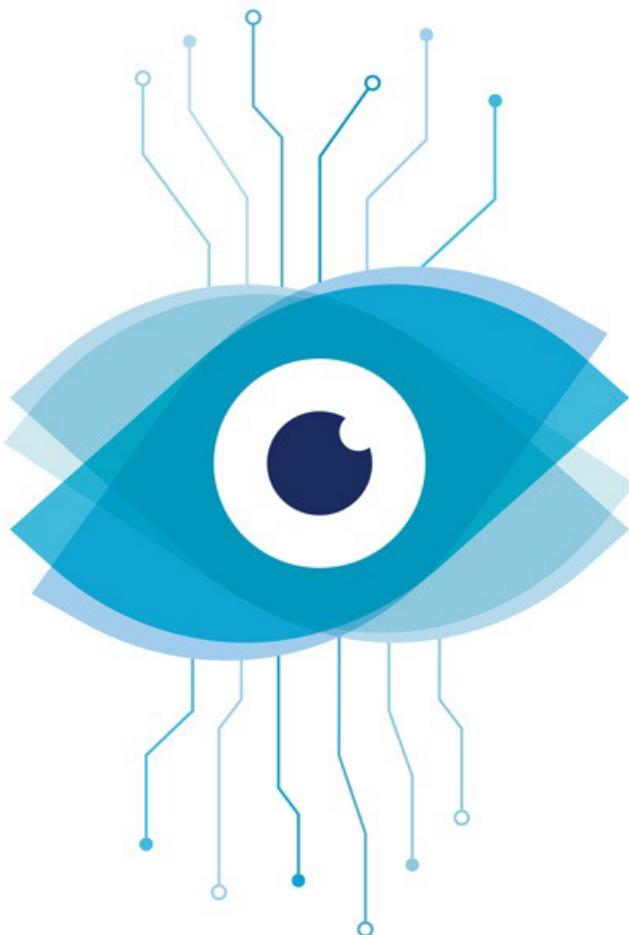


Introdução à Visão Computacional

Uma abordagem prática com Python e OpenCV



Casa do
Código

FELIPE BARELLI

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-57-1

EPUB: 978-85-94188-58-8

MOBI: 978-85-94188-59-5

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Agradeço à editora Casa do Código pela oportunidade de publicar esta obra; ao apoio dos familiares e amigos; aos professores da Universidade Vila Velha e do Instituto Federal do Espírito Santo por todo o conhecimento transmitido durante minha formação acadêmica.

Por fim, agradeço a você, leitor, uma vez que o seu interesse por este assunto foi minha maior motivação. Obrigado a todos, preparem um bom café e aproveitem a leitura.

SOBRE O AUTOR

Felipe da Costa Barelli é bacharel em Ciência da Computação pela Universidade Vila Velha (UVV) e especialista em Engenharia Elétrica com ênfase em Sistemas Inteligentes Aplicados à Automação pelo Instituto Federal do Espírito Santo (Ifes). O autor também é mestrando em Informática pela Universidade Federal do Espírito Santo (Ufes) e membro da Sociedade Brasileira de Computação.

- Site: <https://felipebarelli.github.io>

PREFÁCIO

Os sistemas baseados em Visão Computacional estão cada dia mais presentes em nosso cotidiano, seja nos veículos autônomos, nos robôs industriais ou em equipamentos hospitalares capazes de diagnosticar doenças automaticamente em exames por imagem. Justamente por essa tecnologia permitir que máquinas enxerguem o mundo à nossa volta, sendo capaz de automatizar e solucionar tantos problemas, é que o mercado necessita cada vez mais de profissionais capacitados para atuarem nesse segmento.

A fim de atender a essa demanda, este livro é o pontapé inicial para o profissional que deseja iniciar, de forma prática e pouco teórica, o estudo sobre o desenvolvimento de sistemas baseados em Visão Computacional.

Público-alvo

Este livro é voltado para estudantes de graduação e pós-graduação no campo da engenharia e tecnologia, como também profissionais e técnicos que buscam o primeiro contato prático com o desenvolvimento de sistemas de Visão Computacional e Processamento de Imagens.

Pré-requisitos

Para compreender melhor os conceitos abordados nesta obra, é fundamental ter um conhecimento básico sobre lógica de programação, habilidade em entender e desenvolver algoritmos em Python e compreensão do sistema de numeração binário. É

necessário também que você saiba efetuar operações com matrizes e conheça medidas estatísticas como moda, média e mediana.

Material online

Todas as imagens e os códigos apresentados nesta obra estão disponíveis no GitHub, no repositório: <https://github.com/felipecbarelli/livro-visao-computacional>. Caso você tenha adquirido apenas a versão impressa deste livro, cujas imagens estão em tons de cinza, é importante que você visualize as coloridas em seu computador.

Sumário

1 Introdução à Visão Computacional	1
2 Preparando o ambiente de estudo	10
2.1 Instalação no Windows e Mac OS	12
2.2 Instalação no Linux	14
2.3 Instalação das bibliotecas	15
2.4 Execução de programas	18
3 Aquisição de imagem	20
3.1 Câmeras digitais	20
3.2 Sensores CCD e CMOS	22
3.3 Formação da imagem	27
3.4 Resolução e quantização	31
3.5 Cor de imagens	37
3.6 Formatos de imagem	43
3.7 Prática com Python e OpenCV	47
4 Representação de cores no espaço	52
4.1 Cores no espaço RGB	53
4.2 Cores no espaço HSV	60

5 Pré-processamento	68
5.1 Operações básicas	68
5.2 Histograma de cores	73
5.3 Transformações geométricas	85
5.4 Operações aritméticas	95
5.5 Ruído em imagens	104
6 Aplicação de filtros	110
6.1 Filtro de média	113
6.2 Filtro gaussiano	115
6.3 Filtro de mediana	117
6.4 Filtro bilateral	120
7 Realce de bordas	124
7.1 Operador de Sobel	124
7.2 Operador laplaciano	126
7.3 Filtro máscara de desaguçamento	130
7.4 Detector de bordas de Canny	133
8 Operações morfológicas	136
8.1 Elemento estruturante	138
8.2 Erosão e dilatação	141
8.3 Abertura e fechamento	147
8.4 Gradiente morfológico	153
8.5 Top Hat	154
8.6 Tratamento de ruído	157
9 Segmentação de objetos	160
9.1 Segmentação por binarização	160

Casa do Código	Sumário
9.2 Segmentação por cor	172
9.3 Segmentação por bordas	176
9.4 Segmentação por movimento	178
10 Extração de características	182
10.1 De aspecto	182
10.2 Dimensionais	187
10.3 Inerciais	193
10.4 Topológicas	205
11 Reconhecimento de Padrões	209
12 Classificador K-NN	215
12.1 Algoritmo K-NN	215
12.2 K-NN com scikit-learn	218
13 Algoritmo Haar Cascade	228
13.1 Detecção de objetos	232
14 Aplicações da Visão Computacional	238
14.1 Reconhecimento de objetos	239
14.2 Reconhecimento de cores	242
14.3 Contagem de objetos	244
14.4 Distância entre objetos	246
14.5 Rastreamento de objetos	248
14.6 Reconhecimento de caracteres	252
15 Referências bibliográficas	255

Versão: 21.9.27

CAPÍTULO 1

INTRODUÇÃO À VISÃO COMPUTACIONAL

Programar computadores para enxergar o ambiente ao nosso redor e torná-los capazes de reconhecer e extrair informações de objetos podem parecer tarefas complexas que necessitam de fortes conhecimentos matemáticos e de programação. A verdade é que os sistemas de Visão Computacional estão cada vez mais presentes em nosso cotidiano, bem como mais fáceis de se trabalhar.

Hoje, existem diversas bibliotecas e linguagens de programação para desenvolvimento de softwares nessa área, todas elas abstraindo conceitos matemáticos e tornando seu uso mais fácil. Antes de detalhar essas novas tecnologias, é importante compreender o que é Visão Computacional e como é possível fazer um computador enxergar o que acontece próximo a ele.

No ano de 1982, Ballard e Brown, na obra *Computer Vision*, definiram Visão Computacional como a ciência que estuda e desenvolve tecnologias que permitem que máquinas enxerguem e extraiam características do meio, através de imagens capturadas por diferentes tipos de sensores e dispositivos. Essas informações extraídas permitem reconhecer, manipular e processar dados sobre os objetos que compõem a imagem capturada.

A figura a seguir ilustra alguns desses dispositivos frequentemente utilizados para aquisição de imagens. Observe que alguns deles, por exemplo, o aparelho de Raio-X e o de Ultrassom, são capazes de registrar imagens mesmo onde não há luz visível.



Figura 1.1: Dispositivos para aquisição de imagem

Apesar de trabalhos como o de Ballard e Brown terem sido publicados nos anos 80, as primeiras menções sobre Visão Computacional ocorreram na década de 50. Já os primeiros trabalhos foram iniciados por volta de 20 anos depois.

Como muitas tecnologias existentes hoje, desde o início essa também procurou imitar a natureza humana. Naquela época, acreditava-se que logo seria possível representar em uma máquina o sentido da visão humana de forma completa. Com o passar dos anos, verificou-se uma grande dificuldade em desenvolver um modelo para essa representação, principalmente por falta de

informações sobre como as imagens são interpretadas no cérebro humano.

Nos dias atuais, são realizadas diversas pesquisas a fim de entender o funcionamento desta parte do cérebro, justamente para aplicar as mesmas ideias no desenvolvimento tecnológico da Visão Computacional.

A Visão Computacional também pode ser vista como um complemento da visão biológica. Na biologia, a percepção visual de alguns seres vivos é estudada e representada em modelos que descrevem sua fisiologia. Já a Visão Computacional estuda e implementa sistemas capazes de enxergar por meio de processos artificiais, implementados por hardwares e softwares.

Além do campo da neurobiologia, que estuda os sistemas biológicos de visão, diversas outras ciências contribuem com os estudos sobre Visão Computacional. A Inteligência Artificial é uma das que frequentemente colaboram para esse desenvolvimento tecnológico, principalmente quando aplicada à área da robótica, que utiliza sistemas de Visão Computacional para fornecer ao robô informações sobre o ambiente.

O Processamento de Imagens e o Reconhecimento de Padrões são dois campos que também estão fortemente relacionados. Enquanto estudos sobre Processamento de Imagens apresentam técnicas para manipular informações representadas na imagem – por exemplo, realçar bordas e remover ruídos –, os estudos sobre Reconhecimento de Padrões são realizados a fim de identificar e classificar os objetos representados. A figura a seguir exemplifica outros campos de estudo que estão interligados com o de Visão Computacional.

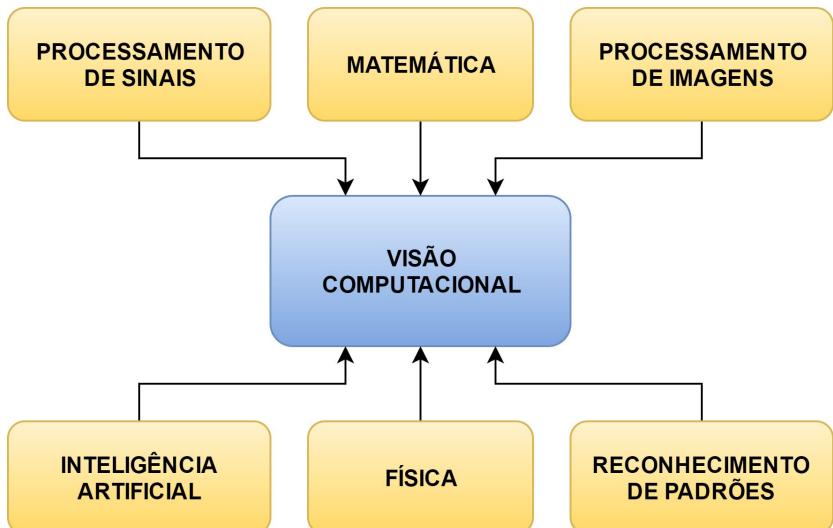


Figura 1.2: Campos de estudo interligados

Inúmeras tarefas que realizamos manualmente podem ser automatizadas por sistemas de Visão Computacional. Os sistemas de monitoramento por imagem, capazes de detectar pessoas em um ambiente, como os robôs industriais, preparados para montar automóveis, são exemplos comuns da aplicação da Visão Computacional em nosso cotidiano. Interessada no desenvolvimento de veículos autônomos, inteligentes para trafegar sem a intervenção humana, a indústria automotiva tem investido cada dia mais no desenvolvimento desse tipo de sistema.

A figura seguinte apresenta um desses veículos, equipado com um avançado sistema de Visão Computacional capaz de rastrear e detectar pessoas, placas, ruas e outros objetos.



Figura 1.3: Veículo autônomo

Assim como no campo industrial, os sistemas baseados em Visão Computacional também são bastante usados no campo da saúde. Essa tecnologia unificada a técnicas de aprendizado de máquina, que permitem ao computador aprender e aperfeiçoar seu desempenho em alguma tarefa, tem sido empregada para detectar anomalias em exames por imagem, como: tomografia computadorizada, ressonância magnética, ultrassom etc.

Desta forma, esses exames citados podem proporcionar precocemente o diagnóstico de doenças, contribuindo para evitar futuras complicações e garantir a qualidade de vida de muitos pacientes. A figura a seguir ilustra um deles. Observe como um tumor cerebral foi identificado por um sistema de Visão Computacional em uma imagem obtida por uma ressonância magnética.

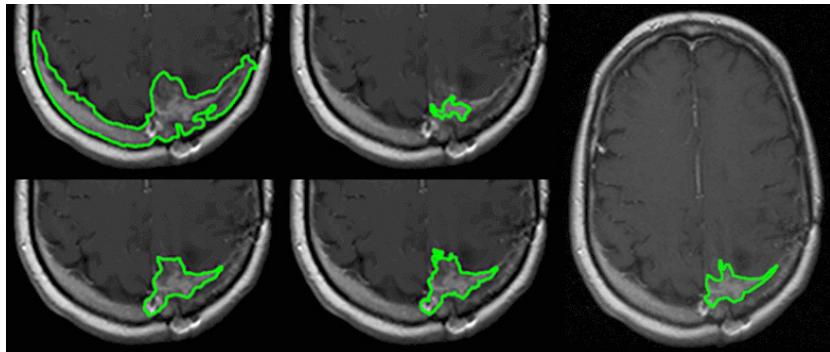


Figura 1.4: Tumor cerebral segmentado (REDDY; SOLMAZ; YAN et al., 2012)

Mesmo empregados em diversas áreas, junto a outras diferentes tecnologias, os sistemas de Visão Computacional geralmente apresentam um fluxo em comum. A figura a seguir apresenta esse fluxo em um diagrama de blocos e, nos parágrafos seguintes, cada uma dessas etapas do diagrama será detalhada.

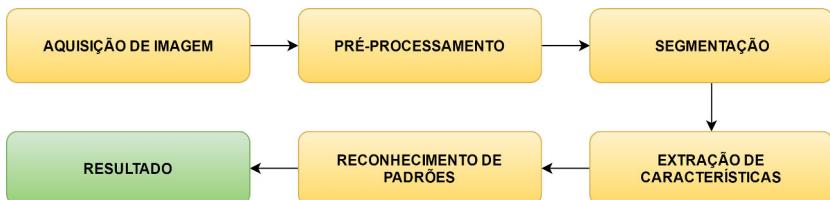


Figura 1.5: Fluxo de um sistema baseado em Visão Computacional

Usualmente a primeira etapa de um sistema de Visão Computacional é a aquisição de imagem. Nela, sensores, como os apresentados na Figura *Dispositivos para aquisição de imagem*, são utilizados para obter e digitalizar a imagem com qual será trabalhada. A imagem digitalizada pode variar entre uma imagem bidimensional, tridimensional ou ainda uma sequência de imagens, como ocorre em vídeos.

Após a primeira etapa, que nos fornece como resultado uma imagem digitalizada, a segunda etapa consiste no pré-processamento dessa imagem. Para que possamos compreendê-la melhor, é essencial que o conceito de região ou objeto de interesse seja assimilado.

REGIÕES OU OBJETOS DE INTERESSE

São as regiões ou elementos presentes na imagem que queremos identificar e obter informações. Imagine que estamos desenvolvendo um sistema para contar o número de fichas de poker em uma imagem, e essas fichas representam os nossos objetos, ou regiões de interesse.

Com os objetos de interesse definidos, algumas técnicas de pré-processamento para destacar bordas e formas geométricas e tratar ruídos são aplicadas na imagem para realçá-la, facilitando a obtenção de suas informações pelo sistema de Visão Computacional. A figura a seguir apresenta uma imagem antes e depois de ser submetida a procedimentos de pré-processamento. Considerando que as fichas de poker na fotografia esquerda são os nossos objetos de interesse e, após a etapa de pré-processamento, elas foram realçadas e o fundo, eliminado.

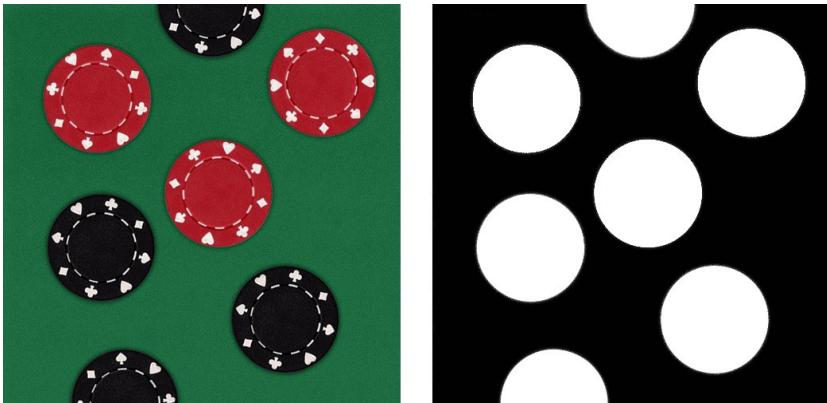


Figura 1.6: Etapa de pré-processamento

Na etapa seguinte, de segmentação, os objetos de interesse são segmentados da imagem original, em uma ou mais imagens, com o propósito de facilitar a extração de características desses objetos. Ainda tendo como base a fotografia à esquerda da figura anterior, suponha que estamos desenvolvendo um sistema para contar o número de fichas na cor vermelha.

Para isso, essa fotografia deverá ser submetida a outra técnica de pré-processamento, a fim de realçar somente essas fichas e possibilitar que estas sejam segmentadas em uma nova imagem. A figura a seguir ilustra esse procedimento de segmentação.

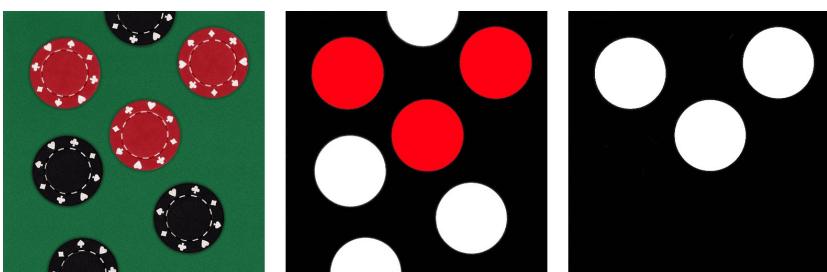


Figura 1.7: Processo de segmentação

Com as regiões de interesse segmentadas, a penúltima etapa consiste na extração de características do objeto. Tendo as fichas de poker mais uma vez como exemplo, após a etapa de segmentação, duas características que poderiam ser extraídas são a área e o diâmetro de cada uma delas. Essas informações podem ser úteis para diferenciá-las de outros objetos circulares na cor vermelha, que poderiam também estar sobre a mesa.

Por fim, na última etapa, ocorre o processamento de alto nível, com o objetivo de reconhecer o objeto segmentado através de suas características e defini-lo em uma determinada classe. Duas classes para esse exemplo poderiam ser "ficha de poker vermelha" e "outro objeto circular vermelho". Em outras palavras, esta é a etapa responsável por validar os resultados e definir se eles são satisfatórios ou não.

Agora que o conceito, as aplicações e o fluxo de um sistema de Visão Computacional foram apresentados, no decorrer da leitura deste livro, você aprenderá na prática como desenvolver sistemas como este. No capítulo seguinte, serão apresentadas as ferramentas utilizadas bem como a instalação de cada uma delas.

CAPÍTULO 2

PREPARANDO O AMBIENTE DE ESTUDO

Depois de uma breve introdução sobre os sistemas de Visão Computacional, chegou a hora de preparar o ambiente de estudo. Seguindo minuciosamente todos os passos apresentados neste capítulo, você poderá colocar em prática todo o conhecimento que vai adquirir durante a leitura dos demais. Para o ambiente estudo ficar completo, será necessária a instalação de alguns softwares, mais especificamente o Python e algumas bibliotecas.

Python é uma linguagem de programação de alto nível, orientada a objeto, interpretada e fácil de trabalhar. Foi lançada em 1991 por Guido van Rossum e, atualmente, possui um modelo de desenvolvimento aberto, gerenciado pela Python Software Foundation, uma organização sem fins lucrativos. É uma linguagem que pode ser usada gratuitamente tanto para fins acadêmicos quanto comerciais.

Para que o seu computador seja capaz de executar programas escritos em Python, é necessário que uma versão do seu interpretador esteja instalada. Existem distribuições de sistemas operacionais Linux e macOS que possuem esse interpretador nativamente instalado, diferente dos sistemas operacionais

Windows, que nem mesmo a versão mais recente (Windows 10) possui.

Para conferir se o sistema operacional que você utiliza contém alguma versão do Python instalada, digite o comando a seguir no terminal ou no prompt de comando:

```
$ python -V
```

A execução desse comando retornará a versão do Python instalada em seu computador; do contrário, uma mensagem de erro será exibida indicando que o comando não foi reconhecido. Caso o comando não seja reconhecido, você pode obter seu programa de instalação para Windows ou macOS no site oficial da linguagem Python (<http://python.org>).

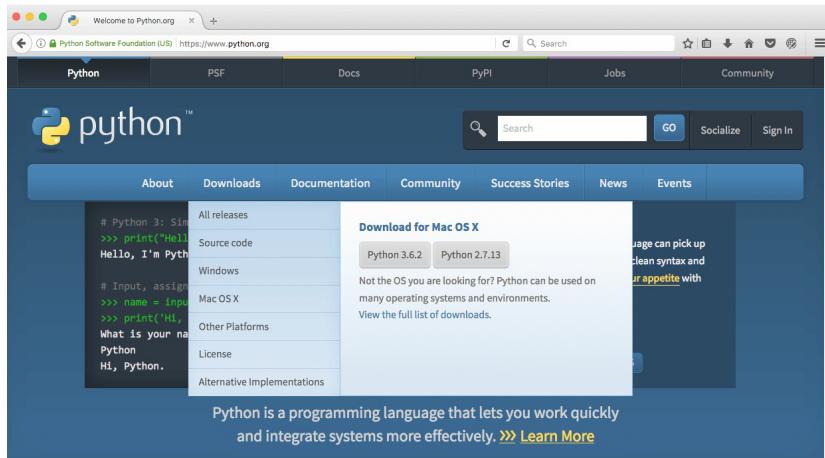


Figura 2.1: Site oficial da linguagem Python

Nesse site, duas versões diferentes do instalador estão disponíveis, uma para o Python versão 3.6 e outra para a 2.7. Para garantir a compatibilidade com as bibliotecas que serão utilizadas,

bem como os códigos exemplificados neste livro, recomendo o download da **versão 2.7**. Na seção a seguir, será detalhado como realizar a instalação do Python no Windows e no macOS.

2.1 INSTALAÇÃO NO WINDOWS E MAC OS

O procedimento é o mesmo para Windows ou macOS. Após o download do instalador, basta executá-lo e seguir os passos apresentados pelo programa de instalação. Durante esse procedimento, a única configuração que você poderá realizar é a definição do diretório do sistema onde o Python será instalado.

A fim de garantir o sucesso da instalação de algumas bibliotecas que serão utilizadas, recomendo instalar o Python no diretório padrão, ou seja, o definido pelo programa de instalação. A figura seguinte apresenta duas telas referentes ao programa de instalação do Python para Windows.

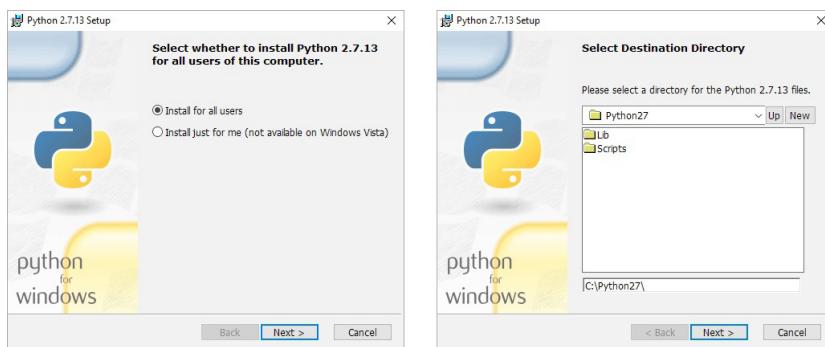


Figura 2.2: Python para Windows

Nos sistemas operacionais Windows, após concluir a instalação do Python, é necessário configurar as variáveis de ambiente. Essa

configuração permite a execução de programas em Python a partir do prompt de comando, sem a necessidade de acessar o diretório onde estão localizados os arquivos executáveis do seu interpretador.

Para configurar as variáveis de ambiente, acesse o Painel de Controle do Windows e, depois, Sistema e Segurança > Sistema. Com o painel aberto, localize na lateral esquerda o botão Definições avançadas do sistema.

Clique nele para abrir a janela de Propriedades do sistema. Nela, clique no botão Variáveis do ambiente... para abrir a janela na qual será feita a configuração dessas variáveis. Na figura a seguir, a imagem à direita apresenta essa janela de configuração, com título Variáveis de Ambiente.

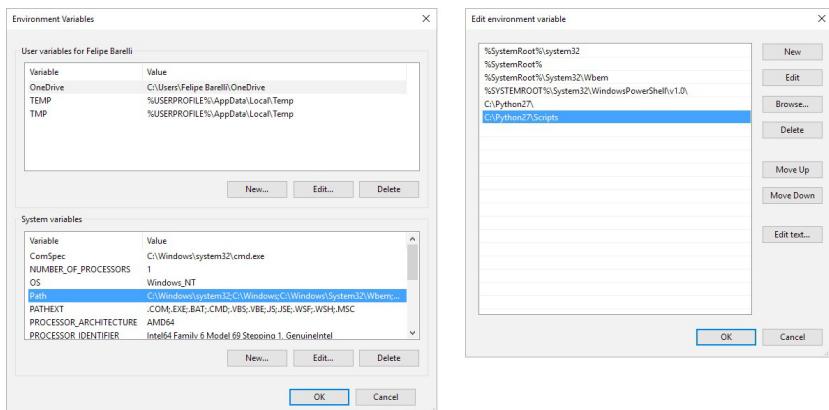


Figura 2.3: Variáveis de ambiente

Selecione o item Path na lista Variáveis do sistema. Em seguida, clique em Editar... para abrir a janela de edição, a mesma apresentada à esquerda na figura anterior. Na janela de

edição, clique no botão Novo e adicione os seguintes valores:

```
C:\Python27\  
C:\Python27\Scripts
```

Para finalizar, clique em Ok em ambas as janelas abertas. Depois de realizar esse procedimento, o Windows estará pronto para executar programas em Python a partir do prompt de comando. Na próxima seção, veja o procedimento de instalação do Python em sistemas operacionais Linux.

2.2 INSTALAÇÃO NO LINUX

As distribuições de Linux mais populares, como Ubuntu e Debian, possuem o Python nativamente instalado. Se por acaso você verificou que a sua não possui, não se preocupe; a instalação do Python pode ser facilmente realizada pelo gerenciador de pacotes apt-get . Para instalar o Python 2.7, digite no terminal:

```
$ sudo apt-get install python2.7
```

Após concluir a instalação do Python via apt-get , instale também o gerenciador de pacotes pip . Este facilita a instalação das principais bibliotecas disponíveis para a linguagem Python. Para os usuários de Windows e macOS, o pip é instalado automaticamente junto ao Python. No Linux, para instalá-lo, digite no terminal:

```
$ sudo apt-get install python-pip
```

Finalizando a instalação do Python, seguimos para a instalação das bibliotecas. Na próxima seção, será abordado o que são essas bibliotecas, os recursos que oferecem e como instalá-las.

2.3 INSTALAÇÃO DAS BIBLIOTECAS

Para potencializar a linguagem Python com ferramentas que podem facilitar o desenvolvimento de sistemas de Visão Computacional, algumas bibliotecas devem ser instaladas. As bibliotecas são pacotes com funções que não são nativas da linguagem. A tabela a seguir apresenta quais serão usadas para o desenvolvimento dos programas estudados neste livro.

Biblioteca	Descrição
OpenCV	Ferramentas para processamento de imagens.
Pillow	Ferramentas para processamento de imagens.
NumPy	Ferramentas para trabalhar com arranjos, vetores e matrizes.
Matplotlib	Ferramentas para gerar gráficos.
Statistics	Ferramentas para cálculos estatísticos.
Scikit-Learn	Ferramentas para aprendizado de máquina.
Python-tesseract	Ferramentas para reconhecimento de caracteres.

Cada uma delas será apresentada em um tópico a seguir. O procedimento para instalação de todas é o mesmo: basta abrir o terminal ou o prompt de comando (no Windows) e executar um comando utilizando o programa `pip`. Esses comandos serão também exemplificados nos tópicos seguintes.

OpenCV

Das sete bibliotecas apresentadas, a OpenCV (<http://opencv.org>) é a principal. Esta é uma biblioteca multiplataforma, projetada para o desenvolvimento de aplicativos na área de Visão Computacional, Processamento de Imagens,

Estrutura de Dados e Álgebra Linear. Foi originalmente desenvolvida pela Intel em 2000, sendo totalmente livre para o uso acadêmico e comercial.

A biblioteca OpenCV tem um conjunto de funções que torna simples a tarefa de manipular e processar imagens digitais. Para instalá-la, execute no terminal o comando:

```
$ pip install opencv-python
```

Pillow

Outra biblioteca para processamento de imagens usada é a Pillow (<https://python-pillow.org>). Ela, assim como a OpenCV, facilita carregar, manipular e salvar imagens com Python. Justamente por oferecer ferramentas que a OpenCV também oferece, essa biblioteca será utilizada apenas por ser compatível com a biblioteca Python-tesseract (usada para reconhecimento de caracteres em imagens). Para instalá-la, execute o comando no terminal:

```
$ pip install Pillow
```

NumPy

A biblioteca NumPy (<http://www.numpy.org>) facilita trabalhar com arranjos, vetores e matrizes. A NumPy apresenta uma sintaxe de fácil compreensão, semelhante ao software proprietário Matlab. Como vamos trabalhar com imagens digitais, que são representadas como matrizes de pixels, esse pacote é indispensável para aumentar nossa produtividade.

A NumPy pode ser facilmente instalada através do `pip`, via

terminal, executando o comando:

```
$ pip install numpy
```

Matplotlib

Para facilitar a criação de gráficos, utilizados para apresentar os resultados obtidos com alguns algoritmos que serão exemplificados, a biblioteca Matplotlib (<https://matplotlib.org>) será utilizada. Esta possui um conjunto de ferramentas que nos permite gerar diferentes tipos de gráficos, por exemplo, histogramas, diagramas de caixa e diagramas de ponto.

Para instalá-la, execute no terminal o comando seguinte:

```
$ pip install matplotlib
```

Statistics

A biblioteca Statistics (<https://docs.python.org>) possui um conjunto de ferramentas que facilitam cálculos estatísticos, como a obtenção da média, moda e mediana de uma lista de valores. Utilizando suas funções, não será necessário desenvolver rotinas para obter essas medidas.

Essa biblioteca é nativa da linguagem Python, logo, não há necessidade de instalar nenhum pacote para utilizá-la. Mais informações sobre a Statistics e exemplos de uso estão disponíveis em seu site.

Scikit-Learn

Durante o estudo sobre processamento de alto nível, que trata sobre reconhecimento de padrões e classificação de objetos, a

biblioteca Scikit-Learn (<http://scikit-learn.org>) será utilizada. Ela possui um conjunto de ferramentas que torna simples a implementação de algoritmos para aprendizado de máquina. Para instalá-la, execute no terminal o comando:

```
$ pip install scikit-learn
```

Concluindo a instalação de todas as bibliotecas, seu computador estará preparado para executar os programas em Python que serão desenvolvidos no decorrer deste estudo. Na próxima seção, será explicado como salvar e executar programas escritos nessa linguagem.

2.4 EXECUÇÃO DE PROGRAMAS

Os programas desenvolvidos em Python são salvos, por padrão, em arquivos .py . Para escrevê-los, você pode utilizar qualquer editor de código, até mesmo o bloco de notas do Windows. A lista a seguir apresenta algumas opções de editores de código, todos gratuitos e disponíveis para Windows, macOS ou Linux.

- Sublime Text – <https://sublimetext.com>
- Atom – <https://atom.io>
- Visual Studio Code – <https://code.visualstudio.com>

Para executar um programa salvo em um arquivo .py , abra o terminal (ou prompt de comando) e digite o comando python seguido do nome do arquivo. Logo após, pressione Enter para executá-lo. O código seguinte exemplifica esse procedimento.

```
$ python meu-programa.py
```

Com todos esses softwares instalados, agora você está pronto para começar os estudos sobre Visão Computacional e Processamento de Imagens. No próximo capítulo, será detalhado como é realizada a aquisição de imagens por esses sistemas.

CAPÍTULO 3

AQUISIÇÃO DE IMAGEM

No primeiro capítulo, foi apresentado o fluxo comum que os sistemas de Visão Computacional geralmente seguem, no qual o procedimento de aquisição de imagem foi definido como a primeira etapa. Nela, são utilizados sensores para obter e digitalizar as imagens que serão processadas por esses sistemas.

Durante o estudo deste livro, uma câmera digital será usada para registrar as imagens que serão trabalhadas, mais especificamente a conhecida como webcam – popularmente utilizada em chamadas de vídeo no Skype.

Antes de manusear uma delas, é importante compreender como funcionam, ou seja, como as imagens são capturadas, digitalizadas e armazenadas em arquivos. Todo esse conteúdo teórico será abordado neste capítulo, e o tópico a seguir trata sobre o funcionamento das câmeras digitais.

3.1 CÂMERAS DIGITAIS

Relembrando os tempos de escola, você provavelmente estudou na disciplina de Ciências o funcionamento básico de uma câmera fotográfica. Lembra de que elas possuem uma câmara escura e um orifício com uma lente de vidro que permite a entrada

de luz? Então, as câmeras digitais também seguem este mesmo princípio.

Para refrescar a memória e facilitar o entendimento, observe a ilustração da figura a seguir.

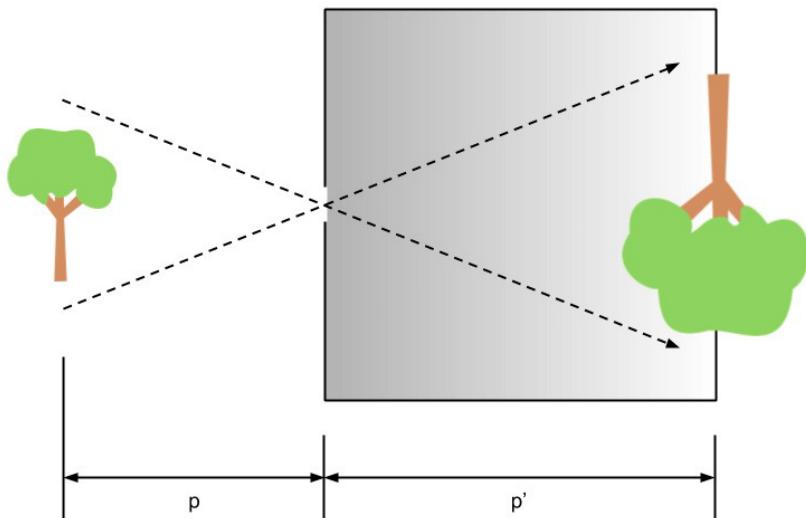


Figura 3.1: Esquema de uma câmara escura

Este pequeno orifício presente na câmara, preenchido com uma lente convergente (ou seja, uma lente que converge os feixes de luz para um único ponto), permite que a luz proveniente do objeto ou cena que se deseja capturar atravesse e incida em seu interior, projetando uma imagem invertida na face interna, contrária ao orifício. Para arquivar a imagem projetada, as câmeras analógicas (muito utilizadas antes da popularização das câmeras digitais) empregavam em seu interior filmes químicos sensíveis à luz.

