Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Extração de Características em Imagens Médicas: Análise de Desempenho de CNNs e Modelos de Machine Learning

Pedro César R. Passos

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Francisco Assis De Oliveira Nascimento

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Extração de Características em Imagens Médicas: Análise de Desempenho de CNNs e Modelos de Machine Learning

Pedro César R. Passos

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Francisco Assis De Oliveira Nascimento (Orientador)
ENE/UnB

Prof. Dr. Alexandre Ricardo Soares Romariz
ENE/UnB

Prof. Dr. Adson Ferreira da Rocha
ENE/UnB

Prof. Dr. Guilherme Novaes Ramos
Coordenador do Curso de Engenharia da Computação

Brasília, 6 de junho de 2025

Dedicatória

À minha mãe, Neurian, um dos pilares da minha vida e da minha formação, que sempre prezou pela minha educação, sendo minha maior inspiração e eterna fonte de força.

À minha avó e primeira professora, Vandice, o outro pilar da minha trajetória, que plantou a semente do meu caminho como engenheiro ao me acompanhar nas manhãs e tardes da minha infância, ensinando com carinho e paciência os primeiros passos na matemática e na vida.

Obrigado por me amarem e acreditarem nos meus sonhos. Amo vocês.

Agradecimentos

Agradeço primeiramente a Deus e ao Senhor Jesus Cristo pelo dom da vida e por estarem comigo quando mais ninguém poderia estar.

Agradeço à minha mãe, pelo amor inabalável e pela força que me inspira todos os dias.

Agradeço à minha avó, pelo amor, sabedoria e pelos ensinamentos que levo comigo.

Agradeço ao meu primo Matheus Ribeiro e à minha tia Neusiran, por me acolherem em sua casa e tornarem possível minha graduação na Universidade de Brasília.

Agradeço à minha madrinha, Maria Helenita, e meu padrinho, José Edilson, pelo apoio, amor e dedicação desde minha infância.

Agradeço à minha tia Nairan por estar presente na minha vida desde meu nascimento.

Agradeço à minha namorada, Manuela Moura, pelo apoio constante, pela paciência e por estar ao meu lado nos momentos difíceis.

Agradeço ao meu orientador, professor Dr. Francisco Assis, pelos ensinamentos e pela oportunidade de integrar seu grupo de pesquisa, o que tanto me agregou em conhecimento e experiências.

Agradeço aos amigos que fizeram parte dessa jornada, em especial à Victoria Ibarra, que por muitas vezes foi companhia de madrugadas antes de entregas de trabalhos.

Agradeço também ao povo brasileiro, por meio de cujo esforço e compromisso coletivo é possível manter a educação pública e gratuita.

Por fim, o presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A extração de características e classificação de imagens médicas é um campo de grande relevância tanto para a comunidade acadêmica quanto para o setor de saúde. Diferentemente de técnicas tradicionais de análise de imagens, como transformadas de *Fourier* ou *Gabor*, o uso de Redes Neurais Convolucionais (CNNs) tem se destacado por sua eficiência na detecção de padrões complexos, como bordas, texturas e formas. Este trabalho explora a aplicação de CNNs para a extração de características e classificação de imagens de raios-X, focando na identificação de anomalias, como pneumonia e outras patologias pulmonares. Foram utilizadas diferentes arquiteturas de CNNs, como *ResNet50, VGG19, InceptionV3 e EfficientNetV2*, cada uma com particularidades em termos de profundidade e desempenho na extração de características. Para o processo de classificação, este estudo utilizou algoritmos de Machine Learning clássicos, como *Support Vector Machine (SVM)*, modelos de Gradiente *Boosting* e árvores. O objetivo central foi comparar o desempenho dessas arquiteturas e classificadores na tarefa de detectar anomalias em imagens médicas. O processo abrange o pré-processamento das imagens, seguido da extração de características da imagem usando CNNs, e por fim a avaliação dos modelos com métricas como acurácia, F1-Score, sensibilidade e especificidade. Como parte dos resultados, foi observada uma alta precisão na identificação de padrões relevantes quando o uso de CNNs é feito em conjunto com algoritmos de classificação como SVM.

Palavras-chave: Redes Neurais Convolucionais. Imagens de Raios-X. Classificação de Patologias. Machine Learning. Extração de Características.

Abstract

The extraction of features and classification of medical images is a field of great relevance both to the academic community and the healthcare sector. Unlike traditional image analysis techniques, such as Fourier or Gabor transforms, the use of Convolutional Neural Networks (CNNs) has stood out for its efficiency in detecting complex patterns, such as edges, textures, and shapes. This study explores the application of CNNs for feature extraction and classification of X-ray images, focusing on the identification of anomalies such as pneumonia and other pulmonary pathologies. Various CNN architectures were employed, including ResNet50, VGG19, InceptionV3, and EfficientNetV2, each with particular characteristics in terms of depth and performance in feature extraction. For the classification process, this study utilized classical Machine Learning algorithms, such as Support Vector Machine (SVM), Gradient Boosting models, and decision trees. The main objective was to compare the performance of these architectures and classifiers in detecting anomalies in medical images. The process includes image preprocessing, followed by feature extraction using CNNs, and finally, model evaluation using metrics such as accuracy, F1-score, sensitivity, and specificity. As part of the results, a high accuracy in identifying relevant patterns was observed when image feature extraction was combined with classification algorithms such as SVM.

Keywords: Convolutional Neural Networks, X-ray Images, Pathology Classification, Machine Learning, Feature Extraction, Anomaly Detection

Sumário

1 Introdução **1**
1.1 Objetivos . . . 3
1.2 Estrutura do Trabalho . . . 4

2 Fundamentação teórica **6**
2.1 A Operação de Convolução e sua Importância . . . 6
2.2 Generalização, Overfitting e Underfitting . . . 10
2.2.1 Underfitting . . . 10
2.2.2 Overfitting . . . 10
2.2.3 Viés e Variância . . . 11
2.3 ResNet50 . . . 11
2.4 VGG16/VGG19 . . . 15
2.5 EfficientNetV2L . . . 17
2.6 InceptionV3 . . . 20
2.7 Métricas de Avaliação de Desempenho . . . 23
2.7.1 Acurácia . . . 23
2.7.2 Precisão (Precision) . . . 24
2.7.3 Sensibilidade / Revocação (Recall) . . . 24
2.7.4 Especificidade (Specificity) . . . 24
2.7.5 F1-Score . . . 25
2.7.6 Agregação para Múltiplas Classes . . . 25
2.7.7 Curva ROC e AUC . . . 25
2.7.8 Coeficiente de Correlação de Matthews (MCC) . . . 26
2.8 Busca de Hiperparâmetros . . . 26
2.9 Funções de Ativação . . . 28
2.9.1 Sigmoide . . . 28
2.9.2 Tanh . . . 29
2.9.3 ReLU (Rectified Linear Unit) . . . 29
2.9.4 ELU (Exponential Linear Unit) . . . 30

2.9.5 Swish e Mish . . . 30
2.9.6 Considerações sobre a Escolha da Função de Ativação . . . 31
2.10 Classificadores Utilizados . . . 31
2.10.1 Logistic Regression (Regressão Logística) . . . 32
2.10.2 K-Nearest Neighbors (KNN) . . . 35
2.10.3 Decision Tree (Árvore de Decisão) . . . 37
2.10.4 Random Forest . . . 40
2.10.5 XGBoost (Extreme Gradient Boosting) . . . 42
2.10.6 Support Vector Machine (SVM) . . . 44
2.10.7 CatBoost (Categorical Boosting) . . . 46
2.10.8 Naive Bayes . . . 48
2.10.9 AdaBoost (Adaptive Boosting) . . . 51
2.10.10 LightGBM . . . 53
2.10.11 SGD Classifier (Stochastic Gradient Descent) . . . 54

3 Trabalho Correlatos **57**
3.1 Comparative Analysis of COVID-19 X-ray Images Classification Using Convolutional Neural Networks, Transfer Learning, and Machine Learning Classifiers Using Deep Features . . . 57
3.2 Performance Evaluation of the Deep Learning Based Convolutional Neural Network Approach for the Recognition of Chest X-Ray Images . . . 59

4 Metodologia **61**
4.1 Dataset Utilizado . . . 61
4.1.1 Composição e Exemplos . . . 61
4.1.2 Análise de Desbalanceamento e Mitigação . . . 62
4.2 Pré-processamento das Imagens . . . 63
4.3 Extração de Características via CNNs . . . 63
4.3.1 Framework e Ferramentas . . . 64
4.3.2 Arquiteturas Selecionadas . . . 64
4.3.3 Processo de Extração . . . 64
4.3.4 Justificativa da Escolha da Camada para Extração de Características 65
4.4 Classificação das Características Extraídas . . . 65
4.4.1 Classificadores Avaliados . . . 65
4.4.2 Otimização de Hiperparâmetros . . . 66
4.4.3 Grades de Hiperparâmetros Exploradas . . . 67
4.5 Avaliação de Desempenho . . . 68

4.5.1 Métricas Utilizadas 68

5 Resultados **70**
5.1 Resultados e Discussão 70
5.2 Desempenho na Extração de Características 70
5.2.1 Tempo de Execução das Arquiteturas CNN 70
5.3 Desempenho dos Classificadores 71
5.3.1 Seleção dos Melhores Modelos (Conjunto de Validação) 71
5.3.2 Desempenho Final no Conjunto de Teste 72
5.3.3 Análise de Desempenho por Classe 73
5.3.4 Melhor Classificador por Arquitetura CNN (Conjunto de Teste) . . 74
5.3.5 Tempo de Execução dos Classificadores 74
5.4 Discussão Geral dos Resultados 75

6 Conclusão **77**

A Código Fonte Implementado **80**
A.1 Código Fonte do Extrator de Características 80
A.2 Código Fonte do Pipeline de Classificação 85

B Visualização de Mapas de Características das CNNs **90**
B.1 Visualizações da EfficientNetV2L 91
B.2 Visualizações da ResNet50 94
B.3 Visualizações da InceptionV3 97
B.4 Visualizações da VGG16 100
B.5 Visualizações da VGG19 103

Referências **106**

Lista de Figuras

2.1 Exemplo de uma convolução em uma imagem. O *kernel* (matriz 3×3) desliza sobre a imagem de entrada, e a soma dos produtos elemento a elemento gera o mapa de características de saída. 7

2.2 Extração de características da *CNN LeNet-5* mesmo com distorções e ruídos (LECUN et al., 1998). 9

2.3 Arquitetura da *CNN LetNet-5* (LECUN et al., 1998) 9

2.4 Ilustração do compromisso entre *underfitting*, boa generalização e *overfitting*. À esquerda, o modelo com grau 1 (linear) é simples demais para capturar a complexidade da função, caracterizando *underfitting*. No centro, o modelo de grau 4 aproxima bem a função original. À direita, o modelo de grau 15 se ajusta excessivamente aos dados, indicando *overfitting*. **Fonte:** Adaptado de *Scikit-learn documentation*. 11

2.5 Aprendizado Residual: o bloco aprende a função residual $\mathcal{F}(x)$ em relação à entrada x (He et al., 2016 [1]) 12

2.6 Ilustração de uma arquitetura ResNet com seus blocos de resíduo (adaptado de Chen et al., 2018 [2]) 14

2.7 Arquitetura *ResNet50* com blocos residuais. 15

2.8 Arquitetura básica da *VGG16* (Adaptado de Simonyan & Zisserman, 2014 [3]). . 16

2.9 Estrutura da arquitetura VGG16 usada neste trabalho. 17

2.10 Estrutura da arquitetura VGG19 usada neste trabalho. 17

2.11 Estrutura comparativa: *MBConv* (esquerda) e *Fused-MBConv* (direita). (Adaptado de Tan et al., 2021 [4]) 18

2.12 Visão geral da arquitetura *EfficientNetV2* (variante L) com blocos *MBConv/Fused-MBConv* e extração final de *features*. 20

2.13 Módulo *Inception* original (GoogLeNet/InceptionV1) com filtros paralelos (Szegedy et al., 2015 [5]) 21

2.14 Módulo *Inception* com fatorização da convolução 5x5 em duas 3x3 (Szegedy et al., 2016 [6]). 21

2.15 Visão geral da arquitetura *InceptionV3* com módulos Inception fatorizados. . . 22

2.16 Comparativo visual: Busca em Grade (esquerda) vs. Busca Aleatória (direita). Fonte: A Kernel Design Approach to Improve Kernel Subspace Identification. Karl Erza 2020. 28
2.17 A curva de Regressão Logística (Cramer et al., 2002) 32
2.18 Exemplo de classificação KNN e fronteiras de decisão (adaptado de [7]) 37

4.1 Exemplo: COVID-19. 62
4.2 Exemplo: Normal. 62
4.3 Exemplo: Opacidade. 62
4.4 Exemplo: Pneumonia Viral. 62
4.5 Distribuição aproximada de amostras por classe no dataset. 62

5.1 Tempo de execução da extração de características por modelo CNN (escala logarítmica). 71
5.2 Tempo médio de execução por classificador (treino + otimização), indicando também os valores mínimo e máximo observados (escala logarítmica). 75

B.1 Mapas de ativação da camada de expansão do bloco `block2a` da EfficientNetV2L. Cada quadrado representa a resposta de um filtro convolucional diferente aplicado à imagem de entrada. 91
B.2 Ativações geradas no bloco `block3a` da EfficientNetV2L, destacando a evolução das representações internas da rede. 92
B.3 Mapas de ativação da camada `stem_conv`, que corresponde à primeira convolução da EfficientNetV2L. Essa camada captura padrões de baixo nível, como bordas e contrastes locais. 93
B.4 Mapas de ativação da camada inicial `conv1_conv` da ResNet50. 94
B.5 Mapas de ativação do bloco `conv2_block1_out` da ResNet50. 95
B.6 Ativações do bloco intermediário `conv3_block1_out` da ResNet50, ilustrando como a complexidade dos padrões aumenta. 96
B.7 Mapas de ativação da camada `conv2d_1` da InceptionV3. 97
B.8 Ativações intermediárias da InceptionV3 na camada `conv2d_3`. 98
B.9 Mapas de ativação do módulo `mixed0`, que combina filtros convolucionais de diferentes tamanhos, característica marcante da InceptionV3. 99
B.10 Ativações da camada `block1_conv1` da VGG16. 100
B.11 Mapas de ativação da VGG16 no início do segundo bloco convolucional. . . 101
B.12 Ativações do bloco `block3_conv1`, demonstrando maior abstração nos padrões detectados. 102
B.13 Mapas de ativação da camada `block1_conv1` da VGG19. 103
B.14 Mapas de ativação da VGG19 no bloco `block2_conv1`. 104

B.15 Ativações do bloco `block3_conv1`, mostrando padrões visuais mais abstratos. 105

Lista de Tabelas

5.1 Top 5 combinações por F1-Score Ponderado no conjunto de validação (‘Val_F1‘). (Valores e ordem corrigidos conforme CSV). 72
5.2 Top 10 combinações por F1-Score Ponderado no conjunto de teste (‘Test$_F$1‘). 72
5.3 Recall (TPR - Taxa de Verdadeiros Positivos) por classe para os 5 melhores modelos da validação (com mapeamento correto). 73
5.4 Melhor F1-Score Ponderado no Teste (*Test_F1*) e classificador correspondente para cada Arquitetura Extratora. 74

Introdução

A extração de características de imagens é uma etapa crucial na construção de modelos robustos de aprendizado de máquina, especialmente no contexto de imagens médicas, como as de raios-X. Redes Neurais Convolucionais (*CNNs*) emergem como uma das principais ferramentas para essa tarefa, devido à sua habilidade de capturar padrões complexos em dados estruturados em grades, como imagens [8]. As *CNNs* funcionam aplicando operações convolucionais nas imagens de entrada, permitindo a identificação de características importantes, como formas, bordas e texturas (ver Apêndice B para detalhes). Após essa etapa, camadas de *pooling* são aplicadas para reduzir a dimensionalidade dos dados, ajudando a evitar o *overfitting* ao manter as informações mais relevantes [9]. O processo, quando usado para classificação ponta-a-ponta, é finalizado com camadas totalmente conectadas (*Fully Connected Layers*), responsáveis por classificar as imagens com base nas características extraídas. Para uma melhor compreensão visual do que as *CNNs* aprendem, o Apêndice B apresenta exemplos dos mapas de ativação extraídos de diferentes camadas das arquiteturas utilizadas.

No campo da medicina, as *CNNs* têm um papel essencial no reconhecimento e classificação de patologias em imagens de raio-X [10], auxiliando no diagnóstico de diversas condições. O processo de desenvolvimento desses modelos envolve etapas como coleta e pré-processamento das imagens, definição da arquitetura e treinamento da rede, que ajusta seus parâmetros (pesos) por meio de algoritmos de otimização, como o *Stochastic Gradient Descent* (SGD), e funções de perda apropriadas. O treinamento eficaz dessas redes geralmente requer grandes volumes de dados rotulados, e a qualidade e quantidade dos dados influenciam diretamente a capacidade de generalização do modelo. Técnicas como *data augmentation* são frequentemente empregadas para aumentar artificialmente a variabilidade dos dados de treinamento, melhorando a robustez do modelo [11].

Uma abordagem alternativa ou complementar ao uso de *CNNs* ponta-a-ponta para classificação é utilizá-las como extratoras de características (*feature extractors*). Nesse método, uma *CNN* pré-treinada em um grande conjunto de dados (como o *ImageNet*) é usada para processar a imagem médica e gerar um vetor de características de alta dimensão, que representa a imagem em um espaço semântico aprendido pela rede. Essas

características podem então ser usadas como entrada para outros modelos de aprendizado de máquina, como *Support Vector Machines* (SVM), *Random Forests*, entre outros, para realizar a classificação final [1]. Essa abordagem híbrida (*CNN* para *features* + classificador ML) tem demonstrado resultados promissores, aproveitando o poder de representação das *CNNs* profundas e a eficácia de diferentes paradigmas de classificação. Arquiteturas como a *ResNet50*, com suas conexões residuais que facilitam o treinamento de redes muito profundas [1], são frequentemente utilizadas neste contexto.

O presente trabalho adota essa abordagem híbrida e realiza uma análise comparativa da eficácia de diferentes arquiteturas *CNN* renomadas como extratoras de características de imagens de raio-X do tórax, visando a classificação entre as condições Normal, COVID-19, Opacidade Pulmonar e Pneumonia Viral. As arquiteturas *CNN* avaliadas incluem: **ResNet50** [1], conhecida por lidar bem com profundidade; **VGG16** e **VGG19** [3], reconhecidas pela simplicidade estrutural e forte desempenho como base para *transfer learning*; **InceptionV3** [6], que utiliza módulos paralelos para captura multi-escala; e **EfficientNetV2L** [4], uma arquitetura moderna focada em eficiência de parâmetros e velocidade de treinamento. Cada uma foi selecionada por suas propriedades distintas, permitindo avaliar diferentes filosofias de design de *CNNs* na qualidade das features extraídas.

Além da comparação entre as *CNNs* na extração, este trabalho investiga o impacto do classificador final no desempenho global. Para isso algoritmos de aprendizado de máquina foram aplicados sobre as características extraídas por cada *CNN*. Foram avaliados 11 classificadores distintos, abrangendo diferentes paradigmas: modelos baseados em margem como *Support Vector Machines* (SVM) [12]; modelos lineares como Regressão Logística; métodos baseados em árvores como *Decision Tree* e *Random Forest* [13]; métodos de *ensemble* baseados em *boosting* como *AdaBoost*, *XGBoost* [14], *LightGBM* e *CatBoost*; modelos probabilísticos como *Naive Bayes*; métodos baseados em vizinhança como *K-Nearest Neighbors* e outros como *SGD Classifier*. Essa análise abrangente permite identificar não apenas as melhores arquiteturas extratoras, mas também os classificadores mais adequados para processar essas características específicas.

As métricas de avaliação utilizadas neste estudo incluirão Acurácia, *F1-Score*, Precisão e *Recall* (Sensibilidade), além de ROC AUC e MCC, conforme detalhado na Fundamentação Teórica 2.7. Essas métricas fornecem uma avaliação abrangente do desempenho dos modelos, especialmente em problemas de classificação médica com classes desbalanceadas, como o presente caso [15]. O *F1-Score* Ponderado foi priorizado na otimização e comparação geral devido ao desbalanceamento, enquanto o *Recall* é crucial para avaliar a capacidade de identificar corretamente os casos positivos (ex: COVID-19, Pneumonia Viral), minimizando falsos negativos.

Este trabalho fornecerá uma análise do desempenho das diferentes arquiteturas *CNNs* na tarefa de extração de features, avaliando também o impacto do uso de uma vasta gama de classificadores na precisão final da detecção de padrões em imagens de raio-X. Espera-se que os resultados indiquem as combinações mais eficazes e eficientes, oferecendo subsídios para o desenvolvimento de sistemas de auxílio ao diagnóstico mais precisos e rápidos em aplicações clínicas.

1.1 Objetivos

O objetivo geral deste trabalho consiste em realizar uma comparação detalhada da eficácia de diferentes arquiteturas de redes neurais convolucionais (**VGG16, VGG19, ResNet50, InceptionV3, EfficientNetV2L**) na extração de características de imagens de raio-X do tórax, e analisar comparativamente o desempenho de um **amplo conjunto de algoritmos classificadores (11 modelos distintos)** aplicados sobre essas características. O propósito é avaliar como as diferentes combinações de extrator *CNN* e classificador se comportam em termos de precisão preditiva (*F1-Score*, Acurácia, etc.), eficiência computacional (tempo de extração e classificação) e capacidade de generalização na tarefa de reconhecimento de padrões médicos (Normal, COVID-19, Opacidade Pulmonar, Pneumonia Viral).

Para atingir esses objetivos gerais, foram definidos os seguintes objetivos específicos:

- Revisar as características, vantagens e desvantagens das arquiteturas *CNN* selecionadas (*VGG16, VGG19, ResNet50, InceptionV3, EfficientNetV2L*) no contexto da extração de características de imagens médicas.
- Implementar um pipeline para extrair vetores de características das imagens de raio-X utilizando cada uma das arquiteturas *CNN* pré-treinadas.
- Avaliar e comparar o custo computacional (tempo de execução) associado à extração de características para cada arquitetura *CNN*.
- Implementar e otimizar (via *Randomized Search* e validação cruzada) um conjunto abrangente de 11 algoritmos classificadores de *Machine Learning* aplicados aos diferentes conjuntos de características extraídas.
- Avaliar e comparar o desempenho preditivo (utilizando métricas como *F1-Score* Ponderado, Acurácia, Precisão, Recall, ROC AUC, MCC) e a eficiência computacional (tempo de treinamento/otimização) dos diferentes classificadores para cada conjunto de características.

- Identificar as melhores combinações {Extrator *CNN* + Classificador} em termos de desempenho preditivo e analisar o trade-off entre desempenho e custo computacional.
- Discutir a adequação das diferentes arquiteturas *CNN* como extratoras de características e a robustez dos diferentes classificadores para esta tarefa específica, considerando inclusive o desempenho nas diferentes classes patológicas.

Estes objetivos guiaram o desenvolvimento da metodologia e a análise dos resultados apresentados nos capítulos subsequentes. A

1.2 Estrutura do Trabalho

O presente trabalho está organizado em seis capítulos, conforme descrito a seguir:

- **Capítulo 1 - Introdução:** Apresenta uma visão geral sobre o tema, contextualizando a extração de características e classificação em imagens de raio-X com *CNNs* e Machine Learning. Define os objetivos e a estrutura do trabalho.
- **Capítulo 2 - Fundamentação Teórica:** Explica os conceitos e o funcionamento das principais arquiteturas *CNNs* estudadas (**VGG16, VGG19, ResNet50, InceptionV3, EfficientNetV2L**), detalha o funcionamento dos **diversos classificadores de Machine Learning** avaliados (abrangendo diferentes famílias de algoritmos), e apresenta as métricas de avaliação e conceitos de otimização de hiperparâmetros utilizados.
- **Capítulo 3 - Trabalhos Correlatos:** Apresenta uma revisão da literatura e de estudos relacionados ao uso de *CNNs* e classificadores de *Machine Learning* na análise de imagens médicas, com foco em raio-X e diagnóstico de condições pulmonares, destacando abordagens e resultados relevantes.
- **Capítulo 4 - Metodologia:** Descreve em detalhes o conjunto de dados utilizado, as etapas de pré-processamento das imagens, o processo de extração de características com cada arquitetura *CNN* (incluindo a camada utilizada), a implementação e otimização (*Randomized Search*, grades de parâmetros) dos 11 classificadores, e as métricas e procedimentos de avaliação final no conjunto de teste.
- **Capítulo 5 - Resultados e Discussão:** Apresenta os resultados quantitativos e qualitativos obtidos nos experimentos, incluindo tempos de execução, métricas de desempenho (gerais e por classe) para as diferentes combinações {Extrator + Classificador}, comparações entre as abordagens, e uma discussão aprofundada dos achados, suas implicações e limitações.

- **Capítulo 6 - Conclusão:** Sintetiza os principais resultados e conclusões do trabalho, retoma os objetivos, discute as contribuições, limitações e propõe direções para trabalhos futuros.

Fundamentação teórica

2.1 A Operação de Convolução e sua Importância

A convolução, no contexto de processamento de imagens, é uma operação matemática que aplica um filtro, também chamado de *kernel*, sobre uma imagem de entrada para gerar um mapa de características. Formalmente, dada uma imagem de entrada $I \in R^{H \times W}$ e um *kernel* $K \in R^{k \times k}$, a operação de convolução discreta bidimensional, sem preenchimento (*padding*) e com passo unitário ($stride = 1$), pode ser expressa como:

$$S(i,j) = (I * K)(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i+m, j+n) \cdot K(m,n) \tag{2.1}$$

onde $S(i,j)$ representa o valor resultante na posição (i,j) do mapa de características de saída. Para incluir deslocamentos maiores (*stride*) ou preenchimento nas bordas (*padding*), ajustes nos índices e dimensões da imagem de entrada são realizados.

Conforme ilustrado na Figura 2.1, o *kernel* é posicionado sobre uma região da imagem de entrada. Os valores do *kernel* são multiplicados elemento a elemento pelos valores dos pixels correspondentes na imagem. A soma de todos esses produtos resulta em um único valor, que constitui um pixel no mapa de características de saída. Este processo é repetido, movendo o *kernel* através da imagem com o passo definido, até que toda a imagem seja processada.

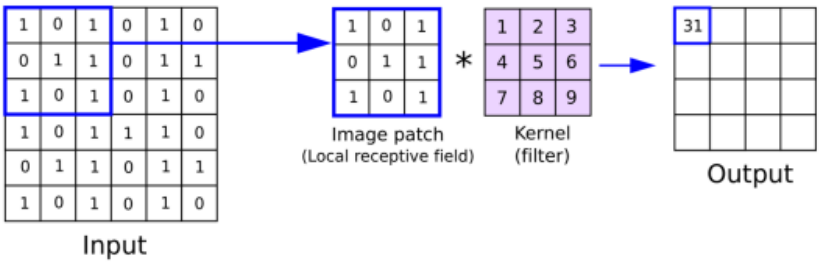


Figura 2.1: Exemplo de uma convolução em uma imagem. O *kernel* (matriz 3×3) desliza sobre a imagem de entrada, e a soma dos produtos elemento a elemento gera o mapa de características de saída.

A importância fundamental da convolução para as *CNNs* reside em três propriedades principais:

- **Extração de Características Locais:** Cada *kernel* é treinado para detectar padrões específicos, como bordas, cantos ou texturas. Ao aplicar múltiplos *kernels*, cada um com diferentes valores, a rede convolucional é capaz de gerar múltiplos mapas de características, capturando diferentes aspectos da imagem. Em níveis mais profundos, essas representações locais são combinadas, permitindo o reconhecimento de estruturas mais complexas, como partes de objetos e objetos inteiros.
- **Compartilhamento de Parâmetros:** O mesmo conjunto de pesos do *kernel* é aplicado em diferentes regiões da imagem. Isso reduz drasticamente o número total de parâmetros da rede. Por exemplo, uma camada convolucional com 10 *kernels* de 3×3 possui apenas 90 pesos treináveis, independentemente das dimensões da imagem de entrada. Em contraste, uma camada totalmente conectada exigiria um número muito maior de parâmetros.
- **Invariância à Translação:** Como o mesmo *kernel* é aplicado uniformemente em toda a imagem, uma característica detectada em uma posição também será reconhecida em outras posições espaciais. Essa propriedade é fundamental para a robustez da *CNN* diante de deslocamentos do objeto de interesse dentro da imagem.

Os pesos que compõem os *kernels* não são pré-definidos, mas aprendidos durante o processo de treinamento. O aprendizado ocorre por meio do algoritmo de retropropagação (*backpropagation*), que ajusta os pesos com base na minimização de uma função de perda, como a entropia cruzada, utilizando métodos como o gradiente descendente estocástico (*SGD*). Durante esse processo, o erro entre a predição da rede e o valor real é propagado da camada de saída para as camadas anteriores, calculando o gradiente da função de

perda em relação a cada peso da rede por meio da regra da cadeia. Em uma camada convolucional, os gradientes são calculados tanto em relação aos pesos do *kernel* quanto aos mapas de características de entrada, levando em consideração a sobreposição das regiões de aplicação do *kernel*. Formalmente, se L representa a função de perda e w um peso qualquer de um *kernel*, a atualização do peso ocorre conforme:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \tag{2.2}$$

onde η é a taxa de aprendizado. Esse processo iterativo permite que os *kernels* se ajustem para extrair automaticamente as características mais discriminantes dos dados, melhorando progressivamente o desempenho da rede na tarefa de classificação ou detecção em questão.

Por fim, a saída da convolução é frequentemente normalizada e submetida a uma função de ativação não linear, como a *ReLU (Rectified Linear Unit)*, definida por:

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

Essa função introduz não-linearidades que são essenciais para permitir que a rede modele relações complexas nos dados.

As aplicações das *CNNs* são vastas e diversificadas. Na medicina, elas são utilizadas no diagnóstico por imagem, auxiliando na detecção de doenças como câncer e anomalias cardíacas em exames como radiografias. Na agricultura, ajudam a identificar doenças em plantas, enquanto na segurança são empregadas em sistemas de vigilância para detecção automática. Na automação, facilitam tarefas como controle de qualidade e manutenção preditiva.

O processo de treinamento das *CNNs* envolve a extração de características de imagens que são representadas por uma matriz de pixels, sendo cada posição da matriz a intensidade de cada pixel. Para imagens *RGB*, por exemplo, há o padrão de três matrizes para representar a intesidade dos pixels para formar uma imagem colorida. Essas características extraídas servem como entradas para algoritmos de classificação e recuperação de imagens baseados em conteúdo (CBIR). A avaliação do desempenho é realizada com métricas como acurácia e *mean average precision* (MAP). A técnica de *data augmentation* tem sido essencial para aumentar a quantidade de imagens de treinamento em cenários de dados insuficientes [11].

Resultados de pesquisas indicam que, mesmo diante de desafios como variação de fundo e sombreamento, as *CNNs* identificam padrões e classificam com eficácia. A*LeNet* demonstrou desempenho superior na classificação de pílulas lícitas devido à maior dispo-

nibilidade de dados, enquanto a *Inception ResNet* apresentou bons resultados mesmo com bases de imagens limitadas [10].

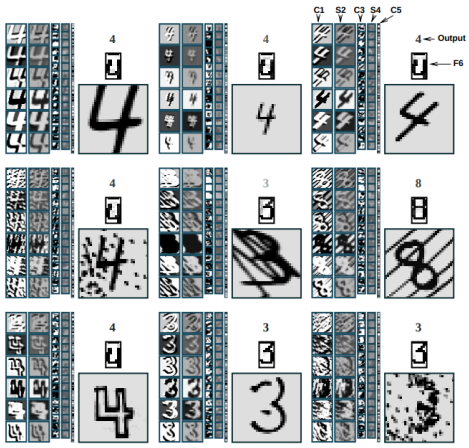


Figura 2.2: Extração de características da *CNN LeNet-5* mesmo com distorções e ruídos (LECUN et al., 1998).

As *CNNs* incluem camadas convolucionais, de *pooling* e *fully connected*. As convolucionais aplicam filtros para detectar características fundamentais, como bordas e texturas, ajustando-os durante o treinamento para padrões específicos [16]. A camada de *pooling* reduz a dimensionalidade das representações extraídas, contribuindo para otimizar o tempo de computação e evitar o *overfitting*, com métodos como *max pooling*, que preserva informações mais relevantes [8].

As *fully connected* combinam as características extraídas nas etapas anteriores e produzem a saída final, normalmente convertida em probabilidades por funções como a *softmax*. Este conjunto de operações permite que as *CNNs* executem tarefas complexas de reconhecimento e classificação de imagens com eficiência, como demonstrado na detecção de doenças em plantas [9].

