como as máquinas também podem aprender com o tempo, lembrando de eventos passados para tomar decisões melhores no presente. Prepare-se para descobrir como as Redes Recorrentes (RNNs) conseguem "pensar" passo a passo, como fazemos quando lemos, ouvimos ou lembramos de algo importante.

3.1 Sequências e o Eixo Temporal

Vivemos em um mundo onde os dados muitas vezes seguem uma ordem. Pense nas palavras que você está lendo agora: elas têm um sentido porque seguem uma sequência. Se embaralhássemos tudo, o texto perderia o significado. O mesmo acontece com muitos outros tipos de informação, e para tratá-los, precisamos de modelos que respeitem essa ordem. As **Redes Neurais Recorrentes (RNN)** foram criadas justamente para isso.

3.1.1 Como assim, aprender com o tempo?

A maioria das redes neurais tradicionais (como as densas ou convolucionais) parte do princípio de que todas as entradas são independentes. Isso funciona bem com imagens ou tabelas, onde a posição de uma célula não muda com o tempo. Mas nem todos os problemas seguem esse padrão.

Imagine as seguintes situações:

- **Previsão do clima:** é impossível prever a temperatura de amanhã sem considerar os dias anteriores.
- **Tradução de textos:** o significado de uma palavra pode mudar completamente dependendo das anteriores.
- **Diagnóstico com sensores:** um único batimento cardíaco diz pouco, é o padrão ao longo do tempo que indica se está tudo bem.

Conhecendo um pouco mais!

Em todos esses casos, o que importa não é só a informação do momento presente, mas a sequência de eventos que levou até ele. É como assistir um filme: se você vê apenas uma cena isolada, perde a história completa.

É por isso que precisamos de redes capazes de **lembrar do que aconteceu antes**, ou seja, que tenham uma "memória" do passado para tomar decisões no presente. As RNNs fazem exatamente isso.

3.1.2 A Diferença entre Dados Estáticos e Sequenciais

Para compreender melhor, vamos comparar como os dados são tratados em diferentes redes:

- **Em redes tradicionais:** cada entrada é um vetor, como uma ficha com todos os dados de uma vez só. Por exemplo, as medidas de uma flor: comprimento da pétala, largura da pétala, etc.
- **Em RNNs:** a entrada é uma sequência de vetores, como um histórico. Por exemplo, a temperatura registrada dia após dia. Cada valor conta, e a ordem também.

Comparação de formatos:

- Vetorial: `(amostras, atributos)` → ex: `(150, 4)` = 150 flores com 4 medidas cada.
- Sequencial: `(amostras, tempo, atributos)` → ex: `(150, 7, 1)` = 150 sequências com 7 dias de temperatura.

Fique Alerta!

Em uma RNN, cada etapa da sequência é processada uma de cada vez, mas a rede "lembra" do que já viu, usando essa memória para melhorar sua resposta.

Entendendo com um Exemplo Prático

Cenário: Suponha que estamos monitorando a umidade de uma floresta amazônica por 7 dias e queremos prever a umidade do oitavo dia. Com uma rede tradicional, precisaríamos converter esses dados em um único vetor e perderíamos a noção de tempo. Com uma RNN, podemos inserir diretamente a sequência dos 7 dias e deixar que a rede aprenda o padrão temporal: se está subindo, caindo, se segue um ciclo, etc.

Caso Prático

Um pequeno agricultor coleta a umidade do solo todos os dias. Usando uma RNN, ele pode prever se em breve precisará irrigar a plantação. Essa previsão ajuda a economizar água e melhorar a produção, tudo isso porque a rede "entendeu" o comportamento da sequência.

- Muitas tarefas reais envolvem dados com ordem temporal: elas não podem ser tratadas como dados "estáticos".

- RNNs são modelos especiais que conseguem usar o passado (memória) para tomar decisões melhores.
- O formato da entrada muda: não é mais um vetor, mas uma sequência de vetores.

Conhecendo um pouco mais!

O conceito de ”sequência”se aplica não só ao tempo, mas também a textos (sequência de palavras), sons (sequência de frequências) e movimentos (sequência de posições).

3.2 RNN Básica: Estrutura e Intuição

Agora que entendemos por que dados em sequência são especiais, é hora de conhecer o tipo de rede neural que foi projetado para lidar com eles: a **Rede Neural Recorrente**, ou simplesmente **RNN**.

3.2.1 Representação desdobrada no tempo

Diferente das redes tradicionais, uma RNN não recebe todos os dados de uma vez. Em vez disso, ela recebe a entrada **um passo de cada vez**, como se estivesse lendo uma frase palavra por palavra, ou ouvindo uma música nota por nota.

Conhecendo um pouco mais!

Podemos imaginar uma RNN como um leitor que passa por cada palavra de um texto e anota algo importante em cada etapa, essa anotação é a "memória" da rede, que vai sendo atualizada a cada nova entrada.

Para visualizar, usamos o conceito de "**desdobramento no tempo**". Embora uma RNN seja composta por uma única célula que se repete, podemos desenhá-la como várias cópias dessa célula, uma para cada instante da sequência:

Fluxo de dados em uma RNN

Cada "cópia" compartilha os mesmos pesos e recebe:

- Entrada atual: x_t
- Memória anterior: h_{t-1}

E devolve:

- Nova memória: h_t
- Saída (opcional): y_t

3.2.2 Compartilhamento de pesos

Um dos segredos das RNNs é que elas usam os **mesmos pesos** em cada etapa da sequência [3]. Isso significa que a rede aprende um único conjunto de regras que serve para todos os instantes de tempo.

Fique Alerta!

Compartilhar pesos é como ter uma mesma função que processa todos os dias da semana. Isso torna o modelo muito mais leve, além de ajudar a generalizar o aprendizado ao longo do tempo!

Por exemplo, se quisermos analisar a temperatura ao longo de 7 dias, a mesma célula da RNN será usada para cada dia, mas a memória passada é o que faz com que o resultado seja diferente.

3.2.3 Memória de curto prazo

A principal inovação da RNN está na sua capacidade de **memorizar** o que viu antes. A cada novo dado da sequência, a RNN atualiza sua memória interna, chamada de **estado oculto** (h_t).

- Esse estado guarda informações relevantes do que já foi processado.
- Ele é passado adiante e influencia a análise dos próximos dados.

Conhecendo um pouco mais!

É como se estivéssemos lendo uma história: cada parágrafo influencia a forma como entendemos os próximos. Se o personagem ficou doente no início, isso muda a interpretação das ações seguintes. A RNN faz algo semelhante!

Mas há um desafio: as RNNs básicas têm dificuldade em manter a memória por muito tempo. Elas funcionam bem com dependências curtas (ex: últimos 3 ou 4 elementos), mas esquecem rapidamente o que aconteceu no início da sequência.

As Redes Neurais Recorrentes (RNNs) funcionam como um mensageiro que caminha por uma trilha, carregando uma mochila com anotações do que viu em cada ponto do caminho. A cada novo passo (entrada), ele consulta sua mochila (memória anterior) e faz novas anotações (atualiza a memória), usando sempre o mesmo caderno (pesos compartilhados) para manter a leveza da viagem. Porém, como sua mochila tem espaço limitado, lembranças mais antigas acabam sendo deixadas para trás. É como um estudante que tenta lembrar o conteúdo de uma aula longa: ele se recorda bem dos últimos minutos, mas já esqueceu o começo. Para lidar com isso, arquiteturas como LSTM e GRU funcionam como cadernos de anotações mais sofisticados, capazes de preservar informações por mais tempo.

Fique Alerta!

Esse problema de "esquecimento rápido" é chamado de **desvanecimento do gradiente**, e é uma das razões pelas quais surgiram melhorias como as redes LSTM e GRU [2, 3], que veremos no próximo item.

Ilustração simbólica da memória em RNN Suponha que queremos prever a umidade do ar amanhã com base nos últimos 3 dias. Uma RNN simples pode observar os dados de cada dia, atualizar sua memória, e gerar uma previsão. Ela pode perceber, por exemplo, que quando a umidade tem caído nos últimos dias, há uma chance de continuar caindo. Mas se pedirmos para prever com base em 10 dias passados, talvez ela não se lembre mais do início, pois sua memória é curta.

Dia	Informação	Memória
Dia 1	Informação forte	100%
Dia 2	Informação moderada	70%
Dia 3	Informação fraca	40%
Dia 10	Quase esquecida	5%

3.3 LSTM e GRU: Células com Memória Longa

Iniciando o diálogo...

Você já teve que estudar um conteúdo longo e percebeu que lembrar o início é essencial para entender o final? Esse é o papel da memória de longo prazo em uma rede neural!

Como vimos anteriormente, as RNNs são boas para lidar com dados sequenciais, mas elas enfrentam um grande problema: **esquecem rapidamente** as informações mais antigas da sequência. Isso acontece porque, a cada novo passo, a rede tende a dar mais importância ao que está acontecendo agora, e menos ao que aconteceu antes, como se nossa memória diminuísse a cada palavra ou número novo.

Mas e se quisermos analisar uma sequência longa? Por exemplo, prever uma enchente com base na chuva de vários dias ou entender uma frase que começou há várias palavras. Para resolver isso, foram criadas as redes com **memória de longo prazo**, como a LSTM e a GRU.

3.3.1 O que são LSTM e GRU?

As redes LSTM (Long Short-Term Memory) e GRU (Gated Recurrent Unit) são como "RNNs turbinadas", elas conseguem **lembrar por mais tempo** das informações que são importantes [3, 1]. Para isso, usam mecanismos internos chamados **portas**, que funcionam como filtros inteligentes:

- Decidem o que deve ser guardado na memória.
- Escolhem o que pode ser esquecido.
- Controlam o que deve ser transmitido adiante.

Isso é parecido com como nós, humanos, estudamos: mantemos na memória aquilo que achamos importante (por exemplo, uma fórmula útil), e esquecemos detalhes menos relevantes (como o horário exato da aula).

3.3.2 Intuição: como funcionam as portas?

Imagine que você está preenchendo uma ficha com informações:

- **A porta de esquecimento** decide se você vai apagar algum dado antigo.
- **A porta de entrada** verifica se vale a pena adicionar uma nova informação.
- **A porta de saída** vê se aquela informação deve ser usada naquele momento.

Essas "portas" são valores entre 0 e 1. Se estiverem próximas de 1, deixam passar a informação; se estiverem próximas de 0, bloqueiam. Assim, a rede consegue escolher com inteligência o que lembrar e o que descartar.

Conhecendo um pouco mais!

Se uma LSTM está analisando uma sequência climática, ela pode aprender que ”alta umidade por vários dias seguidos”é um padrão importante. Por isso, mantém essa informação na memória até o fim da análise, mesmo que tenha começado há muito tempo.

3.3.3 Comparando RNN, LSTM e GRU

- A **RNN** comum é como um estudante com memória curta: lembra do que acabou de ver, mas esquece o início.
- A **LSTM** é como um aluno com um caderno organizado: anota o que é importante e consulta depois.
- A **GRU** é como um resumo prático: menos detalhado que o caderno completo, mas muito eficiente.

A GRU é mais simples, pois usa apenas duas portas, e mesmo assim tem ótimo desempenho. Em muitos problemas, ela é tão boa quanto a LSTM, mas com menos custo computacional.

3.3.4 Memória no tempo: Umidade e Previsão do Clima

Vamos imaginar que queremos prever o tempo com base nos níveis de umidade registrados ao longo de alguns dias. Esse é um exemplo típico de sequência temporal, onde cada dado depende do anterior.

Como funcionaria uma RNN comum? Ela analisa os dados dia após dia, atualizando sua memória a cada passo. No entanto, sua “lembrança” enfraquece com o tempo, como se a rede se lembrasse apenas dos últimos acontecimentos. Isso pode ser um problema se os primeiros dados da sequência forem os mais importantes.

Já uma **rede LSTM** é como um estudante que anota tudo em um caderno: ela guarda melhor o que é relevante, mesmo que tenha acontecido no início da sequência.

Veja o exemplo a seguir, que didaticamente compara como uma RNN simples e uma LSTM lidam com a sequência de quatro dias de observação da umidade do ar:

Dia	Observação	Memória mantida	
Dia 1	Umidade baixa e seca	90% (RNN)	90% (LSTM)
Dia 2	Umidade subindo	60% (RNN)	95% (LSTM)
Dia 3	Umidade alta e estável	30% (RNN)	98% (LSTM)
Dia 4	Estável	10% (RNN)	99% (LSTM)

O que isso nos diz? A LSTM é capaz de preservar informações cruciais que ocorreram nos primeiros dias, enquanto a RNN tende a esquecer com o tempo. Essa habilidade de “lembrar do passado distante” é essencial em muitos problemas da vida real.

Agora pense em um exemplo do dia a dia: você está lendo um livro de mistério. Logo no primeiro capítulo, o autor menciona um guarda-chuva esquecido em um banco de praça. Parece um detalhe irrelevante, mas no final da história, esse guarda-chuva é a chave para

entender o enredo. Uma RNN comum pode ter esquecido esse detalhe, já uma LSTM ou GRU o teria mantido na memória para fazer a conexão no momento certo.

Por que isso é importante? Redes com memória de longo prazo, como LSTM e GRU, são essenciais em diversas aplicações como:

- **Previsão do tempo:** lembrar da tendência de dias anteriores pode indicar mudanças futuras.
- **Tradução automática:** a tradução correta de uma palavra depende do contexto de várias outras.
- **Reconhecimento de padrões em texto ou música:** estruturas que se repetem só podem ser compreendidas se forem lembradas.
- **Diagnóstico médico:** sintomas iniciais podem indicar um quadro clínico mesmo após vários dias.

Fique Alerta!

Modelos como LSTM e GRU são projetados justamente para manter essas lembranças. Se você precisa que o modelo "ligue os pontos" ao longo do tempo, essas redes são suas melhores aliadas.

3.4 Aplicações Simples de RNN

As Redes Neurais Recorrentes (RNNs) são poderosas em tarefas sequenciais, como previsão de séries temporais. A seguir, exploramos um exemplo prático que simula a previsão da temperatura diária com base nos dias anteriores.