Nas câmeras digitais, utilizadas hoje em dia, sensores eletrônicos sensíveis à luz desempenham essa tarefa. Eles atuam convertendo a imagem obtida de forma analógica em dados digitais. Os dois tipos de sensores eletrônicos mais usados para esse procedimento são os CCD e os CMOS, detalhados na seção a seguir.

3.2 SENSORES CCD E CMOS

Estes sensores são os principais itens de uma câmera digital. São eles os responsáveis por registrar as imagens e codificá-las em dados digitais, bits e bytes que serão processados e armazenados em forma de um arquivo digital. A figura a seguir apresenta esses dois sensores: na imagem à esquerda, um sensor CCD, e à direita, um sensor CMOS.

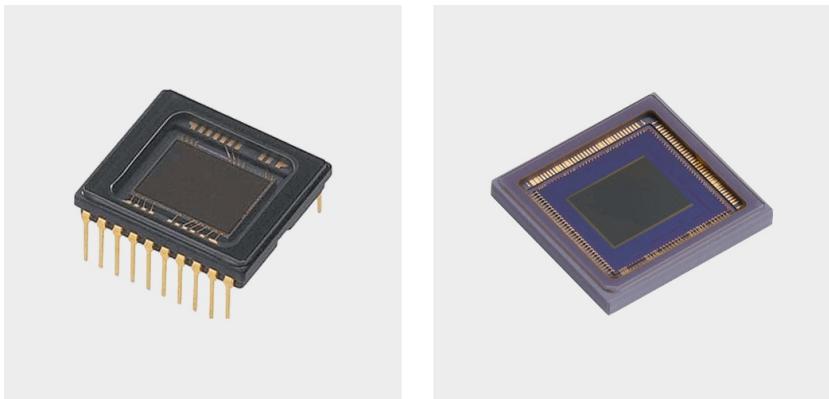


Figura 3.2: Sensores CCD e CMOS

Nesta seção, vamos detalhar o funcionamento desses sensores que convertem luz em elétrons. Primeiro, será apresentado o *Charge Coupled Device* (CCD), que, em português, pode ser

traduzido para *sensor de dispositivo de carga acoplada*.

Utilizado predominantemente nas câmeras digitais (como webcams, câmeras de smartphones e tablets), os sensores CCD usam uma matriz de capacitores fotossensíveis para capturar a luz da cena, transformá-la em imagem digital e transferi-la para o sistema de memória da câmera. Para facilitar o entendimento, imagine esse sensor como milhares de minúsculas placas solares, dessas que captam a luz para gerar energia elétrica.

Cada uma dessas placas receberia uma intensidade luminosa diferente da cena, representando um pixel da imagem gerada. O pixel é o menor ponto que forma uma imagem digital, e o conjunto desses pontos, organizados espacialmente em linhas e colunas, forma a imagem completa. Dessa forma, quanto maior a quantidade de placas, maior a quantidade de pixels, permitindo que mais detalhes sejam representados a fim de gerar imagens mais nítidas.

Uma vez obtida a intensidade luminosa incidente em cada uma dessas células (capacitores fotossensíveis), um conversor analógico para digital (A/D) é usado para transformar o valor da tensão elétrica, obtida de forma analógica, em um valor binário.

O conversor analógico-digital (A/D ou ADC) é um dispositivo eletrônico capaz de converter um sinal analógico em um sinal digital, representado por um nível de tensão ou intensidade de corrente elétrica.

Considerando apenas a intensidade luminosa, incidente sobre as células, somente imagens em tons de cinza poderiam ser registradas. Dessa maneira, o nível mais alto de luminosidade seria registrado como um pixel branco; o mais baixo, como um pixel preto; e os intermediários, tons de cinza.

Para representar imagens em cores, outros sistemas mais elaborados foram propostos. Uma das alternativas seria utilizar três sensores CCD, cada um dedicado à captura de uma determinada cor primária. O uso de três sensores justifica-se pelo fato de a visão humana ser tricromática, isto é, sensível a três cores primárias (o vermelho, o verde e o azul).

Mesmo parecendo uma solução satisfatória para representar os pixels em cores, uma outra mais prática, barata e bem elaborada foi implementada nesses sensores. Utilizando filtros, matemática e algoritmos, os engenheiros conseguiram obter o mesmo resultado com um único sensor.

Conforme apresentado na figura a seguir, filtros coloridos foram sobrepostos às células do sensor. Dessa forma, um mosaico de três cores (vermelho, verde, azul) forma a imagem que será processada posteriormente. Esse processamento é realizado por um algoritmo capaz de estimar a cor ideal para cada pixel, considerando o nível de cor dos pixels adjacentes.

Se um determinado pixel aparece em vermelho, o algoritmo é capaz de detectar a cor ideal para ele através de cálculos que consideram a intensidade de verde e azul nas células vizinhas. A figura a seguir ilustra os filtros sobrepostos às células dos sensores.

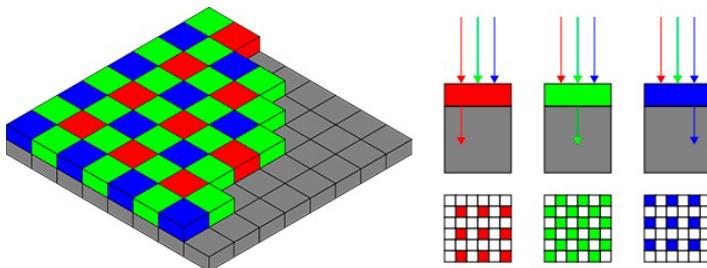


Figura 3.3: Células e filtros de um sensor CCD

Na figura anterior, observe que a quantidade de filtros verdes é superior à quantidade de vermelhos e azuis. O quadro a seguir justifica essa configuração.

CURIOSIDADE

Um pesquisador da Kodak chamado Bryce Bayer percebeu que o olho humano é duas vezes mais sensível à cor verde do que às cores vermelha e azul. Por este motivo, há o dobro de células com filtros verdes (50%) comparado à quantidade de células com filtros vermelhos (25%) e azuis (25%).

Além dos sensores CCD, existem também os *sensores semicondutores de metal-óxido complementar (Complementary Metal-Oxide Semiconductor)*, conhecidos como CMOS. Assim como os CCD, os sensores CMOS seguem o mesmo princípio de funcionamento: atuam convertendo luz em eletricidade e representando cores através de filtros coloridos sobrepostos às células. A principal diferença é que os CMOS, ao capturar a luz incidente sobre as células, não necessitam de um conversor A/D

para converter o sinal analógico em um sinal digital.

Os sensores CCD enviam o sinal analógico de cada célula, linha a linha, para ser convertido sequencialmente por um conversor A/D. Já os sensores CMOS, por meio de transistores presentes nas células, capturam e convertem o sinal analógico para digital em cada uma delas, sem a necessidade de convertê-los posteriormente.

Em resumo, diferente das células de um sensor CCD – que capturam sinais analógicos e enviam-nos para serem processados por um conversor A/D –, cada célula de um sensor CMOS desempenha diretamente essa tarefa.

Não pense que os CMOS, por converterem o sinal diretamente em cada célula e consumirem menos energia, são superiores ou mais caros aos CCD. Pelo contrário, estes geralmente apresentam menor custo de fabricação, são menos sensíveis à iluminação e podem gerar imagens com mais ruídos.

A fim de corrigir os ruídos e torná-los mais sensíveis à luz, os fabricantes investiram bastante em pesquisas para melhorar as tecnologias empregadas nos sensores CMOS. Hoje em dia, eles estão presentes na maioria das webcams disponíveis no mercado, sendo capazes de formar imagens nítidas e em alta resolução. Pesquisar qual tipo de sensor está presente na sua câmera é uma ótima ideia para começar o seu primeiro projeto de Visão Computacional e Processamento de Imagens.

Os sensores apresentados até então necessitam de luz para formar uma imagem digital, no entanto, existem os que são capazes de formar imagens digitais por outros meios. Dois muito

utilizados são os sensores de infravermelho (sensíveis ao calor) e os de ultrassom (sensíveis a vibrações ultrassônicas). Na seção seguinte, será apresentado como esses sensores são capazes de produzir imagens.

3.3 FORMAÇÃO DA IMAGEM

Ao analisarmos o funcionamento dos sensores CCD e CMOS, estudados anteriormente, percebemos que uma imagem digital pode ser definida como um conjunto numérico, bidimensional, que representa uma cena ou objeto. Esses sensores atuam convertendo a luminosidade de uma cena real em valores numéricos, que, quando representados em cor e organizados espacialmente em linhas e colunas, nos permitem perceber uma paisagem, objeto, pessoa etc.

Apesar de esses sensores necessitarem de luz para formar e registrar imagens, existem sensores capazes de realizar essa tarefa mesmo quando não há luz visível. Nos tópicos a seguir, dois deles serão detalhados, o sensor de ultrassom e o de infravermelho.

Sensor de ultrassom

Os sensores de ultrassom são dispositivos capazes de medir, a partir de ondas de som de alta frequência, a distância de objetos em relação à sua posição. Além dessa finalidade, existem sensores de ultrassom capazes de gerar imagens a partir dessas distâncias medidas. Um exemplo são os presentes em máquinas de ultrassonografia, um método de diagnóstico por imagem recorrente na medicina.

A ultrassonografia utiliza o eco de ondas ultrassônicas de alta frequência, para formar imagens em tempo real das estruturas internas de um organismo. Esse procedimento transforma um conjunto de valores de reflexões acústicas, distribuídas no espaço, em um conjunto de sinais de intensidade correspondente.

Esses sinais analógicos, gerados pelas reflexões, são discretizados e convertidos em sinais digitais, que representam a intensidade dos pixels da imagem digital gerada. A figura a seguir ilustra essa tecnologia: a imagem à esquerda apresenta uma máquina de ultrassonografia, e à direita, uma imagem de ultrassom obtida por uma dessas máquinas.

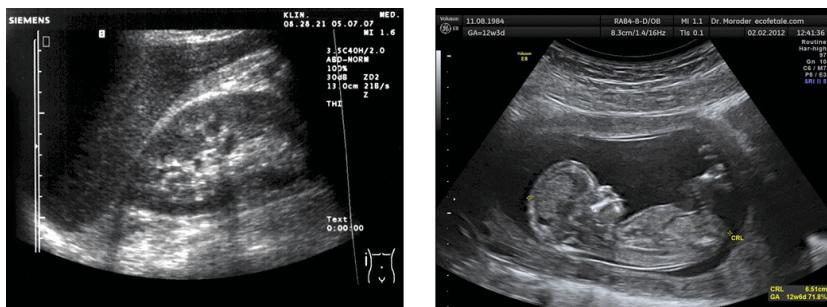


Figura 3.4: Imagens de ultrassonografia

Não só a medicina recorre a imagens geradas a partir de ondas sonoras. Essa tecnologia também é comum em sonares, instrumento utilizado em embarcações e estudos atmosféricos. Diferente dos equipamentos de ultrassonografia, para gerar imagens a partir das reflexões acústicas, os sonares atuam também com ondas de baixa frequência (infrassônicas).

São consideradas ondas sonoras de baixa frequência as que possuem valor inferior a 20 Hz; já as que possuem valor superior a

20.000 Hz são consideradas de alta frequência (ultrassônicas). Tanto os infrassons quanto os ultrassons são inaudíveis para o ouvido humano. A figura a seguir ilustra duas imagens geradas por sonar.

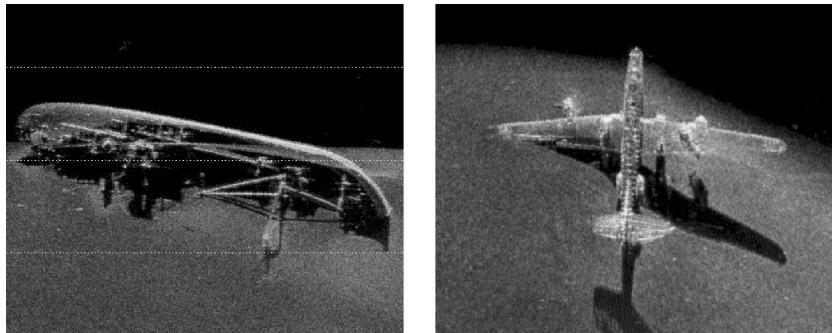


Figura 3.5: Imagens registradas por sonar. Fonte: <http://blueprintsubsea.com>

A aquisição de imagens formadas por ondas ultrassônicas pode ser aplicada, por exemplo, a sistemas de rastreamento, baseados em Visão Computacional e capazes de localizar e identificar objetos em locais onde não há iluminação. Essa tecnologia é usada para explorar fundo de oceanos, poços e túneis.

Assim como os sensores de ultrassom, os sensíveis à radiação infravermelha também não necessitam da luz visível para formar imagens. Mais detalhes sobre os sensores de infravermelho serão apresentados no tópico a seguir.

Sensor de infravermelho

Os sensores de infravermelho são dispositivos sensíveis ao calor emitido por qualquer objeto. Todo corpo sólido a uma temperatura acima do zero absoluto emite radiação infravermelha.

Essa radiação, invisível ao olho humano, pode ser captada por esses dispositivos e convertida em imagem.

Essa tecnologia é aplicada, por exemplo, nas câmeras de visão noturna, capazes de registrar imagens em ambientes escuros ou com baixa luminosidade. A figura a seguir ilustra duas imagens formadas por uma dessas câmeras de visão noturna.



Figura 3.6: Imagens em infravermelho

Conforme ilustrado na figura, essa tecnologia pode ser aplicada, por exemplo, a sistemas de segurança, baseados em Visão Computacional, capazes de detectar pessoas ou veículos nas proximidades de uma residência durante a noite.

Os sensores de infravermelho, assim como os de ultrassom, não necessitam de luz visível para formar imagens. Consequentemente, as imagens geradas por esses dispositivos são representadas em tons de cinza e, para cada pixel, é atribuído um valor entre 0 e 255, referente à intensidade da energia captada.

Conforme apresentado neste capítulo, imagens digitais podem ser obtidas por meio de dispositivos sensíveis a diferentes radiações. O que todos esses sensores apresentam em comum – seja CCD, CMOS, infravermelho ou ultrassônico – é que sinais

analógicos são captados e discretizados, ou seja, são convertidos em dados digitais que representam uma imagem. Esses dados digitais são armazenados em forma de matriz de pixels, onde cada pixel é a representação da intensidade de cor (em um ou mais canais) de um desses valores armazenados na matriz.

A quantidade de informação armazenada de cada ponto registrado de uma cena (ou seja, a quantidade de pixels que a fotografia possui) é o que determina a resolução da imagem digital gerada. Este assunto será melhor detalhado na seção a seguir.

3.4 RESOLUÇÃO E QUANTIZAÇÃO

A resolução de uma imagem (seja em filme ou digital) diz respeito à quantidade de informações representadas, isto é, ao nível de detalhes que ela comporta. Nas imagens digitais, a resolução é definida pelo processo de amostragem e quantização do sinal analógico. Ou seja, ocorre quando os infinitos pontos que representam a imagem analógica são reduzidos a um número finito de pontos (de pixels).

Com o intuito de facilitar o entendimento, a figura a seguir exemplifica o processo de amostragem. A primeira imagem da esquerda para a direita apresenta a cena registrada; a segunda ilustra a matriz de células do sensor que digitalizou o sinal; a terceira exibe o sinal amostrado em um número finito de pontos.

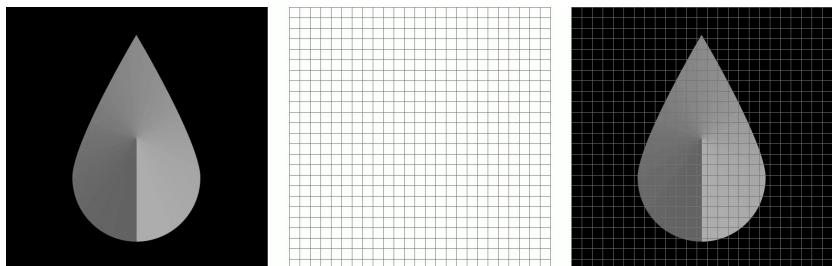


Figura 3.7: Processo de amostragem

Na figura anterior, observe que cada ponto da imagem mostrada (terceira imagem) representa uma subimagem da cena real. Quanto maior a amostragem do sinal – ou seja, quanto mais células são utilizadas para representar a imagem –, mais detalhes são guardados, logo, mais espaço é necessário para armazená-la.

Para cada um desses pontos, uma única cor deve ser definida a fim de representar o pixel correspondente. Esse processo é conhecido como quantização. Nos sensores, a quantização ocorre em cada célula sensível à luz, e a cor dos pixels é determinada pela intensidade de luz incidente em cada uma delas. Por outro lado, quando uma imagem digital é reamostrada – quer dizer, sofre alterações na largura ou altura –, a definição das cores de cada pixel é definida por alguma técnica de quantização.

Existem diversas técnicas de quantização, todas com a finalidade de determinar uma única cor para representar o conjunto de cores da subimagem. Uma delas é o cálculo da média dos valores que representam os tons de cinza do conjunto. Outra consiste em definir que cada pixel receberá a cor do primeiro pixel do conjunto.

A primeira dessas duas técnicas foi aplicada no exemplo

apresentado pela figura a seguir:

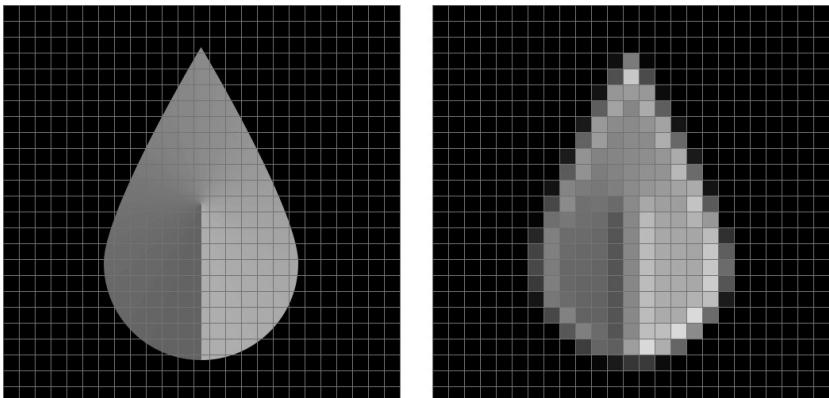


Figura 3.8: Processo de quantização

Essa figura ilustra o processo de quantização. A imagem à esquerda apresenta novamente o sinal amostrado em um número finito de pontos, e à direita exibe a imagem após o processo de quantização do sinal. Observe que a amostragem e a quantização do sinal definem quantos pixels e cores são necessários para representá-la em um arquivo digital, determinando suas resoluções.

Existem duas grandezas principais para especificar a resolução das imagens digitais: a resolução espacial e a resolução de bit. A resolução espacial está relacionada com a quantidade de pixels usados para representar a cena fotografada. A resolução de bit corresponde à quantização de cor, isto é, a quantidade de tons de cinza que podem ser atribuídos para cada ponto da imagem. Nesta seção, essas duas grandezas serão detalhadas nos tópicos a seguir.

Resolução espacial

A resolução espacial de uma imagem (também conhecida como resolução de pixel) está relacionada com a amostragem do sinal digital processado. Quanto maior a amostragem do sinal (quanto mais pontos são usados para representar a imagem), maior será sua resolução espacial.

A figura seguinte apresenta uma mesma fotografia representada em diferentes resoluções espaciais. Observe que, na fotografia à esquerda, que possui resolução espacial superior comparada à da direita, podem ser notados mais detalhes da cena.



Figura 3.9: Fotografia em diferentes resoluções espaciais

Existem inúmeras formas de medir a resolução espacial de uma imagem. Geralmente, são utilizadas unidades relacionadas a tamanhos físicos, tendo como base a proximidade entre uma linha e outra, quantificando o quanto próximas elas podem estar sem que sejam percebidas. Algumas dessas medidas relacionam linhas por milímetros, outras linhas por polegada, pontos por polegada etc.

Quando tratamos de imagens digitais, frequentemente nos referimos à sua resolução contabilizando a quantidade de pixels

em linhas e colunas. Por exemplo, quando dizemos que a imagem tem a resolução de 640px por 480px ou de 800px por 600px, estamos justamente nos referindo à sua resolução espacial. O resultado do produto do número de linhas pelo número de colunas fornece o total de pixels necessários para cobrir o espaço visual que ela ocupa.

É muito comum encontrarmos o termo *megapixel* determinando a resolução de uma imagem. Trata-se de uma convenção popular para definir a resolução espacial de uma imagem. A quantidade de megapixels é determinada pela divisão do total de pixels por um milhão.

Por exemplo, se a resolução da imagem for de 1280px por 1024px, ao multiplicarmos esses valores, obtemos que a imagem possui 1.310.720 pixels. Se dividirmos por um milhão, chegamos ao resultado 1.3, isto é, 1.3 megapixels de resolução, indicando que a imagem possui em torno de 1.300.000 pixels.

Além da resolução espacial, podemos definir uma imagem segundo a sua resolução de bit, que nos fornece informações sobre a quantidade de cores representadas no arquivo. No tópico a seguir, mais detalhes sobre a resolução de bit serão apresentados.

Resolução de bit

A resolução de bit, também conhecida como resolução de intensidade, determina a quantidade de valores de intensidade ou cor que um pixel pode representar. Por exemplo, em uma imagem binária, que possui apenas duas cores (preto ou branco), cada pixel pode ser representado com 1 bit, sendo 0 para preto e 1 para branco (uma resolução de 2 bits).

Em uma imagem em tons de cinza, que geralmente contém até 256 tons diferentes (variando do preto ao branco), são necessários 8 bits para representá-los. Já uma imagem colorida, em que as cores são representadas em 3 canais distintos, são necessários 24 bits, pois cada canal representa uma intensidade em 8 bits.

É válido ressaltar que nem todas as imagens coloridas são representadas com 24 bits e 3 canais. A resolução de bits de uma imagem depende do espaço de cores usado para representá-la, tema que será abordado no próximo capítulo. A figura a seguir exemplifica uma mesma fotografia, em tons de cinza, representada em diferentes resoluções de bit.



Figura 3.10: Diferentes resoluções de uma imagem

Na figura anterior, observe a diferença entre as duas fotografias. A imagem à esquerda possui pixels representados em até 256 tons de cinza, diferente da imagem à direita, com pixels representados em até 4 tons de cinza.

Em resumo, a resolução de bit representa a quantidade de bits necessários para armazenar a informação de cada pixel. Além da

resolução de espacial e da resolução de bit, existe uma outra que vale ser citada, a resolução temporal.

Resolução temporal

Além da resolução espacial e da resolução de bit, uma outra frequentemente utilizada é a resolução temporal. Esta está empregada ao sistema de captura contínua de imagens, como os vídeos, representando o número de imagens capturadas em um dado intervalo de tempo. Geralmente, usa-se a unidade quadros por segundo (qps) para definir a resolução temporal de um vídeo, em que cada quadro é representado por uma única imagem.

Algumas diferenças entre as representações de imagens em preto e branco, tons de cinza e coloridas já foram citadas neste capítulo. Para ficar ainda mais claro, na próxima seção, será apresentado com mais detalhes cada um desses tipos de imagem.

3.5 COR DE IMAGENS

Apesar de o assunto representação de cores no espaço ser abordado apenas no próximo capítulo, é importante que alguns conceitos básicos sobre os principais tipos de imagem sejam apresentados desde já. No tópico a seguir, as características das imagens binárias serão detalhadas e, em seguida, veremos mais sobre imagens em tons de cinza, RGB e ponto flutuante.

Imagens binárias

Esta é a forma mais simples de representar uma imagem. Existem apenas dois valores possíveis para representar cada pixel

de uma imagem binária: o valor 0 para a cor preta, e o valor 1 para a cor branca.

É válido lembrar que cada pixel de uma imagem representa a intensidade de luz capturada pelo sensor, correspondente a um ponto do objeto ou cena fotografada. Consequentemente, para que imagens binárias possam ser registradas, um limiar deve ser definido, com finalidade de determinar o valor analógico mínimo ou máximo registrado. Esse limiar define se o pixel será representado como preto (levemente iluminado) ou branco (fortemente iluminado).

A figura a seguir ilustra como cada pixel dessas imagens são representados. Observe na imagem à esquerda os valores que representam os pixels ampliados da imagem à direita.

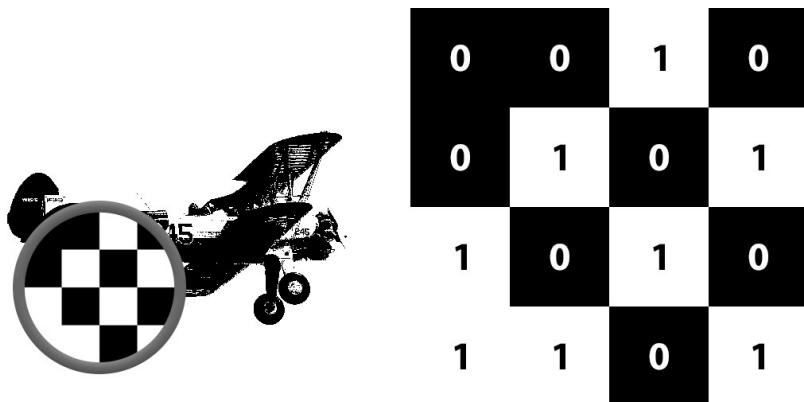


Figura 3.11: Exemplos de imagens binárias

Nos sistemas baseados em Visão Computacional, este tipo de imagem é muito usado, tanto na etapa de segmentação de objetos quanto na de extração de características. Mais detalhes sobre essas etapas serão abordados em capítulos a seguir, bem como a técnica

de limiarização (também conhecida como binarização), que nos permite converter qualquer outra imagem em uma binária.

Neste momento, é importante apenas que essas imagens sejam apresentadas. Para isto, a figura seguinte expõe outros exemplos de imagens binárias.

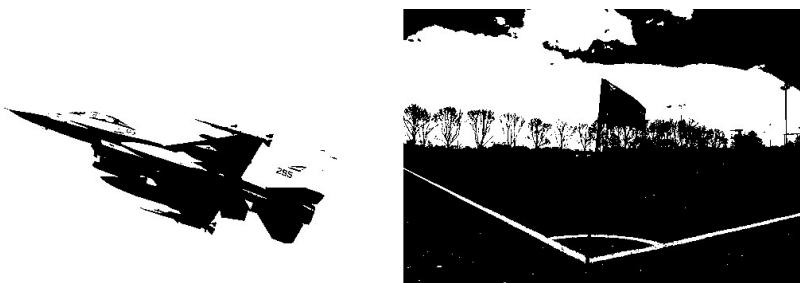


Figura 3.12: Exemplos de imagens binárias

Essas imagens podem ser facilmente representadas por uma sequência de bits, entretanto, na prática, elas geralmente são armazenadas consumindo 8 bits por pixel. Isso ocorre porque utilizamos formatos de arquivos padronizados, que serão listados e talhados ainda neste capítulo.

Imagens em tons de cinza

As imagens em tons de cinza vão além da capacidade das imagens binárias. Por meio delas, é possível armazenar valores intermediários, referentes à intensidade de luz, que necessariamente não precisam ser representados pela cor preta ou branca. Para as imagens em tons de cinza, podem ser representados até 256 tons diferentes, variando de 0 (preto) até 255 (branco). Para a apresentação numérica desses valores inteiros em

binário, são necessários 8 bits, sendo assim, cada pixel de uma imagem em tons de cinza é geralmente representado por 8 bits.

A figura a seguir ilustra como cada pixel dessas imagens é representado. Observe na imagem à esquerda os valores que representam os pixels ampliados da imagem à direita.

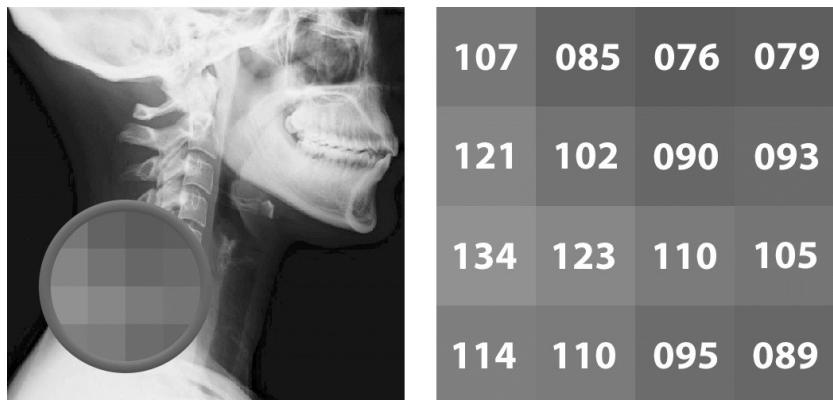


Figura 3.13: Exemplos de imagens binárias

Em sistemas de Visão Computacional, imagens em tons de cinza são bastante usadas. Muitas vezes é necessário apenas reconhecer ou contar objetos em imagens, em que a cor do objeto não é uma informação relevante. Nessas situações, imagens em tons de cinza são consideradas uma boa alternativa, pois torna o processamento mais rápido, uma vez que há menos informações para serem processadas. A figura a seguir apresenta algumas imagens em tons de cinza.



Figura 3.14: Exemplos de imagens em tons de cinza

Imagens em RGB

Neste capítulo, na seção sobre as diferenças entre os sensores CCD e CMOS, também foi abordado como as imagens coloridas são formadas e armazenadas. As imagens em RGB são imagens coloridas, também conhecidas como imagens de cores reais.

Relembrando rapidamente, filtros de três cores primárias (vermelho, verde e azul) são sobrepostos às células do sensor. Deste modo, três valores diferentes são registrados, sendo cada valor relacionado à intensidade luminosa de cada uma dessas cores.

Consequentemente, para cada pixel de uma imagem em RGB, temos a intensidade de luz vermelha, verde e azul representada no mesmo pixel. São necessários 8 bits para registrar a intensidade luminosa de cada uma das cores primárias, logo, uma imagem em RGB geralmente necessita de 24 bits para representar cada pixel.

Conceitualmente, uma imagem em RGB pode ser vista como três matrizes bidimensionais, sendo que cada matriz armazena cada canal de cor – ou seja, uma matriz para representar a imagem segundo a intensidade de cor vermelha, outra para a intensidade

de verde e outra para a de azul.

Uma outra forma de pensar é considerar a imagem como uma matriz tridimensional, $L \times C \times 3$, em que L é o número de linhas e C de colunas, permitindo que 3 valores sejam armazenados para cada pixel.

Em sistemas de Visão Computacional, imagens em RGB não são muito utilizadas por dois motivos principais. O primeiro é que, em muitos casos, uma imagem em tons de cinza, mais leve e rápida de ser processada, é suficiente para obter o resultado desejado. Outro motivo é a existência de outros métodos para representação de cor, ou seja, outros espaços de cores mais eficientes para manipular essas imagens (por exemplo, o espaço HSV, detalhado no próximo capítulo).

Para exemplificar as imagens em RGB, a figura apresenta algumas neste formato:

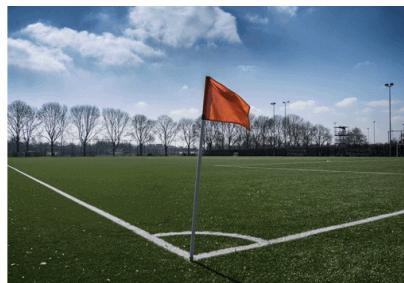


Figura 3.15: Exemplos de imagens em tons de cinza

Imagens em ponto flutuante

Existem também as imagens em ponto flutuante que, diferente dos outros tipos citados, não utiliza valores inteiros para

representação dos pixels, e sim um número em ponto flutuante. Esse modelo de representação também suporta imagens coloridas, que são comumente armazenadas no formato TIFF.

3.6 FORMATOS DE IMAGEM

Assim como existem diferentes tipos de imagem, existem também diferentes formatos de arquivos para armazená-las. A escolha do formato pode ser determinada tanto pelo tipo de imagem quanto pelo seu conteúdo. Cada formato de imagem possui um padrão próprio, um cabeçalho de arquivo que contém informações sobre como os valores numéricos (que representam cada pixel) estão armazenados e a sua posição no espaço.

Os mais de 30 anos de estudos realizados nessa área resultaram na criação de formatos de imagens distintos, porém, poucos são usados. Os mais comuns para armazenamento de imagens bidimensionais serão detalhados a seguir.

É importante observar que a maioria deles usa algoritmos para armazenar os dados de forma compactada, com objetivo de armazenar o maior número de informações possíveis, em um menor espaço – ou seja, consumindo menos bytes. Entretanto, o primeiro formato detalhado a seguir, conhecido popularmente como Bitmap, geralmente não utiliza nenhum algoritmo de compressão.

Imagens no formato BMP

Este é um formato de imagem desenvolvido pela Microsoft, nativo no sistema operacional Windows a partir da versão 3.0. O

Device Independent Bitmap (DIB) – também conhecido como Windows Bitmap (BMP) – é um formato que gera arquivos não compactados, isto é, geralmente sem nenhum tipo de compressão e perda, resultando em arquivos que ocupam bastante espaço em disco comparado aos outros formatos.

Podemos classificar os arquivos BMP de acordo com a quantidade de bits usados para representar um pixel, existindo as seguintes versões: 1 bit/pixel (duas cores), 4 bits/pixel (16 cores), 8 bits/pixel (256 cores), 24 bits/pixel (aproximadamente 16.8 milhões de cores) e também 32 bits/pixel – tornando possível representar um pouco mais de 4 bilhões de cores.

Por ser um formato de arquivo raramente utilizado com compressão, podemos facilmente calcular o tamanho do arquivo de uma imagem em BMP. Basta multiplicarmos o número de pixels da imagem pelo número de bits usados para representar um único pixel, e dividir o resultado por 8, assim obteremos o tamanho da imagem em bytes.

Por exemplo, se uma imagem possui a resolução espacial de 800px por 600px, com cores representadas em 8 bits (1 byte), temos 480 mil pixels, logo, 480 mil bytes ou 468 KB. Este formato é pouco utilizado comparado aos outros que serão apresentados justamente por gerar arquivos que ocupam muito espaço e pesados para serem transmitidos pela rede. Como, na maioria das vezes, a apresentação exata da imagem, sem nenhuma perda, não é necessária, formatos como JPEG e GIF são mais usados.

Imagens no formato JPEG

O Joint Photographic Experts Group (JPEG) é, na verdade, um

método para comprimir imagens fotográficas. Ele foi desenvolvido no ano de 1983 por um grupo conhecido pelo mesmo nome. Inicialmente, o termo utilizado para se referir a imagens compactadas pelo método JPEG era o JFIF (JPEG File Interchange Format).

Estes arquivos JFIF usam a extensão popularmente conhecida como JPG. O sucesso desse formato se deu por sua capacidade de armazenar imagens coloridas, com qualidade razoavelmente boa, em arquivos com tamanhos bem reduzidos quando comparado a outros formatos. Por essa característica, o JPEG se popularizou, sendo muito usado para transmissão de imagens na internet.

Outra qualidade deste método é a possibilidade de variar a intensidade da compressão, permitindo trocar qualidade por tamanho de acordo com as suas necessidades. O JPEG pode trabalhar com imagens de cores em até 24 bits. Isso significa que esse método e formato de imagem aceita aproximadamente 16.8 milhões de cores.