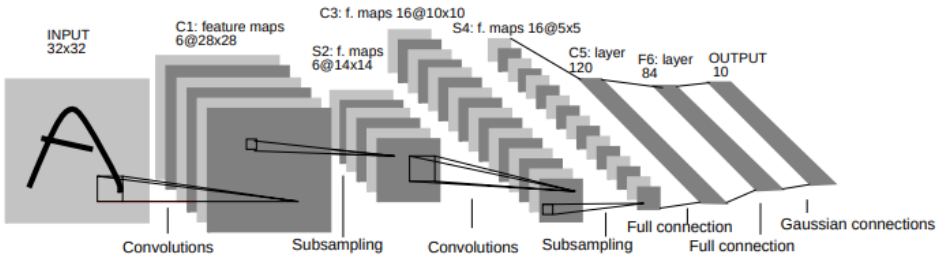


Figura 2.3: Arquitetura da *CNN LetNet-5* (LECUN et al., 1998)

2.2 Generalização, Overfitting e Underfitting

O principal objetivo de um modelo de rede neural é generalizar bem, ou seja, apresentar bom desempenho não apenas nos dados de treinamento, mas também em dados novos e não vistos. A falha nessa generalização se manifesta como *underfitting* ou *overfitting*, fenômenos que refletem, respectivamente, um modelo com baixa ou excessiva complexidade em relação ao problema.

2.2.1 Underfitting

O *underfitting* ocorre quando o modelo é incapaz de aprender os padrões dos dados, resultando em baixa acurácia tanto no conjunto de treino quanto no de validação. Em redes neurais, isso pode ser causado por arquiteturas muito simples (poucas camadas ou neurônios), treinamento insuficiente (poucas épocas) ou dados pouco informativos.

Como mitigar:

- Aumentar a complexidade do modelo;
- Treinar por mais épocas;
- Melhorar a representação dos dados de entrada.

2.2.2 Overfitting

Já o *overfitting* ocorre quando a rede aprende demasiadamente os dados de treinamento, incluindo ruídos e padrões específicos, perdendo a capacidade de generalização. Isso se reflete em alta acurácia no treino, mas desempenho ruim na validação.

Principais causas:

- Rede muito complexa para o volume de dados;
- Conjunto de treinamento pequeno;
- Treinamento prolongado demais.

Técnicas de regularização para evitar o overfitting:

- **Data augmentation:** aumenta a diversidade dos dados de entrada;
- **Dropout:** desativa aleatoriamente neurônios durante o treinamento;
- **L1/L2 regularization:** penaliza pesos excessivamente grandes;
- **Early stopping:** interrompe o treino com base no desempenho de validação;
- **Batch normalization:** ajuda na estabilidade e atua como regularizador leve.

2.2.3 Viés e Variância

O equilíbrio entre *underfitting* e *overfitting* é conhecido como o compromisso viés-variância. Modelos simples apresentam alto viés (*underfitting*), enquanto modelos complexos têm alta variância (*overfitting*). O objetivo é encontrar uma configuração que minimize o erro total, conciliando a capacidade da rede com a complexidade dos dados.

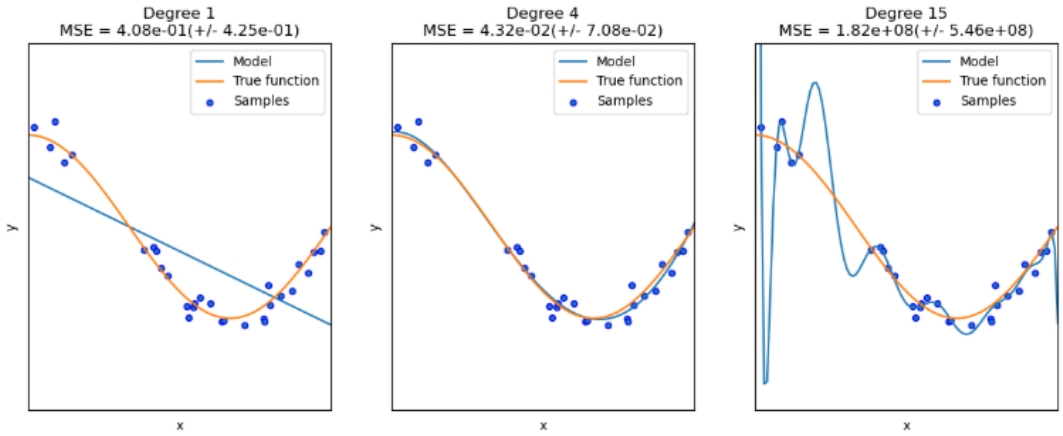


Figura 2.4: Ilustração do compromisso entre *underfitting*, boa generalização e *overfitting*. À esquerda, o modelo com grau 1 (linear) é simples demais para capturar a complexidade da função, caracterizando *underfitting*. No centro, o modelo de grau 4 aproxima bem a função original. À direita, o modelo de grau 15 se ajusta excessivamente aos dados, indicando *overfitting*. **Fonte:** Adaptado de *Scikit-learn documentation*.

Como ilustrado na Figura 2.4, modelos com complexidade inadequada resultam em erro elevado (à esquerda), enquanto modelos muito complexos tendem a memorizar o ruído (à direita), prejudicando a generalização.

2.3 ResNet50

A *ResNet (Residual Network)* é uma arquitetura de rede neural convolucional profunda que se tornou extremamente influente desde sua introdução por [1]. A variante *ResNet50*, com 50 camadas, é uma das mais populares. Seu design foi motivado pela dificuldade encontrada no treinamento de redes neurais muito profundas, especificamente o problema de **degradação**, onde adicionar mais camadas a uma rede suficientemente profunda levava a um aumento no erro de treinamento, um fenômeno não atribuível a *overfitting* [1].

A principal inovação da *ResNet* é a introdução das **conexões residuais** (*residual connections* ou *shortcut connections*), conforme ilustrado na Figura 2.5. Em vez de esperar que um conjunto de camadas aprenda diretamente uma função subjacente desejada $\mathcal{H}(x)$, a ResNet reformula o problema, fazendo com que as camadas aprendam uma **função**

residual $\mathcal{F}(x)$ em relação à entrada do bloco x, de modo que a saída do bloco seja $\mathcal{H}(x) = \mathcal{F}(x)+x$. Essa adição é realizada através de uma conexão de atalho que "pula"uma ou mais camadas e adiciona a entrada x diretamente à saída das camadas $\mathcal{F}(x)$.

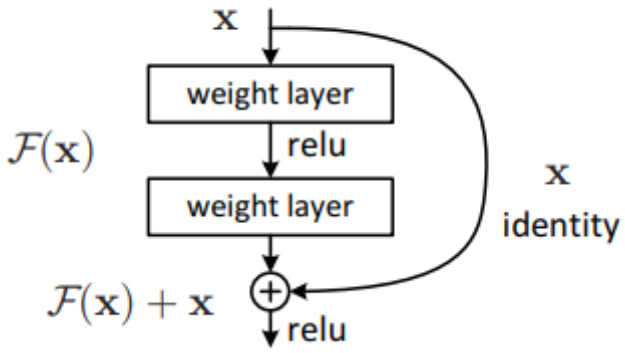


Figura 2.5: Aprendizado Residual: o bloco aprende a função residual $\mathcal{F}(x)$ em relação à entrada x (He et al., 2016 [1])

Essa abordagem de aprendizado residual facilita significativamente a **otimização** de redes profundas. Se, por exemplo, a função identidade fosse a ideal para um bloco ($\mathcal{H}(x) = x$), seria muito mais fácil para as camadas aprenderem $\mathcal{F}(x) = 0$ do que aprender a função identidade através de múltiplas transformações não lineares. Embora as conexões residuais também ajudem no fluxo de gradientes durante a retropropagação, mitigando o problema do **desaparecimento de gradiente** (um desafio conhecido em redes profundas, como discutido em [3]), seu principal benefício reside em combater o problema da degradação, permitindo que redes mais profundas sejam treinadas efetivamente e alcancem melhor desempenho [1].

A rede *ResNet50* é composta por 50 camadas ponderadas. Sua arquitetura utiliza blocos de construção residuais específicos chamados **blocos bottleneck** (Figura 2.6). Cada bloco *bottleneck* consiste em uma sequência de três camadas convolucionais: uma convolução de 1×1 para reduzir a dimensionalidade das *features*, uma convolução de 3×3 (o gargalo, onde a computação principal ocorre em menor dimensão), e uma convolução final de 1×1 para restaurar a dimensionalidade original. A conexão de atalho então adiciona a entrada do bloco à saída da terceira convolução. O uso de blocos *bottleneck* torna redes muito profundas como a *ResNet50* computacionalmente mais eficientes do que se usassem apenas blocos com camadas 3×3 (como nas *ResNets* mais rasas, 18 e 34). Crucialmente, **Batch Normalization (BN)** [17] é aplicada após cada convolução e antes da função de ativação (tipicamente $ReLU$), o que também é fundamental para estabilizar e acelerar o treinamento [1]. As conexões de atalho são geralmente identidades,

mas podem incluir uma convolução 1×1 (projeção) se for necessário ajustar as dimensões ou o número de canais para a adição [1].

A **Batch Normalization** (BN) é um componente integral da arquitetura ResNet, cuja importância vai além da simples aceleração. A técnica normaliza as saídas de uma camada para cada mini-lote de dados, ajustando suas ativações para que tenham média zero e variância unitária. Em seguida, ela aplica uma transformação afim com dois parâmetros treináveis, *gama* e *beta*, que permitem à rede aprender a escala e o deslocamento ótimos para essas ativações, preservando sua capacidade de representação. Ao fazer isso, a BN combate o problema conhecido como **Internal Covariate Shift** (a mudança na distribuição das entradas das camadas durante o treinamento), o que estabiliza o processo de aprendizado. Na *ResNet50*, o papel da BN é ainda mais crítico: ao garantir um fluxo de gradientes mais estável através de dezenas de camadas, ela é fundamental para viabilizar o treinamento de uma rede com tamanha profundidade, permitindo o uso de taxas de aprendizado mais altas e atuando como um regularizador que diminui a necessidade de outras técnicas como o **Dropout** [1, 17].

Comparada a redes como a VGG-19 [3], que tem 19.6 bilhões de *FLOPs*, a *ResNet50* é mais eficiente, com cerca de 3.8 bilhões de *FLOPs* (usando blocos *bottleneck*), apesar de ser muito mais profunda [1].

Entre as **vantagens** mais notáveis da *ResNet50* (e das *ResNets* em geral) está sua capacidade comprovada de treinar redes com profundidade sem precedentes (até 152 camadas no paper original, e mais em trabalhos subsequentes) sem sofrer com a degradação de desempenho, alcançando resultados estado da arte em diversas tarefas. [1] demonstraram ganhos claros de acurácia com o aumento da profundidade (ex: *ResNet-34* superando *ResNet-18*) e um desempenho excepcional no *ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2015*, com um erro top-5 de 3.57% (*ensemble*) no *ImageNet test set*, consolidando a arquitetura como uma base para tarefas de classificação, detecção e segmentação de objetos.

Os **desafios** associados a redes profundas como a *ResNet50* incluem a complexidade na implementação e no ajuste fino de hiperparâmetros, e a demanda significativa por recursos computacionais (GPU, memória) durante o treinamento [1].

Aplicações práticas da *ResNet50* são vastas em visão computacional, incluindo classificação de imagens, detecção de objetos, segmentação semântica e reconhecimento facial. Sua robustez e capacidade de extrair features hierárquicas complexas a tornam uma escolha popular como *backbone* em modelos mais complexos. Embora o custo computacional seja considerável, o ganho em precisão e a capacidade de lidar com a complexidade dos dados visuais justificam seu uso extensivo.

Em suma, a *ResNet50*, com sua arquitetura baseada em blocos *bottleneck* e aprendizado residual, representa um marco no desenvolvimento de redes neurais profundas. Ela superou a barreira da degradação, permitindo a exploração de redes muito mais profundas e eficientes, e estabeleceu um novo padrão de desempenho em tarefas de visão computacional [1].

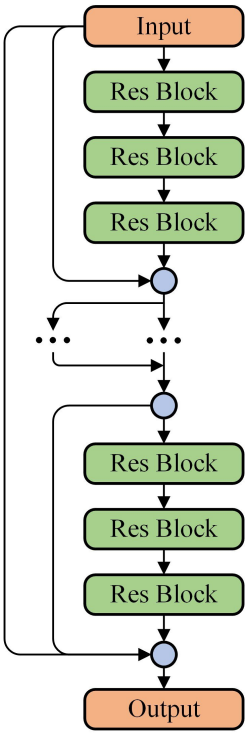


Figura 2.6: Ilustração de uma arquitetura ResNet com seus blocos de resíduo (adaptado de Chen et al., 2018 [2])

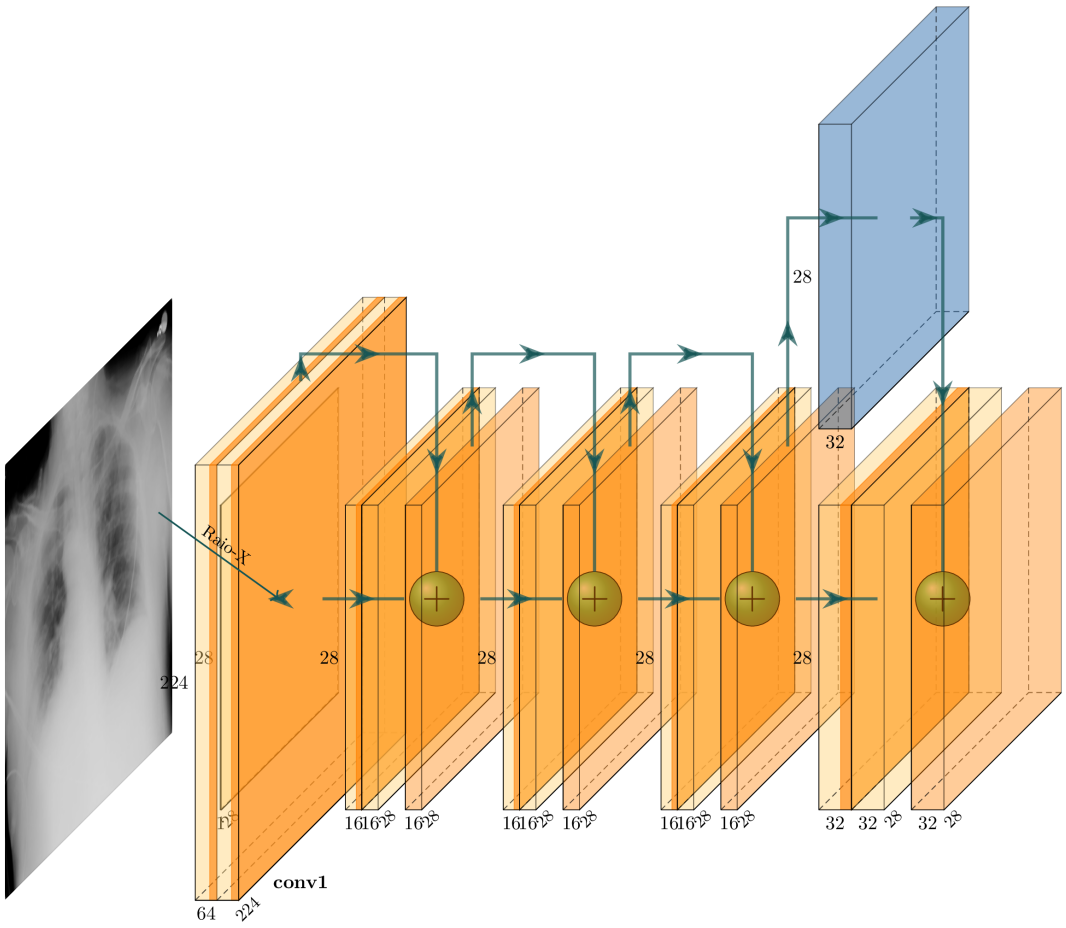


Figura 2.7: Arquitetura *ResNet50* com blocos residuais.

2.4 VGG16/VGG19

As arquiteturas *VGG16* e *VGG19*, propostas por Simonyan et al., representam um marco no estudo do impacto da profundidade em Redes Neurais Convolucionais (*CNNs*) para reconhecimento de imagens em larga escala. Com 16 e 19 camadas com pesos treináveis, respectivamente (incluindo camadas convolucionais e totalmente conectadas), essas redes se destacam por sua notável **simplicidade e homogeneidade estrutural**. A filosofia de design principal foi utilizar exclusivamente filtros convolucionais muito pequenos, de tamanho **3x3**, empilhados em profundidade, intercalados com camadas de *max pooling* (2x2 com *stride* 2) para redução da resolução espacial [3].

A organização das *VGGs* segue um padrão regular e bem definido: blocos de 2 ou mais camadas convolucionais 3x3 consecutivas (com *padding* 1 para preservar a resolução espacial dentro do bloco e função de ativação **ReLU** após cada convolução), seguidos por

uma camada de *max pooling*. Essa estrutura permite que a rede aprenda características hierárquicas das imagens, começando com bordas e texturas em camadas mais baixas e progredindo para conceitos mais abstratos e complexos, como formas e objetos, em camadas mais profundas. O uso exclusivo de filtros 3x3, em vez de filtros maiores usados em arquiteturas anteriores, permitiu aumentar a profundidade da rede (incorporando mais camadas não lineares) com um aumento mais controlado no número de parâmetros [3].

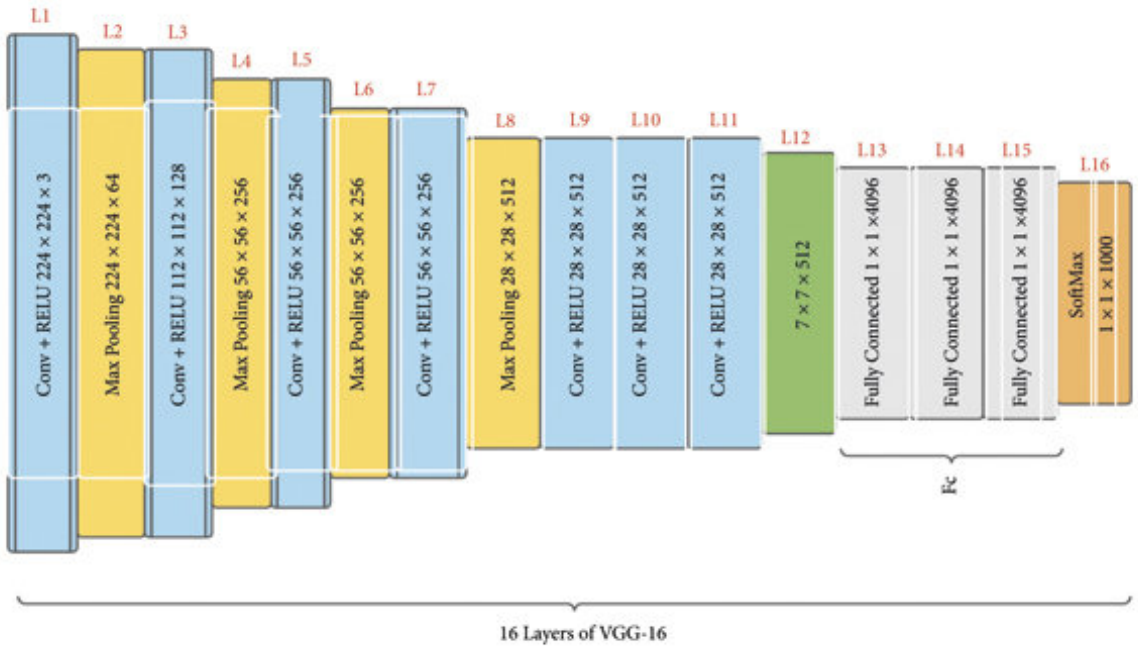


Figura 2.8: Arquitetura básica da *VGG16* (Adaptado de Simonyan & Zisserman, 2014 [3]).

As redes *VGG* alcançaram desempenho estado da arte em competições como o ILSVRC 2014, demonstrando alta precisão em tarefas de classificação [3]. Sua estrutura modular e padronizada também as tornou altamente **adaptáveis** para outras tarefas de visão computacional, como detecção de objetos e segmentação, e uma escolha extremamente popular como *backbone* para extração de características e *transfer learning* ou *fine-tuning* em novos datasets, especialmente quando os dados são limitados.

No entanto, as *VGGs* apresentam **limitações** notáveis, principalmente relacionadas ao seu **alto custo computacional**. A profundidade, combinada com as grandes camadas totalmente conectadas no final da rede original, resulta em um número muito grande de parâmetros: aproximadamente **138 milhões para a VGG16 e 144 milhões para a VGG19** [3]. Isso exige quantidades significativas de memória (especialmente de GPU) e poder computacional, além de prolongar o tempo de treinamento. Esse alto número de parâmetros também aumenta o risco de **overfitting**, exigindo estratégias de regularização

robustas (como *dropout*, aplicado nas camadas totalmente conectadas no trabalho original [3]) e grandes volumes de dados para um treinamento eficaz.

Apesar dessas limitações, as arquiteturas *VGG16* e *VGG19* foram fundamentais para demonstrar o poder da profundidade em *CNNs* e influenciaram significativamente pesquisas subsequentes. A disponibilização pública dos modelos pré-treinados contribuiu para seu uso generalizado como extratores de *features* robustos e como ponto de partida para o desenvolvimento de novas arquiteturas e aplicações em visão computacional. Mesmo com o surgimento de redes mais eficientes, as *VGGs* permanecem relevantes devido à sua simplicidade e forte desempenho como base para *transfer learning*.

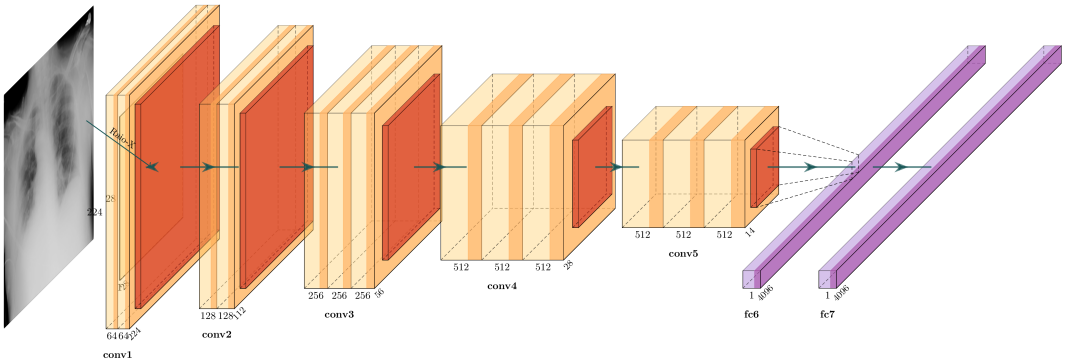


Figura 2.9: Estrutura da arquitetura VGG16 usada neste trabalho.

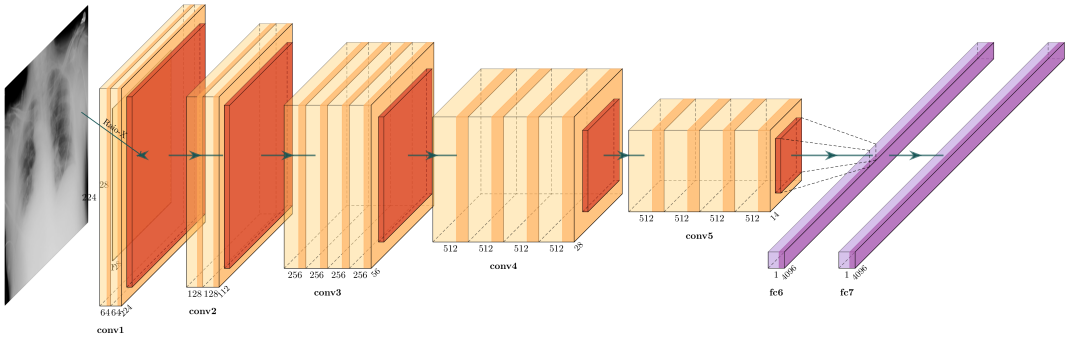


Figura 2.10: Estrutura da arquitetura VGG19 usada neste trabalho.

2.5 EfficientNetV2L

A família *EfficientNetV2*, introduzida por Tan et al., representa uma evolução das arquiteturas *EfficientNet* originais, com foco em melhorar significativamente a **eficiência do treinamento** e a **eficiência de parâmetros**. O modelo *EfficientNetV2-L* é uma das

variantes maiores e de maior desempenho desta família. Essas arquiteturas foram desenvolvidas utilizando uma combinação de *Neural Architecture Search* (NAS) consciente do treinamento (*training-aware*) e uma estratégia de escalonamento (*scaling*) aprimorada, visando otimizar conjuntamente acurácia, velocidade de treinamento e número de parâmetros [4]. O resultado são modelos que treinam consideravelmente mais rápido que redes anteriores de desempenho similar.

Uma das inovações chave introduzidas na arquitetura base do *EfficientNetV2* (e presente nas variantes como a V2-L) é o uso seletivo de blocos **Fused-MBConv** (Figura 2.11) nas camadas iniciais da rede. Diferente do bloco *MBConv* padrão (usado no *EfficientNetV1*), que emprega uma convolução *depthwise* 3x3 separada de uma convolução de expansão 1x1, o Fused-MBConv **substitui** ambas por uma única convolução 3x3 padrão. Embora isso possa aumentar ligeiramente os *FLOPs* e parâmetros localmente, melhora a utilização de hardware (GPUs/TPUs) e a velocidade de treinamento nas primeiras camadas [4]. A arquitetura base do *EfficientNetV2* foi encontrada através de *NAS*, buscando a melhor combinação de blocos (*MBConv* e *Fused-MBConv*), número de camadas, tamanho de kernel e taxa de expansão, otimizando para acurácia e velocidade de treino [4].

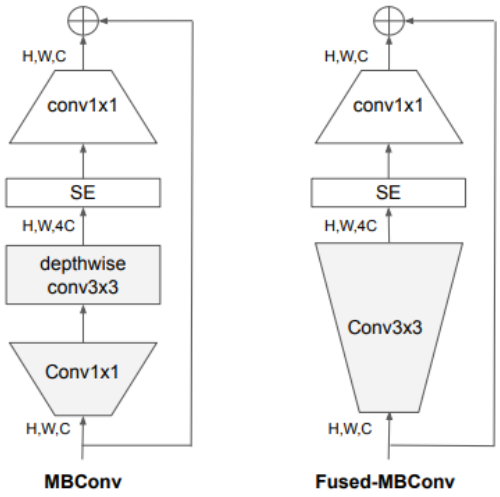


Figura 2.11: Estrutura comparativa: *MBConv* (esquerda) e *Fused-MBConv* (direita). (Adaptado de Tan et al., 2021 [4])

Entre os principais benefícios da família *EfficientNetV2*, destaca-se a expressiva **economia no tempo de treinamento**. Por exemplo, o *EfficientNetV2-M* (uma variante um pouco menor que a L) demonstrou treinar até 4.1 vezes mais rápido que o *EfficientNet-B7* (V1) alcançando acurácia similar, com 17% menos parâmetros e 37% menos *FLOPs* [4]. O *EfficientNetV2-L*, embora seja uma rede maior em parâmetros (120M) e *FLOPs* (53B) comparada ao *EfficientNet-B7* (66M/38B), treina significativamente mais rápido e atinge maior acurácia final (85.7% vs 84.7% no *ImageNet*) [4]. Parte dessa aceleração

no treinamento também se deve à técnica de **aprendizado progressivo (*progressive learning*)** proposta no mesmo trabalho: o treinamento inicia com imagens de tamanho menor e regularização mais fraca (ex: *data augmentation, dropout*), e ambos (tamanho da imagem e força da regularização) são aumentados progressivamente ao longo das épocas. Isso permite que a rede aprenda mais rapidamente nas fases iniciais sem sacrificar a acurácia final [4]. A eficiência geral torna os modelos *EfficientNetV2* atraentes para aplicações com restrições de tempo de treinamento ou recursos computacionais, incluindo visão em tempo real ou em dispositivos embarcados.

No entanto, a arquitetura *EfficientNetV2* (especialmente as variantes maiores como a L) ainda apresenta **desafios**. A complexidade inerente ao *NAS* e aos blocos otimizados pode dificultar modificações ou ajustes finos. Além disso, alcançar o desempenho ótimo, especialmente com o treinamento progressivo, exige um ajuste cuidadoso de hiperparâmetros (como a força da regularização em cada estágio), demandando experimentação e recursos computacionais [4].

A **escalabilidade** continua sendo uma característica central. A estratégia de escalonamento composto foi refinada no V2 para gerar as variantes S, M e L, ajustando profundidade, largura e resolução, mas com modificações como limitar o tamanho máximo da imagem (para evitar custos excessivos de memória) e adicionar mais camadas aos estágios finais da rede [4]. Embora mais eficientes que gerações anteriores como *VGG* ou *ResNet*, as versões maiores como a *V2-L* ainda demandam considerável poder de processamento.

Em resumo, o *EfficientNetV2L* representa um avanço em direção a redes convolucionais que não apenas alcançam alta precisão, mas também treinam significativamente mais rápido e são mais eficientes em parâmetros que modelos anteriores de capacidade comparável. Suas inovações arquiteturais (como *Fused-MBConv* e *NAS* focado em treino) e de treinamento (aprendizado progressivo) o tornam uma opção poderosa, embora seu uso eficaz possa exigir um ajuste cuidadoso.

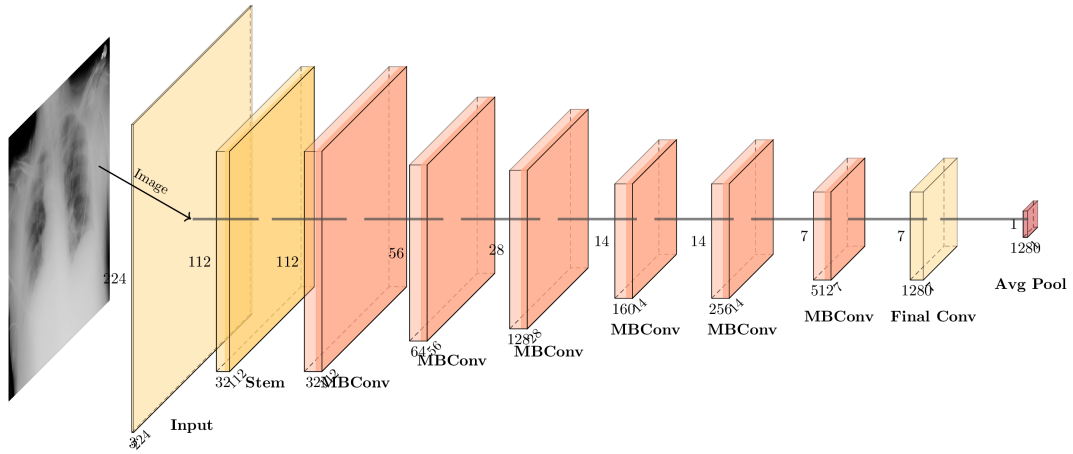


Figura 2.12: Visão geral da arquitetura *EfficientNetV2* (variante L) com blocos *MBConv/Fused-MBConv* e extração final de *features*.

2.6 InceptionV3

O *InceptionV3*, proposto por Szegedy et al., é uma evolução da arquitetura *GoogLeNet/InceptionV1*, projetada com base em princípios para escalar redes convolucionais de forma eficiente, buscando maximizar a acurácia enquanto se mantém um custo computacional e contagem de parâmetros relativamente modestos. Sua principal característica continua sendo os **módulos de Inception** (Figura 2.13), que realizam convoluções com diferentes tamanhos de filtro (ou fatorizações equivalentes) em paralelo dentro do mesmo bloco, concatenando seus resultados. Isso permite à rede capturar informações visuais em múltiplas escalas simultaneamente, melhorando sua capacidade de aprender representações complexas [6].

O *InceptionV3* introduz e aplica diversas técnicas de **fatorização de convoluções** para reduzir a carga computacional sem sacrificar (e frequentemente melhorando) o desempenho. Isso inclui:

- **Fatorização de filtros espaciais grandes:** Convoluções com filtros grandes, como 5x5, são substituídas por uma sequência de filtros menores, como duas convoluções 3x3 consecutivas (Figura 2.14). Isso reduz o número de parâmetros e o custo computacional, ao mesmo tempo que adiciona mais não-linearidades. O paper também aplica isso à camada inicial 7x7, fatorizando-a em uma sequência de 3x3.
- **Fatorização em convoluções assimétricas:** Convoluções $n \times n$ são substituídas por uma convolução $1 \times n$ seguida por uma convolução $n \times 1$. Por exemplo, uma 3x3 pode ser substituída por 1x3 e 3x1. O paper destaca o uso de $n = 7$ (1x7 seguido

de 7x1) como particularmente eficaz em mapas de features de tamanho médio (ex: 17x17).

