Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Detecção de Marcas d'Água em Imagens de Anúncios de Imóveis Utilizando Redes Neurais Convolucionais

Gabriel Brandão de Almeida

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Prof.ª Dr.ª Nina S. T. Hirata

São Paulo
2025

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)

Agradecimentos

Defeat is not the worst of failures. Not to have tried is the true failure.
— George Edward Woodberry

Gostaria de registrar meus agradecimentos à minha família, pelo apoio durante todo o período da minha graduação, aos meus amigos e namorado, pelo suporte durante os momentos difíceis, e à minha orientadora, Prof.ªNina S. T. Hirata, pela orientação durante o desenvolvimento deste trabalho.

Resumo

Gabriel Brandão de Almeida. **Detecção de Marcas d'Água em Imagens de Anúncios de Imóveis Utilizando Redes Neurais Convolucionais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

No setor imobiliário, a proteção do uso de imagens publicadas em anúncios online tornou-se um desafio devido à facilidade de compartilhamento e reutilização sem permissão. Para resolver esse problema, muitas imobiliárias utilizam marcas d'água em suas fotografias. Este trabalho propõe uma abordagem baseada em redes neurais convolucionais (CNNs) para detectar marcas d'água em imagens de imóveis. O conjunto de dados foi coletado de sites de imobiliárias e contém 60.258 imagens, sendo 31.365 amostras com marcas d'água de diferentes tipos e posicionamentos. Aplicamos técnicas de pré-processamento, aumento de dados e otimizão de hiperparâmetros para garantir um treinamento eficiente. Para a detecção, foram avaliados três modelos de CNN pré-treinadas: ResNet, EfficientNet e ConvNeXt. Os resultados mostraram que o modelo com maior capacidade de generalização para a tarefa foi a ConvNeXt, com acurácia de 87,8% e F1 de 87,1%, enquanto a EfficientNet apresentou um bom balanço entre desempenho e eficiência computacional, com 84% de acurácia e 83.% de F1, mas utilizando 25% menos parâmetros.

Palavras-chave: Redes Neurais Convolucionais. Marcas d'Água. Aprendizado de Máquina. Aprendizado Profundo. Classificação.

Abstract

Gabriel Brandão de Almeida. **Watermark Detection in Real Estate Listing Images With Convolutional Neural Networks**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

The protection of photographs used in online listings is a challenge for real estate business. Images can be easily reused without permission, making watermarking a common method for protection. This study explores the use of convolutional neural networks (CNNs) to detect watermarks in images from real estate online listings. The dataset was collected from different websites and contains 60,258 images, including 31,365 samples with watermarks of various types and positions. To achieve efficient training, we applied preprocessing techniques, data augmentation, and hyperparameter optimization. We evaluated three pre-trained models: ResNet, EfficientNet, and ConvNeXt. The results show that ConvNeXt had the highest performance, with 87.8% accuracy and 87.1% F1-score. The EfficientNet, otherwise, offers a balance between performance and computational efficiency, reaching 84% accuracy and an F1-score of 83%, while using 25% fewer parameters.

Keywords: Convolutional Neural Networks. Watermark. Machine Learning. Deep Learning. Classification.

Lista de abreviaturas

CNN	Rede Neural Convolucional (*Convolutional Neural Network*)
FFN	Rede Neural Feedforward (*Feed-Forward Network*)
ML	Aprendizado de Máquina (*Machine Learning*)
MSE	Erro Quadrático Médio (*Mean Squared Error*)
GD	Gradiente Descendente (*Gradient Descent*)
SGD	Gradiente Descendente Estocástico (*Stochastic Gradient Descent*)
FCL	Camadas Totalmente Conectadas (*Fully Connected Layers*)
TP	Verdadeiro Positivo (*True Positive*)
TN	Verdadeiro Negativo (*True Negative*)
FP	Falso Positivo (*False Positive*)
FN	Falso Negativo (*False Negative*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Lista de figuras

1.1 Exemplo de um sistema genérico de marca d'água. 3
1.2 Exemplos de imagens retiradas de bancos de imagens. 4
1.3 Divisão do aprendizado de máquina em categorias principais e exemplos de problemas. 5
1.4 Perceptron simples com entradas $x_1, x_2, x_3, \ldots x_n$, pesos $w_1, w_2, w_3, \ldots w_n$, *bias* b e função de ativação f. 6

1.5 Conjunto de perceptrons organizados em uma camada. 7
1.6 Exemplo de um *Multilayer Perceptron* (MLP) com camadas ocultas. (adaptado) 8
1.7 *Backprogapation* dos erros δ_i da camada $l+1$ para a camada l (adaptado). 12
1.8 Exemplo de convolução com um filtro de tamanho 3. (adaptado) 16
1.9 Ilustração de uma camada convolucional com múltiplos filtros. Cada filtro gera um mapa de características distinto. 17
1.10 Exemplo de *max pooling* com *stride* 2. 17
1.11 Ilustração de uma CNN genérica com camadas de convolução, *pooling* e camadas totalmente conectadas (adaptado). 18

2.1 Exemplos de imagens com marca d'água do conjunto de dados. 21
2.2 Exemplos de imagens sem marca d'água do conjunto de dados. 22
2.3 Exemplo de redimensionamento de imagem. 23
2.4 Exemplos de transformações do aumento de dados. 24

3.1 Matrizes de confusão dos modelos treinados. 28

Lista de tabelas

1.1 Matriz de confusão para classificação binária. 14

2.1 Características das arquiteturas utilizadas. 25

3.1 Métricas obtidas após treinamento dos modelos. 27

Sumário

Introdução **1**

1 Fundamentação Teórica **3**

1.1 Marca d'água digital . . . 3

1.1.1 Aplicações em Imagens . . . 4

1.2 Aprendizado de Máquina . . . 5

1.3 Redes Neurais . . . 6

1.3.1 Perceptron . . . 6

1.3.2 Multilayer Perceptron . . . 7

1.3.3 Função de Custo . . . 8

1.3.4 Função de Ativação . . . 9

1.3.5 Retropropagação e Gradiente Descendente . . . 12

1.3.6 Treinamento . . . 13

1.3.7 Avaliação de Modelos . . . 14

1.4 Redes Neurais Convolucionais . . . 15

1.4.1 Convolução . . . 15

1.4.2 *Pooling* . . . 16

1.4.3 Camadas Totalmente Conectadas . . . 17

1.5 Arquiteturas . . . 17

2 Metodologia **19**

2.1 Ferramentas Utilizadas . . . 19

2.2 Coleta de Dados . . . 20

2.3 Preparação dos Dados . . . 20

2.4 Aprendizado por Transferência . . . 24

2.5 Treinamento . . . 25

2.6 Otimização de Limiar de Decisão e Avaliação . . . 26

3 Resultados **27**

3.1 Métricas 27
3.2 Discussão 28

4 Conclusão **29**

Referências **31**

Introdução

A utilização de marcas d'água é um conceito antigo, originado da prática de aplicação de marcas em documentos de papel para garantir autenticidade e propriedade. No contexto digital, essa técnica evoluiu para proteger diferentes tipos de mídia, sendo mais comumente aplicada em imagens. O principal objetivo das marcas d'água digitais é adicionar informações para dificultar a cópia e o uso não autorizado de conteúdos.

Com o crescimento da internet e a facilidade de compartilhamento de dados, tornou-se cada vez mais difícil controlar a reprodução e distribuição de imagens. Esse cenário é particularmente relevante no setor imobiliário, no qual imagens de anúncios são frequentemente reutilizadas sem permissão. Para garantir a exclusividade do uso dessas imagens, muitas imobiliárias aplicam marcas d'água em suas fotografias.

O problema do uso indevido de imagens já resultou em disputas judiciais, como o caso ocorrido em 2022 entre as empresas QuintoAndar e Loft,[1] em que houve acusações de uso de imagens de anúncios sem autorização. Conflitos como esse corroboram a importância de mecanismos para detectar a presença de marcas d'água, protegendo os direitos sobre as imagens e prevenindo o uso indevido.

Nesse cenário, as redes neurais convolucionais apresentam grande potencial. Elas revolucionaram o campo de visão computacional ao introduzirem métodos eficientes para resolver tarefas de reconhecimento de padrões complexos em imagens, como detecção de objetos, segmentação e classificação.

Este trabalho tem como objetivo avaliar a aplicação de redes neurais convolucionais na detecção de marcas d'água em imagens de anúncios de imóveis e está organizado da seguinte forma: o Capítulo 1 apresenta os conceitos fundamentais sobre redes neurais e marcas d'água; o Capítulo 2 descreve a metodologia adotada para a realização dos experimentos; o Capítulo 3 discute os resultados obtidos; e o Capítulo 4 apresenta as conclusões e sugestões para trabalhos futuros.

[1] Matéria de O Globo.

Capítulo 1

Fundamentação Teórica

Neste capítulo serão apresentados os conceitos explorados no projeto, como definições sobre marcas d'água, exemplos de aplicação e noções de aprendizado de máquina e redes neurais.

1.1 Marca d'água digital

Marcas d'água digitais são sinais portadores de informação inseridos em dados digitais, como imagens, vídeos ou áudios, com o objetivo de possibilitar sua identificação ou extração em etapas posteriores. Esses sinais podem assumir diferentes formas, incluindo sequências numéricas, elementos gráficos ou padrões sonoros, dependendo das características do dado ao qual são aplicados e do método de incorporação utilizado (Nematollahi *et al.*, 2016).