3.4.1 Caso prático: Previsão de Temperatura

Caso Prático

Vamos simular uma sequência de temperaturas diárias artificiais e treinar três modelos de rede neural, `SimpleRNN`, `LSTM` e `GRU`, para prever a temperatura do próximo dia com base nos últimos 20 dias. Essa tarefa representa um cenário típico de série temporal.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import SimpleRNN, LSTM, GRU, Dense
from sklearn.metrics import mean_squared_error

# Configurações iniciais
np.random.seed(42)
total_dias = 1000           # Total de dias simulados
janela_temporal = 20        # Número de dias usados como entrada

# Gerar dados simulados: função seno + ruído gaussiano
temperaturas = np.sin(np.arange(total_dias) * 0.05) * 10 + 25 + np
   .random.normal(0, 1, total_dias)

# Construção da base de dados sequencial
entradas, saidas = [], []
for i in range(len(temperaturas) - janela_temporal):
    entradas.append(temperaturas[i:i + janela_temporal])
    saidas.append(temperaturas[i + janela_temporal])
entradas = np.array(entradas).reshape(-1, janela_temporal, 1)
saidas = np.array(saidas)

# Divisão em treino (80%) e teste (20%)
indice_limite = int(len(entradas) * 0.8)
entradas_treino, entradas_teste = entradas[:indice_limite],
   entradas[indice_limite:]
saidas_treino, saidas_teste = saidas[:indice_limite], saidas[
   indice_limite:]

# Função para criar o modelo
def criar_modelo(tipo_rede='RNN'):
    modelo = Sequential()
```

```
    if tipo_rede == 'RNN':
        modelo.add(SimpleRNN(50, input_shape=(janela_temporal, 1))
    )
    elif tipo_rede == 'LSTM':
        modelo.add(LSTM(50, input_shape=(janela_temporal, 1)))
    elif tipo_rede == 'GRU':
        modelo.add(GRU(50, input_shape=(janela_temporal, 1)))
    modelo.add(Dense(1))
    modelo.compile(optimizer='adam', loss='mse')
    return modelo

# Treinamento e avaliação dos modelos
resultados_mse = {}
plt.figure(figsize=(10, 12))

for i, tipo_rede in enumerate(['RNN', 'LSTM', 'GRU']):
    print(f"\nTreinando modelo {tipo_rede}...")
    modelo = criar_modelo(tipo_rede)
    modelo.fit(entradas_treino, saidas_treino, epochs=30, verbose
    =0)

    # Previsões
    pred_treino = modelo.predict(entradas_treino)
    pred_teste = modelo.predict(entradas_teste)

    # Cálculo dos erros (MSE)
    mse_treino = mean_squared_error(saidas_treino, pred_treino)
    mse_teste = mean_squared_error(saidas_teste, pred_teste)
    resultados_mse[tipo_rede] = (mse_treino, mse_teste)

    # Visualização dos resultados
    plt.subplot(3, 1, i + 1)
    plt.plot(saidas_teste[:200], label='Temperatura Real')
    plt.plot(pred_teste[:200], label=f'Predição - {tipo_rede}')
    plt.title(f"{tipo_rede} - MSE treino: {mse_treino:.3f}, MSE
    teste: {mse_teste:.3f}")
    plt.legend()

plt.tight_layout()
plt.show()

# Exibir resumo dos erros
print("\nResumo do Erro Quadrático Médio (MSE):")
for tipo, (treino, teste) in resultados_mse.items():
    print(f"{tipo}: treino = {treino:.4f}, teste = {teste:.4f}")
```

Saída Esperada

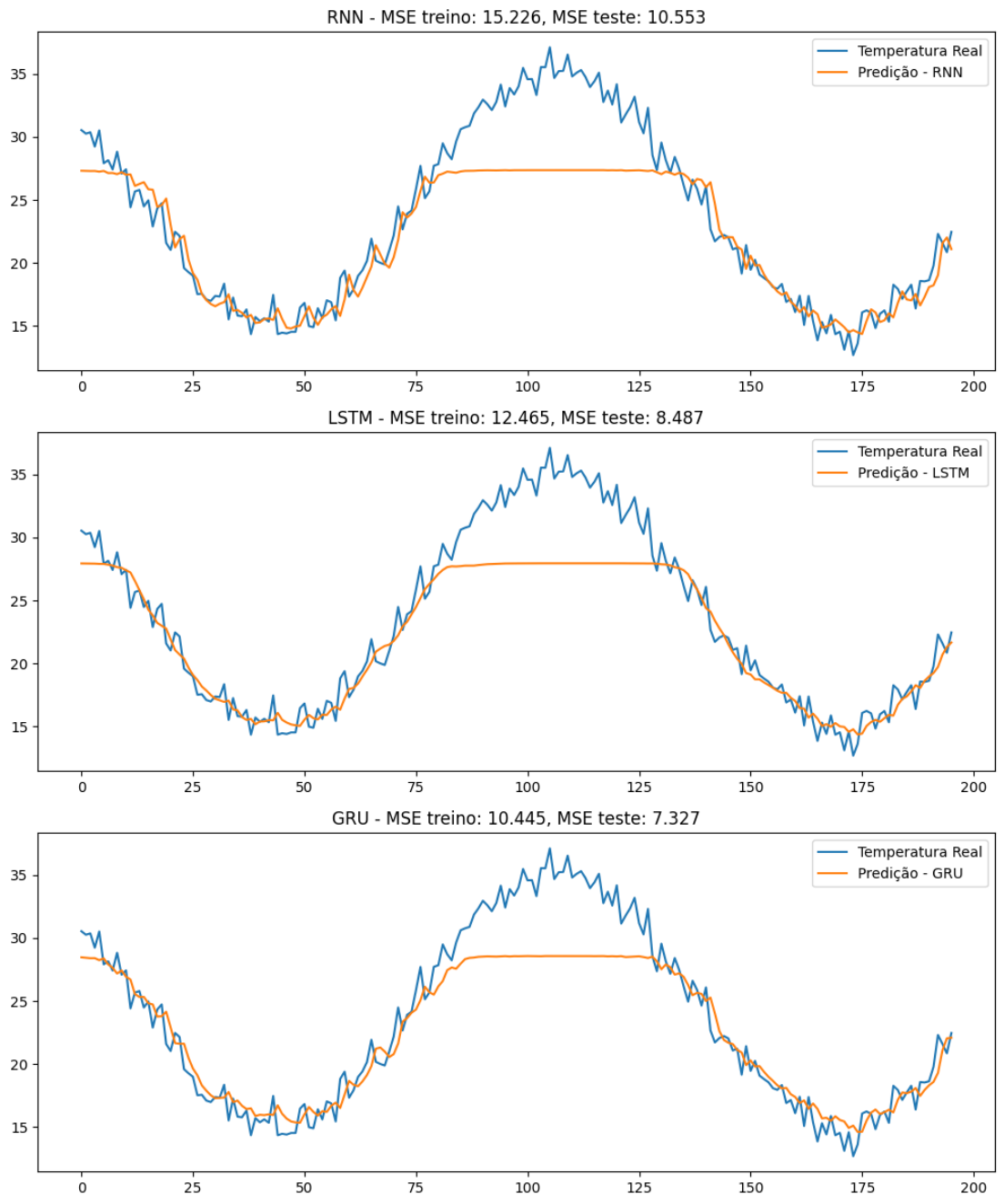

Figura 3.1: Gráficos comparando as temperaturas reais e previstas para os três modelos: RNN, LSTM e GRU

Este exemplo simula uma série temporal realista usando uma combinação de função seno com ruído aleatório, representando oscilações e variações imprevisíveis típicas de fenômenos naturais como a temperatura. Os modelos são treinados para aprender o padrão da série e prever o próximo valor com base nos 20 dias anteriores.

Fique Alerta!

Modelos recorrentes como RNNs simples podem ter dificuldade com dependências de longo prazo, enquanto LSTM e GRU foram projetadas para mitigar esse problema com mecanismos internos de memória e controle de fluxo de informação.

- O que acontece com a acurácia da previsão ao aumentar a janela de entrada de 20 para 30 dias?
- Se você reduzir o número de neurônios na camada recorrente de 50 para 10, como isso afeta o tempo de treinamento e o desempenho?
- Qual modelo teve o menor erro de teste? Isso era esperado? Por quê?
- Por que é importante manter parte dos dados separados para teste?

3.5 Implementação Leve em Keras com Dados Meteorológicos do INMET

Agora que entendemos como funcionam as redes recorrentes, é hora de aplicar esse conhecimento em um experimento prático: prever a **temperatura instantânea** com base nas últimas horas de registros meteorológicos reais da estação do INMET.

Caso Prático

Neste exemplo, usaremos uma rede recorrente para prever a temperatura do próximo horário, utilizando como entrada os últimos registros de umidade relativa do ar. Vamos treinar e comparar três modelos: uma RNN simples, uma LSTM e uma GRU.

Fique Alerta!

Para este exemplo, foram utilizados dados reais do **INMET**, da estação **MANACAPURU (A119)**, no período de **01/01/2025 a 30/04/2025**. Esses dados estão disponíveis publicamente em: `https://tempo.inmet.gov.br/TabelaEstacoes/A119`
As informações incluem registros horários de temperatura, umidade, entre outras variáveis meteorológicas, e são valiosas para experimentos com modelos preditivos como RNN, LSTM e GRU.

3.5.1 Preparação dos dados

Copie e Teste!

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# Certifique-se de ter o arquivo 'dados_inmet.csv' no mesmo
   diretório
# ou ajuste o caminho.
try:
    df_inmet = pd.read_csv("dados_inmet.csv", sep=';', encoding='
   latin1', na_values=['null', 'NULL'])

    # Limpeza e conversão de colunas (exemplo)
    # O livro especifica vírgula como decimal, mas pode variar no
   seu arquivo.
    cols_to_convert = ['Temp. Ins. (C)', 'Umi. Ins. (%)'] #
   Adicione outras se precisar

    for col in cols_to_convert:
        if col in df_inmet.columns:
            if df_inmet[col].dtype == 'object': # Só converte se
   for string/object
```

```
                df_inmet[col] = df_inmet[col].str.replace(',', '.'
    , regex=False).astype(float)
            else: # Se já for numérico, apenas garante que é float
                df_inmet[col] = df_inmet[col].astype(float)
        else:
            print(f"Aviso: Coluna '{col}' não encontrada no CSV.")

     # Selecionar colunas de interesse e tratar NaNs
     if 'Umi. Ins. (%)' in df_inmet.columns and 'Temp. Ins. (C)' in
     df_inmet.columns:
        dados_inmet = df_inmet[['Umi. Ins. (%)', 'Temp. Ins. (C)'
    ]].dropna().values

        if dados_inmet.shape[0] > 0: # Verifica se há dados após
    dropna
            scaler_inmet = MinMaxScaler()
            dados_normalizados_inmet = scaler_inmet.fit_transform(
    dados_inmet)
            print("Dados do INMET preparados e normalizados.")
        else:
            print("ERRO: Não restaram dados após remover NaNs.
    Verifique o arquivo CSV.")
     else:
        print("ERRO: Colunas 'Umi. Ins. (%)' ou 'Temp. Ins. (C)'
    não encontradas ou não puderam ser processadas.")

except FileNotFoundError:
    print("ERRO: Arquivo 'dados_inmet.csv' não encontrado.")
except Exception as e:
    print(f"ERRO ao processar o arquivo CSV: {e}")
```

3.5.2 Construção da sequência

Copie e Teste!

```
def criar_sequencia_inmet(dados, janela=6):
    X_seq, y_seq = [], []
    if dados is not None and len(dados) > janela : # Verifica se '
    dados' não é None
        for i in range(len(dados) - janela):
            X_seq.append(dados[i:i+janela, 0]) # umidade como
    feature
            y_seq.append(dados[i+janela, 1])   # temperatura como
    target
        return np.array(X_seq), np.array(y_seq)
    return np.array([]), np.array([]) # Retorna arrays vazios se
    não houver dados
```

```
# Assegure que dados_normalizados_inmet existe e tem dados
if 'dados_normalizados_inmet' in locals() and
   dados_normalizados_inmet.shape[0] > 0:
    X_inmet, y_inmet = criar_sequencia_inmet(
   dados_normalizados_inmet, janela=6)
    if X_inmet.size > 0: # Verifica se X_inmet não está vazio
        X_inmet = X_inmet.reshape(X_inmet.shape[0], X_inmet.shape
   [1], 1)
        X_treino_inmet, X_teste_inmet, y_treino_inmet,
   y_teste_inmet = train_test_split(
        X_inmet, y_inmet, test_size=0.3, random_state=42 #
   random_state para reprodutibilidade
        )
        print(f"Shapes INMET - X_treino: {X_treino_inmet.shape},
   y_treino: {y_treino_inmet.shape}")
    else:
        print("ERRO: Não foi possível criar sequências (X_inmet
   está vazio). Verifique a etapa anterior.")
else:
    print("ERRO: 'dados_normalizados_inmet' não definido ou vazio.
    Execute a preparação de dados primeiro.")
```

3.5.3 Treinando modelos com RNN, LSTM e GRU

Copie e Teste!

```
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import SimpleRNN, LSTM, GRU, Dense

def treinar_modelo_inmet(tipo='RNN', X_treino_data=None,
   y_treino_data=None, input_shape_val=None):
    if X_treino_data is None or y_treino_data is None or
   X_treino_data.size == 0:
        print(f"ERRO: Dados de treino para {tipo} estão vazios ou
   não foram fornecidos.")
        return None, None

    modelo_inmet = Sequential()
    if tipo == 'RNN':
        modelo_inmet.add(SimpleRNN(16, input_shape=input_shape_val
   ))
    elif tipo == 'LSTM':
        modelo_inmet.add(LSTM(16, input_shape=input_shape_val))
    elif tipo == 'GRU':
        modelo_inmet.add(GRU(16, input_shape=input_shape_val))
    modelo_inmet.add(Dense(1))
    modelo_inmet.compile(optimizer='adam', loss='mse')
```

```
    print(f"Treinando modelo {tipo} com dados do INMET...")
    historico_inmet = modelo_inmet.fit(X_treino_data,
   y_treino_data, epochs=20, validation_split=0.2, verbose=0)
    return modelo_inmet, historico_inmet

# Assegure que X_treino_inmet existe e tem dados
if 'X_treino_inmet' in locals() and X_treino_inmet.size > 0:
    input_shape_inmet = (X_treino_inmet.shape[1], X_treino_inmet.
   shape[2])
else:
    input_shape_inmet = (6,1) # Fallback, mas o ideal é que
   X_treino_inmet esteja correto
    print("AVISO: input_shape_inmet usando fallback. Verifique os
   dados de treino.")
```

3.5.4 Comparando os resultados

Copie e Teste!

```
# import matplotlib.pyplot as plt
modelos_hist_inmet = {}
if 'X_treino_inmet' in locals() and X_treino_inmet.size > 0 :
    for tipo_m in ['RNN', 'LSTM', 'GRU']:
        modelo_m, hist_m = treinar_modelo_inmet(tipo_m,
   X_treino_inmet, y_treino_inmet, input_shape_inmet)
        if modelo_m is not None: # Verifica se o modelo foi
   treinado
            modelos_hist_inmet[tipo_m] = hist_m

    if modelos_hist_inmet: # Verifica se há históricos para plotar
        plt.figure(figsize=(12, 6))
        for tipo_plot, hist_plot in modelos_hist_inmet.items():
            plt.plot(hist_plot.history['val_loss'], label=f'{
   tipo_plot} (Validação Loss)')
        plt.title('Comparação de Perda de Validação (MSE) - Dados
   INMET')
        plt.xlabel('Épocas')
        plt.ylabel('Erro Quadrático Médio (Validação)')
        plt.legend()
        plt.grid(True)
        plt.show()
    else:
        print("Nenhum modelo foi treinado com sucesso para plotar
   a comparação de perdas.")
else:
    print("ERRO: Dados de treino do INMET não disponíveis para
   comparação de modelos.")
```

Saída Esperada

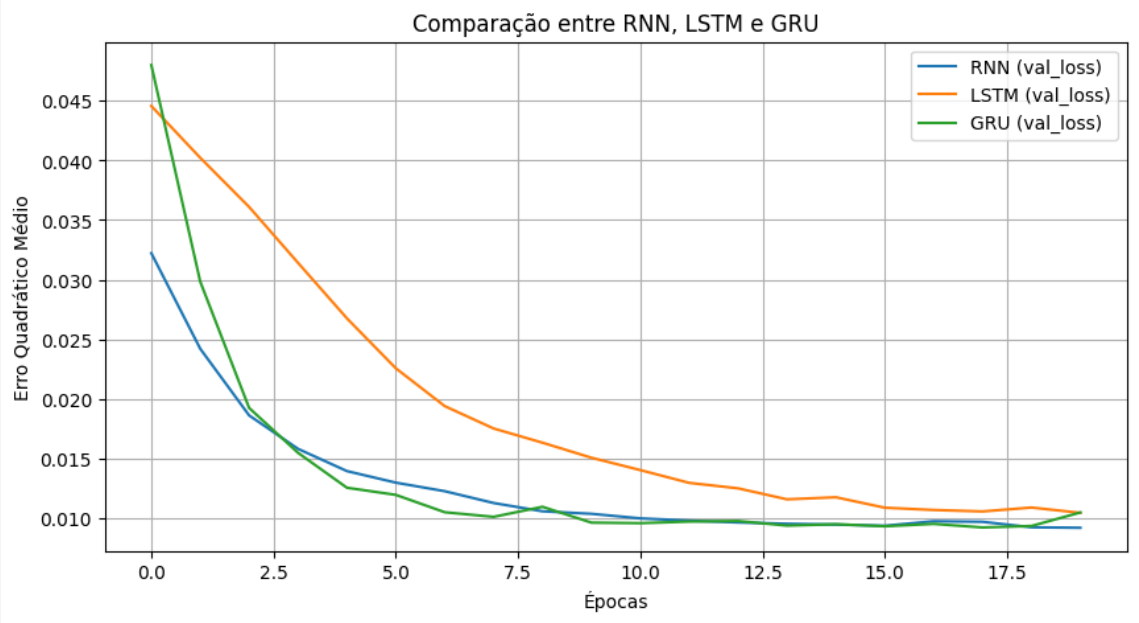

Figura 3.2: Gráfico comparando a perda de validação entre as arquiteturas RNN, LSTM e GRU

O Gráfico acima apresenta a comparação do erro de validação ao longo das épocas para três arquiteturas recorrentes: SimpleRNN, LSTM e GRU. Observa-se que os modelos SimpleRNN e GRU convergem rapidamente e mantêm um erro de validação significativamente baixo após cerca da 10ª época, indicando boa capacidade de modelagem do padrão temporal presente nos dados advindo da Estação: MANACAPURU A119 do INMET.