Essas fatorizações, juntamente com o uso agressivo de convoluções 1x1 para redução de dimensionalidade antes das convoluções mais custosas dentro dos módulos *Inception*, contribuem significativamente para a eficiência da arquitetura. O design modular também oferece flexibilidade para escalar a rede.

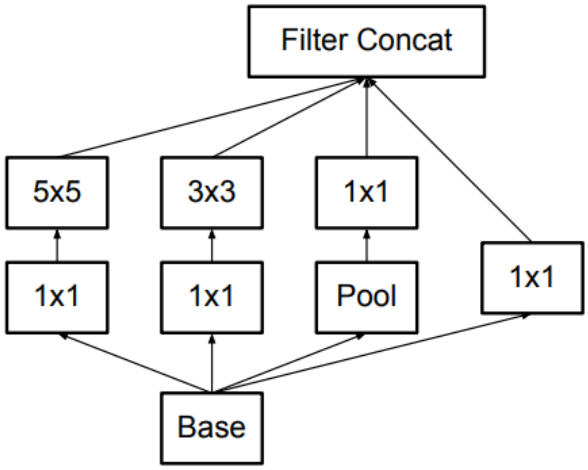


Figura 2.13: Módulo *Inception* original (GoogLeNet/InceptionV1) com filtros paralelos (Szegedy et al., 2015 [5])

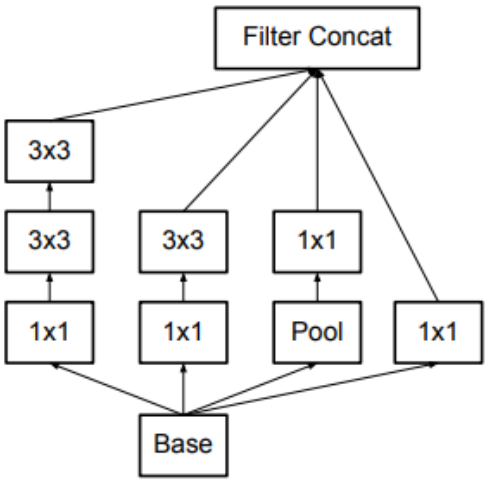


Figura 2.14: Módulo *Inception* com fatorização da convolução 5x5 em duas 3x3 (Szegedy et al., 2016 [6]).

As melhorias arquiteturais baseadas nesses princípios de fatorização e design eficiente (como uma nova forma de reduzir a dimensão do *grid*, ver Figura 10 em [6]) são frequentemente referidas como definindo o **InceptionV2**. O modelo **InceptionV3** incorpora essas

melhorias arquiteturais e adiciona otimizações focadas no treinamento e regularização, incluindo:

- Otimizador **RMSProp**.
- Regularização por **Suavização de Rótulos (Label Smoothing)**, que desencoraja a rede de fazer predições excessivamente confiantes.
- Uso de **Batch Normalization** na camada auxiliar de classificação (além das camadas convolucionais).

Um dos maiores destaques do *InceptionV3* é seu custo computacional e número de parâmetros significativamente menores em comparação com arquiteturas como a *VGGNet*, apesar de alcançar desempenho superior. Com menos de 25 milhões de parâmetros, ele é muito mais leve que os 138M da *VGG16*, tornando-o viável para cenários com restrições de recursos computacionais ou memória.

Entretanto, a **complexidade** da arquitetura, com seus múltiplos ramos paralelos e fatorizações, pode tornar a implementação, modificação e o ajuste fino mais desafiadores em comparação com arquiteturas mais simples e homogêneas. Otimizar todos os hiperparâmetros e configurações pode exigir conhecimento técnico e experimentação consideráveis.

O *InceptionV3* representa um avanço importante em *CNNs*, demonstrando como princípios de design inteligentes, como a fatorização de convoluções e o processamento multi-escala dentro dos módulos *Inception*, podem levar a redes profundas que são ao mesmo tempo eficientes computacionalmente e altamente precisas. Apesar da complexidade na sua concepção, seus benefícios em desempenho e eficiência o estabeleceram como uma arquitetura influente e uma escolha sólida para diversas tarefas de visão computacional.

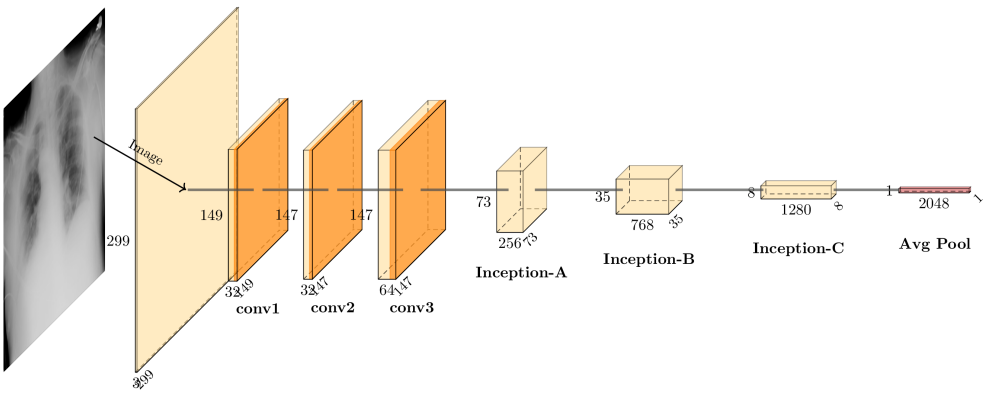


Figura 2.15: Visão geral da arquitetura *InceptionV3* com módulos Inception fatorizados.

2.7 Métricas de Avaliação de Desempenho

As métricas de avaliação desempenham um papel essencial na análise quantitativa do desempenho de algoritmos de aprendizado de máquina, proporcionando uma visão detalhada sobre a eficácia e as características dos modelos de classificação em diferentes contextos. As métricas selecionadas para este trabalho são baseadas em suas propriedades e relevância para problemas de classificação, especialmente em cenários com múltiplas classes e potencial desbalanceamento, como o encontrado no dataset utilizado. As definições consideram uma abordagem binária (positivo vs. negativo) como base, mas são estendidas para o caso multiclasse conforme descrito posteriormente.

Para um problema de classificação binária, as predições de um modelo podem ser resumidas em uma matriz de confusão, da qual derivam as métricas primárias:

- TP (True Positives): Número de instâncias positivas corretamente classificadas como positivas.
- TN (True Negatives): Número de instâncias negativas corretamente classificadas como negativas.
- FP (False Positives): Número de instâncias negativas incorretamente classificadas como positivas (Erro Tipo I).
- FN (False Negatives): Número de instâncias positivas incorretamente classificadas como negativas (Erro Tipo II).

2.7.1 Acurácia

A acurácia (*accuracy*) representa a proporção geral de previsões corretas (tanto positivas quanto negativas) em relação ao total de previsões realizadas. É uma das métricas mais intuitivas e utilizadas:

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.4}$$

Embora útil como uma medida global, a acurácia pode ser enganosa em conjuntos de dados desbalanceados. Se uma classe é muito mais frequente que as outras, um modelo trivial que sempre prevê a classe majoritária pode alcançar uma alta acurácia, mascarando sua incapacidade de identificar corretamente as classes minoritárias [15]. Por essa razão, métricas que avaliam o desempenho por classe são frequentemente preferíveis ou usadas em conjunto.

2.7.2 Precisão (Precision)

A precisão (*Precision*) mede a proporção de instâncias classificadas como positivas que são de fato positivas. Em outras palavras, dentre todas as vezes que o modelo previu a classe positiva, quantas vezes ele acertou?

$$\text{Precisão} = \frac{TP}{TP + FP} \tag{2.5}$$

Uma alta precisão é desejável em cenários onde o custo de um Falso Positivo é alto (ex: classificar um e-mail legítimo como spam, diagnosticar erroneamente uma condição saudável como doença). O foco é na confiabilidade das predições positivas.

2.7.3 Sensibilidade / Revocação (Recall)

A sensibilidade (*Recall*), também conhecida como Revocação ou Taxa de Verdadeiros Positivos (*True Positive Rate* - TPR), mede a proporção de instâncias positivas que foram corretamente identificadas pelo modelo. Ou seja, de todas as instâncias que eram realmente positivas, quantas o modelo conseguiu encontrar?

$$\text{Recall} = \text{Sensibilidade} = \text{TPR} = \frac{TP}{TP + FN} \tag{2.6}$$

Um alto recall é crucial em aplicações onde o custo de um Falso Negativo é alto (ex: deixar de detectar uma doença grave, não identificar uma transação fraudulenta). O foco é na capacidade de encontrar todas as instâncias positivas [15].

2.7.4 Especificidade (Specificity)

Complementar ao Recall, a especificidade (*Specificity*), ou Taxa de Verdadeiros Negativos (*True Negative Rate* - TNR), mede a proporção de instâncias negativas que foram corretamente identificadas como negativas.

$$\text{Especificidade} = \text{TNR} = \frac{TN}{TN + FP} \tag{2.7}$$

É particularmente relevante em diagnósticos, indicando a capacidade do modelo de corretamente identificar indivíduos saudáveis (ou a ausência de uma condição). Note que Especificidade = 1 - Taxa de Falsos Positivos (FPR = FP / (TN + FP)).

2.7.5 F1-Score

O *F1-Score* é a média harmônica da Precisão e do *Recall*, fornecendo um único valor que busca equilibrar ambas as métricas:

$$F1\text{-Score} = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \tag{2.8}$$

A média harmônica penaliza valores extremos; portanto, o *F1-Score* só é alto se tanto a precisão quanto o *recall* forem altos. É especialmente útil em cenários com desbalanceamento de classes, onde a acurácia pode ser enganosa, pois considera tanto os Falsos Positivos (via Precisão) quanto os Falsos Negativos (via *Recall*). Pode-se utilizar variantes como o F_β-Score, que permite dar peso β maior ao *recall* ($\beta > 1$) ou à precisão ($\beta < 1$) [15].

2.7.6 Agregação para Múltiplas Classes

Como o problema abordado neste trabalho envolve 4 classes, as métricas de Precisão, *Recall* e *F1-Score*, calculadas individualmente para cada classe (considerando-a como a classe "positiva"e as demais como "negativas"em uma abordagem *One-vs-Rest*), precisam ser agregadas para fornecer uma visão geral do desempenho multiclasse:

- **Média Macro (*Macro Average*):** Calcula a métrica para cada classe independentemente e depois calcula a média aritmética simples desses valores. Todas as classes contribuem igualmente para a média final, independentemente do número de instâncias (suporte) de cada classe.
- **Média Ponderada (*Weighted Average*):** Calcula a métrica para cada classe independentemente e depois calcula a média ponderada pelo suporte (número de instâncias verdadeiras) de cada classe. Esta média leva em conta o desbalanceamento das classes, dando mais peso às classes com mais instâncias. O **F1-Score Ponderado** foi a métrica principal utilizada neste trabalho para otimização e comparação.

A Acurácia é inerentemente uma métrica global e não requer agregação.

2.7.7 Curva ROC e AUC

A curva ROC (*Receiver Operating Characteristic*) é um gráfico que ilustra a capacidade diagnóstica de um classificador binário à medida que seu limiar de discriminação varia. Ela plota a Taxa de Verdadeiros Positivos (*Recall/Sensibilidade*) no eixo Y contra a Taxa de Falsos Positivos (FPR = 1 - Especificidade) no eixo X, para diferentes limiares. A

Área Sob a Curva ROC (AUC) é uma métrica escalar que resume o desempenho geral do classificador em todos os limiares. A AUC varia de 0 a 1, onde:

- AUC = 0.5 indica um desempenho aleatório (equivalente a jogar uma moeda).
- AUC = 1.0 indica um classificador perfeito.
- AUC < 0.5 indica um desempenho em que não há nenhum tipo de aprendizado.

A AUC é útil por ser independente do limiar de decisão escolhido e por ser insensível ao desbalanceamento das classes. Para problemas multiclasse, a AUC pode ser calculada usando estratégias como *One-vs-Rest (OvR)* ou *One-vs-One (OvO)*, e depois opcionalmente agregada (ex: média macro ou ponderada) [7]. Neste trabalho, utilizou-se a estratégia *OvR*.

2.7.8 Coeficiente de Correlação de Matthews (MCC)

O Coeficiente de Correlação de Matthews (MCC) é outra métrica considerada particularmente informativa para classificação binária, especialmente em cenários desbalanceados, pois leva em conta todas as quatro entradas da matriz de confusão (TP, TN, FP, FN). É definido como:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.9}$$

O MCC varia entre -1 e +1:

- +1 representa uma predição perfeita.
- 0 representa uma predição não melhor que aleatória.
- -1 representa uma discordância total (inversão) entre predição e observação.

Uma versão multiclasse generalizada do MCC também existe e foi utilizada [18].

Além dessas, outras métricas como o índice de Youden ($J = Sensibilidade + Especificidade - 1$) e a potência discriminante (DP), mencionadas por Sokolova et al., poderiam ser utilizadas para análises mais aprofundadas, mas não foram o foco principal deste estudo.

2.8 Busca de Hiperparâmetros

O desempenho de muitos algoritmos de aprendizado de máquina é sensível à escolha de seus hiperparâmetros (parâmetros não aprendidos diretamente dos dados, mas definidos antes do treinamento, como a taxa de aprendizado em boosting ou o número de vizinhos no

KNN). Encontrar uma combinação ótima desses hiperparâmetros é crucial para maximizar a performance do modelo.

Neste trabalho, a **Busca Aleatória (*Randomized Search*)** com validação cruzada foi utilizada como técnica principal para a otimização de hiperparâmetros dos classificadores aplicados às características extraídas pelas *CNNs*. Diferente da abordagem exaustiva do *Grid Search*, a *Randomized Search* amostra um número fixo de combinações de hiperparâmetros a partir de distribuições especificadas ou listas de valores [19]. Para cada combinação amostrada, o modelo é treinado e avaliado utilizando validação cruzada sobre os dados de treinamento. O desempenho médio na validação cruzada (neste caso, utilizando o *F1-Score Ponderado* como métrica principal devido ao desbalanceamento do dataset) é usado para selecionar a melhor combinação de hiperparâmetros encontrada dentro do número de iterações definido [20].

A principal vantagem da *Randomized Search* sobre a *Grid Search* é sua maior **eficiência computacional**, especialmente quando o espaço de busca é grande (muitos hiperparâmetros ou muitos valores possíveis para cada um). Estudos mostram que a *Randomized Search* frequentemente encontra combinações de hiperparâmetros tão boas quanto ou melhores que a *Grid Search*, mas com um custo computacional significativamente menor, pois nem todos os hiperparâmetros são igualmente importantes e a busca aleatória tem maior probabilidade de explorar as dimensões mais relevantes do espaço de busca [19].

Embora a *Randomized Search* tenha sido a técnica principal aqui, outras abordagens como a *Grid Search* (que avalia sistematicamente todas as combinações) e a **Otimização Bayesiana** (que utiliza resultados de iterações anteriores para escolher probabilisticamente as próximas combinações a testar, buscando convergir mais rapidamente para boas regiões do espaço) são alternativas comuns para a otimização de hiperparâmetros [20].

Ferramentas modernas, como a biblioteca **Scikit-Learn** em *Python*, oferecem implementações eficientes e fáceis de usar para essas técnicas (como *RandomizedSearchCV* e *GridSearchCV*), automatizando o processo de busca e validação cruzada e permitindo que o pesquisador se concentre na definição das grades/distribuições de busca e na interpretação dos resultados. Os detalhes específicos da implementação da *Randomized Search* neste trabalho (número de iterações, número de folds na validação cruzada, métrica de score e as grades/distribuições exatas para cada classificador) são apresentados no capítulo de Metodologia.

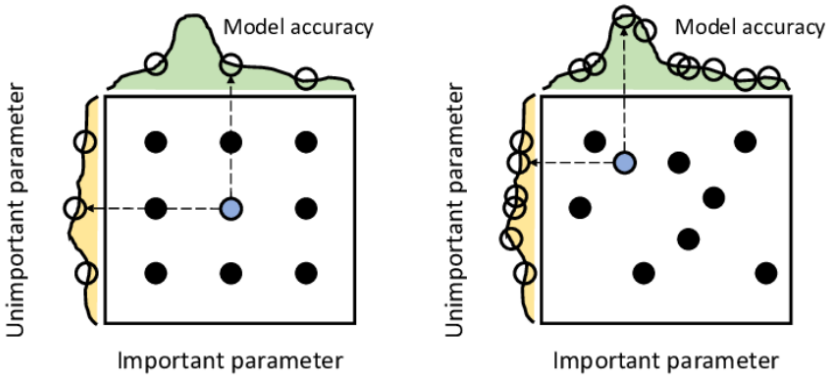


Figura 2.16: Comparativo visual: Busca em Grade (esquerda) vs. Busca Aleatória (direita). Fonte: A Kernel Design Approach to Improve Kernel Subspace Identification. Karl Erza 2020.

2.9 Funções de Ativação

As funções de ativação desempenham um papel crucial nas redes neurais, determinando como as entradas são transformadas em saídas. Elas introduzem não-linearidades nos modelos, permitindo que as redes aprendam padrões complexos e representem relações intricadas nos dados. Sem essas funções, as redes neurais se comportariam como modelos lineares, limitando sua capacidade de modelar relações não lineares.

As funções de ativação mais utilizadas incluem:

2.9.1 Sigmoide

A função *Sigmoide* mapeia as entradas para o intervalo $[0, 1]$, sendo particularmente útil em problemas de classificação binária e em camadas de saída para representar probabilidades. Sua fórmula matemática é dada por:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

onde e representa a base do logaritmo natural. O gráfico da função possui uma forma em "S", garantindo que valores negativos tendam a 0 e valores positivos a 1.

Embora útil, a *sigmoide* apresenta algumas limitações:

- **Problema do gradiente desaparecendo**: Em redes profundas, os gradientes podem se tornar extremamente pequenos durante a retropropagação, dificultando o aprendizado das camadas iniciais.
- **Não centralizada em zero**: A saída da função sigmoide varia entre 0 e 1, o que pode resultar em atualizações de peso mais lentas durante o treinamento [21].

2.9.2 Tanh

A função *Tanh* (tangente hiperbólica) é semelhante à sigmoide, mas mapeia as entradas para o intervalo $[-1, 1]$, o que melhora a normalização dos dados ao redor de zero. Sua fórmula é:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.11}$$

Vantagens da função *Tanh*:

- **Centrada em zero**: Diferente da sigmoide, os valores de saída variam entre -1 e 1, o que facilita a propagação do gradiente e melhora a convergência do modelo.
- **Melhor para valores negativos e positivos**: Representa melhor distribuições simétricas dos dados.

No entanto, a *Tanh* ainda sofre com o problema do gradiente desaparecendo para valores extremos [21].

2.9.3 ReLU (Rectified Linear Unit)

A função *ReLU* é amplamente utilizada devido à sua simplicidade e eficiência computacional. Ela é definida como:

$$\text{ReLU}(x) = \max(0, x) \tag{2.12}$$

Ou seja, para valores negativos, a saída é 0, e para valores positivos, a saída é o próprio valor da entrada.

Principais vantagens da *ReLU*:

- **Não sofre com o problema do gradiente desaparecendo em valores positivos**: Mantém um gradiente constante para valores positivos, acelerando o treinamento.
- **Eficiência computacional**: Simples de calcular e amplamente usada em arquiteturas profundas.

No entanto, a *ReLU* pode levar ao problema de **"neurônios mortos"**, onde neurônios podem parar de aprender se receberem entradas negativas continuamente [21].

2.9.4 ELU (Exponential Linear Unit)

A função *ELU* (*Exponential Linear Unit*) é uma variação da *ReLU* que busca resolver o problema dos neurônios mortos ao permitir valores negativos suavizados por uma exponencial. Sua fórmula é:

$$\text{ELU}(x) = \begin{cases} x, & \text{se } x > 0 \\ \alpha(e^x - 1), & \text{se } x \leq 0 \end{cases} \tag{2.13}$$

onde α é um hiperparâmetro positivo que controla o comportamento da função para valores negativos.

Vantagens da *ELU*:

- **Melhor convergência**: Para valores negativos, a exponencial reduz o impacto dos neurônios mortos.
- **Maior estabilidade**: Suaviza o aprendizado e reduz a variação dos gradientes [21].

2.9.5 Swish e Mish

As funções *Swish* e *Mish* são funções de ativação mais recentes que apresentam um desempenho superior em redes profundas. A função *Swish* foi introduzida pelo Google e é definida como:

$$\text{Swish}(x) = x \cdot \sigma(x) = x \cdot \frac{1}{1 + e^{-x}} \tag{2.14}$$

Ela combina elementos da *ReLU* e da Sigmoide, permitindo valores negativos e suavizando a ativação.

A função *Mish* é uma alternativa à *Swish* com uma ativação ainda mais suave:

$$\text{Mish}(x) = x \cdot \tanh(\ln(1 + e^x)) \tag{2.15}$$

Vantagens das funções *Swish* e *Mish*:

- **Suavização do gradiente**: Reduz a variação abrupta dos gradientes durante o treinamento.
- **Melhor desempenho em redes profundas**: Demonstram resultados superiores na prática, evitando saturação [21].

2.9.6 Considerações sobre a Escolha da Função de Ativação

A escolha da função de ativação é um fator crítico no desenvolvimento de modelos de aprendizado de máquina. Cada função apresenta vantagens e desafios que devem ser considerados de acordo com o problema específico:

- Para **classificação binária**, a Sigmoide pode ser útil na camada de saída.
- Em redes profundas, *ReLU* e suas variantes (*ELU*, *Leaky ReLU*, *Mish*, *Swish*) são preferidas devido à eficiência computacional e à melhor propagação do gradiente.
- Para evitar neurônios mortos, funções como *ELU*, *Mish* e *Swish* são boas alternativas à *ReLU*.

Pesquisas recentes exploram funções de ativação treináveis, que se ajustam automaticamente durante o aprendizado, oferecendo um potencial de melhoria ainda maior no desempenho dos modelos [21]. A escolha apropriada da função de ativação influencia diretamente a capacidade de generalização e a eficiência do treinamento da rede neural.

2.10 Classificadores Utilizados

Os classificadores de aprendizado de máquina desempenham um papel fundamental na tarefa de prever categorias a partir de um conjunto de dados de entrada. Esses algoritmos utilizam padrões extraídos dos dados para associar novas instâncias a classes predefinidas, sendo amplamente empregados em aplicações como diagnósticos médicos, detecção de fraudes, reconhecimento de padrões e processamento de linguagem natural.

Cada classificador possui características próprias, influenciadas por aspectos como a distribuição dos dados, a complexidade do problema e a necessidade de interpretabilidade do modelo. Enquanto alguns algoritmos, como a Regressão Logística, assumem relações lineares entre as variáveis preditoras, outros, como Árvores de Decisão e *Random Forest*, lidam bem com relações não lineares e fornecem interpretações intuitivas dos critérios de decisão. Modelos mais sofisticados, como *Support Vector Machines (SVM)* e algoritmos baseados em redes neurais, são capazes de capturar padrões mais complexos e generalizar melhor em conjuntos de dados de grande escala.

A escolha do classificador adequado depende do contexto da aplicação, do volume e da qualidade dos dados disponíveis, além das métricas de avaliação utilizadas. Nesta seção, são apresentados os algoritmos de classificação empregados neste trabalho, destacando suas características, vantagens e limitações.

2.10.1 Logistic Regression (Regressão Logística)

A Regressão Logística, conforme descrito por Cramer et al., é um modelo estatístico amplamente empregado para prever a probabilidade de ocorrência de um evento, especialmente em situações onde a variável dependente é categórica, como sucesso/falha ou sim/não. Este modelo é conhecido por sua simplicidade, eficiência computacional e facilidade de interpretação, tornando-se uma escolha popular em problemas de classificação binária.

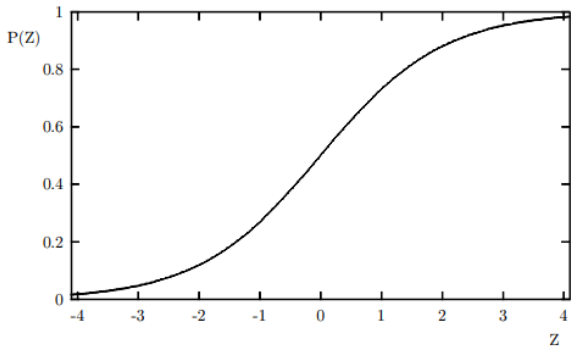


Figura 2.17: A curva de Regressão Logística (Cramer et al., 2002)

Uma das principais características da Regressão Logística é sua capacidade de gerar probabilidades de classe, permitindo a análise de incertezas e auxiliando na tomada de decisões baseadas em classificação probabilística. O modelo apresenta bom desempenho em cenários onde os dados são linearmente separáveis e as classes estão balanceadas, sendo particularmente útil quando interpretabilidade e previsões probabilísticas são necessárias.

No entanto, a Regressão Logística possui limitações importantes. Seu desempenho pode ser significativamente reduzido em contextos onde as relações entre as variáveis independentes e dependentes são altamente complexas ou não lineares. Além disso, sua eficácia é sensível ao equilíbrio entre as classes, o que a torna menos adequada para conjuntos de dados desbalanceados. Para lidar com tais cenários, técnicas como *oversampling*, *undersampling* ou a introdução de pesos nas classes podem ser aplicadas.

Diferentemente da Regressão Linear, que prevê valores contínuos, a Regressão Logística utiliza a função logística, ou sigmoide, para transformar as saídas do modelo em probabilidades que variam entre 0 e 1. A fórmula subjacente é expressa como:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}} \tag{2.16}$$

onde $P(y = 1|x)$ é a probabilidade de ocorrência do evento $y = 1$ dado o vetor de características x, β_0 é o intercepto, e $\beta_1, \beta_2, \ldots, \beta_n$ são os coeficientes das variáveis

independentes. A função sigmoide assegura que a saída seja limitada ao intervalo [0, 1], facilitando a interpretação probabilística.

A Regressão Logística modela, na verdade, o logaritmo da razão de chances (*logit*) da probabilidade $P(y = 1|x)$ como uma função linear das variáveis preditoras:

$$\text{logit}(P(y = 1|x)) = \ln\left(\frac{P(y = 1|x)}{1 - P(y = 1|x)}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \tag{2.17}$$

A aplicação da função sigmoide (inversa da função logit) a esta combinação linear resulta na probabilidade $P(y = 1|x)$ apresentada na Equação 2.16. A suposição de linearidade do modelo refere-se, portanto, à relação entre as variáveis preditoras e o logit da probabilidade.

Os parâmetros do modelo $(\beta_0, \beta_1, \ldots, \beta_n)$ são estimados por meio do método de máxima verossimilhança (*Maximum Likelihood Estimation* - MLE), que encontra os valores dos coeficientes que maximizam a probabilidade de observar os dados de treinamento fornecidos, dado o modelo. Os coeficientes β_i podem ser interpretados em termos de *odds ratio* (razão de chances): e^{β_i} representa a mudança multiplicativa na razão de chances de $y = 1$ para cada aumento unitário em x_i, mantendo as outras variáveis constantes.

Para evitar o sobreajuste (*overfitting*), especialmente em cenários com muitas variáveis preditoras ou multicolinearidade, é comum aplicar regularização ao estimar os coeficientes β_i. As formas mais comuns são a regularização L1 (Lasso), que adiciona a soma dos valores absolutos dos coeficientes $(\lambda \sum |\beta_i|)$ à função de custo e pode zerar alguns coeficientes (atuando como seleção de variáveis), e a regularização L2 (*Ridge*), que adiciona a soma dos quadrados dos coeficientes $(\lambda \sum \beta_i^2)$ e tende a encolhê-los. O hiperparâmetro C, frequentemente utilizado em implementações como a do *Scikit-learn* (onde $C = 1/\lambda$), controla a força da regularização: valores menores de C indicam uma regularização *mais forte*.

A Regressão Logística tem diversas aplicações em áreas como:

- **Medicina**: Para prever a presença ou ausência de doenças com base em características clínicas.
- **Marketing**: Para estimar a probabilidade de um cliente realizar uma compra ou cancelar um serviço.
- **Finanças**: Para avaliar a probabilidade de inadimplência em empréstimos.
- **Ciências Sociais**: Para analisar fatores que influenciam decisões ou comportamentos binários.

Embora fundamentalmente binária, a Regressão Logística pode ser estendida para problemas de classificação com múltiplas classes (K > 2). Abordagens comuns incluem

a Regressão Logística Multinomial (também conhecida como Regressão *Softmax*), que modela diretamente as probabilidades das K classes, ou estratégias como *One-vs-Rest (OvR)*, onde K classificadores binários são treinados, cada um separando uma classe das demais.

É fundamental estabelecer a conexão entre a Regressão Logística e a camada de saída de uma rede neural de classificação. Em uma rede neural completa, a camada final, que geralmente utiliza uma função de ativação *Softmax* (a generalização da função *sigmoid* para múltiplas classes), pode ser formalmente interpretada como um classificador de Regressão Logística Multinomial aplicado sobre as representações de alto nível geradas pelas camadas anteriores da rede.

Seja $x \in R^d$ uma entrada, e $z = Wx + b \in R^K$ o vetor de *logits* (pontuações não normalizadas) para K classes (neste caso, $K = 4$). A função *Softmax* transforma esses *logits* em um vetor de probabilidades $\hat{y} \in R^K$ tal que:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{para } i = 1, 2, \ldots, K,$$

com a garantia de que $\sum_{i=1}^{K} \hat{y}_i = 1$ e $0 < \hat{y}_i < 1$. Essa operação é análoga à Regressão Logística Multinomial, cuja função de perda associada é a *categorical cross-entropy*:

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log(\hat{y}_i),$$

onde y é o vetor *one-hot* da classe verdadeira.

A diferença crucial entre os paradigmas está no processo de treinamento: em uma rede neural treinada de forma *end-to-end*, os pesos tanto das camadas de extração de características (e.g., convolucionais) quanto da camada de classificação final (*Softmax*) são ajustados simultaneamente via *backpropagation*, otimizando conjuntamente as representações x e os pesos W da camada de saída.

Por outro lado, na abordagem de usar uma *CNN* como extratora de características para alimentar um modelo separado de Regressão Logística, o processo é desacoplado: primeiro, a rede neural (frequentemente pré-treinada com pesos congelados) produz um vetor de características x, que é mantido fixo. Em seguida, treina-se um classificador linear $\hat{y} = \text{softmax}(Wx + b)$ usando apenas esses vetores como entrada.

Ambas as abordagens utilizam a mesma fórmula matemática na camada de saída, mas diferem em como x e W são obtidos e otimizados. A abordagem *end-to-end* tende a ser mais poderosa por permitir que x seja adaptado especificamente à tarefa de classificação, enquanto a abordagem desacoplada é mais modular e eficiente computacionalmente, permitindo explorar rapidamente diferentes classificadores lineares sobre representações robustas já aprendidas.

Apesar de suas limitações, como a dependência de linearidade entre as variáveis independentes e o logit da variável dependente, a Regressão Logística continua sendo uma ferramenta robusta para modelagem e previsão de eventos binários. Como observado por Cramer et al., sua simplicidade e eficácia em muitos cenários tornam-na um elemento essencial no conjunto de ferramentas para aprendizado de máquina e estatística aplicada.

2.10.2 K-Nearest Neighbors (KNN)

O *K-Nearest Neighbors* (KNN) é um algoritmo de classificação não paramétrico e baseado em instâncias (*instance-based learning*), amplamente utilizado em aprendizado de máquina devido à sua simplicidade e eficácia intuitiva [7]. O princípio fundamental do KNN é classificar um novo ponto de dados com base na classe majoritária de seus "k" vizinhos mais próximos no conjunto de treinamento.

A proximidade é geralmente medida por uma métrica de distância. A mais comum é a distância Euclidiana, definida entre um novo ponto x e um ponto de treinamento x_i como:

$$d(x, x_i) = \sqrt{\sum_{j=1}^{D} (x_j - x_{ij})^2} \tag{2.18}$$

onde:

- $x = (x_1, \ldots, x_D)$ é o vetor de características do novo ponto a ser classificado,
- $x_i = (x_{i1}, \ldots, x_{iD})$ representa o i-ésimo ponto do conjunto de treinamento (com N pontos no total),
- x_j e x_{ij} são os valores da j-ésima característica de x e x_i, respectivamente,
- D é o número total de características (dimensão) do espaço de entrada.

Outras métricas, como a distância de Manhattan ou a similaridade de cosseno (útil para dados textuais, por exemplo), também podem ser empregadas dependendo da natureza dos dados.

Após calcular as distâncias entre o novo ponto e todos os N pontos do conjunto de treinamento, o KNN seleciona os k vizinhos mais próximos (o conjunto $\mathcal{N}_k$). A classificação do novo ponto, $\hat{y}$, é então determinada pela classe C mais frequente entre esses vizinhos (voto majoritário):

$$\hat{y} = \arg\max_{C} \sum_{i \in \mathcal{N}_k} 1(y_i = C) \tag{2.19}$$

onde $1(y_i = C)$ é uma função indicadora que retorna 1 se o vizinho i pertence à classe C, e 0 caso contrário.