Acontece que o olho humano não é capaz de enxergar todas essas cores de uma vez, tornando possível a compactação da imagem e a redução de informações representadas. Assim como o JPEG, um outro formato muito utilizado na internet é o GIF, que também gera arquivos que ocupam pouco espaço em disco e são fáceis de serem transmitidos pela rede.

Imagens no formato GIF

O Graphics Interchange Format (GIF) é de propriedade da empresa CompuServe, desenvolvido em 1987. Nesta época, a CompuServe oferecia serviços online de e-mail e fórum, acessados

através de um cliente também desenvolvido pela própria empresa.

Inicialmente, a troca de imagens nesses serviços suportava apenas imagens em preto e branco, levando a empresa a desenvolver o GIF, um formato que suporta imagens coloridas e compactadas. Logo, este novo formato se popularizou e até hoje é muito utilizado em imagens para internet. Contudo, por ser limitado em apenas 256 cores, não é tão usado para armazenar fotografias.

É uma ótima solução para armazenar ícones e pequenas animações. Além disso, o GIF permite armazenar diversas imagens no mesmo arquivo, permitindo a criação de pequenas animações quadro a quadro, muito comuns na internet. Não demorou muito para esse algoritmo de compressão ser patenteado, motivando o desenvolvimento do formato PNG.

Imagens no formato PNG

Este foi um formato lançado no ano de 1996 pela W3C. O objetivo era criar um novo formato de arquivo de imagem para substituir o GIF. A motivação para o desenvolvimento do Portable Network Graphics (PNG) surgiu em 1995, quando o algoritmo de compressão utilizado pelo GIF foi patenteado. Outro fator que contribuiu para sua criação foi a incapacidade do formato GIF trabalhar com mais de 256 cores.

O formato PNG foi desenvolvido para ser livre e amplamente usado. Assim como o JPEG, ele também opera com cores representadas em até 24 bits, permitindo a representação de aproximadamente 16.8 milhões de cores. Além disso, possui um algoritmo de compressão mais eficiente, sem proporcionar perda

de qualidade a cada salvamento e com uma maior fidelidade à imagem original.

Diferente do GIF, o formato PNG é estático, ou seja, não suporta animações. É válido ressaltar que, apesar da qualidade da imagem armazenada, os arquivos PNG não são tão compactos quando comparados aos JPEG.

Imagens no formato TIF/TIFF

O Tagged Image File Format (TIFF) é um formato de arquivo de imagem desenvolvido pela Aldus em 1987. Por ter baixa (ou quase nenhuma) compressão, é considerado por muitos profissionais um bom formato para edição de imagens e impressão. Assim como o Bitmap (BMP), este também resulta em arquivos grandes, podendo armazenar imagens em preto e branco ou coloridas, com até 32 bits por pixel.

O TIFF, além de suportar um fundo transparente (como PNG e GIF), suporta o uso de camadas (diferentes versões da imagem no mesmo arquivo) quando manipulado com o software Photoshop, da Adobe, empresa responsável pelo formato nos dias atuais.

3.7 PRÁTICA COM PYTHON E OPENCV

Depois de todo esse estudo sobre aquisição de imagens, seus diferentes tipos e formatos, chegou a hora de aprender na prática como carregar imagens com Python e OpenCV. Nesta seção, serão apresentadas três maneiras diferentes de trabalhar com elas utilizando essas tecnologias. A primeira delas trata sobre como

carregar imagens a partir de arquivos.

Carregando imagens a partir de arquivos

O código apresentado a seguir exemplifica como carregar um arquivo de imagem e exibi-la para o usuário. A função da biblioteca OpenCV que carrega uma imagem a partir de um arquivo é a `imread`, e a que exibe a imagem carregada em uma variável, a `imshow`. O código seguinte apresenta como trabalhar com essas duas funções:

```
import cv2

imagem = cv2.imread("imagem.jpg")
cv2.imshow("Imagen", imagem)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Observe na primeira linha do código que a função `import` é executada. Ela é responsável por incluir no programa as bibliotecas que serão usadas – nesse exemplo, apenas a biblioteca OpenCV.

Ainda no código anterior, a terceira linha exemplifica o uso da função `imread`. Veja que, para armazenar uma imagem em uma variável, basta indicar o arquivo desejado como parâmetro dessa função. Na quarta linha, a função `imshow` exibe para o usuário a imagem armazenada na variável `imagem`.

Observe que essa função requer dois parâmetros. O primeiro deles refere-se à legenda da imagem, ou melhor, o título da janela onde a imagem será exibida. O segundo parâmetro indica a variável na qual a imagem está armazenada.

Carregando imagens a partir de vídeos

Avançando um pouco mais, nesse tópico será exemplificado como carregar imagens de um arquivo de vídeo, frame a frame. Essa tarefa será realizada pela função `VideoCapture` da biblioteca OpenCV, e a terceira linha do código a seguir exemplifica como usar essa função.

```
import cv2

captura = cv2.VideoCapture("video.mp4")

while True:
    ret, frame = captura.read()
    cv2.imshow("Imagen", frame)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

captura.release()
cv2.destroyAllWindows()
```

Nesse código, observe que, assim como a função `imread`, a `VideoCapture` requer um único parâmetro, dessa vez, responsável por indicar o arquivo de vídeo que será carregado.

Sendo um vídeo um conjunto de imagens exibidas em um dado intervalo de tempo, é necessária uma estrutura de repetição para que cada frame seja exibido. Para mostrar todos esses frames, o método `.read()` deve ser utilizado a cada loop de execução, junto à função `imshow`, já apresentada.

Nesse exemplo, o critério de parada do loop foi definido em uma estrutura condicional. Quando a tecla `q` for pressionada, o programa terá a sua execução interrompida. A função `break` força o fim da execução do loop. Já as duas últimas instruções do programa são, respectivamente, para fechar o arquivo de vídeo e as janelas abertas. No próximo tópico, será detalhado como obter

imagens a partir de uma webcam.

Capturando imagens a partir de uma webcam

Diferente de outras tecnologias, obter imagens a partir de uma webcam é uma tarefa simples e descomplicada quando trabalhamos com Python e a biblioteca OpenCV. Ela é tão simples, que o algoritmo é basicamente o mesmo mostrado no tópico anterior. A única diferença é o parâmetro definido para a função `VideoCapture`.

Para obtermos imagens em tempo real a partir de uma webcam em vez de um arquivo de vídeo, um valor inteiro (que identifica a webcam instalada no computador) deve ser informado. Se houver uma única webcam conectada ao computador, o interior referente a ela será o número 0. Consequentemente, se uma outra câmera for conectada, esta será referenciada com o número 1, e assim sucessivamente.

Compare o código exemplificado a seguir com o do tópico anterior. Veja que a diferença é apenas o parâmetro da função `VideoCapture`.

```
import cv2

captura = cv2.VideoCapture(0)

while True:
    ret, frame = captura.read()
    cv2.imshow("Imagem", frame)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

captura.release()
cv2.destroyAllWindows()
```

Este tópico finaliza o estudo sobre aquisição de imagens, em que foram apresentadas todas as etapas necessárias para obtê-las e carregá-las através do Python e da biblioteca OpenCV. No próximo capítulo, vamos aprofundar o estudo sobre imagens em cores e as diferentes maneiras de representá-las, bem como discutir detalhes sobre os modelos de representação de cores.

CAPÍTULO 4

REPRESENTAÇÃO DE CORES NO ESPAÇO

No capítulo anterior, foram apresentados diferentes tipos de imagem no que diz respeito à cor. Na seção *Tipos de imagens*, foram discutidas as características das binárias, com tons de cinza e coloridas. Neste capítulo, veremos com mais detalhes como as cores das imagens digitais podem ser representadas matematicamente.

Existem diferentes modelos matemáticos que tornam possível representá-las, conhecidos como *espaço de cor*. Vamos conhecer os dois mais usados em sistemas de Visão Computacional e Processamento de Imagens nas seções seguintes.

Estudar os diferentes modelos de representação de cor em imagens digitais é importante para facilitar o desenvolvimento de sistemas baseados em Visão Computacional. Ainda neste capítulo, você perceberá que algumas tarefas – por exemplo, segmentação de objetos em imagens por cor – podem ser realizadas de maneira mais rápida, exigindo menos processamento e dependendo do espaço de cor usado para tratar a imagem.

As imagens binárias e em tons de cinza são mais fáceis de

representar, e não há necessidade de tratá-las com diferentes espaços de cor. Relembrando o capítulo anterior: as binárias utilizam espaço de cor limitado a 2 bits, em que cada pixel representa apenas a cor preta ou branca. Já as imagens em tons de cinza, conhecidas também como imagens em escala de cinza, representam a intensidade de luz em cada pixel através de um único canal, em uma escala que varia de 0 (preto) a 255 (branco).

As imagens coloridas, diferente da em tons de cinza, utilizam de mais de um canal para serem representadas. O espaço de cor RGB é um dos modelos para representação de imagens coloridas, e ele utiliza três canais diferentes, cada um para representar a intensidade de uma cor primária em cada pixel da imagem. Na seção seguinte, iniciaremos o estudo teórico e prático sobre o espaço de cor RGB.

4.1 CORES NO ESPAÇO RGB

O modelo de representação de cores RGB é o mais usado em todos os aspectos, tanto para armazenar informações de cor em arquivos quanto para exibi-las em monitores – sejam telas de LCD (*Liquid Crystal Display*) ou CRT (*Catodic Ray Tube*). O espaço de cor RGB considera três cores primárias: vermelho, verde e azul. Estas, quando misturadas, permitem a representação das demais cores.

São justamente essas três cores que dão nome a esse espaço, em que cada letra da sigla é respectivamente a letra inicial do nome delas na língua inglesa (*red, green, blue*). A figura a seguir apresenta duas imagens: a primeira delas, da esquerda para a direita, ilustra a mistura das três cores primárias que representam

esse modelo de cor; a segunda ilustra a representação de todas as cores possíveis. Veja que essa representação forma um cubo no espaço.

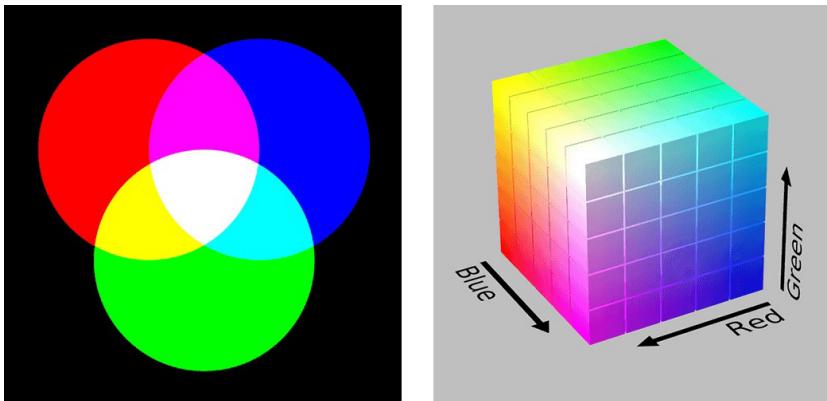


Figura 4.1: Cores no espaço RGB

As imagens em RGB, também conhecidas como imagens de cores reais, são matematicamente representadas por matrizes tridimensionais, que podem ser consideradas como três planos bidimensionais (ou três matrizes bidimensionais distintas), uma para cada canal. Cada elemento dessas matrizes bidimensionais é um número inteiro, variando de 0 a 255, que representa a intensidade de cor de um pixel referente a um canal.

Desta forma, uma matriz é utilizada para representar os valores de intensidade da cor vermelha, outra para a cor verde, e mais uma para a azul. Uma vez que cada elemento dessas matrizes é representado por valores inteiros (que variam de 0 a 255), cada canal de cor pode ser visualizado separadamente como uma imagem em tons de cinza. Este assunto será abordado de forma prática a seguir.

Segmentando canais de uma imagem RGB

Neste tópico, será detalhado como os canais de uma imagem colorida, representada em RGB, podem ser segmentados com a função `split` da biblioteca OpenCV. Vamos usar a figura seguinte como exemplo deste capítulo.



Figura 4.2: Imagem das frutas em RGB

O código exemplifica o procedimento de segmentação dos canais de uma imagem em RGB em três imagens distintas:

```
import cv2

# Carregando imagem RGB e segmentando canais
imagem = cv2.imread("frutas.jpeg")
azul, verde, vermelho = cv2.split(imagem)
```

```
# Exibindo imagens dos canais separados
cv2.imshow("Canal R", vermelho)
cv2.imshow("Canal G", verde)
cv2.imshow("Canal B", azul)

# Salvando imagens dos canais separados
cv2.imwrite("frutas-canal-vermelho.jpeg", vermelho)
cv2.imwrite("frutas-canal-verde.jpeg", verde)
cv2.imwrite("frutas-canal-azul.jpeg", azul)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

O primeiro passo desse código é carregar a imagem que será segmentada. Para isso, usamos a função `imread`, apresentada no capítulo anterior. Em seguida, definimos três matrizes: azul, verde e vermelho, cada uma para armazenar os números inteiros que representam a intensidade de cor de cada pixel de um determinado canal.

Por exemplo, na variável azul, será armazenada uma matriz, em que cada elemento dela representa a intensidade de cor azul de cada pixel da imagem. Observe que segmentar uma imagem em RGB, em seus respectivos canais, é uma tarefa extremamente simples quando utilizamos a função `split`, da biblioteca OpenCV. Basta chamar a função passando como parâmetro a imagem carregada, que ela nos retorna três matrizes, cada uma referente a um canal de cor.

Para exibir cada canal de cor individualmente, como uma imagem em tons de cinza, podemos utilizar a função `imshow` (também apresentada no capítulo anterior), passando como parâmetro a legenda da imagem e a matriz onde ela está armazenada na memória. Além de exibir as imagens geradas, podemos salvá-las em novos arquivos de imagem. Para isso, é

possível usar a função `imwrite`, basta definir como parâmetro o nome do arquivo que desejamos e a matriz onde ela se encontra.

A figura a seguir apresenta as três imagens geradas pelo código anterior. Cada uma delas, da esquerda para a direita, apresenta um canal de cor segmentado, na seguinte ordem: vermelho, verde e azul.



Figura 4.3: Canais da imagem em RGB segmentados

OBSERVAÇÃO

Geralmente, a biblioteca OpenCV trata o espaço de cor RGB (*red, green, blue*) como BGR (*blue, green, red*), invertendo a ordem dos canais. Por este motivo, quando utilizamos a função `split`, conforme demonstrado no código, declaramos as variáveis na sequência azul, verde, vermelho.

Assim como é possível segmentar os canais de uma imagem em RGB, no tópico a seguir veremos como combinar novamente os canais resultando na imagem original colorida.

Combinando canais em uma imagem RGB

Da mesma maneira como podemos segmentar os canais de uma imagem RGB, também é possível realizar a operação inversa, ou seja, mesclar novamente esses canais resultando em uma única imagem colorida. O código a seguir demonstra como esta tarefa pode ser realizada facilmente com Python e OpenCV.

```
import cv2

# Carregando imagem RGB e segmentando canais
imagem = cv2.imread("frutas.jpeg")
azul, verde, vermelho = cv2.split(imagem)

# Combinando os três canais em uma única imagem
imagem = cv2.merge((azul, verde, vermelho))
cv2.imshow("Imagen", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A lógica é basicamente a mesma comparada a do algoritmo para segmentar imagens, entretanto, apenas uma matriz é necessária para armazenar a imagem final, resultante da combinação dos três canais. Para combinar os canais, a função `merge` foi usada, e as matrizes referentes aos três canais segmentados foram definidas como parâmetros na função `split`.

Imagens coloridas em RGB também podem ser convertidas para imagens em tons de cinza, com um único canal de cor. Veremos mais sobre esse procedimento a seguir.

Convertendo imagens em RGB para tons de cinza

Com o objetivo de reduzir a quantidade de informações a serem processadas, a conversão de imagens coloridas em RGB para imagens em tons de cinza é uma tarefa quase sempre realizada em sistemas de Visão Computacional. As imagens em tons de cinza,

apesar de representarem menos detalhes, mantêm as características importantes dos objetos ou regiões de interesse, tais como bordas, regiões, manchas e junções.

Uma vez que as imagens em tons de cinza são representadas por uma única matriz, a fim de otimizar o processamento – tornando-o mais rápido –, elas são quase sempre usadas em sistemas de Visão Computacional. O código a seguir exemplifica como converter uma imagem em RGB para uma em tons de cinza, tarefa facilmente executada com a função `cvtColor` da biblioteca OpenCV.

```
import cv2

# Carregando a imagem em RGB
imagem = cv2.imread("frutas.jpeg")

# Convertendo e exibindo a imagem em tons de cinza
imagem = cv2.cvtColor(imagem, cv2.COLOR_RGB2GRAY)
cv2.imshow("Imagen", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Fazendo a leitura do código anterior, notamos que dois parâmetros são fundamentais quando utilizamos a função `cvtColor`. O primeiro é a matriz onde a imagem está armazenada na memória, e o segundo parâmetro é uma função que indica a conversão que será realizada. Nesse exemplo, foi utilizada a função `COLOR_RGB2GRAY`, definindo que a imagem RGB será convertida para a escala de cinza.

A figura a seguir apresenta duas imagens: a original em RGB à direita, e a imagem convertida em tons de cinza (gerada pela execução do código) à esquerda.



Figura 4.4: Canais da imagem em RGB segmentados

Apesar de não ser o objetivo deste livro abordar detalhes matemáticos teóricos, é válido mencionar que a imagem em tons de cinza, após a conversão, é gerada a partir da soma ponderada dos canais de cor vermelho, verde e azul. Nessa soma, os canais vermelho e verde apresentam um peso (coeficiente de ponderação) maior quando comparado ao canal azul, justamente pelo fato do olho humano ser mais sensível a essas cores.

Os coeficientes de ponderação são diferentes para cada canal, e todos são padronizados e definidos com base na resposta perceptual do olho humano para cada uma dessas cores.

Na seção a seguir, será apresentado o espaço de cor HSV, também utilizado para representar imagens coloridas.

4.2 CORES NO ESPAÇO HSV

Em sistemas de Visão Computacional e Processamento de Imagens, o espaço de cor HSV é um dos modelos principais e mais utilizados para a representação de cor. Ao representar as informações de uma imagem colorida, ele permite um grande grau

de separação entre a cor e a iluminação. Por este motivo, é frequentemente usado em algoritmos para segmentar objetos coloridos em imagens.

Diferente do espaço RGB, o espaço HSV não utiliza canais para representação de cores primárias, mas usa três canais para representar a matiz (*hue*), a saturação (*saturation*) e o valor (*value*). A primeira letra do nome de cada um desses canais na língua inglesa, quando unificadas, formam a sigla HSV, que denomina esse espaço de cor. A seguir, veja a definição de cada um desses componentes.

Matiz

Este é o componente que nos permite diferenciar visualmente o azul do vermelho e do verde. Representa a tonalidade da cor, o que nos permite descrever uma cor "pura" como semelhante ou diferente de outra, sem adição de preto ou branco. Pode ser vista fisicamente como o comprimento de onda dominante da cor.

A figura a seguir exibe a imagem com as cores reais ao centro, a com -80% de matiz à esquerda, e a com +80% de matiz à direita. Analise e perceba as diferenças entre as imagens. Observe que a luminosidade geral da imagem, bem como as cores intensamente claras ou escuras, como preto e branco, tendem a ser preservadas.



Figura 4.5: Variação na matiz da imagem

Saturação

É a pureza ou intensidade da cor. Quanto maior o valor da saturação, mais pura será a cor; quanto menor, mais próxima ao seu tom de cinza ela será representada. Ao reduzir completamente a saturação de uma imagem, ela se transformará em uma imagem em tons de cinza.

A figura a seguir exibe a imagem com as cores reais ao centro, a com -80% de saturação à esquerda, a com +80% de saturação à direita. Analise e note as diferenças entre elas.



Figura 4.6: Variação na saturação da imagem

Valor

Este valor é referente ao brilho da cor, à luminosidade ou escala de claridade. Quanto maior a luminosidade, ou o valor deste componente, mais próximo ao branco a cor será representada; quanto menor, mais próxima ao preto. Ao aumentar completamente a luminosidade de uma imagem, ela se transformará em uma imagem completamente branca, mas, ao realizarmos a operação inversa, ela se transformará em uma imagem completamente preta.

A figura a seguir exibe a imagem com as cores reais ao centro, a com -80% de luminosidade à esquerda, a com +80% de

luminosidade à direita. Veja também as diferenças entre as imagens.



Figura 4.7: Variação no brilho da imagem

Diferente da representação em cubo do espaço RGB, o espaço HSV é representado preferencialmente por um cilindro ou um cone. Essas duas figuras geométricas tridimensionais representam melhor as cores representadas por esse modelo. A figura seguinte apresenta geometricamente o espaço HSV.

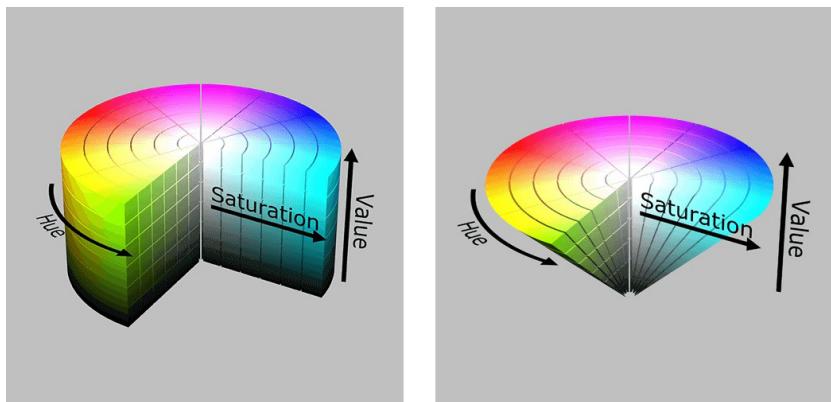


Figura 4.8: Cores no espaço HSV

Entendido que o espaço de cor HSV representa imagens por meio de três componentes, no tópico seguinte será demonstrado como segmentar cada uma delas com a biblioteca OpenCV.

Segmentando canais em uma imagem HSV

Assim como as imagens em RGB podem ser segmentadas em três canais distintos com a função `split` da biblioteca OpenCV, o mesmo pode ser feito com imagens representadas no espaço HSV. O código seguinte demonstra como a imagem da cesta de frutas pode ter seus canais de matiz, saturação e valor segmentados.

```
import cv2

imagem = cv2.imread("frutas.jpeg")
imagem = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)
matiz, saturacao, valor = cv2.split(imagem)

cv2.imshow("Canal H", matiz)
cv2.imshow("Canal S", saturacao)
cv2.imshow("Canal V", valor)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Observe no código que, antes de segmentar os canais da imagem, é necessário convertê-la para o espaço de cor HSV utilizando novamente a função `cvtColor`, mas, desta vez, com o segundo parâmetro definido como `COLOR_BGR2HSV`, para indicar a conversão do espaço RGB para o HSV. A figura a seguir apresenta os três canais da imagem representada em HSV segmentados em três imagens em tons de cinza.



Figura 4.9: Canais da imagem em HSV segmentados

Na figura anterior, a primeira imagem da esquerda para a direita representa a matiz; a segunda, a saturação; e a última, o brilho (ou luminosidade).

Combinando os canais em uma imagem HSV

Assim como é possível segmentar os canais de uma imagem representada no espaço HSV, também é possível combiná-los, restaurando a imagem original. Este procedimento pode ser realizado com a função `merge`, a mesma demonstrada na seção anterior quando os canais de uma imagem em RGB foram combinados.

O código demonstra como realizar essa operação com a biblioteca OpenCV:

```
import cv2

imagem = cv2.imread("frutas.jpeg")
imagem = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)

matiz, saturacao, valor = cv2.split(imagem)

cv2.imshow("Canal H", matiz)
cv2.imshow("Canal S", saturacao)
cv2.imshow("Canal V", valor)

imagem = cv2.merge((matiz, saturacao, valor))
imagem = cv2.cvtColor(imagem, cv2.COLOR_HSV2BGR)

cv2.imshow("Imagen", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Repare no código que, após os canais serem combinados com a função `merge`, foi necessário converter a imagem para o espaço RGB antes de exibi-la. É válido lembrar que nossos monitores

operam exibindo imagens representadas no espaço de cor RGB e, por este motivo, essa conversão é necessária.

No tópico seguinte, serão citadas algumas aplicações práticas do espaço HSV para representação de cores em sistemas de Visão Computacional.

Aplicações do espaço HSV

Quando desejamos segmentar objetos coloridos em uma imagem, o espaço de cor HSV é o modelo ideal a ser utilizado, justamente por representar as informações relacionadas a cor em um canal exclusivo e ajudar a distinguir objetos de uma cor de outra.

Pensando novamente no exemplo das fichas de poker, abordado no primeiro capítulo, imagine que desejamos desenvolver um sistema para contar o número de fichas vermelhas sobre a mesa. Nesse caso, precisaríamos converter a imagem capturada pela câmera, originalmente no espaço RGB, para o espaço HSV. Isso facilitaria bastante o trabalho, tornando mais simples a tarefa de segmentar somente as fichas vermelhas para contabilizá-las posteriormente.

Rastrear objetos coloridos em movimento também é uma aplicação muito comum em sistemas baseados em Visão Computacional. Esta técnica é frequentemente usada para acompanhar a movimentação de um robô, pessoas vestindo roupas de uma determinada cor, bolas coloridas em jogos ou até mesmo uma boia flutuando em um reservatório.

A fim de ilustrar uma dessas aplicações, a imagem a seguir

exibe um frame de uma captura em tempo real. Nesse frame, um objeto de interesse amarelo foi segmentado, possibilitando o rastreamento da sua posição na imagem. Mais à frente, veremos como segmentar objetos coloridos usando o espaço de cor HSV.

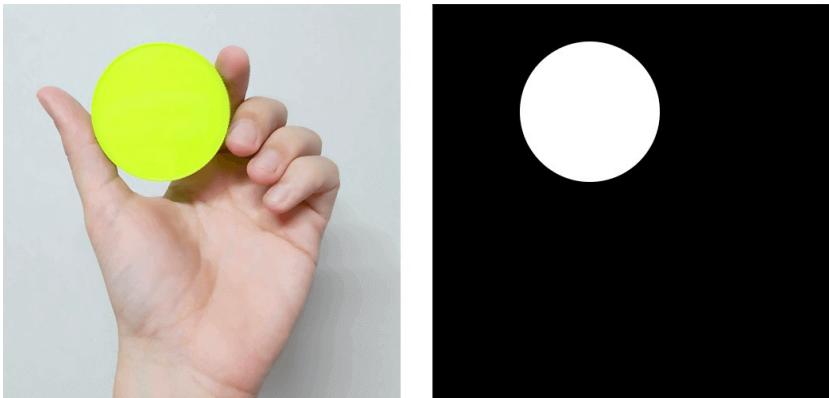


Figura 4.10: Rastreamento de objeto colorido

Neste capítulo, foram apresentados os dois principais modelos matemáticos para representação de cor. Estes são frequentemente utilizados em sistemas baseados em Visão Computacional. O próximo capítulo trata sobre a etapa de pré-processamento da imagem, e as operações e os métodos para realçar objetos de interesse.

CAPÍTULO 5

PRÉ-PROCESSAMENTO

Em um sistema baseado em Visão Computacional, a etapa de pré-processamento consiste em realçar objetos de interesse em imagens, facilitando segmentá-los posteriormente. Para realçar esses objetos, existem inúmeros procedimentos que podem ser realizados, como operações aritméticas, operações geométricas, métodos para ajuste de contraste e tratamento de ruído. Neste capítulo, vamos apresentá-los e exemplificá-los, bem como abordar os conceitos e os diferentes tipos de ruído em imagens.

5.1 OPERAÇÕES BÁSICAS

Existem algumas operações básicas que são fundamentais quando trabalhamos com processamento de imagens. Duas bastante utilizadas são: a obtenção e a modificação de valores numéricos que representam a cor de um determinado pixel.

Além disso, frequentemente precisamos obter informações sobre a imagem, extraíndo dados como: tamanho do arquivo, número de canais e quantidade de linhas e colunas. Nesta seção, essas operações serão detalhadas em dois tópicos. O primeiro deles trata sobre como acessar e modificar valores de pixels.

Acessando e modificando valores de pixels

Visto que OpenCV trata uma imagem como uma matriz de pixels, podemos acessar e modificar seus valores, assim como acessamos e modificamos os valores dos elementos de uma matriz qualquer. É necessário apenas informar a coordenada, o número da linha e a coluna do pixel cujo valor desejamos obter ou alterar.

O código exemplifica como podemos obter o valor numérico, que representa a cor de um pixel em uma imagem colorida:

```
import cv2
imagem = cv2.imread("frutas.jpeg")
valorPixel = imagem[150,150]
print(valorPixel)
```

Veja o resultado da execução:

```
[25 33 187]
[Finished in 0.6s]
```

Observe que foi retornado um vetor contendo três valores inteiros. Isso ocorre pois estamos trabalhando com uma imagem colorida, representada em RGB. Sendo assim, cada pixel possui três valores, referentes respectivamente à intensidade de vermelho, verde e azul.

No caso de uma imagem representada em tons de cinza, o resultado da execução será um único valor inteiro, referente à intensidade de luz representada pelo pixel. O código a seguir exemplifica esta execução, convertendo a imagem original em RGB em uma imagem em tons de cinza.

```
import cv2
imagem = cv2.imread("frutas.jpeg")
imagem = cv2.cvtColor(imagem, cv2.COLOR_RGB2GRAY)
valorPixel = imagem[150,150]
print(valorPixel)
```

Veja o resultado dessa execução:

```
48  
[Finished in 0.5s]
```

Ainda sobre imagens coloridas, podemos obter, de um determinado pixel, o valor da intensidade da cor de um único canal. Desta forma, será retornado um único valor inteiro, diferente do primeiro exemplo, em que os três valores são retornados em um vetor. O código seguinte exemplifica esta execução.

```
import cv2  
imagem = cv2.imread("frutas.jpeg")  
valorPixel = imagem[150, 150, 0]  
print(valorPixel)
```

Observe no código que basta considerar a matriz que representa a imagem como uma matriz tridimensional. A terceira dimensão indica o canal de cor do espaço RGB, variando de 0 até 2. Neste caso, o canal 0 é referente à intensidade de azul; o canal 1, à intensidade de verde; e o 2, de vermelho.

O resultado da execução será:

```
25  
[Finished in 0.1s]
```

Para alterar o valor de um determinado pixel da imagem, basta realizar a operação inversa: atribuir novos valores a um determinado elemento da matriz. Veja o exemplo:

```
import cv2  
imagem = cv2.imread("frutas.jpeg")  
print(imagem[150, 150])  
imagem[150, 150] = [255, 255, 255]  
print(imagem[150, 150])
```

O resultado dessa execução será:

```
[25 33 187]  
[255 255 255]  
[Finished in 1.5s]
```

Na terceira linha do código exemplificado, exibimos o valor numérico referente ao pixel da linha 150 e coluna 150. Na quarta linha, esse valor foi modificado para o equivalente à cor branca. No resultado da execução, o valor do elemento foi impresso antes e após a alteração.

Acessando informações sobre a imagem

A biblioteca OpenCV também nos permite obter algumas informações básicas sobre a imagem, como o número de linhas e colunas, os canais e a quantidade de pixels. Neste tópico, veremos como é possível retornar cada uma dessas informações.

Pelo método `shape`, conseguimos obter o número de linhas, colunas e canais de uma determinada imagem:

```
import cv2  
imagem = cv2.imread("frutas.jpeg")  
print(imagem.shape)
```

Veja o resultado da execução:

```
(600, 800, 3)  
[Finished in 0.2s]
```

Analizando a execução desse código, observamos que a imagem `frutas.jpeg` possui 600 linhas, 800 colunas e está representada em 3 canais de cor. Caso a imagem carregada seja em tons de cinza, será retornado um vetor contendo apenas dois elementos, referentes ao número de linhas e colunas. Desta forma,

podemos utilizar esse método para verificar se a figura carregada possui cores ou não.

Para obtermos o total de pixels da imagem, podemos multiplicar o total de linhas por colunas ou, simplesmente, utiliza o método `size`, exemplificado a seguir.

```
import cv2
imagem = cv2.imread("frutas.jpeg")
print(imagem.size)
```

O resultado da execução será:

```
1440000
[Finished in 1.6s]
```

Observe que o valor retornado pelo método `size` é o produto da quantidade de linhas, colunas e canais. Quando estamos trabalhando com imagens coloridas, representadas em três canais, devemos dividir o valor obtido por três. Dessa maneira, o produto pelo número de canais será desconsiderado.

Nesse exemplo, como o valor `1440000` foi retornado, dividindo-o por três, obtemos o total de `480000` pixels, equivalente ao produto do número de linhas pelo número de colunas. Esse mesmo método, quando aplicado a imagens em tons de cinza representadas em um único canal, retornará de imediato o seu total de pixels, sem que nenhuma outra operação seja necessária.

Outro recurso que pode nos revelar dados importantes sobre a imagem chama-se histograma de cores. Na seção seguinte, vamos entender o seu funcionamento e quais informações ele pode nos fornecer.

5.2 HISTOGRAMA DE CORES

O histograma de cores de uma imagem é a distribuição de frequência dos níveis de cinza em relação ao número de amostras. Essa distribuição nos fornece informações sobre a qualidade da imagem, principalmente no que diz respeito à intensidade luminosa e ao contraste.

Nos tópicos desta seção, será detalhado como podemos obter o histograma de diferentes tipos de imagens. A figura seguinte ilustra respectivamente três fotografias de uma folha: uma binária, outra em tons de cinza e, por último, uma colorida. Elas serão usadas nos exemplos desta seção. Seguindo a mesma ordem, o primeiro tópico exemplifica como é possível obtermos o histograma de uma imagem binária.



Figura 5.1: Imagens de uma folha em diferentes espaços de cor

Histograma em uma imagem binária

Por possuir pixels representados apenas pela cor preta ou branca, o histograma de cores de uma imagem binária pode ser facilmente obtido. O total de pixels da imagem subtraído do total de pixels de uma determinada cor nos fornece a quantidade de pixels pretos ou brancos representados.

Estes dados são suficientes para plotar o histograma de uma imagem binária. O total de pixels pretos ou brancos pode ser obtido percorrendo toda a matriz que representa a imagem, contando-os individualmente, assim como exemplificado no código:

```
import cv2

imagem = cv2.imread("folha-binaria.bmp", 0)
totalPixelsPreto = 0;
totalPixelsBranco = 0;

for x in range(0, 499):
    for y in range(0, 499):
        if imagem[x,y] == 255:
            totalPixelsBranco += 1;
        else:
            totalPixelsPreto += 1;

print(totalPixelsBranco)
print(totalPixelsPreto)
```

O resultado dessa execução será:

```
190903
58098
[Finished in 0.6s]
```

Observe no código que a imagem binária carregada foi salva em um arquivo do tipo Bitmap, justamente para garantir a fidelidade das cores dos pixels, sendo eles pretos ou brancos. Salvar uma imagem binária utilizando um formato compactado, como JPEG, poderia resultar em pixels com valores intermediários, em tons de cinza, não apenas pretos e brancos.

Analisando o mesmo código, repare que o parâmetro `0` foi utilizado na função `imread`. Ele indica a leitura da imagem em tons de cinza; por esse motivo, os pixels brancos são representados pelo valor 255 e os pretos por 0.

Mesmo sendo um algoritmo simples e de fácil entendimento, esse procedimento não é muito utilizado para obter o total de pixels de cada cor. Isso porque a biblioteca Matplotlib possui a função `hist`, que, além de abstrair essa tarefa, gera a figura que ilustra o histograma de cores da imagem. O código exemplifica como importar essa biblioteca e utilizar a função `hist`:

```
import cv2
import numpy as np
from matplotlib import pyplot as grafico

imagem = cv2.imread("folha-binaria.bmp", 0)
grafico.hist(imagem.ravel(), 256, [0,256])
grafico.show()
```

Observe que a função `hist` possui 3 parâmetros fundamentais: o primeiro é a imagem com a qual desejamos trabalhar, o segundo é o número de elementos distintos que podem ser representados, e o terceiro indica o intervalo entre os elementos. Como se trata de uma imagem em tons de cinza, é válido lembrar que estes elementos são números inteiros, variando de 0 (preto) a 225 (branco), ou seja, 256 tonalidades distintas.