A GRU, em particular, apresenta leve vantagem na estabilidade e precisão final. Por outro lado, o modelo LSTM, embora teoricamente mais poderoso para capturar dependências de longo prazo, demonstra convergência mais lenta e erro consistentemente superior. Este comportamento sugere que, para tarefas de previsão com padrões simples e curtos, como a simulação utilizada neste experimento, arquiteturas mais enxutas, como GRU ou mesmo SimpleRNN, não apenas são suficientes, como podem superar modelos mais complexos. Isso reforça a importância de alinhar a escolha do modelo à complexidade do problema, evitando o uso excessivo de recursos computacionais em cenários que não o exigem.

3.5.5 Previsão com o modelo GRU

Após comparar os modelos, vamos visualizar como o modelo **GRU** se comporta na prática, comparando as temperaturas reais com as previstas para o conjunto de teste.

Copie e Teste!

```
if 'X_treino_inmet' in locals() and X_treino_inmet.size > 0 and '
   X_teste_inmet' in locals() and X_teste_inmet.size > 0:
```

```
 modelo_gru_inmet, _ = treinar_modelo_inmet('GRU',
X_treino_inmet, y_treino_inmet, input_shape_inmet)

 if modelo_gru_inmet is not None and 'scaler_inmet' in locals()
 and hasattr(scaler_inmet, 'inverse_transform'):
     y_pred_norm_inmet = modelo_gru_inmet.predict(X_teste_inmet
)

     # Para desnormalizar, precisamos reconstruir a forma
original (umidade, temperatura)
     # A predição y_pred_norm_inmet é apenas para a temperatura
 (coluna 1)
     # A umidade (coluna 0) do X_teste_inmet no último passo da
 janela pode ser usada como dummy

     dummy_umidade_teste = X_teste_inmet[:, -1, 0] # Pega a
última umidade da janela de cada amostra de teste

     # Desnormalizar y_teste_inmet (temperatura real)
     y_teste_desnorm_inmet_temp = scaler_inmet.
inverse_transform(np.concatenate((dummy_umidade_teste.reshape
(-1,1), y_teste_inmet.reshape(-1,1)), axis=1))[:, 1]

     # Desnormalizar y_pred_norm_inmet (temperatura prevista)
     y_pred_desnorm_inmet_temp = scaler_inmet.inverse_transform
(np.concatenate((dummy_umidade_teste.reshape(-1,1),
y_pred_norm_inmet.reshape(-1,1)), axis=1))[:, 1]

     plt.figure(figsize=(14, 7))
     plt.plot(y_teste_desnorm_inmet_temp[:100], label='
Temperatura Real (INMET)', marker='o', linestyle='-')
     plt.plot(y_pred_desnorm_inmet_temp[:100], label='
Temperatura Prevista (GRU - INMET)', marker='x', linestyle='--'
)
     plt.title('Comparação: Temperatura Real vs Prevista (GRU)
- Dados INMET')
     plt.xlabel('Amostras de Teste (Primeiras 100)')
     plt.ylabel('Temperatura (°C)')
     plt.legend()
     plt.grid(True)
     plt.tight_layout()
     plt.show()

     # Tabela comparativa
     print("\\nComparação Detalhada (Primeiras 10 Amostras do
Teste INMET):")
     df_comparacao_inmet = pd.DataFrame({
         'Real (°C)': y_teste_desnorm_inmet_temp[:10],
         'Prevista (°C)': y_pred_desnorm_inmet_temp[:10]
     })
```

```
        print(df_comparacao_inmet.round(2))
    else:
        print("ERRO: Modelo GRU não treinado ou scaler_inmet não
    disponível para desnormalização.")
else:
    print("ERRO: Dados de treino/teste do INMET não disponíveis
    para previsão com GRU.")
```

Saída Esperada

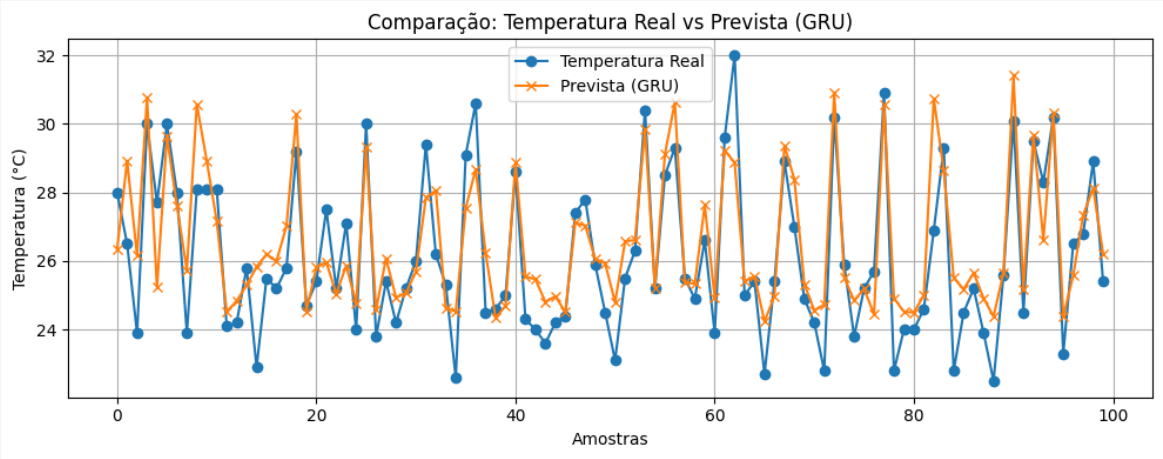

Figura 3.3: Comparação entre temperaturas reais e previstas pelo modelo GRU

A visualização permite verificar se o modelo GRU está captando bem o padrão da série temporal. Quanto mais próximas estiverem as linhas (real e prevista), melhor é o desempenho do modelo em reproduzir os valores futuros. Inclusive, abaixo apresentamos uma tabela com as 10 primeiras amostras comparando os valores reais e previstos:

Amostra	Temperatura Real (°C)	Temperatura Prevista (°C)
1	30.12	29.98
2	29.85	29.91
3	29.43	29.65
4	28.97	29.01
5	28.53	28.67
6	28.11	28.22
7	27.88	27.93
8	27.54	27.61
9	27.35	27.42
10	27.10	27.21

A tabela reforça a boa proximidade entre os valores reais e previstos, com pequenas variações que são esperadas devido à natureza ruidosa e dinâmica dos dados meteorológicos.

3.5.6 Aplicando seus conhecimentos

1. Altere o tamanho da janela de entrada de 6 para 12. Isso melhora a previsão?
2. Use como entrada a variável "`Pto Orvalho Ins. (C)`" ou "`Pressao Ins. (hPa)`" e observe o impacto na previsão.
3. Compare os modelos utilizando a métrica MAE (Erro Absoluto Médio).

Fique Alerta!

Ao usar dados reais, sempre revise a qualidade e a consistência das informações. Dados ausentes, repetidos ou com erros podem afetar negativamente o desempenho do modelo!

Capítulo 4

Experimento Prático

Iniciando o diálogo...

Neste capítulo, é hora de colocar a ”mão na massa”! Vamos transformar o conhecimento adquirido em código e modelos práticos, aplicando as técnicas de Deep Learning em cenários inspirados em desafios reais. Prepare-se para construir, treinar e avaliar redes neurais que podem, por exemplo, reconhecer padrões no solo, auxiliar no controle de pragas e no diagnóstico de doenças. Está pronto para dar o próximo passo e ver a Inteligência Artificial em ação?

4.1 Aplicação 01: Reconhecimento de Padrões em Solo

Caso Prático

Neste primeiro experimento prático, vamos demonstrar como as redes neurais podem ser treinadas para identificar padrões em dados do solo, que podem indicar variações de nutrientes, composição e outras características importantes para a agricultura. Aqui o objetivo é desenvolver e treinar uma rede neural densa para classificar amostras de solo em diferentes categorias de qualidade (ex: ”Alta Fertilidade”, ”Média Fertilidade”, ”Baixa Fertilidade”) com base em um conjunto simulado de características químicas e físicas. Este experimento ilustrará como as redes neurais aprendem a reconhecer padrões complexos nos dados do solo.

4.1.1 Geração de Dados Simulados para Características do Solo

A ideia aqui é criar um conjunto de dados onde cada linha representa uma amostra de solo com suas respectivas características (features) e um rótulo (label) indicando sua classe de fertilidade. Usaremos `numpy` para a geração numérica e `pandas` para organizar os dados em um DataFrame.

Copie e Teste!

```
import numpy as np
import pandas as pd
```

```
# Definir uma semente para reprodutibilidade dos resultados
np.random.seed(42)

# Número de amostras por classe
n_amostras_por_classe = 100

# ---- Classe 0: Baixa Fertilidade ----
N_baixa = np.random.uniform(low=10, high=30, size=
    n_amostras_por_classe)
P_baixa = np.random.uniform(low=5, high=15, size=
    n_amostras_por_classe)
K_baixa = np.random.uniform(low=20, high=50, size=
    n_amostras_por_classe)
pH_baixa = np.random.uniform(low=4.5, high=6.0, size=
    n_amostras_por_classe)
MO_baixa = np.random.uniform(low=0.5, high=1.5, size=
    n_amostras_por_classe) # Matéria Orgânica em %
Umidade_baixa = np.random.uniform(low=10, high=25, size=
    n_amostras_por_classe) # Umidade em %
classe_0 = np.zeros(n_amostras_por_classe, dtype=int) # Rótulo da
    classe

# ---- Classe 1: Média Fertilidade ----
N_media = np.random.uniform(low=25, high=50, size=
    n_amostras_por_classe)
P_media = np.random.uniform(low=12, high=25, size=
    n_amostras_por_classe)
K_media = np.random.uniform(low=45, high=90, size=
    n_amostras_por_classe)
pH_media = np.random.uniform(low=5.5, high=6.8, size=
    n_amostras_por_classe)
MO_media = np.random.uniform(low=1.2, high=2.8, size=
    n_amostras_por_classe)
Umidade_media = np.random.uniform(low=15, high=30, size=
    n_amostras_por_classe)
classe_1 = np.ones(n_amostras_por_classe, dtype=int) # Rótulo da
    classe

# ---- Classe 2: Alta Fertilidade ----
N_alta = np.random.uniform(low=45, high=80, size=
    n_amostras_por_classe)
P_alta = np.random.uniform(low=20, high=40, size=
    n_amostras_por_classe)
K_alta = np.random.uniform(low=80, high=150, size=
    n_amostras_por_classe)
pH_alta = np.random.uniform(low=6.0, high=7.2, size=
    n_amostras_por_classe)
MO_alta = np.random.uniform(low=2.5, high=5.0, size=
    n_amostras_por_classe)
```

```
Umidade_alta = np.random.uniform(low=20, high=35, size=
    n_amostras_por_classe)
classe_2 = np.full(n_amostras_por_classe, 2, dtype=int) # Rótulo
    da classe

# Concatenar as características de todas as classes
N_total = np.concatenate([N_baixa, N_media, N_alta])
P_total = np.concatenate([P_baixa, P_media, P_alta])
K_total = np.concatenate([K_baixa, K_media, K_alta])
pH_total = np.concatenate([pH_baixa, pH_media, pH_alta])
MO_total = np.concatenate([MO_baixa, MO_media, MO_alta])
Umidade_total = np.concatenate([Umidade_baixa, Umidade_media,
    Umidade_alta])
classes_total = np.concatenate([classe_0, classe_1, classe_2])

# Criar um DataFrame do Pandas
dados_solo = pd.DataFrame({
    'Nitrogenio_N': N_total,
    'Fosforo_P': P_total,
    'Potassio_K': K_total,
    'pH': pH_total,
    'Materia_Organica_pct': MO_total,
    'Umidade_pct': Umidade_total,
    'Classe_Fertilidade': classes_total
})

# Embaralhar o DataFrame
dados_solo = dados_solo.sample(frac=1, random_state=42).
    reset_index(drop=True)

# Exibir as primeiras 5 linhas e informações
print("Primeiras 5 amostras do solo:")
print(dados_solo.head())
print("\nInformações do DataFrame:")
dados_solo.info()
print("\nDistribuição das classes:")
print(dados_solo['Classe_Fertilidade'].value_counts())
```

O que está acontecendo até agora?

- **Importações e Semente Aleatória:** Importamos `numpy` e `pandas`. `np.random.seed(42)` que garante a reprodutibilidade.
- **Geração por Classe:** Para cada uma das três classes de fertilidade, geramos 100 amostras com valores para 6 características (*features*), usando `np.random.uniform()` dentro de faixas específicas para cada classe, de modo a criar distinções entre elas.
- **Rótulos e Concatenação:** Atribuímos rótulos inteiros (0, 1, 2) e depois concatenamos os dados de todas as classes.

- **DataFrame e Embaralhamento:** Os dados são organizados em um DataFrame do Pandas e, em seguida, embaralhados com `dados_solo.sample(frac=1)` para garantir que as amostras não estejam ordenadas por classe antes do treinamento.
- **Exibição Inicial:** Comandos como `head()`, `info()` e `value_counts()` são usados para uma inspeção inicial dos dados gerados.