Uma variação comum ao voto majoritário simples é a **votação ponderada pela distância**, onde a contribuição de cada vizinho para a votação é inversamente proporcional à sua distância do ponto a ser classificado (ex: peso $w_i = 1/d(x, x_i)$). Dessa forma, vizinhos mais próximos têm maior influência na decisão final [7].

Uma das grandes vantagens do KNN é sua ausência de suposições rígidas sobre a distribuição dos dados (não paramétrico), tornando-o flexível para fronteiras de decisão complexas. Além disso, sua implementação é direta e a lógica é fácil de entender. Contudo, o KNN apresenta desafios importantes.

Sua abordagem é classificada como *Lazy Learning* (aprendizado preguiçoso), pois não constrói um modelo explícito durante a fase de treinamento; ele simplesmente armazena todo o conjunto de dados. Toda a computação (cálculo de distâncias e busca por vizinhos) ocorre no momento da classificação. Isso torna a classificação de novos pontos computacionalmente custosa, especialmente para grandes conjuntos de dados (N grande), com uma complexidade de busca ingênua tipicamente $O(N \cdot D)$. Para mitigar este custo na busca dos vizinhos, podem ser usadas estruturas de dados espaciais como **KD-Trees** ou **Ball Trees**, que aceleram a busca em dimensões baixas a moderadas, embora sua eficácia também diminua em dimensionalidade muito alta [7].

O desempenho do KNN é particularmente sensível ao valor de k e à dimensionalidade dos dados. Valores de k muito pequenos tornam o modelo vulnerável a ruído nos dados (alta variância), enquanto valores muito grandes podem suavizar excessivamente as fronteiras de decisão (alto viés), obscurecendo padrões locais. A escolha ótima de k é dependente do problema e geralmente determinada através de **validação cruzada** no conjunto de treinamento [7]. Além disso, o KNN sofre com a **Maldição da Dimensionalidade** (*Curse of Dimensionality*): em espaços com muitas dimensões (D grande), a noção de vizinhança se torna menos significativa, pois os pontos ficam esparsos e as distâncias entre eles tendem a se tornar mais uniformes, degradando o desempenho do algoritmo [7].

É crucial notar que, como o KNN se baseia em distâncias, *features* com escalas (ranges) muito diferentes podem distorcer o resultado, pois a *feature* com maior range pode dominar a métrica. Portanto, uma etapa essencial de pré-processamento para o KNN é o **escalonamento das features** (por exemplo, normalização para média zero e desvio padrão unitário, ou reescalonamento para o intervalo [0, 1]) para garantir que todas as features contribuam de forma equitativa para o cálculo da distância [7].



Figura 2.18: Exemplo de classificação KNN e fronteiras de decisão (adaptado de [7])

Em resumo, o KNN é uma técnica conceitualmente simples e versátil para classificação. Suas principais vantagens são a flexibilidade e a falta de necessidade de treinar um modelo complexo. No entanto, seu uso eficaz requer atenção cuidadosa à escolha de k, à métrica de distância apropriada, à necessidade crítica de pré-processamento (escalonamento de *features*) e às suas limitações inerentes relacionadas à escalabilidade computacional e à maldição da dimensionalidade.

2.10.3 Decision Tree (Árvore de Decisão)

As árvores de decisão são técnicas supervisionadas amplamente utilizadas em aprendizado de máquina para tarefas de classificação e regressão [22]. Sua estrutura hierárquica, semelhante a um fluxograma, inicia com um nó raiz que representa todo o conjunto de dados. A partir desse nó, a árvore se ramifica recursivamente: cada nó interno representa um teste em um atributo específico, cada ramo representa o resultado desse teste, e cada nó folha (ou nó terminal) representa uma classe final (em classificação) ou um valor preditivo (em regressão).

Uma das principais vantagens das árvores de decisão é sua **facilidade de interpretação e visualização**. As regras de decisão explícitas podem ser facilmente compreendidas por humanos, tornando-as ferramentas valiosas em áreas como medicina e finanças, onde a transparência é essencial. Além disso, são capazes de capturar relações não lineares nos dados e geralmente requerem menos pré-processamento de dados em comparação com outros métodos (por exemplo, não exigem escalonamento de features por padrão).

Entretanto, árvores de decisão individuais possuem limitações significativas. Elas podem ser muito propensas ao ***overfitting***, especialmente em conjuntos de dados pequenos ou com ruído, aprendendo padrões muito específicos que não generalizam bem para dados novos. Além disso, são conhecidas por sua **instabilidade**: pequenas variações nos dados

de treinamento podem levar à escolha de atributos diferentes nos nós superiores da árvore, resultando em estruturas radicalmente diferentes e alterando significativamente as predições [22].

Durante a construção da árvore (processo de indução), que geralmente segue uma abordagem *top-down* e gulosa, o algoritmo seleciona o melhor atributo e o melhor ponto de divisão (para atributos numéricos) em cada nó para particionar os dados. A "melhor"divisão é tipicamente aquela que maximiza a pureza dos nós filhos resultantes. Métricas comuns para avaliar a qualidade da divisão em **árvores de classificação** incluem:

- **Entropia e Ganho de Informação (Information Gain):** A entropia mede o grau de desordem ou incerteza em um conjunto de dados S. Para um problema com C classes, é definida como:

$$H(S) = -\sum_{i=1}^{C} p_i \log_2 p_i \tag{2.20}$$

  onde p_i é a proporção de instâncias da classe i no conjunto S. O Ganho de Informação para um atributo A com possíveis valores V é a redução na entropia alcançada ao dividir S usando A:

$$IG(S, A) = H(S) - \sum_{v \in V} \frac{|S_v|}{|S|} H(S_v) \tag{2.21}$$

  onde S_v é o subconjunto de S correspondente ao valor v do atributo A. O algoritmo ID3 [22] utiliza o Ganho de Informação para selecionar o atributo de divisão.

- **Índice Gini (Gini Impurity):** Mede a probabilidade de classificar incorretamente uma instância escolhida aleatoriamente do conjunto S, se ela fosse rotulada aleatoriamente de acordo com a distribuição das classes em S. É calculado como:

$$Gini(S) = 1 - \sum_{i=1}^{C} p_i^2 \tag{2.22}$$

  O critério de divisão busca maximizar a redução no Índice Gini (*Gini Gain*). É o critério padrão no algoritmo CART (*Classification and Regression Trees*).

- **Gain Ratio:** Uma modificação do Ganho de Informação que penaliza atributos com muitos valores distintos, normalizando o ganho pela "informação de divisão"(*Split Information*) do próprio atributo. Foi proposto por Quinlan [22] para mitigar o viés do IG puro.

O atributo e o ponto de divisão que proporcionam a maior redução na incerteza (maior Ganho de Informação, maior *Gini Gain*, ou melhor *Gain Ratio*) são escolhidos para formar o nó de decisão.

Para tarefas de **regressão**, onde o objetivo é prever um valor contínuo, o critério de divisão é diferente. Em vez de minimizar a incerteza da classe, busca-se minimizar a heterogeneidade dos valores de saída nos nós filhos. O critério mais comum é a **redução da variância** ou, equivalentemente, a minimização do Erro Quadrático Médio (MSE) dos valores dentro dos nós resultantes da divisão. Quando se lida com **atributos numéricos**, tanto em classificação quanto em regressão, o algoritmo tipicamente ordena os valores únicos do atributo presentes no nó e avalia todos os pontos de corte (*thresholds*) possíveis (geralmente os pontos médios entre valores consecutivos). O *threshold* que otimiza o critério de divisão escolhido (IG, Gini, redução de variância) é selecionado para criar uma divisão binária (ex: $X \leq threshold$ e $X > threshold$).

A propensão ao *overfitting* é talvez a limitação mais significativa das árvores de decisão individuais. Uma árvore totalmente crescida pode capturar ruído ou padrões muito específicos do conjunto de treinamento, generalizando mal para dados não vistos. Para combater isso, técnicas de **poda (*pruning*)** são essenciais:

- **Pré-poda (Pre-pruning):** Interrompe o crescimento da árvore antecipadamente. Critérios comuns incluem definir uma profundidade máxima (`max_depth`), exigir um número mínimo de amostras para dividir um nó (`min_samples_split`) ou para formar uma folha (`min_samples_leaf`), ou requerer que a melhoria no critério de divisão exceda um certo limiar. Quinlan [22] também discute o uso de testes de significância estatística para avaliar se uma divisão é relevante.
- **Pós-poda (Post-pruning):** Permite que a árvore cresça completamente (ou quase) e, em seguida, remove ramos (subárvores) que são considerados menos confiáveis ou que não melhoram significativamente a capacidade de generalização, geralmente avaliada em um conjunto de validação. Uma técnica conhecida é a Poda por Custo-Complexidade Mínima (*Minimal Cost-Complexity Pruning*).

A poda resulta em árvores mais simples e com melhor desempenho em dados não vistos.

As árvores de decisão têm aplicações variadas, desde a classificação e regressão até a identificação de atributos importantes (*feature importance*). Outra característica importante é sua capacidade inerente (dependendo da implementação) de lidar com **dados faltantes**. Algoritmos como o ID3 original [22] podem, por exemplo, distribuir instâncias com valores faltantes probabilisticamente pelos ramos durante o cálculo do ganho ou durante a classificação, usar o valor mais comum, ou empregar técnicas como *surro-*

gate splits (divisões substitutas baseadas em outros atributos). Além disso, sua estrutura gráfica facilita a apresentação e compreensão dos resultados.

Apesar de suas limitações, como a propensão ao *overfitting* (mitigada pela poda) e a instabilidade, as árvores de decisão permanecem como uma técnica fundamental em aprendizado de máquina, frequentemente servindo como base para métodos de *ensemble* mais poderosos, como *Random Forests* e *Gradient Boosting*. Sua simplicidade, interpretabilidade e eficácia as tornam uma escolha recorrente em diversas aplicações práticas.

2.10.4 Random Forest

O método *Random Forest*, introduzido por Breiman et al., é uma técnica robusta de aprendizado de máquina do tipo *ensemble learning*, que combina múltiplas árvores de decisão para realizar classificações ou previsões com maior acurácia e capacidade de generalização do que árvores individuais.

O princípio central está na criação de um conjunto (floresta) de B árvores de decisão $f_b(x)$, onde duas fontes principais de aleatoriedade são introduzidas:

1. **Amostragem Bootstrap (Bagging):** Cada árvore f_b é treinada em uma amostra de dados diferente, obtida através de amostragem com reposição (*bootstrap sampling*) do conjunto de treinamento original. Isso significa que cada árvore "vê"uma versão ligeiramente diferente dos dados.

2. **Seleção Aleatória de Atributos (Random Subspace):** Ao construir cada nó de cada árvore, em vez de avaliar todos os atributos disponíveis para encontrar a melhor divisão, apenas um subconjunto aleatório de m atributos (onde m é um hiperparâmetro, geralmente muito menor que o número total de atributos D) é considerado. Isso ajuda a descorrelacionar as árvores da floresta.

As árvores individuais dentro da floresta são tipicamente crescidas até a profundidade máxima permitida pelos dados (ou até um limite pré-definido), e, importante, **geralmente não são podadas** (*pruned*) [13]. A aleatoriedade e a agregação do ensemble mitigam o alto viés que árvores profundas individuais teriam.

Para realizar uma predição para um novo ponto x, as previsões de todas as B árvores são agregadas. Em problemas de **classificação**, a predição final $\hat{y}$ é a classe mais votada (votação majoritária) entre todas as árvores:

$$\hat{y} = \arg\max_{c} \sum_{b=1}^{B} 1(f_b(x) = c) \tag{2.23}$$

onde $1(\cdot)$ é a função indicadora e c representa as classes possíveis. Para problemas de **regressão**, a predição final $\hat{y}$ é a média das predições individuais das B árvores:

$$\hat{y} = \frac{1}{B} \sum_{b=1}^{B} f_b(x) \tag{2.24}$$

Essa abordagem combinada (*Bagging + Random Subspace*) impacta o **trade-off viés-variância**. O *Bagging* por si só reduz principalmente a variância das predições. A introdução da seleção aleatória de atributos em cada nó descorrelaciona ainda mais as árvores, o que permite uma redução adicional e significativa da variância do modelo final, geralmente sem um aumento substancial no viés [13]. O resultado é um modelo mais robusto a ruídos e a pequenas variações nos dados de treinamento em comparação com uma única árvore de decisão.

Os **hiperparâmetros** mais importantes são o número de árvores na floresta (B, ou `n_estimators`) e o número de atributos considerados em cada nó (m, ou `max_features`). Um valor maior de B geralmente melhora o desempenho (até um ponto de convergência), mas aumenta o custo computacional. Um número muito pequeno de árvores pode resultar em alta variância. Para m, valores comuns e heurísticas recomendadas são $m \approx \sqrt{D}$ para classificação e $m \approx D/3$ para regressão, onde D é o número total de atributos [13].

Uma das grandes vantagens do *Random Forest* é sua capacidade de lidar bem com conjuntos de dados de alta dimensionalidade e modelar relações complexas entre variáveis. Ele também é notável por sua resistência ao ruído nos dados e por ser menos sensível a *outliers* do que outros métodos. Além disso, o processo de bootstrap permite calcular estimativas de erro *out-of-bag* (OOB) e de importância das variáveis:

- **Erro Out-of-Bag (OOB):** Para cada árvore, aproximadamente um terço das amostras originais não são incluídas no bootstrap sample. Essas amostras OOB podem ser usadas para avaliar o desempenho daquela árvore específica. Agregando essas predições OOB para todas as árvores, obtém-se uma estimativa não viesada do erro de generalização da floresta sem a necessidade de um conjunto de validação separado [13].
- **Importância das Variáveis (Feature Importance):** Uma medida comum é a importância por permutação. Para cada árvore, calcula-se o erro OOB. Em seguida, para uma variável específica, seus valores são permutados aleatoriamente nas amostras OOB e o erro OOB é recalculado. A média da degradação no desempenho (aumento do erro) causada pela permutação dessa variável, calculada sobre todas as árvores, indica sua importância [13].

No entanto, o *Random Forest* apresenta desvantagens. O treinamento pode ser computacionalmente intensivo, especialmente para grandes conjuntos de dados e um número elevado de árvores (embora seja facilmente paralelizável). Além disso, o modelo final é uma "caixa-preta"e **menos interpretável** diretamente em comparação com uma única árvore de decisão, o que pode dificultar a compreensão das razões por trás de uma predição específica.

O *Random Forest* possui aplicações diversas, sendo amplamente utilizado em áreas como medicina (diagnóstico, prognóstico), biologia (genômica, proteômica), sensoriamento remoto, finanças, marketing (segmentação de clientes, previsão de churn) e ciências ambientais. Sua robustez e precisão frequentemente o colocam entre os algoritmos de melhor desempenho para dados tabulares. Tornando um dos algoritmos de referência em muitas aplicações práticas.

2.10.5 XGBoost (Extreme Gradient Boosting)

O *XGBoost* (*Extreme Gradient Boosting*) é uma implementação otimizada e escalável do algoritmo de *gradient boosting*, proposta por Chen et al. Como outras técnicas de *boosting*, ele constrói um modelo preditivo robusto combinando iterativamente múltiplos modelos fracos, tipicamente árvores de decisão. O *XGBoost* se destaca por sua eficiência computacional, flexibilidade e desempenho de ponta, tornando-se uma escolha frequente em competições de aprendizado de máquina e aplicações industriais [23, 14].

O *XGBoost* formula o aprendizado como um problema de otimização aditiva. A previsão final para uma amostra x_i após t iterações, $\hat{y}_i^{(t)}$, é a soma das previsões das t árvores construídas até então:

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \tag{2.25}$$

onde $f_t(x)$ representa a nova árvore (função) adicionada na t-ésima iteração, treinada para minimizar os erros residuais do modelo anterior.

O objetivo em cada iteração t é encontrar a função f_t que minimiza a seguinte função de custo, que inclui a função de perda l e um termo de regularização Ω:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{k=1}^{t} \Omega(f_k) \tag{2.26}$$

Para otimizar isso eficientemente, o *XGBoost* utiliza uma aproximação de Taylor de segunda ordem da função de perda em torno da predição atual $\hat{y}_i^{(t-1)}$:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constante} \tag{2.27}$$

onde $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ e $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ são, respectivamente, o gradiente de primeira e segunda ordem (Hessiana) da função de perda avaliados na iteração anterior. O uso da informação de segunda ordem (h_i) é um diferencial que permite ao *XGBoost* otimizar funções de perda mais genéricas e convergir mais rapidamente [14].

Um dos principais diferenciais do *XGBoost* é seu termo de **regularização** $\Omega(f)$, que controla a complexidade do modelo para prevenir *overfitting*:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{2.28}$$

onde:

- T é o número de folhas na árvore f.
- w_j são os scores (pesos) preditos nas folhas.
- γ é o hiperparâmetro que penaliza a complexidade da árvore baseada no número de folhas (similar a uma regularização L0).
- λ é o hiperparâmetro que controla a força da **regularização L2** (Ridge) sobre os pesos das folhas, encolhendo-os em direção a zero.

Além dessa regularização explícita na função objetivo, o *XGBoost* emprega outras técnicas para melhorar a generalização, como **shrinkage** (onde a contribuição de cada nova árvore é escalada por uma taxa de aprendizado η, similar ao *stochastic gradient boosting* [24]) e **subamostragem** (amostragem aleatória de colunas/features por árvore ou por divisão, e/ou amostragem de linhas/instâncias por árvore) [14].

Outra vantagem significativa é o tratamento eficiente de **dados esparsos e valores faltantes**. O *XGBoost* implementa um algoritmo *sparsity-aware* que aprende automaticamente a melhor direção (ramo esquerdo ou direito) para encaminhar instâncias com valores faltantes em cada nó durante o treinamento, eliminando a necessidade de pré-imputação e otimizando o cálculo das divisões [14].

O *XGBoost* é reconhecido por sua alta **escalabilidade e eficiência**, resultado de otimizações de sistema como paralelização (na construção das árvores), estrutura de dados otimizada (*column blocks*) e computação *out-of-core* (*cache-awareness*, compressão de blocos) [14]. Isso permite treinar modelos em grandes volumes de dados com uso eficiente de recursos, sendo uma razão para sua popularidade em competições como *Kaggle* e em ambientes industriais. Sua flexibilidade permite integração com linguagens como Python, R e Julia.

Suas **aplicações** abrangem tarefas de classificação, regressão, ranking e previsão de séries temporais, sendo amplamente utilizado em áreas como finanças, marketing, bioinformática e saúde [14].

Apesar de suas vantagens, o *XGBoost* pode apresentar desafios no **ajuste de hiperparâmetros**. A grande quantidade de parâmetros (número de árvores, taxa de aprendizado η, profundidade máxima das árvores, γ, λ, parâmetros de subamostragem, etc.) exige um processo de otimização cuidadoso (ex: *Grid Search*, *Random Search*, Otimização Bayesiana) para encontrar a melhor configuração e evitar *overfitting* ou *underfitting* [23].

O *XGBoost* é uma ferramenta poderosa e versátil que representa o estado da arte em *gradient boosting*. Suas inovações em regularização, otimização (uso de gradientes de segunda ordem), tratamento de esparsidade e eficiência de sistema o tornam uma escolha ideal para uma vasta gama de problemas de aprendizado de máquina, embora sua implementação demande atenção ao ajuste de hiperparâmetros para garantir resultados ótimos.

2.10.6 Support Vector Machine (SVM)

As Máquinas de Vetores de Suporte (*Support Vector Machines* - SVMs) são algoritmos de aprendizado supervisionado robustos e eficazes, amplamente utilizados em tarefas de classificação e regressão [12]. Baseadas em sólidos fundamentos da teoria de aprendizado estatístico (*Statistical Learning Theory*), as SVMs se destacam especialmente em problemas com dados de alta dimensionalidade, comuns na área biológica, por exemplo. Segundo Noble et al., o funcionamento das SVMs pode ser compreendido através de quatro conceitos principais: o hiperplano separador, o hiperplano de margem máxima, a margem suave (*soft margin*) e a função kernel (e o *kernel trick*).

O objetivo central de uma SVM de classificação é encontrar um **hiperplano separador** ótimo no espaço de características, que divida os dados pertencentes a classes distintas. Para um problema de classificação binária, dado um conjunto de treinamento $\{(x_i, y_i)\}_{i=1}^{m}$, onde x_i é um vetor de características e $y_i \in \{-1, 1\}$ é o rótulo da classe, o hiperplano é definido por $\mathbf{w} \cdot x + b = 0$. A SVM busca os parâmetros $\mathbf{w}$ (vetor de pesos perpendicular ao hiperplano) e b (termo de viés ou bias) que não apenas separem as classes, mas que o façam com a maior **margem** possível. A margem é a distância entre o hiperplano separador e os pontos de dados mais próximos de cada classe. Esses pontos que definem a margem são chamados de **vetores de suporte** (*support vectors*).

No caso ideal em que os dados são linearmente separáveis, a SVM encontra o **hiperplano de margem máxima** resolvendo o seguinte problema de otimização (formulação

primal):

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.29}$$

$$\text{sujeito a} \quad y_i(\mathbf{w} \cdot x_i + b) \geq 1, \quad \forall i = 1, \ldots, m \tag{2.30}$$

Minimizar $\frac{1}{2}\|\mathbf{w}\|^2$ é equivalente a maximizar a margem $(2/\|\mathbf{w}\|)$. A restrição $y_i(\mathbf{w}\cdot x_i+b) \geq 1$ garante que todos os pontos sejam classificados corretamente e estejam fora ou sobre a margem. Notavelmente, a solução (o hiperplano $\mathbf{w}, b$) depende apenas dos vetores de suporte, tornando o modelo eficiente em termos de memória na predição.

Para lidar com dados que não são linearmente separáveis (devido a ruído ou sobreposição de classes), introduz-se a **margem suave** (*soft margin*). Variáveis de folga (*slack variables*) $\xi_i \geq 0$ são adicionadas para permitir que alguns pontos violem a margem ou até mesmo sejam classificados incorretamente. O problema de otimização torna-se:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \xi_i \tag{2.31}$$

$$\text{sujeito a} \quad y_i(\mathbf{w} \cdot x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \forall i = 1, \ldots, m \tag{2.32}$$

O hiperparâmetro $C > 0$ é o parâmetro de regularização que controla o **trade-off entre maximizar a margem e minimizar os erros de classificação** no treino. Um valor baixo de C permite uma margem maior, tolerando mais violações (modelo mais simples, risco de underfitting), enquanto um valor alto de C penaliza fortemente as violações, buscando classificar mais pontos corretamente, mesmo que isso resulte em uma margem menor (modelo mais complexo, risco de *overfitting*).

Quando a fronteira de decisão entre as classes é inerentemente não linear, a SVM utiliza o **truque do kernel (*kernel trick*)**. A ideia é mapear implicitamente os dados do espaço original de características para um espaço de dimensionalidade maior, onde se espera que os dados se tornem linearmente separáveis (ou mais facilmente separáveis). Isso é feito através de uma **função kernel** $K(x_i, x_j)$, que calcula o produto escalar entre as imagens dos pontos x_i e x_j no espaço de alta dimensão, sem a necessidade de computar explicitamente as coordenadas nesse espaço. Algumas funções kernel comuns são:

- **Linear:** $K(x_i, x_j) = x_i \cdot x_j$ (corresponde à SVM linear no espaço original).
- **Polinomial:** $K(x_i, x_j) = (\gamma(x_i \cdot x_j)+r)^d$, com grau d, coeficiente γ e termo constante r.
- **Gaussiano (RBF - Radial Basis Function):** $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$, onde $\gamma > 0$. Este é um kernel popular e poderoso, capaz de criar fronteiras de decisão complexas. O parâmetro γ controla a "largura"da função gaussiana; valores pequenos de γ resultam em fronteiras mais suaves (maior influência dos pontos),

enquanto valores grandes resultam em fronteiras mais ajustadas aos dados (menor influência, risco de overfitting).

- **Sigmoide:** $K(x_i, x_j) = \tanh(\gamma(x_i \cdot x_j) + r)$.

O *kernel trick* permite que a SVM modele relações não lineares de forma computacionalmente eficiente.

Apesar de suas vantagens, as SVMs apresentam limitações. A complexidade computacional do treinamento pode ser alta para conjuntos de dados muito grandes. Embora o pior caso teórico possa ser $O(N^3)$ ou $O(N^2D)$ (onde N é o número de amostras e D o número de features), solvers modernos e otimizações (como *SMO - Sequential Minimal Optimization*) tornam o treinamento prático para muitos problemas, com desempenho frequentemente observado entre linear e quadrático [12]. A escolha do kernel e o ajuste fino dos hiperparâmetros (como C e γ para o kernel RBF) são cruciais para o bom desempenho e podem exigir validação cruzada extensiva, o que é computacionalmente caro [12].

Além da classificação, existe a variante **Support Vector Regression (SVR)**. O objetivo do SVR é encontrar uma função que se ajuste aos dados de forma que o máximo possível de pontos esteja dentro de um 'tubo' de margem ϵ ao redor da função, enquanto se minimiza a complexidade (achatamento) da função (minimizando $\frac{1}{2}\|\mathbf{w}\|^2$). Pontos fora do tubo ϵ são penalizados de forma similar à margem suave na classificação.

As SVMs têm demonstrado excelentes resultados em diversas aplicações, como classificação de câncer com base em perfis de expressão gênica, análise de imagens médicas, reconhecimento de texto e categorização de sequências de DNA e proteínas [12]. Sua combinação de base teórica sólida (maximização de margem ligada à capacidade de generalização), eficácia empírica e bom desempenho em espaços de alta dimensão as torna uma ferramenta indispensável no aprendizado de máquina supervisionado.

2.10.7 CatBoost (Categorical Boosting)

O *CatBoost* (*Categorical Boosting*) é um algoritmo de *gradient boosting* sobre árvores de decisão (GBDT) de código aberto desenvolvido pela Yandex [25]. Ele foi projetado especificamente para lidar de forma eficaz com características categóricas e para combater um problema estatístico inerente a muitas implementações de GBDT: o *prediction shift* (desvio de predição), causado por uma forma de *target leakage* (vazamento do alvo). O algoritmo introduz duas inovações principais para mitigar esses problemas: *ordered target statistics* para codificação de variáveis categóricas e *ordered boosting* como uma modificação do processo de *boosting* tradicional [25].

Ordered Target Statistics

Uma das formas mais eficazes de lidar com variáveis categóricas em GBDT é convertê-las em variáveis numéricas usando estatísticas baseadas no valor alvo (*target statistics* - TS), como a média do alvo para cada categoria [26]. No entanto, a abordagem ingênua (*greedy*) de calcular essa média usando todos os dados de treino, incluindo o próprio exemplo sendo codificado, leva ao *target leakage*, onde a informação do alvo y_k vaza para a característica $\hat{x}_k^i$ gerada [25]. Isso causa um desvio condicional $(\hat{x}^i|y)$ entre os dados de treino e teste.

Para evitar isso, o *CatBoost* implementa as *Ordered Target Statistics*. A ideia central é baseada em um princípio de ordenação inspirado em aprendizado online:

1. Uma permutação aleatória σ dos dados de treinamento é gerada.
2. Para cada exemplo x_k, a estatística alvo (TS) para uma característica categórica i é calculada usando apenas os exemplos x_j que o precedem na permutação ($\sigma(j) < \sigma(k)$). A fórmula geral, incorporando um termo de suavização com um prior p e um peso a, é:
$$\hat{x}_k^i = \frac{\sum_{j:\sigma(j)<\sigma(k)} I_{\{x_j^i = x_k^i\}} \cdot y_j + ap}{\sum_{j:\sigma(j)<\sigma(k)} I_{\{x_j^i = x_k^i\}} + a} \tag{2.33}$$
3. Para dados de teste, a TS é calculada usando todo o conjunto de treinamento.

Essa abordagem garante que o valor alvo y_k não seja usado para calcular $\hat{x}_k^i$, eliminando o *target leakage* direto e satisfazendo a propriedade de que a distribuição condicional $E(\hat{x}^i|y = v)$ seja a mesma para treino e teste. Para reduzir a variância das estimativas para os primeiros exemplos na permutação (que têm um histórico pequeno), o *CatBoost* utiliza múltiplas permutações durante o treinamento [25].

Ordered Boosting

O problema de *prediction shift* também ocorre no processo padrão de *gradient boosting*. Os gradientes $g^t(x_k, y_k)$ usados para treinar a árvore na iteração t são calculados usando o modelo F^{t-1}, que por sua vez foi treinado usando (potencialmente todos) os exemplos de treino, incluindo x_k. Isso faz com que a distribuição do gradiente $g^t(x_k, y_k)|x_k$ para um exemplo de treino x_k seja diferente da distribuição $g^t(x, y)|x$ para um exemplo de teste x, levando a um viés no modelo final [25].

O *Ordered Boosting* do *CatBoost* aborda isso aplicando o mesmo princípio de ordenação:

1. Utiliza uma permutação aleatória σ dos dados (idealmente a mesma usada para as *Ordered TS*).

2. Mantém múltiplos modelos de suporte M_i durante o treinamento, onde M_i é treinado usando apenas os primeiros i exemplos na permutação σ.

3. Para calcular o resíduo (ou gradiente) do exemplo x_k na iteração t, utiliza-se o modelo $M^{t-1}_{\sigma(k)-1}$, que não foi treinado com o exemplo x_k.

Embora a ideia conceitual exija n modelos, a implementação prática no *CatBoost* usa aproximações eficientes e múltiplas permutações para tornar o processo viável computacionalmente, mantendo complexidade assintótica similar ao GBDT padrão [25]. O *CatBoost* oferece dois modos: *Ordered* (que implementa esta lógica) e *Plain* (GBDT padrão, mas que ainda utiliza *Ordered TS* para as variáveis categóricas). O modo *Ordered* tende a ser particularmente benéfico em datasets menores, onde o viés do *prediction shift* é mais pronunciado [25].

Outras Características

Além das inovações principais, o CatBoost implementa outras técnicas relevantes:

- **Árvores Oblívias:** Utiliza árvores de decisão onde o mesmo critério de divisão é aplicado a todos os nós de um mesmo nível. Essas árvores são balanceadas, menos propensas a *overfitting* e permitem uma avaliação mais rápida do modelo treinado [25].
- **Combinação de Características Categóricas:** Constrói novas características categóricas combinando de forma gulosa as características existentes (e suas combinações) que já foram usadas em divisões anteriores na árvore atual. Essas combinações são então convertidas em *Ordered TS* [25].
- **Tratamento Eficiente de Categóricas:** O foco em *Ordered TS* e árvores oblívias permite um tratamento nativo e eficiente de variáveis categóricas, mesmo as de alta cardinalidade, sem a necessidade de pré-processamento manual como *one-hot encoding*.

2.10.8 Naive Bayes

O algoritmo *Naive Bayes* (NB) é uma abordagem de classificação probabilística simples, mas poderosa, baseada no teorema de Bayes. É amplamente utilizado em tarefas como filtragem de spam, diagnóstico médico, classificação de textos e sistemas de recomendação [27]. O modelo calcula a probabilidade de uma instância pertencer a uma determinada classe com base nas probabilidades observadas das suas características (*features*).

Dado um conjunto de características $x = (x_1, x_2, \ldots, x_n)$, a probabilidade de uma classe y ser a correta, $P(y|x)$, é calculada usando o Teorema de Bayes:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{2.34}$$

onde:

- $P(y|x)$ é a **probabilidade posterior** da classe y, dados os atributos x. É o que queremos estimar para classificar x.
- $P(x|y)$ é a **verossimilhança** (*likelihood*), a probabilidade de observar os atributos x dado que a classe é y.
- $P(y)$ é a **probabilidade a priori** da classe y, geralmente estimada pela frequência da classe no conjunto de treinamento.
- $P(x)$ é a **evidência**, a probabilidade marginal dos atributos x. Como $P(x)$ é constante para todas as classes para um dado x, ele atua como um fator de normalização e pode ser ignorado ao comparar as probabilidades posteriores entre as classes.

A característica fundamental e simplificadora do *Naive Bayes* é a sua suposição *naive* (ingênua) de **independência condicional** entre as variáveis preditoras (*features*), dado a classe. Ou seja, assume-se que o valor de uma feature x_i é independente do valor de qualquer outra feature x_j (para $i \neq j$), uma vez que a classe y é conhecida. Essa suposição, embora forte e frequentemente irrealista em dados do mundo real, permite simplificar drasticamente o cálculo da verossimilhança $P(x|y)$:

$$P(x|y) = P(x_1, x_2, \ldots, x_n|y) \approx \prod_{i=1}^{n} P(x_i|y) \tag{2.35}$$

Substituindo isso na fórmula de Bayes e ignorando o denominador $P(x)$, a regra de decisão do *Naive Bayes* para classificar uma nova instância x se torna escolher a classe y que maximiza o produto:

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i|y) \tag{2.36}$$

Onde $\hat{y}$ é a classe predita. Durante o treinamento, o algoritmo essencialmente "aprende"as probabilidades a priori $P(y)$ e as probabilidades condicionais $P(x_i|y)$ a partir das frequências observadas nos dados de treinamento.