O método `ravel`, aplicado no primeiro parâmetro, tem a finalidade de transformar a imagem em um vetor, isto é, organizar todos os elementos em uma estrutura, contendo uma única linha e N colunas – em que N representa o total de pixels da imagem. Executando o código exemplificado, obtemos o seguinte histograma para a imagem carregada:

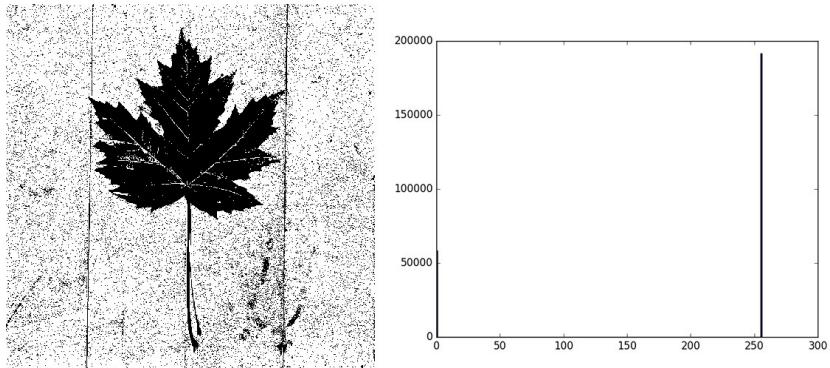


Figura 5.2: Histograma de uma imagem binária

Analisando o histograma e os valores obtidos, podemos perceber algumas características da imagem, por exemplo, o número de pixels brancos é predominante. Note que eles representam três vezes mais que o de pixels pretos.

A priori, pode não parecer uma informação muito relevante, entretanto, características como esta podem ajudar a reconhecer objetos em imagens. Esse assunto será tratado com mais detalhes em outro capítulo. No próximo tópico, a função `hist` será utilizada para plotar o histograma de uma imagem em tons de cinza.

Histograma em uma imagem em tons de cinza

Nas imagens em tons de cinza, por haver grande diversidade de tons, o histograma pode ser representado em classes que contabilizam os valores em determinados intervalos. Quanto maior o número de classes, mais informações sobre a imagem podem ser representadas.

Por se tratar de uma imagem em 8 bits, o número máximo de classes para representar os intervalos é 256. Executando o mesmo código apresentado no tópico anterior, carregando a imagem em tons de cinza da folha, obtemos o histograma ilustrado na figura a seguir. Uma das informações que ele nos fornece é o nível de contraste da imagem.

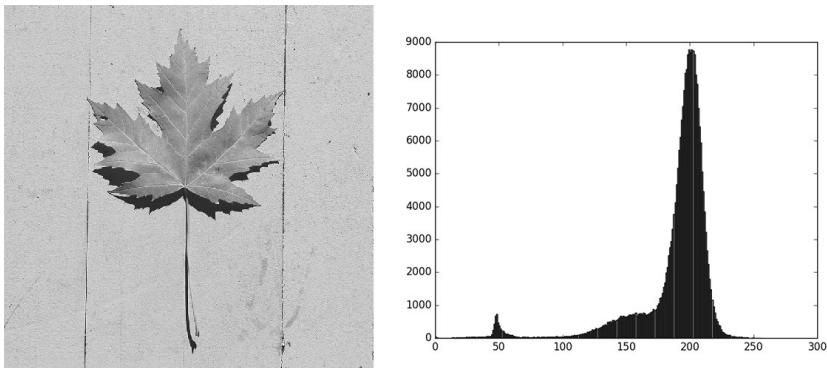


Figura 5.3: Histograma de uma imagem em tons de cinza

O histograma ilustrado na figura revela que a imagem processada é uma com baixo nível de contraste. Imagens com baixo nível de contraste são caracterizadas por apresentarem histogramas estreitos, ou seja, muitos elementos concentrados em intervalos pequenos.

Mais detalhes sobre análise de contraste, em histogramas de imagens, serão apresentados no último tópico desta seção. A seguir, veremos como obter o histograma de uma imagem em cores.

Histograma em uma imagem colorida

Uma imagem colorida possui um histograma para cada canal de cor. Uma vez que cada canal é representado por pixels em tons de cinza, o mesmo método estudado nos tópicos anteriores pode ser utilizado para plotar esses histogramas.

O código a seguir demonstra como obter o histograma de cada canal individualmente.

```
import cv2
import numpy as np
from matplotlib import pyplot as grafico

imagem = cv2.imread("folha-colorida.bmp")
azul, verde, vermelho = cv2.split(imagem)

grafico.hist(azul.ravel(), 256, [0,256])

grafico.figure();
grafico.hist(verde.ravel(), 256, [0,256])

grafico.figure();
grafico.hist(vermelho.ravel(), 256, [0,256])

grafico.show()
```

Nesse código, carregamos uma imagem em RGB para ser processada. Em seguida, seus canais foram segmentados em azul, verde e vermelho. Com cada canal segmentado, utilizamos a função `hist` para plotar cada histograma.

A função `figure`, também usada no algoritmo anterior, indica que uma nova figura será gerada, possibilitando criar os três histogramas em uma só execução. A figura seguinte apresenta os histogramas retornados pela execução do código. O primeiro histograma, da esquerda para a direita, apresenta os dados referente ao canal vermelho; o histograma do meio, os dados do canal verde; e o último, os dados do canal azul.

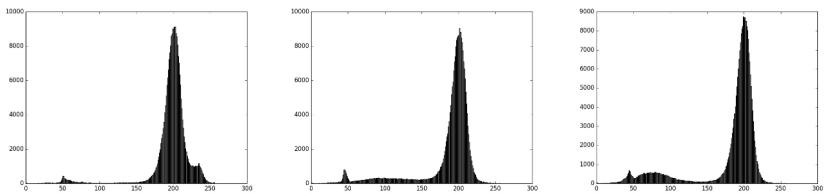


Figura 5.4: Histograma dos canais de uma imagem em RGB

Para facilitar a análise dos histogramas apresentados, a figura a seguir sobrepõe os traçados em cores que representam seus respectivos canais. Observe a diferença entre eles.

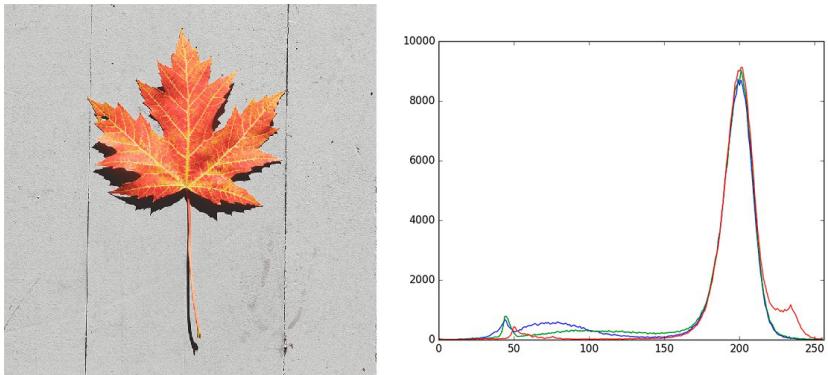


Figura 5.5: Histograma de uma imagem em RGB

O histograma de uma imagem colorida pode nos fornecer informações sobre a cor predominante, ou que possui maior destaque na imagem. Observe o histograma do canal vermelho. Ele possui uma curva entre os valores 200 e 250 que se diferencia consideravelmente do canal azul e verde. Isso ocorre porque a folha ilustrada na fotografia, além de possuir tons avermelhados, ocupa uma área significativa da imagem.

As condições de luminosidade do ambiente no qual as imagens

são capturadas podem comprometer a nitidez dos objetos de interesse. Um método conhecido como equalização de histograma nos permite realçar os objetos tornando-os mais nítidos.

No tópico seguinte, será detalhado o funcionamento do método de equalização de histograma, como também veremos como executá-lo através de uma função da biblioteca OpenCV.

Equalização de histograma

Antes de estudar o procedimento de equalização de histograma, é preciso aprender a interpretá-lo, obtendo informações sobre a imagem que ele representa. A exposição à luz da cena capturada e o nível de contraste da fotografia são dados importantes que podem ser extraídos de um histograma.

Imagens superexpostas (ou seja, com alto nível de luminosidade) geralmente apresentam histogramas com a maior parte dos elementos concentrados à direita. Do contrário, histogramas que possuem maior parte dos elementos concentrados à esquerda tendem a representar imagens subexpostas, com baixo nível de luminosidade.

Na figura a seguir, o histograma à esquerda representa uma imagem subexposta, e o à direita, uma imagem superexposta.

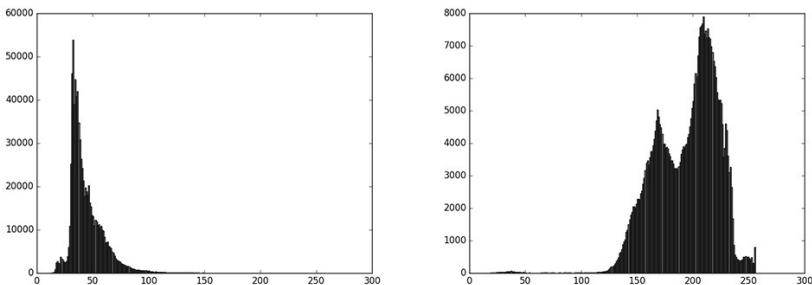


Figura 5.6: Análise de exposição em histograma

O histograma também pode fornecer informações sobre o contraste da imagem que ele representa. Imagens com baixo nível de contraste apresentam menor nitidez, sendo caracterizadas por histogramas estreitos, nos quais os elementos estão concentrados em intervalos menores. Do contrário, imagens com alto nível de contraste apresentam histogramas largos, com elementos distribuídos por toda faixa de tons de cinza disponível.

Para facilitar o entendimento, analise a figura seguinte, em que a imagem à esquerda ilustra o histograma de uma imagem com baixo nível de contraste, e a imagem à direita mostra o histograma de outra imagem, com alto nível de contraste.

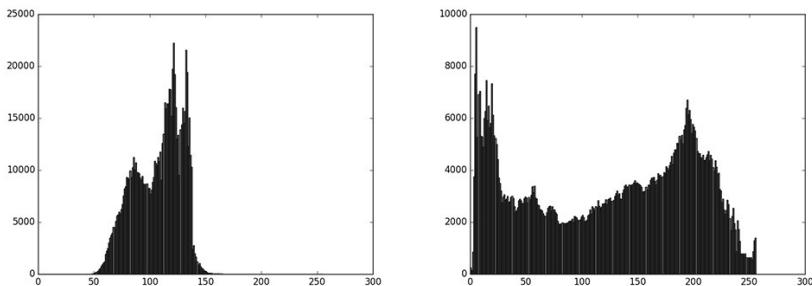


Figura 5.7: Análise de contraste em histograma

Dependendo do ambiente onde a captura da imagem é realizada, o objeto de interesse pode não apresentar a nitidez desejada. Observando a fotografia apresentada na figura a seguir, imagine um software capaz de ler o valor marcado pelos ponteiros da máquina. Repare que o ambiente enfumaçado pode dificultar a leitura dos ponteiros.

A fumaça do ambiente contribui para a captura de imagens com menor nível de contraste, resultando em ponteiros menos nítidos. O procedimento de equalização de histograma pode ser uma possível solução para esse problema, sendo capaz de melhorar o contraste e a nitidez do objeto de interesse.

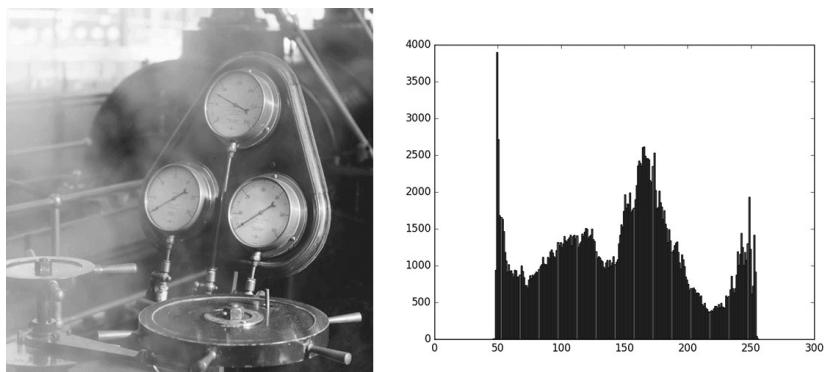


Figura 5.8: Histograma de uma imagem em RGB

Veja que, no eixo horizontal do histograma apresentado na figura, nenhum elemento foi representado em tons de cinza variando de 0 a 50. Isso significa que os elementos não estão distribuídos por toda a faixa de tons disponível, condição ideal para representar uma fotografia com nível de contraste apropriado.

O objetivo da equalização de histograma é justamente modificar a tonalidade dos pixels da imagem, com intuito de redistribuir o histograma. Isso resultará em uma imagem mais nítida, contendo pixels com tonalidades de cinza variando de 0 a 255.

A função `equalizeHist` da biblioteca OpenCV nos permite equalizar histogramas de imagens. Ela possui como único parâmetro obrigatório a matriz que representa a imagem carregada. Executando essa função, uma nova imagem com o histograma equalizado será retornada.

O código seguinte exemplifica esse procedimento. O resultado da execução exibe a imagem original, a imagem com o histograma equalizado e também gráficos referentes ao histograma de ambas as imagens.

```
import cv2
import numpy as np
from matplotlib import pyplot as grafico

imagemOriginal = cv2.imread("maquina.jpg", 0)
imagemEqualizada = cv2.equalizeHist(imagemOriginal)

cv2.imshow("Imagen Original", imagemOriginal)
cv2.imshow("Imagen Equalizada", imagemEqualizada)

grafico.hist(imagemOriginal.ravel(), 256, [0,256])

grafico.figure();
grafico.hist(imagemEqualizada.ravel(), 256, [0,256])

grafico.show()
```

A figura a seguir apresenta a imagem e o histograma equalizado gerado pelo código anterior. Compare as figuras originais, tanto da imagem quanto do histograma, com as geradas

após a execução da função `equalizeHist`.

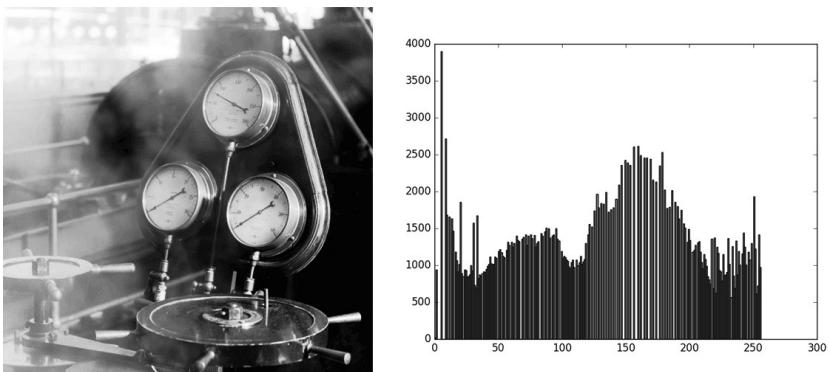


Figura 5.9: Histograma de uma imagem em RGB

Observe que a imagem gerada após a execução da equalização apresenta ponteiros mais nítidos, bem como os demais objetos da cena. O histograma da nova imagem possui elementos que variam a tonalidade entre 0 a 255, mesmo não havendo pixels representados por todos esses valores.

Sempre que uma imagem é submetida a esse procedimento, seu histograma torna-se mais largo, com pequenos intervalos ou valores nos quais não há nenhum elemento representado. Sendo assim, essa análise pode ser utilizada para reconhecer imagens que foram manipuladas digitalmente ou, pelo menos, submetidas ao procedimento de equalização.

Nesta seção, apresentamos como é possível equalizar o histograma de uma imagem em tons de cinza, que contém apenas um único canal de cor. A ideia de equalizar uma imagem em cores aplicando o mesmo método em cada um dos seus três canais (R, G, B) pode parecer tentadora, entretanto, não funcionaria.

Os valores que representam os pixels, em uma imagem em RGB, determinam tanto a intensidade quanto à cromaticidade do pixel. Aplicando esse mesmo método nelas, as cores reais sofreriam distorções, e o conteúdo cromático da imagem seria afetado.

Uma solução para equalizar uma imagem colorida é convertê-la para o espaço de cor HSV. Nesse espaço, os canais H e S representam as informações relacionadas à cor; já o canal V representa somente a intensidade de luz referente ao pixel. Dessa maneira, após a conversão da imagem original em RGB para o espaço HSV, basta segmentar os canais, aplicar a função de equalização ao canal de intensidade (canal V) e, por fim, converter o resultado novamente para o espaço de cor RGB.

Assim como podemos olhar para uma fotografia observando suas cores, tonalidades e contrastes, os dados tabulados de um histograma permitem através dos números – que computadores também possam enxergar essas características em uma imagem. Por este motivo, essa ferramenta estatística é tão importante para os sistemas de Visão Computacional.

Na seção seguinte, veremos as transformações geométricas que podem ser aplicadas a imagens.

5.3 TRANSFORMAÇÕES GEOMÉTRICAS

Muitas vezes precisamos aplicar transformações geométricas em imagens. Estas transformações consistem em invertê-la horizontalmente ou verticalmente, rotacioná-la, ajustar sua escala ou a perspectiva. Nos tópicos desta seção, será exemplificado como

cada uma dessas transformações podem ser realizadas com auxílio da biblioteca OpenCV.

Operação de rotação

A operação de rotação consiste em girar a imagem em um ângulo predeterminado. Para realizar essa operação, duas funções da biblioteca OpenCV são necessárias: a `getRotationMatrix2D` e a `warpAffine`. A primeira delas é utilizada para obter a matriz de rotação, que será usada como parâmetro na função `warpAffine`. A função `getRotationMatrix2D` também possui parâmetros, e cada um deles foi discriminado na tabela:

Parâmetro	Descrição
<code>center</code>	Centro da imagem.
<code>angle</code>	Ângulo de rotação em graus.
<code>scale</code>	Fator de escala.

O parâmetro `center` da função `getRotationMatrix2D` indica o centro da imagem, ou seja, uma coordenada definida por dois pontos. O primeiro deles é o total de colunas dividido por 2; o segundo, o total de linhas dividido por 2. O segundo parâmetro, `angle`, define o ângulo no qual desejamos rotacionar a imagem, que pode variar de 0° até 360° . O último parâmetro, `scale`, é o fator de escala. Para manter a imagem com a mesma dimensão, basta atribuir o valor 1.

O código a seguir exemplifica como é possível obter uma matriz de rotação, para girar uma imagem em 90° sem alterar sua dimensão.

```
matriz = cv2.getRotationMatrix2D(
```

```
)  
    (totalColunas / 2, totalLinhas / 2), 90, 1
```

Com a matriz de rotação obtida, a próxima etapa consiste em gerar a imagem rotacionada. Para esse procedimento, vamos utilizar a função `warpAffine`, que possui seus três parâmetros obrigatórios discriminados a seguir.

Parâmetro	Descrição
<code>src</code>	Matriz referente à imagem.
<code>matriz</code>	Matriz de rotação.
<code>dsize</code>	Tamanho da imagem rotacionada.

Todo o procedimento será exemplificado no código adiante. Nesse exemplo, a imagem `folha.jpeg` é rotacionada em 90° sem que sua dimensão original seja alterada.

```
import cv2  
import numpy as np  
  
imagemOriginal = cv2.imread("folha.jpeg", 0)  
totalLinhas, totalColunas = imagemOriginal.shape  
  
matriz = cv2.getRotationMatrix2D(  
    (totalColunas / 2, totalLinhas / 2), 90, 1  
)  
  
imagemRotacionada = cv2.warpAffine(  
    imagemOriginal,  
    matriz,  
    (totalColunas, totalLinhas)  
)  
  
cv2.imshow("Resultado", imagemRotacionada)  
  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

A figura seguinte apresenta duas imagens. A primeira delas, da esquerda para a direita, exibe a imagem original carregada, a segunda, a nova imagem gerada rotacionada em 90º.



Figura 5.10: Operação de rotação

No próximo tópico, veremos a operação de translação.

Operação de translação

A operação de translação consiste em deslocar uma imagem de posição. A função que nos permite realizar essa operação é a `warpAffine`, a mesma utilizada para rotacioná-la. Para deslocar a imagem, precisamos de uma matriz de translação, que indicará para qual posição a imagem será movida.

Para gerar a matriz de translação ou deslocamento, usaremos a função `float32` da biblioteca Numpy. Dois vetores com três elementos são os parâmetros necessários para esse procedimento. O último elemento do primeiro vetor indica a quantidade de pixels referente ao deslocamento horizontal. O mesmo ocorre com o último elemento do segundo vetor, entretanto, este indica o

deslocamento vertical em pixels.

O código a seguir exemplifica a operação de translação em uma imagem. A matriz de translação gerada tem por finalidade deslocar a imagem horizontalmente em 100 pixels para a direita, como também deslocar verticalmente em 100 pixels para baixo.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("folha.jpeg")
totalLinhas, totalColunas = imagemOriginal.shape[:2]

matriz = np.float32([[1, 0, 100], [0, 1, 100]])
imagemDeslocada = cv2.warpAffine(
    imagemOriginal,
    matriz,
    (totalColunas, totalLinhas)
)

cv2.imshow("Resultado", imagemDeslocada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura a seguir apresenta duas imagens. A primeira delas, da esquerda para a direita, exibe a imagem original carregada. A segunda imagem apresenta o resultado do processamento realizado pelo código anterior, uma nova imagem gerada deslocada horizontal e verticalmente em 100 pixels.



Figura 5.11: Operação de translação

Em sistemas de Visão Computacional, a fim de tornar o processamento de imagens mais rápido, muitas vezes precisamos redimensionar imagens que foram obtidas em altas resoluções. No tópico seguinte, veremos como realizar ajuste de escala em imagens.

Ajuste de escala

A biblioteca OpenCV possui a função `resize`, que nos permite realizar ajuste de escala em imagens – isto é, alterar proporcionalmente suas dimensões sem que o objeto representado seja distorcido.

Na tabela a seguir, foram discriminados os cinco parâmetros necessários para executar essa função:

Parâmetro	Descrição
<code>src</code>	Matriz referente à imagem.
<code>dst</code>	Imagen de saída.

Parâmetro	Descrição
<code>fx</code>	Fator de escala horizontal.
<codefy< code=""></codefy<>	Fator de escala vertical.
<code>interpolation</code>	Método de interpolação.

Os parâmetros `fx` e `fy` são valores numéricos positivos. Estes definem o quanto ampliada ou reduzida será a imagem. Mantendo esses dois parâmetros com valor igual a 1.0, a imagem não sofrerá ajustes de escala. Dobrando os valores para 2.0, será gerada uma nova imagem ampliada, com o dobro da largura e altura.

O objetivo do código exemplificado é reduzir o tamanho da imagem `folha.jpeg`, mais especificamente, gerar uma idêntica com as dimensões reduzidas pela metade. Nesse caso, o valor 0.5 (metade de 1.0) será atribuído aos parâmetros `fx` e `fy`.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("folha.jpeg")

imagemModificada = cv2.resize(
    imagemOriginal, None, fx = 0.5, fy = 0.5,
    interpolation = cv2.INTER_CUBIC
)
cv2.imshow("Resultado", imagemModificada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

No código anterior, observe que, no último parâmetro da função `resize`, é definida a função de interpolação. Existem diversos algoritmos para reduzir ou ampliar uma imagem. Esse

parâmetro nos permite definir qual desejamos utilizar.

A biblioteca OpenCV nos fornece 5 opções de funções de interpolação e, geralmente, a mais utilizada é a `cv2.INTER_CUBIC`. Neste livro, não será necessário detalhar cada uma delas, uma vez que todas desempenham a mesma tarefa. Se você quiser ver mais informações sobre cada função de interpolação, pode consultar a documentação oficial da biblioteca OpenCV.

A figura seguinte ilustra a execução do código exemplificado. A imagem à esquerda apresenta a fotografia carregada do arquivo `folha.jpeg` e, na imagem à direita, vemos a nova fotografia gerada após o processo de ajuste de escala.



Figura 5.12: Operação de ajuste de escala

Muitas vezes desejamos trabalhar com a vista superior do objeto fotografado, entretanto, a câmera que faz a captura das

imagens pode não estar corretamente posicionada. Em casos como esse, faz-se necessário um procedimento conhecido como ajuste de perspectiva. No tópico seguinte, vamos apresentá-lo e exemplificá-lo.

Ajuste de perspectiva

O posicionamento incorreto, a lente ou até mesmo o balanço da câmera podem interferir na perspectiva da fotografia, causando inclinação ou distorção do objeto representado na imagem. Em fotografias que possuem linhas horizontais, verticais ou formas geométricas, as distorções tornam-se ainda mais perceptíveis.

Para corrigir as distorções de perspectiva, podemos usar a função `warpPerspective` da biblioteca OpenCV. Ela ajusta a perspectiva de uma imagem tendo como referência uma matriz predefinida de pontos, gerada pela função `getPerspectiveTransform`.

A fim de facilitar o entendimento, observe a figura a seguir: a imagem à esquerda apresenta uma fotografia com distorções. Nela, as arestas do quadrado que representa o jogo foram definidas como pontos iniciais para o ajuste. A fotografia à direita apresenta a nova imagem gerada após o ajuste ser processado. Repare que os pontos iniciais, definidos à esquerda, tornaram-se as arestas da nova imagem gerada após o processamento.

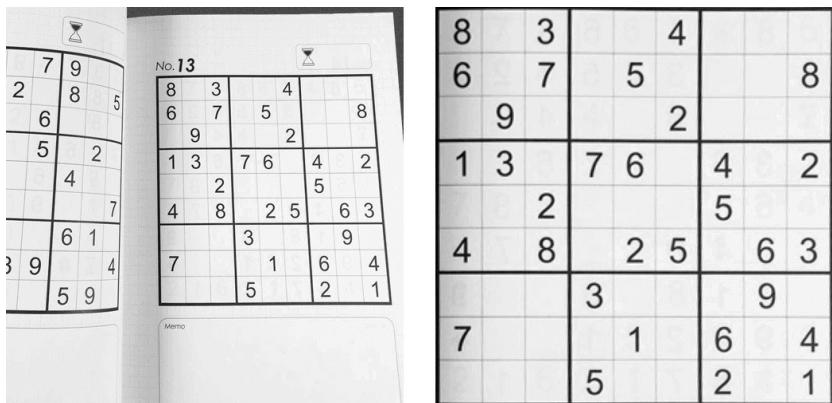


Figura 5.13: Operação de ajuste de perspectiva

O código exemplificado demonstra como as funções `getPerspectiveTransform` e `warpPerspective` podem ser usadas para ajustar a perspectiva de uma imagem:

```

import cv2
import numpy as np

imagemOriginal = cv2.imread("sudoku.jpeg")

pontosIniciais = np.float32(
    [[189,87], [459,84], [192,373], [484,372]])
)
pontosFinais = np.float32(
    [[0,0], [500,0], [0,500], [500,500]])
)
matriz = cv2.getPerspectiveTransform(
    pontosIniciais, pontosFinais
)
imagemModificada = cv2.warpPerspective(
    imagemOriginal, matriz, (500, 500)
)

cv2.imshow("Imagen Original", imagemOriginal)
cv2.imshow("Imagen Modificada", imagemModificada)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Para gerar a matriz de pontos iniciais e de pontos finais, usamos a função `float32` da biblioteca Numpy. Ela indica que os números armazenados nessa matriz são do tipo `float`. Além disso, eles apresentam 8 bits para representação do expoente e 23 bits para mantissa. Essa conversão é necessária para executar a função `getPerspectiveTransform`, que opera com valores numéricos nesse formato.

As duas matrizes geradas são os dois parâmetros necessários para que a função `getPerspectiveTransform` gere a matriz de ajuste de perspectiva. Essa nova matriz é o segundo parâmetro necessário para a execução da função `warpPerspective`, responsável por gerar uma nova imagem com o ajuste realizado.

Esta possui três parâmetros obrigatórios. O primeiro deles é a matriz referente à imagem original; o segundo, a matriz de ajuste de perspectiva; e o terceiro, as dimensões da nova imagem que será gerada. É importante estar atento ao fato de que as coordenadas dos pontos definidos na matriz de pontos finais devem respeitar os limites das dimensões definidas na função `warpPerspective`.

Além das operações geométricas, existem também algumas operações aritméticas que podem ser aplicadas a imagens. Vamos apresentá-las na próxima seção.

5.4 OPERAÇÕES ARITMÉTICAS

As operações aritméticas em imagens nos permitem adicionar o conteúdo de uma imagem em outra, ou até mesmo misturá-los. Nos tópicos seguintes, veremos algumas dessas operações bem como exemplos.

Geralmente, para que as operações de adição e mistura possam ser realizadas, é necessário que as imagens possuam a mesma dimensão (largura x altura) e tipo (8 bits, por exemplo).

Ao realizarmos operações aritméticas em imagens, devemos nos lembrar que a cor de cada pixel é representada em 8 bits (nas imagens em que estamos trabalhando) – ou seja, um número inteiro variando de 0 a 255. Dessa maneira, é importante estar atento à possibilidade de *overflow*. Isso significa que o valor 255 não pode ser ultrapassado nas operações.

Operação de adição

A operação de adição consiste em somar os valores dos pixels de uma imagem com outra(s), resultando em uma nova imagem. A função `add` da biblioteca OpenCV nos permite realizar rapidamente essa operação. Veja o código exemplificado e o resultado da sua execução:

```
import cv2

imagemFichasVermelhas = cv2.imread("fichas-vermelhas.bmp")
imagemFichasPretas = cv2.imread("fichas-pretas.bmp")

imagem = cv2.add(imagemFichasVermelhas, imagemFichasPretas)

cv2.imshow("Resultado", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante apresenta três imagens. A primeira delas, da esquerda para a direita, mostra o arquivo `fichas-vermelhas.bmp`, que exibe apenas fichas vermelhas. A segunda mostra o arquivo `fichas-pretas.bmp`, que exibe apenas as pretas. A última imagem apresenta o resultado da execução do código anterior, ou seja, a soma das duas primeiras através da função `add`.

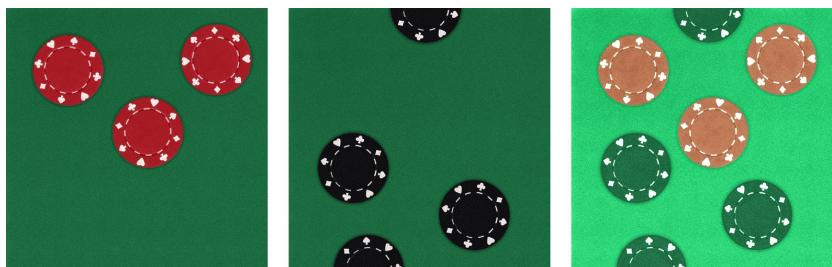


Figura 5.14: Operação de adição

Observe que a imagem resultante da soma da primeira com a segunda apresenta as fichas com ruído e sem muita nitidez. Isso ocorre porque as informações que representam o fundo da imagem, o pano da mesa, foram somadas aos pixels que representam as fichas.

A operação de adição também pode ser usada como um simples método de ajuste de contraste. Somando um valor positivo ou negativo, ao valor de cada pixel da imagem, é possível gerar uma imagem mais clara ou escura. Consequentemente, as operações de subtração, multiplicação e divisão também podem ser utilizadas para esse propósito.

O código exemplifica como o contraste de uma imagem pode ser alterado por meio da operação de adição:

```
import cv2
import numpy as np
from matplotlib import pyplot as grafico

imagemOriginal = cv2.imread("containers.jpg", 0)
imagemClara = cv2.add(imagemOriginal, 40)
imagemEscura = cv2.add(imagemOriginal, -40)

cv2.imshow("Imagen Original", imagemOriginal)
cv2.imshow("Imagen Clara", imagemClara)
cv2.imshow("Imagen Escura", imagemEscura)

grafico.hist(imagemOriginal.ravel(), 256, [0,256])

grafico.figure();
grafico.hist(imagemClara.ravel(), 256, [0,256])

grafico.figure();
grafico.hist(imagemEscura.ravel(), 256, [0,256])

grafico.show()

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante apresenta três imagens geradas a partir do código anterior. A primeira delas, da esquerda para a direita, exibe o arquivo original. A segunda apresenta a fotografia original após o valor 40 ser somado a todos os elementos que a representa. A última imagem apresenta o oposto, ou seja, a original com o valor -40 somado a todos os elementos que a representam.

Observe a diferença de contraste entre elas. A segunda imagem, com os valores incrementados em 40, apresenta maior contraste quando comparada às outras duas.

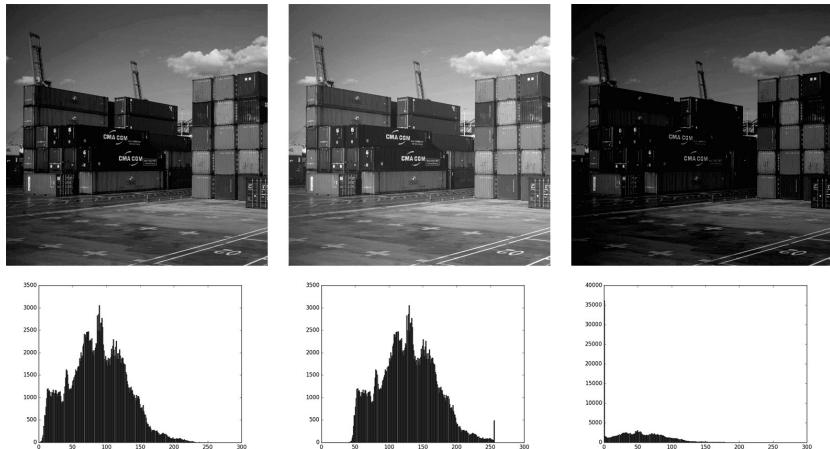


Figura 5.15: Operação de subtração

Nos histogramas apresentados, note que, após a execução da operação de soma, as novas imagens geradas possuem contraste diferentes da original.

Assim como é possível somar uma imagem a outra, ou somar valores aos pixels de uma determinada imagem, a subtração também é uma operação praticável. No tópico a seguir, veremos mais sobre a operação de subtração em imagens.

Operação de subtração

A subtração de uma imagem por outra nos fornece as diferenças entre as duas imagens. Com uma câmera fixa, ao fotografarmos uma sequência de imagens de um objeto em movimento, este método nos permite visualizar as mudanças ou os movimentos que ocorreram entre uma captura e outra.