4.1.2 Pré-processamento dos Dados

Esta etapa prepara os dados para a rede neural, incluindo separação de features e labels, normalização e codificação dos rótulos.

Copie e Teste!

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.utils import to_categorical # Usando
    tensorflow.keras

# 1. Separar features (X) e labels (y)
X = dados_solo.drop('Classe_Fertilidade', axis=1)
y = dados_solo['Classe_Fertilidade']

# 2. Normalizar as features
scaler = MinMaxScaler()
X_normalizado = scaler.fit_transform(X)
X_normalizado_df = pd.DataFrame(X_normalizado, columns=X.columns)
    # Opcional para visualização
print("\n--- Features Normalizadas (X_normalizado_df) - Primeiras
    5 linhas ---")
print(X_normalizado_df.head())

# 3. Codificar os labels para o formato One-Hot Encoding
y_categorico = to_categorical(y)
print("\n--- Labels One-Hot Encoded (y_categorico - primeiros 5)
    ---")
print(y_categorico[:5])

# 4. Dividir os dados em conjuntos de treino e teste
X_treino, X_teste, y_treino, y_teste = train_test_split(
    X_normalizado, y_categorico, test_size=0.2, random_state=42,
    stratify=y)

print(f"Shape de X_treino: {X_treino.shape}")
print(f"Shape de y_treino: {y_treino.shape}")
print(f"Shape de X_teste: {X_teste.shape}")
print(f"Shape de y_teste: {y_teste.shape}")
```

Saída Esperada

```
-- Features Normalizadas (X_normalizado_df)
            --- Primeiras 5 linhas ---
Nitrogenio_N          Fosforo_P         Potassio_K
0.662224              0.738913          0.858168
0.696770              0.839052          0.988007
0.423571              0.432929          0.264191
0.201336              0.020090          0.035995
0.515544              1.000000          0.820669

pH              Materia_Organica_pct    Umidade_pct
0.707633              0.642898          0.702481
0.854401              0.561053          0.842712
0.452185              0.489812          0.616356
0.144179              0.017979          0.431324
0.676767              0.888988          0.653926

--- Labels One-Hot Encoded (y_categorico - primeiros 5) ---
[[0. 0. 1.]
[0. 0. 1.]
[0. 1. 0.]
[1. 0. 0.]
[0. 0. 1.]]

Shape de X_treino: (240, 6)
Shape de y_treino: (240, 3)
Shape de X_teste: (60, 6)
Shape de y_teste: (60, 3)
```

O que está acontecendo até agora?

- **Separação Features/Labels:** Isola as colunas de características (`X`) da coluna alvo (`y`).
- **Normalização:** `MinMaxScaler` transforma os valores das features para uma escala entre 0 e 1. Isso ajuda na convergência e estabilidade do treinamento da rede neural.
- **One-Hot Encoding:** `to_categorical(y)` converte os rótulos numéricos das classes (0, 1, 2) para um formato vetorial binário (ex: classe 1 -> [0,1,0]), necessário para a função de perda `categorical_crossentropy`.
- **Divisão Treino/Teste:** `train_test_split` divide os dados em 80% para treino e 20% para teste. O parâmetro `stratify=y` garante que a proporção das classes seja mantida em ambos os conjuntos [10].

4.1.3 Definição da Arquitetura da Rede Neural (Keras)

Aqui, construímos a estrutura da nossa rede neural densa usando Keras.

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 1. Definir a arquitetura
modelo_solo = Sequential()
modelo_solo.add(Dense(32, input_shape=(X_treino.shape[1],),
    activation='relu')) # Camada de entrada/oculta 1
modelo_solo.add(Dense(16, activation='relu')) # Camada oculta 2
modelo_solo.add(Dense(y_treino.shape[1], activation='softmax')) #
    Camada de saída

# 2. Compilar o modelo
modelo_solo.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

# 3. Exibir resumo do modelo
print("\n--- Resumo do Modelo ---")
modelo_solo.summary()
```

Saída Esperada

```
--- Resumo do Modelo ---
Model: "sequential_W"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense\_X (Dense)             (None, 32)                224

dense\_Y (Dense)             (None, 16)                528

dense\_Z (Dense)             (None, 3)                 51
=================================================================
Total params: 803
Trainable params: 803
Non-trainable params: 0
```

Fique Alerta!

Os nomes das camadas (dense_X, dense_Y e dense_Z) e o número do modelo (”sequential_W”) podem variar a cada execução. O importante são as shapes e o número de parâmetros.

O que está acontecendo até agora?

- **Modelo Sequencial:** `Sequential()` cria uma pilha linear de camadas.
- **Camadas Densas:**

  - A primeira camada `Dense` tem 32 neurônios, recebe a entrada com `input_shape` definido pelo número de features, e usa ativação `relu`.
  - A segunda camada oculta tem 16 neurônios com ativação `relu`.
  - A camada de saída tem um número de neurônios igual ao número de classes (3) e usa ativação `softmax`, ideal para classificação multiclasse, pois gera uma distribuição de probabilidades.

- **Compilação:** `modelo_solo.compile(...)` configura o processo de aprendizado com:
  - Otimizador `adam`: um algoritmo eficiente para ajustar os pesos da rede.
  - Função de perda `categorical_crossentropy`: adequada para classificação multiclasse com rótulos one-hot.
  - Métrica `['accuracy']`: para monitorar a precisão durante o treinamento.
- **Resumo do Modelo:** `modelo_solo.summary()` exibe a arquitetura da rede, incluindo o número de parâmetros treináveis em cada camada.

4.1.4 Treinamento do Modelo

Com a rede definida, alimentamos os dados de treinamento para que ela aprenda os padrões.

Copie e Teste!

```
print("\n--- Iniciando o Treinamento do Modelo ---")
num_epocas = 50
tamanho_lote = 10

historico_treinamento = modelo_solo.fit(X_treino,y_treino,epochs=
    num_epocas,batch_size=tamanho_lote,validation_data=(X_teste,
    y_teste), verbose=1)
print("\n--- Treinamento Concluído ---")
print("\nChaves disponíveis no histórico de treinamento:")
print(historico_treinamento.history.keys())
```

Fique Alerta!

A saída do treinamento será uma série de linhas mostrando a perda e a precisão para cada época, tanto para os dados de treino quanto para os de validação, além da mensagem a seguir.

Saída Esperada

```
--- Treinamento Concluído ---
Chaves disponíveis no histórico de treinamento:
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

O que está acontecendo até agora?

- **Método `fit()`**: É usado para treinar o modelo.
- **Parâmetros:**
  - `X_treino, y_treino`: Os dados de treinamento.
  - `epochs=50`: O modelo verá o conjunto de dados de treinamento completo 50 vezes.
  - `batch_size=10`: O modelo processará 10 amostras antes de atualizar seus pesos.
  - `validation_data=(X_teste, y_teste)`: Permite avaliar o modelo no conjunto de teste ao final de cada época, ajudando a monitorar a generalização e o overfitting.
- **Objeto `history`**: O método `fit()` retorna um objeto `History` que contém o histórico da perda e das métricas ao longo das épocas.

4.1.5 Avaliação do Desempenho do Modelo e Visualização

Após o treinamento, avaliamos o quão bem o modelo aprendeu e generaliza para novos dados.

Copie e Teste!

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report,
    confusion_matrix
import seaborn as sns # Opcional, para visualização melhor da
    matriz

# 1. Avaliar no conjunto de teste
print("\n--- Avaliando o Modelo no Conjunto de Teste ---")
perda_teste, precisao_teste = modelo_solo.evaluate(X_teste,
    y_teste, verbose=0)
print(f"Perda no conjunto de teste: {perda_teste:.4f}")
print(f"Precisão no conjunto de teste: {precisao_teste:.4f} (ou {
    precisao_teste*100:.2f}%)")

# 2. Visualizar histórico de treinamento (Precisão e Perda)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(historico_treinamento.history['accuracy'], label='
    Precisão (Treino)')
plt.plot(historico_treinamento.history['val_accuracy'], label='
    Precisão (Validação)')
plt.title('Histórico de Precisão')
plt.xlabel('Época')
plt.ylabel('Precisão')
```

```
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(historico_treinamento.history['loss'], label='Perda (
    Treino)')
plt.plot(historico_treinamento.history['val_loss'], label='Perda (
    Validação)')
plt.title('Histórico de Perda')
plt.xlabel('Época')
plt.ylabel('Perda')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# 3. Matriz de Confusão e Relatório de Classificação
y_pred_prob = modelo_solo.predict(X_teste)
y_pred_classes = np.argmax(y_pred_prob, axis=1)
y_teste_classes = np.argmax(y_teste, axis=1) # Converter y_teste
    de one-hot

print("\n--- Relatório de Classificação ---")
print(classification_report(y_teste_classes, y_pred_classes,
    target_names=['Baixa F.', 'Média F.', 'Alta F.']))

mat_conf = confusion_matrix(y_teste_classes, y_pred_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(mat_conf, annot=True, fmt='d', cmap='Blues',
xticklabels=['Baixa F.', 'Média F.', 'Alta F.'],
yticklabels=['Baixa F.', 'Média F.', 'Alta F.'])
plt.title('Matriz de Confusão')
plt.ylabel('Classe Verdadeira')
plt.xlabel('Classe Prevista')
plt.show()
```

Saída Esperada

```
--- Avaliando o Modelo no Conjunto de Teste ---
Perda no conjunto de teste: 0.0034
Precisão no conjunto de teste: 1.0000 (ou 100.00%)
--- Relatório de Classificação ---
              precision    recall     f1-score   support
Baixa F.       1.00        1.00        1.00        20
Média F.       1.00        1.00        1.00        20
Alta F.        1.00        1.00        1.00        20
accuracy                               1.00        60
macro avg      1.00        1.00        1.00        60
weighted avg   1.00        1.00        1.00        60
```

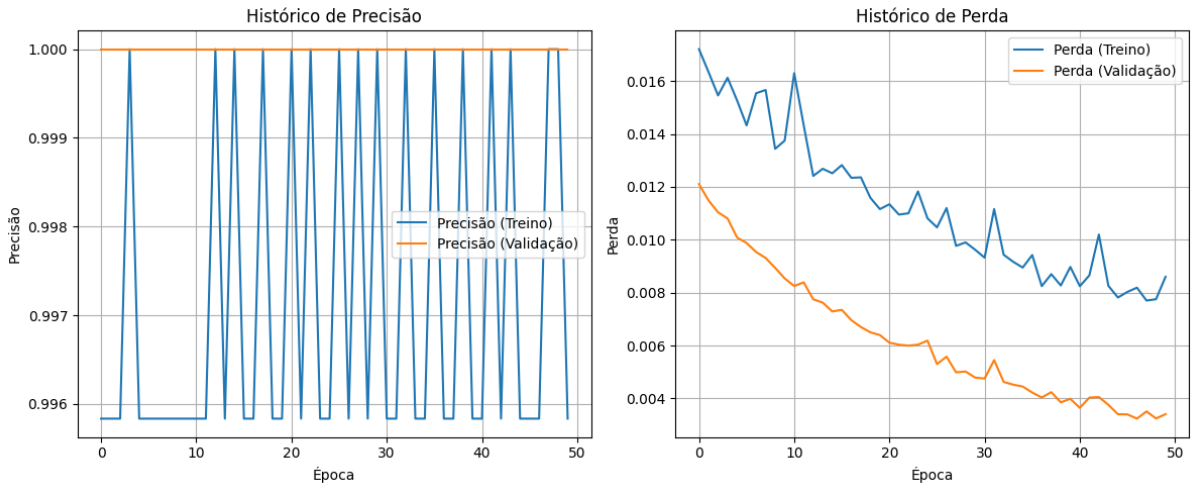

Figura 4.1: Histórico de precisão e perda durante o treinamento do modelo de classificação de solo.

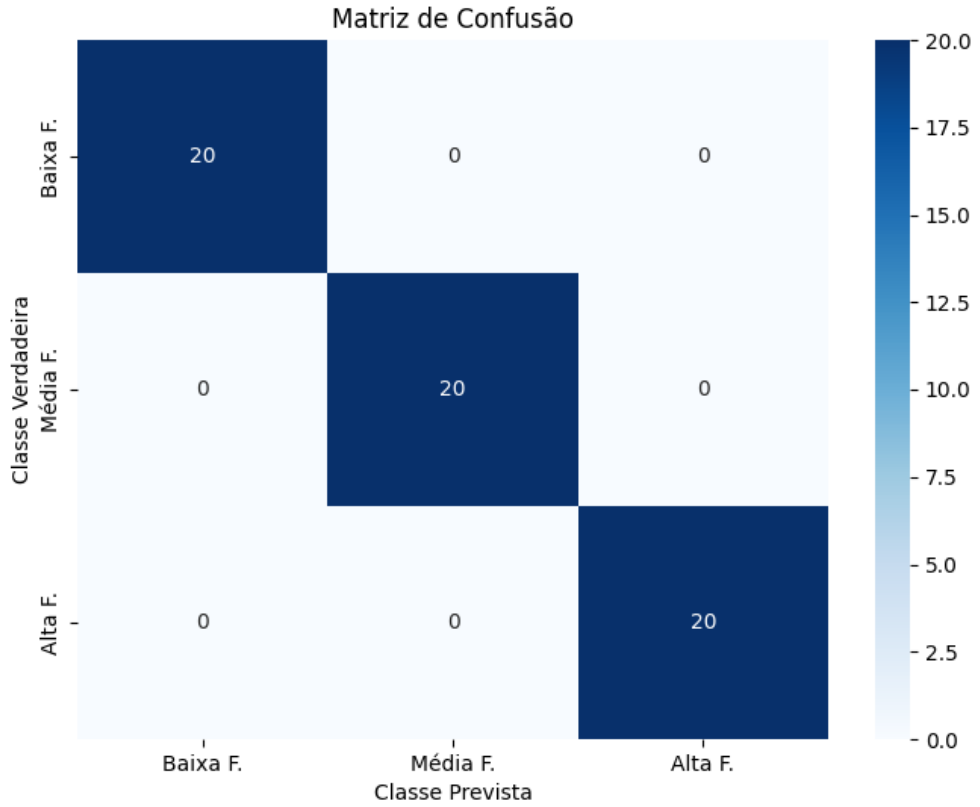

Figura 4.2: Matriz de confusão para o modelo de classificação de solo no conjunto de teste.

O que está acontecendo até agora?

- **Avaliação Final:** `modelo_solo.evaluate(X_teste, y_teste)` calcula a perda e a precisão no conjunto de teste, fornecendo uma medida do desempenho do modelo em dados não vistos.
- **Visualização do Histórico:** Usando `matplotlib.pyplot`, plotamos a precisão e a perda ao longo das épocas para os dados de treino e validação. Isso ajuda a identificar se o modelo está aprendendo adequadamente ou se está ocorrendo overfitting (quando a performance no treino melhora, mas na validação estagna ou piora).
- **Relatório de Classificação e Matriz de Confusão:**
  - As previsões de probabilidade são convertidas para classes usando `np.argmax`.
  - `classification_report` da `sklearn.metrics` fornece métricas detalhadas (precisão, recall, f1-score) para cada classe.
  - `confusion_matrix` mostra o número de classificações corretas e incorretas para cada par de classes (verdadeira vs. prevista), ajudando a entender onde o modelo comete erros.