Ao longo deste trabalho, utilizaremos o termo "marca" como equivalente à "marca d'água digital" ou "marca d'água visível", desde que essa simplificação não comprometa a clareza e compreensão do texto.

De modo geral, um sistema de marcação digital é composto por três elementos principais: a marca d'água, o aplicador (*embedder*) e o detector, conforme ilustrado na Figura 1.1.

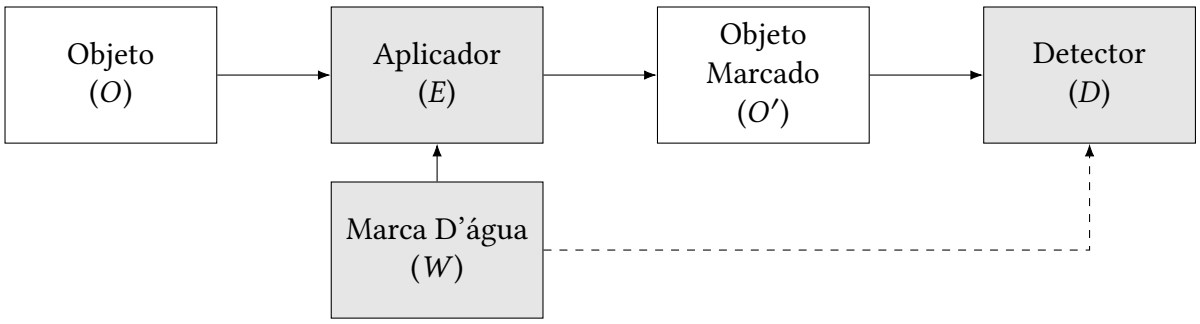


Figura 1.1: *Exemplo de um sistema genérico de marca d'água.*

O aplicador é responsável por receber o dado original ou hospedeiro, denotado por

O, e empregar um algoritmo de inserção para incorporar a marca W. O resultado desse processo é o dado marcado O', que contém a marca d'água embutida. Essa modificação introduz informações adicionais ao dado, possibilitando, por exemplo, a identificação de autoria, a verificação de autenticidade ou a proteção contra uso não autorizado.

O detector, por sua vez, utiliza um método de extração ou analisa o dado de entrada para verificar a presença da marca W. Durante esse processo, o detector pode, ainda, ter acesso ao dado original O ou à própria marca W, dependendo do método empregado. Este trabalho tem como objetivo testar a utilização de redes neurais convolucionais para substituir o detector para marcas d'água visíveis em imagens.

1.1.1 Aplicações em Imagens

Marcas d'água digitais tem como uma de suas principais aplicações o uso em imagens. Devido à facilidade com que podem ser replicadas e editadas, as imagens digitais apresentam uma alta vulnerabilidade à cópia e distribuição não autorizadas. Nesse contexto, as marcas d'água desempenham um papel muito importante, fornecendo uma camada adicional de proteção ao incorporar diretamente informações relacionadas aos direitos de uso no conteúdo digital.

Fotógrafos e bancos de imagens frequentemente utilizam marcas d'água visíveis como um mecanismo preventivo contra o uso comercial não autorizado de suas criações, conforme ilustrado na Figura 1.2.

(a) *Imagem retirada de gettyimages (2024).*

(b) *Imagem retirada de shutterstock (2024).*

Figura 1.2: *Exemplos de imagens retiradas de bancos de imagens.*

Como o próprio termo sugere, as marcas d'água visíveis são perceptíveis ao olho humano e têm por finalidade indicar a propriedade ou autoria do conteúdo. Sua presença em uma imagem sinaliza a existência de restrições de uso, e a utilização sem a devida autorização pode acarretar implicações legais.

O processo de inserção de uma marca d'água visível consiste em sobrepor uma imagem semitransparente com elementos gráficos, como logotipos, texto ou uma combinação de ambos, sobre a imagem original, sem comprometer a sua visualização. Essa técnica altera os valores dos pixels de uma região da imagem hospedeira I de acordo com os valores dos pixels da marca d'água W . Se W tem dimensões $m \times n$ e a marca é aplicada a partir da posição (x_0, y_0), então a região modificada na imagem original é dada pela fórmula:

$$I'(i,j) = (1-\alpha)\cdot I(x_0+i, y_0+j) + \alpha \cdot W(i,j), \quad 0 \leq i < m,\ 0 \leq j < n,$$

onde $\alpha \in [0,1]$ representa o nível de transparência da marca d'água.

1.2 Aprendizado de Máquina

De acordo com Goodfellow *et al.* (2016), os algoritmos de ML podem ser classificados em dois tipos principais: aprendizado supervisionado e não supervisionado.

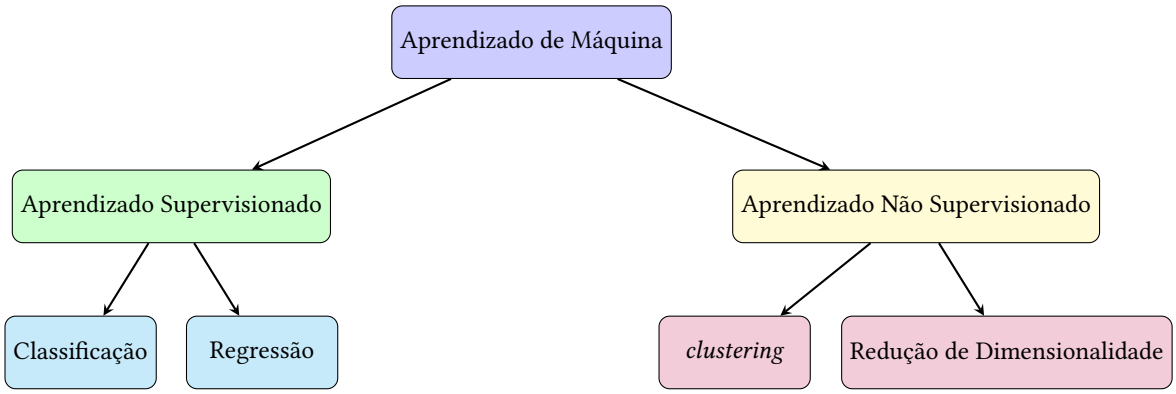


Figura 1.3: *Divisão do aprendizado de máquina em categorias principais e exemplos de problemas.*

No aprendizado supervisionado, o conjunto de dados é composto por exemplos rotulados, representados por pares $\{(x_i, y_i)\}_{i=1}^{N}$, onde $x_i \in \mathbb{R}^d\,(d > 0)$ representa um vetor de atributos, e y_i corresponde ao rótulo ou *label*. Os rótulos podem assumir diferentes formatos, como números reais, elementos de um conjunto finito de classes $\{1, 2, \ldots, C\}$, ou dados mais complexos, como vetores ou matrizes. O objetivo desse tipo de abordagem é que o modelo "aprenda" uma função capaz de mapear corretamente as entradas x_i para os respectivos valores de saída y_i, de modo a generalizar o conhecimento adquirido para novos dados ainda não observados.

Como mostra a Figura 1.3, no aprendizado supervisionado, as tarefas podem ser divididas em dois grupos: classificação e regressão. A classificação é utilizada quando o objetivo é categorizar os dados em classes ou grupos distintos. Nesse caso, o modelo aprende a associar as amostras de entrada a rótulos discretos, como, por exemplo, classificar e-mails como "spam"ou "não spam", ou identificar objetos em imagens. Já a regressão é aplicada em cenários que exigem a previsão de valores contínuos, buscando estimar uma variável numérica a partir de um conjunto de atributos fornecidos como entrada.

Por outro lado, o aprendizado não supervisionado utiliza um conjunto não rotulado $\{x_i\}_{i=1}^{N}$, com o intuito de identificar padrões ou estruturas subjacentes nos dados. Uma aplicação comum nessa abordagem é o *clustering*, no qual o modelo agrupa os vetores de características em *clusters* ou categorias com base em suas similaridades. Outra aplicação relevante é a redução de dimensionalidade, que visa transformar os dados de entrada x_i em vetores de saída com menor número de atributos, preservando informações essenciais.

Além do aprendizado supervisionado e não supervisionado, existem outras categorias de algoritmos de ML. No entanto, devido à delimitação do escopo deste trabalho, essas abordagens não serão abordadas em detalhes.

1.3 Redes Neurais

Redes neurais são uma classe de algoritmos de ML inspirada na estrutura do cérebro humano. Esse conceito foi introduzido por McCulloch e Pitts (1943), que propuseram a construção de um modelo computacional utilizando circuitos elétricos para explicar como os neurônios biológicos processam informações. A partir dessa contribuição, novos modelos foram propostos, onde neurônios passaram a ser representados como nós interconectados, ativados com base em um conjunto de valores de entrada.

1.3.1 Perceptron

O perceptron, ou neurônio artificial, é um dos exemplos mais simples de rede neural, introduzido por Rosenblatt, 1958. A Figura 1.4 ilustra um exemplo de perceptron.