A forma como as probabilidades condicionais $P(x_i|y)$ são estimadas depende do tipo de feature x_i:

- **Gaussian Naive Bayes:** Para *features* contínuas, assume-se que $P(x_i|y)$ segue uma distribuição Gaussiana (Normal). A média e o desvio padrão para cada classe y e feature x_i são estimados a partir dos dados de treino.
- **Multinomial Naive Bayes:** Frequentemente usado para features discretas que representam contagens (ex: contagem de palavras em classificação de texto). Assume-se uma distribuição Multinomial para os vetores de *features*, e as probabilidades $P(x_i|y)$ são estimadas pelas frequências relativas suavizadas.
- **Bernoulli Naive Bayes:** Usado quando as features são binárias (ex: indicando a presença ou ausência de uma palavra). Assume uma distribuição de Bernoulli para cada $P(x_i|y)$.

Entre as **vantagens** do *Naive Bayes* estão sua simplicidade conceitual e de implementação, e sua alta velocidade e eficiência computacional, tanto no treinamento (que envolve basicamente contagem de frequências) quanto na classificação. Ele funciona bem mesmo com datasets de alta dimensionalidade e requer relativamente poucos dados de treinamento para estimar os parâmetros necessários. Por isso, é uma escolha viável para aplicações em tempo real e como um bom baseline inicial.

As **limitações**, no entanto, decorrem principalmente da sua suposição "ingênua":

- **Suposição de Independência:** Como mencionado, essa suposição raramente é válida no mundo real. Se as features forem altamente correlacionadas, o NB pode "contar dobrado"a evidência, levando a predições excessivamente confiantes e potencialmente incorretas. Apesar disso, o NB frequentemente apresenta bom desempenho mesmo quando a suposição é violada.
- **Problema da Frequência Zero (*Zero Frequency*):** Se um valor de *feature* x_i específico nunca foi observado junto com uma classe y no treinamento, a probabilidade condicional estimada $P(x_i|y)$ será zero. Devido à natureza multiplicativa da fórmula (Eq. 2.36), isso zeraria toda a probabilidade posterior $P(y|x)$, independentemente dos valores das outras *features*, o que é indesejável. Para mitigar isso, utiliza-se a **suavização (*smoothing*)**, sendo a mais comum a **Suavização de Laplace** (ou *add-one smoothing*). Ela consiste em adicionar uma pequena constante α (tipicamente $\alpha = 1$) à contagem de cada combinação feature-classe e ajustar o denominador apropriadamente, garantindo que nenhuma probabilidade seja exatamente zero.

Apesar de suas limitações, o *Naive Bayes* continua sendo um algoritmo popular e eficaz para muitas tarefas de classificação, especialmente em processamento de linguagem

natural (classificação de texto, filtragem de spam) e sistemas de recomendação. Sua combinação de simplicidade, eficiência e desempenho surpreendentemente robusto o consolida como uma ferramenta importante no campo do aprendizado de máquina.

2.10.9 AdaBoost (Adaptive Boosting)

O *AdaBoost* (*Adaptive Boosting*), introduzido por Freund e Schapire, é um dos algoritmos de *ensemble* mais influentes e estudados, projetado para aumentar a precisão de modelos preditivos através da combinação adaptativa de múltiplos classificadores fracos (*weak learners*) em um único classificador forte e robusto [28]. A ideia central é construir sequencialmente classificadores fracos (como árvores de decisão simples, chamadas de *decision stumps*, ou árvores um pouco mais profundas), onde cada novo classificador foca nos exemplos que foram classificados incorretamente pelos classificadores anteriores.

O algoritmo funciona atribuindo pesos w_i a cada exemplo de treinamento (x_i, y_i) (com $y_i \in \{-1, +1\}$) e ajustando esses pesos iterativamente. Inicialmente, todos os exemplos recebem pesos iguais, $w_i^{(1)} = 1/n$, onde n é o número total de amostras. Em cada iteração $t = 1, \ldots, T$:

1. Um classificador fraco $h_t(x) : \mathcal{X} \to \{-1, +1\}$ é treinado usando o conjunto de treinamento com os pesos atuais $D_t = (w_1^{(t)}, \ldots, w_n^{(t)})$. O objetivo do aprendiz fraco é encontrar um classificador h_t que minimize o erro ponderado:

$$\epsilon_t = \sum_{i=1}^{n} w_i^{(t)} 1(h_t(x_i) \neq y_i) = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i] \tag{2.37}$$

   onde $1(\cdot)$ é a função indicadora. É crucial a **hipótese do aprendizado fraco**: assume-se que o algoritmo base consegue encontrar um h_t com erro $\epsilon_t < 0.5$ em cada iteração.

2. Calcula-se um peso α_t para o classificador h_t, que mede sua importância ou confiabilidade, baseado no seu erro ϵ_t:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{2.38}$$

   Note que $\alpha_t > 0$ se $\epsilon_t < 0.5$. Classificadores com menor erro recebem maior peso.

3. Os pesos dos exemplos são atualizados para a próxima iteração $t + 1$, aumentando o peso dos exemplos classificados incorretamente por h_t e diminuindo o peso dos

classificados corretamente:

$$w_i^{(t+1)} = \frac{w_i^{(t)} \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \tag{2.39}$$

onde Z_t é um fator de normalização (a soma dos pesos não normalizados) que garante que D_{t+1} seja uma distribuição de probabilidade ($\sum_i w_i^{(t+1)} = 1$). O termo $y_i h_t(x_i)$ é -1 se a classificação for incorreta e +1 se for correta.

Após T iterações, a predição final do *AdaBoost* para uma nova instância x é obtida pela combinação dos classificadores fracos ponderados por seus respectivos pesos α_t, geralmente tomando o sinal da soma ponderada:

$$H(x) = \text{sign}\left(F(x)\right) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{2.40}$$

Pode-se mostrar que o algoritmo *AdaBoost* minimiza iterativamente uma **função de perda exponencial** sobre os dados de treinamento: $L_{exp}(F) = \sum_{i=1}^{n} \exp(-y_i F(x_i))$. As regras específicas para calcular α_t e atualizar os pesos w_i derivam diretamente da otimização desta função de perda de forma aditiva e gulosa [28, 29].

Entre as **vantagens** do *AdaBoost*, destacam-se sua simplicidade relativa de implementação, sua capacidade comprovada de melhorar significativamente a precisão de classificadores fracos (transformando-os em um classificador forte), e o fato de requerer poucos hiperparâmetros para ajuste (principalmente o número de iterações T e a complexidade do classificador fraco). Além disso, possui garantias teóricas de convergência do erro de treinamento a zero sob a hipótese do aprendizado fraco [28].

No entanto, o *AdaBoost* não está isento de **limitações**. Sua principal desvantagem é a **sensibilidade a ruído e outliers** nos dados de treinamento. Como o algoritmo foca progressivamente nos exemplos mais difíceis (aqueles classificados incorretamente), exemplos ruidosos ou com rótulos incorretos podem receber pesos muito altos, levando o algoritmo a se ajustar excessivamente a eles (*overfitting*) e prejudicando a generalização [30, 28].

Curiosamente, apesar da sensibilidade a ruído, *AdaBoost* demonstra em muitos casos uma notável resistência ao overfitting mesmo com muitas iterações. Uma explicação para isso vem da teoria das margens, que sugere que *AdaBoost* tende a aumentar a margem de confiança das classificações nos dados de treino, o que está associado a uma melhor generalização [28, 31].

Apesar dessa sensibilidade, o *AdaBoost* permanece uma técnica de *ensemble* poderosa e versátil, amplamente utilizada em diversas áreas e frequentemente servindo como benchmark para outros algoritmos de *boosting*.

2.10.10 LightGBM

O *LightGBM* (*Light Gradient Boosting Machine*) é um framework de *gradient boosting* de código aberto desenvolvido pela Microsoft, projetado especificamente para ser altamente eficiente e escalável, especialmente em conjuntos de dados grandes com alta dimensionalidade [32]. Como outras técnicas de *GBDT (Gradient Boosting Decision Tree)*, ele constrói um modelo preditivo forte combinando iterativamente árvores de decisão fracas. No entanto, o *LightGBM* introduz otimizações significativas para acelerar o processo de treinamento sem sacrificar (ou com perda mínima de) acurácia.

O principal gargalo computacional em implementações tradicionais de GBDT, mesmo as baseadas em histogramas, reside na necessidade de escanear todos os dados para cada feature ao procurar o melhor ponto de divisão, um processo custoso quando o número de instâncias (N) e/ou *features* (D) é grande [32]. Para resolver isso, o *LightGBM* implementa duas técnicas inovadoras principais:

1. **Gradient-based One-Side Sampling (GOSS):** Esta técnica foca em reduzir o número de instâncias de dados que precisam ser consideradas ao calcular o ganho de informação para uma divisão. A intuição é que instâncias com gradientes maiores (em valor absoluto) contribuem mais para o cálculo do ganho, pois representam exemplos onde o modelo atual ainda comete erros significativos (instâncias "subtreinadas"). O GOSS, então, mantém todas as instâncias com gradientes grandes (uma fração a do total) e seleciona aleatoriamente uma amostra das instâncias com gradientes pequenos (uma fração b das restantes). Para compensar o viés introduzido na distribuição dos dados, as instâncias amostradas com gradientes pequenos recebem um peso maior (multiplicadas por $(1-a)/b$) ao calcular o ganho de informação. Dessa forma, o GOSS consegue uma estimativa acurada do ganho usando um subconjunto significativamente menor de dados, focando nos exemplos mais informativos [32].

2. **Exclusive Feature Bundling (EFB):** Esta técnica visa reduzir o número efetivo de features que precisam ser avaliadas. Ela explora a esparsidade comum em dados de alta dimensão, onde muitas features são mutuamente exclusivas ou quase exclusivas (i.e., raramente possuem valores não nulos simultaneamente para a mesma instância, como *features one-hot*). O EFB agrupa essas features exclusivas em um único "pacote"(*bundle*). Isso é feito de forma que os valores originais de cada *feature* dentro do pacote possam ser identificados, geralmente adicionando um *offset* aos intervalos de valores discretizados (bins) de cada feature dentro do pacote. Por exemplo, se a *feature* A usa bins $[0, 10)$ e a *feature* B (exclusiva de A) usa bins $[0, 20)$, B pode ter seus valores deslocados para $[10, 30)$, e ambas podem ser representadas

por um único pacote com bins $[0, 30)$. Isso reduz a complexidade do cálculo do histograma de $O(N \times D)$ para $O(N \times \#\text{bundles})$, onde o número de pacotes é muito menor que D. Encontrar o agrupamento ótimo é NP-difícil, mas o *LightGBM* usa um algoritmo guloso eficiente (baseado em contagem de não-zeros ou em coloração de grafos) para criar os pacotes, permitindo até mesmo uma pequena taxa de conflito (features não totalmente exclusivas no mesmo pacote) para reduzir ainda mais o número de pacotes, com impacto mínimo na acurácia [32].

Além dessas duas técnicas principais, o LightGBM se baseia fundamentalmente no **algoritmo baseado em histograma** para encontrar os pontos de divisão, que já é mais eficiente em termos de memória e velocidade do que algoritmos baseados em pré-ordenação (como o usado no CART original ou na opção *exact* do *XGBoost*) [32]. Outra característica frequentemente associada ao *LightGBM* (embora não exclusiva) é a estratégia de crescimento da árvore **leaf-wise (por folha)**, em contraste com a estratégia *level-wise* (por nível). No crescimento *leaf-wise*, a árvore é expandida dividindo-se o nó folha que resulta na maior redução da perda, o que pode levar a uma convergência mais rápida e modelos mais precisos, mas requer controle de hiperparâmetros como 'maxdepth' para evitar overfitting.

As **vantagens** do *LightGBM* são principalmente sua alta velocidade de treinamento, baixo consumo de memória, boa acurácia (comparável a outros GBDTs de ponta como *XGBoost*), suporte a computação paralela e distribuída, e capacidade de lidar com dados em grande escala [32].

As **considerações** ao usar *LightGBM* incluem a necessidade de ajustar seus hiperparâmetros específicos (como as taxas de amostragem a e b para GOSS, parâmetros de profundidade/folhas para controlar o crescimento *leaf-wise*) e o fato de que, para datasets muito pequenos, os ganhos de eficiência podem não ser tão pronunciados ou as otimizações (como GOSS) podem até ser menos benéficas em comparação com métodos que usam todos os dados.

Em resumo, *LightGBM* é um framework de GBDT otimizado para velocidade e eficiência, especialmente adequado para grandes volumes de dados e/ou alta dimensionalidade, alcançando esses ganhos através das técnicas inovadoras GOSS e EFB, mantendo alta acurácia preditiva.

2.10.11 SGD Classifier (Stochastic Gradient Descent)

O *Stochastic Gradient Descent (SGD)* não é um modelo de classificação em si, mas sim um algoritmo de otimização amplamente utilizado para treinar uma vasta gama de modelos de aprendizado de máquina, incluindo modelos lineares e redes neurais profundas [33]. Sua

eficiência, especialmente em datasets de grande escala, o torna uma escolha fundamental em muitas aplicações práticas.

A ideia central do SGD é atualizar os parâmetros do modelo ($\mathbf{w}$) de forma iterativa, movendo-os na direção oposta ao gradiente da função de perda (L), calculado sobre um subconjunto selecionado aleatoriamente dos dados de treinamento (um *mini-batch*) a cada passo. A regra de atualização básica pode ser expressa como:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w}; x_i, y_i)$$

onde η é a taxa de aprendizado (*learning rate*), e $\nabla L(\mathbf{w}; x_i, y_i)$ é o gradiente da função de perda calculado para um único exemplo (x_i, y_i) ou um *mini-batch*.

No contexto deste trabalho, foi utilizado o `SGDClassifier` da biblioteca Scikit-learn. Esta implementação utiliza o SGD para treinar classificadores lineares sob diferentes funções de perda. Ao escolher a função de perda apropriada, o `SGDClassifier` pode replicar o comportamento de outros modelos lineares:

- **Hinge Loss:** Resulta em uma Máquina de Vetores de Suporte (SVM) linear.
- **Log Loss (modificada):** Resulta em uma Regressão Logística.
- Outras perdas como *modified Huber* também estão disponíveis.

O `SGDClassifier` é particularmente útil para problemas de grande escala (muitas amostras ou muitas *features*) onde treinar outros modelos pode ser computacionalmente proibitivo. Ele permite treinamento online (incremental) e é sensível ao escalonamento das *features* e à escolha da taxa de aprendizado e esquema de regularização (L1, L2 ou Elastic Net).

O estudo de [33] sobre a dinâmica do SGD no treinamento de redes neurais oferece insights relevantes, mesmo para modelos lineares treinados com SGD. Eles observam experimentalmente que, nas épocas iniciais do treinamento, o SGD tende a aprender funções de complexidade crescente [33]. Frequentemente, a melhoria inicial no desempenho pode ser largamente atribuída ao aprendizado de um classificador linear ou correlacionado a um classificador linear ótimo para os dados [33]. Além disso, o trabalho sugere que o SGD possui um "viés implícito"que pode favorecer soluções de menor complexidade, o que ajuda a explicar por que modelos treinados com SGD, mesmo em regimes sobreparametrizados, tendem a generalizar bem [33]. Essa característica de aprender padrões mais simples primeiro e reter essa informação mesmo ao se ajustar a padrões mais complexos ou ruído posteriormente [33] pode ser uma vantagem do uso do SGD como algoritmo de otimização.

A escolha dos hiperparâmetros do `SGDClassifier`, como a função de perda (*loss*), o tipo de regularização (*penalty*), a força da regularização (*alpha*), a taxa de aprendizado (*learning_rate*) e o número máximo de iterações (*max_iter*), é crucial e foi realizada através de *Randomized Search* neste trabalho, conforme detalhado na Metodologia.

Trabalho Correlatos

Para orientar o desenvolvimento deste projeto, foi conduzida uma revisão de literatura abrangente sobre implementações de Redes Neurais Convolucionais (CNNs) aplicadas à extração de *features* de imagens de raio-X. O foco esteve em arquiteturas como *VGG*, *ResNet* e *Inception*, que são amplamente reconhecidas por seu desempenho em tarefas de classificação de imagens. O objetivo dessa revisão foi identificar as soluções mais eficazes para a extração e classificação de características em imagens médicas, utilizando tanto algoritmos de aprendizado de máquina clássico quanto técnicas mais avançadas de aprendizado profundo. A análise comparativa entre essas arquiteturas contribuiu para estabelecer estratégias otimizadas para o trabalho proposto.

3.1 Comparative Analysis of COVID-19 X-ray Images Classification Using Convolutional Neural Networks, Transfer Learning, and Machine Learning Classifiers Using Deep Features

No trabalho desenvolvido por Rajagopal (2021), foi realizada uma análise comparativa entre diferentes abordagens de aprendizado profundo e aprendizado de máquina, com foco na classificação de imagens de raio-X de tórax de pacientes diagnosticados com COVID-19. O estudo explorou técnicas de classificação, incluindo Redes Neurais Convolucionais (CNNs), *Transfer Learning* e classificadores tradicionais de aprendizado de máquina integrados a *features* extraídas de camadas profundas de CNNs.

A metodologia proposta utilizou modelos como *CNN, SVM e XGBoost*, aplicados a um conjunto de dados de raio-X para identificar três categorias: COVID-19, pneumonia e condições saudáveis. As *features* extraídas pelas CNNs foram empregadas como entrada para os modelos de aprendizado de máquina, evidenciando maior precisão e robustez em comparação aos métodos de classificação tradicionais.

Os resultados destacaram o modelo SVM combinado com *features* extraídas de CNNs como a abordagem mais eficaz. Este modelo alcançou métricas de desempenho notáveis: precisão de 95,27%, *recall* de 94,52% e *F1-score* de 94,94%. Esses resultados evidenciam a eficácia da integração entre aprendizado profundo e aprendizado de máquina em tarefas de classificação de imagens médicas.

O estudo também trouxe uma contribuição relevante ao propor a combinação dessas técnicas para triagens mais rápidas e eficazes de pacientes com COVID-19. Essa abordagem é especialmente valiosa em contextos onde métodos tradicionais, como o teste RT-PCR, enfrentam limitações logísticas ou apresentam longos tempos de resposta. A solução proposta surge como uma alternativa promissora para auxiliar sistemas de saúde durante crises sanitárias.

Rajagopal (2021) conclui que a integração de aprendizado profundo com aprendizado de máquina melhora significativamente a detecção precoce de COVID-19 por meio de imagens de raio-X. Além disso, essa abordagem oferece soluções tecnológicas avançadas para enfrentar desafios relacionados a pandemias, evidenciando o potencial dessas técnicas para transformar o diagnóstico médico e contribuir para a evolução de sistemas de saúde mais rápidos e precisos.

Embora ambos os estudos explorem a classificação de imagens de raio-X para a detecção de COVID-19, o trabalho conduzido por Rajagopal (2021) concentrou-se na análise comparativa de diferentes abordagens de aprendizado profundo e aprendizado de máquina, utilizando modelos como *CNNs, SVM e XGBoost*, com foco na integração de *features* extraídas de CNNs em classificadores tradicionais. Em contrapartida, o presente trabalho aprofunda-se na avaliação detalhada de arquiteturas específicas de Redes Neurais Convolucionais (*VGG16, VGG19, ResNet50, InceptionV3 e EfficientNetV2L*), explorando suas capacidades individuais para a extração de *features* em imagens médicas, além de realizar uma análise mais abrangente do impacto de cada arquitetura no desempenho de classificadores clássicos e de *deep features*. Ademais, a utilização do *COVID-19 Radiography Database*, que inclui um conjunto de dados maior e mais diversificado em relação a quantidade de classes na classificação, e a implementação de técnicas avançadas de pré-processamento e normalização, oferecem uma base metodológica mais robusta, voltada para otimizar a generalização e a precisão dos modelos.

3.2 Performance Evaluation of the Deep Learning Based Convolutional Neural Network Approach for the Recognition of Chest X-Ray Images

O artigo de Sharma et al. (2022), intitulado *"Performance Evaluation of the Deep Learning Based Convolutional Neural Network Approach for the Recognition of Chest X-Ray Images"*, apresenta uma análise detalhada sobre a aplicação de Redes Neurais Convolucionais (CNNs) na detecção de pneumonia em imagens de raio-X do tórax. A pneumonia, uma infecção pulmonar potencialmente letal, é uma das principais causas de mortalidade em crianças menores de cinco anos, especialmente em regiões com acesso limitado a cuidados médicos. O diagnóstico precoce desempenha um papel crucial na redução de complicações graves, como insuficiência respiratória.

Neste estudo, os autores desenvolveram e implementaram um modelo baseado em CNN para classificar imagens de raio-X em duas categorias: saudáveis e pneumônicas. Utilizando um conjunto de dados público contendo 10.000 imagens (5.000 de cada classe), o modelo foi treinado e validado com uma divisão de 80% para treinamento e 20% para teste. A análise de desempenho do modelo revelou que ele alcançou uma precisão de validação de até 98% com o otimizador *RMSprop*, superando o otimizador *Adam*, que obteve 97% de precisão. Além disso, o *RMSprop* também apresentou a maior média de precisão (90%) e a menor perda média (0,33), evidenciando sua superioridade em relação ao *Adam* neste contexto.

Os autores destacaram a importância de grandes volumes de dados no treinamento de modelos para o reconhecimento e classificação de imagens médicas. Eles também enfatizaram o uso de técnicas de processamento seletivo, que concentram a análise em regiões relevantes das imagens, otimizando o tempo de processamento e aumentando a eficiência dos modelos.

Na conclusão, Sharma et al. (2022) apontam que o *RMSprop* demonstrou desempenho superior ao *Adam* na tarefa de classificação de imagens de raio-X. O estudo destaca o papel fundamental das CNNs no aprimoramento da detecção de pneumonia, especialmente em cenários clínicos e de saúde pública onde diagnósticos rápidos e precisos podem salvar vidas. Os autores sugerem que o avanço contínuo das tecnologias de aprendizado profundo, aliado à disponibilidade crescente de conjuntos de dados, pode melhorar ainda mais a eficácia e a precisão dos diagnósticos médicos.

Por fim, os autores reconhecem que suas contribuições originais residem na avaliação comparativa entre diferentes otimizadores e na aplicação prática das CNNs em imagens médicas. Eles recomendam que futuros estudos explorem formas de reduzir o tempo de processamento e otimizar ainda mais a eficácia dos modelos de aprendizado profundo na

análise de imagens médicas, abrindo novas possibilidades para diagnósticos mais rápidos e acessíveis.

Uma comparação entre o trabalho de Sharma et al. (2022) e o presente estudo revela semelhanças e distinções significativas em seus objetivos e abordagens. Ambos os trabalhos utilizam Redes Neurais Convolucionais (CNNs) como a principal ferramenta para a análise de imagens de raio-X, com foco na classificação de patologias pulmonares. No entanto, enquanto Sharma et al. concentram-se exclusivamente na detecção de pneumonia e na avaliação de diferentes otimizadores, este trabalho expande o escopo ao comparar múltiplas arquiteturas de CNNs (como *ResNet50, VGG16/VGG19, EfficientNetV2L e InceptionV3*) e analisar o desempenho de diversos classificadores de *Machine Learning*, como SVM e Regressão Logística, aplicados a *features* extraídas.

Além disso, enquanto Sharma et al. enfatizam a utilização de um único conjunto de dados com duas classes (saudáveis e pneumônicas) e o impacto dos otimizadores na precisão do modelo, este trabalho explora uma abordagem mais abrangente, que inclui a avaliação comparativa de classificadores tradicionais e a utilização de técnicas de extração de *features* para maximizar a representatividade dos dados. Os resultados obtidos neste estudo também destacam a capacidade das CNNs de fornecer *features* altamente discriminativas, permitindo que modelos como SVMs alcancem desempenhos competitivos. Assim, enquanto Sharma et al. enfatizam a otimização de hiperparâmetros e técnicas de treinamento, este trabalho contribui com uma análise mais ampla, incluindo aspectos como a eficiência computacional e a adequação de diferentes arquiteturas de CNNs a tarefas variadas de classificação médica.

Metodologia

A metodologia adotada neste trabalho foi estruturada para garantir uma análise completa da eficácia das arquiteturas de Redes Neurais Convolucionais (CNNs) na extração de *características* de imagens de raio-X, além da comparação do desempenho de múltiplos algoritmos classificadores sobre essas características. Os detalhes de cada etapa são apresentados a seguir.

4.1 Dataset Utilizado

Para a realização deste trabalho, utilizou-se o **COVID-19 Radiography Database**, um conjunto de dados público amplamente reconhecido e disponível na plataforma Kaggle[1]. O dataset foi desenvolvido por uma equipe de pesquisadores das universidades do Catar, Bangladesh, Paquistão e Malásia, em colaboração com médicos de diversas instituições. Este banco de dados contém imagens de raios-X de tórax de alta qualidade, representando quatro classes distintas: COVID-19, Normal (sem achados pulmonares), Opacidade Pulmonar (infecções não relacionadas à COVID-19) e Pneumonia Viral.

4.1.1 Composição e Exemplos

O *COVID-19 Radiography Database* reuniu imagens de diversas fontes públicas e foi atualizado incrementalmente. A versão utilizada neste trabalho contém a seguinte distribuição de imagens:

- **COVID-19:** 3.616 imagens. Exemplo: Figura 4.1.
- **Normal:** 10.192 imagens. Exemplo: Figura 4.2.
- **Opacidade Pulmonar (*Lung Opacity*):** 6.012 imagens. Exemplo: Figura 4.3.
- **Pneumonia Viral:** 1.345 imagens. Exemplo: Figura 4.4.

[1]Dataset URL: `https://www.kaggle.com/datasets/tawsifurrahman/covid19-radiography-database`

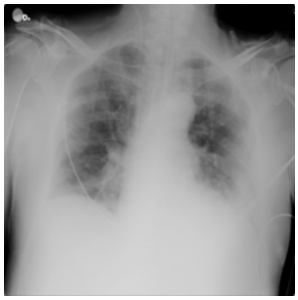


Figura 4.1: Exemplo: COVID-19.



Figura 4.2: Exemplo: Normal.

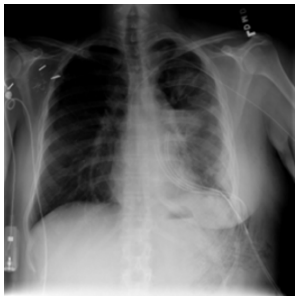


Figura 4.3: Exemplo: Opacidade.

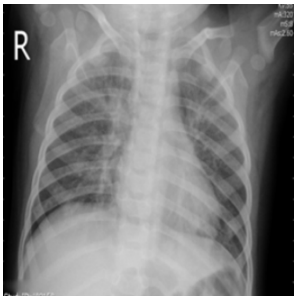


Figura 4.4: Exemplo: Pneumonia Viral.

4.1.2 Análise de Desbalanceamento e Mitigação

A distribuição das amostras no conjunto de dados completo apresenta um considerável desbalanceamento entre as classes, como ilustrado na Figura 4.5. A classe *Normal* é a mais representativa, enquanto *Pneumonia Viral* é significativamente minoritária.

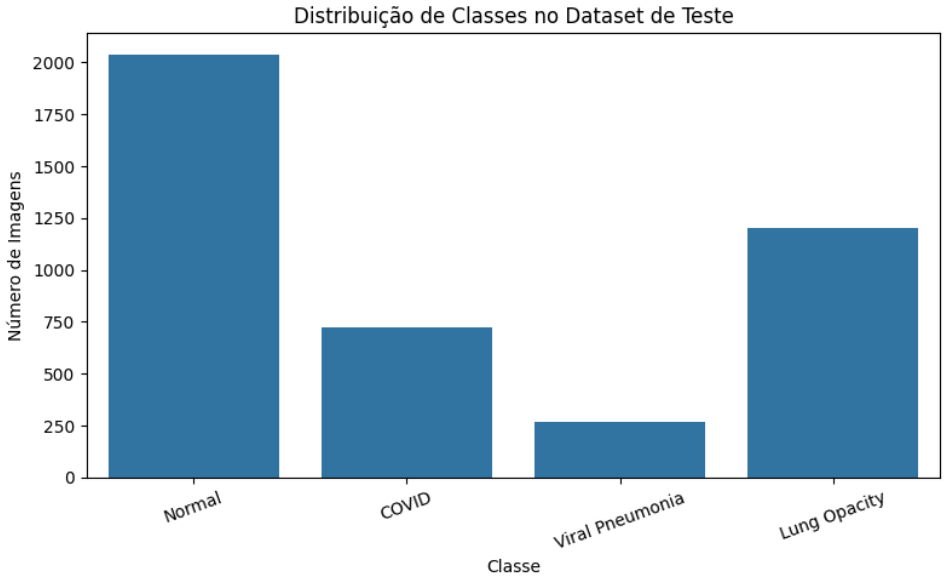


Figura 4.5: Distribuição aproximada de amostras por classe no dataset.

Esse desequilíbrio pode enviesar o treinamento dos modelos e levar a métricas de avaliação enganosas. Para mitigar esses efeitos:

- **Métrica de Avaliação Principal:** Adotou-se o **F1-Score Ponderado (*Weighted F1-Score*)** como principal métrica para seleção de hiperparâmetros e avaliação final, devido à sua robustez em cenários desbalanceados.

- **Validação Cruzada Estratificada:** Utilizou-se validação cruzada estratificada para garantir que a proporção original das classes fosse mantida em cada fold durante a otimização de hiperparâmetros e na avaliação final de CV.
- **Divisão Estratificada:** A divisão inicial do conjunto de dados em conjuntos de treino, validação e teste também foi realizada de forma estratificada.

4.2 Pré-processamento das Imagens

Todas as imagens do conjunto de dados foram fornecidas no formato PNG com resolução padronizada de 299x299 pixels e em escala de cinza. O pré-processamento aplicado antes da extração de características consistiu nas seguintes etapas:

- **Conversão de Cores:** As imagens em escala de cinza foram convertidas para o formato RGB (3 canais), replicando o canal único.
- **Redimensionamento:** As imagens foram redimensionadas para as dimensões de entrada exigidas por cada arquitetura CNN: 224x224 (*VGG16/19*, *ResNet50*), 299x299 (*InceptionV3*) ou 480x480 (*EfficientNetV2L*).
- **Normalização Específica da Arquitetura:** Aplicou-se a função de pré-processamento específica (*preprocess_input*) fornecida pela biblioteca Keras para cada modelo carregado (*ResNet, VGG, Inception, EfficientNet*), que ajusta os valores de pixel conforme esperado pelo modelo pré-treinado no *ImageNet*.
- **Divisão dos Dados:** O conjunto de dados completo foi dividido de forma estratificada, utilizando a função *train_test_split* do *Scikit-learn* com semente aleatória (*random_state=42*) para reprodutibilidade, nas seguintes proporções:
  - Conjunto de Treinamento: **60%** das imagens.
  - Conjunto de Teste: **20%** das imagens (utilizado para a avaliação final dos modelos).
  - Conjunto de Validação: **20%** das imagens (utilizado durante a otimização de hiperparâmetros dos classificadores).

4.3 Extração de Características via CNNs

O núcleo da primeira parte da metodologia consistiu em utilizar diferentes arquiteturas CNN como extratores de características (*feature extractors*) das imagens de raio-X pré-processadas.

4.3.1 Framework e Ferramentas

A implementação foi realizada em Python, utilizando principalmente as bibliotecas **TensorFlow** e sua API de alto nível **Keras**. A aceleração do processamento foi garantida pelo uso de Unidades de Processamento Gráfico (GPUs), gerenciadas conforme configuração inicial do script.

4.3.2 Arquiteturas Selecionadas

Conforme detalhado no capítulo de Fundamentação Teórica, as seguintes arquiteturas CNN, pré-treinadas na base de dados ImageNet, foram selecionadas: *VGG16, VGG19, ResNet50, InceptionV3 e EfficientNetV2L*. Os pesos pré-treinados foram utilizados, aplicando a técnica de *transfer learning*.

4.3.3 Processo de Extração

Para cada uma das arquiteturas selecionadas:

1. O modelo pré-treinado foi carregado utilizando Keras, excluindo-se as camadas superiores de classificação (`include_top=False`).
2. As imagens pré-processadas (dos conjuntos de treino, validação e teste) foram passadas através da rede convolucional base.
3. Os vetores de características foram extraídos da saída das seguintes camadas específicas para cada arquitetura, conforme identificado no código implementado:
    - **ResNet50:** Saída da camada *Global Average Pooling* (identificada como *avg_pool*).
    - **VGG16:** Saída da primeira camada totalmente conectada (identificada como *fc1*).
    - **VGG19:** Saída da primeira camada totalmente conectada (identificada como *fc1*).
    - **InceptionV3:** Saída da camada *Global Average Pooling* (identificada como *avg_pool*).
    - **EfficientNetV2L:** Saída da camada *Global Average Pooling* (identificada como *avg_pool*).
4. Como passo final do processo de extração, os vetores de características resultantes para cada imagem foram **normalizados utilizando a norma L2**, garantindo que todos os vetores tivessem magnitude unitária antes de serem utilizados pelos classificadores.