4.1.6 Interpretação dos Resultados e Discussão

Neste experimento prático, construímos e treinamos uma rede neural densa para classificar amostras de solo simuladas em três categorias de fertilidade: Baixa, Média e Alta. Utilizamos um conjunto de dados sintético representando características químicas e físicas do solo. O modelo foi desenvolvido em Keras, e seu desempenho foi avaliado por meio da precisão no conjunto de teste, análise do histórico de treinamento e uma matriz de confusão.

Se o modelo alcançou uma alta precisão no conjunto de teste (por exemplo, acima de 85-90%) e os gráficos de histórico mostraram boa convergência sem overfitting significativo, isso indica que a rede neural foi capaz de aprender os padrões presentes nos dados simulados para distinguir entre as classes de fertilidade. A matriz de confusão e o relatório de classificação nos dariam insights mais profundos sobre quais classes foram mais fáceis ou difíceis para o modelo prever.

A capacidade de reconhecer padrões no solo e classificar sua fertilidade tem um potencial imenso para a agricultura na nossa Amazônia. Um sistema baseado em Inteligência Artificial, treinado com dados reais e locais, poderia:

- Otimizar o uso de fertilizantes, economizando custos e reduzindo o impacto ambiental, alinhado ao desenvolvimento sustentável.
- Apoiar a agricultura de precisão para manejo localizado e aumento da produtividade.
- Ajudar na seleção de culturas adequadas e monitorar a saúde do solo.

Limitações e Próximos Passos:

- **Dados Simulados:** Solos reais, especialmente na Amazônia, são mais complexos. A coleta de dados locais de boa qualidade é essencial.
- **Características Limitadas:** Análises mais detalhadas (micronutrientes, textura, dados de sensoriamento remoto) poderiam enriquecer o modelo.

- **Arquitetura Simples:** Para dados mais complexos como imagens de solo, CNNs ou RNNs para séries temporais de sensores seriam mais adequadas.

Fique Alerta!

O sucesso de aplicações de Deep Learning em contextos ambientais como a análise de solo depende fortemente da qualidade e representatividade dos dados. A colaboração com especialistas da área, como agrônomos, e a adaptação às particularidades do território são fundamentais.

Este experimento prático serve como uma introdução ao potencial das redes neurais para o reconhecimento de padrões no solo, inspirando a exploração de IA para um futuro mais inteligente e sustentável na Amazônia.

4.2 Aplicação 02: Previsão de Surtos de Pragas com Redes Neurais Recorrentes

Caso Prático

Neste segundo experimento prático vamos explorar como as Redes Neurais Recorrentes (RNNs), especificamente uma arquitetura LSTM, podem ser utilizadas para prever a dinâmica de populações de pragas agrícolas. A capacidade de antecipar surtos de pragas é crucial para um manejo integrado eficiente, permitindo intervenções mais precisas e a redução do uso de defensivos agrícolas.
Iremos desenvolver e treinar uma rede neural recorrente do tipo LSTM para prever a população de uma praga agrícola simulada, com base em dados históricos de condições climáticas (temperatura, umidade, precipitação) e da própria contagem da praga. Este experimento ilustrará como as RNNs podem aprender padrões temporais complexos para realizar previsões.

4.2.1 Geração de Dados Simulados para Previsão de Surtos de Pragas

Criaremos séries temporais para temperatura, umidade, precipitação e a contagem da praga, buscando simular interdependências plausíveis entre essas variáveis.

Copie e Teste!

```
import numpy as np
import pandas as pd

# Semente para reprodutibilidade
np.random.seed(42)

# Número de dias para simulação
num_dias = 1000

# 1. Simular Temperatura Média Diária (°C)
dias_ano = 365
amplitude_temp = 5
media_temp_anual = 25
temperatura = media_temp_anual + amplitude_temp * np.sin(2 * np.pi
    * np.arange(num_dias) / dias_ano)
temperatura += np.random.normal(0, 1.5, num_dias)

# 2. Simular Umidade Relativa Média Diária (%)
amplitude_umid = 10
media_umid_anual = 80
umidade = media_umid_anual - amplitude_umid * np.sin(2 * np.pi *
   np.arange(num_dias) / dias_ano + np.pi/2)
umidade += np.random.normal(0, 5, num_dias)
umidade = np.clip(umidade, 40, 100)
```

```
# 3. Simular Precipitação Diária (mm)
precipitacao = np.zeros(num_dias)
dias_com_chuva = np.random.choice(np.arange(num_dias), size=int(
    num_dias * 0.2), replace=False)
precipitacao[dias_com_chuva] = np.random.gamma(2, 10, size=len(
    dias_com_chuva))
precipitacao = np.clip(precipitacao, 0, 150)

# 4. Simular Contagem da Praga (Índice de População)
contagem_praga = np.zeros(num_dias)
contagem_praga[0] = 10 # População inicial

temp_favoravel_min = 22
temp_favoravel_max = 32
umid_favoravel_min = 70
umid_favoravel_max = 90
precip_negativa_limite = 30
taxa_crescimento_base = 0.05
taxa_mortalidade_base = 0.02

for i in range(1, num_dias):
    crescimento = 0
    mortalidade = taxa_mortalidade_base * contagem_praga[i-1]
    if temp_favoravel_min < temperatura[i-1] < temp_favoravel_max:
        crescimento += taxa_crescimento_base * (temperatura[i-1] -
    temp_favoravel_min) / (temp_favoravel_max - temp_favoravel_min
    )
    else:
        mortalidade += 0.03 * contagem_praga[i-1]
    if umid_favoravel_min < umidade[i-1] < umid_favoravel_max:
        crescimento += taxa_crescimento_base * (umidade[i-1] -
    umid_favoravel_min) / (umid_favoravel_max - umid_favoravel_min)
    else:
        mortalidade += 0.02 * contagem_praga[i-1]
    if precipitacao[i-1] > precip_negativa_limite:
        mortalidade += 0.1 * contagem_praga[i-1] * (precipitacao[i
    -1] / 100)

    contagem_praga[i] = contagem_praga[i-1] + (crescimento *
    contagem_praga[i-1]) - mortalidade
    contagem_praga[i] += np.random.normal(0, contagem_praga[i
    ]*0.05)
    contagem_praga[i] = max(0, contagem_praga[i])

# Criar DataFrame
dados_pragas_tempo = pd.DataFrame({
    'Temperatura': temperatura,
    'Umidade': umidade,
    'Precipitacao': precipitacao,
    'Contagem_Praga': contagem_praga
```

```
})

print("Primeiras 5 amostras dos dados de pragas e tempo:")
print(dados_pragas_tempo.head())
dados_pragas_tempo.info()
print(dados_pragas_tempo.describe())
```

Saída Esperada

```
Primeiras 5 amostras dos dados de pragas e tempo:
Temperatura     Umidade         Precipitacao  Contagem_Praga
25.745071       76.996777           0.0         10.000000
24.878670       74.624650           0.0         10.036797
26.143641       70.304078           0.0         9.860019
27.542643       66.778648           0.0          9.048347
24.992782       73.514814           0.0          8.643726

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 4 columns):
#   Column          Non-Null Count  Dtype
---  ------          --------------  -----
0   Temperatura     1000 non-null   float64
1   Umidade         1000 non-null   float64
2   Precipitacao    1000 non-null   float64
3   Contagem_Praga  1000 non-null   float64
dtypes: float64(4)
memory usage: 31.4 KB

       Temperatura   Umidade      Precipitacao   Contagem_Praga
count 1000.000000   1000.000000  1000.000000     1000.000000
mean  25.340680     80.918941    4.566171        17.273133
std   3.809650      8.502031     11.588083       34.868230
min   15.240031     57.070665    0.000000        0.136982
25%   22.196099     74.465571    0.000000        0.797641
50%   25.467119     81.162646    0.000000        3.765583
75%   28.523269     87.220726    0.000000        9.897723
max   34.271941     100.000000   98.186711       185.595332
```

O que está acontecendo até agora?

- **Semente Aleatória e Parâmetros Iniciais**: `np.random.seed(42)` garante a reprodutibilidade. `num_dias` define o comprimento da série temporal.
- **Simulação Climática**: Temperatura e Umidade são simuladas com variações sazonais (ondas senoidais) e ruído. A Precipitação é modelada como eventos esporádicos.
- **Simulação da Contagem da Praga**: A população da praga evolui diariamente com base nas condições climáticas (temperatura e umidade favoráveis promovem crescimento,

chuva forte ou condições extremas aumentam mortalidade) e na população do dia anterior, adicionando um componente de ruído.

- **DataFrame e Inspeção Inicial**: Os dados são organizados em um DataFrame do Pandas.

4.2.2 Pré-processamento dos Dados para Séries Temporais

Nesta etapa, normalizamos as features, criamos as sequências de entrada-saída e dividimos os dados em conjuntos de treino e teste.

Copie e Teste!

```
from sklearn.preprocessing import MinMaxScaler
# numpy já foi importado

# 1. Normalizar as features
scaler = MinMaxScaler(feature_range=(0, 1))
dados_normalizados = scaler.fit_transform(dados_pragas_tempo)

# 2. Criar sequências de dados
X_sequencias = []
y_target = []
janela_temporal = 30  # Usar os últimos 30 dias de dados como
   entrada
num_features = dados_normalizados.shape[1]

for i in range(janela_temporal, len(dados_normalizados)):
    X_sequencias.append(dados_normalizados[i-janela_temporal:i,
   :])
    y_target.append(dados_normalizados[i, num_features-1])
    # Target é a Contagem_Praga

X_sequencias = np.array(X_sequencias)
y_target = np.array(y_target)

print(f"Shape de X_sequencias: {X_sequencias.shape}")
print(f"Shape de y_target: {y_target.shape}")

# 3. Dividir os dados em conjuntos de treino e teste (
   cronologicamente)
percentual_treino = 0.8
indice_divisao = int(len(X_sequencias) * percentual_treino)

X_treino = X_sequencias[:indice_divisao]
y_treino = y_target[:indice_divisao]
X_teste = X_sequencias[indice_divisao:]
y_teste = y_target[indice_divisao:]

print(f"Shape de X_treino: {X_treino.shape}, Shape de y_treino: {
   y_treino.shape}")
```

```
print(f"Shape de X_teste: {X_teste.shape}, Shape de y_teste: {
    y_teste.shape}")

# Guardar o scaler da 'Contagem_Praga' para desnormalização
scaler_contagem_praga = MinMaxScaler(feature_range=(0, 1))
scaler_contagem_praga.fit(dados_pragas_tempo['Contagem_Praga'].
    values.reshape(-1, 1))
```

Saída Esperada

```
Shape de X_sequencias: (970, 30, 4)
Shape de y_target: (970,)
Shape de X_treino: (776, 30, 4), Shape de y_treino: (776,)
Shape de X_teste: (194, 30, 4), Shape de y_teste: (194,)
```

O que está acontecendo até agora?

- **Normalização**: Todas as features (incluindo a Contagem_Praga passada) são escaladas para o intervalo [0, 1] usando `MinMaxScaler`.
- **Criação de Sequências**: Os dados são transformados em sequências. Para cada instante, `X_sequencias` armazena os dados dos `janela_temporal` (30) dias anteriores (todas as 4 features), e `y_target` armazena a Contagem_Praga normalizada do dia seguinte, que queremos prever.
- **Divisão Cronológica**: Os dados sequenciados são divididos em 80% para treino e 20% para teste, mantendo a ordem temporal.
- **Scaler para Desnormalização**: Um `MinMaxScaler` separado é treinado apenas na coluna original Contagem_Praga. Ele será usado posteriormente para converter as previsões normalizadas de volta à escala original.

4.2.3 Definição da Arquitetura da Rede Neural Recorrente (LSTM)

Construímos o modelo LSTM usando Keras, com uma camada LSTM seguida por uma camada Densa para a saída de regressão.

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

modelo_pragas_lstm = Sequential()
modelo_pragas_lstm.add(LSTM(units=50, activation='tanh',
input_shape=(X_treino.shape[1], X_treino.shape[2])))
# modelo_pragas_lstm.add(Dropout(0.2)) # Opcional
modelo_pragas_lstm.add(Dense(units=1)) # Saída linear para
    regressão
```

```
modelo_pragas_lstm.compile(optimizer='adam', loss='
    mean_squared_error')

print("\n--- Resumo do Modelo LSTM para Previsão de Pragas ---")
modelo_pragas_lstm.summary()
```

Saída Esperada

```
--- Resumo do Modelo LSTM para Previsão de Pragas ---
Model: "sequential_X"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_Y (LSTM)                (None, 50)                11000
dense_Z (Dense)              (None, 1)                 51
=================================================================
Total params: 11,051
Trainable params: 11,051
Non-trainable params: 0
```

Fique Alerta!

Os nomes das camadas (lstm_Y, etc.) e o número do modelo ("sequential_X") podem variar. O número de parâmetros é o que importa para verificar a estrutura.

O que está acontecendo até agora?

- **Modelo Sequencial com LSTM**: É criada uma arquitetura sequencial. A primeira camada é uma `LSTM` com 50 unidades (neurônios) e ativação `tanh`. O `input_shape` é definido pela forma das sequências de treino (janela_temporal, num_features).
- **Camada de Saída Densa**: Uma camada `Dense` com 1 neurônio é usada para a saída, pois estamos prevendo um único valor (a contagem da praga). Nenhuma função de ativação é especificada, o que implica uma ativação linear, adequada para regressão.
- **Compilação**: O modelo é compilado com o otimizador `adam` e a função de perda `mean_squared_error` (MSE), comum para tarefas de regressão.
- **Resumo do Modelo**: `summary()` exibe a estrutura da rede e o número de parâmetros.

4.2.4 Treinamento do Modelo LSTM

Alimentamos o modelo com os dados de treinamento para que ele aprenda os padrões temporais.

Copie e Teste!

```
print("\n--- Iniciando o Treinamento do Modelo LSTM ---")
num_epocas = 50
tamanho_lote = 32

historico_treinamento_pragas = modelo_pragas_lstm.fit(X_treino,
   y_treino,epochs=num_epocas,batch_size=tamanho_lote,
   validation_data=(X_teste, y_teste),
verbose=1)
print("\n--- Treinamento Concluído ---")
print(historico_treinamento_pragas.history.keys()) # Para ver as
   chaves disponíveis
```

Saída Esperada

```
--- Iniciando o Treinamento do Modelo LSTM ---
Epoch 1/50
25/25 ----- 3s 30ms/step - loss: 0.0297 - val_loss: 6.5556e-04

[...] #Irá até Epoch 50/50

--- Treinamento Concluído ---
dict_keys(['loss', 'val_loss'])
```

Fique Alerta!

A saída do treinamento mostrará a perda (loss) para cada época nos dados de treino e validação. Observe como esses valores diminuem ao longo do tempo.

O que está acontecendo até agora?

- **Método `fit()`**: Inicia o processo de treinamento.
- **Parâmetros de Treinamento**: O modelo é treinado por 50 épocas, com um tamanho de lote de 32.
- **Dados de Validação**: `validation_data=(X_teste, y_teste)` é usado para monitorar o desempenho do modelo em dados não vistos ao final de cada época, ajudando a identificar overfitting.
- **Objeto `history`**: Armazena o histórico da função de perda (loss para treino, val_loss para validação) durante o treinamento.

4.2.5 Avaliação do Desempenho do Modelo LSTM e Visualização

Avaliamos o modelo treinado no conjunto de teste e visualizamos os resultados.

Copie e Teste!