A função da biblioteca OpenCV que nos permite fazer isso é a

`subtract` . O código seguinte exemplifica como utilizá-la.

```
import cv2

imagemFichaPosicao1 = cv2.imread("ficha-posicao-1.bmp")
imagemFichaPosicao2 = cv2.imread("ficha-posicao-2.bmp")

imagem = cv2.subtract(imagemFichaPosicao1, imagemFichaPosicao2)

cv2.imshow("Resultado", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante ilustra a execução desse código. As duas primeiras imagens, da esquerda para a direita, apresentam a posição inicial e final da ficha. A primeira imagem, carregada do arquivo `ficha-posicao-1.bmp`, mostra a posição inicial do objeto de interesse. A segunda, carregada do `ficha-posicao-2.bmp`, apresenta a posição final após a movimentação do objeto. Por fim, a terceira exibe o resultado da subtração da primeira com a segunda, gerada pelo código exemplificado.



Figura 5.16: Operação de subtração

Nessa figura, a última imagem ilustra o processamento que pode ser realizado a partir da terceira imagem. Ao convertermos a terceira imagem (resultado da operação de subtração) para uma imagem binária, obtemos uma imagem simplificada, que pode ser usada para obtermos a distância pela qual o objeto se movimentou.

Essa distância pode ser calculada pelo comprimento da reta que une o centro geométrico de cada círculo branco. Mais adiante, veremos detalhes sobre o procedimento de binarização de uma imagem, bem como um algoritmo para calcular a distância entre dois objetos.

No tópico a seguir, vamos conhecer uma operação para misturar informações entre duas imagens.

Operação de mistura

A operação de mistura consiste em mesclar, com perda de dados, as informações duas imagens em apenas uma. Em outras palavras, ela também realiza uma soma entre os valores dos pixels das imagens, entretanto, com pesos diferentes, mesclando-as e proporcionando um efeito de mistura ou transparência.

A função da biblioteca OpenCV que nos permite realizar essa operação é a `addWeighted`, que necessita obrigatoriamente de 5 parâmetros:

Parâmetro	Descrição
<code>src1</code>	Matriz referente à primeira imagem.
<code>alpha</code>	Intensidade da primeira imagem.
<code>src2</code>	Matriz referente à segunda imagem.
<code>beta</code>	Intensidade da segunda imagem.
<code>gamma</code>	Valor escalar adicionado a cada soma.

É válido lembrar que as imagens que serão misturadas precisam ter a mesma dimensão, tipo e número de canais. Outra informação importante é que a intensidade referente aos

parâmetros `alpha` e `beta` é um valor que varia entre 0.0 e 1.0 – sendo 0.0 a imagem completamente transparente e 1.0 com intensidade máxima.

O parâmetro `gamma` pode ser usado para realizar ajustes no brilho na imagem. Entretanto, no exemplo demonstrado a seguir, ele não será utilizado. O código exemplifica o procedimento de mistura entre duas imagens, sendo a primeira com parâmetro `alpha` igual a `0.2`, e a segunda, com o parâmetro `beta` igual a `1.0`.

```
import cv2

imagemFichasVermelhas = cv2.imread("fichas-vermelhas.bmp")
imagemFichasPretas = cv2.imread("fichas-pretas.bmp")

imagem = cv2.addWeighted(
    imagemFichasPretas, 0.2, imagemFichasVermelhas, 1.0, 0
)

cv2.imshow("Resultado", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura a seguir ilustra a execução desse código. A última imagem, da direita para a esquerda, apresenta a imagem gerada, ou seja, a mistura entre a primeira e a segunda.

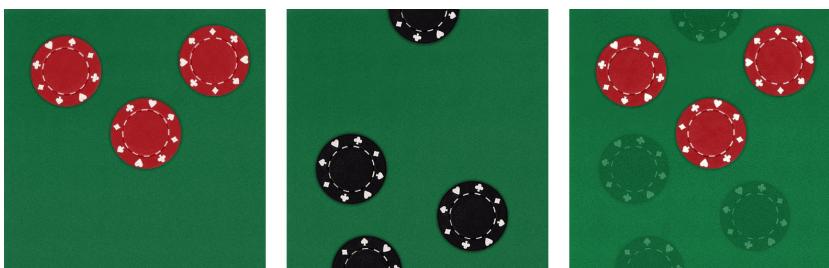


Figura 5.17: Operação de mistura

No próximo tópico, duas outras operações aritméticas serão sucintamente apresentadas, as operações de multiplicação e divisão.

Operações de multiplicação e divisão

A operação de multiplicação entre duas imagens pode ser efetuada pela função `multiply` da biblioteca OpenCV. Esta operação efetua o produto escalar entre os valores dos pixels de mesma posição nas duas imagens. Essa função pode ser usada também para multiplicar os valores dos pixels por um mesmo número inteiro positivo.

Uma outra operação que pode ser efetuada é a de divisão. Além de funcionar como um simples método para ajuste de contraste, ela pode ser usada para discriminar uma imagem da outra, ou seja, verificar se elas são idênticas ou não. Para que duas imagens sejam idênticas, quando realizada a divisão de uma pela outra, os valores que representam cada pixel devem ser iguais a 1.

Apesar de parecer uma boa solução, dividir uma imagem por outra exige mais processamento do que as subtrair. Sendo assim, não é um algoritmo muito eficiente para determinar se duas imagens são idênticas ou não.

A função da biblioteca OpenCV que nos permite dividir uma imagem por outra é a `divide`. Mais informações sobre esta e a função `multiply` podem ser encontradas na documentação oficial do OpenCV. As operações de multiplicação e divisão de imagens não são muito usadas e, por esse motivo, não serão exemplificadas aqui.

5.5 RUÍDO EM IMAGENS

Neste capítulo, foram apresentados diferentes métodos para tratamento de imagens. Os ajustes de perspectiva e escala, assim como as operações aritméticas e a equalização de histogramas, são procedimentos utilizados para realçar objetos de interesse em imagens. Em contrapartida, eles também podem, indesejavelmente, realçar ruídos nas imagens processadas.

Igualmente a qualquer outro sistema capaz de processar um sinal, o ruído também é o principal obstáculo para a eficiência do processamento de imagens. Ruídos são considerados como pequenas variações aleatórias, sofridas pelo sinal, que dificultam a leitura do valor real. No processo de captura e processamento de imagens, diversos fatores podem produzir ruídos. Os mais relevantes serão apresentados a seguir.

Ruído de captura

São considerados como ruído de captura as variações indesejadas provocadas por: poeira no ambiente, vibração da câmera, distorção da lente, iluminação inadequada e ruído elétrico no sensor.

Ruído de amostragem

Ocorre quando o sinal digital amostrado não corresponde à representação verdadeira da imagem analógica. As limitações na amostragem e na quantização de intensidade de cor provocam esse tipo de ruído.

Ruído de processamento

São gerados quando há limitações na precisão numérica. Ocorrem quando há overflow de números inteiros ou aproximações matemáticas de valores representados em ponto flutuante. Esse tipo de ruído é comum ocorrer quando filtros, para aguçar informações de contornos, são aplicados a imagens. Na figura seguinte, a fotografia à esquerda ilustra um ruído de processamento.

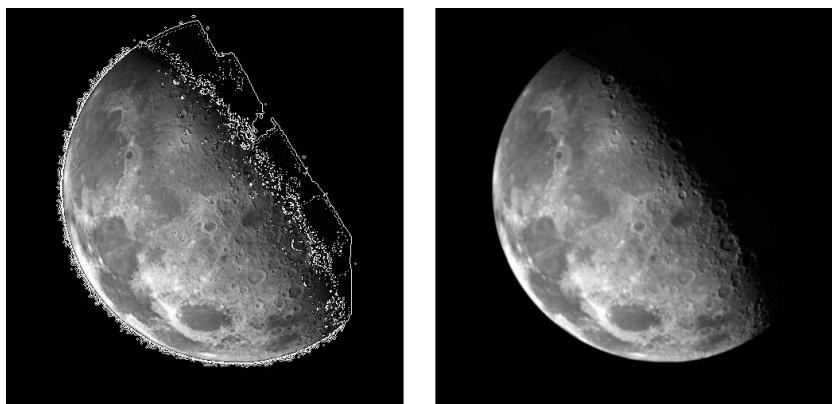


Figura 5.18: Ruído de processamento

Ruído de codificação de imagem

Este ruído é gerado por técnicas de compressão de imagens com perda. Por exemplo, a compactação de dados realizada pelo método JPEG gera ruído de codificação de imagem. A remoção de informações visuais que representam uma imagem, com objetivo de compactar o arquivo – mesmo quando não perceptíveis ao olho humano –, podem prejudicar o processamento da imagem.

Na figura a seguir, a imagem à direita apresenta a fotografia original sem compactação, e a figura à esquerda mostra a

fotografia com ruído de codificação após ser compactada em um arquivo JPEG.



Figura 5.19: Ruído de codificação de imagem

Oclusão de cena

Ocorre quando o objeto de interesse está obscurecido por outro. A vista parcial do objeto até pode fornecer algumas informações que tornam possível reconhecê-lo, entretanto, não será tão eficiente quanto uma imagem com sua representação completa. Na figura seguinte, a fotografia à esquerda ilustra uma imagem com ruído de oclusão de cena. Observe que um objeto dificulta a leitura da placa exibida na imagem.



Figura 5.20: Ruído de oclusão de cena

Ruído sal e pimenta

Também conhecido como ruído impulsivo, é caracterizado pela adição aleatória de pixels pretos e brancos, intensos ou fracos, na imagem. Não é muito comum de ocorrer em sensores modernos de captura de imagens. Veja na fotografia à esquerda uma imagem com ruído do tipo "sal e pimenta".



Figura 5.21: Ruído sal e pimenta

Ruído gaussiano

O ruído gaussiano, também conhecido como ruído aditivo, ocorre quando a variação aleatória do sinal segue a distribuição gaussiana. Em imagens digitais, esses ruídos são geralmente provocados por problemas de iluminação ou alta temperatura durante a aquisição.

Assim como os demais, ele pode ser reduzido ou suavizado utilizando técnicas de filtros espaciais. Algumas técnicas para redução de ruído serão apresentadas nos próximos capítulos. Na figura a seguir, a fotografia à esquerda apresenta uma imagem com ruído gaussiano.



Figura 5.22: Ruído gaussiano

Ruído em imagens binárias

Uma imagem gerada, a partir do procedimento de binarização, quase sempre apresenta ruídos que precisam ser eliminados. A figura ilustra esse processo em três imagens:

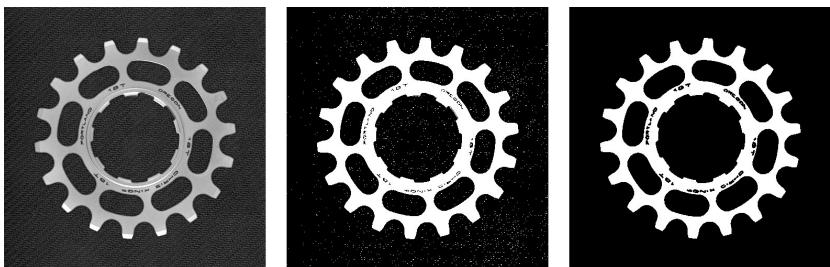


Figura 5.23: Tratamento de ruído

A primeira imagem apresentada, da esquerda para a direita, exibe o arquivo original em tons de cinza. A segunda apresenta o arquivo original convertido em uma imagem binária. Considerando a engrenagem como objeto de interesse a ser estudado, as informações referentes ao fundo da imagem podem ser descartadas.

Observe na segunda imagem que pixels brancos, que

representam informações irrelevantes sobre o plano de fundo, estão representados. Uma vez que eles não compõem o objeto de interesse, nem mesmo fornecem informações sobre ele, podemos considerá-los como ruídos na imagem.

Submetendo a imagem binarizada a técnicas de tratamento de ruído, a fim de eliminar esses pixels brancos, obtemos a terceira imagem apresentada na figura anterior. Nessa nova imagem gerada, após o procedimento de tratamento de ruído, apenas o objeto de interesse está representado. Observe que todas as informações referentes ao plano de fundo foram eliminadas.

Existem diferentes operações e filtros para tratamentos de ruído. Por esse motivo, esse assunto foi segmentado nos próximos capítulos deste livro. No capítulo a seguir, serão apresentados filtros espaciais para sua suavização.

CAPÍTULO 6

APLICAÇÃO DE FILTROS

Os filtros detalhados neste capítulo são conhecidos como filtros espaciais. São matrizes que percorrem toda a imagem, alterando os valores dos pixels, a fim de corrigir, suavizar ou realçar determinadas regiões. Essas matrizes, denominadas máscaras ou núcleos, atuam modificando os valores referentes ao nível de cinza de cada pixel da imagem.

Esse procedimento de deslocamento da máscara sobre a imagem que efetua as devidas operações é denominado convolução. A figura a seguir ilustra esse procedimento:

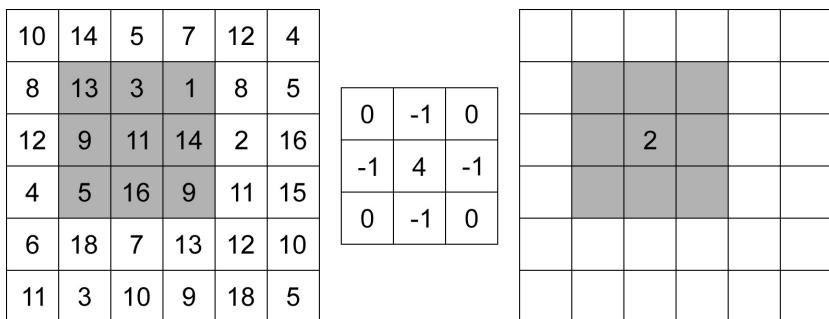


Figura 6.1: Operação de convolução

Na figura anterior, a primeira matriz, da esquerda para a direita, representa uma imagem de entrada em tons de cinza. A

segunda matriz representa a máscara, ou seja, o filtro que será aplicado à imagem de entrada. A terceira apresenta o resultado da convolução, no qual o pixel-alvo, posicionado na linha 3 e coluna 3, teve o seu valor alterado de 11 para 2.

O pixel-alvo é o qual será manipulado em um dado instante no processo de filtragem. O conjunto de pixels ao redor do pixel-alvo é denominado *vizinhança*.

Na operação de convolução, em cada iteração, a máscara sobrepõe um pixel-alvo da imagem submetida ao procedimento. Em cada uma dessas etapas, é realizado o produto ponto a ponto dos elementos da região sobreposta com os elementos da máscara de posição equivalente. Para facilitar o entendimento, acompanhe o esquema apresentado na figura a seguir, que representa uma das iterações da operação de convolução.

$$\begin{array}{|c|c|c|} \hline 13 & 3 & 1 \\ \hline 9 & 11 & 14 \\ \hline 5 & 16 & 9 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 0 & -1 & 0 \\ \hline -1 & 4 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & -3 & 0 \\ \hline -9 & 44 & -14 \\ \hline 0 & -16 & 0 \\ \hline \end{array}$$

Figura 6.2: Produto ponto a ponto

Na figura anterior, a primeira matriz, da esquerda para a direita, apresenta a região da imagem que será sobreposta pela máscara, representada pela segunda matriz. Nesta iteração, é realizado o produto ponto a ponto entre os elementos dessas duas matrizes, resultando em uma terceira. O somatório dos elementos da matriz resultante, ilustrada pela terceira matriz da figura anterior, retorna o novo valor do pixel-alvo que foi sobreposto.

O esquema apresentado a seguir ilustra o deslocamento da máscara sobre a imagem em três iterações consecutivas:

0	-1	0		0	-1	0		0	-1	0		
-1	4	-1	5	7	12	4	-1	4	-1	7	12	4
0	-1	0	3	1	8	5	0	-1	0	1	8	5
	12	9	11	14	2	16	12	9	11	14	2	16
	4	5	16	9	11	15	4	5	16	9	11	15
	6	18	7	13	12	10	6	18	7	13	12	10
	11	3	10	9	18	5	11	3	10	9	18	5

Figura 6.3: Operação de convolução

Na figura anterior, observe que, uma vez que parte da máscara sobrepõe pixels inexistentes, as bordas da imagem precisam receber um tratamento especial. Uma possível solução para este problema é assumir o valor zero para esses pixels. Outra solução é desconsiderar os pixels da borda, efetuando a operação somente nas posições onde a máscara se encaixa.

Os filtros podem ser classificados como lineares ou não lineares. A diferença entre eles é o tipo de operação realizada entre a máscara e o pixel-alvo da imagem. Os filtros lineares são mais comuns; eles usam máscaras que realizam somas ponderadas das intensidades dos pixels ao longo da imagem. Na prática, eles suavizam, realçam detalhes e minimizam efeitos de ruído, sem alterar o nível médio de cinza da imagem.

Já os filtros não lineares realizam somas não ponderadas. Na prática, realizam transformações que podem alterar o nível médio de cinza da imagem, por exemplo, destacar bordas, linhas e manchas.

Além da classificação quanto à linearidade, alguns filtros podem ser classificados como: passa-baixas ou passa-altas. Essa classificação refere-se ao tipo de frequência que será filtrada.

Por exemplo, um filtro passa-baixas atenua as altas frequências, ou seja, suaviza a imagem atenuando regiões que representam bordas ou contornos. Do contrário, o filtro passa-altas atenua as baixas frequências, realçando as regiões de bordas ou contornos.

Mas atenção, nem todos os filtros podem ser classificados segundo esse critério. Neste capítulo, serão apresentados filtros capazes de suavizar a imagem preservando a nitidez das bordas representadas.

Uma das principais aplicações dos filtros é o tratamento de ruído em imagens. Entretanto, medir e comparar o desempenho de cada um deles para esse propósito é uma tarefa um tanto quanto subjetiva, uma vez que o resultado desejado depende da informação que o observador deseja extrair da imagem.

Neste capítulo, quatro filtros diferentes para suavização de imagens serão apresentados e exemplificados. O primeiro que será detalhado é o filtro de média.

6.1 FILTRO DE MÉDIA

O filtro de média é um filtro linear, classificado como passa-baixas, ou seja, para suavização de imagens. Ele substitui cada pixel da imagem pelo valor médio de sua vizinhança. Quanto maior a ordem da matriz que representa a máscara, maior será o número de pixels vizinhos considerados, logo, mais intenso será o efeito de suavização.

O filtro de média pode ser aplicado pela função `blur` da biblioteca OpenCV, que requer apenas dois parâmetros. O primeiro refere-se à imagem que será submetida ao filtro, o segundo, à dimensão da máscara que será aplicada. O código seguinte exemplifica como realizar a aplicação do filtro de média:

```
import cv2

imgOriginal = cv2.imread("moedas.jpeg")
imgTratada = cv2.blur(imgOriginal, (5,5))

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Na figura adiante, a imagem à esquerda apresenta o arquivo original carregado; e, à direita, vemos o mesmo arquivo após a aplicação do filtro de média. Observe que o conteúdo de alta frequência, como as bordas mais nítidas da imagem, foi suprimido, e toda a imagem foi suavizada.



Figura 6.4: Aplicação do filtro de média

Apesar da sua importância didática e dos bons resultados obtidos para suavização de imagens, o filtro de média não é muito utilizado na prática. Outros filtros, assim como o gaussiano (que será apresentado no tópico a seguir), são mais eficazes.

6.2 FILTRO GAUSSIANO

Assim como o filtro de média, o gaussiano também é um filtro linear, passa-baixas. Este apresenta bons resultados no tratamento de imagens com ruído gaussiano. Sua aplicação com a biblioteca OpenCV pode ser realizada pelo método `GaussianBlur`.

Três parâmetros são necessários para execução desse método. O primeiro deles é a imagem a ser tratada; o segundo, a dimensão da matriz que representa a máscara de filtragem; e o último, o grau de suavização. É recomendado que a máscara seja uma matriz quadrada, com número ímpar de linhas e colunas. Quanto maior a dimensão da máscara, maior será o efeito do filtro sobre a imagem. O código seguinte exemplifica a aplicação do filtro gaussiano com o método `GaussianBlur`.

```
import cv2

imgOriginal = cv2.imread("moedas.jpeg")
imgTratada = cv2.GaussianBlur(imgOriginal, (5,5), 0)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Na figura, a imagem à esquerda apresenta o arquivo original carregado; à direita, o mesmo após a aplicação do filtro gaussiano. Observe que, assim como o resultado obtido com o filtro de média, o conteúdo de alta frequência foi suprimido, suavizando toda a imagem.



Figura 6.5: Aplicação do filtro gaussiano

Diferente do filtro de média, o filtro gaussiano possui um parâmetro que define o grau de suavização. Esse parâmetro, denominado σ (sigma) refere-se ao desvio padrão da função gaussiana. Na prática, quanto maior o valor desse parâmetro, mais intensa será a suavização. O valor de sigma é definido por um número inteiro positivo.

A figura a seguir demonstra como a variação do parâmetro de intensidade e a ordem da matriz que representa a máscara influenciam no efeito do filtro sobre a imagem.

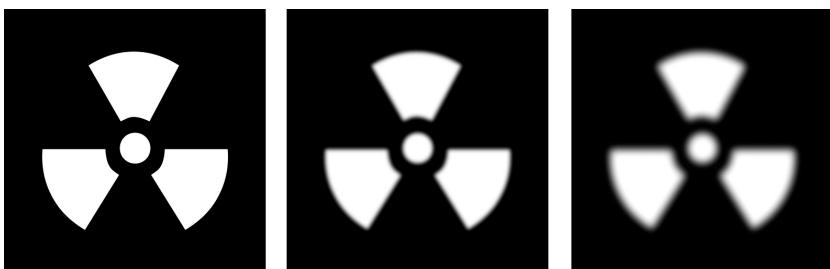


Figura 6.6: Diferentes intensidades do filtro gaussiano

A primeira imagem da figura, da esquerda para a direita, apresenta o arquivo original carregado. A segunda apresenta o arquivo original após ser submetido ao filtro gaussiano, com

máscara de ordem 21x21 e $\sigma = 4$. A última imagem demonstra a suavização desse mesmo arquivo, com máscara de ordem 41x41 e $\sigma = 8$.

A fim de demonstrar a eficiência do filtro gaussiano para o tratamento de imagens com ruído gaussiano, a figura seguinte apresenta duas imagens. A imagem à esquerda exibe uma fotografia com ruído gaussiano, e à direita, exibe essa mesma fotografia após o tratamento com o filtro.



Figura 6.7: Tratamento do ruído gaussiano

Observe na imagem à direita que o ruído gaussiano foi levemente atenuado após a aplicação do filtro gaussiano, com máscara de ordem 3x3 e sigma igual a 5.

6.3 FILTRO DE MEDIANA

O filtro de mediana é não linear e apresenta bons resultados no tratamento de ruído do tipo "sal e pimenta". Comparado ao gaussiano, o filtro de mediana é mais eficaz na preservação de

detalhes de alta frequência, como bordas ou contornos.

Assim como o filtro bilateral, que será detalhado na próxima seção, esse filtro não pode ser classificado quanto ao tipo de frequência filtrada. Ambos são classificados como *Edge-preserving smoothing*, ou seja, são técnicas de tratamento de imagem capazes de suavizá-la preservando bordas ou contornos.

Matematicamente, este filtro atua alterando o valor de cada pixel-alvo pela mediana estatística dos valores dos pixels vizinhos. O método `medianBlur` da biblioteca OpenCV nos permite aplicá-lo a imagens.

Apenas dois parâmetros são necessários para utilizá-lo. O primeiro deles é a imagem que receberá o tratamento, e o segundo é um valor inteiro, ímpar e positivo, que indicará a sua intensidade. O código seguinte exemplifica esse procedimento.

```
import cv2

imgOriginal = cv2.imread("moedas.jpeg")
imgTratada = cv2.medianBlur(imgOriginal, 5)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Na figura, a imagem à esquerda apresenta o arquivo original carregado; à direita, o mesmo após a aplicação do filtro de mediana. Compare visualmente o resultado obtido com o filtro gaussiano com o resultado obtido pelo de mediana. As regiões de contorno após a aplicação de mediana são menos atenuadas quando comparadas ao resultado obtido com o filtro gaussiano.



Figura 6.8: Aplicação do filtro de mediana

Para demonstrar a eficácia do filtro de mediana, no tratamento de imagens com ruído do tipo sal e pimenta, analise a figura adiante. A imagem à esquerda ilustra o arquivo original com ruído; já a imagem à direita, o mesmo arquivo após a aplicação do filtro. O mesmo código exemplificado foi utilizado para gerar a imagem tratada, apenas foi alterado o arquivo carregado.



Figura 6.9: Tratamento de ruído "sal e pimenta"

Aplicando o filtro de mediana na imagem com o símbolo da radioatividade, a mesma utilizada no exemplo do filtro gaussiano, é possível notar como esse filtro preserva os detalhes de contorno.

Na figura adiante, repare que a imagem tratada por esse filtro, diferente da tratada com o gaussiano, não apresenta bordas sem nitidez – apesar de deformá-las quando aplicado com muita intensidade.

A primeira imagem da figura, da esquerda para a direita, ilustra o arquivo original; na segunda, vemos esse mesmo arquivo após a aplicação do filtro de mediana com intensidade 5. Já a terceira imagem, temos a intensidade 15. Repare nessa terceira, que, mesmo as bordas estando nítidas, as arestas do objeto de interesse sofreram distorção, foram arredondadas.

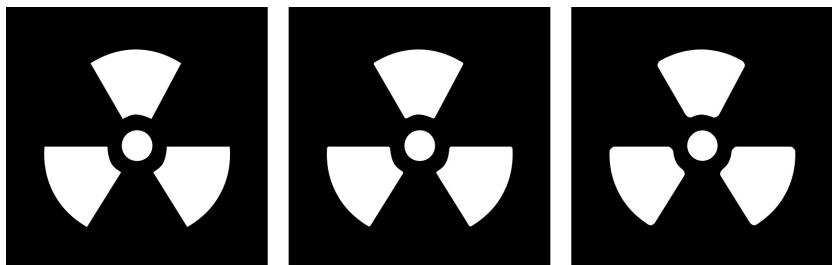


Figura 6.10: Diferentes intensidades do filtro de mediana

Na próxima seção, será apresentado outro filtro não linear, o bilateral. Comparado ao filtro de mediana, ele apresenta resultados ainda melhores quando o objetivo é preservar bordas e contornos.

6.4 FILTRO BILATERAL

De todos os filtros apresentados até então, o filtro bilateral é o mais indicado quando o objetivo é suavizar a imagem preservando os detalhes de bordas e contornos. A convolução desse filtro é similar a do gaussiano, ou seja, o processo de suavização é muito

semelhante a de um filtro passa-baixas, entretanto, modificado para preservar os detalhes de contorno.

O filtro bilateral pode ser aplicado pelo método `bilateralFilter` da biblioteca OpenCV. Para executar esse método, quatro parâmetros são requeridos. O primeiro deles é a imagem que receberá o tratamento, e o segundo indica o tamanho do filtro. Quanto maior o valor do segundo parâmetro, mais lenta será a execução desse método, não sendo recomendado trabalhar com valores superiores a 5 em processamento de imagens em tempo real.

Geralmente, os mesmos valores são utilizados no terceiro e no quarto parâmetro. O terceiro é conhecido como σ_{Color} (sigma color), e o quarto como σ_{Space} (sigma space). Na prática, quando menores que 10, esses valores não apresentam resultados significativos no tratamento de ruído e suavização.

Entretanto, quando maiores que 150, podem provocar um tratamento muito intenso, fazendo com que a imagem perca detalhes importantes. Quanto maior o valor do parâmetro sigma color, maior será a mistura das cores vizinhas, e quanto maior o valor do parâmetro sigma space, maior será a influência do filtro nos pixels vizinhos – desde que suas cores sejam próximas.

Os valores do segundo, terceiro e quarto parâmetro devem ser inteiros e positivos. O código apresentado exemplifica a execução do método `bilateralFilter` :

```
import cv2

imgOriginal = cv2.imread("moedas.jpeg")
imgTratada = cv2.bilateralFilter(imgOriginal, 9, 75, 75)
```

```
cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Na figura, a imagem à esquerda apresenta o arquivo original carregado; à direita, o mesmo após a aplicação do filtro bilateral. Observe como esse filtro suavizou detalhes do interior da moeda, demonstrando ser mais eficiente que o de mediana ao preservar os detalhes de bordas e contornos.



Figura 6.11: Aplicação do filtro bilateral

Outro exemplo da aplicação do filtro bilateral é apresentado na imagem adiante. A imagem à direita apresenta a fotografia original do estacionamento; à esquerda, a mesma fotografia após ser tratada com um filtro bilateral. Observe a redução no ruído preservando a nitidez das bordas e contornos.



Figura 6.12: Tratamento de ruído com filtro bilateral

No próximo capítulo, serão apresentadas técnicas para realçar bordas em imagens.

CAPÍTULO 7

REALCE DE BORDAS

No capítulo anterior, vimos filtros para tratamento de ruído em imagens. Porém, o procedimento de filtragem em imagens possui duas outras aplicações tão importantes quanto o tratamento de ruído: extração de características, e realce de arestas, bordas ou contornos em imagens.

Neste capítulo, serão apresentadas três técnicas para extrair informações de alta frequência (contornos) e duas para realçá-las. A primeira delas é o operador de Sobel.

7.1 OPERADOR DE SOBEL

O operador de Sobel, também nomeado de filtro de Sobel, é uma operação usada para realçar contornos em imagens. Esse filtro não linear realça linhas verticais e horizontais mais escuras que o fundo, sem realçar pontos isolados.

Além dessa característica, ele possibilita realçar as arestas independente da direção: a filtragem pode ser realizada tanto vertical quanto horizontalmente. As regiões destacadas por esse procedimento resultam em bordas mais "grossas" comparadas ao resultado obtido com outras técnicas. Por isso, o operador de Sobel não é tão utilizado para realçar bordas em imagens, sendo mais

comum usá-lo para detecção de bordas. Mesmo assim, esse operador continua sendo de grande importância, pois outros filtros para realçar imagens utilizam-no como base.

O filtro de Sobel pode ser aplicado pela função `Sobel` da biblioteca OpenCV. Ele requer cinco parâmetros obrigatórios: o primeiro é a imagem que receberá o tratamento; o segundo refere-se ao tipo de variável que armazenará o valor que representa o pixel. Quando o valor do segundo parâmetro é definido como `cv2.CV_8U`, os pixels são tratados como valores inteiros, entre 0 a 255, e armazenados em 8 bytes. O terceiro e quarto parâmetro definem como o realce será aplicado. Quando o terceiro parâmetro é igual 1 e o quarto igual a 0, o realce ocorrerá na direção horizontal; do contrário, na direção vertical. O quinto, e último, parâmetro é o tamanho da máscara de filtragem. Seu valor deve ser um número inteiro, ímpar, preferencialmente 1, 3, 5 ou 7.

O código exemplificado demonstra como usar a função `Sobel`. Ao final da execução, duas novas imagens serão geradas, uma com realce aplicado horizontalmente (`sobelx`), e outra verticalmente (`sobely`):

```
import cv2

imagem = cv2.imread("estacionamento.jpeg", 0)

sobelx = cv2.Sobel(imagem, cv2.CV_8U, 1, 0, ksize = 3)
sobely = cv2.Sobel(imagem, cv2.CV_8U, 0, 1, ksize = 3)

cv2.imshow("Original", imagem)
cv2.imshow("Sobel X", sobelx)
cv2.imshow("Sobel Y", sobely)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante ilustra a execução desse código. A primeira imagem, da esquerda para a direita, apresenta o arquivo original carregado; a segunda, o mesmo arquivo após receber o tratamento com o operador de Sobel no eixo X; e a terceira, no eixo Y.



Figura 7.1: Aplicação do operador de Sobel

Na próxima seção, veremos o operador laplaciano, um outro filtro para realçar bordas em imagens, que, comparado ao operador de Sobel, é mais eficiente para realçar contornos. Ambos possuem alta sensibilidade a ruídos e, por esse motivo, é comum a aplicação de um filtro passa-baixas na imagem antes de utilizá-los. Esse procedimento evita que bordas indesejadas, provocadas por ruídos, sejam detectadas.

7.2 OPERADOR LAPLACIANO

O operador laplaciano é um dos mais populares e usados para realçar bordas. Assim como o operador de Sobel, o laplaciano também é um filtro espacial. Ele possui uma máscara de ordem 3, que percorre toda a imagem alterando o pixel-alvo pela média ponderada dos pixels vizinhos, e depois eleva ao quadrado o valor obtido.

O operador laplaciano tende a produzir bordas mais finas comparado ao filtro de Sobel, entretanto, também é muito sensível

a ruído. Ele pode ser aplicado por meio da função `Laplacian` da biblioteca OpenCV.

Essa função requer obrigatoriamente apenas dois parâmetros: o primeiro é a imagem que receberá o tratamento, o segundo, assim como no método `Sobel`, refere-se ao tipo de variável que armazenará o valor que representa o pixel. O código exemplificado demonstra como utilizar o método `Laplacian`:

```
import cv2

imgOriginal = cv2.imread("estacionamento.jpeg", 0)
imgTratada = cv2.Laplacian(imgOriginal, cv2.CV_8U)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante ilustra a execução desse código. A primeira imagem, da esquerda para a direita, apresenta o arquivo original carregado; e a segunda, o mesmo arquivo após receber o tratamento com o operador laplaciano. Repare que, mesmo apresentando um desempenho superior ao filtro de Sobel, os ruídos ainda comprometem a qualidade da imagem.



Figura 7.2: Aplicação do filtro laplaciano

Para solucionar o problema com ruídos, um filtro passa-baixas, como o gaussiano, pode ser aplicado antes de a imagem ser submetida ao filtro laplaciano. Esse procedimento é tão eficaz que um filtro conhecido como laplaciano de gaussiano foi implementado a fim de aprimorar essa técnica conjunta.

O filtro laplaciano de gaussiano unifica essa ideia aplicando uma única máscara e reduzindo consideravelmente o número de operações. Infelizmente, a biblioteca OpenCV não possui uma função específica para o filtro laplaciano de gaussiano. Consequentemente, para obtermos o resultado proporcionado por ele, os filtros gaussiano e laplaciano devem ser aplicados individualmente.

No tópico a seguir, será apresentado como é possível aguçar bordas de uma imagem pelo laplaciano, sem que suas características originais sejam comprometidas.

Aguçamento de bordas

O aguçamento de bordas de uma imagem pode ser realizado por meio da subtração da imagem original com o laplaciano dessa mesma imagem. Essa operação realça artificialmente detalhes finos da imagem, entretanto, pode gerar pixels com valores negativos.

Para contornar essa situação, uma solução é utilizar o método `subtract` da biblioteca OpenCV. Dessa forma, durante o procedimento de subtração, os pixels que apresentarem valores inferiores a zero serão tratados automaticamente por esse método.

A figura a seguir ilustra a operação de aguçamento de bordas através da subtração dessas duas imagens. A primeira fotografia, da

esquerda para a direita, é o arquivo original carregado; a segunda apresenta o arquivo original após o tratamento com o filtro laplaciano. A última imagem apresentada é a imagem original aguçada, resultado da subtração da imagem original pela gerada pelo filtro laplaciano.

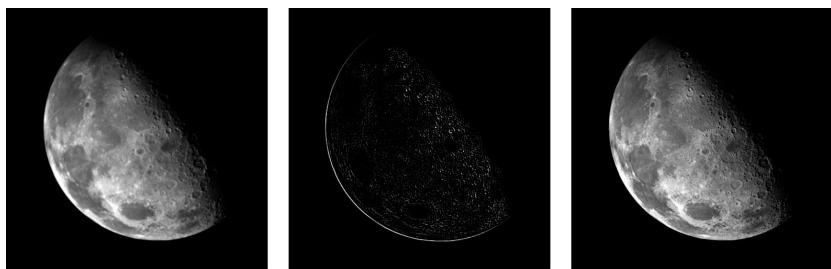


Figura 7.3: Aguçamento de bordas laplaciano

O código exemplifica esse procedimento:

```
import cv2

imgOriginal = cv2.imread("lua.jpeg", 0)
imgFiltrada = cv2.Laplacian(imgOriginal, cv2.CV_8U)
imgRealcada = cv2.subtract(imgOriginal, imgFiltrada)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Filtrada", imgFiltrada)
cv2.imshow("Realcada", imgRealcada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Para facilitar a comparação entre a imagem original e a realçada, a figura a seguir apresenta-as lado a lado. A figura à esquerda é o arquivo original carregado.



Figura 7.4: Lua com bordas aguçadas

Na próxima seção, será apresentado o filtro máscara de desaguçamento, uma alternativa ao filtro laplaciano.

7.3 FILTRO MÁSCARA DE DESAGUÇAMENTO

Uma outra técnica para realçar bordas em imagens é conhecida como filtro máscara de desaguçamento, ou filtro de realce. Este tem como base a subtração da imagem original por uma versão suavizada dela mesma. A imagem resultante desse procedimento contém as características de contornos da original, ou seja, as informações de alta frequência.

A suavização da imagem original pode ser realizada por um filtro passa-baixas, por exemplo, o filtro gaussiano. Por último, a original é somada com a que contém as características de alta frequência. O resultado dessa soma gera uma nova imagem com bordas realçadas. As equações a seguir sintetizam esse procedimento.