5. Os vetores de características normalizados foram armazenados em arquivos (CSV, Parquet, Numpy), juntamente com seus rótulos de classe correspondentes, separadamente para os conjuntos de treino, validação e teste.

Uma classe `FeatureExtractor` em Python foi desenvolvida para automatizar este processo.

4.3.4 Justificativa da Escolha da Camada para Extração de Características

A seleção da camada específica para a extração de características em cada arquitetura CNN (*e.g.*, *avg_pool* para *ResNet50, InceptionV3, EfficientNetV2L e 'fc1' para VGG16/VGG19*) foi guiada por abordagens comuns na literatura de *transfer learning* e pela natureza hierárquica da representação aprendida por essas redes. Optou-se por camadas que tipicamente precedem a classificação final no modelo original pré-treinado no *ImageNet*, assumindo que estas capturam informações semânticas de alto nível, agregadas a partir das características de níveis inferiores (bordas, texturas, partes de objetos) detectadas nas camadas convolucionais anteriores. A camada `Global Average Pooling` (*avg_pool*), em particular, é frequentemente usada por sua capacidade de resumir os mapas de características espaciais em um vetor de tamanho fixo, robusto a variações espaciais. A camada 'fc1' nas *VGGs* representa uma transformação densa dessas características agregadas. Embora características de camadas intermediárias pudessem oferecer representações alternativas (como visualizado no Apêndice B), a escolha por essas camadas finais visou maximizar a informação semântica para a tarefa de classificação subsequente. Investigações futuras poderiam analisar o impacto da fusão de características de múltiplas camadas ou da seleção de camadas intermediárias diferentes.

4.4 Classificação das Características Extraídas

Com os vetores de características gerados, foram treinados e avaliados diversos algoritmos de Machine Learning sobre essas representações.

4.4.1 Classificadores Avaliados

Os seguintes algoritmos classificadores foram implementados utilizando as bibliotecas *Scikit-learn, XGBoost, LightGBM e CatBoost*:

- AdaBoost Classifier
- CatBoost Classifier

- Decision Tree Classifier
- K-Nearest Neighbors Classifier
- LightGBM Classifier
- Logistic Regression
- Gaussian Naive Bayes
- Random Forest Classifier
- Support Vector Machine (SVC)
- XGBoost Classifier
- Stochastic Gradient Descent

4.4.2 Otimização de Hiperparâmetros

Para cada combinação de {Arquitetura CNN Extratora} × {Classificador}, foi realizada uma otimização dos hiperparâmetros do classificador.

- **Técnica:** Utilizou-se **Randomized Search com Validação Cruzada** (`RandomizedSearchCV` do Scikit-learn), configurada para testar **100 combinações** aleatórias de parâmetros (*n_iter=100*) para cada classificador. A busca foi realizada com paralelização (*n_jobs=6*) e semente aleatória (*random_state=42*) para reprodutibilidade.
- **Validação Cruzada (na busca):** Foi aplicada validação cruzada **estratificada com 3 folds** (*StratifiedKFold(n_splits=3)*) sobre o conjunto de *treinamento* das características extraídas.
- **Métrica de Otimização:** O critério para selecionar a melhor combinação de hiperparâmetros foi o **F1-Score Ponderado (*Weighted F1-Score*)**.
- **Treinamento Final do Melhor Modelo:** Após a busca, o modelo com a melhor combinação de hiperparâmetros foi retreinado utilizando todo o conjunto de treinamento das características extraídas.
- **Avaliação Adicional por CV:** Adicionalmente, para uma estimativa mais robusta do desempenho no conjunto de treinamento, o melhor modelo encontrado para cada classificador foi reavaliado usando validação cruzada estratificada com 5 folds (*cross_val_score* do *Scikit-learn* sobre os dados de treino), utilizando o *F1-Score* Ponderado como métrica.

4.4.3 Grades de Hiperparâmetros Exploradas

As grades de parâmetros utilizadas na busca via Randomized Search para cada classificador foram definidas conforme o arquivo de configuração 'config.yaml' e são apresentadas no Listagem 4.1.

```
# AdaBoost
'learning_rate': [0.001, 0.01, 0.1, 1.0]
'n_estimators': [50, 100, 200]

# CatBoost (Exemplo - par metros reais podem variar)
'depth': [6] # Distribui  es podem ser usadas em RandomizedSearch
'iterations': [200]
'learning_rate': [0.1]

# Decision Tree
'criterion': ['gini', 'entropy']
'max_depth': [5, 10, 20, None]
'min_samples_leaf': [1, 2, 4]
'min_samples_split': [2, 5, 10]

# K-Nearest Neighbors
'metric': ['euclidean', 'manhattan', 'cosine']
'n_neighbors': [1, 5, 10, 15]
'weights': ['uniform', 'distance']

# LightGBM (Exemplo - par metros reais podem variar)
'learning_rate': [0.01, 0.1]
'max_depth': [5, 10, 20]
'min_child_samples': [5, 10, 20]
'min_split_gain': [0.0, 0.1]
'n_estimators': [100, 200, 300]
'num_leaves': [31, 60, 120]

# Logistic Regression
'C': [0.1, 1, 10, 50] # Ou distribui  o como loguniform(0.1, 50)
'penalty': ['l1', 'l2']
'solver': ['liblinear']
'max_iter': [500, 1000, 2000] # Aumentado max_iter

# Naive Bayes (GaussianNB)
'var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6] # Par metro principal a variar

# Random Forest
'n_estimators': [50, 100, 200, 300] # Expandido
'max_features': ['sqrt', 'log2', None]
'min_samples_split': [2, 5, 10]
'min_samples_leaf': [1, 2, 4]
'max_depth': [10, 20, 30, None] # Expandido

# SGD Classifier
'loss': ['hinge', 'log_loss', 'modified_huber', 'perceptron']
'penalty': ['l2', 'l1', 'elasticnet']
'alpha': [0.0001, 0.001, 0.01]
'max_iter': [1000, 2000, 5000]
'learning_rate': ['optimal', 'adaptive'] # Adicionado 'adaptive'
```

```
'eta0': [0.001, 0.01] # Necess rio se learning_rate='adaptive'

# SVM
'C': [0.1, 1, 10, 100] # Ajustado C
'gamma': ['scale', 'auto', 0.1, 1, 0.01] # Expandido gamma
'kernel': ['linear', 'rbf']
'probability': [True]

# XGBoost (Exemplo - par metros reais podem variar)
'learning_rate': [0.01, 0.1, 0.3]
'max_depth': [3, 5, 7]
'n_estimators': [50, 100, 200]
'subsample': [0.8, 1.0]
'colsample_bytree': [0.8, 1.0]
'gamma': [0, 0.1, 0.5]
'reg_lambda': [1, 0.1, 10]
'reg_alpha': [0, 0.1, 1]
```

Listing 4.1: Grades de Hiperparâmetros Utilizadas no Randomized Search (baseado em config.yaml)

Nota: Para Randomized Search, em vez de listas explícitas, poderiam ter sido usadas distribuições de parâmetros (ex: 'loguniform', 'uniform') de onde os 100 conjuntos foram amostrados.

4.5 Avaliação de Desempenho

A avaliação final do desempenho dos melhores modelos otimizados foi realizada utilizando o conjunto de teste (20% dos dados originais), mantido intocado durante as fases de treinamento e otimização.

4.5.1 Métricas Utilizadas

Os modelos finais foram avaliados e comparados com base nas seguintes métricas, detalhadas no capítulo de Fundamentação Teórica:

- Acurácia Global
- F1-Score Ponderado (*Weighted F1-Score*) - Métrica principal
- Matriz de Confusão (para análise visual dos erros por classe)
- Precisão (média ponderada e por classe)
- Recall / Sensibilidade (média ponderada e por classe)
- Curva ROC AUC (*Area Under the Receiver Operating Characteristic Curve*, calculada com estratégia One-vs-Rest para multiclasse)

- Coeficiente de Correlação de Matthews (MCC)

Além dessas métricas no conjunto de teste, o *F1-Score* Ponderado médio obtido na validação cruzada final de 5 folds (sobre o conjunto de treino) também foi registrado como indicador de robustez.

Resultados

5.1 Resultados e Discussão

Nesta seção, apresentamos e discutimos os principais resultados obtidos a partir dos experimentos de extração de características com diferentes arquiteturas CNN e da subsequente classificação utilizando diversos algoritmos supervisionados. Conforme detalhado na Metodologia, devido ao desbalanceamento das classes no conjunto *COVID-19 Radiography*, o **F1-Score Ponderado (*Weighted F1-Score*)** foi utilizado como métrica principal para otimização de hiperparâmetros (via *Randomized Search* no conjunto de validação) e para a comparação geral do desempenho final, avaliado no conjunto de teste. Outras métricas como Acurácia, Precisão, *Recall* (Sensibilidade), ROC AUC e MCC também foram computadas para uma análise mais completa.

5.2 Desempenho na Extração de Características

5.2.1 Tempo de Execução das Arquiteturas CNN

O custo computacional para extrair as características das imagens do conjunto de teste variou significativamente entre as arquiteturas CNN, conforme medido no ambiente experimental e ilustrado na Figura 5.1. A *EfficientNetV2L* foi a mais lenta ($\approx$ 1046 segundos), enquanto *InceptionV3* ($\approx$ 69 s) e *ResNet50* ($\approx$ 87 s) foram as mais rápidas. As arquiteturas *VGG16* ($\approx$ 151 s) e *VGG19* ($\approx$ 185 s) tiveram custos intermediários. Esta análise temporal é relevante, mas precisa ser considerada em conjunto com a qualidade das *features* extraídas, avaliada pelo desempenho dos classificadores.

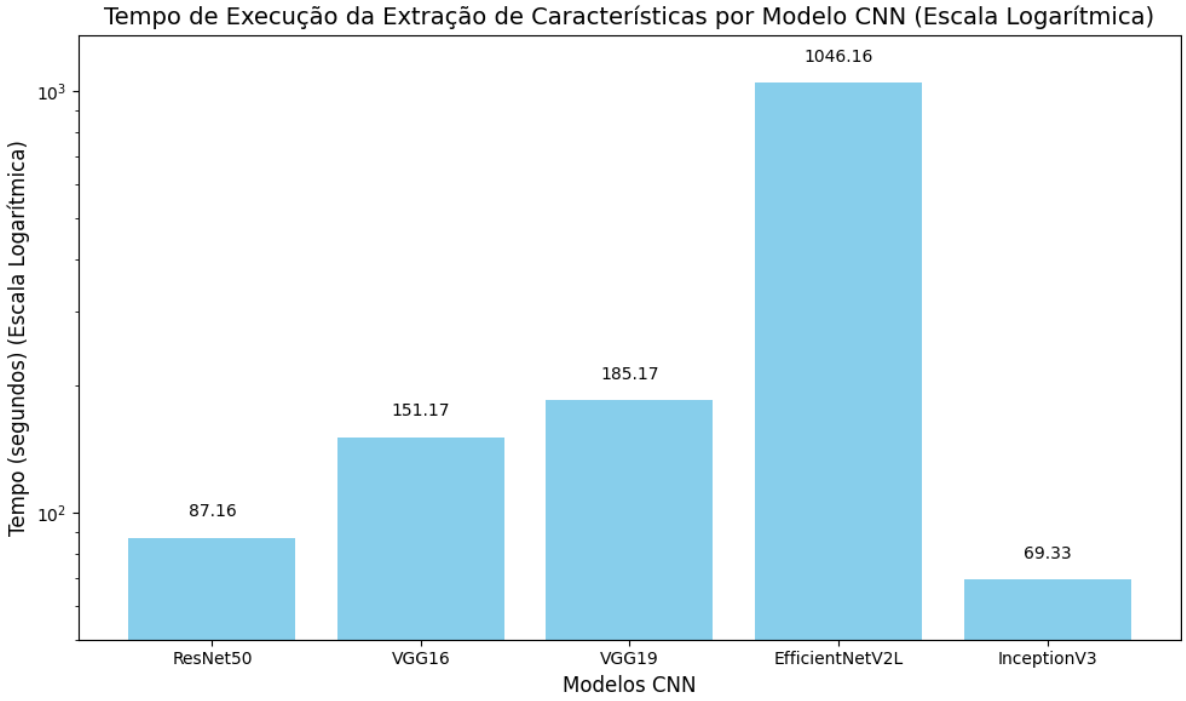


Figura 5.1: Tempo de execução da extração de características por modelo CNN (escala logarítmica).

5.3 Desempenho dos Classificadores

Os vetores de características extraídos por cada CNN foram usados para treinar e avaliar os 11 classificadores detalhados na Metodologia.

5.3.1 Seleção dos Melhores Modelos (Conjunto de Validação)

A Tabela 5.1 mostra as cinco melhores combinações encontradas durante a fase de otimização de hiperparâmetros, classificadas pelo *F1-Score* Ponderado no conjunto de validação (*Val_F1*). Estes resultados orientaram a seleção dos modelos mais promissores, mas não representam a performance final de generalização.

Tabela 5.1: Top 5 combinações por F1-Score Ponderado no conjunto de validação ('Val_F1'). (Valores e ordem corrigidos conforme CSV).

Arquivo Feature	**Classificador**	**Val_F1**	**Tempo Exec (s)**
EfficientNetV2L_features	SVM	0.9054	88.1600
VGG19_features	SVM	0.9052	1542.5200
ResNet50_features	SVM	0.8995	200.3000
EfficientNetV2L_features	Regressão Logística	0.8995	21.4000
VGG16_features	SVM	0.8983	1574.4200

Nota: Tempo de execução corresponde à soma dos tempos de treino e otimização/validação do classificador.

O SVM dominou os resultados de validação, especialmente com *features EfficientNetV2L* e *VGG19*. A Regressão Logística também se mostrou competitiva e muito eficiente.

5.3.2 Desempenho Final no Conjunto de Teste

A avaliação definitiva do desempenho foi realizada no conjunto de teste. A Tabela 5.2 apresenta as 10 melhores combinações {Extrator CNN + Classificador} classificadas pelo *F1-Score* Ponderado no teste (*Test_F1*).

Tabela 5.2: Top 10 combinações por F1-Score Ponderado no conjunto de teste ('Test$_F$1').

Arquivo Feature	**Classificador**	**Test_F1**	**Test_Acc**	**Test_Prec**	**Test_Rec**	**ROC-AUC**	**MCC**	**Tempo Exec (s)**
ResNet50_features	SVM	0.9085	0.9091	0.9102	0.9091	0.9837	0.8608	200.3000
VGG16_features	SVM	0.9058	0.9067	0.9073	0.9067	0.9818	0.8573	1574.4200
VGG16_features	Regressão Logística	0.8964	0.8973	0.8978	0.8973	0.9784	0.8425	17.2300
EfficientNetV2L_features	SVM	0.8955	0.8961	0.8957	0.8961	0.9791	0.8407	88.1600
EfficientNetV2L_features	Regressão Logística	0.8942	0.8949	0.8963	0.8949	0.9768	0.8386	21.4000
ResNet50_features	Regressão Logística	0.8930	0.8937	0.8956	0.8937	0.9771	0.8371	27.3800
VGG16_features	SGD	0.8922	0.8926	0.8927	0.8926	NaN	0.8352	502.9700
EfficientNetV2L_features	SGD	0.8893	0.8902	0.8928	0.8902	NaN	0.8312	157.3400
VGG19_features	SVM	0.8886	0.8890	0.8889	0.8890	0.9818	0.8296	1542.5200
ResNet50_features	SGD	0.8820	0.8831	0.8863	0.8831	0.9733	0.8203	247.1700

Nota: Tempo de execução refere-se à soma dos tempos de treino e otimização/validação do classificador, conforme detalhado na Tabela 5.1.

Os resultados finais no teste confirmam a alta performance da abordagem, com *F1-Scores* Ponderados superiores a 0.90 para as duas melhores combinações: **ResNet50+SVM** (F1=0.9085) e **VGG16+SVM** (F1=0.9058). Outros classificadores como Regressão Logística e SGD também alcançaram F1 $>$ 0.89 com *features* específicas. Nota-se que a

ordem dos melhores modelos mudou ligeiramente entre a validação e o teste, com **ResNet50+SVM** assumindo a liderança e **VGG19+SVM** caindo no ranking, reforçando a importância da avaliação no conjunto de teste.

5.3.3 Análise de Desempenho por Classe

Para compreender como os modelos de melhor desempenho lidam com as especificidades de cada diagnóstico, analisou-se a métrica *Recall* (Sensibilidade ou TPR) individualmente para cada classe. A Tabela 5.3 apresenta esses resultados para os 5 modelos com maior F1-Score de validação, utilizando o mapeamento de classes correto: Classe 0 (*Normal*), Classe 1 (*COVID*), Classe 2 (*Viral Pneumonia* - a classe minoritária) e Classe 3 (*Lung Opacity*).

Tabela 5.3: Recall (TPR - Taxa de Verdadeiros Positivos) por classe para os 5 melhores modelos da validação (com mapeamento correto).

Modelo (Extrator + Classificador)	**Normal**	**COVID**	**Viral Pneum**	**Opacity**
ResNet50_SVM	0.9436	0.9310	0.9629	0.8250
EffNetV2L_SVM	0.9240	0.9241	0.9815	0.8125
EffNetV2L_LR	0.9486	0.8690	0.9259	0.8125
VGG16_SVM	0.9486	0.9172	1.0000	0.8083
VGG19_SVM	0.9216	0.8828	0.9629	0.8208

A análise da Tabela 5.3 revela um panorama interessante:

- O desempenho nas classes **Normal** e **COVID** permaneceu alto, com *Recalls* consistentemente acima de 92% e 86%, respectivamente, nos modelos de topo.
- A classe minoritária, **Viral Pneumonia**, apresentou um **Recall extremamente alto** em todos os 5 melhores modelos, variando de 92.6% até um **recall perfeito de 100%** obtido pela combinação **VGG16+SVM**. Isso indica que, apesar de sua baixa representatividade no conjunto de dados, os modelos otimizados conseguiram aprender a identificar muito bem esta classe.
- Em contrapartida, a classe **Lung Opacity**, que possui mais instâncias que COVID e Viral Pneumonia, apresentou o **menor Recall** entre os modelos de topo, variando de 80.8% a 82.5% (média de 81.6%). Isso sugere que esta classe pode possuir características visuais mais desafiadoras ou sobreposição com outras classes, tornando sua identificação mais difícil para os modelos, mesmo os de melhor desempenho global.

Portanto, a dificuldade maior de classificação não residiu na classe minoritária (*Viral Pneumonia*), mas sim na classe *Lung Opacity*. A capacidade dos modelos de identificar a classe minoritária com alta sensibilidade é um resultado positivo, enquanto a performance na classe *Lung Opacity* indica um ponto para potencial melhoria futura ou análise mais aprofundada das características que a dificultam. Estes achados detalhados por classe complementam a avaliação geral fornecida pelo *F1-Score* Ponderado.

5.3.4 Melhor Classificador por Arquitetura CNN (Conjunto de Teste)

Confirmando a robustez do SVM, ele se manteve como o classificador de melhor desempenho no conjunto de teste para todas as cinco arquiteturas CNN extratoras, embora os *F1-Scores* variem dependendo das features (Tabela 5.4).

Tabela 5.4: Melhor F1-Score Ponderado no Teste (*Test_F1*) e classificador correspondente para cada Arquitetura Extratora.

Arquivo Feature	**Melhor Classificador (Teste)**	**Test_F1**
ResNet50_features	SVM	0.9085
VGG16_features	SVM	0.9058
EfficientNetV2L_features	SVM	0.8955
VGG19_features	SVM	0.8886
InceptionV3_features	SVM	0.8660

5.3.5 Tempo de Execução dos Classificadores

A análise do tempo de execução para treinamento e otimização dos classificadores (Figura 5.2) confirmou a grande variabilidade entre os métodos. **Naive Bayes** foi o mais rápido (média $\approx$ 1s), enquanto **LightGBM** foi o mais lento (média $>$ 3000s). **Regressão Logística** e **Random Forest/Decision Tree** apresentaram tempos muito baixos (média de 20-30s). O **SVM**, apesar de ser o melhor em desempenho, teve um custo computacional consideravelmente alto, especialmente com *features* VGG (média $>$ 700s, chegando a $>$ 1500s). **XGBoost** também se mostrou computacionalmente intensivo.

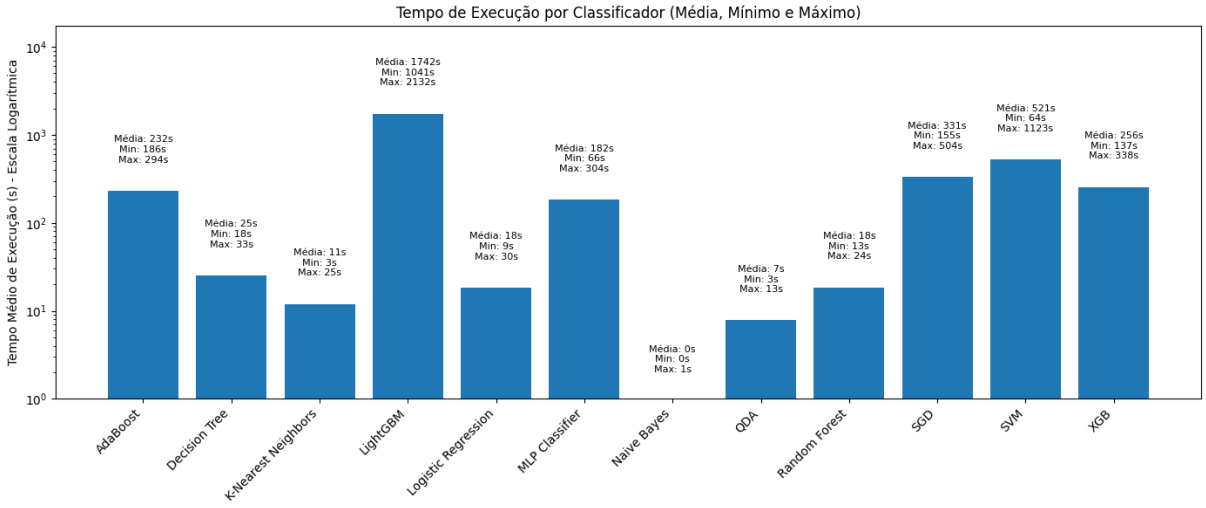


Figura 5.2: Tempo médio de execução por classificador (treino + otimização), indicando também os valores mínimo e máximo observados (escala logarítmica).

5.4 Discussão Geral dos Resultados

A análise combinada dos resultados de desempenho no conjunto de teste e do custo computacional permite extrair conclusões importantes sobre a abordagem proposta. A estratégia de usar CNNs pré-treinadas como extratores de características, seguida pela aplicação de classificadores otimizados, mostrou-se altamente eficaz para a classificação das imagens de raio-X do *dataset COVID-19 Radiography*, alcançando *F1-Scores* Ponderados superiores a 0.90 nas melhores combinações.

O classificador **SVM emergiu como o mais performático** em termos de F1-Score no teste, especialmente quando combinado com features extraídas pela **ResNet50** (melhor resultado geral, F1=0.9085) e **VGG16** (segundo melhor, F1=0.9058). Isso sugere que a capacidade do SVM de lidar com espaços de alta dimensão e encontrar margens ótimas foi particularmente adequada para as representações geradas por essas redes profundas. No entanto, o custo computacional do SVM, especialmente com as features VGG, foi elevado.

A **Regressão Logística** demonstrou ser uma alternativa **altamente competitiva e eficiente**. A combinação **VGG16+LR** obteve o terceiro melhor *F1-Score* no teste (0.8964) com um tempo de execução ordens de magnitude menor que o **SVM+VGG16**. Isso a posiciona como uma excelente opção quando o equilíbrio entre desempenho e eficiência é prioridade. O classificador **SGD** também apresentou resultados promissores e relativamente eficientes.

Em contraste, algoritmos baseados em árvores e ensembles complexos (**DT, RF, AdaBoost, XGBoost, LightGBM** não alcançaram o mesmo nível de desempenho

das melhores combinações **SVM/LR/SGD** nesta tarefa específica, e alguns deles (*XGBoost, LightGBM*) apresentaram os maiores custos computacionais. Isso pode indicar que as features extraídas pelas camadas profundas das CNNs já continham representações linearmente separáveis (ou separáveis por margem máxima) que beneficiaram mais os classificadores lineares ou baseados em margem.

A análise por classe (Tabela 5.3) oferece insights importantes. Confirma-se o alto desempenho para as classes **Normal** e **COVID**. A classe minoritária, **Pneumonia Viral**, apresentou um **Recall excelente** em todos os modelos de topo (entre 92,6% e 100%), indicando que, apesar da baixa representatividade, sua identificação não foi o principal desafio para estes modelos. A combinação VGG16+SVM, inclusive, atingiu um *Recall* perfeito para esta classe. Em contraste, a classe **Lung Opacity** foi a que apresentou maior dificuldade, com os *Recalls* mais baixos entre os modelos de topo (na faixa de 80,8% a 82,5%). O modelo ResNet50+SVM, campeão geral no teste, obteve o maior *Recall* (82,5%) para esta classe mais desafiadora (C3). Portanto, embora o desempenho geral seja alto, a análise por classe revela que a maior dificuldade relativa para os melhores modelos residiu na distinção da classe *Lung Opacity*, e não da classe minoritária *Viral Pneumonia*.

Considerando o **trade-off entre tempo de extração de features e desempenho final**, a *ResNet50* se destaca: teve um tempo de extração relativamente baixo (87s) e produziu as features que levaram ao melhor resultado de classificação final com SVM. A *VGG16*, embora com extração mais lenta (151s), também gerou features de alta qualidade para SVM e LR. A *EfficientNetV2L*, apesar do maior custo de extração (1046s), produziu features que levaram a bons resultados com SVM e LR, mas não os melhores no teste.

Em conclusão, para obter o máximo desempenho preditivo nesta tarefa, a combinação **ResNet50 (features) + SVM (classificador)** foi a melhor escolha. Para cenários que exigem maior **eficiência computacional** com desempenho ainda elevado, a **Regressão Logística combinada com features da VGG16 ou ResNet50** surge como uma alternativa muito atraente. A escolha dependerá, portanto, das prioridades e restrições da aplicação final.

Conclusão

Este trabalho teve como objetivo principal realizar uma análise comparativa da eficácia de diferentes arquiteturas de Redes Neurais Convolucionais (CNNs) pré-treinadas — *VGG16, VGG19, ResNet50, InceptionV3 e EfficientNetV2L* — como extratoras de características de imagens de raio-X do tórax para classificação diagnóstica (*COVID-19, Normal, Opacidade Pulmonar, Pneumonia Viral*). Adicionalmente, buscou-se comparar o desempenho de um amplo espectro de algoritmos classificadores de aprendizado de máquina aplicados sobre essas características extraídas, avaliando não apenas a acurácia preditiva (com foco no *F1-Score* Ponderado devido ao desbalanceamento do conjunto de dados), mas também a eficiência computacional das abordagens.

Para atingir esses objetivos, foi adotada uma metodologia rigorosa que envolveu o pré-processamento das imagens do *dataset* público *COVID-19 Radiography Database*, a extração de vetores de características a partir de camadas específicas das CNNs pré-treinadas no ImageNet (seguida de normalização L2), a otimização de hiperparâmetros para 11 classificadores distintos (incluindo *SVM, Regressão Logística, SGD, Naive Bayes, KNN, Árvore de Decisão, Random Forest, AdaBoost, XGBoost, LightGBM, CatBoost*) através de *Randomized Search* com validação cruzada estratificada, e a avaliação final do desempenho no conjunto de teste utilizando múltiplas métricas (*F1-Score, Acurácia, Precisão, Recall, ROC AUC, MCC*).

Os resultados obtidos demonstraram consistentemente a viabilidade e a eficácia da abordagem de *transfer learning* com CNNs para extração de características seguida por classificação com algoritmos de *machine learning*. As melhores combinações alcançaram *F1-Scores* Ponderados superiores a 0.90 no conjunto de teste, indicando alta capacidade discriminativa para as quatro classes do problema.

A análise comparativa revelou achados importantes:

- O classificador **Support Vector Machine (SVM)** destacou-se como o de melhor desempenho geral, liderando o ranking de *F1-Score* no teste quando combinado com features da maioria das arquiteturas CNN, com as combinações **ResNet50+SVM** (F1=0.9085) e **VGG16+SVM** (F1=0.9058) alcançando os melhores resultados absolutos.

- A **Regressão Logística** e o **SGD Classifier** surgiram como alternativas altamente competitivas e muito eficientes, obtendo *F1-Scores* próximos aos do SVM com certas *features (ex: VGG16+LR com F1*=0.8964), porém com um custo computacional significativamente menor.
- As características extraídas pela **ResNet50** e **VGG16** mostraram-se as mais eficazes para a classificação final com os melhores algoritmos. *Features* da *EfficientNetV2L* e *VGG19* também levaram a bons resultados, enquanto as da *InceptionV3* resultaram em desempenho consistentemente inferior nesta tarefa.
- A análise do **custo computacional** evidenciou um claro trade-off. A extração de *features* com *ResNet50* foi relativamente rápida e levou ao melhor desempenho, configurando um bom equilíbrio. A classificação com SVM, embora performática, foi computacionalmente custosa, especialmente com *features VGG*. Classificadores como Naive Bayes, Regressão Logística, DT e RF foram muito mais rápidos. *Ensembles* como LightGBM e XGBoost, apesar de complexos, não superaram SVM/LR/SGD neste cenário e tiveram alto custo.
- A análise de desempenho por classe, focando nos melhores modelos, indicou excelente capacidade de identificação para as classes Normal, COVID-19 e, surpreendentemente, para a classe minoritária Pneumonia Viral (Recall $> 92\%$). A classe que apresentou maior dificuldade relativa foi a Lung Opacity (Recall $\approx$ 81-83%), sugerindo que suas características podem ser mais ambíguas ou sobrepostas às demais.

Estes resultados têm implicações práticas para o desenvolvimento de sistemas de auxílio ao diagnóstico baseados em imagens de raio-X. Eles sugerem que a combinação de *transfer learning* (especialmente com ResNet50 ou VGG16) e classificadores robustos como SVM ou eficientes como Regressão Logística pode fornecer alta precisão. A escolha entre eles pode depender do balanço desejado entre máxima acurácia e velocidade/custo computacional.

Reconhecem-se, no entanto, algumas **limitações** neste estudo. A análise foi restrita a um único dataset público, cujas características podem não generalizar perfeitamente para outros contextos clínicos ou equipamentos. A extração de features foi feita a partir de uma única camada por CNN, sem explorar técnicas de fusão de features de múltiplas camadas. A otimização de hiperparâmetros, embora extensa (*Randomized Search*), não garante a descoberta do ótimo global. Além disso, o trabalho focou no desempenho preditivo, sem abordar a interpretabilidade dos modelos (XAI) ou realizar uma validação clínica.

Como **trabalhos futuros**, sugerem-se diversas direções:

- Avaliar a abordagem em outros *datasets* de raio-X ou outras modalidades de imagem médica.

- Explorar arquiteturas CNN mais recentes (ex: *Vision Transformers*) ou técnicas de *fine-tuning* completo das CNNs.
- Investigar métodos de fusão de características de diferentes camadas ou diferentes modelos (*ensembles* de extratores).
- Aprofundar a análise de erros, especialmente para a classe *Lung Opacity*, utilizando técnicas de visualização ou XAI para entender as falhas do modelo.
- Realizar testes de significância estatística para comparar mais formalmente o desempenho dos melhores modelos.
- Explorar técnicas para melhorar o desempenho em classes específicas, se necessário.
- Colaborar com profissionais da saúde para uma validação clínica dos modelos mais promissores.

Por fim, este trabalho contribuiu para a compreensão do desempenho relativo de diferentes combinações de extratores CNN e classificadores de *machine learning* na tarefa de classificação de imagens de raio-X do tórax para diagnóstico relacionado à COVID-19. Foram identificadas combinações de alta performance (*ResNet50/VGG16 + SVM*) e alternativas eficientes (LR, SGD), fornecendo subsídios para o desenvolvimento de ferramentas computacionais de apoio à decisão médica, sempre ressaltando a necessidade de validação e consideração do *trade-off* entre precisão e custo computacional no contexto da aplicação final.

Código Fonte Implementado

Este apêndice contém o código fonte principal utilizado nos experimentos deste trabalho.