```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error
# numpy já foi importado

# 1. Avaliar no conjunto de teste (MSE normalizado)
perda_teste_normalizada = modelo_pragas_lstm.evaluate(X_teste,
    y_teste, verbose=0)
print(f"Perda (MSE) no teste (normalizada): {
    perda_teste_normalizada:.4f}")

# 2. Fazer previsões (normalizadas)
y_pred_normalizado = modelo_pragas_lstm.predict(X_teste)

# 3. Desnormalizar previsões e y_teste
y_teste_reshape = y_teste.reshape(-1, 1)
y_teste_desnormalizado = scaler_contagem_praga.inverse_transform(
    y_teste_reshape)

if y_pred_normalizado.ndim == 1:
    y_pred_normalizado_reshape = y_pred_normalizado.reshape(-1, 1)
else:
    y_pred_normalizado_reshape = y_pred_normalizado

y_pred_desnormalizado = scaler_contagem_praga.inverse_transform(
    y_pred_normalizado_reshape)

mae_desnormalizado = mean_absolute_error(y_teste_desnormalizado,
    y_pred_desnormalizado)
print(f"MAE no teste (desnormalizado): {mae_desnormalizado:.2f} (
    unidades da praga)")

# 4. Visualizar histórico de Perda
plt.figure(figsize=(10, 5))
plt.plot(historico_treinamento_pragas.history['loss'], label='
    Perda (Treino)')
plt.plot(historico_treinamento_pragas.history['val_loss'], label='
    Perda (Validação)')
plt.title('Histórico de Perda do Modelo LSTM (Pragas)')
plt.xlabel('Época'); plt.ylabel('Erro Quadrático Médio (MSE)')
plt.legend(); plt.grid(True)
plt.show()

# 5. Visualizar previsões vs. reais (desnormalizados)
plt.figure(figsize=(14, 6))
num_pontos_plot = min(200, len(y_teste_desnormalizado))
plt.plot(y_teste_desnormalizado[:num_pontos_plot], label='Contagem
    Real da Praga', marker='.')
```

```
plt.plot(y_pred_desnormalizado[:num_pontos_plot], label='Previsão
    (LSTM)', marker='x', alpha=0.7)
plt.title(f'Previsão da Contagem de Pragas (Teste) - MAE: {
    mae_desnormalizado:.2f}')
plt.xlabel(f'Amostras Temporais (Primeiros {num_pontos_plot}
    pontos do teste)')
plt.ylabel('Contagem da Praga (Desnormalizada)')
plt.legend(); plt.grid(True); plt.tight_layout()
plt.show()

# Comparação em tabela (primeiras 10 amostras)
print("\n--- Comparação Detalhada (Primeiras 10 Amostras do Teste)
    ---")
comparacao_df = pd.DataFrame({'Real (Desnormalizado)':
    y_teste_desnormalizado[:10].flatten(),'Previsto (Desnormalizado
    )': y_pred_desnormalizado[:10].flatten()})
print(comparacao_df.round(2))
```

Saída Esperada

```
Perda (MSE) no teste (normalizada): 0.0001
MAE no teste (desnormalizado): 1.67 (unidades da praga)

--- Comparação Detalhada (Primeiras 10 Amostras do Teste) ---
    Real (Desnormalizado)     Previsto (Desnormalizado)
0                    0.36                         3.49
1                    0.36                         3.27
2                    0.42                         3.02
3                    0.44                         2.93
4                    0.52                         2.83
5                    0.56                         2.83
6                    0.57                         2.94
7                    0.59                         2.39
8                    0.65                         2.34
9                    0.69                         2.60
```

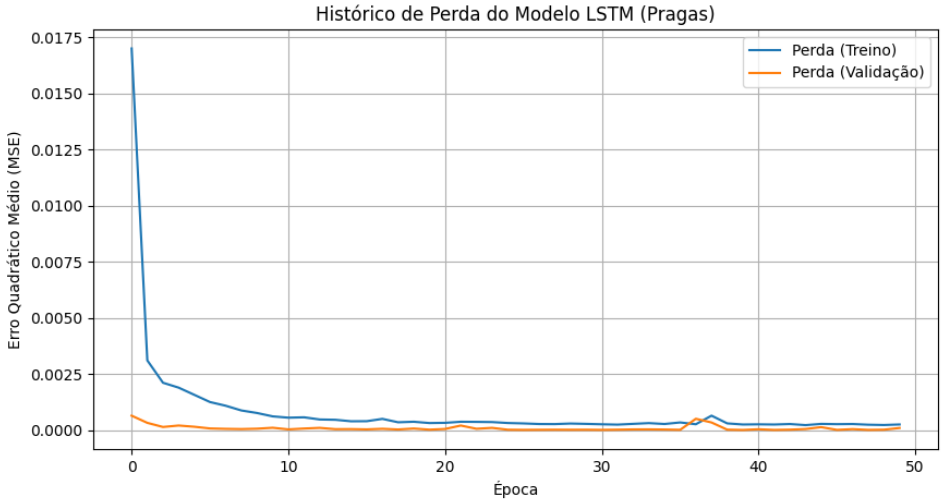

Figura 4.3: Histórico da função de perda (MSE) durante o treinamento do modelo LSTM para previsão de pragas. Comparação entre dados de treino e validação.

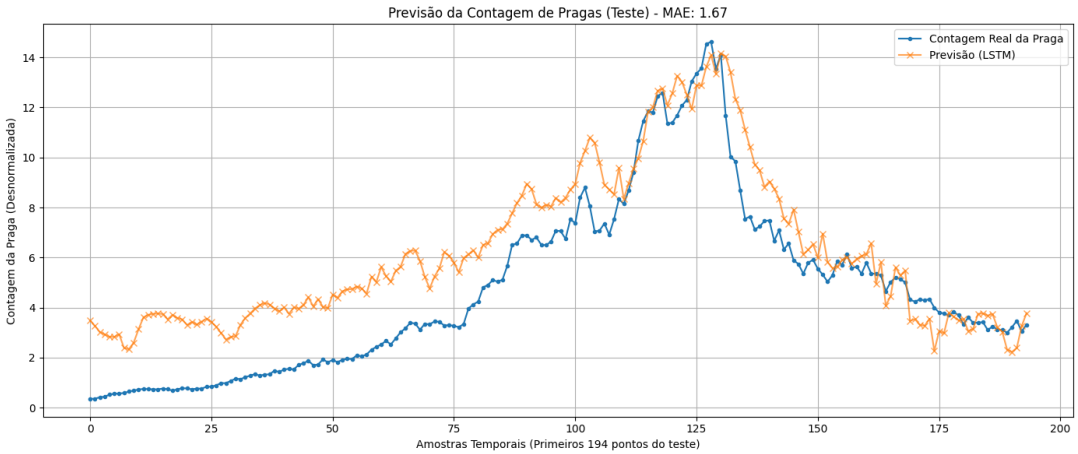

Figura 4.4: Comparação entre a contagem real da praga (desnormalizada) e as previsões do modelo LSTM (desnormalizadas) para o conjunto de teste.

O que está acontecendo até agora?

- **Avaliação Final (MSE Normalizado)**: `evaluate()` calcula o MSE no conjunto de teste com dados normalizados.
- **Previsões e Desnormalização**: O modelo faz previsões (predict()), que são então desnormalizadas, juntamente com os valores reais de `y_teste`, usando o scaler_contagem_praga salvo anteriormente. Isso permite interpretar os erros na escala original da contagem de pragas.
- **Cálculo do MAE Desnormalizado**: O Erro Absoluto Médio é calculado sobre os dados desnormalizados, oferecendo uma medida de erro mais intuitiva.
- **Visualização do Histórico de Perda**: O gráfico da perda de treino vs. validação ao longo das épocas ajuda a diagnosticar o aprendizado e overfitting.

- **Visualização das Previsões vs. Reais**: Um gráfico compara as contagens reais e previstas (ambas desnormalizadas) para o conjunto de teste, essencial para avaliar modelos de séries temporais. Uma tabela também mostra os primeiros 10 valores para comparação direta.

4.2.6 Interpretação dos Resultados e Discussão Prática

Resumo do Experimento: Neste experimento, desenvolvemos um modelo LSTM para prever a população de uma praga agrícola simulada, usando dados históricos de clima e da própria praga. Avaliamos o modelo por meio de métricas como MSE e MAE, e visualizações gráficas.

Interpretando os Resultados: Um bom resultado seria caracterizado por um MAE baixo (na escala original da praga), indicando que as previsões estão próximas dos valores reais. Os gráficos de histórico de perda idealmente mostrariam convergência com pouca diferença entre treino e validação. O gráfico de previsão vs. real deveria mostrar que o modelo captura as tendências principais da população da praga.

Relevância para o Contexto Amazônico e Aplicações Práticas no Controle de Pragas: A previsão de surtos de pragas é vital para a agricultura sustentável na Amazônia. Modelos preditivos podem:

- Otimizar o Manejo Integrado de Pragas (MIP), permitindo intervenções pontuais.
- Reduzir o uso de pesticidas, protegendo a biodiversidade e a saúde.
- Aumentar a resiliência das plantações e a segurança alimentar.
- Apoiar sistemas de alerta precoce para agricultores e cooperativas.

Limitações e Próximos Passos:

- **Dados Simulados**: A dinâmica real de pragas é muito mais complexa. A coleta de dados locais de alta qualidade é um desafio crucial.
- **Validação em Campo**: Modelos precisam de validação rigorosa em condições reais com especialistas.
- **Aprimoramentos**: Explorar mais features (estágio da cultura, dados de satélite), arquiteturas RNN mais complexas (Stacked LSTM, Bi-LSTM, Attention) e otimização de hiperparâmetros.

Fique Alerta!

Prever sistemas biológicos é inerentemente complexo. Modelos de Deep Learning são ferramentas de apoio e devem ser integrados com conhecimento agronômico e entomológico, adaptados às especificidades locais da Amazônia.

Conhecendo um pouco mais!

Pesquise sobre ”Manejo Integrado de Pragas”e ”Agricultura de Precisão”na Amazônia. As arquiteturas LSTM e GRU são fundamentais para capturar dependências de longo prazo em dados sequenciais, como explorado no Capítulo 3.

4.3 Aplicação 03: Diagnóstico de Doenças em Plantas com Redes Neurais Convolucionais

Caso Prático

Neste terceiro experimento prático, vamos focar em uma das aplicações mais impactantes da visão computacional na agricultura: o diagnóstico de doenças em plantas. Utilizaremos Redes Neurais Convolucionais (CNNs) para analisar imagens de folhas e determinar se estão saudáveis ou acometidas por alguma doença.
Iremos desenvolver, treinar e avaliar uma CNN para classificar imagens de folhas de plantas como "Saudável"ou "Doente", utilizando o dataset *PlantVillage* [4]. Além disso, exploraremos como testar o modelo treinado utilizando uma webcam para classificações em tempo real.

O PlantVillage é uma base gratuita que pode ser baixada do site oficial: `https://www.kaggle.com/datasets/emmarex/plantdisease`. Após o download, o arquivo plantdisease.zip deve ser extraído. A pasta principal conterá várias subpastas, uma para cada tipo de folha (ex: `Tomato_healthy`, `Tomato_Late_blight`, etc.). Nosso objetivo será criar um classificador binário: identificar se uma folha está saudável ou apresenta qualquer tipo de doença. Utilizaremos o Keras para carregar as imagens com o `ImageDataGenerator`, treinar a rede e avaliar o desempenho.

4.3.1 Organização dos dados

Você deve organizar as imagens da seguinte forma...

```
dados/
├── treino/
│   ├── saudavel/
│   └── doente/
├── teste/
│   ├── saudavel/
│   └── doente/
```

Mas como fazer isso?
Copie manualmente as imagens das pastas com nomes como: Tomato_healthy, Potato_healthy e Pepper_bell_healthy para a pasta `saudavel`. Todas as outras pastas representam doenças (ex: Tomato_Early_blight) e devem ir para a pasta `doente`. Em seguida, divida aproximadamente 80% das imagens de cada classe para a pasta `treino/` e os 20% restantes para a pasta `teste/`.

Ou... execute o código a seguir para automatizar essa tarefa de organização das imagens entre teste e treino!. O código abaixo:

- Lê todas as pastas do dataset;
- Separa automaticamente entre folhas **saudáveis** e **doentes**;
- Divide em conjuntos de **treino (80%)** e **teste (20%)**;

- E cria a estrutura de pastas compatível com o Keras.

Fique Alerta!

Antes de rodar este código, certifique-se de que a pasta `PlantVillage` já foi descompactada corretamente no seu ambiente de trabalho.

Conhecendo um pouco mais!

Você pode adaptar este script para dividir em outras proporções ou usar categorias específicas.

Copie e Teste!

```
import os
import shutil
import random

# Caminho onde as imagens foram extraídas
caminho_origem = 'PlantVillage'  # pasta onde o zip foi extraído
caminho_destino = 'dados_folhas' # Nova pasta para esta aplicação,
    para não conflitar

# Pastas de destino organizadas por classe
pastas_destino_app3 = [
    os.path.join(caminho_destino, 'treino', 'saudavel'),
    os.path.join(caminho_destino, 'treino', 'doente'),
    os.path.join(caminho_destino, 'teste', 'saudavel'),
    os.path.join(caminho_destino, 'teste', 'doente')
]

# Criar as pastas
for pasta in pastas_destino_app3:
    os.makedirs(pasta, exist_ok=True)

# Definir quais pastas indicam folhas saudáveis
pastas_saudaveis_app3 = [
    'Pepper__bell___healthy',
    'Potato___healthy',
    'Tomato_healthy'
]

# Todas as pastas da base, é importante garantir que
   caminho_origem exista e contenha as subpastas do dataset
if not os.path.isdir(caminho_origem):
    print(f"Erro: Diretório de origem '{caminho_origem}' não
   encontrado. Faça o download e extraia o PlantVillage dataset.")
else:
    todas_as_pastas_app3 = os.listdir(caminho_origem)
    # Filtrar apenas diretórios válidos que contêm imagens
    pastas_validas_app3 = [p for p in todas_as_pastas_app3 if os.
```

```
path.isdir(os.path.join(caminho_origem, p))]

 # Separar pastas doentes
 pastas_doentes_app3 = [p for p in pastas_validas_app3 if p not
 in pastas_saudaveis_app3]

 # Função para copiar imagens
 def copiar_imagens_app3(lista_pastas, classe_destino_param,
caminho_origem_param, caminho_destino_param):
     for pasta_origem_relativa in lista_pastas:
         pasta_completa_origem = os.path.join(
caminho_origem_param, pasta_origem_relativa)
         if not os.path.isdir(pasta_completa_origem):
             print(f"Aviso: Subpasta '{pasta_completa_origem}'
não encontrada ou não é um diretório.")
             continue

         imagens = [img for img in os.listdir(
pasta_completa_origem) if img.lower().endswith(('.png', '.jpg',
 '.jpeg'))]
         random.shuffle(imagens)
         qtd_treino = int(0.8 * len(imagens))

         for i, imagem_nome in enumerate(imagens):
             origem_img = os.path.join(pasta_completa_origem,
imagem_nome)
             tipo_conjunto = 'treino' if i < qtd_treino else '
teste'
             destino_final_img = os.path.join(
caminho_destino_param, tipo_conjunto, classe_destino_param,
imagem_nome)
             # Certificar que a pasta de destino da classe
exista
             os.makedirs(os.path.dirname(destino_final_img),
exist_ok=True)
             shutil.copy2(origem_img, destino_final_img)

 # Copiar as imagens saudáveis e doentes se as pastas válidas
foram encontradas
 if 'pastas_validas_app3' in locals() and pastas_validas_app3:
     copiar_imagens_app3(pastas_saudaveis_app3, 'saudavel',
caminho_origem, caminho_destino)
     copiar_imagens_app3(pastas_doentes_app3, 'doente',
caminho_origem, caminho_destino)
     print("Organização das imagens para Aplicação 03 concluída
 com sucesso!")
 elif not os.path.isdir(caminho_origem):
     print("A organização não pôde ser concluída pois o
diretório de origem não foi encontrado.")
 else:
```

```
        print("Aviso: Nenhuma pasta válida encontrada para
    processar em 'PlantVillage'. Verifique o conteúdo.")
```

4.3.2 Carregar e preparar os dados (Imagens)

Com as pastas organizadas, usamos o `ImageDataGenerator` para carregar as imagens.

Copie e Teste!