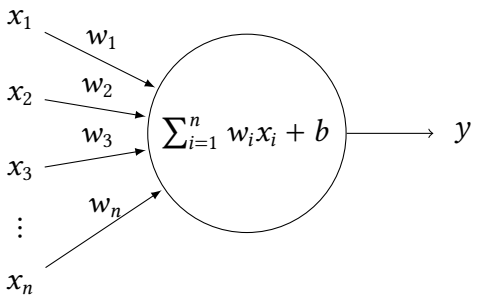


Figura 1.4: *Perceptron simples com entradas $x_1, x_2, x_3, \ldots x_n$, pesos $w_1, w_2, w_3, \ldots w_n$, bias b e função de ativação f.*

Podemos representá-lo como uma função que recebe como entrada um vetor $X = (x_1, x_2, \ldots, x_n)$, calcula uma combinação linear com pesos $W = (w_1, w_2, \ldots, w_n)$ e obtém um valor de saída y, dado por:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

onde f é denominada função de ativação e o termo b é conhecido como viés (*bias*). O viés desloca o resultado da função de ativação do seu valor na origem. No modelo proposto por Rosenblatt, a função de ativação utilizada era uma função degrau com um valor de limiar (*threshold*) t:

$$f(x) = \begin{cases} 1, & \text{se } x > t \\ 0, & \text{caso contrário.} \end{cases}$$

É possível ainda combinar múltiplos perceptrons em uma camada, como ilustrado na Figura 1.5, para resolver problemas mais complexos. Note que cada nó possui o seu próprio conjunto de pesos, ou seja, no total, existem $n \times m$ pesos e m valores de *bias*.

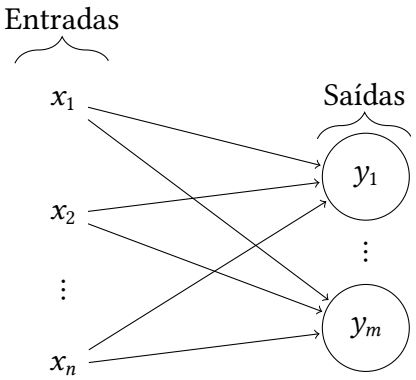


Figura 1.5: *Conjunto de perceptrons organizados em uma camada.*

Devido à linearidade do modelo e à saída binária, o perceptron é capaz de resolver apenas problemas definidos em espaços linearmente separáveis.

1.3.2 Multilayer Perceptron

De maneira análoga à organização de neurônios no cérebro humano, múltiplos perceptrons podem ser combinados sequencialmente. Dessa forma, a saída de uma camada é utilizada como entrada para a camada subsequente, formando uma arquitetura de múltiplas camadas, conforme exemplificado na Figura 1.6.

Uma rede neural *Multilayer Perceptron* (MLP) consiste em uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. Os neurônios presentes nas camadas ocultas recebem como entrada a saída da camada anterior. Não existem conexões diretas entre camadas não adjacentes ou conexões entre nós de uma mesma camada. Além disso, a transmissão de sinais ocorre de forma unidirecional: cada camada envia informações apenas para a camada seguinte, e cada nó de uma camada está conectado com todos os nós da próxima camada. Devido a essas características, essa arquitetura é chamada de *Feed-Forward Network* (FFN).

Formalmente, uma MLP pode ser descrita por meio de operações matriciais. A entrada é representada por um vetor $x \in \mathbb{R}^N$, enquanto a saída é definida como $y \in \mathbb{R}^M$. Cada camada $j \in \{1, 2, \dots, l\}$ é composta por s_j nós (neurônios), uma matriz de pesos $W^{(j)}$ de dimensões $s_j \times s_{j-1}$, um vetor de vieses $b^{(j)}$, uma função de ativação $f^{(j)}$ e um vetor de saída $z^{(j)} \in \mathbb{R}^{s_j}$. Sendo $x = z^{(0)}$ e $y = z^{(l)}$, o comportamento da rede é descrito recursivamente pela expressão:

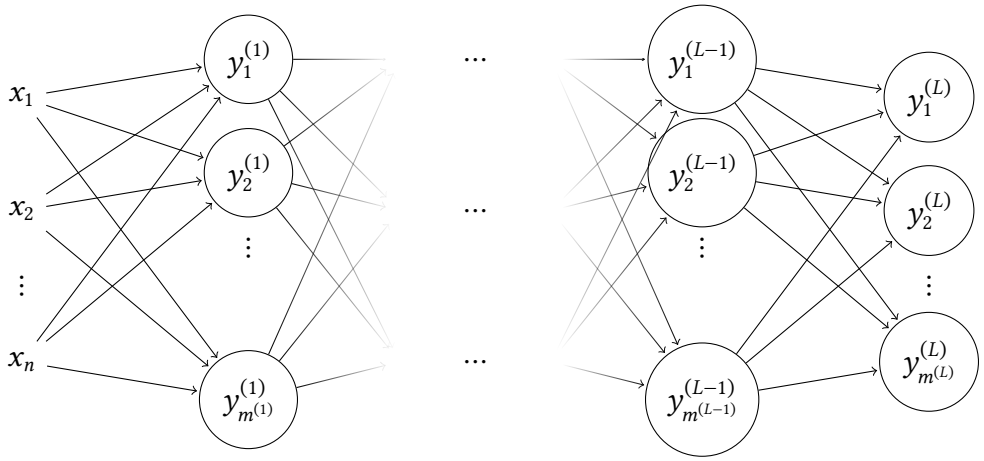


Figura 1.6: *Exemplo de um Multilayer Perceptron (MLP) com camadas ocultas. (adaptado)*

$$z^{(j)} = f^{(j)}\left(W^{(j)} z^{(j-1)} + b^{(j)}\right), \quad j \in \{1, 2, \ldots, l\}.$$

Essa equação define o processo iterativo de cálculo da rede neural, conhecido como *forward propagation*. Note como o vetor de entrada alimenta a rede e os sinais são propagados pelas camadas até a saída.

O aprendizado de uma FFN é baseado no treinamento com um conjunto de dados rotulados. O objetivo dessa etapa é encontrar os pesos $W^{(j)}$ que minimizem as diferenças entre os valores esperados y_i e os valores $\hat{y}_i$ previstos pela rede, dado um exemplo x_i.

1.3.3 Função de Custo

A função de custo quantifica o erro do modelo, isto é, mede a diferença entre os valores previstos e os valores reais fornecidos no conjunto de dados de treinamento. Essa função fornece uma métrica que agrega o erro global e funciona como um guia para a otimização dos parâmetros do modelo, como pesos e vieses.

Seja $D = \{(x_i, y_i)\}_{i=1}^{N}$ um conjunto de dados com N exemplos, onde x_i é o vetor de entrada de dimensão d e y_i é o rótulo correspondente. Considere ainda θ como o conjunto de parâmetros ajustáveis da rede. A função de custo $L(\theta)$ pode ser expressa por:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, \hat{y}_i),$$

onde $\hat{y}_i$ é o valor previsto pelo modelo para a entrada x_i, e $\ell(y_i, \hat{y}_i)$ é a função de perda, que avalia o erro para um único exemplo.

A função de perda ℓ varia de acordo com o tipo de problema abordado. Para tarefas de regressão, a função de perda frequentemente utilizada é o erro quadrático médio (*Mean*

Squared Error - MSE), que calcula o erro como o quadrado da diferença entre os valores previstos e os valores reais. A função de custo correspondente é dada por:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

Para tarefas de classificação, a função de perda comumente utilizada é a *Cross-Entropy Loss*, que mede a diferença entre as distribuições probabilísticas das classes previstas e das classes reais. Em problemas de classificação binária, isto é, quando o rótulo assume apenas dois valores, 0 ou 1, a entropia cruzada pode ser expressa como:

$$L(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right].$$

As funções de custo não apenas avaliam o desempenho do modelo, mas também guiam o processo de ajuste dos parâmetros da rede durante o treinamento. Algoritmos de otimização, como o gradiente descendente (*Gradient Descent* - GD), utilizam a função de custo para minimizar o erro e melhorar a performance do modelo.

1.3.4 Função de Ativação

Um dos elementos essenciais das redes neurais são as funções de ativação, responsáveis por introduzir não-linearidade no modelo. Essas funções permitem que modelos aprendam funções complexas, algo que seria impossível em uma rede composta apenas por *Fully Connected Layers* (FCL), uma vez que a composição de operações lineares continua sendo uma função linear.

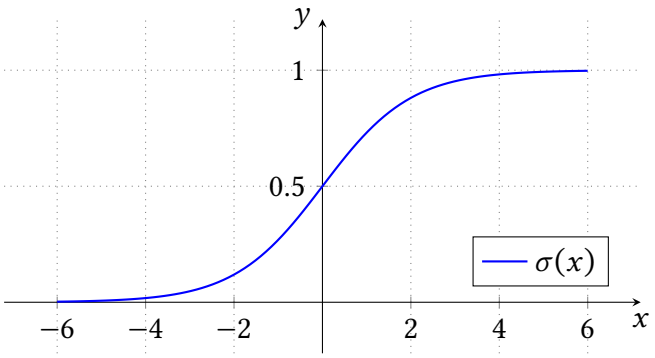
Dentro da arquitetura das redes, a função de ativação é aplicada à saída de cada neurônio, transformando o valor resultante antes de propagá-lo para a próxima camada. Isto é, dado um vetor de entrada x, uma matriz de pesos W, um vetor de viés b e uma função de ativação f, a saída final z de um neurônio pode ser expressa por:

$$z = f(Wx + b).$$

Há uma grande variedade de funções de ativação utilizadas em redes neurais, cada uma com características específicas que influenciam o desempenho e a eficiência do treinamento:

Sigmoide: A função Sigmoide mapeia os valores para o intervalo $[0, 1]$ e é definida por:

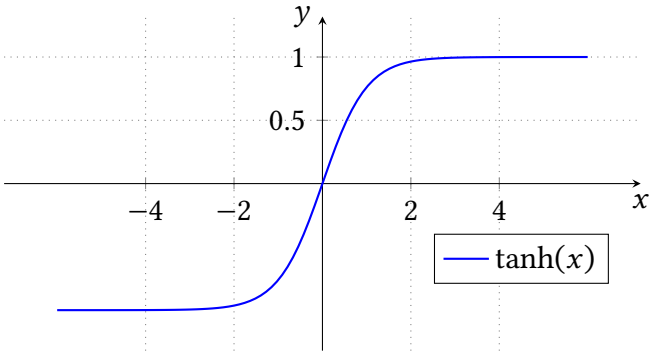
$$\sigma(x) = \frac{1}{1 + e^{-x}},$$



Essa função é muito comum em problemas de classificação binária, pois permite interpretar a saída como uma probabilidade. No entanto, pode sofrer com o desaparecimento de gradientes (*Vanishing Gradient*), pois a derivada da função assume valores muito pequenos para entradas distantes de zero.