```
imgDetalhes = imgOriginal - imgSuavizada  
imgRealçada = imgOriginal + imgDetalhes
```

Na figura adiante, a primeira imagem da esquerda para a direita apresenta o arquivo original carregado. A segunda imagem apresenta o arquivo original após ser suavizado pelo filtro gaussiano. A terceira é o resultado da subtração entre a imagem original e a suavizada. Observe que somente as informações de alta frequência estão representadas.



Figura 7.5: Operação de subtração

A figura adiante exemplifica o procedimento de soma, utilizado para aguçar as informações de alta frequência da imagem original. A primeira imagem, da esquerda para a direita, apresenta o arquivo original, e a segunda contém as informações de alta frequência referentes às bordas. Por fim, a terceira exibe o resultado da soma da primeira com a segunda. Observe que as características de borda foram realçadas quando comparadas à imagem original.



Figura 7.6: Operação de soma

O código mostra esse procedimento:

```
import cv2

imgOriginal = cv2.imread("radiografia.jpeg", 0)
imgSuavizada = cv2.GaussianBlur(imgOriginal, (13,13), 3)
imgDetalhes = 3 * cv2.subtract(imgOriginal, imgSuavizada)
imgRealcada = cv2.add(imgOriginal, imgDetalhes)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgSuavizada)
cv2.imshow("Bordas", imgDetalhes)
cv2.imshow("Realcada", imgRealcada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Nesse código, com objetivo de aguçar ainda mais as características de alta frequência, a matriz que representa essas informações foi multiplicada pelo escalar 3. A figura apresenta o resultado desse processamento:



Figura 7.7: Radiografia realçada

Na figura anterior, a primeira imagem, da esquerda para a direita, exibe o arquivo original e, na segunda, vemos a nova imagem gerada após a aplicação do filtro de máscara de desaguçamento. Observe que a fotografia à direita apresenta as bordas do objeto de interesse mais nítidas e realçadas.

Na próxima seção, será apresentado o detector de bordas de Canny, um algoritmo ainda mais eficiente para detectar e realçar bordas em imagens.

7.4 DETECTOR DE BORDAS DE CANNY

O detector de bordas de Canny é um dos mais eficientes algoritmos para detectar bordas em imagens. Esse método foi desenvolvido por John Canny, que definiu que um bom detector de bordas deveria respeitar três características principais.

1. O algoritmo deve ser capaz de identificar todas as bordas possíveis;

2. Todas as bordas detectadas devem estar próximas das bordas originais da imagem;
3. Bordas falsas não podem ser criadas, ou seja, cada borda deve ser definida uma única vez.

Esse algoritmo, além de utilizar o filtro gaussiano, também faz uso do operador de Sobel.

Nesta seção, vamos exemplificar sua aplicação com a função Canny da biblioteca OpenCV. Ela requer três parâmetros obrigatórios. O primeiro deles é a imagem em tons de cinza que receberá o tratamento. O segundo e o terceiro estão relacionados à intensidade de detecção.

Na prática, quanto maior forem os valores do segundo e do terceiro parâmetros, menos bordas serão detectadas. O código a seguir exemplifica a aplicação desse filtro:

```
import cv2

imgOriginal = cv2.imread("estacionamento.jpeg", 0)
imgTratada = cv2.Canny(imgOriginal, 100, 200)

cv2.imshow("Original", imgOriginal)
cv2.imshow("Tratada", imgTratada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura seguinte apresenta o resultado da execução do código anterior. A imagem da esquerda exibe o arquivo original carregado e, na da direita, vemos o resultado após aplicação do filtro de Canny. Observe na imagem à direita as bordas detectadas na imagem original.



Figura 7.8: Aplicação do filtro de Canny

O próximo capítulo trata sobre operações morfológicas que podem ser utilizadas para realçar ou corrigir os objetos de interesse representados na imagem.

CAPÍTULO 8

OPERAÇÕES MORFOLÓGICAS

No campo de estudo sobre processamento de imagens, as operações morfológicas são operações que modificam o formato ou a estrutura dos objetos representados em uma imagem. Essas operações podem ser aplicadas a qualquer tipo de imagem, entretanto, são frequentemente utilizadas em imagens binárias, com a finalidade de realçar o objeto de interesse, ou tratar ruídos provocados pelo processo de binarização.

Um exemplo no qual essas operações são aplicadas são os sistemas que realizam leitura de placa de veículo por imagem. É comum que as fotografias capturadas das placas não apresentem, de imediato, a nitidez necessária para que elas sejam lidas pelo sistema. O tratamento dessas imagens com operações morfológicas, para realçar os caracteres representados na fotografia, é uma solução recorrente para este problema.

A figura a seguir exemplifica este processo. A imagem à esquerda apresenta a fotografia original da placa do veículo, e à direita, a mesma fotografia após ser submetida ao procedimento de binarização. Observe as falhas existentes nos caracteres representados na imagem à direita; estas podem dificultar a leitura

da placa pelo sistema.



Figura 8.1: Procedimento de binarização

Considere que o sistema identifique como um único caractere regiões de pixels brancos, isoladas por pixels pretos. Nesse caso, considerando a imagem binária apresentada na figura anterior, mais que 8 caracteres seriam contabilizados, ou seja, uma informação incorreta sobre a placa do veículo.

Submetendo essa mesma imagem a um tratamento com operações morfológicas, é possível corrigir as falhas nos caracteres. A figura adiante apresenta, à esquerda, a imagem binária antes do tratamento; à direita, após o tratamento. Compare visualmente ambas e observe que, após o tratamento, os caracteres estão corretamente preenchidos, podendo ser considerados como regiões isoladas de pixels brancos.



Figura 8.2: Placa após tratamento como operações morfológicas

Neste capítulo, vamos apresentar diferentes operações morfológicas. Antes de detalhar cada uma delas, é importante

compreender o conceito de elemento estruturante, um elemento-chave para que essas operações sejam realizadas.

8.1 ELEMENTO ESTRUTURANTE

Esse elemento pode ser visto como uma imagem binária, menor que a imagem original, armazenado geralmente em uma matriz quadrada. Ele é a base para que qualquer operação morfológica seja executada.

Assim como os filtros espaciais, estudados nos capítulos anteriores, a matriz que representa o elemento estruturante percorrerá toda a imagem a ser tratada. Nesse processo, uma determinada operação morfológica será realizada pixel a pixel na imagem, alterando o valor de cada um deles para seguir o padrão definido por esse elemento.

A figura seguinte ilustra três desses padrões, ou seja, matrizes de elementos estruturantes comumente utilizados. São eles, respectivamente, retangular, elipse e cruz.

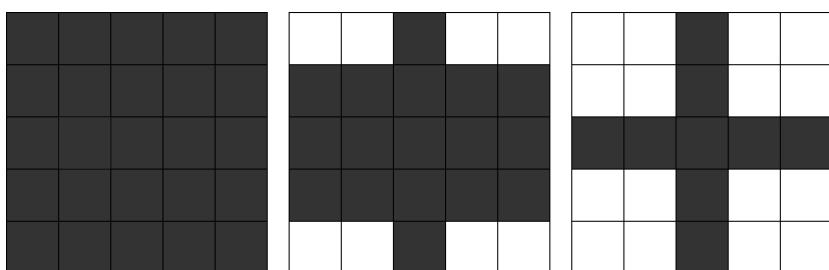


Figura 8.3: Representação dos elementos estruturantes

A biblioteca OpenCV possui esses três elementos estruturantes predefinidos, prontos para serem usados. O código exemplifica

como é possível obter a matriz de cada um:

```
# Retangular
```

```
>>> cv2.getStructuringElement(cv2.MORPH_RECT, (5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)
```

```
# Elíptico
```

```
>>> cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

```
# Cruz
```

```
>>> cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

Em muitos casos, quando o objetivo é realçar objetos de interesse com formato predominantemente circular, o elemento estruturante elíptico apresenta melhores resultados. Por outro lado, quando o objeto de interesse apresenta formato retangular, ou composto por mais retas que curvas, o elemento estruturante retangular é o mais indicado.

A fim de exemplificar o uso desses diferentes elementos estruturantes, a figura adiante apresenta um objeto submetido a uma operação morfológica para dilatá-lo. Essa operação foi aplicada a mesma imagem com diferentes elementos estruturantes,

possibilitando a comparação dos resultados obtidos em cada um deles.

Na primeira imagem, da esquerda para a direita, o arquivo original do objeto é exibido. A segunda exibe esse mesmo objeto após a operação de dilatação, em que um elemento estruturante elíptico foi utilizado. A terceira imagem apresenta o mesmo objeto submetido a mesma operação, entretanto, nesse processo, usamos um elemento estruturante retangular.

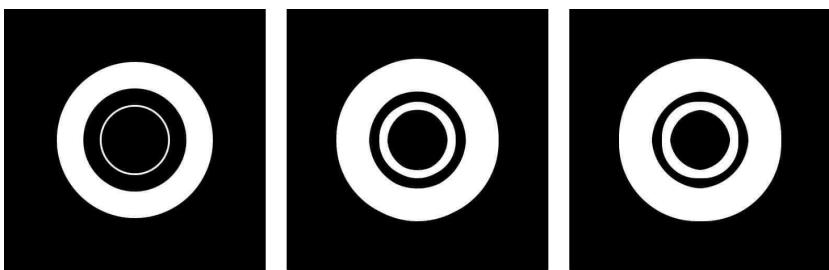


Figura 8.4: Aplicação de diferentes elementos estruturantes

A terceira imagem exibida nessa figura, quando comparada com a segunda, apresenta o objeto dilatado com deformações mais intensas no formato. Essa ocorrência demonstra a eficiência do elemento estruturante elíptico aplicado no processamento de imagens que representam objetos circulares.

Em casos especiais, quando nenhum dos elementos estruturantes predefinidos apresentam bons resultados, podemos criar um elemento estruturante personalizado. O código a seguir exemplifica esse procedimento. Observe que basta defini-lo em uma matriz binária.

```
import cv2
import numpy as np
```

```
elementoEstruturante = np.matrix([
    [0, 0, 1, 0, 0],
    [0, 1, 1, 1, 0],
    [1, 1, 1, 1, 1],
    [0, 1, 1, 1, 0],
    [0, 0, 1, 0, 0]
], np.uint8)
```

A seguir, veremos mais sobre as operações morfológicas de erosão e dilatação.

8.2 EROSÃO E DILATAÇÃO

Nesta seção, veremos as operações de erosão e dilatação em dois tópicos distintos. Essas duas operações primitivas são a base para todas as outras existentes. Consequentemente, as demais operações podem ser reduzidas a uma sequência de erosões e dilatações.

Operação de erosão

A operação de erosão é caracterizada pela corrosão das arestas do objeto de interesse, resultando em uma imagem "encolhida" do objeto. A figura a seguir ilustra a operação.

A primeira, da esquerda para a direita, apresenta uma imagem binária; na segunda, vemos a representação de um elemento estruturante retangular; e na terceira, o resultado do procedimento de erosão da primeira imagem com o elemento estruturante apresentado na segunda.

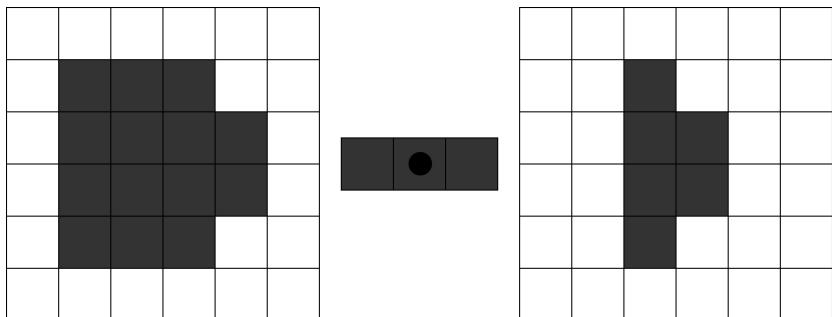


Figura 8.5: Operação de erosão

Na figura anterior, observe que a imagem de saída (terceira imagem) apresenta o objeto de interesse encolhido, com área reduzida ao representado na imagem de entrada (primeira imagem).

O algoritmo da operação de erosão consiste em sobrepor cada pixel da imagem de entrada com o centro do elemento estruturante. Se todos os pontos do elemento estruturante coincidirem com os pontos do objeto de interesse, então, esse ponto torna-se parte do objeto na imagem de saída. É importante ressaltar que, neste procedimento, a imagem de saída é inicializada apenas com o fundo, ou seja, sem nenhum objeto inicialmente representado.

A operação de erosão pode ser aplicada a uma imagem com a função `erode`, da biblioteca OpenCV. Esta requer três parâmetros fundamentais: a imagem a ser tratada, o elemento estruturante e o número de iterações desejadas.

No código exemplificado, utilizamos o elemento estruturante elíptico, definido em uma matriz quadrada de 5 linhas e 5 colunas. O tamanho dessa matriz e o número de iterações são definidos por

tentativa e erro, até que o resultado esperado seja obtido. Uma sugestão é realizar o primeiro teste com apenas uma iteração e uma matriz de ordem 3 e, caso o resultado não seja satisfatório, aumente gradualmente a ordem da matriz ou a quantidade de iterações.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("rolamento.bmp", 0)
elementoEstruturante = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (5,5)
)
imagemProcessada = cv2.erode(
    imagemOriginal, elementoEstruturante, iterations = 2
)

cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura seguinte apresenta o resultado da execução do código anterior. A imagem à direita ilustra o arquivo original carregado, e à esquerda, o resultado após as duas iterações da operação de erosão.

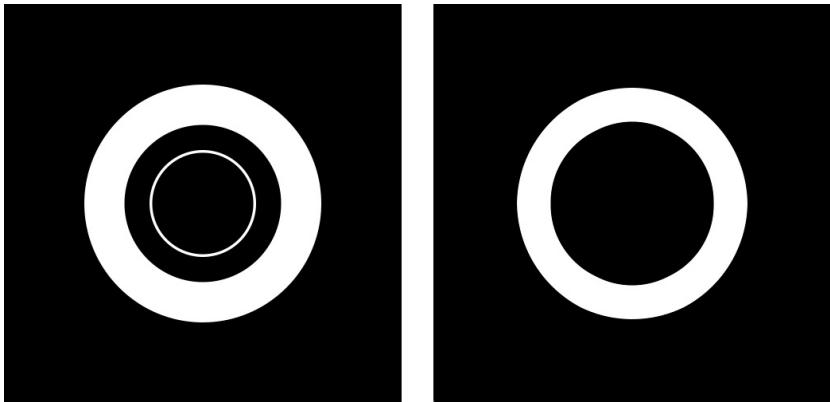


Figura 8.6: Operação de erosão

Observe na figura que a corrosão do objeto de interesse, além de reduzir a área do objeto, eliminou completamente o círculo representado em seu interior. Consequentemente, a operação de erosão apresenta bons resultados no tratamento de imagens ruidosas, justamente por corroer também os pixels que podem representar ruídos na imagem.

No próximo tópico, veremos a operação de dilatação.

Operação de dilatação

A operação de dilatação é o oposto da operação de erosão. Nesse procedimento, a área do objeto de interesse será dilatada, ou seja, o objeto do primeiro plano ficará maior do que era inicialmente. A figura a seguir ilustra esta operação.

A primeira imagem, da esquerda para a direita, apresenta uma imagem binária; a segunda, a representação de um elemento estruturante em forma de cruz; e a terceira, o resultado do procedimento de dilatação da primeira imagem com o elemento

estruturante apresentado na segunda imagem.

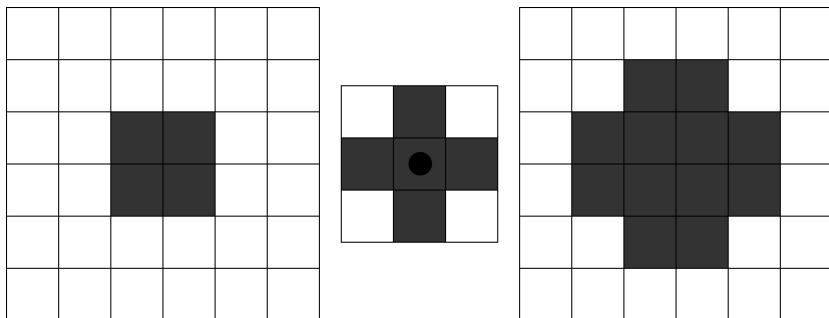


Figura 8.7: Operação de dilatação

Na figura anterior, observe que a imagem de saída apresenta o objeto de interesse dilatado, com área maior que a representada na imagem de entrada.

O algoritmo da operação de dilatação, assim como o de erosão, consiste em sobrepor cada pixel da imagem de entrada com o centro do elemento estruturante. Se todos os pontos do elemento estruturante coincidirem com os pontos do objeto de interesse, então, esse ponto torna-se parte do objeto na imagem de saída. Diferente da operação de erosão, nesta, a imagem de saída é inicializada como uma cópia da imagem de entrada.

A função `dilate`, da biblioteca OpenCV, possibilita a execução dessa operação. Ela requer os mesmos parâmetros apresentados na função `erode`. O código a seguir exemplifica a operação:

```
imagemOriginal = cv2.imread("rolamento.bmp", 0)
elementoEstruturante = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (5,5)
)
imagemProcessada = cv2.dilate(
```

```
    imagemOriginal, elementoEstruturante, iterations = 2
)
cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura apresenta o resultado da execução desse código:

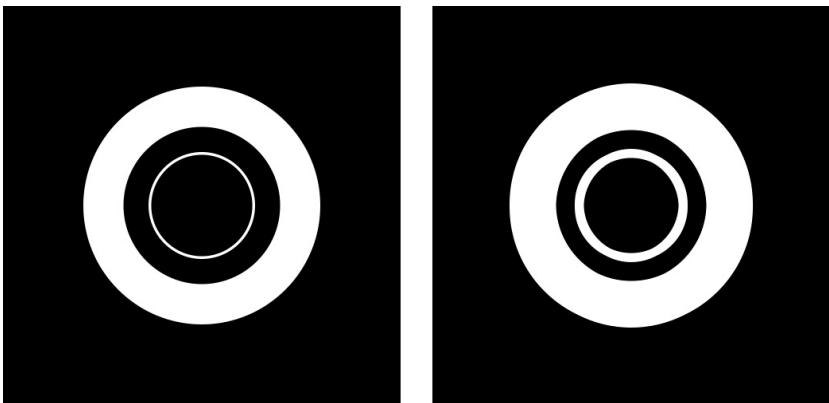


Figura 8.8: Operação de dilatação

Observe que a imagem à direita ilustra o arquivo original carregado, e à esquerda, o resultado após as duas iterações da operação de dilatação.

A operação de dilatação apresenta bons resultados quando desejamos realçar o objeto de interesse. Ela aplica-se perfeitamente ao exemplo apresentado no início deste capítulo.

Para corrigir a fotografia da placa do carro, que apresenta falhas nos caracteres, essa operação poderia ser uma das mais indicadas. A dilatação do objeto de interesse ajudaria a preencher as falhas presentes em seu interior.

Na próxima seção, serão detalhadas as operações de abertura e fechamento.

8.3 ABERTURA E FECHAMENTO

Operação de abertura

A operação de abertura é caracterizada pela operação de erosão seguida da operação de dilatação. Por esse motivo, assim como a operação de erosão, esta também é frequentemente usada para tratar ruídos.

A operação de erosão, quando aplicada com esse propósito, além de eliminar ruídos, corrói as arestas do objeto de interesse. A fim de recuperar as arestas corroídas, essa operação pode ser aplicada em sequência. O mesmo resultado obtido com a aplicação dessas duas operações também pode ser alcançado com a operação de abertura.

A função `morphologyEx` da biblioteca OpenCV nos permite executar a operação de abertura rapidamente. Para utilizar essa função, três parâmetros obrigatórios precisam ser informados. O primeiro deles é imagem a ser tratada; o segundo, a operação que será realizada; por último, o elemento estruturante.

O código apresentado exemplifica como aplicar a operação de abertura:

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("rolamento-ruido-externo.bmp", 0)

elementoEstruturante = cv2.getStructuringElement(
```

```

cv2.MORPH_ELLIPSE, (3,3)
)
imagemProcessada = cv2.morphologyEx(
    imagemOriginal, cv2.MORPH_OPEN, elementoEstruturante
)

cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

Observe que o segundo parâmetro da função `morphologyEx` foi definido como `MORPH_OPEN`, indicando a execução da operação de abertura. Essa operação, assim como as demais que serão apresentadas, não possui uma função específica para ser executada, justamente por se tratarem de operações que possuem a erosão e a dilatação como base. Vamos utilizar essa mesma função para executar as demais operações morfológicas.

A figura a seguir apresenta o resultado da execução do código anterior. A imagem à direita ilustra o arquivo original carregado; e à esquerda, o resultado após a operação de abertura ser realizada.

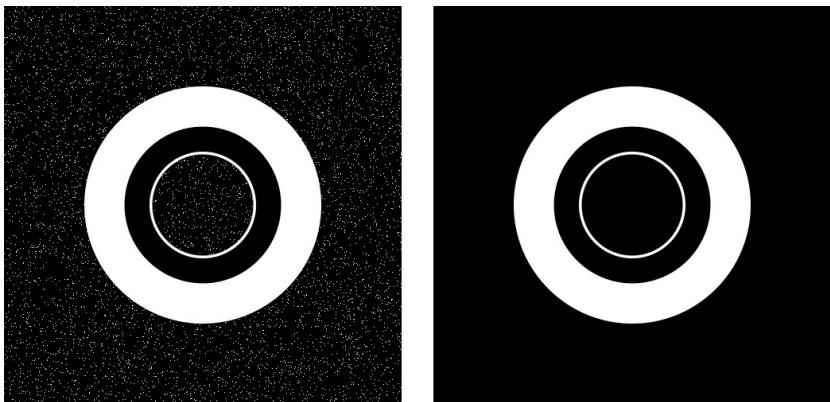


Figura 8.9: Operação de abertura

Observe na figura que o ruído do tipo sal e pimenta, presente no arquivo original carregado, foi completamente eliminado após a execução da operação de abertura.

No próximo tópico, será apresentada uma operação inversa a de abertura, conhecida como operação de fechamento.

Operação de fechamento

A operação de fechamento é usada para preencher a imagem, corrigindo pontos no objeto de interesse que foram danificados durante o processo de binarização. Essa operação consiste na operação de dilatação, seguida da operação de erosão – sequência oposta executada na de abertura.

Utilizando a função `morphologyEx` da biblioteca OpenCV, com o parâmetro `MORPH_CLOSE`, podemos aplicar a operação de fechamento a uma imagem. O código a seguir exemplifica esse procedimento:

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("rolamento-ruido-interno.bmp", 0)

elementoEstruturante = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (5,5)
)
imagemProcessada = cv2.morphologyEx(
    imagemOriginal, cv2.MORPH_CLOSE, elementoEstruturante
)

cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura seguinte apresenta o resultado dessa execução do código. A imagem à direita mostra o arquivo original carregado; e à esquerda, o resultado após a operação de fechamento ser realizada.

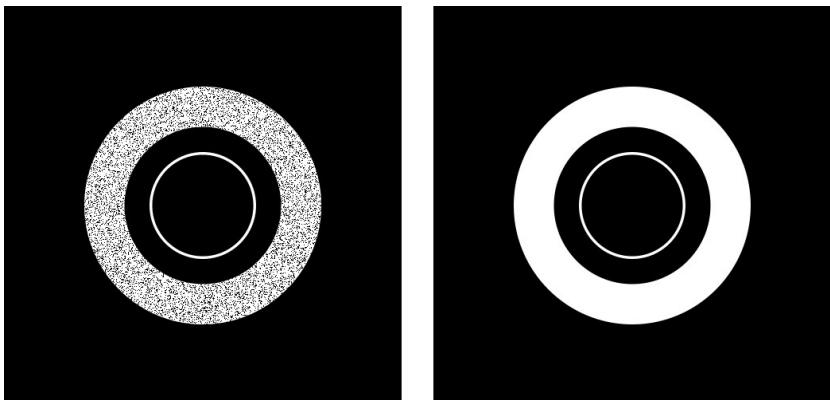


Figura 8.10: Operação de fechamento

Observe na figura que os pontos pretos, presentes no interior do objeto de interesse, foram completamente preenchidos após a execução dessa operação.

Abertura e fechamento em tons de cinza

Apesar de ser uma técnica pouco utilizada, as operações morfológicas podem ser aplicadas também a imagens em tons de cinza. O procedimento para realizar esse processamento é o mesmo, entretanto, em vez de carregar uma imagem binária, basta carregar uma em tons de cinza.

A operação de abertura, quando aplicada a imagens em tons de cinza, tende a suprimir pequenas regiões brilhantes nela. De maneira oposta, a operação de fechamento tende a suprimir

pequenas regiões escuras. Consequentemente, essas operações são geralmente usadas com intuito de uniformizar a iluminação dos objetos fotografados.



Figura 8.11: Abertura em imagens em tons de cinza

A primeira imagem, da esquerda para a direita, apresenta o arquivo original carregado. A segunda imagem exibe o resultado da operação de abertura na primeira imagem. Observe que a segunda representa informações sobre a iluminação da primeira.

Subtraindo a segunda imagem da primeira, obtemos a terceira imagem. Esta nova, obtida com a operação de subtração, apesar de mais escura que a original, apresenta os grãos uniformemente iluminados. Para que um resultado ainda melhor seja alcançado, basta submeter a terceira imagem a um procedimento de ajuste de contraste, por exemplo, somá-la com ela mesma.

Esse procedimento de soma resultará em uma imagem mais clara, capaz de representar o objeto de interesse com maior nitidez. O código seguinte mostra esse procedimento de correção de iluminação.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("arroz.bmp", 0)
```

```
elementoEstruturante = cv2.getStructuringElement(  
    cv2.MORPH_CROSS, (100, 100)  
)  
imagemProcessada = cv2.morphologyEx(  
    imagemOriginal, cv2.MORPH_OPEN, elementoEstruturante  
)  
  
imagemSubtraida = cv2.subtract(imagemOriginal, imagemProcessada)  
  
# Ajusta o contraste da imagem  
imagemTratada = cv2.add(imagemSubtraida, imagemSubtraida)  
  
cv2.imshow("Original", imagemOriginal)  
cv2.imshow("Resultado", imagemProcessada)  
cv2.imshow("Subtraida", imagemSubtraida)  
cv2.imshow("Tratada", imagemTratada)  
  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

A figura a seguir apresenta duas imagens geradas por esse código. A imagem à esquerda apresenta o arquivo original carregado. Na segunda, à direita, temos o resultado final do tratamento para corrigir a variação de iluminação na fotografia. Observe a diferença entre elas: a resultante desse procedimento representa os grãos mais nítidos, uniformemente iluminados.

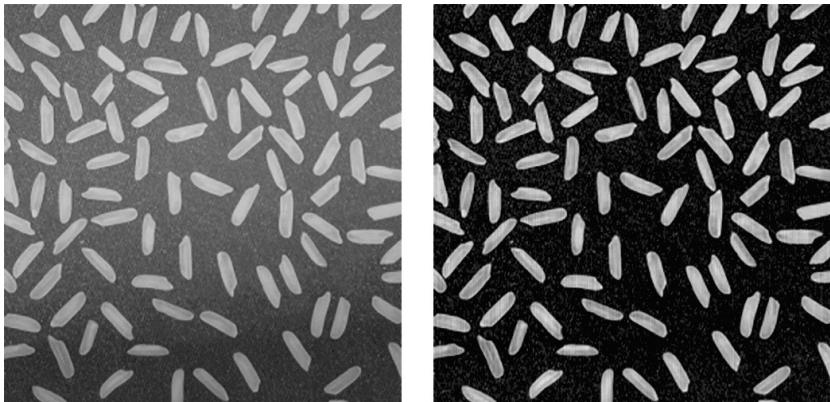


Figura 8.12: Abertura em imagens em tons de cinza

Na seção a seguir, veremos a operação de gradiente morfológico.

8.4 GRADIENTE MORFOLÓGICO

Essa operação é a diferença entre a operação de dilatação e erosão de uma imagem. O resultado da operação de gradiente morfológico é uma imagem que representa a borda do objeto de interesse. O código a seguir exemplifica sua execução, utilizando a função `morphologyEx` com o parâmetro `MORPH_GRADIENT`.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("rolamento.bmp", 0)

elementoEstruturante = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (3,3)
)
imagemProcessada = cv2.morphologyEx(
    imagemOriginal, cv2.MORPH_GRADIENT, elementoEstruturante
)
```

```
cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante apresenta as imagens obtidas com a execução desse código. A imagem à direita ilustra o arquivo original carregado, e à esquerda, o resultado após a operação de gradiente morfológico ser realizada.

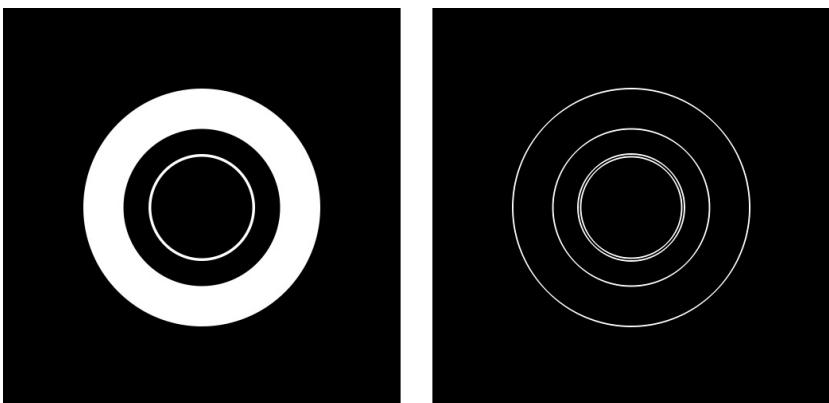


Figura 8.13: Operação de gradiente morfológico

Na seção a seguir, as operações de Top Hat e Black Hat serão resumidamente apresentadas.

8.5 TOP HAT

A operação de Top Hat consiste na subtração da versão morfológicamente aberta de uma imagem com a versão original. Essa operação pode ser efetuada com o parâmetro `MORPH_TOPHAT` na função `morphologyEx` da biblioteca OpenCV.

Frequentemente, a operação de Top Hat é aplicada a fim de realçar objetos brilhantes em fundos escuros. Além dessa utilidade, ela também é aplicada com intuito de corrigir a variação de luminosidade em imagens, possibilitando que objetos representados em regiões mais escuras sejam realçados.

A figura a seguir apresenta três imagens que ilustram esse procedimento. A primeira, da esquerda para a direita, apresenta o arquivo original carregado, exibindo grãos de arroz espalhados em uma superfície escura. A segunda imagem mostra o resultado da operação de Top Hat aplicada à primeira.

Observe que, apesar de mais escura, os grãos estão uniformemente iluminados. Já a terceira imagem apresenta a segunda após um procedimento de ajuste de contraste, com a finalidade de realçar os grãos representados.

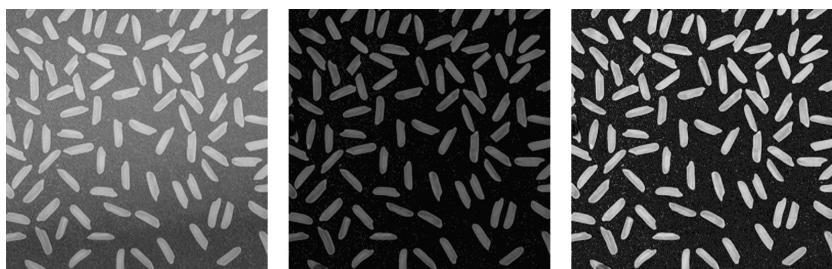


Figura 8.14: Operação de Top Hat para correção de luminosidade

O código a seguir mostra como a iluminação da imagem dos grãos, apresentada nessa figura, pode ser uniformizada através da operação de Top Hat.

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("arroz.bmp", 0)
```

```
elementoEstruturante = cv2.getStructuringElement(  
    cv2.MORPH_ELLIPSE, (25,25)  
)  
imagemProcessada = cv2.morphologyEx(  
    imagemOriginal, cv2.MORPH_TOPHAT, elementoEstruturante  
)  
  
# ajuste de contraste  
imagemTratada = cv2.add(imagemProcessada, imagemProcessada)  
  
cv2.imshow("Original", imagemOriginal)  
cv2.imshow("Resultado", imagemProcessada)  
cv2.imshow("Final", imagemTratada)  
  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

Nesse código, observe que usamos um elemento estruturante de ordem 25. Nesse exemplo, elementos estruturantes menores poderiam impossibilitar a visualização do objeto de interesse. Entretanto, a operação de Top Hat, aplicada a elementos estruturantes menores, pode ser utilizada para suprimir grandes regiões na imagem. A figura adiante exemplifica esse procedimento.

A imagem à esquerda apresenta uma fotografia da galáxia Andrômeda. Esta, quando submetida à operação de Top Hat com um elemento estruturante elíptico (representado em uma matriz de ordem 5), resulta na imagem à direita. Note que, após o processo, a galáxia foi completamente removida da fotografia.

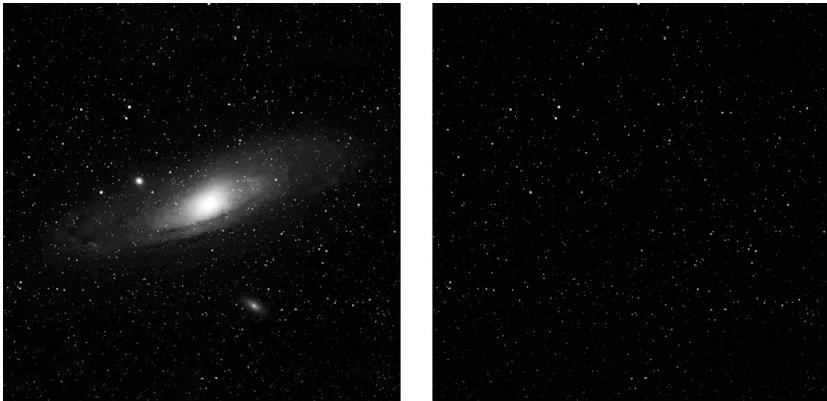


Figura 8.15: Operação de Top Hat para suprimir regiões da imagem

Além da operação de Top Hat, existe uma outra, pouco usada, conhecida como Black Hat (ou Bottom Hat). Esta consiste na subtração da imagem original por sua versão morfologicamente fechada. Ela pode ser aplicada com o parâmetro `MORPH_BLACKHAT` na função `morphologyEx` da biblioteca OpenCV.

Justamente por se tratar de uma operação que não é muito comum, ela não será exemplificada. Mais informações sobre a operação de Black Hat podem ser encontradas na documentação oficial da biblioteca OpenCV.

Na próxima seção, vamos conferir como podemos usar as operações morfológicas no tratamento de ruídos em imagens binárias.

8.6 TRATAMENTO DE RUÍDO

No final do capítulo anterior, foi apresentado um exemplo de como o procedimento de binarização pode gerar uma imagem com

ruídos. Nesta seção, veremos como os ruídos gerados por esse procedimento podem ser tratados com operações morfológicas. O código adiante exemplifica como é possível remover os ruídos da imagem binarizada da engrenagem.