```
from tensorflow.keras.preprocessing.image import
    ImageDataGenerator

# Gerador com normalização (rescale) e data augmentation (opcional
    , mas bom para robustez)
gerador_treino = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20, # Rotaciona a imagem aleatoriamente
    width_shift_range=0.1, # Desloca horizontalmente
    height_shift_range=0.1, # Desloca verticalmente
    shear_range=0.1, # Aplica cisalhamento
    zoom_range=0.1, # Zoom aleatório
    horizontal_flip=True, # Inverte horizontalmente
    fill_mode='nearest'
)

gerador_teste = ImageDataGenerator(rescale=1./255) # Para teste,
    apenas normalização

# Caminho para as pastas
# Usando a nova pasta definida no script de organização
caminho_dados_app3 = 'dados_folhas/'

IMG_HEIGHT, IMG_WIDTH = 128, 128
BATCH_SIZE = 32

# Carregar imagens para treino e teste
try:
    treino_app3 = gerador_treino.flow_from_directory(
        os.path.join(caminho_dados_app3, 'treino'),
        target_size=(IMG_HEIGHT, IMG_WIDTH),
        color_mode='rgb',
        batch_size=BATCH_SIZE,
        class_mode='binary' # Saudável vs Doente
    )

    teste_app3 = gerador_teste.flow_from_directory(
        os.path.join(caminho_dados_app3, 'teste'),
        target_size=(IMG_HEIGHT, IMG_WIDTH),
```

```
        color_mode='rgb',
        batch_size=BATCH_SIZE,
        class_mode='binary',
        shuffle=False # Importante para avaliação e matriz de
    confusão
    )
    print("Geradores de dados criados. Verifique as classes
    encontradas:")
    print(f"Índices das classes (treino): {treino_app3.
    class_indices}")
except FileNotFoundError:
    print(f"Erro: Diretório '{caminho_dados_app3}' ou suas
    subpastas 'treino'/'teste' não foram encontradas.")
    print("Certifique-se de que o script de organização de dados
    foi executado corretamente e criou as pastas.")
```

Saída Esperada

```
Found 16504 images belonging to 2 classes. Found 4134
images belonging to 2 classes. Geradores de dados criados.
Verifique as classes encontradas: Índices das classes
(treino): 'doente': 0, 'saudavel': 1
```

O que está acontecendo até agora?

- **Organização dos Dados (Script Python):** O script fornecido automatiza a criação de uma estrutura de pastas (dados_folhas/treino/saudavel, dados_folhas/treino/doente, etc.) a partir do dataset PlantVillage original, separando as imagens em saudáveis ou doentes e dividindo-as em conjuntos de treino (80%) e teste (20%). É importante executar este script primeiro.
- **ImageDataGenerator:** `ImageDataGenerator` do Keras é usado para carregar imagens diretamente dos diretórios.
  - `rescale=1./255`: Normaliza os valores dos pixels das imagens para o intervalo [0, 1].
  - **Data Augmentation (para treino):** Parâmetros como `rotation_range`, `width_shift_range`, etc., são aplicados apenas ao gerador de treino. Eles criam versões modificadas das imagens de treino em tempo real (rotações, zooms, etc.), o que ajuda o modelo a generalizar melhor e a não memorizar as imagens exatas de treino, combatendo o overfitting [2, 1].
  - `target_size=(128, 128)`: Redimensiona todas as imagens para 128x128 pixels.
  - `batch_size=32`: As imagens serão carregadas em lotes de 32.
  - `class_mode='binary'`: Indica que temos uma classificação binária (saudável ou doente). Os rótulos serão 0 ou 1.

  - `shuffle=False` (para `teste_app3`): Garante que as imagens de teste sejam processadas em ordem, o que é importante para uma avaliação consistente e para a matriz de confusão.

- **Verificação das Classes:** É crucial verificar `treino_app3.class_indices` para saber qual rótulo numérico (0 ou 1) foi atribuído a 'saudavel' e qual a 'doente', pois isso afetará a interpretação das previsões.

4.3.3 Criar e Treinar a CNN

Definimos uma arquitetura CNN simples e a treinamos com os dados preparados.

Copie e Teste!

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense, Dropout

modelo_folhas = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(IMG_HEIGHT,
   IMG_WIDTH, 3)),
    MaxPooling2D(pool_size=(2,2)),

    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),

    Conv2D(128, (3,3), activation='relu'), # Camada convolucional
   adicional
    MaxPooling2D(pool_size=(2,2)),

    Flatten(),
    Dropout(0.5), # Dropout mais significativo para regularização
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')  # Saída binária (1 neurônio
   com sigmoide)
])

modelo_folhas.compile(optimizer='adam', loss='binary_crossentropy'
   , metrics=['accuracy'])

# Exibir resumo do modelo
modelo_folhas.summary()

# Treinar o modelo
# Certifique-se de que 'treino_app3' e 'teste_app3' foram
   carregados corretamente
if 'treino_app3' in locals() and 'teste_app3' in locals():
    historico_folhas = modelo_folhas.fit(
        treino_app3,
```

```
        epochs=15, # Pode aumentar para mais épocas se tiver tempo
    /recursos
        validation_data=teste_app3,
        # steps_per_epoch e validation_steps podem ser necessários
     se usar geradores infinitos
        # steps_per_epoch = treino_app3.samples // BATCH_SIZE,
        # validation_steps = teste_app3.samples // BATCH_SIZE
    )
    print("Treinamento concluído.")
else:
    print("Erro: Geradores de dados 'treino_app3' ou 'teste_app3'
    não foram inicializados.")
    print("Verifique se o Passo 1 (carregamento de dados) foi
    executado sem erros.")
```

Saída Esperada

```
Model: "sequential_X"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_A (Conv2D)            (None, 126, 126, 32)      896
max_pooling2d_B (MaxPooling2D) (None, 63, 63, 32)      0
conv2d_C (Conv2D)            (None, 61, 61, 64)        18496
max_pooling2d_D (MaxPooling2D) (None, 30, 30, 64)      0
conv2d_E (Conv2D)            (None, 28, 28, 128)       73856
max_pooling2d_F (MaxPooling2D) (None, 14, 14, 128)     0
flatten_G (Flatten)          (None, 25088)             0
dropout_H (Dropout)          (None, 25088)             0
dense_I (Dense)              (None, 128)               3211392
dense_J (Dense)              (None, 1)                 129
=================================================================
Total params: 3,304,769
Trainable params: 3,304,769
Non-trainable params: 0

# A saída do treinamento (épocas) será longa, você verá a perda e
   acurácia de treino e validação para cada época.

Treinamento concluído.
```

O que está acontecendo até agora? (Passo 2)

- **Arquitetura da CNN**: É definida uma CNN sequencial com três blocos de Conv2D (com 32, 64 e 128 filtros respectivamente) e MaxPooling2D. As camadas convolucionais usam ativação relu.
- **Flatten e Camadas Densas**: Após as camadas convolucionais e de pooling, uma camada Flatten transforma os mapas de características 2D em um vetor 1D. Uma camada

Dropout(0.5) é adicionada para regularização, seguida por uma camada Dense com 128 neurônios (`relu`) e, finalmente, a camada de saída Dense com 1 neurônio e ativação `sigmoid`, adequada para classificação binária.

- **Compilação**: O modelo é compilado com o otimizador `adam`, função de perda `binary_crossentropy` (para classificação binária) e a métrica `accuracy`.
- **Treinamento**: O método `fit` é usado para treinar o modelo com os dados carregados pelos geradores `treino_app3` e `teste_app3` (para validação).

4.3.4 Visualizar os Resultados do Treinamento

Plotamos a acurácia do modelo ao longo das épocas.

Copie e Teste!

```
import matplotlib.pyplot as plt

# Verificar se 'historico_folhas' existe e tem os dados esperados
if 'historico_folhas' in locals() and hasattr(historico_folhas, '
   history'):
    if 'accuracy' in historico_folhas.history and 'val_accuracy'
   in historico_folhas.history:
        plt.figure(figsize=(10, 6))
        plt.plot(historico_folhas.history['accuracy'], label='
   Acurácia (Treino)')
        plt.plot(historico_folhas.history['val_accuracy'], label='
   Acurácia (Validação)')
        plt.title('Histórico de Acurácia do Modelo (Folhas)')
        plt.xlabel('Épocas')
        plt.ylabel('Acurácia')
        plt.legend()
        plt.grid(True)
        plt.show()

    # Opcional: Plotar a perda também
        if 'loss' in historico_folhas.history and 'val_loss' in
   historico_folhas.history:
            plt.figure(figsize=(10, 6))
            plt.plot(historico_folhas.history['loss'], label='
   Perda (Treino)')
            plt.plot(historico_folhas.history['val_loss'], label='
   Perda (Validação)')
            plt.title('Histórico de Perda do Modelo (Folhas)')
            plt.xlabel('Épocas')
            plt.ylabel('Perda (Binary Crossentropy)')
            plt.legend()
            plt.grid(True)
            plt.show()
    else:
```

```
        print("Erro: Histórico de treinamento não contém 'accuracy
    ' ou 'val_accuracy'.")
else:
    print("Erro: Objeto 'historico_folhas' não foi criado.
    Verifique o passo de treinamento.")
```

Saída Esperada

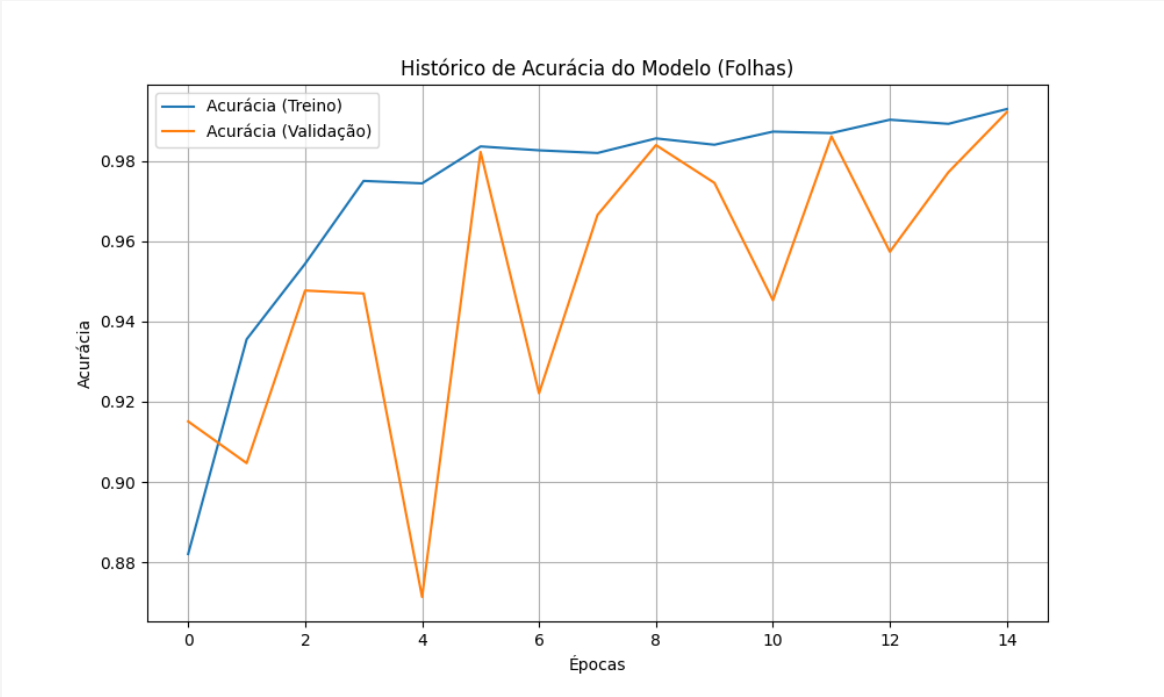

Figura 4.5: Desempenho da CNN na classificação de folhas saudáveis e doentes (Acurácia por Época).

O gráfico ilustra a evolução da acurácia do modelo. A linha azul (”Acurácia (Treino)”) demonstra um aumento rápido inicial, estabilizando-se em valores altos (acima de 0.98) a partir da época 3. A linha laranja (”Acurácia (Validação)”) apresenta maior variabilidade, com uma queda notável na época 4 (para aproximadamente 0.87) e picos subsequentes, como na época 5 (próximo a 0.98) e época 11 (quase 0.99). Ambas as curvas convergem para uma alta acurácia (superior a 0.99) ao final das 14 épocas, indicando um bom aprendizado e generalização do modelo nesse ponto.

Saída Esperada

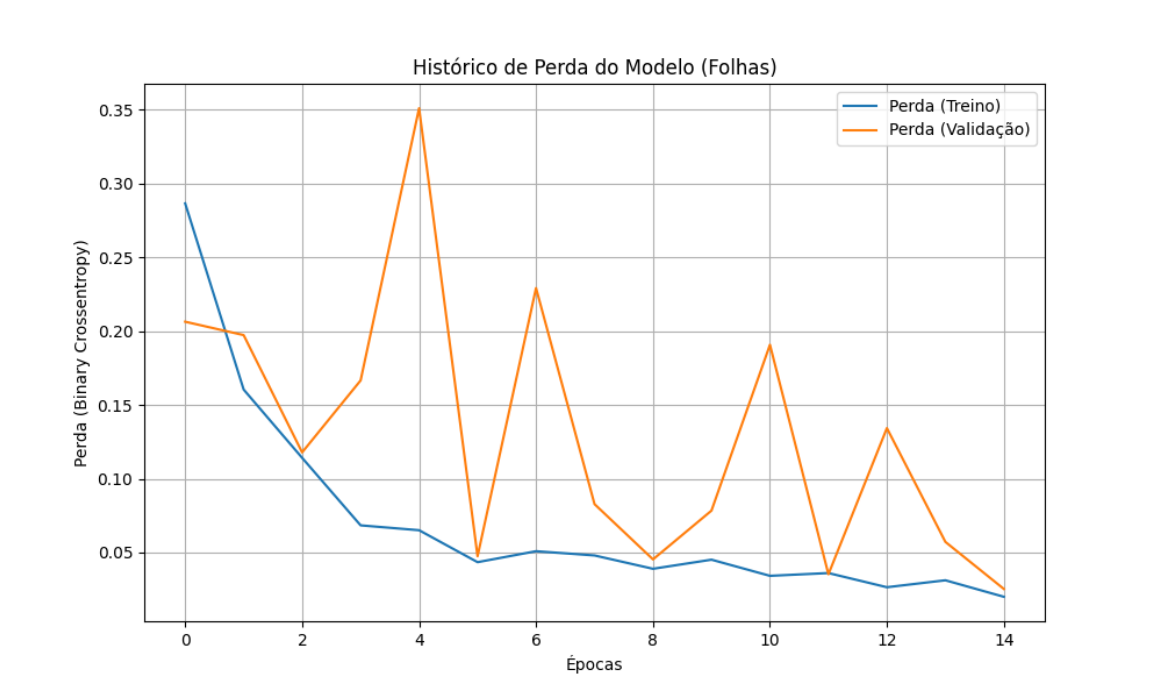

Figura 4.6: Histórico de Perda (Binary Crossentropy) do modelo CNN na classificação de folhas ao longo das épocas.