A.1 Código Fonte do Extrator de Características

O código a seguir (‘feature_extractor.py‘) implementa a classe responsável por carregar as arquiteturas CNN pré-treinadas, pré-processar as imagens e extrair os vetores de características.

```
import os
import numpy as np
import pandas as pd
import tensorflow as tf
import datetime
import logging
import yaml
from tqdm import tqdm
from PIL import Image
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import (
    ResNet50, VGG16, VGG19, EfficientNetV2L, InceptionV3
)
from tensorflow.keras.applications.resnet50 import preprocess_input as
   preprocess_input_resnet
from tensorflow.keras.applications.vgg16 import preprocess_input as
   preprocess_input_vgg16
from tensorflow.keras.applications.vgg19 import preprocess_input as
   preprocess_input_vgg19
from tensorflow.keras.applications.efficientnet_v2 import
   preprocess_input as preprocess_input_effnet
from tensorflow.keras.applications.inception_v3 import preprocess_input
   as preprocess_input_inception
from tensorflow.keras.models import Model
import time  # Importando time para medir o tempo de execu  o

```

```
class GPUManager:
    @staticmethod
    def configure_gpu():
        gpus = tf.config.list_physical_devices('GPU')
        if gpus:
            try:
                tf.config.set_visible_devices(gpus[0], 'GPU')
                logical_gpus = tf.config.list_logical_devices('GPU')
                print(len(gpus), "Physical GPUs,", len(logical_gpus), "
    Logical GPU")
            except RuntimeError as e:
                print(e)


class FeatureExtractor:
    def __init__(self, model_name, config):
        GPUManager.configure_gpu()
        self.model_name = model_name
        self.config = config
        self.output_dir = config['dataset']['output_dir']
        self.test_dir = os.path.join(self.output_dir, 'Test')
        self.reports_dir = os.path.join(self.output_dir, 'Reports')
        os.makedirs(self.reports_dir, exist_ok=True)

        self.class_labels = None
        self.target_size = (224, 224)
        self._load_model()

        logging.basicConfig(filename=self.config['paths']['log_file'],
                            level=getattr(logging, self.config['logging'
    ]['log_level']),
                            format='%(asctime)s - %(levelname)s - %(
    message)s')

    def _load_model(self):
        if self.model_name == 'ResNet50':
            base_model = ResNet50(weights='imagenet')
            self.model = Model(inputs=base_model.input, outputs=
    base_model.get_layer('avg_pool').output)
            self.preprocess_input = preprocess_input_resnet
        elif self.model_name == 'VGG16':
            base_model = VGG16(weights='imagenet')
            self.model = Model(inputs=base_model.input, outputs=
    base_model.get_layer('fc1').output)
            self.preprocess_input = preprocess_input_vgg16
```

```
62          elif self.model_name == 'VGG19':
63              base_model = VGG19(weights='imagenet')
64              self.model = Model(inputs=base_model.input, outputs=
    base_model.get_layer('fc1').output)
65              self.preprocess_input = preprocess_input_vgg19
66          elif self.model_name == 'EfficientNetV2L':
67              base_model = EfficientNetV2L(weights='imagenet')
68              self.model = Model(inputs=base_model.input, outputs=
    base_model.get_layer('avg_pool').output)
69              self.preprocess_input = preprocess_input_effnet
70              self.target_size = (480, 480)
71          elif self.model_name == 'InceptionV3':
72              base_model = InceptionV3(weights='imagenet')
73              self.model = Model(inputs=base_model.input, outputs=
    base_model.get_layer('avg_pool').output)
74              self.preprocess_input = preprocess_input_inception
75              self.target_size = (299, 299)
76          else:
77              raise ValueError("Unsupported model: " + self.model_name)
78
79      def extract_with_labels(self, batch_size=32):
80          start_time = time.time()  # Inicia a contagem do tempo de
    extra  o
81          features, labels = [], []
82          self.class_directories = [d for d in os.listdir(self.test_dir)
    if os.path.isdir(os.path.join(self.test_dir, d))]
83          self.class_labels = {class_dir: idx for idx, class_dir in
    enumerate(self.class_directories)}
84
85          all_images = []
86          all_labels = []
87
88          for class_dir, class_idx in self.class_labels.items():
89              class_path = os.path.join(self.test_dir, class_dir)
90              image_files = os.listdir(class_path)
91              for img_name in image_files:
92                  img_path = os.path.join(class_path, img_name)
93                  all_images.append(img_path)
94                  all_labels.append(class_idx)
95
96          dataset = tf.data.Dataset.from_tensor_slices((all_images,
    all_labels))
97          dataset = dataset.batch(batch_size)
98
```

```
99          for batch_paths, batch_labels in tqdm(dataset, total=len(
    all_images) // batch_size + 1, desc=f"Extracting {self.model_name}"):
100             batch_imgs = []
101             for img_path in batch_paths.numpy():
102                 img = Image.open(img_path.decode('utf-8')).resize(self.
    target_size).convert('RGB')
103                 x = image.img_to_array(img)
104                 batch_imgs.append(x)
105 
106             x_batch = np.array(batch_imgs)
107             x_batch = self.preprocess_input(x_batch)
108             batch_features = self.model.predict(x_batch, verbose=0)
109             batch_features = batch_features / np.linalg.norm(
    batch_features, axis=1, keepdims=True)
110 
111             features.extend(batch_features)
112             labels.extend(batch_labels.numpy())
113 
114         elapsed_time = time.time() - start_time  # Calcula o tempo total
     de extra  o
115         logging.info(f"Feature extraction completed for {self.model_name
    } in {elapsed_time:.2f} seconds.")
116         return features, labels, elapsed_time  # Retorna o tempo de
    execu  o tamb m
117 
118     def save_features_with_labels(self):
119         features, labels, elapsed_time = self.extract_with_labels()
120         features = np.array(features)
121         labels = np.array(labels)
122 
123         npy_dir = os.path.join(self.output_dir, 'NumpyFeatures')
124         os.makedirs(npy_dir, exist_ok=True)
125 
126         output_features = os.path.join(npy_dir, f'{self.model_name}
    _features.npy')
127         output_labels = os.path.join(npy_dir, f'{self.model_name}_labels
    .npy')
128         output_parquet = os.path.join(npy_dir, f'{self.model_name}
    _features.parquet')
129         output_csv = os.path.join(npy_dir, f'{self.model_name}_features.
    csv')
130 
131         np.save(output_features, features)
132         np.save(output_labels, labels)
133 
```

```
134         df = pd.DataFrame(features)
135         df['label'] = labels
136         df.to_parquet(output_parquet, index=False)
137         df.to_csv(output_csv, index=False)
138 
139         print(f"Features salvos em {output_features}, {output_labels}, {
    output_parquet} e {output_csv}")
140         logging.info(f"Features salvos em {output_features}, {
    output_labels}, {output_parquet} e {output_csv}")
141         self._write_log()
142 
143         return elapsed_time  # Retorna o tempo de execu  o para
    registro no CSV
144 
145     def _write_log(self):
146         timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S'
    )
147         log_filename = os.path.join(self.reports_dir, f"{self.model_name
    }_feature_extraction_log.txt")
148         with open(log_filename, 'a') as log_file:
149             log_file.write(f"Feature Extraction Log - {timestamp}\n")
150             log_file.write(f"Model Used: {self.model_name}\n")
151             log_file.write(f"Main Directory: {self.test_dir}\n")
152             for class_dir in self.class_directories:
153                 class_idx = self.class_labels[class_dir]
154                 num_images = len(os.listdir(os.path.join(self.test_dir,
    class_dir)))
155                 log_file.write(f"Class '{class_dir}' (Label Index: {
    class_idx}) - Images Processed: {num_images}\n")
156             log_file.write("\n")
157         logging.info(f"Log written to {log_filename}")
158 
159 
160 class FeatureExtractionPipeline:
161     def __init__(self, config_path):
162         with open(config_path, 'r') as file:
163             self.config = yaml.safe_load(file)
164 
165     def run(self):
166         execution_times = []  # Lista para armazenar tempos de
    execu  o
167 
168         for model_name in self.config['feature_extraction']['models']:
169             extractor = FeatureExtractor(model_name, self.config)
170             elapsed_time = extractor.save_features_with_labels()
```

```
171             execution_times.append({'Model': model_name, 'Time (seconds)
    ': elapsed_time})  # Adiciona tempo    lista
172 
173         # Salva os tempos de execu  o em um arquivo CSV
174         df_times = pd.DataFrame(execution_times)
175         df_times.to_csv(os.path.join(self.config['dataset']['output_dir'
    ], 'execution_times.csv'), index=False)
176 
177 
178 if __name__ == "__main__":
179     import sys
180     config_path = sys.argv[1]
181     pipeline = FeatureExtractionPipeline(config_path)
182     pipeline.run()
```

Listing A.1: Código fonte: feature_extractor.py

A.2 Código Fonte do Pipeline de Classificação

O código a seguir (‘classifier.py‘) implementa o pipeline para carregar as características extraídas, dividir os dados, realizar a busca de hiperparâmetros (Randomized Search), treinar os classificadores, avaliar o desempenho e salvar os resultados.

```
1  import os
2  import shutil
3  import logging
4  import yaml
5  from tqdm import tqdm
6  from sklearn.model_selection import train_test_split
7 
8  class DatasetSplitter:
9      def __init__(self, config_path):
10         with open(config_path, 'r') as file:
11             self.config = yaml.safe_load(file)
12 
13         self.source_dir = self.config['dataset']['source_dir']
14         self.output_dir = self.config['dataset']['output_dir']
15         self.test_size = self.config['dataset']['test_size']
16 
17         self.train_dir = os.path.join(self.output_dir, 'Train')
18         self.test_dir = os.path.join(self.output_dir, 'Test')
19         self.logs_dir = os.path.join(self.output_dir, 'Logs')
20 
```

```
21        os.makedirs(self.logs_dir, exist_ok=True)
22
23        self.config['paths']['train_folder'] = self.train_dir
24        self.config['paths']['test_folder'] = self.test_dir
25        self.config['paths']['log_file'] = os.path.join(self.logs_dir, '
    pipeline.log')
26
27        logging.basicConfig(filename=self.config['paths']['log_file'],
28                            level=getattr(logging, self.config['logging'
    ]['log_level']),
29                            format='%(asctime)s - %(levelname)s - %(
    message)s')
30
31        # Salva config atualizada
32        with open(config_path, 'w') as file:
33            yaml.dump(self.config, file)
34
35    def split(self):
36        for class_dir in tqdm(os.listdir(self.source_dir), desc="
    Splitting dataset"):
37            class_path = os.path.join(self.source_dir, class_dir)
38
39            if not os.path.isdir(class_path):
40                continue
41
42            images = [f for f in os.listdir(class_path) if f.lower().
    endswith(('.png', '.jpg', '.jpeg', '.tiff', '.bmp', '.gif'))]
43
44            if not images:
45                continue
46
47            train_files, test_files = train_test_split(images, test_size
    =self.test_size, stratify=[class_dir]*len(images), random_state=42)
48
49            train_class_dir = os.path.join(self.train_dir, class_dir)
50            test_class_dir = os.path.join(self.test_dir, class_dir)
51            os.makedirs(train_class_dir, exist_ok=True)
52            os.makedirs(test_class_dir, exist_ok=True)
53
54            for file in train_files:
55                shutil.copy2(os.path.join(class_path, file), os.path.
    join(train_class_dir, file))
56
57            for file in test_files:
```

```
                shutil.copy2(os.path.join(class_path, file), os.path.
    join(test_class_dir, file))

            logging.info(f"Classe '{class_dir}' - {len(train_files)}
    imagens para treino, {len(test_files)} para teste.")

        print(f"Dataset split conclu do. Treino em '{self.train_dir}',
    Teste em '{self.test_dir}'.")
        logging.info("Dataset split conclu do.")


if __name__ == "__main__":
    import sys
    config_path = sys.argv[1]
    splitter = DatasetSplitter(config_path)
    splitter.split()
```

Listing A.2: Código fonte: classifier.py

O código a seguir (‘dataset_splitter.py‘) implementa o pipeline para fazer a divisão dos dados em treino, validação e teste.

```
import os
import shutil
import logging
import yaml
from tqdm import tqdm
from sklearn.model_selection import train_test_split

class DatasetSplitter:
    def __init__(self, config_path):
        with open(config_path, 'r') as file:
            self.config = yaml.safe_load(file)

        self.source_dir = self.config['dataset']['source_dir']
        self.output_dir = self.config['dataset']['output_dir']
        self.test_size = self.config['dataset']['test_size']

        self.train_dir = os.path.join(self.output_dir, 'Train')
        self.test_dir = os.path.join(self.output_dir, 'Test')
        self.logs_dir = os.path.join(self.output_dir, 'Logs')

        os.makedirs(self.logs_dir, exist_ok=True)

        self.config['paths']['train_folder'] = self.train_dir
        self.config['paths']['test_folder'] = self.test_dir
```

```
25          self.config['paths']['log_file'] = os.path.join(self.logs_dir, '
    pipeline.log')
26
27          logging.basicConfig(filename=self.config['paths']['log_file'],
28                              level=getattr(logging, self.config['logging'
    ]['log_level']),
29                              format='%(asctime)s - %(levelname)s - %(
    message)s')
30
31          # Salva config atualizada
32          with open(config_path, 'w') as file:
33              yaml.dump(self.config, file)
34
35      def split(self):
36          for class_dir in tqdm(os.listdir(self.source_dir), desc="
    Splitting dataset"):
37              class_path = os.path.join(self.source_dir, class_dir)
38
39              if not os.path.isdir(class_path):
40                  continue
41
42              images = [f for f in os.listdir(class_path) if f.lower().
    endswith(('.png', '.jpg', '.jpeg', '.tiff', '.bmp', '.gif'))]
43
44              if not images:
45                  continue
46
47              train_files, test_files = train_test_split(images, test_size
    =self.test_size, stratify=[class_dir]*len(images), random_state=42)
48
49              train_class_dir = os.path.join(self.train_dir, class_dir)
50              test_class_dir = os.path.join(self.test_dir, class_dir)
51              os.makedirs(train_class_dir, exist_ok=True)
52              os.makedirs(test_class_dir, exist_ok=True)
53
54              for file in train_files:
55                  shutil.copy2(os.path.join(class_path, file), os.path.
    join(train_class_dir, file))
56
57              for file in test_files:
58                  shutil.copy2(os.path.join(class_path, file), os.path.
    join(test_class_dir, file))
59
60              logging.info(f"Classe '{class_dir}' - {len(train_files)}
    imagens para treino, {len(test_files)} para teste.")
```

```
61
62          print(f"Dataset split conclu do. Treino em '{self.train_dir}',
    Teste em '{self.test_dir}'.")
63          logging.info("Dataset split conclu do.")
64
65
66 if __name__ == "__main__":
67     import sys
68     config_path = sys.argv[1]
69     splitter = DatasetSplitter(config_path)
70     splitter.split()
```

Listing A.3: Código fonte: dataset_*splitter.py*

Visualização de Mapas de Características das CNNs

Este apêndice apresenta exemplos de mapas de ativação extraídos de diferentes camadas e blocos das arquiteturas convolucionais utilizadas neste trabalho. Essas visualizações fornecem uma perspectiva qualitativa sobre os padrões que as redes aprendem a identificar em diferentes estágios do processo de extração de características. Camadas iniciais tendem a capturar bordas e texturas simples, enquanto camadas mais profundas respondem a formas mais complexas e semânticas específicas.

B.1 Visualizações da EfficientNetV2L

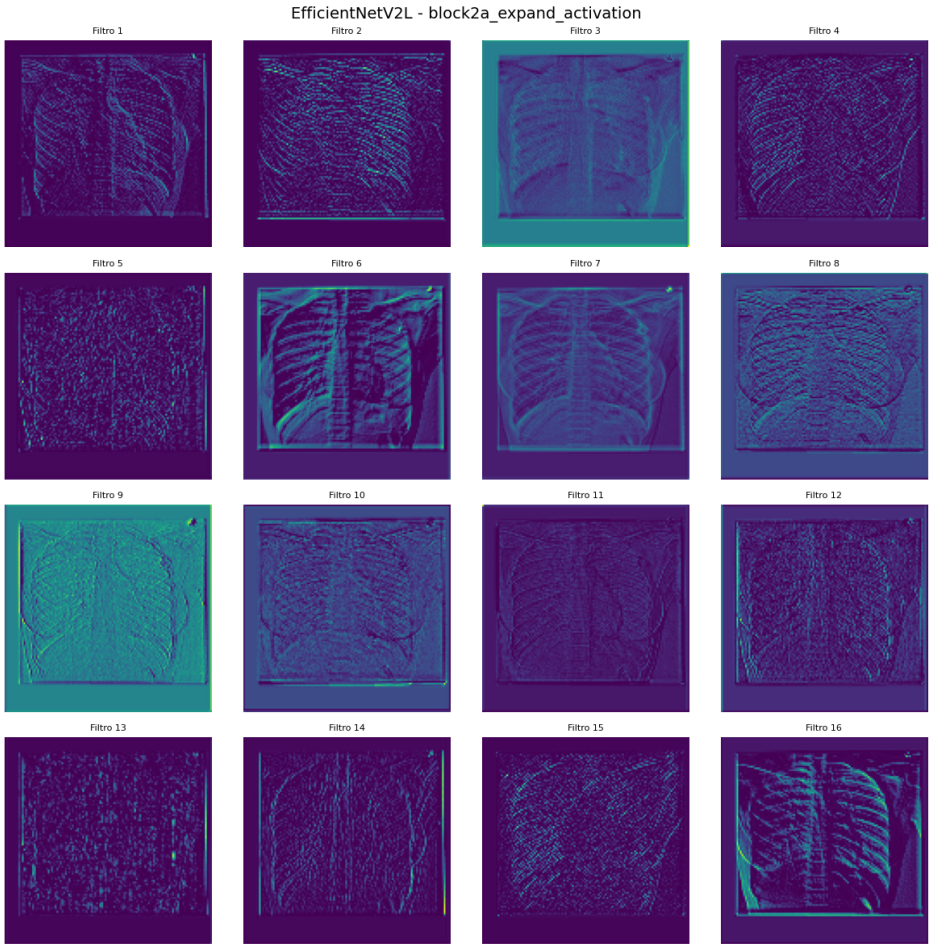


Figura B.1: Mapas de ativação da camada de expansão do bloco `block2a` da EfficientNetV2L. Cada quadrado representa a resposta de um filtro convolucional diferente aplicado à imagem de entrada.

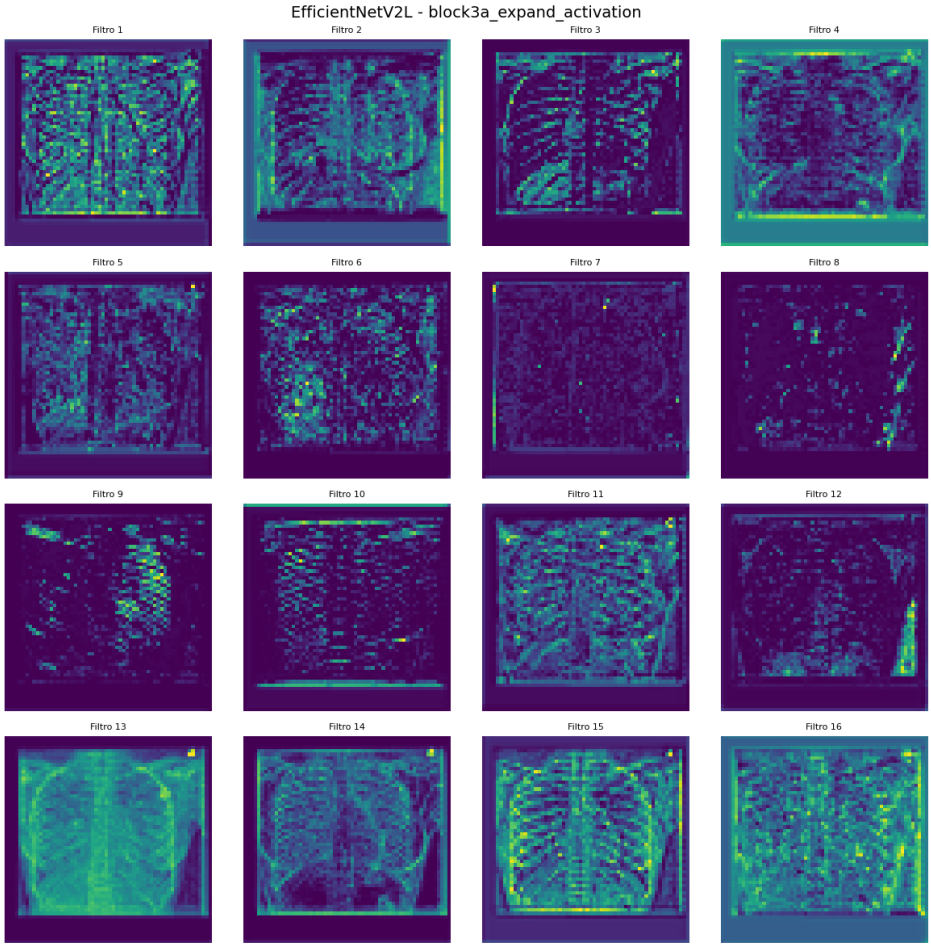


Figura B.2: Ativações geradas no bloco `block3a` da EfficientNetV2L, destacando a evolução das representações internas da rede.

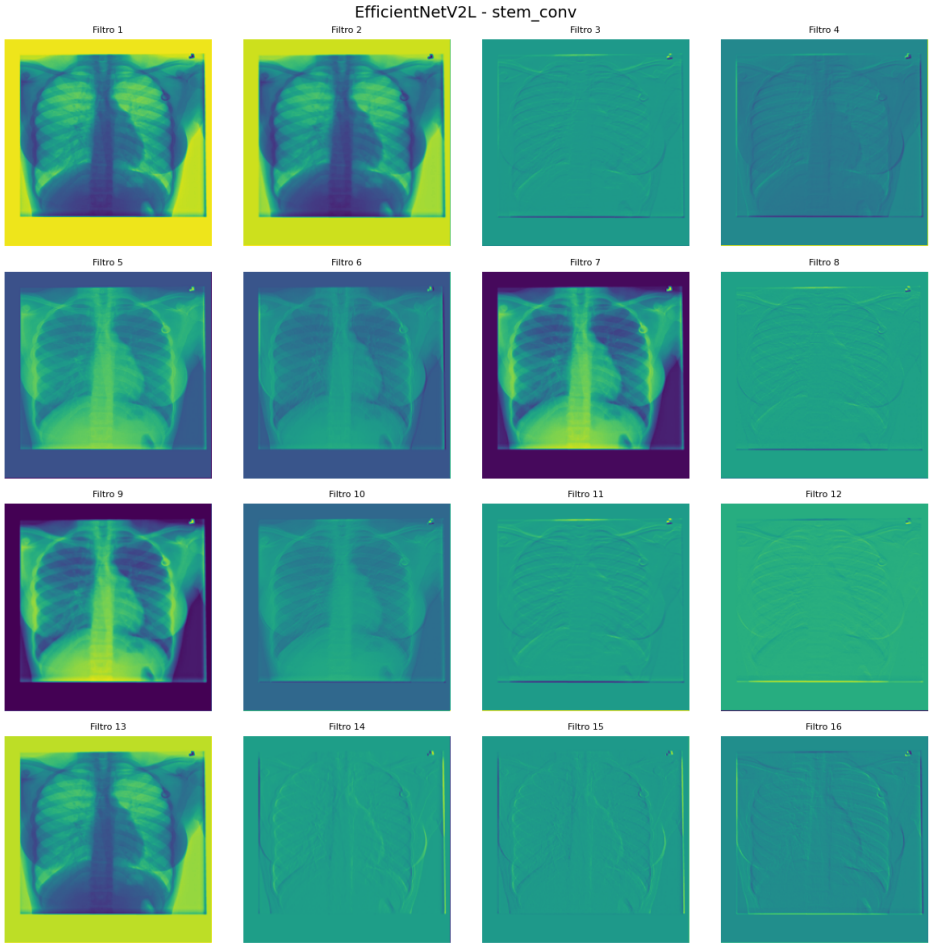


Figura B.3: Mapas de ativação da camada `stem_conv`, que corresponde à primeira convolução da EfficientNetV2L. Essa camada captura padrões de baixo nível, como bordas e contrastes locais.

B.2 Visualizações da ResNet50

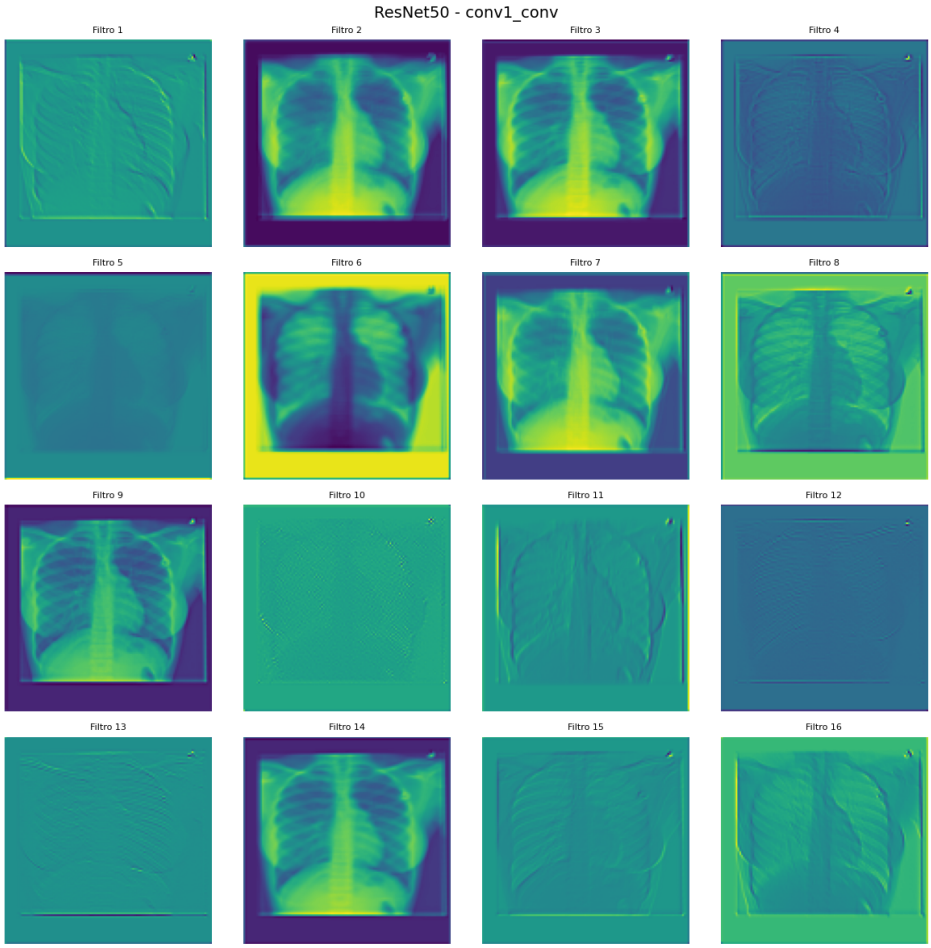


Figura B.4: Mapas de ativação da camada inicial `conv1_conv` da ResNet50.

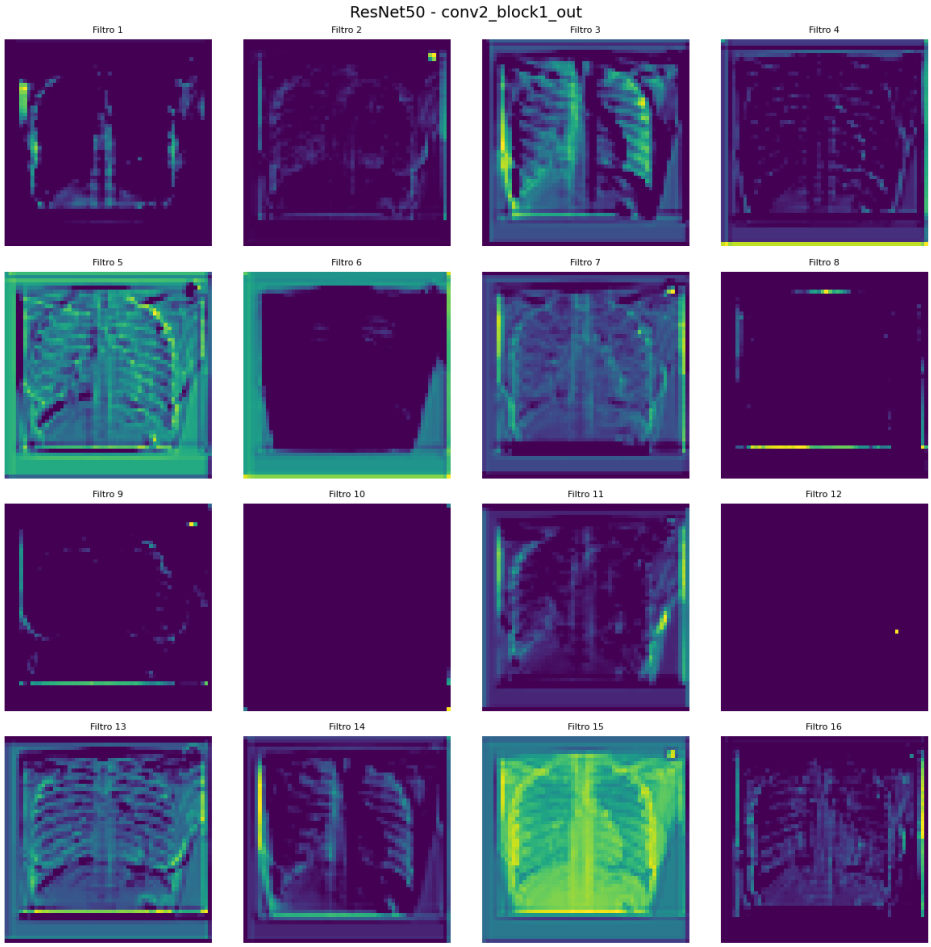


Figura B.5: Mapas de ativação do bloco `conv2_block1_out` da ResNet50.

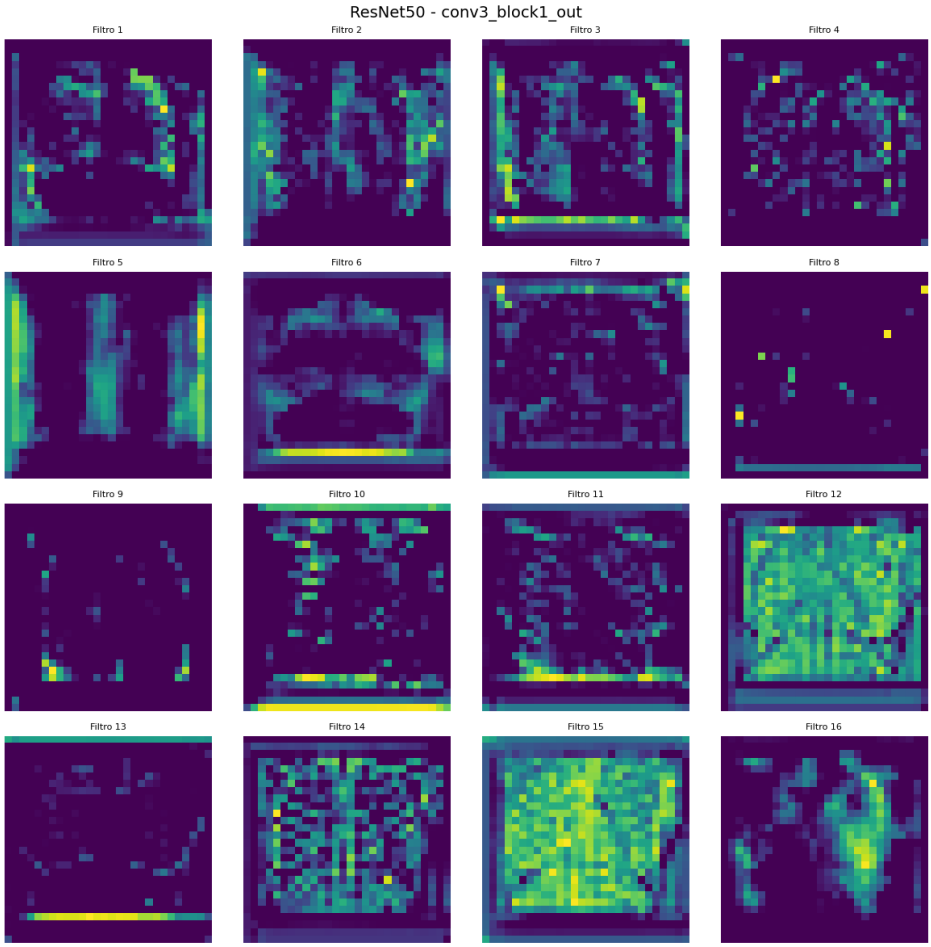


Figura B.6: Ativações do bloco intermediário `conv3_block1_out` da ResNet50, ilustrando como a complexidade dos padrões aumenta.