Tangente Hiperbólica (Tanh): A função tangente hiperbólica mapeia os valores para o intervalo $(-1, 1)$ e é definida por:

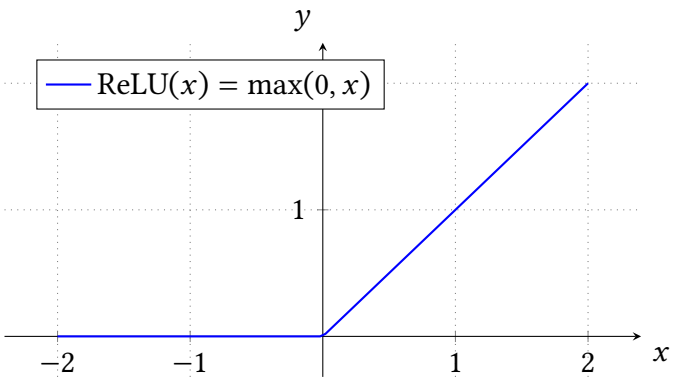
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$



Semelhante à função Sigmoide, a função Tanh também pode sofrer do problema de *Vanishing Gradient*. Contudo, por ser centrada em zero, ela geralmente proporciona uma convergência mais rápida durante o treinamento.

Unidade Linear Retificada (ReLU): A função ReLU é definida por:

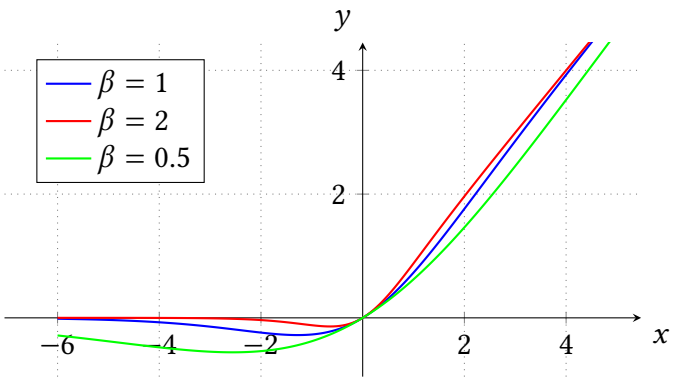
$$\text{ReLU}(x) = \max(0, x).$$



Ela mapeia valores negativos para zero e mantém os valores positivos inalterados. A ReLU é amplamente utilizada em redes neurais devido à sua eficiência computacional. Além disso, ajuda a mitigar o problema do *Vanishing Gradient*. Porém, pode causar a "morte"de neurônios quando as entradas resultam em valores negativos constantes, levando a gradientes nulos.

Swish A função Swish é uma função de ativação mais recente que tem demonstrado desempenho superior em diversas aplicações de aprendizagem profunda. Ela é definida por:

$$\text{swish}(x) = x \cdot \sigma(\beta x),$$



onde $\sigma(x)$ é a função Sigmoide e β é um parâmetro opcional que controla a forma da função. Quando $\beta = 1$, a função Swish simplifica-se para:

$$\text{swish}(x) = x \cdot \sigma(x).$$

Diferentemente da ReLU, que anula todos os valores negativos, a Swish permite que esses sinais sejam transmitidos de forma atenuada. Isso pode resultar em uma melhor propagação de gradiente e, consequentemente, em um treinamento mais eficiente da rede neural.

1.3.5 Retropropagação e Gradiente Descendente

Retropropagação, ou *Backpropagation*, é um algoritmo utilizado para calcular os gradientes da função de custo em relação aos parâmetros de uma rede neural. Introduzido por Rumelhart *et al.*, 1986, o método revolucionou o campo das redes neurais, possibilitando o treinamento eficiente de modelos com grandes números de parâmetros.

O algoritmo recebe esse nome porque o cálculo dos gradientes ocorre de forma reversa: inicia-se na última camada e propaga-se para as camadas anteriores. Após o *feedforward pass*, isto é, o processamento da entrada até a saída, calcula-se a função de custo. Em seguida, o erro é propagado de volta pelas camadas da rede no *backward pass*, como ilustrado na Figura 1.7. Nesse processo, utiliza-se a regra da cadeia para calcular de forma iterativa o gradiente da função de custo em relação a cada parâmetro.

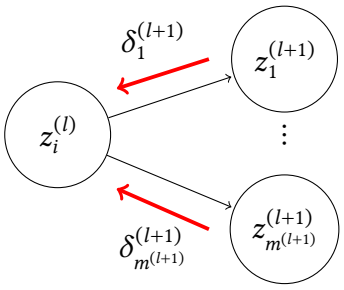


Figura 1.7: *Backprogapation dos erros δ_i da camada $l+1$ para a camada l (adaptado).*

A retropropagação facilita o treinamento de redes neurais através do método de gradiente descendente. De forma simplificada, este algoritmo de otimização minimiza a função de custo $L(\theta)$ atualizando os parâmetros do modelo na direção oposta ao gradiente $\nabla L(\theta)$, de acordo com a fórmula:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)}),$$

onde η representa a taxa de aprendizado, que determina o tamanho do passo na direção que minimiza a função de perda, e t denota uma iteração do processo de otimização.

O método de gradiente descendente tradicional é simples e intuitivo, mas depende do processamento do conjunto de dados de treinamento inteiro para realizar uma única atualização dos parâmetros. Por isso, ele pode se tornar muito lento e inviável para uma grande quantidade de dados.

Para mitigar essas limitações, foram criadas variantes do GD que reduzem o custo computacional do processo de otimização. Um exemplo simples é o gradiente descendente estocástico, ou *Stochastic Gradient Descent* (SGD), que utiliza um subconjunto aleatório de tamanho reduzido, ou até mesmo um único par do conjunto de treinamento. Com essa alteração, o modelo converge mais rápido, mas também fica suscetível a maior variância entre iterações.

Essas oscilações podem ser reduzidas com a introdução de métodos que suavizam a atualização de parâmetros. Uma modificação do SGD que leva em consideração o histórico das iterações anteriores para reduzir a variância durante o treinamento é o *Momentum*. Esse método introduz um vetor de velocidade v que mantém uma média dos gradientes anteriores e realiza a atualização de parâmetros da seguinte forma:

$$v^{(t+1)} = \beta v^{(t)} - \eta \nabla L(\theta^{(t)}),$$
$$\theta^{(t+1)} = \theta^{(t)} + v^{(t+1)},$$

onde β é o coeficiente de *Momentum* (geralmente 0.9).

Técnicas de adaptação da taxa de aprendizado também podem ser utilizadas para aprimorar o processo de otimização e convergência de modelos. Métodos como o *Adagrad*, *RMSProp* e *Adam* ajustam esse parâmetro durante o treinamento de forma dinâmica. O *Adagrad* adapta a taxa de aprendizado de cada parâmetro individualmente, acumulando gradientes passados e ajustando-os proporcionalmente.

Já o *RMSProp* introduz um mecanismo de decaimento exponencial, permitindo que o modelo mantenha um histórico limitado dos gradientes anteriores. Enquanto o *Adam* combina as ideias de *Momentum* e *RMSProp*, utilizando momentos de primeira e segunda ordem dos gradientes para realizar atualizações mais precisas.

1.3.6 Treinamento

O treinamento de uma rede neural é o processo que realiza os ajustes de parâmetros do modelo e permite o aprendizado de uma tarefa, como classificação ou regressão. Esse processo envolve três elementos principais: um conjunto de dados, uma função de custo e um algoritmo de otimização.

Inicialmente, o conjunto de dados é dividido em três partes: treino, validação e teste.

Conjunto de treino: é a parte dos dados utilizada para ajustar os pesos e vieses do modelo. Durante o processamento deste conjunto, o modelo aprende a reconhecer padrões e estabelecer relações entre os dados de entrada e os valores esperados de saída.

Conjunto de validação: tem como objetivo monitorar o desempenho do modelo durante o treinamento. Ele é utilizado para obter uma estimativa da performance do modelo em dados não vistos, fornecendo uma medida de sua capacidade de generalização ainda durante o treinamento. Essa estimativa é útil para identificar problemas de *overfitting*, isto é, quando o modelo memoriza os exemplos do conjunto de treino em vez de aprender padrões gerais. Além disso, os dados de validação também são utilizados na otimização de hiperparâmetros, como a taxa de aprendizado.

Conjunto de teste: é usado somente após a conclusão do treinamento para avaliar o desempenho final do modelo. Como ele é independente dos conjuntos de treino e validação,

ou seja, contém dados nunca vistos pelo modelo, o conjunto de teste oferece uma avaliação não enviesada do modelo em dados novos.

Não existe uma regra fixa para a divisão dos dados, mas, em geral, utiliza-se 80% para treinamento, 10% para validação e 10% para teste.