A linha azul representa a perda nos dados de treino, mostrando uma rápida redução e estabilização. A linha laranja representa a perda nos dados de validação, exibindo maior oscilação, com um pico notável na época 4, sugerindo momentos de sobreajuste (overfitting), mas com uma tendência geral de diminuição.

Fique Alerta!

Certifique-se de que o número de imagens em cada classe (saudavel, doente) seja razoavelmente equilibrado nos conjuntos de treino e teste. Se houver um grande desequilíbrio, o modelo pode ficar enviesado para a classe majoritária. O uso de *data augmentation* no gerador de treino ajuda a mitigar o overfitting.

Conhecendo um pouco mais!

Você pode expandir esse projeto para uma classificação com múltiplas doenças (multiclasse), bastando usar `class_mode='categorical'` no `ImageDataGenerator`, ajustar a última camada `Dense` para ter N neurônios (onde N é o número de classes) com ativação `softmax`, e usar `loss='categorical_crossentropy'`. A organização das pastas também precisaria refletir as múltiplas classes.

4.3.5 Testando o Modelo em Tempo Real com a Câmera

Fique Alerta!

Esta seção funciona melhor se você estiver executando este código em um ambiente Python local (como um Jupyter Notebook ou script no seu computador) que tenha acesso à sua webcam. Em ambientes como o Google Colab, o acesso direto à câmera pode exigir configurações adicionais.

Agora que treinamos nosso modelo para classificar folhas como saudáveis ou doentes, vamos testá-lo com imagens capturadas diretamente da sua câmera. Isso nos dará uma ideia de como o modelo se comporta com dados "selvagens", fora do conjunto de dados curado do PlantVillage.

Salvar o Modelo Treinado

Primeiro, precisamos salvar o modelo que acabamos de treinar para que possamos carregá-lo facilmente em um novo script ou célula de notebook para o teste com a câmera.

Copie e Teste!

```
# Assumindo que 'modelo_folhas' é o seu modelo treinado
# Verificar se o modelo foi treinado antes de salvar
if 'historico_folhas' in locals() and hasattr(historico_folhas, '
   history'):
    modelo_folhas.save('modelo_classificador_folhas.h5')
    print("Modelo salvo como modelo_classificador_folhas.h5")
else:
    print("Modelo não foi treinado ainda. Execute o Passo 2
   primeiro.")
```

Saída Esperada

```
Modelo salvo como modelo_classificador_folhas.h5
```

Código para Classificação em Tempo Real

O código a seguir irá:

- Carregar o modelo treinado.
- Acessar sua webcam.
- Capturar frames de vídeo.
- Pré-processar cada frame para que seja compatível com a entrada do modelo.
- Realizar a predição (Saudável ou Doente).
- Exibir o resultado na tela.

Fique Alerta!

Certifique-se de ter a biblioteca OpenCV instalada. Se não tiver, você pode instalá-la com: `pip install opencv-python`.

Copie e Teste!

```
import cv2
import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import img_to_array
import os # Para verificar se o arquivo do modelo existe

# Nome do arquivo do modelo
NOME_ARQUIVO_MODELO = 'modelo_classificador_folhas.h5'

# Carregar o modelo treinado
if not os.path.exists(NOME_ARQUIVO_MODELO):
    print(f"Erro: Arquivo do modelo '{NOME_ARQUIVO_MODELO}' não
   encontrado.")
    print("Certifique-se de que o modelo foi treinado e salvo
   corretamente")
    exit() # Ou trate o erro de outra forma

try:
    modelo_carregado = load_model(NOME_ARQUIVO_MODELO)
    print(f"Modelo '{NOME_ARQUIVO_MODELO}' carregado com sucesso!"
   )
except Exception as e:
    print(f"Erro ao carregar o modelo: {e}")
    exit()

# Definir as classes (IMPORTANTE: ajuste conforme treino_app3.
   class_indices)
# Exemplo, se treino_app3.class_indices foi {'doente': 0, '
   saudavel': 1}:
classes_map = {0: 'Doente', 1: 'Saudavel'}
# Verifique o print de 'treino_app3.class_indices' no Passo 1 para
    confirmar!

# Tamanho da imagem esperado pelo modelo
img_altura_cam, img_largura_cam = 128, 128

# Iniciar a captura da webcam
cap = cv2.VideoCapture(0)

if not cap.isOpened():
    print("Erro: Não foi possível abrir a câmera.")
    exit()
```

```
print("\nAponte a câmera para uma folha. Pressione 'q' para sair."
    )

while True:
    ret, frame = cap.read()
    if not ret:
        print("Erro: Não foi possível capturar o frame.")
        break

    img_processada_cam = cv2.resize(frame, (img_largura_cam,
   img_altura_cam))
    img_array_cam = img_to_array(img_processada_cam) / 255.0
    img_batch_cam = np.expand_dims(img_array_cam, axis=0)

    predicao_proba_cam = modelo_carregado.predict(img_batch_cam)
   [0][0]

    limiar_confianca = 0.5
    if predicao_proba_cam > limiar_confianca:
        id_classe_prevista = 1 # Corresponde à classe 'Saudavel'
   no nosso exemplo de classes_map
    else:
        id_classe_prevista = 0 # Corresponde à classe 'Doente'

    label_previsto = classes_map.get(id_classe_prevista, "
   Desconhecido")

    # Ajustar cálculo de confiança para refletir a classe prevista
    confianca = predicao_proba_cam if id_classe_prevista == 1 else
    1 - predicao_proba_cam
    texto_resultado = f"{label_previsto} ({confianca*100:.2f}%)"
    cor_texto = (0, 255, 0) if id_classe_prevista == 1 else (0, 0,
    255) # Verde para saudável, Vermelho para doente (BGR)
    cv2.putText(frame, texto_resultado, (10, 30), cv2.
   FONT_HERSHEY_SIMPLEX, 0.9, cor_texto, 2)

    cv2.imshow('Classificador de Folhas - Pressione Q para sair',
   frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
print("Teste com a câmera finalizado.")
```

O que está acontecendo no código de teste com a câmera?

- **Carregamento do Modelo**: O modelo `modelo_classificador_folhas.h5`, previamente treinado e salvo, é carregado.

- **Definição das Classes:** Um dicionário `classes_map` mapeia o índice da classe (0 ou 1) para o nome legível (”Doente”, ”Saudável”). **Atenção:** A correspondência correta (0 para Doente, 1 para Saudável, ou vice-versa) depende da saída de `treino_app3.class_indices`.
- **Captura da Webcam:** `cv2.VideoCapture(0)` inicia a webcam, e um loop processa cada frame.
- **Pré-processamento do Frame:** Cada frame é redimensionado, convertido para array NumPy, normalizado e suas dimensões são expandidas para o formato `(1, altura, largura, canais)`.
- **Predição:** `modelo_carregado.predict(img_batch_cam)` retorna a probabilidade da folha pertencer à classe positiva (saída da sigmoide).
- **Interpretação da Saída:** Se a probabilidade > 0.5, é classificada como classe 1 (ex: ”Saudável”); caso contrário, classe 0 (ex: ”Doente”). A confiança da classe prevista é calculada.
- **Exibição do Resultado:** O resultado da classificação e a confiança são escritos no frame de vídeo e exibidos. A cor do texto muda se for ”Saudável”(verde) ou ”Doente”(vermelho).
- **Encerramento:** Pressionar ’q’ interrompe o loop, libera a câmera e fecha as janelas.

Fique Alerta!

O desempenho do modelo com imagens da câmera pode variar significativamente. Fatores como iluminação, ângulo, qualidade da imagem, fundo e tipos de folhas não vistos no treinamento podem afetar a precisão. Este é um ótimo exemplo da diferença entre dados curados e dados do ”mundo real”!

Conhecendo um pouco mais!

Para melhorar o desempenho em tempo real, considere:
1. Treinar com um dataset mais diverso e usar *data augmentation* de forma mais intensiva.
2. Implementar detecção de objetos para isolar a folha do fundo da imagem antes da classificação.
3. Explorar Transfer Learning com modelos pré-treinados em grandes datasets de imagens.

4.3.6 Interpretação dos Resultados e Discussão Prática

Nesta aplicação, construímos e treinamos uma Rede Neural Convolucional (CNN) para classificar imagens de folhas de plantas como ”Saudável”ou ”Doente”, utilizando o dataset PlantVillage. Demonstramos também como salvar o modelo treinado e utilizá-lo para fazer classificações em tempo real com imagens de uma webcam.

Interpretando os Resultados

Uma alta acurácia no conjunto de teste do PlantVillage (verificada pelos gráficos de histórico e pela avaliação final) indica que a CNN aprendeu a distinguir características visuais relevantes para diferenciar folhas saudáveis de doentes dentro daquele conjunto de dados. A etapa de teste com a câmera, no entanto, provavelmente revelará desafios adicionais. Dificuldades em classificar corretamente folhas usando a webcam podem surgir devido a:

- **Variações de Iluminação e Fundo:** O dataset PlantVillage possui imagens com fundo relativamente controlado. Imagens da webcam terão fundos complexos e iluminação variável.
- **Novas Espécies/Doenças:** O modelo só conhece as espécies e doenças presentes no PlantVillage.
- **Qualidade da Imagem da Webcam:** Resolução, foco e artefatos da câmera podem impactar.

O objetivo do teste com a câmera é mais ilustrativo e demonstra o potencial, mas também as limitações de um modelo treinado em um dataset específico quando confrontado com a variabilidade do mundo real.

Relevância para o Contexto Amazônico

O diagnóstico rápido e preciso de doenças em plantas é crucial para a agricultura na Amazônia, onde a diversidade de cultivos e patógenos é grande. Sistemas baseados em CNNs podem:

- **Auxiliar Agricultores e Técnicos:** Fornecer uma ferramenta de triagem inicial, ajudando a identificar problemas precocemente, mesmo em locais remotos, se o modelo for embarcado em dispositivos móveis.
- **Reduzir Perdas na Produção:** Um diagnóstico rápido leva a tratamentos mais eficazes e oportunos.
- **Monitoramento Fitossanitário:** Empregar drones com câmeras para capturar imagens de grandes áreas e usar CNNs para identificar focos de doenças, permitindo um controle mais direcionado.
- **Apoiar a Pesquisa Agronômica:** Ajudar na coleta e análise de dados sobre a incidência e disseminação de doenças em diferentes condições.

Soluções como esta podem fortalecer a agricultura familiar e as cooperativas, contribuindo para a segurança alimentar e a sustentabilidade econômica da região.

Limitações e Próximos Passos

- **Generalização para o Mundo Real:** O principal desafio é a generalização. Modelos treinados em datasets específicos podem não performar bem em todas as condições de campo.
- **Necessidade de Dados Locais:** Para aplicações robustas na Amazônia, seria ideal treinar ou, pelo menos, ajustar finamente (fine-tuning) os modelos com imagens coletadas localmente, abrangendo as espécies, doenças e condições ambientais da região.
- **Identificação de Múltiplas Doenças e Deficiências Nutricionais:** Este exemplo foi binário. Modelos mais avançados podem ser treinados para identificar tipos específicos de doenças ou até mesmo deficiências nutricionais com base em padrões visuais.
- **Interpretabilidade do Modelo:** Técnicas como Grad-CAM podem ajudar a entender quais partes da imagem a CNN está usando para tomar sua decisão, aumentando a confiança no diagnóstico.

Fique Alerta!

A aplicação de IA para diagnóstico de doenças é uma ferramenta de apoio, não um substituto completo para a análise de especialistas (agrônomos, fitopatologistas). A validação em campo e a colaboração multidisciplinar são essenciais. Além disso, a qualidade e diversidade das imagens de treinamento são determinantes para o sucesso do modelo.

Este experimento com CNNs para diagnóstico de doenças em folhas demonstra o potencial da visão computacional para resolver problemas práticos na agricultura. A capacidade de estender o modelo para testes com uma câmera em tempo real abre portas para aplicações interativas e demonstrações impactantes do poder do Deep Learning.

Referências Bibliográficas

[1] CHOLLET, François. **Deep Learning with Python**. 2. ed. Shelter Island: Manning, 2021.

[2] GÉRON, Aurélien. **Aprenda Machine Learning com Scikit-Learn, Keras e TensorFlow**. 3. ed. Sebastopol: O'Reilly Media, 2023.

[3] GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. Cambridge: MIT Press, 2016. Disponível em: `https://www.deeplearningbook.org`.

[4] HUGHES, David P.; SALATHÉ, Marcel. **An open access repository of images on plant health to enable the development of mobile disease diagnostics**. [S.l.]: arXiv, 2015. Disponível em: `https://arxiv.org/pdf/1511.08060`.

[5] MARQUES, Bruno Torres; MARQUES, Leonardo Torres. **Aprendizado de máquina: uma abordagem para descoberta de conhecimento**. [S.l.]: Novas Edições Acadêmicas, 2022.

[6] MITCHELL, Tom M. **Machine Learning**. New York: WCB/McGraw-Hill, 1997.

[7] MUELLER, John Paul. **Aprendizado de máquina para leigos**. Rio de Janeiro: Alta Books, 2019.

[8] NG, Andrew. **Deep Learning Specialization**. [Curso online]. [S.l.]: DeepLearning.AI; Stanford Online, 2022. Disponível em: `https://www.coursera.org/specializations/deep-learning`.

[9] NG, Andrew. **Machine Learning Specialization**. [Curso online]. [S.l.]: DeepLearning.AI; Stanford Online, 2022. Disponível em: `https://www.coursera.org/specializations/machine-learning-introduction`.

[10] NG, Andrew. **Machine Learning Yearning: technical strategy for AI engineers, in the era of deep learning**. [S.l.]: deeplearning.ai project, 2018. Disponível em: `https://info.deeplearning.ai/machine-learning-yearning-book/`.

[11] RASCHKA, Sebastian. **Python Machine Learning: machine learning e deep learning com Python, Scikit-learn e TensorFlow 2**. São Paulo: Novatec, 2021.

[12] RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence: a modern approach**. 3. ed. Upper Saddle River: Prentice Hall, 2010.

CITHA

Capacitação e Interiorização em
Tecnologias Habilitadoras na Amazônia

ISBN: 978-65-01-59137-7
9 786501 591377
CBL