Na figura adiante, observe que o ruído é caracterizado por pixels brancos, mesma cor que representa o objeto de interesse. Nesse caso, a operação de abertura, ou a de erosão, poderia proporcionar bons resultados. No código, utilizamos a operação de erosão com o elemento estruturante `MORPH_CROSS` :

```
import cv2
import numpy as np

imagemOriginal = cv2.imread("engrenagem-binaria.bmp", 0)

elementoEstruturante = cv2.getStructuringElement(
    cv2.MORPH_CROSS, (3,3)
)
imagemProcessada = cv2.erode(
    imagemOriginal, elementoEstruturante, iterations = 1
)

cv2.imshow("Original", imagemOriginal)
cv2.imshow("Resultado", imagemProcessada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura a seguir apresenta o resultado obtido na execução desse código. Observe que o ruído foi completamente removido da imagem. Nesse exemplo, a operação de erosão foi aplicada a uma única iteração, com um elemento estruturante em formato de cruz.

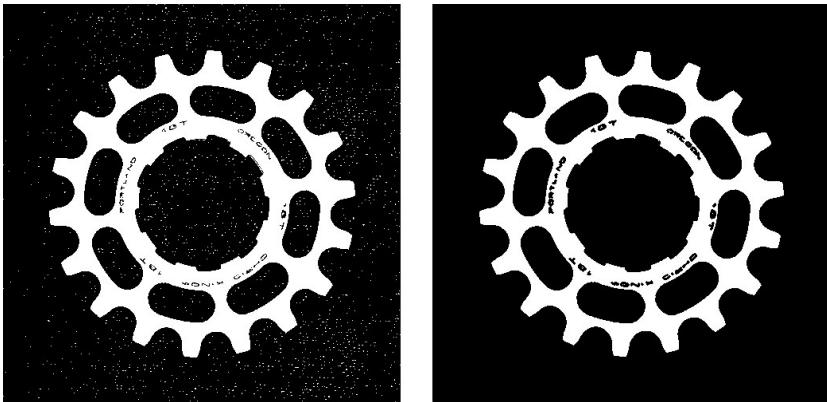


Figura 8.16: Tratamento de ruído em imagem binária

Outras operações, elementos estruturantes e número de iterações também poderiam fornecer bons resultados. Não existe uma regra para definir quais parâmetros devem ser usados. O ideal é experimentar diversas operações, ou sequências de operações, até alcançar o resultado esperado.

Neste capítulo, vimos as operações morfológicas e como elas podem ser utilizadas, tanto para o tratamento de ruído quanto para realçar objetos de interesse em imagens binárias. O próximo capítulo tratará sobre técnicas de segmentação de objetos em imagens. Nele, discutiremos como separar o objeto de interesse dos demais representados na imagem.

CAPÍTULO 9

SEGMENTAÇÃO DE OBJETOS

A segmentação de objetos em imagem é uma das principais etapas de um sistema baseado em Visão Computacional. Essa etapa consiste em separar somente a área que representa o objeto de interesse em uma nova imagem, excluindo também o segundo plano dessa região.

O objeto a ser estudado é considerado o primeiro plano da imagem. Os pixels que não fazem parte de sua representação são denominados como segundo plano. A segmentação é o primeiro passo para que possamos extrair características do objeto.

Somente com ele segmentado, é possível obter informações e detalhes que tornam possível classificá-lo. Neste capítulo, veremos quatro métodos para segmentação de objetos em imagens, e o primeiro deles será por binarização.

9.1 SEGMENTAÇÃO POR BINARIZAÇÃO

A segmentação por binarização também é conhecida como aplicação de limiar de intensidade. A separação do objeto de interesse por meio desse método ocorre pela definição do valor de

um limiar. Quando uma imagem é submetida a esse procedimento, os pixels representados por valores maiores que o limiar são estabelecidos como o objeto de interesse, sendo redefinidos para a cor preta ou branca. Do contrário, os pixels representados por valores menores que o limiar são estabelecidos como o segundo plano, sendo redefinidos para a cor oposta à do objeto de interesse.

A segmentação por binarização resulta em uma imagem binária, na qual geralmente o objeto de interesse é representado pela cor branca e o segundo plano pela preta. Esse procedimento de binarização, ou aplicação de limiar de intensidade, pode ser realizado pela função `threshold` da biblioteca OpenCV.

Essa função requer quatro parâmetros obrigatórios. O primeiro deles é a imagem em tons de cinza que receberá o tratamento, e o segundo é o valor do limiar, geralmente definido por tentativa e erro. O terceiro parâmetro define o valor da intensidade que receberá os pixels representados por valores superiores ao limiar. Por fim, quarto indica o método de aplicação do limiar.

A tabela a seguir apresenta os dois métodos mais utilizados:

Parâmetro	Descrição
<code>THRESH_BINARY</code>	Objeto de interesse na cor preta
<code>THRESH_BINARY_INV</code>	Objeto de interesse na cor branca

A diferença entre esses dois tipos é que o `THRESH_BINARY` segmenta o objeto de interesse na cor preta, logo, o segundo plano é representado em branco. Já o `THRESH_BINARY_INV` faz justamente o oposto: representa o objeto de interesse na cor branca, e o segundo plano, na cor preta.

O código a seguir exemplifica o procedimento de segmentação por binarização. Neste exemplo, grãos de café são segmentados dos demais elementos representados na imagem. O valor 135, obtido através de tentativa e erro, foi definido como limiar.

```
import cv2
import numpy as np

imgOriginal = cv2.imread("graos-de-cafe.jpeg", 0)

metodo = cv2.THRESH_BINARY_INV
ret, imgBinarizada = cv2.threshold(imgOriginal, 135, 255, metodo)

cv2.imshow("Imagen Original", imgOriginal)
cv2.imshow("Imagen Tratada", imgBinarizada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura seguinte ilustra a execução desse código. A imagem à esquerda apresenta o arquivo original carregado antes de ser convertido para tons de cinza, e à direita, a imagem após sofrer o procedimento de binarização, com os grãos de café segmentados. Observe que o uso do parâmetro `cv2.THRESH_BINARY_INV` definiu os objetos de interesse na cor branca e os demais elementos da imagem como plano de fundo, na cor preta.



Figura 9.1: Segmentação por binarização.

Nessa figura, note também que as áreas que representam os grãos de café estão falhadas, possuindo pontos pretos quando deveriam ser completamente brancas. Essas falhas são provocadas pelo procedimento de binarização. Elas ocorrem quando pixels pertencentes ao objeto de interesse são representados por um valor inferior ao limiar.

Falhas como estas são consideradas ruídos na imagem, entretanto, podem ser facilmente tratadas com operações morfológicas. A figura a seguir ilustra todo o processo.

A primeira imagem, da esquerda para a direita, apresenta a original convertida para tons de cinza; a segunda, a mesma após o procedimento de binarização; e por último, na terceira, vemos a segunda após o tratamento com operações morfológicas, com intuito de corrigir as falhas nos grãos.

Para corrigi-las, aplicamos a operação morfológica de fechamento e, em seguida, a de erosão – ambas com o mesmo elemento estruturante, retangular de ordem 3.

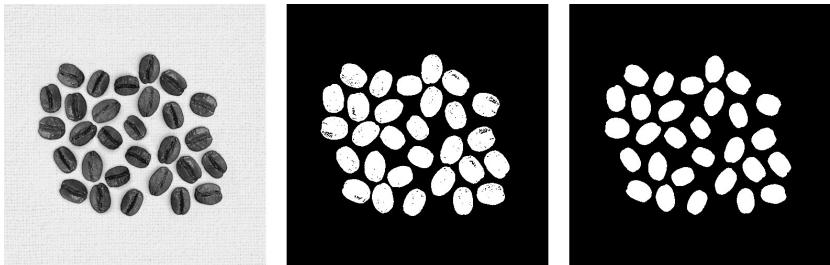


Figura 9.2: Tratamento com operação morfológica

Além do procedimento de binarização de imagens, existem outros dois que serão detalhadas a seguir.

Binarização adaptativa

No procedimento anterior, de binarização da imagem dos grãos de café, um valor global foi definido para o limiar de binarização. Em outras palavras, um mesmo valor foi utilizado como parâmetro para tratar cada pixel da imagem. Esse procedimento apresenta bons resultados quando a fotografia está uniformemente iluminada; caso contrário, a variação da luminosidade pode dificultar a segmentação do objeto.

Na figura adiante, a fotografia apresentada à esquerda exemplifica essa ocorrência. Nela, observe que as pílulas representadas na região inferior esquerda estão menos iluminadas quando comparadas às da região superior direita. Submetendo-a ao procedimento de binarização global, e definindo o valor do limiar em 127, obtemos como resultado a imagem apresentada à direita na figura a seguir.

Nesse contexto, observe que a operação foi malsucedida, pois a

falta de iluminação na região inferior da imagem provocou o desaparecimento de algumas pílulas após o procedimento de binarização.

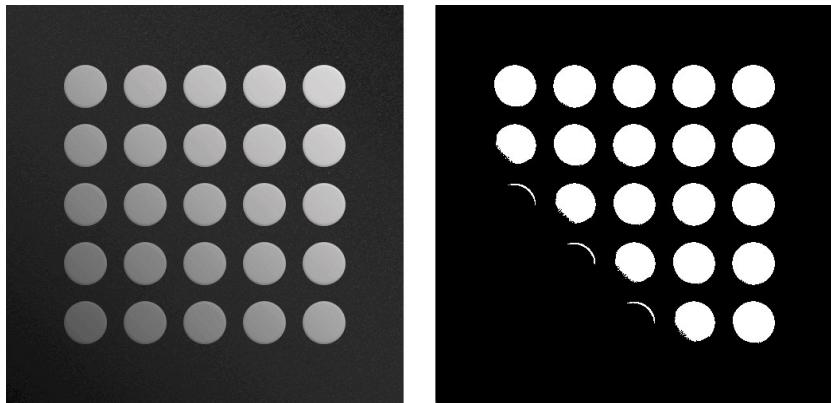


Figura 9.3: Segmentação por binarização malsucedida

Em situações como esta, em que a imagem não possui iluminação adequada para o procedimento de binarização, o algoritmo de binarização adaptativa apresenta bons resultados. Este calcula diferentes valores de limiar para cada região da imagem, logo, cada região é tratada a fim de obter o melhor resultado considerando o seu contraste.

A biblioteca OpenCV possui a função `adaptiveThreshold`, que nos permite aplicar um algoritmo de binarização adaptativa a uma imagem. Para utilizar essa função, seis parâmetros devem ser informados, cada um deles discriminado na tabela seguinte.

Parâmetro	Descrição
<code>src</code>	Matriz referente à imagem.
<code> maxValue</code>	Valor de intensidade máxima do pixel.

Parâmetro	Descrição
adaptiveMethod	Algoritmo de binarização adaptativa.
thresholdType	Tipo de binarização.
blockSize	Tamanho da máscara.
C	Constante de subtração da média ou da média ponderada.

O primeiro parâmetro indica a imagem que receberá o tratamento. O segundo indica o valor da intensidade máxima que um pixel pode ser representado em tons de cinza. O terceiro nos permite decidir o algoritmo de binarização adaptativa que será utilizado. A biblioteca OpenCV nos oferece as duas opções listadas:

- ADAPTIVE_THRESH_MEAN_C
- ADAPTIVE_THRESH_GAUSSIAN_C

Esses dois algoritmos desempenham a mesma tarefa, entretanto, utilizando métodos matemáticos distintos. Na prática, é válido experimentar cada um deles, assim podemos determinar qual apresenta melhores resultados para a solução do problema em questão.

O quarto parâmetro indica o tipo de aplicação do limiar, sendo os dois tipos mais usados o THRESH_BINARY e o THRESH_BINARY_INV , ambos apresentados no início deste capítulo. O quinto parâmetro indica o tamanho da máscara, ou seja, a quantidade de pixels vizinhos que serão afetados a cada pixel alvo processado. O valor que define o tamanho da máscara deve ser um número inteiro, positivo e ímpar. Geralmente, usamos valores de 3 a 11.

Por último, o sexto parâmetro define uma constante que será subtraída do valor definido para cada pixel pelo algoritmo de binarização. Esta pode ser qualquer número inteiro.

Com tantos parâmetros e uma infinidade de combinações possíveis para defini-los, ficar em dúvida sobre qual a melhor combinação de valores a ser utilizada é uma complicação frequente. Mesmo assim, não há regras ou dicas que possam ser aplicadas. Esses valores devem ser definidos, geralmente por tentativa e erro, para solucionar cada problema específico.

O código exemplifica como utilizar a função `adaptiveThreshold` para binarizar a fotografia dos comprimidos:

```
import cv2
import numpy as np

imgOriginal = cv2.imread("comprimidos.jpeg", 0)
imgTratada = cv2.medianBlur(imgOriginal, 7)

imgBinarizada = cv2.adaptiveThreshold(
    imgTratada, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY_INV, 11, 5
)

cv2.imshow("Imagen Original", imgOriginal)
cv2.imshow("Imagen Tratada", imgBinarizada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Na quarta linha do código, observe que um filtro de mediana foi aplicado à imagem. Esse procedimento é necessário para evitar ruídos durante o processo de binarização. Na figura a seguir, a imagem à esquerda apresenta a fotografia original carregada, e à

direita apresenta o resultado do processo de binarização adaptativa dessa mesma foto.

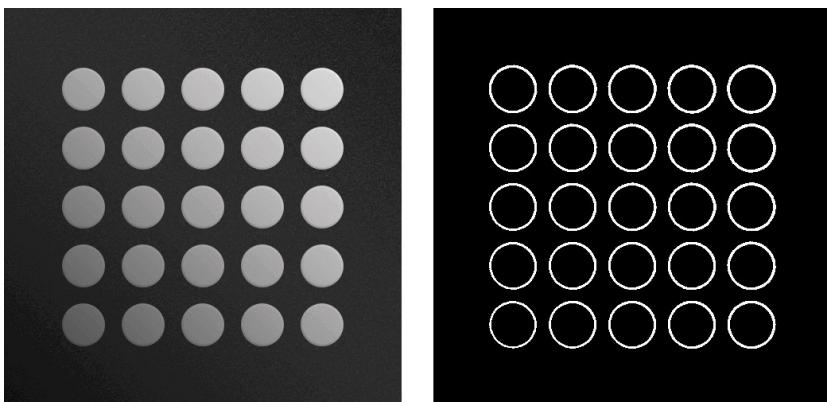


Figura 9.4: Resultado da binarização adaptativa

Observe na imagem binarizada das pílulas, apresentada à direita, que a variação da iluminação não interferiu na qualidade do resultado obtido pelo procedimento de binarização adaptativa. Diferente do algoritmo global, aplicado anteriormente, nesse procedimento nenhuma pílula foi eliminada da imagem.

Uma característica desfavorável desse processo, apresentada na figura anterior, é que ele pode gerar imagens onde os objetos de interesse nem sempre estão devidamente preenchidos. A binarização adaptativa apresenta bons resultados, por exemplo, quando aplicada a imagens com texto. A figura ilustra esse processo:

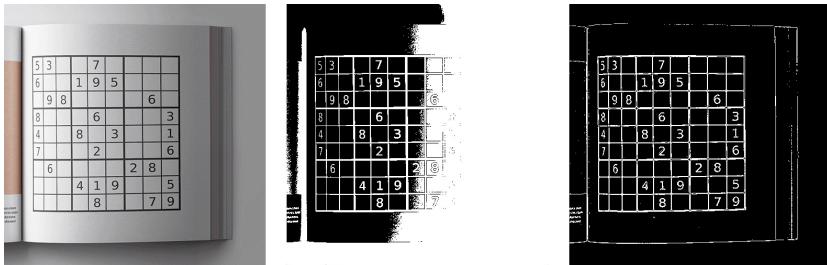


Figura 9.5: Comparação entre a binarização global com a adaptativa

Nessa figura, observe que a primeira imagem, da esquerda para a direita, foi submetida a dois procedimentos de binarização. A segunda imagem apresenta o resultado obtido com a binarização global. Repare que a iluminação desfavorável e mal distribuída provocou a perda de informações relevantes representadas na foto original.

A última imagem apresenta a fotografia do livro submetida ao procedimento de binarização adaptativa. Nela, note que, nesse procedimento, não houve perda de informações relevantes na imagem, possibilitando a leitura de todos os números.

Tanto a função `threshold` quanto a função `adaptiveThreshold` da biblioteca OpenCV nos permitem binarizar uma imagem qualquer, entretanto, ambas requerem parâmetros que precisam ser definidos por tentativa e erro. Felizmente, a biblioteca OpenCV nos fornece um terceiro método para binarizar imagens, conhecido como binarização de Otsu.

Esse método define automaticamente um limiar de binarização para a imagem. Mais detalhes sobre a binarização de Otsu serão vistos a seguir.

Binarização de Nobuyuki Otsu

No início desta seção, vimos a função `threshold` da biblioteca OpenCV (para binarização global de imagens). Diferente da `adaptiveThreshold`, a função `threshold` binariza cada pixel de uma imagem a partir de um limiar predefinido, sem considerar os pixels vizinhos ao pixel-alvo.

O grande contratempo dessa função é que um limiar precisa ser definido pelo usuário. A fim de automatizar esse procedimento, um algoritmo desenvolvido por Nobuyuki Otsu define um limiar baseado no histograma da imagem.

Esse algoritmo pode ser usado junto à função `threshold`, necessitando apenas que a constante `THRESH_OTSU` seja somada ao tipo definido para a binarização. O código seguinte exemplifica como binarizar uma imagem com o algoritmo de Otsu para definição do limiar.

```
import cv2
import numpy as np

imgOriginal = cv2.imread("graos-de-cafe.jpeg", 0)

tipo = cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU
limiar, imgBinarizada = cv2.threshold(imgOriginal, 0, 255, tipo)

print(limiar)

cv2.imshow("Imagen Original", imgOriginal)
cv2.imshow("Imagen Tratada", imgBinarizada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Nesse código, observe que o segundo parâmetro foi definido como 0, indicando que o valor será automaticamente estabelecido

pelo algoritmo de Otsu. O resultado de sua execução é apresentado na figura:

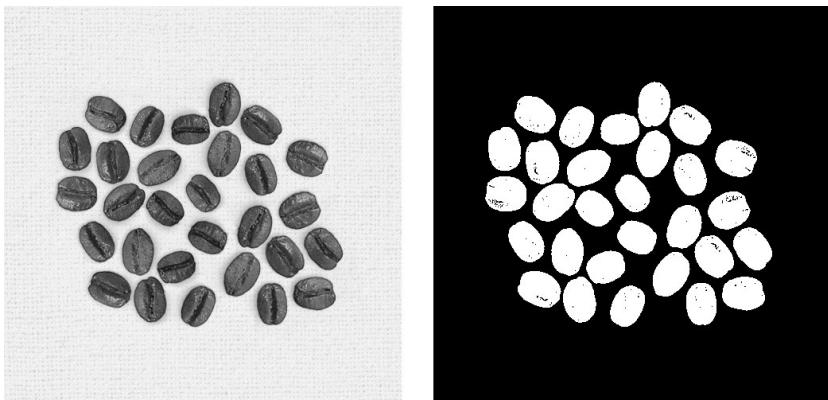


Figura 9.6: Segmentação por binarização de Nobuyuki Otsu

A imagem à esquerda apresenta a fotografia original após ser convertida em tons de cinza. A imagem à direita apresenta o resultado do processo de binarização de Otsu. Para essa imagem, o algoritmo de Otsu estabeleceu o valor do limiar igual a 150.

Com o objetivo de apresentar a eficácia do algoritmo de Otsu para definição do limiar, a figura seguinte compara o resultado obtido com o limiar definido por tentativa e erro com o obtido pelo algoritmo.

A imagem à esquerda foi binarizada com o limiar igual a 135, definido por tentativa e erro. A imagem à direita foi binarizada com o limiar igual a 150, definido pelo algoritmo de Otsu.

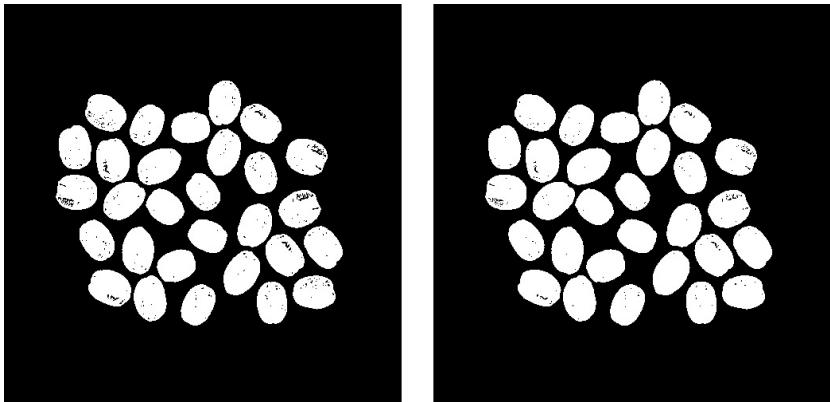


Figura 9.7: Comparação dos resultados obtidos com a binarização

Observe que o limiar definido pelo algoritmo de Otsu apresentou melhores resultados quando comparado ao obtido por tentativa e erro. Os grãos de café, representados na figura à direita, apresentam menos falhas, ou seja, possuem a área melhor preenchida por pixels brancos. Definir o valor do limiar por tentativa e erro pode se tornar uma tarefa um tanto quanto cansativa, uma vez que mais de 200 valores podem ser verificados.

Na próxima seção, será apresentado o procedimento de segmentação por cor. Por meio desse procedimento, é possível separar objetos de uma determinada cor da imagem.

9.2 SEGMENTAÇÃO POR COR

O procedimento de segmentação por cor nos permite separar objetos de interesse representados por uma cor específica na imagem. Para facilitar essa tarefa, o espaço de cor HSV é quase sempre utilizado, justamente por representar as informações referentes à cor (matiz) em um único canal.

O método `inRange` da biblioteca OpenCV pode ser usado para executar esse procedimento. Ele requer três parâmetros obrigatórios: o primeiro é a imagem representada no espaço HSV; o segundo, um vetor contendo os valores referentes ao limite inferior da cor desejada, representada também no espaço HSV; e o terceiro, assim como o segundo, também é um vetor, entretanto, contém os valores referentes ao limite superior da cor desejada.

O limite inferior corresponde à tonalidade mais clara da cor do objeto de interesse, já o limite superior corresponde à mais forte ou escura. Para obter esses valores, a tabela a seguir pode ser consultada, ou, então, basta converter uma determinada cor do espaço RGB para o HSV.

Cor	Limite inferior	Limite superior
Amarelo	10, 100, 100	50, 255, 255
Azul	100, 100, 100	140, 255, 255
Verde	40, 100, 100	80, 255, 255
Vermelho	160, 100, 100	200, 255, 255

Para obter os valores de uma cor do espaço RGB no espaço HSV, podemos executar o procedimento exemplificado pelo código seguinte. Basta definir os valores referentes à tonalidade de vermelho, verde e azul de uma cor no espaço RGB em um vetor. Em seguida, usamos o método `cvtColor` para a conversão, o mesmo estudado no capítulo sobre representação de cores no espaço.

```
import cv2
import numpy as np

verdeRGB = np.uint8([[[0,255,0]]])
```

```
verdeHSV = cv2.cvtColor(verdeRGB, cv2.COLOR_BGR2HSV)

print verdeHSV
```

O resultado dessa execução será:

```
[[[60 255 255]]]
```

O vetor retornado pela execução do código apresenta os valores que representam respectivamente os canais de matiz, a saturação e o valor da cor no espaço HSV. Para obter o limite superior e inferior, basta somar +20 e subtrair -20 ao valor do primeiro canal.

Para o limite inferior, o valor dos canais de saturação e valor devem ser definidos como 100. Para o limite superior, esses canais devem ser definidos para o valor máximo, no caso, 255. Essas definições são apenas sugestões que se aplicam à maioria dos casos, fornecendo bons resultados.

Veja a seguir um exemplo de como seria esse procedimento para a cor verde, obtida no exemplo anterior.

```
# Limite inferior:
# 60 - 20 = 40
tomClaro = np.array([40 100 100])

# Limite superior:
# 60 + 20 = 80
tomEscuro = np.array([80 255 255])
```

O código a seguir exemplifica como segmentar objetos coloridos em uma imagem com o método `inRange`. Nesse exemplo, as faces vermelhas de um cubo mágico são segmentadas em uma imagem binária.

```
import cv2
import numpy as np
```

```

imgRGB = cv2.imread("cubo-magico.jpeg")
imgHSV = cv2.cvtColor(imgRGB, cv2.COLOR_BGR2HSV)

tomClaro = np.array([160, 100, 100])
tomEscuro = np.array([200, 255, 255])

imgSegmentada = cv2.inRange(imgHSV, tomClaro, tomEscuro)

cv2.imshow("Original", imgRGB)
cv2.imshow("Segmentada", imgSegmentada)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

A figura seguinte apresenta o resultado da execução do código anterior. A imagem à esquerda ilustra o arquivo original carregado, e à direita, o resultado da segmentação das faces vermelhas.

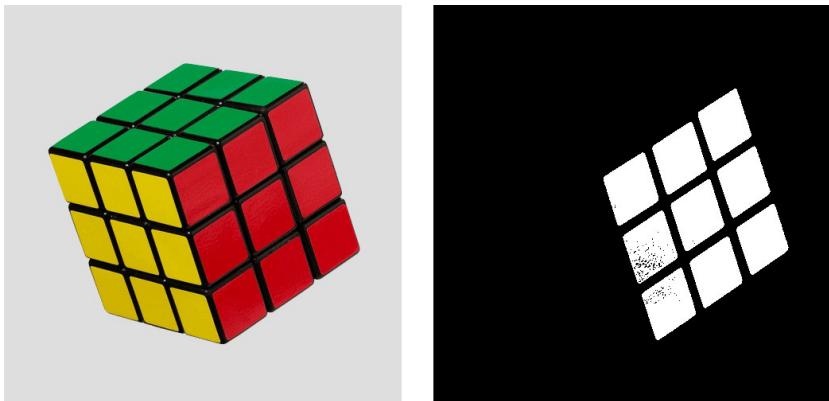


Figura 9.8: Segmentação por cor

Observe que a área vermelha de algumas faces possui falhas, com pontos pretos quando deveriam ser completamente brancas. Assim como no procedimento de segmentação por binarização, essas falhas são provocadas também pelo procedimento de

segmentação.

Elas ocorrem quando o objeto de interesse possui pixels que não estão representados entre a tonalidade mais clara e mais escura definidas. O tratamento delas também pode ser realizado através de operações morfológicas. A figura adiante ilustra esse processo.

A imagem à esquerda apresenta a imagem segmentada com falhas, e à direita, a mesma após sofrer o tratamento com a operação morfológica de fechamento – com elemento estruturante retangular de ordem 3.

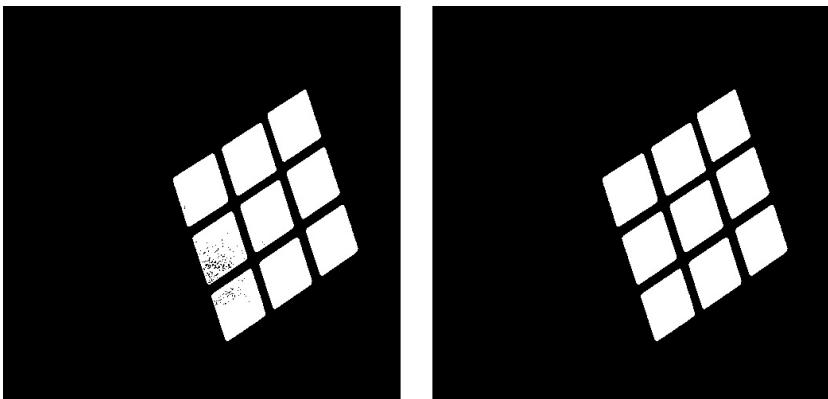


Figura 9.9: Segmentação por cor após tratamento

Na próxima seção, abordaremos outro método de segmentação de objetos em imagem, a segmentação através de bordas.

9.3 SEGMENTAÇÃO POR BORDAS

No capítulo anterior, alguns métodos para realçar contornos em imagens foram apresentados. Eles podem ser usados também para procedimentos de segmentação de objetos por borda. O mais

eficiente desses métodos é o detector de bordas de Canny, utilizado na segmentação dos grãos de café na figura a seguir.



Figura 9.10: Segmentação por borda

O código seguinte exemplifica como a segmentação por borda dos grãos de café foi realizada. O primeiro passo foi converter a imagem original para tons de cinza. Em seguida, aplicamos o procedimento de segmentação por binarização, removendo as informações sobre o segundo plano.

Após essa etapa, foram aplicadas as operações morfológicas de fechamento e erosão, para tratamento de ruído. Por último, usamos o método Canny para detectar as bordas dos grãos, representando em uma nova imagem somente as bordas referentes aos objetos de interesse.

```
import cv2
import numpy as np

imagem = cv2.imread("graos-de-cafe.jpeg", 0)

metodo = cv2.THRESH_BINARY_INV
ret, imgBinarizada = cv2.threshold(imagem, 135, 255, metodo)
```

```

e = np.ones((3, 3), np.uint8)
imgTratada = cv2.morphologyEx(imgBinarizada, cv2.MORPH_CLOSE, e)
imgTratada = cv2.erode(imgTratada, e, iterations = 2)

imgSegmentada = cv2.Canny(imgTratada, 100, 200)

cv2.imshow("Binarizada", imgBinarizada)
cv2.imshow("Tratada", imgTratada)
cv2.imshow("Segmentada", imgSegmentada)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

A figura seguinte ilustra a execução desse código. A primeira imagem, da esquerda para a direita, apresenta a dos grãos de café após o procedimento de binarização. A segunda mostra a primeira após o tratamento com operações morfológicas. A terceira imagem apresenta o resultado final da segmentação por bordas, representando apenas os contornos dos objetos de interesse.

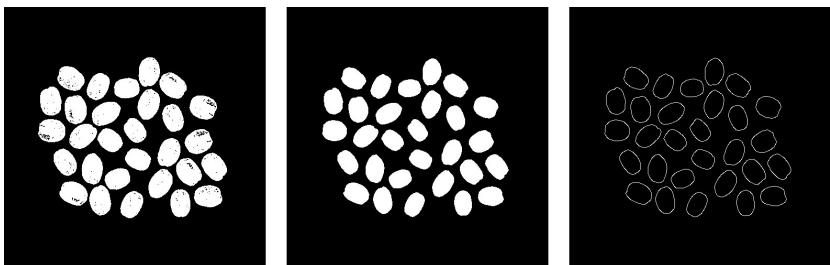


Figura 9.11: Procedimento de segmentação por borda

Na próxima seção, veremos um método para segmentação de objetos em movimento.

9.4 SEGMENTAÇÃO POR MOVIMENTO

O procedimento de segmentação por movimento consiste em detectar objetos que se moveram entre uma captura e outra, a partir de uma câmera fixa. O método mais eficaz para realizar essa tarefa é o `subtract` da biblioteca OpenCV, já apresentado no capítulo sobre pré-processamento de imagens.

A subtração de uma imagem por outra nos fornece a diferença entre elas, que vai representar o objeto de interesse. Os demais pixels que não sofreram mudanças são considerados o segundo plano da imagem, e todos serão apagados ou definidos como pixels pretos após esse procedimento.

```
import cv2
import numpy as np

cap1 = cv2.imread("captura-1.jpeg")
cap2 = cv2.imread("captura-2.jpeg")

imgRGB = cv2.subtract(cap1, cap2)
imgHSV = cv2.cvtColor(imgRGB, cv2.COLOR_BGR2HSV)

tomClaro = np.array([0, 120, 120])
tomEscuro = np.array([180, 255, 255])

imgSegmentada = cv2.inRange(imgHSV, tomClaro, tomEscuro)
cv2.imshow("Segmentada", imgSegmentada)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura adiante ilustra o processo de segmentação por movimento exemplificado por esse código. A primeira imagem, da esquerda para a direita, é a primeira captura feita da mesa de poker; a segunda, outra captura feita após a movimentação de uma das fichas. Com intuito de segmentar o objeto movimentado, a terceira imagem apresenta o resultado da operação de subtração da segunda pela primeira.

Observe na terceira imagem que somente o objeto de interesse está representado, e os pixels referentes ao segundo plano foram redefinidos para a cor preta pelo procedimento de subtração:

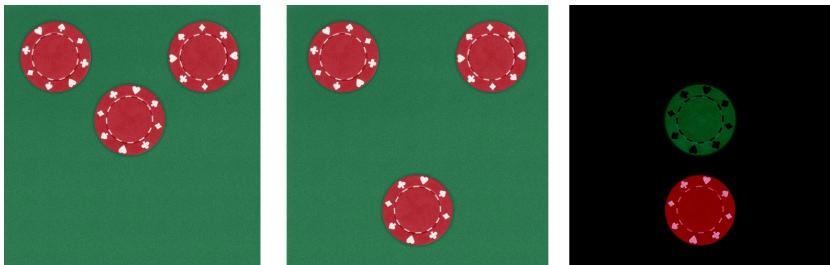


Figura 9.12: Segmentação por movimento

Ainda nessa figura, a terceira imagem apresenta o objeto de interesse já segmentado, entretanto, contém duas representações do objeto, antes e após a movimentação. Para segmentá-lo somente após a movimentação, uma solução é aplicar a segmentação por cor, uma vez que estão representados por cores diferentes. Em seguida, aplicaremos operações morfológicas para preencher a imagem ou tratar os ruídos.

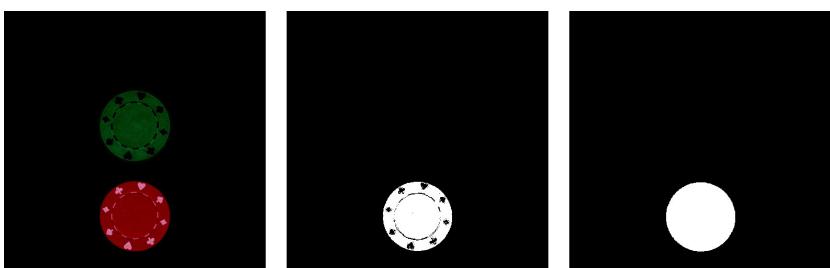


Figura 9.13: Segmentação por movimento após tratamento

Observe na figura anterior que a primeira imagem, da esquerda para a direita, apresenta o objeto de interesse antes e após a

movimentação. Na segunda, temos somente o objeto de interesse após a movimentação, segmentado pelo procedimento de segmentação por cor. Por último, a terceira imagem apresenta somente o objeto de interesse, já tratado e preenchido pela operação morfológica de fechamento – realizada através de um elemento estruturante retangular de ordem 10.

No próximo capítulo, serão apresentadas técnicas para extrair características de objetos já segmentados.

CAPÍTULO 10

EXTRAÇÃO DE CARACTERÍSTICAS

A extração de características de um objeto em uma imagem é um procedimento fundamental de qualquer sistema de Visão Computacional. Ela consiste em obter informações que tornam possível classificar ou identificar um objeto.

As características de um objeto podem ser classificadas em categorias, sendo as principais: características de aspecto, dimensionais, inerciais e topológicas. Neste capítulo, vamos detalhá-las nas seções a seguir. As informações que podem ser obtidas em cada uma delas serão exemplificadas em tópicos e, na primeira seção, veremos as de aspecto.

10.1 DE ASPECTO

As características de aspecto de um objeto de interesse definem informações sobre sua cor ou textura. As informações extraídas sobre a textura podem ser úteis para distinguir objetos de diferentes materiais, por exemplo, produzidos em madeira, metal ou cerâmica.

Por se tratar de um procedimento que requer maior

fundamentação matemática, a extração de informação de texturas não será abordada nesta obra. Aqui, estudaremos como as características sobre a cor de um objeto podem ser obtidas.

Cor

Das características que podem ser extraídas de um objeto, a cor é uma das mais importantes, tanto para a visão humana quanto para os sistemas baseados em Visão Computacional. Muitas vezes, somos capazes de identificar um objeto apenas por sua cor, e o mesmo é válido para os sistemas de Visão Computacional.

A figura a seguir apresenta três imagens de uma tampa plástica segmentada. A primeira delas, da esquerda para a direita, é uma imagem binária; a segunda, em tons de cinza; e a última, uma imagem colorida representada em RGB. Elas três serão usadas nos exemplos desse tópico, em que será demonstrado como obter a cor predominante em cada uma delas.



Figura 10.1: Imagens de objetos de interesse segmentados

Para extrair as características de cor sobre cada uma dessas imagens, as funções `ravel` (da biblioteca OpenCV) e `mode` (da biblioteca Statistics) são duas ferramentas eficientes e adequadas. A biblioteca Statistics é uma com funções estatísticas, que nos

permite obter rapidamente, por exemplo, moda, média e mediana de um Rol.

Já a função `ravel` foi apresentada no capítulo sobre pré-processamento de imagens. Relembrando, essa função transforma uma matriz em uma lista de valores. No exemplo a seguir, ela é usada para armazenar, em uma lista, o valor da intensidade de todos os pixels que representam a imagem do objeto segmentado.

Essa lista pode ser vista estatisticamente como um Rol. Em seguida, o valor que se repete mais vezes na lista, conhecido como moda, é obtido pela função `mode` da biblioteca Statistics.