Após a separação dos dados e a escolha de um método de otimização e uma função de custo apropriada, o treinamento pode ser descrito como um processo iterativo, em que cada iteração é chamada de época. Durante cada época, o modelo processa o conjunto de treino inteiro, geralmente dividido em *batches* de tamanho pré-definido. O processamento de um *batch* consiste no *forward pass* do subconjunto de dados e a avaliação da função de custo. Em seguida, no *backward pass*, o modelo atualiza seus parâmetros, guiado pelo método de otimização escolhido. O treinamento é finalizado quando um critério de parada é atingido. Esse critério pode ser o número máximo de épocas ou alguma decisão baseada nas métricas de treinamento, como a identificação de *overfitting* baseada no desempenho do conjunto de validação.

1.3.7 Avaliação de Modelos

Neste trabalho, abordamos um problema de classificação binária, em que podemos categorizar os exemplos do conjunto de dados em duas classes: positiva (imagem que contém marca) e negativa (imagem sem marca).

Em situações como essa, é comum utilizarmos uma matriz de confusão, como ilustrada na Tabela 1.1, para agregar a quantidade de acertos e erros do modelo. Nessa tabela, podemos ver que a diagonal principal contém a quantidade de acertos do modelo, enquanto as demais células indicam a quantidade de erros.

	Predito Negativo	**Predito Positivo**
Real Negativo	Verdadeiro negativo (TN)	Falsos positivo (FP)
Real Positivo	Falso negativo (FN)	Verdadeiros positivo (TP)

Tabela 1.1: *Matriz de confusão para classificação binária.*

A partir desses valores, podemos calcular diversas métricas de avaliação, como acurácia, precisão, sensibilidade, especificidade e F1.

Acurácia: avalia o percentual de acertos em relação ao total de amostras. Utilizando os valores da matriz de confusão, a acurácia pode ser calculada pela fórmula:

$$\text{Acurácia} = \frac{\text{VP} + \text{VN}}{\text{VP} + \text{VN} + \text{FP} + \text{FN}}$$

Precisão: avalia o percentual de acertos em relação a todas as amostras classificadas como positivas. A precisão é calculada pela fórmula:

$$\text{Precisão} = \frac{\text{VP}}{\text{VP} + \text{FP}}$$

Sensibilidade: avalia a capacidade do modelo de classificar corretamente as amostras positivas. Ela é calculada como a razão entre os verdadeiros positivos e a soma dos verdadeiros positivos e falsos negativos:

$$\text{Revocação} = \frac{\text{VP}}{\text{VP} + \text{FN}}$$

Especificidade: avalia a capacidade do modelo de classificar corretamente as amostras negativas. Ela é calculada como a razão entre os verdadeiros negativos e a soma dos verdadeiros negativos e falsos positivos:

$$\text{Especificidade} = \frac{\text{VN}}{\text{VN} + \text{FP}}$$

F1: combina precisão e sensibilidade em uma única métrica, calculando a média harmônica das duas métricas:

$$F1 = 2 \cdot \frac{\text{Precisão} \cdot \text{Revocação}}{\text{Precisão} + \text{Revocação}}$$

1.4 Redes Neurais Convolucionais

Redes Neurais Convolucionais, ou *Convolutional Neural Networks* (CNN), são redes *feed-forward* baseadas no uso de filtros de convolução. Diferentemente das redes totalmente conectadas, as CNNs aproveitam a estrutura espacial dos dados, reduzindo significativamente o número de parâmetros a serem treinados.

Essas redes são compostas por uma combinação de camadas de convolução, camadas de pooling e camadas totalmente conectadas.

1.4.1 Convolução

A camada de convolução é responsável por extrair características, ou *features*, de regiões locais do dado de entrada, capturando diferentes níveis de abstração, desde formas básicas, como bordas, até padrões mais complexos, como objetos completos ou rostos.

Cada camada convolucional contém uma coleção de filtros (*kernels*), que deslizam sobre a entrada e produzem mapas de características (*feature maps*). A convolução consiste em multiplicar os valores dos pixels da entrada por um filtro e somar os resultados, conforme ilustrado na Figura 1.8.

A operação de convolução pode ser descrita matematicamente como:

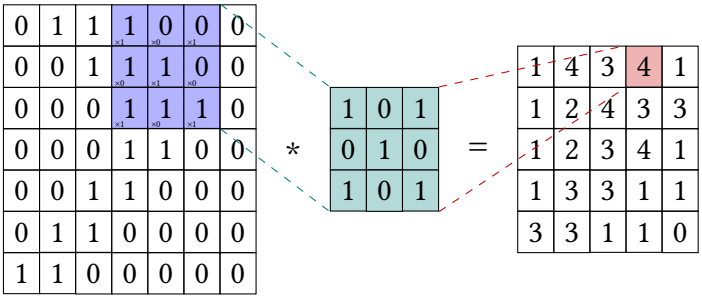


Figura 1.8: *Exemplo de convolução com um filtro de tamanho 3. (adaptado)*

$$o_{i,j}^{(h)} = \sum_{l=1}^{m} \sum_{k=1}^{n} K_{k,l} z_{i+l,j+k}^{(h-1)} + b,$$

onde K é o filtro de convolução de tamanho $m \times n$, $z^{(h-1)}$ é a saída da camada anterior, $o^{(h)}$ é o mapa de características resultante e b é o viés.

Os pesos de K são compartilhados por todas as regiões da entrada, permitindo que o mesmo padrão seja detectado independentemente de sua posição. Essa característica confere às CNNs a propriedade de invariância à translações.

Dois parâmetros influenciam o resultado da convolução diretamente: *stride* e *padding*. O *stride* define o número de pixels que o filtro se desloca a cada passo. Um stride igual a 1 cobre todas as posições possíveis, enquanto strides maiores fazem o filtro pular posições, resultando em mapas de características menores. Já o *padding* adiciona pixels ao redor da entrada. Sem *padding*, as dimensões do mapa de características diminuem após cada camada convolucional, podendo causar perda de informações localizadas nas bordas.

Em geral, uma camada convolucional possui múltiplos filtros, como ilustrado na Figura 1.9. O número de canais na saída é diretamente proporcional ao número de filtros aplicados, enquanto as dimensões espaciais (altura e largura) são determinadas pelo tamanho dos filtros e pelas configurações de deslocamento do filtro sobre a entrada.

À medida que a profundidade da rede aumenta, o número de filtros aplicados também aumenta, enquanto as dimensões espaciais dos mapas de características tendem a diminuir, reduzindo o custo computacional. Nessa organização, camadas mais profundas da rede são capazes de aprender padrões mais complexos, enquanto camadas iniciais capturam características mais simples e locais.

As camadas de convolução são acompanhadas por funções de ativação, que introduzem não-linearidade ao modelo e, em geral, são seguidas por camadas de *pooling*.

1.4.2 *Pooling*

As camadas de *pooling* combinam features adjacentes da saída da camada anterior em um único valor, permitindo reduzir a dimensão dos dados e, consequentemente, diminuir

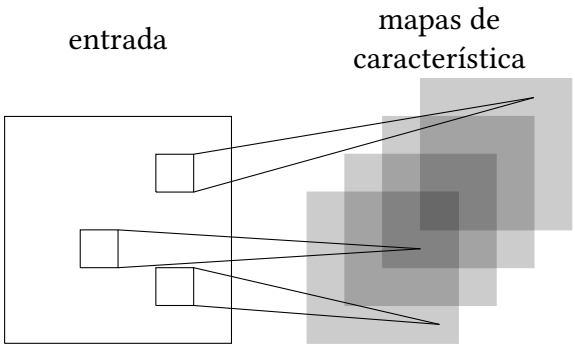


Figura 1.9: *Ilustração de uma camada convolucional com múltiplos filtros. Cada filtro gera um mapa de características distinto.*

o número de parâmetros em uma rede neural.

Elas funcionam de forma semelhante às camadas de convolução, com filtros de tamanho fixo deslocando-se sobre os dados de entrada, no entanto não possuem pesos ou parâmetros. O *max pooling*, por exemplo, seleciona o valor máximo de cada região do mapa de características, destacando as ativações mais "fortes" e tornando o modelo mais robusto a variações pequenas nos dados de entrada, como ruído ou deslocamentos. Já o *average pooling* calcula a média dos valores de uma região, gerando uma representação mais suave.

A Figura 1.10 ilustra um exemplo de *max pooling* com filtro de tamanho 2.

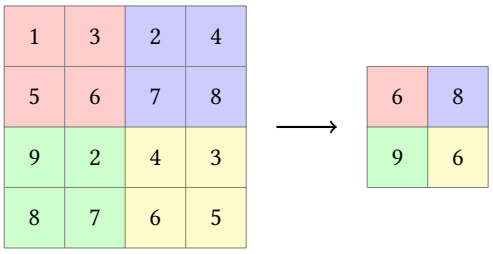


Figura 1.10: *Exemplo de max pooling com stride 2.*

1.4.3 Camadas Totalmente Conectadas

Os mapas de características são "achatados"em um vetor unidimensional e utilizados como entrada para camadas totalmente conectadas, responsáveis por agregar as informações extraídas pelas camadas anteriores e produzir a saída final da rede. A Figura 1.11 ilustra a arquitetura genérica de uma CNN.

1.5 Arquiteturas

A arquitetura se refere a organização e disposição das camadas de uma rede neural. Isso inclui a quantidade e o tipo de camadas e a forma como essas camadas estão conectadas.