B.3 Visualizações da InceptionV3

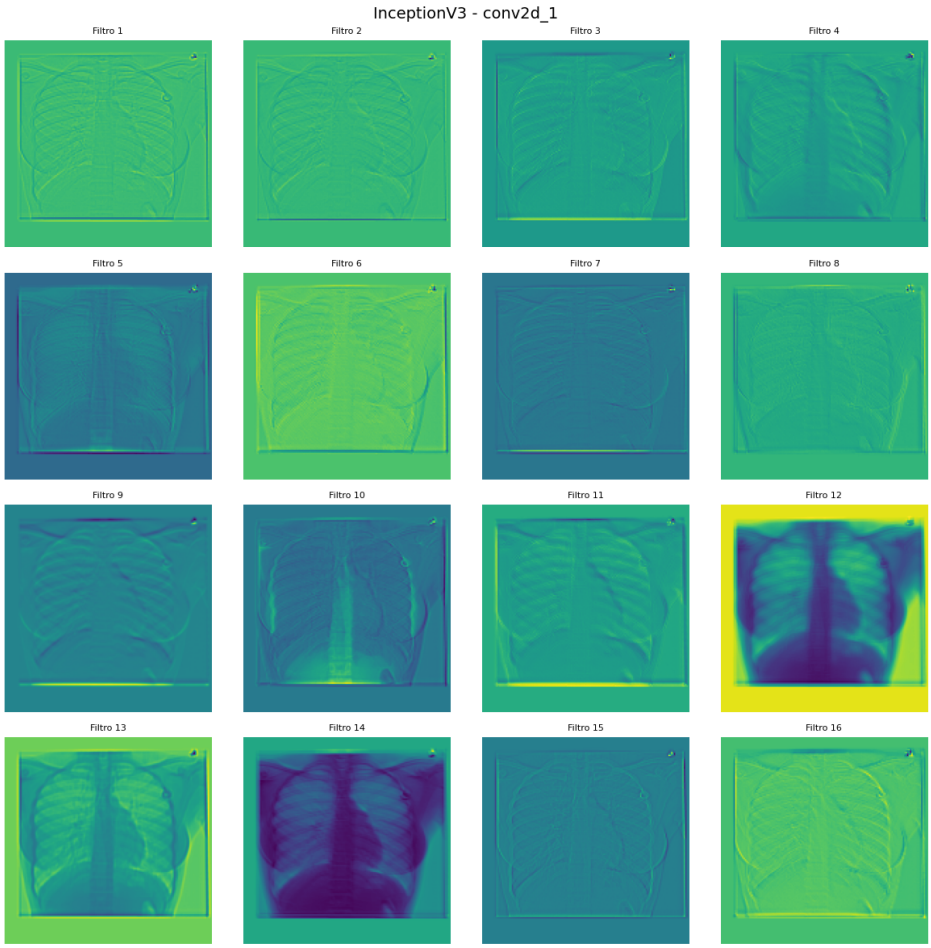


Figura B.7: Mapas de ativação da camada `conv2d_1` da InceptionV3.

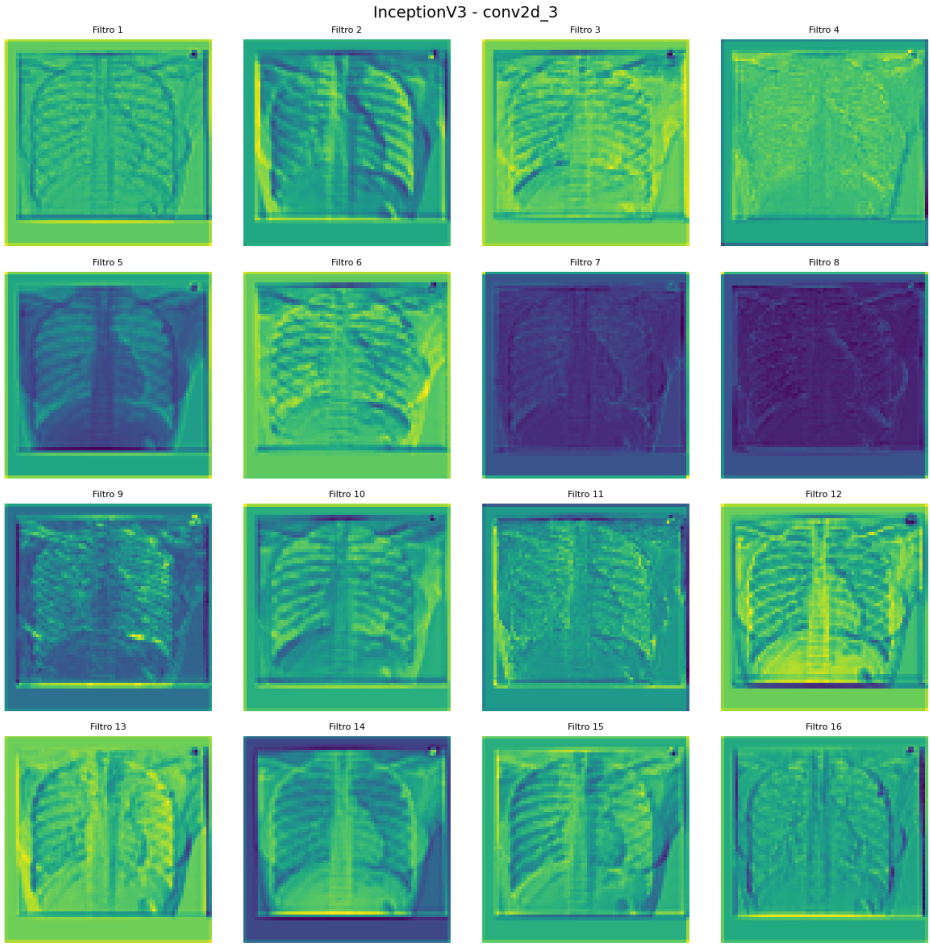


Figura B.8: Ativações intermediárias da InceptionV3 na camada `conv2d_3`.

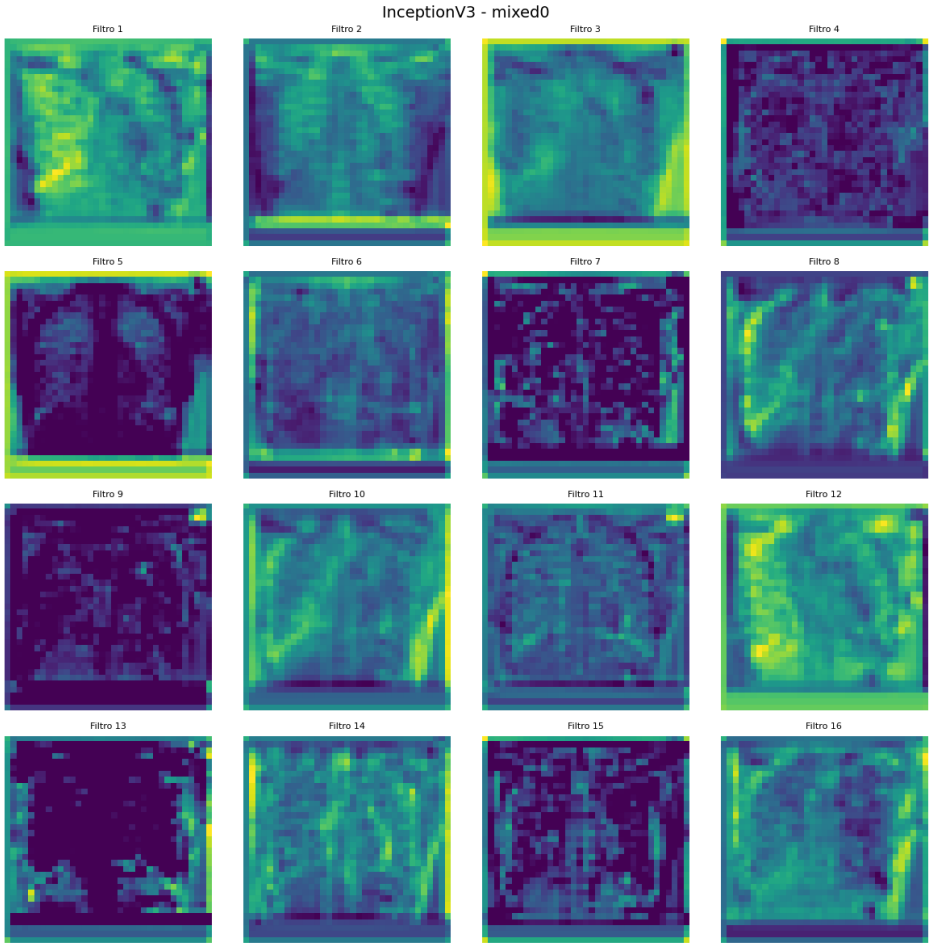


Figura B.9: Mapas de ativação do módulo `mixed0`, que combina filtros convolucionais de diferentes tamanhos, característica marcante da InceptionV3.

B.4 Visualizações da VGG16

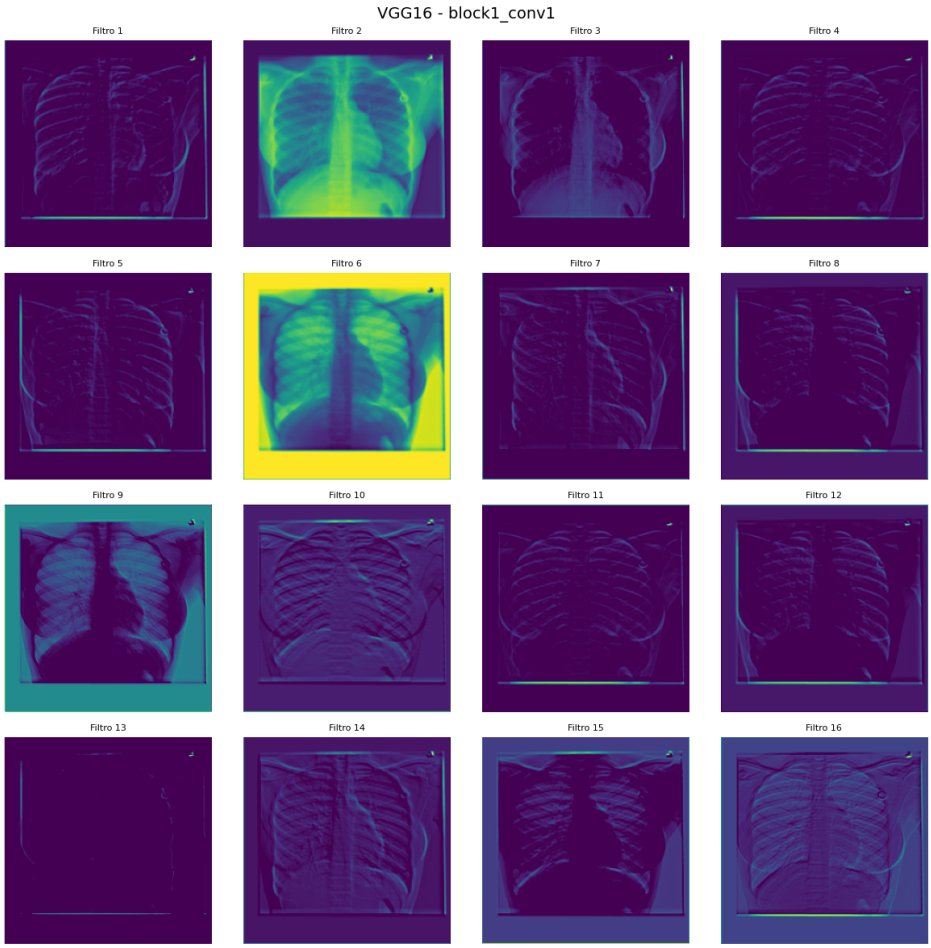


Figura B.10: Ativações da camada `block1_conv1` da VGG16.

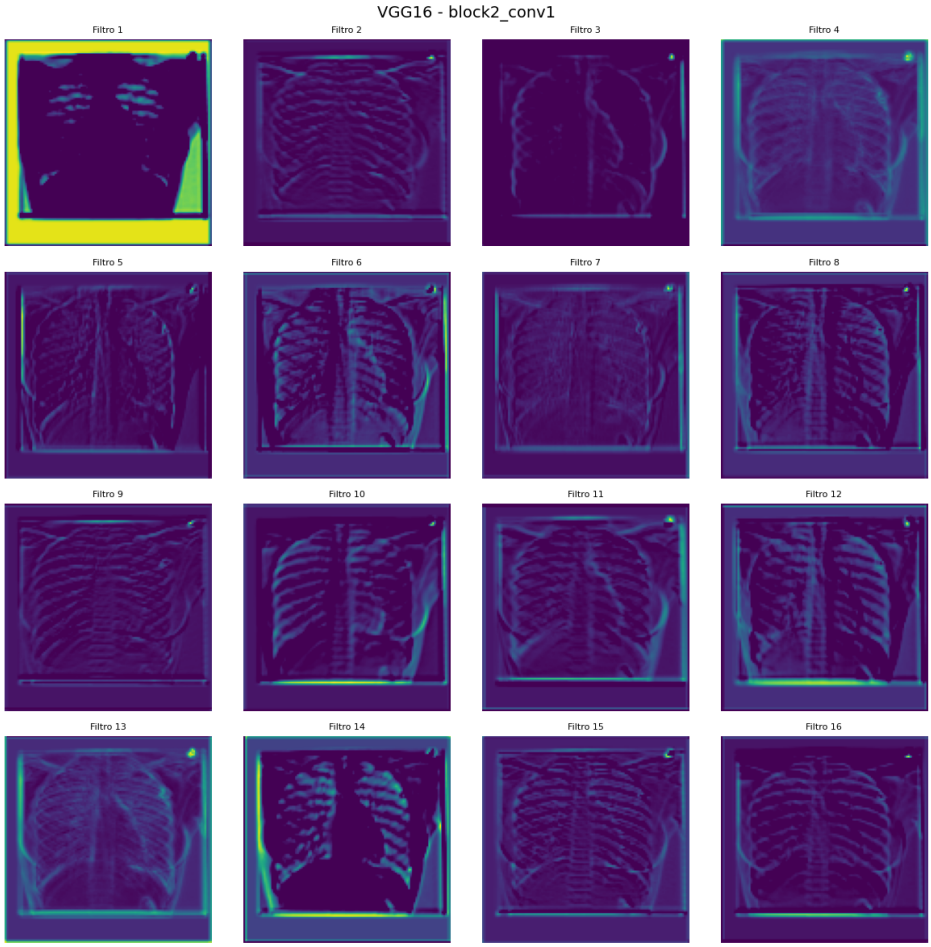


Figura B.11: Mapas de ativação da VGG16 no início do segundo bloco convolucional.

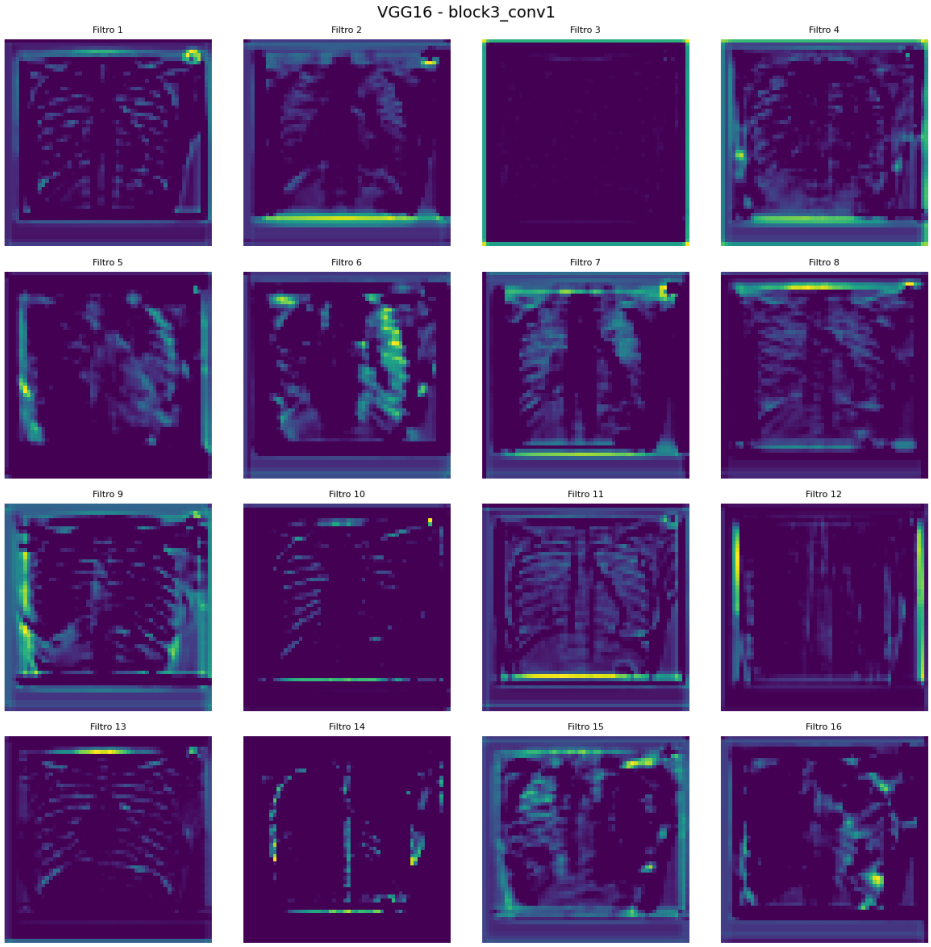


Figura B.12: Ativações do bloco `block3_conv1`, demonstrando maior abstração nos padrões detectados.

B.5 Visualizações da VGG19

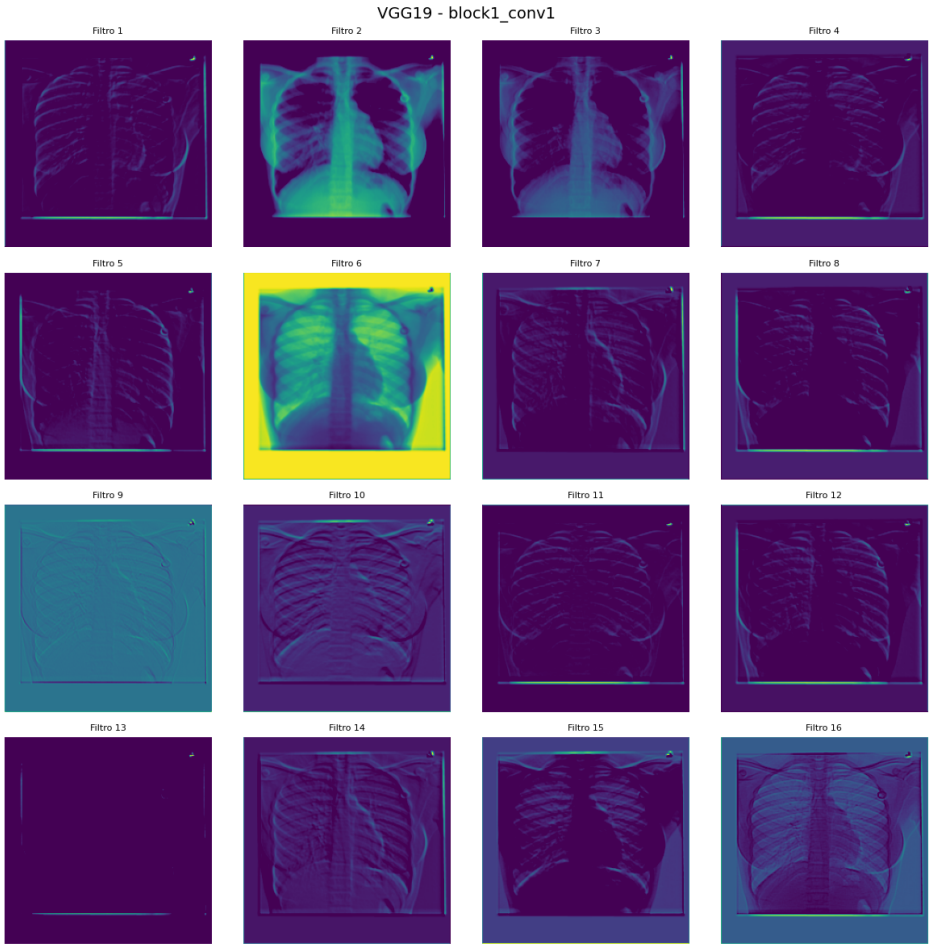


Figura B.13: Mapas de ativação da camada `block1_conv1` da VGG19.

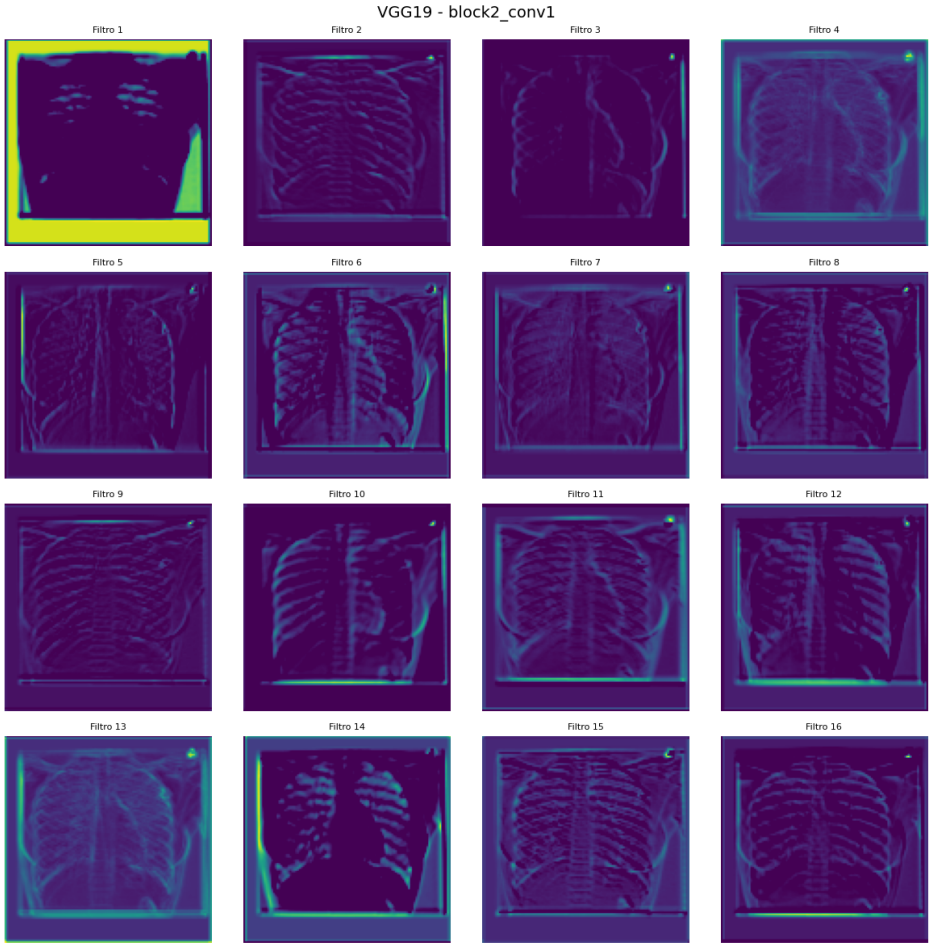


Figura B.14: Mapas de ativação da VGG19 no bloco `block2_conv1`.

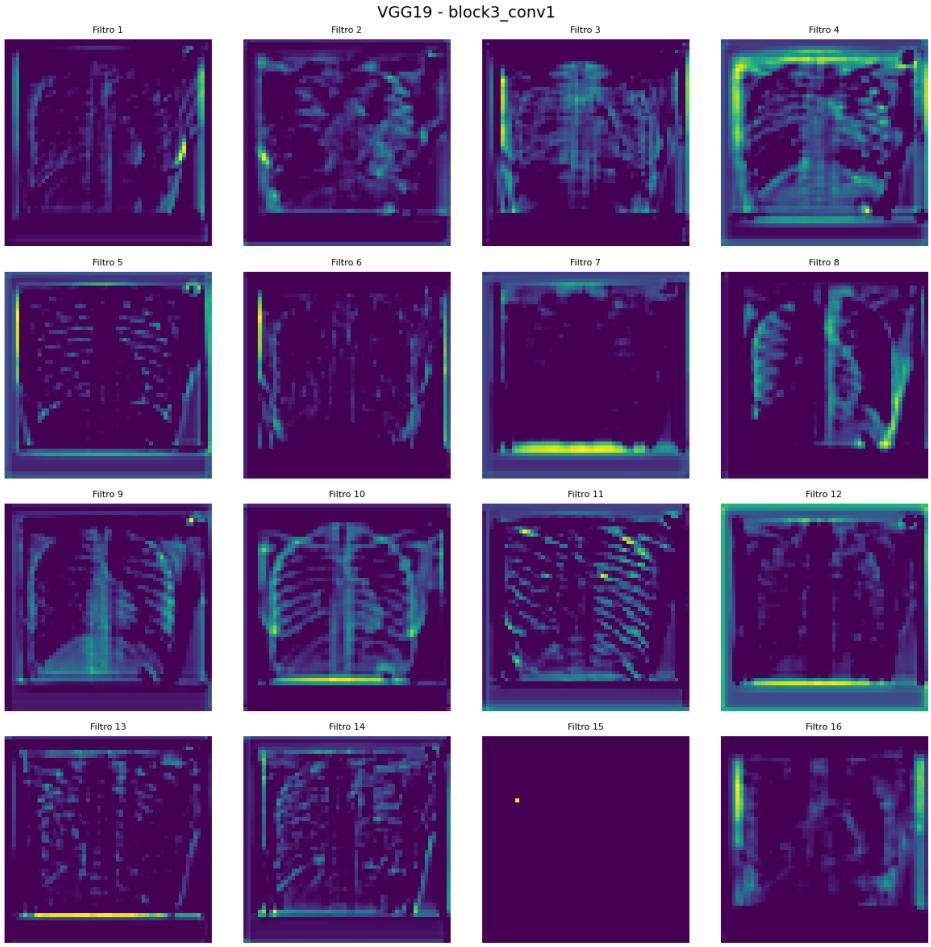


Figura B.15: Ativações do bloco `block3_conv1`, mostrando padrões visuais mais abstratos.

Referências

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren e Jian Sun: *Deep residual learning for image recognition*. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 770–778, 2016. x, 2, 11, 12, 13, 14

[2] Chen, Y, J Li, H Xiao, X Jin, S Yan e J Feng: *Resnetplus: An improved resnet structure with skip connections*. IEEE Transactions on Image Processing, 28(1):19–31, 2018. x, 14

[3] Simonyan, Karen e Andrew Zisserman: *Very deep convolutional networks for large-scale image recognition*, 2014. x, 2, 12, 13, 15, 16, 17

[4] Tan, Mingxing e Quoc Le: *Efficientnetv2: Smaller models and faster training*. Em *International conference on machine learning*, páginas 10096–10106. PMLR, 2021. x, 2, 18, 19

[5] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke e Andrew Rabinovich: *Going deeper with convolutions*. Em *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, páginas 1–9, 2015. x, 21

[6] Szegedy, Christian *et al.*: *Rethinking the inception architecture for computer vision*. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 2818–2826, 2016. x, 2, 20, 21

[7] Hastie, Trevor, Robert Tibshirani e Jerome Friedman: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, second edição, 2009. xi, 26, 35, 36, 37

[8] Carneiro, Allan Cordeiro, Jose Gerardo Fonteles Lopes, Flávio Henrique Duarte de Araújo, Romuere Rodrigues Veloso e Silva, Cornelia Janayna Pereira Passarinho, Jeová Farias Sales Rocha Neto e Fátima Nelsizeuma Sombra de Medeiros: *Análise de fotografias de pílulas por redes neurais convolucionais*. Em *Simpósio de Instrumentação e Imagens Médicas (SIIM'2017), 8º.; Simpósio de Processamento de Sinais (SPS'2017), 7º.*, São Bernardo do Campo, SP, 2017. 29 nov.-01 dez. 1, 9

[9] Abade, André da Silva: *Estudo e análise de Redes Neurais Convolucionais Profundas na identificação de doenças em plantas por imagens*. Tese (doutorado em sistemas mecatrônicos), Universidade de Brasília, Brasília, 2022. 1, 9

[10] Da Silva Ribeiro, Eduardo et al.: *Ldmat: função de custo baseada em matrizes de distâncias para o treinamento de redes neurais convolucionais*, 2021. 1, 9

[11] Tozevich, Luís Gustavo Werle *et al.*: *Classificação de patologias em estruturas usando redes neurais convolucionais: diferenciação em trincas, fissuras e rachaduras.* Revista ComInG-Communications and Innovations Gazette, 7(1):73–86, 2023. 1, 8

[12] Noble, William S: *What is a support vector machine?* Nature biotechnology, 24(12):1565–1567, 2006. 2, 44, 46

[13] Breiman, Leo: *Random forests.* Machine learning, 45:5–32, 2001. 2, 40, 41

[14] Chen, Tianqi e Carlos Guestrin: *Xgboost: A scalable tree boosting system.* Em *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, páginas 785–794, 2016. 2, 42, 43, 44

[15] Sokolova, Marina, Nathalie Japkowicz e Stan Szpakowicz: *Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation.* Em *Australasian joint conference on artificial intelligence*, páginas 1015–1021. Springer Berlin Heidelberg, 2006. 2, 23, 24, 25

[16] Antonio, André Lara Temple de: *Combinação de características para recuperação e classificação de imagens médicas*, 2023. 9

[17] Ioffe, Sergey e Christian Szegedy: *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* Em *International conference on machine learning*, páginas 448–456. PMLR, 2015. 12, 13

[18] Chicco, Davide e Giuseppe Jurman: *The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation.* BMC genomics, 21(1):1–13, 2020. 26

[19] Bergstra, James e Yoshua Bengio: *Random search for hyper-parameter optimization.* Em *Journal of machine learning research*, volume 13, páginas 281–305, 2012. 27

[20] Yu, Tong e Hong Zhu: *Hyper-parameter optimization: A review of algorithms and applications*, 2020. 27

[21] Dubey, Shiv Ram, Satish Kumar Singh e Bidyut Baran Chaudhuri: *Activation functions in deep learning: A comprehensive survey and benchmark.* Neurocomputing, 503:92–108, 2022. 28, 29, 30, 31

[22] Quinlan, J. Ross: *Induction of decision trees.* Machine learning, 1:81–106, 1986. 37, 38, 39

[23] Bentéjac, Candice, Anna Csörgő e Gonzalo Martínez-Muñoz: *A comparative analysis of gradient boosting algorithms.* Artificial Intelligence Review, 54:1937–1967, 2021. 42, 44

[24] Friedman, Jerome H: *Stochastic gradient boosting.* Em *Computational statistics & data analysis*, volume 38, páginas 367–378. Elsevier, 2002. 43

[25] Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush e Andrey Gulin: *Catboost: unbiased boosting with categorical features*. arXiv preprint arXiv:1706.09516, 2017. 46, 47, 48

[26] Micci-Barreca, Daniele: *A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems*. ACM SIGKDD Explorations Newsletter, 3(1):27–32, 2001. 47

[27] Wickramasinghe, Indika e Harsha Kalutarage: *Naive bayes: applications, variations and vulnerabilities: a review of literature with code snippets for implementation*. Soft Computing, 25(3):2277–2293, 2021. 48

[28] Schapire, Robert E: *Explaining adaboost*. Em *Empirical inference: Festschrift in honor of Vladimir N. Vapnik*, páginas 37–52. Springer Berlin Heidelberg, 2013. 51, 52

[29] Friedman, Jerome, Trevor Hastie e Robert Tibshirani: *Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)*. The Annals of Statistics, 28(2):337–407, 2000. 52

[30] Freund, Yoav e Robert E Schapire: *A decision-theoretic generalization of on-line learning and an application to boosting*. Journal of computer and system sciences, 55(1):119–139, 1997. 52

[31] Schapire, Robert E, Yoav Freund, Peter Bartlett e Wee Sun Lee: *Boosting the margin: A new explanation for the effectiveness of voting methods*. The annals of statistics, 26(5):1651–1686, 1998. 52

[32] Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye e Tie Yan Liu: *Lightgbm: A highly efficient gradient boosting decision tree*. Em *Advances in neural information processing systems 30 (NIPS 2017)*, páginas 3146–3154, 2017. 53, 54

[33] Nakkiran, Preetum, Gal Kaplun, Dimitris Kalimeris, Tristan Yang, Benjamin L. Edelman, Fred Zhang e Boaz Barak: *SGD on neural networks learns functions of increasing complexity*. arXiv preprint arXiv:1905.11604, abs/1905.11604, 2019. `https://arxiv.org/abs/1905.11604`. 54, 55