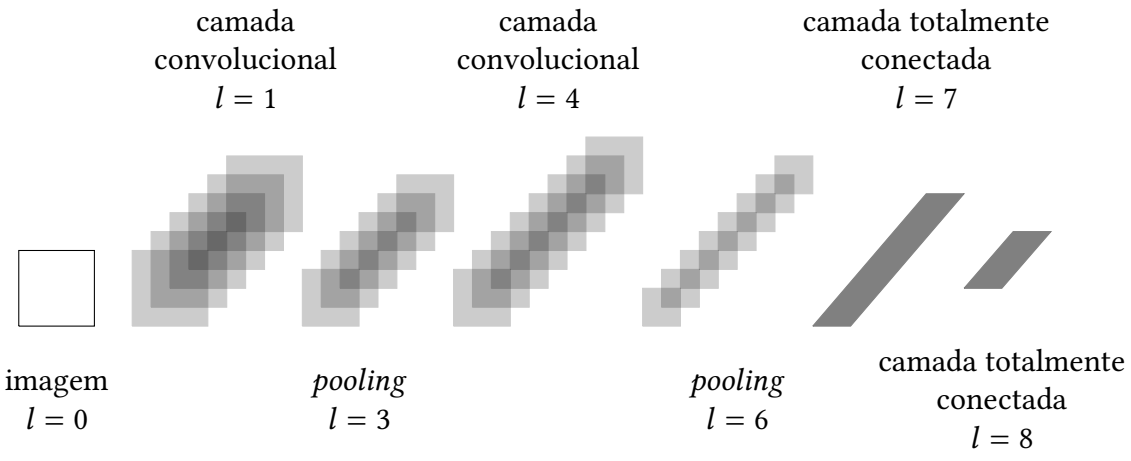


Figura 1.11: *Ilustração de uma CNN genérica com camadas de convolução, pooling e camadas totalmente conectadas (adaptado).*

As primeiras arquiteturas seguiam uma organização que veio a se tornar um padrão para redes convolucionais profundas: camadas de convolução acompanhadas de funções de ativação, seguidas por camadas de pooling e, no fim, camadas totalmente conectadas. Um exemplo que segue esse padrão é a AlexNet, apresentada por Krizhevsky *et al.*, 2012. A principal inovação dessa arquitetura foi o incremento do número de camadas, o uso de ativações ReLU, normalização local, dropout e o treinamento em múltiplas GPUs, o que resultou em melhorias de desempenho significativas em benchmarks de visão computacional.

O uso de arquiteturas mais profundas também trouxe desafios para o treinamento de redes, pois com maior número de camadas os modelos ficam suscetíveis ao desaparecimento de gradientes. Para resolver esse problema, He *et al.* (2015) apresentaram a ResNet. Essa arquitetura propôs o conceito de conexões residuais, que permitem que gradientes se propaguem mais facilmente ao longo das camadas, possibilitando o treinamento de redes com mais camadas.

Outra arquitetura relevante é a EfficientNet (Tan e Le, 2020), que utiliza uma técnica chamada *compound scaling*. Essa estratégia adapta a profundidade da rede, largura dos filtros e dimensões da imagem de entrada de maneira balanceada para obter o melhor ganho de desempenho. Como resultado, se obeteve redes menores e mais eficientes em termos computacionais, mas que mantêm alta performance na classificação de imagens.

Mais recentemente, surgiu a ConvNeXt (Liu *et al.*, 2022) com a combinação de camadas de convolução e blocos que se inspiram no funcionamento de Transformers, um tipo de rede neural originalmente desenvolvido para processamento de linguagem natural.

Capítulo 2

Metodologia

Neste capítulo detalharemos o desenvolvimento do projeto e a metodologia aplicada. Serão abordadas as etapas de coleta e preparação de dados, definição dos modelos, treinamento e avaliação.

2.1 Ferramentas Utilizadas

No desenvolvimento deste trabalho, utilizamos ferramentas para coleta de dados, pré-processamento, treinamento e avaliação de modelos de aprendizado de máquina. A seguir, listamos as principais bibliotecas e ferramentas utilizadas.

- **Selenium**[1]: uma ferramenta que realiza a automação de tarefas em navegadores web, permitindo a interação com páginas dinâmicas.
- **BeautifulSoup**[2]: biblioteca Python que analisa documentos HTML e permite extrair dados de páginas web.
- **Keras**: biblioteca Python que oferece uma interface de alto nível para construção, treinamento e avaliação de modelos de redes neurais.
- **keras-tuner**: *framework* que simplifica o processo de otimização de hiperparâmetros. Ela é integrada com o Keras e oferece uma interface simples para algoritmos de busca complexos.
- **Kaggle**: a plataforma Kaggle oferece um ambiente de execução para *notebooks Jupyter* equipado com uma GPU NVIDIA Tesla P100 com 16 GB de RAM e CPU Intel Xeon 2.20 GHz de 4 núcleos reais com 32 GB de RAM.

[1] Selenium with Python

[2] Beatiful Soup Documentation

2.2 Coleta de Dados

Para este trabalho, foi utilizado um conjunto de dados composto por imagens de anúncios de imóveis de diversas plataformas online de imobiliárias. A coleta foi realizada utilizando bibliotecas Python de *web scraping* que fazem a automação do processo de navegação e extração de dados de páginas web.

Foram selecionados websites de oito imobiliárias localizadas nas cidades de São Paulo, Rio de Janeiro e Belo Horizonte, abrangendo anúncios de imóveis que se classificam como casas ou apartamentos. Essa delimitação foi feita com o objetivo de priorizar imagens de cômodos internos ou, no caso de apartamentos, de áreas comuns do condomínio. Além disso, a escolha incluiu imobiliárias que utilizam diferentes tipos e posicionamentos de marcas d'água, de forma a aumentar a diversidade do conjunto de dados e preparar o modelo para lidar com essas variações.

Após a seleção das fontes, realizamos a extração das URLs das imagens dos anúncios. Nessa etapa, utilizamos a biblioteca Selenium para simular a interação de um usuário com o website – como fazer rolagem de páginas e cliques em botões – para navegar em páginas dinâmicas. Em conjunto com Selenium, utilizamos também a biblioteca BeautifulSoup para analisar o conteúdo HTML das páginas coletadas e extrair as URLs das imagens.

A partir da lista de URLs, realizamos o download de 68.712 imagens com dimensões, resoluções e proporções diversas. Por essa razão, o conjunto de dados passou por algumas etapas de seleção e preparação que serão descritas a seguir.

As Figuras 2.1 e 2.2 mostram exemplos de imagens coletadas.

2.3 Preparação dos Dados

Para garantir uma certa padronização do conjunto de dados, realizamos algumas etapas de pré-processamento.

Análise Manual

Considerando que as imagens foram coletadas de diferentes fontes e produzidas utilizando equipamentos diversos, era esperado encontrar uma grande variedade de dimensões e qualidade. Por esse motivo, foi realizada uma inspeção manual de um subconjunto das imagens. Durante essa análise, notamos que aquelas com tamanho muito reduzido (ocupando menos de 64kB de armazenamento) apresentavam baixa qualidade, dificultando a distinção de marcas d'água mesmo por meio de inspeção humana.

Dessa forma, decidimos remover todas as imagens com tamanho inferior a 64kB, obtendo um *dataset* balanceado com 60.258 imagens, sendo 31.365 amostras com marcas d'água e 28.893 imagens limpas (sem marca).

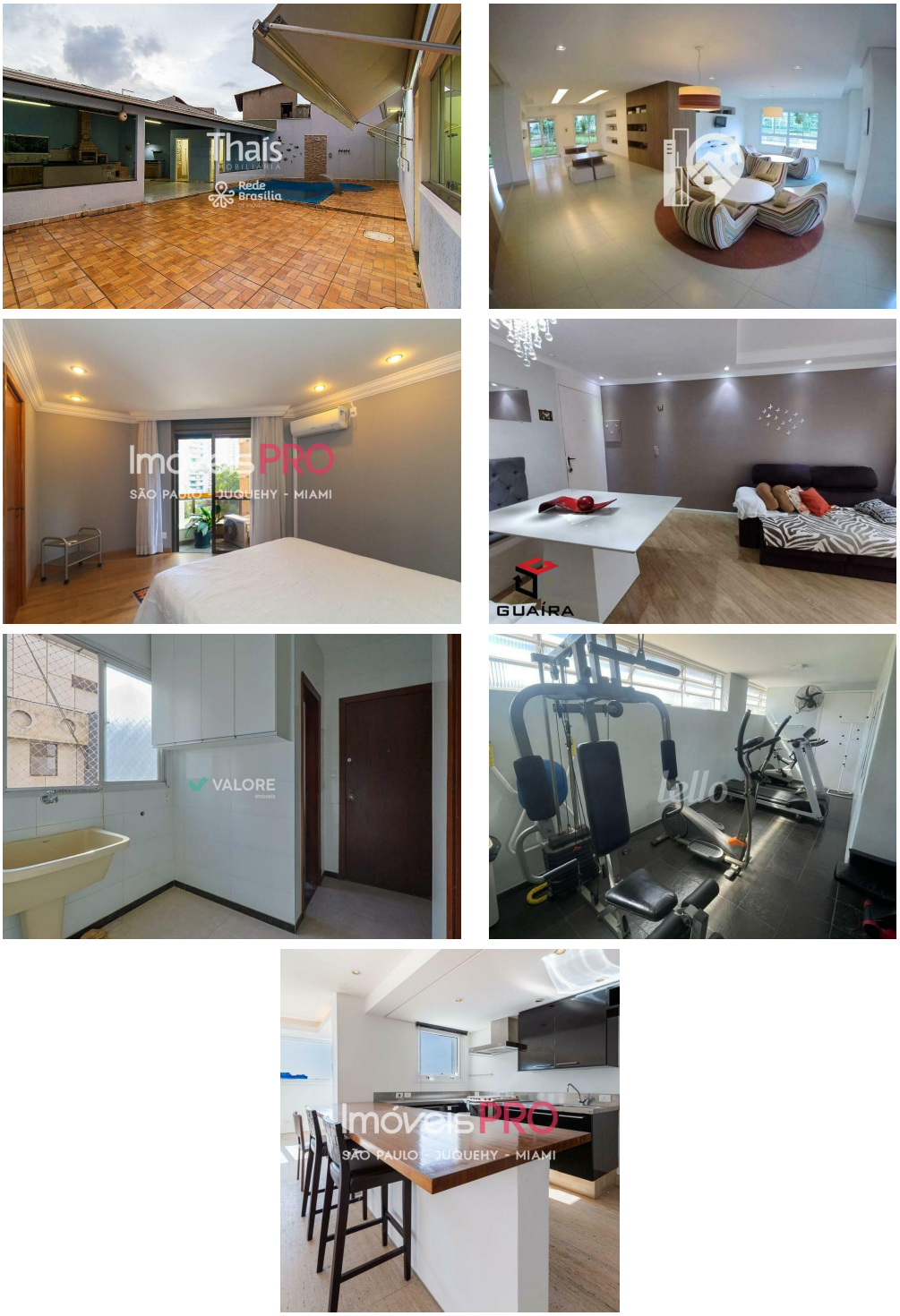


Figura 2.1: *Exemplos de imagens com marca d'água do conjunto de dados.*

Figura 2.2: *Exemplos de imagens sem marca d'água do conjunto de dados.*

Redimensionamento

O redimensionamento das imagens é necessário para garantir a compatibilidade com os modelos de CNN. Para este trabalho, todas as imagens foram redimensionadas para dimensões fixas de 380×380 pixels, utilizando interpolação bilinear. Esse tamanho permite preservar bastante informações visuais para a detecção de marcas d'água e aproveita ao máximo o poder computacional da GPU.

A Figura 2.3 mostra um exemplo de imagem redimensionada.

Imagem original.

Imagem redimensionada para 380×380.

Figura 2.3: *Exemplo de redimensionamento de imagem.*

Apesar de existirem outras alternativas para redimensionamento, como o corte ou preenchimento com bordas, optamos por distorcer as imagens para uma proporção 1:1 pois essa opção apresentou os melhores resultados durante experimentação.

Separação de Dados

O conjunto de dados foi dividido em três partes: 80% destinados ao treinamento, 10% para validação e 10% para teste, resultando na distribuição a seguir.

- **Treinamento**: 48.203 imagens.
- **Validação**: 6.025 imagens.
- **Teste**: 6.026 imagens.

O conjunto de treinamento é utilizado para ajustar os parâmetros do modelo, enquanto o conjunto de validação serve para monitorar seu desempenho durante o treinamento, prevenindo também o *overfitting*. Já o conjunto de teste permanece isolado do treinamento e só é utilizado no final para obter uma métrica de performance do modelo.

Aumento de Dados

O aumento de dados, ou *data augmentation*, consiste na aplicação de transformações às imagens do conjunto de treinamento, como rotações, reflexões e recorte, com o objetivo de ampliar artificialmente a diversidade do conjunto de dados sem a necessidade de

coletar novas imagens. Essa etapa de pré-processamento pode melhorar a generalização do modelo para dados nunca vistos.

Figura 2.4: *Exemplos de transformações do aumento de dados.*

Neste trabalho, aplicamos rotações e reflexões horizontais, como mostra a Figura 2.4. Essas transformações preservam bem as imagens, ao mesmo tempo que simulam mudanças que podem ser encontradas nas imagens de anúncios de imóveis online.

2.4 Aprendizado por Transferência

No início do projeto, desenvolvemos uma versão reduzida de uma arquitetura ResNet para classificação das imagens e determinar se havia ou não uma marca d'água. Porém, o modelo apresentou alto *underfitting*, ou seja, não conseguiu aprender a relação entre as imagens e as classes, demonstrando assim, que o problema era complexo demais para ser resolvido com uma arquitetura simples. Diante disso, decidimos adotar a técnica de aprendizado por transferência, ou *transfer learning*, na qual utilizamo um modelo pré-treinado e ajustamos seus parâmetros para resolver uma nova tarefa.

Com o *transfer learning*, nós aproveitamos o conhecimento de uma rede neural treinada em um conjunto de dados grande e diversificado e reutilizamos esse aprendizado para a tarefa de detecção de marcas d'água. Isso é possível porque as camadas iniciais e

intermediárias de uma CNN são responsáveis por extrair padrões ou características gerais, como bordas, texturas e formas, que são úteis para diversas tarefas de visão computacional. Para aproveitar essas informações, substituímos as camadas de classificação originais e congelamos os demais pesos. Além de trocar as camadas de classificação, incluímos normalização de *batch* e *dropout* para controlar o *overfitting* do modelo durante o treinamento. Após algumas épocas do treinamento, descongelamos todas as camadas da rede e continuamos o treinamento com uma taxa de aprendizado reduzida, permitindo um ajuste fino de todos os parâmetros da rede.

Neste trabalho, utilizamos as redes ResNet, EfficientNet e ConvNeXt pré-treinada no conjunto de dados ImageNet. A Tabela 2.1 detalha as características das variantes de cada arquetetura utilizada.

Arquitetura	**Variante**	**Nº de Parâmetros (M)**	**Tamanho do Modelo (MB)**
ResNet	ResNet50	25.6	98
EfficientNet	EfficientNetV2S	21.6	88
ConvNeXt	ConvNeXtTiny	286	109

Tabela 2.1: *Características das arquiteturas utilizadas.*

2.5 Treinamento

Além da rotina de ajuste de pesos, o treinamento também é influenciado pela escolha de hiperparâmetros, isto é, configurações de "alto-nível" que controlam o processo de aprendizado e, portanto, influenciam os parâmetros do modelo. De forma simplificada, todos os valores que são definidos no início e não mudam durante o treinamento são considerados hiperparâmetros. Alguns exemplos são: número de camadas, método de otimização, taxa de aprendizado, taxa de dropout e tamanho do batch.

Não existe uma fórmula ou regra para definir valores de hiperparâmetros, portanto, é comum realizar uma busca por hiperparâmetros satisfatórios antes de treinar um modelo. Neste trabalho, utilizamos a biblioteca keras-tuner (O'Malley *et al.*, 2019) com o algoritmo *Hyperband* (Li *et al.*, 2018), que acelera a busca aleatória utilizando alocação dinâmica de recursos e parada antecida (*early-stopping*), para encontrar valores para a taxa de aprendizado, a taxa de dropout e definir o algoritmo de otimização e o agendador da taxa de aprendizado, no espaço de busca a seguir:

- **Taxa de aprendizado**: Valores entre 10^{-5} e 10^{-1}.
- **Taxa de dropout**: Valores entre 0.1 e 0.5.
- **Otimizadores**: Adam, RMSprop e SGD.
- **Agendador da taxa de aprendizado**: *CosineDecay* e *ExponencialDecay*.

Com os hiperparâmetros definidos, iniciamos a rotina de treinamento, que começa com a configuração do modelo. Utilizamos a biblioteca *Keras* para carregar modelos pré-treinados e substituímos suas camadas de classificação para resolver a tarefa de detecção

de marcas d'água. Especificamente, definimos a saída do modelo como um único valor, que, após ser processado pela função sigmoide, pode ser interpretado como a probabilidade de a imagem conter uma marca d'água.

Em seguida, configuramos o módulo de treinamento do *Keras* para utilizar os hiperparâmetros selecionados e definimos a função de perda *Binary Cross-Entropy*, apropriada para classificação binária. O treinamento foi realizado com *batches* de 64 imagens e incluímos um critério de parada antecipada (*early stopping*) om uma tolerância de 4 épocas, ou seja, o treinamento é interrompido se a métrica de validação não melhorar por 4 épocas consecutivas. Os modelos foram treinados por, no máximo, 20 épocas com os pesos originais do modelo congelados. Depois, descongelamos todas as camadas do modelo e realizamos um ajuste fino com uma taxa de aprendizado bem pequena por até 5 epócas. Após o treinamento, recuperamos os parâmetros da época em que o modelo obteve a maior acurácia no conjunto de validação.

2.6 Otimização de Limiar de Decisão e Avaliação

Em tarefas de classificação binária, a saída dos modelos geralmente é um valor contínuo entre 0 e 1, que pode ser interpretado como a probabilidade de uma amostra pertencer à classe positiva. No nosso caso, indicar se a imagem contém uma marca d'água.

Para converter esse valor em uma classe discreta (0 ou 1), é comum utilizar um limiar de decisão (*threshold*). Por padrão, esse valor costuma ser 0.5, ou seja, se a predição do modelo p for maior ou igual a 0.5, a amostra é classificada como positiva; caso contrário, é atribuída à classe negativa. No entanto, o *threshold* pode ser ajustado para otimizar uma métrica de interesse.

Neste projeto, em vez de utilizar o *threshold* padrão, avaliamos o conjunto de validação para diferentes valores de limiar distribuídos entre 0 e 1. Para cada um desses valores, calculamos o F1 e selecionamos o *threshold* que obteve o melhor desempenho. Por fim, avaliamos os modelos no conjunto de teste, calculando métricas de acurácia e F1. Além disso, geramos a matriz de confusão para observar o desempenho em cada classe.

Capítulo 3

Resultados

Neste capítulo apresentamos os resultados obtidos através da metodologia apresentada e comparamos as redes treinadas com o objetivo de determinar o modelo com melhor desempenho na tarefa de classificação de imagens com e sem marcas d'água.

3.1 Métricas

A tabela 3.1 mostra as métricas avaliadas no conjunto de validação e no conjunto de teste para os candidatos de arquiteturas.

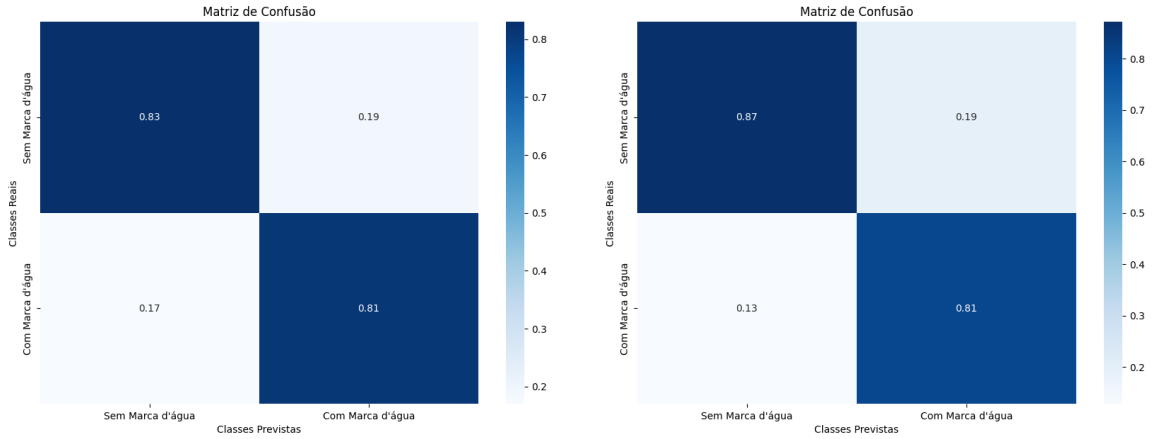
Arquitetura	**Acurácia**		**F1-score**	
	Validação	**Teste**	**Validação**	**Teste**
ResNet	0.853	0.820	0.849	0.818
EfficientNet	0.837	0.840	0.828	0.835
ConvNeXt	0.881	0.878	0.883	0.871

Tabela 3.1: *Métricas obtidas após treinamento dos modelos.*

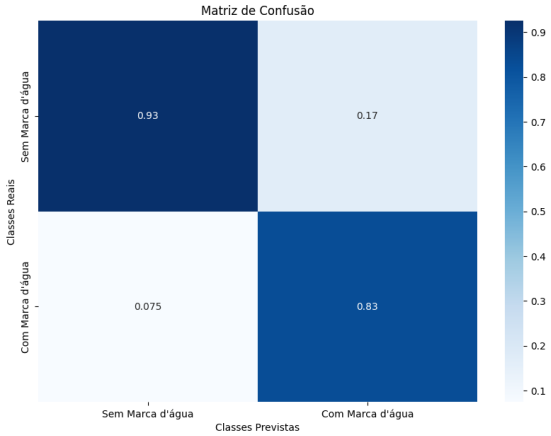
Observamos que a ConvNeXt apresentou o melhor desempenho, alcançando 87.8% de acurácia e 87.1% de F1 no conjunto de teste. Seguida pela EfficientNet, com 84% de acurácia e 83.5% de F1, por fim, a ResNet, atingindo 82% e 81.8% de acurácia e F1, respectivamente. É válido ressaltar que todas as redes apresentaram resultados satisfatórios, com acurácia e F1 acima de 80% no conjunto de teste. Além disso, a diferença entre as métricas de validação e teste é pequena em todas as arquiteturas, sendo a maior discrepância de aproximadamente 4%, no caso da acurácia da ResNet. Esse resultado indica a ausência de *overfitting* e reforça que a metodologia de treinamento adotada foi consistente e eficaz para a tarefa de classificação.

A Figura 3.1 mostra as matrizes de confusão obtidas no conjunto de teste para cada arquitetura.

Destaca-se a vantagem da ConvNeXt, com maior quantidade de verdadeiros negativos e verdadeiros positivos. A ResNet, por outro lado, apresenta maior quantidade de falsos



(a) *Matriz de confusão da ResNet* **(b)** *Matriz de confusão da EfficientNet*



(c) *Matriz de confusão da ConvNeXt*

Figura 3.1: *Matrizes de confusão dos modelos treinados.*

positivos e falsos negativos. A EfficientNet, embora apresente resultados intermediários, ainda se mantém próxima das demais arquiteturas.

3.2 Discussão

Os resultados sugerem que a ConvNeXt, com 28.6M de parâmetros, possui melhor capacidade de generalização, apresentando alto desempenho no conjunto de teste. Em casos de limitação de recursos computacionais, a EfficientNet, que conta com 21.6M de parâmetros, mostra-se como uma alternativa satisfatória, pois alcança um bom desempenho com um modelo mais enxuto. Já a ResNet, embora apresente uma queda ligeiramente maior entre validação e teste, ainda se mantém próxima das demais em termos de métricas de teste.

Tais observações evidenciam a relevância do *trade-off* entre desempenho e custo computacional na seleção da arquitetura adequada. Se o objetivo principal for obter a maior acurácia possível, mesmo que isso exija um hardware mais robusto, a ConvNeXt se sobressai. Por outro lado, em cenários com recursos limitados, a EfficientNet torna-se uma alternativa atrativa, pois sacrifica no máximo 4% do desempenho, mas reduz o número de parâmetros em quase 25% quando comparada com a ConvNeXt.

Capítulo 4

Conclusão

Neste trabalho, investigamos a aplicação de redes neurais convolucionais para a detecção de marcas d'água em imagens de anúncios de imóveis. A partir de um conjunto de dados coletado de diferentes imobiliárias, aplicamos uma metodologia que incluia processamento de imagens, aumento de dados e otimização de hiperparâmetros para treinar diferentes modelos, utilizando transferência de conhecimento. Conduzimos experimentos comparativos para avaliar o desempenho das arquiteturas ResNet, EfficientNet e ConvNeXt na tarefa de classificação binária de imagens para determinar presença ou ausência de marcas d'água.

Os resultados obtidos mostram que todas as arquiteturas testadas foram capazes de generalizar o conhecimento adquirido durante treinamento, alcançando mais de 80% de acurácia no conjunto de teste. A ConvNeXt destacou-se como o modelo de melhor desempenho, com 87.8% de acurácia e 87.1% de F1. No entanto, a EfficientNet se revelou como uma alternativa satisfatória em cenários com restrições de recursos, oferecendo um bom equilíbrio entre desempenho e custo computacional. Essa comparação reforça a importância da escolha da arquitetura mais adequada para a tarefa de interesse, considerando fatores como complexidade do modelo e recursos disponíveis para aplicações práticas.

Para trabalhos futuros, sugerimos a extensão do atual estudo para realizar também a localização da marca d'água dentro da imagem, isto é, obter a posição que ela ocupa. Além disso, é possível estudar o uso de redes neurais para a remoção de marcas visíveis ou detecção de marcas d'água não visíveis. Com foco em uma aplicação prática, também é viável realizar a implementação de um detector de marcas em tempo real, explorando aspectos de engenharia de *software* e otimização de desempenho em casos de limitações de recursos, como em aplicativos *mobile*.

Referências

[Goodfellow *et al.* 2016] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. MIT Press, 2016 (citado na pg. 5).

[He *et al.* 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren e Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. url: https://arxiv.org/abs/1512.03385 (citado na pg. 18).

[Krizhevsky *et al.* 2012] Alex Krizhevsky, Ilya Sutskever e Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. Ed. por F. Pereira, C.J. Burges, L. Bottou e K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. url: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (citado na pg. 18).

[Li *et al.* 2018] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh e Ameet Talwalkar. "Hyperband: a novel bandit-based approach to hyperparameter optimization". *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52. url: http://jmlr.org/papers/v18/16-558.html (citado na pg. 25).

[Liu *et al.* 2022] Zhuang Liu *et al. A ConvNet for the 2020s*. 2022. arXiv: 2201.03545 [cs.CV]. url: https://arxiv.org/abs/2201.03545 (citado na pg. 18).

[McCulloch e Pitts 1943] Warren S. McCulloch e Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. doi: 10.1007/BF02478259 (citado na pg. 6).

[Nematollahi *et al.* 2016] M.A. Nematollahi, C. Vorakulpipat e H.G. Rosales. *Digital Watermarking: Techniques and Trends*. Springer Topics in Signal Processing. Springer Nature Singapore, 2016. isbn: 9789811020957 (citado na pg. 3).

[O'Malley *et al.* 2019] Tom O'Malley *et al. KerasTuner*. https://github.com/keras-team/keras-tuner. 2019 (citado na pg. 25).

[Rosenblatt 1958] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". *Psychological Review* 65.6 (1958), pp. 386–408. doi: 10.1037/h0042519 (citado na pg. 6).

[Rumelhart *et al.* 1986] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. "Learning representations by back-propagating errors". *Nature* 323.6088 (1986), pp. 533–536. doi: 10.1038/323533a0 (citado na pg. 12).

[Tan e Le 2020] Mingxing Tan e Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG]. url: https://arxiv.org/abs/1905.11946 (citado na pg. 18).