```
import cv2
import statistics

imgBinaria = cv2.imread("tampa-binaria.bmp", 0)
imgTonsDeCinza = cv2.imread("tampa-tons-de-cinza.jpeg", 0)

rolBinaria = imgBinaria.ravel()
rolTonsDeCinza = imgTonsDeCinza.ravel()

print(statistics.mode(rolBinaria))
print(statistics.mode(rolTonsDeCinza))
```

Veja o resultado dessa execução:

```
255
239
[Finished in 0.5s]
```

A execução do programa anterior retornou a cor predominante de pixel nas duas imagens. No caso da imagem binária, o valor 255 retornado nos indica que a imagem é predominantemente composta por pixels brancos. Do contrário, o valor 0 seria retornado, indicando a predominância de pixels pretos. Em relação à imagem em tons de cinza, uma vez que a variação dos valores

ocorre entre 0 a 255, a cor de pixel predominante pode não ser uma informação muito relevante.

Outra informação que pode ser extraída é a média da cor dos pixels que representam o objeto segmentado. Dessa forma, a variação da luz sobre o objeto influenciará menos no valor obtido. Esse procedimento pode ser realizado com a função `mean`, da biblioteca OpenCV.

Ela nos retorna a média dos valores armazenados em uma matriz, no caso, em uma imagem. Desse mesmo modo, podemos obter a média da intensidade de cinza de cada canal de uma imagem colorida em RGB. O código seguinte exemplifica esse procedimento:

```
import cv2

imgRGB = cv2.imread("tampa-rgb.jpeg")
imgTonsDeCinza = cv2.imread("tampa-tons-de-cinza.jpeg", 0)

valorMedioRGB = cv2.mean(imgRGB)
valorMedioCinza = cv2.mean(imgTonsDeCinza)

print(valorMedioRGB)
print(valorMedioCinza)
```

O resultado da execução será:

```
(108.505552, 111.45012, 213.38306, 0.0)
(160.6873, 0.0, 0.0, 0.0)
[Finished in 2.6s]
```

A execução desse código retornou dois conjuntos de valores, cada um deles com quatro elementos. Isso ocorre pois a função `mean` consegue processar imagens com até 4 canais. No primeiro conjunto, os três primeiros valores são referentes à média dos valores obtidos nos canais B, G e R, respectivamente, da imagem

colorida em RGB.

O primeiro valor no segundo conjunto é referente à média dos valores que representam a intensidade dos pixels na imagem em tons de cinza. Os demais não citados, definidos como 0.0, podem ser ignorados, já que indicam apenas que o canal é inexistente na imagem.

Repare que os valores obtidos estão representados em números decimais. Considerando que estamos tratando sobre intensidade de tons de cinza, representadas como números inteiros em 8 bits, esses valores podem ser arredondados ou truncados. Por exemplo, podemos considerar que o tom de cinza médio da imagem em tons de cinza tem intensidade igual a 161.

Para definirmos a cor predominante na imagem em RGB, vamos considerar os valores apresentados na tabela:

Canal	Referência	Valor obtido	Valor truncado
Vermelho	R	213.38306	213
Verde	G	111.45012	111
Azul	B	108.505552	108

Observe que o valor da média referente ao canal vermelho é superior ao dos demais. Por essa característica, podemos deduzir que o objeto é predominantemente composto por pixels em tons de vermelho.

Na próxima seção, serão detalhadas as características referentes às dimensões do objeto de interesse.

10.2 DIMENSIONAIS

As características dimensionais definem informações sobre o tamanho do objeto de interesse. A área, o perímetro e o diâmetro são as três principais dessa categoria. A figura seguinte ilustra três imagens binárias de objetos segmentados, que serão utilizadas nos próximos exemplos.

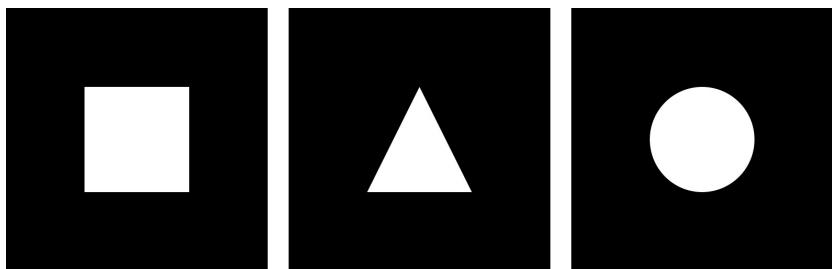


Figura 10.2: Imagens de objetos de interesse segmentados

A primeira característica dimensional que será detalhada é a área do objeto.

Área do objeto

A área de um objeto de interesse é definida pelo total de pixels que o representa. Essa informação pode ser obtida através da função `contourArea` da biblioteca OpenCV. Ela requer apenas o objeto segmentado como parâmetro, obtido com a função `findContours`.

A função `findContours` extrai de uma imagem binária os pontos que representam os contornos dos objetos segmentados. Para a sua execução, três parâmetros são obrigatórios: o primeiro deles é a imagem binária de onde os objetos serão extraídos; e o

segundo e o terceiro definem, respectivamente, o modo e o método de aproximação de contorno.

O modo indica como os pontos extraídos serão armazenados na lista `contornos`. O padrão é utilizar o modo `cv2.RETR_TREE`, indicando que todos os pontos obtidos serão armazenados.

O método de aproximação de contorno define o critério que será usado para obter os pontos na imagem binária, e o mais comum é o `cv2.CHAIN_APPROX_SIMPLE`. Este indica que os segmentos horizontais, verticais e diagonais serão comprimidos, deixando apenas os pontos finais. Na prática, significa que, se um quadrado estiver representado na imagem, somente os quatro pontos referentes aos vértices serão retornados.

Mais informações sobre os modos e os métodos de aproximação podem ser consultadas na documentação online da biblioteca OpenCV.

Mesmo com tantos detalhes, obter a área de um objeto, através da função `contourArea`, é uma tarefa tanto quanto simples. Veja o código:

```
import cv2
import numpy as np

imagem = cv2.imread("quadrado.bmp", 0)
tipo = cv2.THRESH_BINARY
ret, imgBinarizada = cv2.threshold(imagem, 127, 255, tipo)

# Obtendo os contornos dos objetos na imagem
modo = cv2.RETR_TREE;
metodo = cv2.CHAIN_APPROX_SIMPLE;
contornos, hierarquia = cv2.findContours(
    imgBinarizada, modo, metodo
)
```

```

# Obtendo os contornos do primeiro objeto segmentado
objeto = contornos[0]

# Obtendo a área do objeto segmentado
area = cv2.contourArea(objeto)
print(area)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

Veja o resultado da execução:

```

39601.0
[Finished in 0.5s]

```

Repare no código que, além da lista `contornos`, uma lista chamada `hierarquia` também precisa obrigatoriamente ser definida. Esta armazena informações sobre a topologia da imagem, ou seja, dados referentes aos pontos extraídos como contornos. A execução desse exemplo retornará o valor `39601.0`, indicando a quantidade de pixels que representam a área do quadrado segmentado na imagem.

O mesmo código anterior foi executado para as três imagens apresentadas no início desta seção. A primeira imagem apresenta um quadrado de largura igual a 200 pixels; a segunda, um triângulo com a base e a altura igual a 200 pixels; e a última, um círculo com 200px de diâmetro.

Com essas informações, podemos obter para cada um desses objetos o valor esperado para sua área. A tabela seguinte apresenta os resultados esperados, os obtidos e o erro percentual para cada um deles:

Objeto	Valor esperado	Valor obtido	Erro percentual
Quadrado	40000	39601	0.1%

Objeto	Valor esperado	Valor obtido	Erro percentual
Triângulo	20000	19900	0.5%
Círculo	31416	31495	0.2%

Observe que os valores obtidos são próximos ao esperado, entretanto, não representam com exatidão a área do objeto de interesse. O objeto pode sofrer pequenas mudanças durante a segmentação por binarização. Além disso, pode ser que a função `findContours` não identifique os pontos exatos referentes aos contornos do objeto. Essas questões justificam a diferença entre o valor esperado e o obtido.

A área desses objetos pode ser obtida também contabilizando o total de pixels brancos representados. Esse procedimento já foi exemplificado no capítulo sobre pré-processamento de imagens. A tabela a seguir apresenta os resultados obtidos para as mesmas imagens por meio desse procedimento.

Objeto	Valor esperado	Valor obtido	Erro percentual
Quadrado	40000	40000	0.00%
Triângulo	20000	20000	0.00%
Círculo	31416	31432	0.05%

Comparando os resultados obtidos com a função `findContours` e `contourArea`, com a contagem de pixels brancos, evidentemente a segunda alternativa apresenta melhores resultados. Mesmo assim, é válido estudar a função `findContours`, pois ela será usada para obter as demais características dimensionais de um objeto. No próximo tópico, veremos como obter o perímetro de objetos de interesse.

Perímetro do objeto

O perímetro de um objeto é definido pela soma dos pixels que representam o seu contorno. Este valor pode ser obtido pela função `arcLength` da biblioteca OpenCV. Ela requer dois parâmetros obrigatórios. O primeiro indica o objeto de interesse obtido com a função `findContours`, e o segundo define o tipo de contorno que será obtido.

Quando o segundo parâmetro é definido como `True`, obteremos um contorno fechado do objeto, com o primeiro vértice conectado ao último. Do contrário, quando definido como `False`, um contorno aberto pode ser retornado. Uma vez que desejamos obter o perímetro do objeto, o segundo parâmetro deve ser definido como `True`.

```
perimetro = cv2.arcLength(objeto, True)
print(perimetro)
```

O resultado dessa execução será:

```
796.0
[Finished in 0.5s]
```

A tabela adiante compara o valor obtido com o esperado para o perímetro do objeto de cada imagem. O valor esperado foi calculado a partir dos dados reais dos objetos representados na imagem. Os dados desses objetos são os mesmos apresentados no tópico anterior.

Objeto	Valor esperado	Valor obtido	Erro percentual
Quadrado	800.00	796.0	0.5%
Triângulo	647.21	680.0	5.0%
Círculo	628.32	662.4	5.4%

Repare que o valor obtido se aproxima novamente do valor esperando, entretanto, sem representar com exatidão o perímetro do objeto. Este erro também é provocado pelo procedimento de binarização, ou pela função `findContours`.

Uma outra abordagem para obtermos o perímetro de um objeto é segmentar sua borda através do método de detecção de bordas de Canny. Com a borda do objeto segmentada, bastar percorrer toda a imagem contabilizando o total de pixels brancos. A figura a seguir apresenta as imagens obtidas após a segmentação por borda.

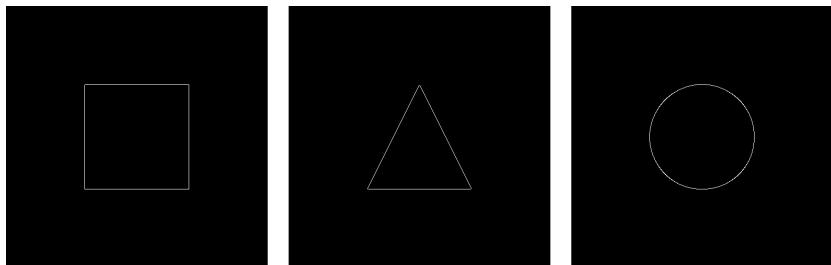


Figura 10.3: Objetos segmentados por borda

O código para execução desse algoritmo também está disponível no repositório no GitHub, em <https://github.com/felipecbarelli/livro-visao-computacional>. A tabela seguinte apresenta os resultados obtidos com esse procedimento:

Objeto	Valor esperado	Valor obtido	Erro percentual
Quadrado	800.00	796	0.5%
Triângulo	647.21	593	8.4%
Círculo	628.32	660	5.0%

Comparando as duas abordagens para extração do perímetro, observe que não houve diferença no valor obtido para a borda segmentada do quadrado. Para o triângulo, a função `arcLength` apresentou melhores resultados, porém, para a extração do perímetro do círculo, o algoritmo para contabilizar pixels brancos demonstrou ser mais eficiente.

Na próxima seção, veremos as características inerciais de um objeto.

10.3 INERCIAIS

As características inerciais definem informações sobre os momentos, o centro geométrico e as formas geométricas envolventes de um objeto de interesse. As características de um objeto obtidas a partir dos seus momentos estatísticos são as mais relevantes apresentadas neste capítulo.

Algumas delas possibilitam o reconhecimento de um objeto mesmo que ele tenha sofrido alterações na escala, rotação ou translação. No tópico a seguir, será detalhado o que é e como obter os momentos estatísticos de um objeto de interesse.

Momentos de uma imagem

Os momentos de uma imagem (também conhecidos como momentos estatísticos) é um dos principais métodos para extração de características de objetos em imagens. Eles são definidos por funções matemáticas, com base estatística, que fornecem valores que representam um determinado objeto.

Esses valores são calculados a partir de imagens binárias do objeto segmentado. Os resultados obtidos com esses cálculos descrevem características como área, orientação e ponto geométrico do objeto de interesse.

Alguns desses momentos são invariantes, ou seja, não sofrem alterações significativas no valor quando o objeto é rotacionado ou redimensionado. Detalhá-los matematicamente foge do escopo do livro, mas, caso você tenha interesse e queira conhecer um pouco mais sobre, recomendo a leitura do artigo *Aplicação da técnica de momentos invariantes no reconhecimento de padrões em imagens digitais* (ALBUQUERQUE; ALBUQUERQUE; CHACON et al., 2011).

A biblioteca OpenCV possui a função `moments`, que nos retorna uma estrutura contendo os valores dos 24 momentos que caracterizam o objeto. O código a seguir exemplifica esse procedimento.

```
import cv2
import numpy as np

imagem = cv2.imread("quadrado.bmp", 0)
momentos = cv2.moments(imagem)

print(momentos)
```

Veja o resultado dessa execução:

```
{'mu02': 33999150000.0, 'mu03': 0.0, 'm11': 634952550000.0, 'nu02': 0.00032678921568627447, 'm12': 166903449150000.0, 'mu21': 0.0, 'mu20': 33999150000.0, 'nu20': 0.00032678921568627447, 'm30': 183869025000000.0, 'nu21': 0.0, 'mu11': 0.0, 'mu12': 0.0, 'nu11': 0.0, 'nu12': 0.0, 'm02': 668951700000.0, 'm03': 183869025000000.0, 'm00': 10200000.0, 'm01': 2544900000.0, 'mu30': 0.0, 'nu30': 0.0, 'nu03': 0.0, 'm10': 2544900000.0, 'm20': 668951700000.0, 'm21': 166903449150000.0}
```

[Finished in 0.8s]

A priori, esses valores retornados podem parecer confusos e sem muita relevância, entretanto, nos próximos tópicos, vamos utilizá-los para que outras características do objeto sejam obtidas. A seguir, veremos os momentos invariantes de Hu, sete características do objeto calculadas a partir dos momentos aqui apresentados.

Momentos invariantes de Hu

Os momentos invariantes de Hu são sete momentos calculados a partir dos momentos de uma imagem. Através deles, podemos obter a área, o centro geométrico e até mesmo um vetor de características invariantes em escala, rotação e translação de um objeto.

Ou seja, mesmo que o objeto sofra uma dessas transformações, os valores dos sete momentos não apresentarão grandes mudanças. Por essa característica, esses momentos são bastante utilizados para classificação de objetos.

A figura seguinte apresenta três imagens binárias de um mesmo objeto segmentado. Em todas, o objeto foi rotacionado. Além disso, a imagem do meio apresenta o objeto reduzido quando comparado às demais. Essas três imagens serão usadas no exemplo de como obter os sete momentos invariantes.

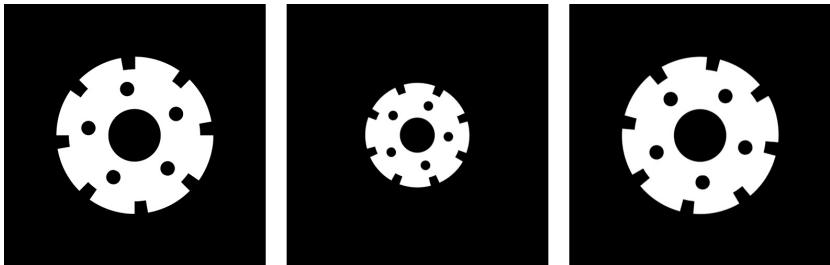


Figura 10.4: Momentos invariantes de Hu

Pela função `HuMoments` da biblioteca OpenCV, podemos obter os sete momentos invariantes de Hu. Ela requer como parâmetro apenas os momentos do objeto, que podem ser obtidos com a função `moments`, já apresentada no tópico anterior. O código seguinte mostra esse procedimento.

```
import cv2
import numpy as np

imagem = cv2.imread("engrenagem-1.bmp", 0)
momentos = cv2.moments(imagem)
momentosDeHu = cv2.HuMoments(momentos)

print(momentosDeHu)

# Aplicando a transformação logarítmica
print(-np.sign(momentosDeHu) * np.log10(np.abs(momentosDeHu)))
```

A execução desse código retornará duas listas com sete valores referentes aos momentos invariantes de Hu. A primeira delas apresenta os dados reais obtidos com a função `HuMoments`. A segunda lista, a fim de atenuar ainda mais as variações, apresenta os dados tratados através de uma transformação logarítmica.

Geralmente, os valores tratados com a transformação logarítmica apresentam melhores resultados quando o objetivo é

extrair características de figuras planas. Executando esse mesmo código para as três imagens, considerando os valores tratados com a transformação logarítmica, obtemos os resultados apresentados na tabela:

Momento	1 ^a imagem	2 ^a imagem	3 ^a imagem
M1	3.06261058	3.0625094	3.06258244
M2	11.36211953	10.88617117	11.34657291
M3	14.55757342	14.54584006	14.56955423
M4	16.22574365	16.24349816	16.19318771
M5	31.67569626	31.64907366	31.61578064
M6	-22.32208083	-21.91209342	-22.21729364
M7	31.93146255	32.29313435	31.95565654

Analisando a tabela, note que, mesmo o objeto sofrendo rotação ou alteração no tamanho, os sete momentos apresentam valores muito próximos para todas as imagens. Se os valores obtidos forem arredondados para números inteiros, podemos considerar os momentos como invariantes. Por esse motivo, essa é uma das características mais usadas para o reconhecimento e a classificação de objetos.

Imagine, por exemplo, uma esteira seletora, na qual engrenagens de diferentes dimensões são separadas em recipientes distintos, segundo o número de dentes que cada uma possui. Neste cenário, as dimensões dos objetos e a forma na qual estão posicionados sobre a esteira são características irrelevantes.

Podendo haver variações nas características dimensionais das engrenagens, os momentos invariantes de Hu são capazes de

melhor representar as características de cada uma delas.

No próximo tópico, será detalhado como obter o centro geométrico de um objeto segmentado.

Centro geométrico

O centro geométrico de um objeto, também conhecido como centroide, é o ponto que representa o centro de uma figura geométrica. Este pode ser obtido através dos momentos estudados no primeiro tópico desta seção.

O código adiante apresenta as equações necessárias para obter os valores das variáveis x e y que representam o ponto central. Usamos a imagem binarizada do quadrado, a primeira apresentada na seção anterior.

```
momentos = cv2.moments(objeto)
cx = int(momentos['m10'] / momentos['m00'])
cy = int(momentos['m01'] / momentos['m00'])
print(cx, cy)
```

O resultado da execução será:

```
249 249
[Finished in 0.5s]
```

No próximo tópico, será detalhado como obter o retângulo envolvente de um objeto segmentado.

Retângulo envolvente

O retângulo envolvente de um objeto de interesse é o menor retângulo no qual ele pode ser inscrito. Na figura a seguir, a imagem à esquerda ilustra o menor retângulo envolvente do objeto

de interesse representado na imagem à direita.

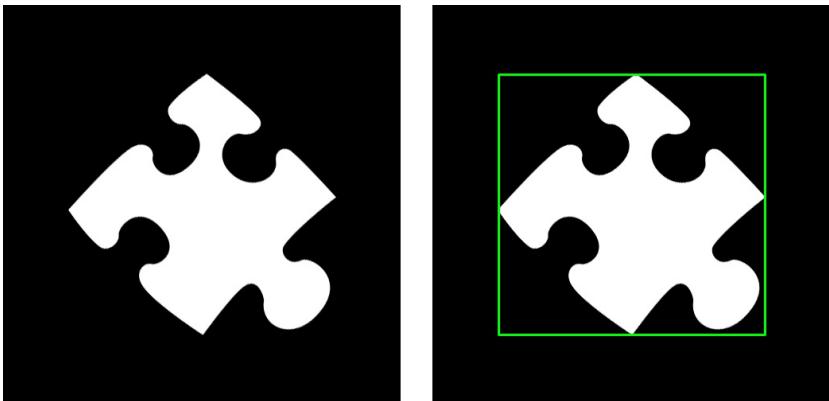


Figura 10.5: Retângulo envolvente

A função `boundingRect` da biblioteca OpenCV nos permite obter os dois vértices necessários para traçar o retângulo, requerendo apenas o objeto como parâmetro. O código a seguir mostra como obter o menor retângulo envolvente e desenhá-lo na imagem com a função `rectangle`.

```
import cv2
import numpy as np

imagemRGB = cv2.imread("puzzle.bmp")
imagemTonsDeCinza = cv2.imread("puzzle.bmp", 0)

tipo = cv2.THRESH_BINARY
ret, imgBinarizada = cv2.threshold(
    imagemTonsDeCinza, 127, 255, tipo
)

modo = cv2.RETR_TREE;
metodo = cv2.CHAIN_APPROX_SIMPLE;

contornos, hierarquia = cv2.findContours(
    imgBinarizada, modo, metodo
)
```

```

objeto = contornos[0]

# Obtendo os vértices do retângulo
x, y, w, h = cv2.boundingRect(objeto)

# Desenhando o retângulo na imagem imagemRGB
cv2.rectangle(imagemRGB, (x,y), (x+w, y+h), (0, 255, 0), 2)

cv2.imshow("Retangulo Envolvente", imagemRGB)
print(x, y, w, h)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

Veja o resultado da execução:

```

83 87 335 327
[Finished in 0.3s]

```

Observe que, para desenhar o retângulo na imagem através da função `rectangle`, cinco parâmetros são obrigatórios. A tabela seguinte define cada um deles:

Parâmetro	Descrição
<code>img</code>	Imagen em RGB na qual o retângulo será traçado.
<code>pt1</code>	Vértice do retângulo.
<code>pt2</code>	Vértice oposto ao <code>pt1</code> .
<code>color</code>	Cor em BGR da linha.
<code>thickness</code>	Espessura da linha em pixels.

No próximo tópico, será detalhado como obter a circunferência envolvente de um objeto segmentado.

Circunferência envolvente

A circunferência envolvente de um objeto de interesse é a menor circunferência na qual ele pode ser inscrito. Na figura a seguir, a imagem à esquerda ilustra a menor circunferência envolvente do objeto de interesse representado na imagem à direita.

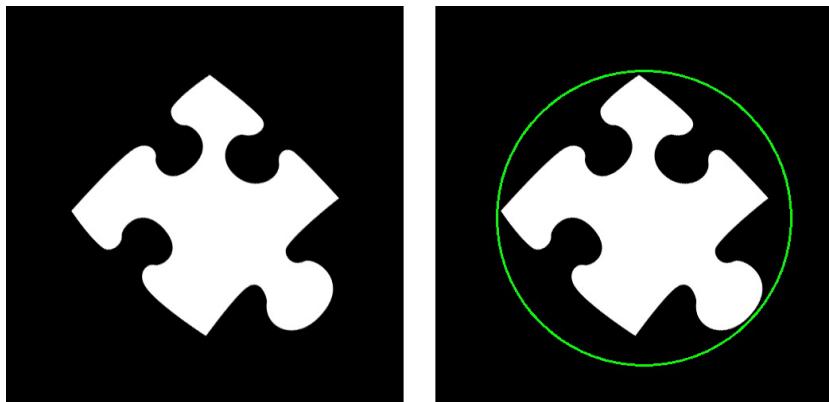


Figura 10.6: Circunferência envolvente

A função `minEnclosingCircle` da biblioteca OpenCV nos permite obter o ponto central e o raio dessa circunferência, sendo apenas necessário o objeto como parâmetro. O código a seguir nos mostra como obter a circunferência envolvente e desenhá-la na imagem com a função `circle`.

```
import cv2
import numpy as np

# Obtendo o ponto central e o raio da circunferência
(x,y), raio = cv2.minEnclosingCircle(objeto)

centro = (int(x), int(y))
raio = int(raio)

# Desenhando a circunferência na imagem imagemRGB
cv2.circle(imagemRGB, centro, raio, (0, 255, 0), 2)
```

```
cv2.imshow("Circunferencia Envolvente", imagemRGB)

print(x, y, raio)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

O resultado da execução será:

```
262.382720947 266.540405273 185
[Finished in 4.4s]
```

Observe que, para desenhar a circunferência na imagem através da função `circle`, são necessários cinco parâmetros. A tabela a seguir define cada um deles.

Parâmetro	Descrição
<code>img</code>	Imagen em RGB na qual o retângulo será traçado.
<code>center</code>	Ponto central da circunferência.
<code>radius</code>	Raio da circunferência.
<code>color</code>	Cor em BGR da linha.
<code>thickness</code>	Espessura da linha em pixels.

No próximo tópico, veremos como obter a elipse ajustada de um objeto segmentado.

Elipse ajustada

A elipse ajustada de um objeto de interesse é a menor elipse na qual ele pode ser inscrito. Na figura a seguir, a imagem à esquerda mostra a menor elipse ajustada do objeto de interesse representado na imagem à direita.

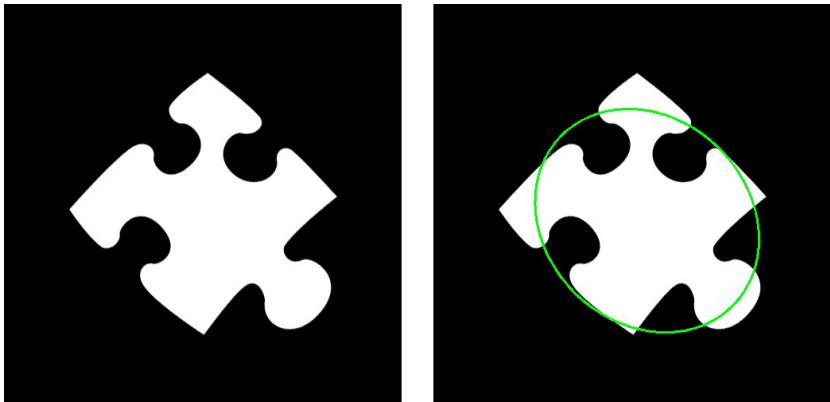


Figura 10.7: Elipse ajustada

A função `fitEllipse` da biblioteca OpenCV nos permite obter o menor retângulo rotacionado em que essa elipse pode ser inscrita, requerendo apenas o objeto como parâmetro. O código a seguir exemplifica como obter os valores dos vértices desse retângulo, o ângulo de rotação e como desenhar a elipse na imagem com a função `ellipse`.

```
import cv2
import numpy as np

# Obtendo a elipse
ellipse = cv2.fitEllipse(objeto)

# Desenhando a elipse na imagem imagemRGB
cv2.ellipse(imagemRGB, ellipse, (0, 255, 0), 2)
cv2.imshow("Elipse ajustada", imagemRGB)

print(ellipse)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Veja o resultado da execução:

```
((268, 271), (256, 303), 134)
[Finished in 0.5s]
```

A execução desse código retorna os dois pontos referentes aos vértices do menor retângulo e o ângulo de rotação. Para desenhar a circunferência na imagem através da função `ellipse`, quatro parâmetros são obrigatórios. A tabela a seguir define cada um deles.

Parâmetro	Descrição
<code>img</code>	Imagen em RGB na qual o retângulo será traçado.
<code>ellipse</code>	Ellipse obtida com a função <code>fitEllipse</code> .
<code>color</code>	Cor em BGR da linha.
<code>thickness</code>	Espessura da linha em pixels.

No próximo tópico, será detalhado como obter a orientação de um objeto segmentado.

Orientação

A característica de orientação de um objeto é definida pelo ângulo ao qual uma elipse ajustada ao objeto está direcionada. Como exemplificado no tópico anterior, o ângulo referente à orientação é o terceiro valor retornado na lista pela função `fitEllipse`. O código exemplifica como obter somente esse valor:

```
import cv2
import numpy as np

ellipse = cv2.fitEllipse(objeto)
angulo = ellipse[2]
print(angulo)
```

```
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

O resultado dessa execução será:

```
134.290405273  
[Finished in 0.5s]
```

Na próxima seção, as características topológicas de um objeto serão apresentadas.

10.4 TOPOLOGICAS

As características topológicas definem informações que não variam quando o objeto de interesse é movido, rotacionado ou sofre distorções na largura ou altura. A quantidade de vértices e furos que um objeto possui são exemplos de características dessa categoria.

Nesta seção, veremos como podemos obter essas informações de um objeto. A seguir, veja como extrair o número de vértices de um objeto segmentado.

Número de vértices

O número de vértices de um objeto segmentado pode ser obtido com o auxílio da função `approxPolyDP` da biblioteca OpenCV. Esta define um polígono para o objeto, tendo como base os pontos que representam seu contorno.

A função `approxPolyDP` requer três parâmetros obrigatórios. O primeiro deles é o objeto que será tratado, e o segundo indica a precisão de aproximação do objeto a um polígono – definido geralmente como 10% do valor do perímetro do objeto.

Na prática, quanto maior o valor do segundo parâmetro, menor será a sensibilidade do algoritmo para detectar os vértices. O terceiro e último parâmetro, quando definido como `True`, indica que a aproximação será realizada através de uma curva fechada, ou seja, com o primeiro e o último vértice conectados. Do contrário, quando definido como `False`, a aproximação será realizada por uma curva aberta.

A função `approxPolyDP` retorna uma lista de pontos referentes aos vértices do objeto. O total de vértices é obtido contabilizando o número de pontos nessa lista. O código exemplifica esse procedimento:

```
# Obtendo o perímetro do objeto
perimetro = cv2.arcLength(objeto, True)

# Obtendo os pontos referentes aos vértices
poligono = cv2.approxPolyDP(objeto, 0.1 * perimetro, True)

# Obtendo o total de vértices
totalVertices = len(poligono)

print(totalVertices)
```

Veja o resultado da execução:

```
4
[Finished in 0.5s]
```

Outra característica topológica de um objeto é o número de furos que ele possui. No próximo tópico, será detalhado como extrair essa informação.

Número de furos

O número de furos de um objeto segmentado pode ser obtido pela função `findContours`. Contabilizando o total de contornos

detetados na imagem e subtraindo uma unidade, obtemos o total de furos.

A unidade subtraída refere-se ao contorno que representa o próprio objeto. A quantidade de furos que um objeto possui pode ser uma característica determinante para diferenciá-lo de outros.

A figura a seguir apresenta três imagens de objetos circulares perfurados, e uma delas será utilizada para exemplificar esse procedimento.

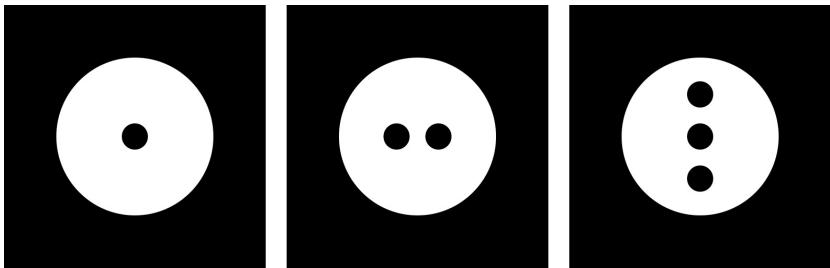


Figura 10.8: Objetos circulares perfurados

O código seguinte mostra como utilizar a função `findContours` para obter o total de furos de um objeto. Uma das imagens apresentadas na figura anterior foi submetida ao código, e o resultado da execução apontará quantos furos o objeto possui, permitindo o seu reconhecimento.

```
import cv2
import numpy as np

# carrega a imagem do objeto com dois furos
imagem = cv2.imread("circulo-perfurado-2.bmp", 0)

tipo = cv2.THRESH_BINARY
ret, imgBinarizada = cv2.threshold(imagem, 127, 255, tipo)
```

```
modo = cv2.RETR_TREE;
metodo = cv2.CHAIN_APPROX_SIMPLE;
contornos, hierarquia = cv2.findContours(
    imgBinarizada, modo, metodo
)

# obtém o total de contornos e subtrai um
furos = len(contornos) - 1

print(furos)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

O resultado dessa execução será:

```
2
[Finished in 0.5s]
```

Neste capítulo, vimos diferentes características que podem ser extraídas de um mesmo objeto. No próximo, detalharemos como estes podem ser classificados a partir dessas características.

CAPÍTULO 11

RECONHECIMENTO DE PADRÕES

A capacidade do ser humano de reconhecer padrões e classificar objetos sempre impressionou cientistas dos mais diversificados campos de estudo, principalmente os que se dedicam a desenvolver tecnologias capazes de imitar a natureza humana. O Reconhecimento de Padrões é o campo da ciência que estuda técnicas para descrever padrões de objetos a fim de classificá-los.

Nos sistemas baseados em Visão Computacional, as técnicas de reconhecimento de padrões são essenciais, pois possibilitam a classificação automática de um objeto de interesse. Em outras palavras, essas técnicas permitem que computadores enxerguem o mundo à nossa volta, reconhecendo placas, peças, caracteres, faces humanas e outros objetos.

Tanto para as máquinas como para nós, humanos, o reconhecimento de objetos é proporcionado por suas características particulares. Para realizar essa tarefa, esses sistemas executam basicamente três procedimentos. O diagrama de blocos a seguir apresenta sequencialmente cada um deles.

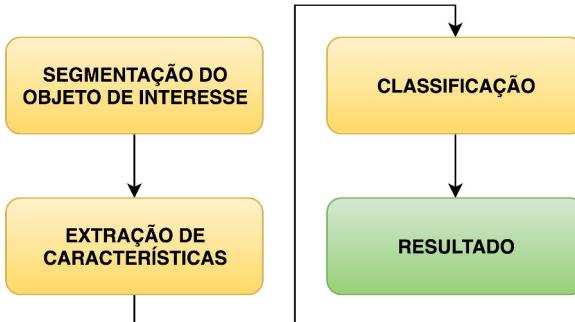


Figura 11.1: Sistemas de Reconhecimento de Padrões

A segmentação do objeto e a extração de características são assuntos que foram abordados nos capítulos anteriores. Neste, com intuito de definir o objeto de interesse em uma determinada classe, vamos estudar como reconhecer padrões nessas características.

Nesse contexto, uma classe pode ser definida como um conjunto de padrões que possuem as mesmas características. Para facilitar o entendimento, pense no problema de classificação de figuras geométricas, por exemplo. Sabemos que todo quadrilátero é um polígono simples de quatro lados, cuja soma dos ângulos internos é igual a 360° . Nesse exemplo, quadrilátero é uma classe que possui características padronizadas.

O algoritmo que classifica o objeto a partir de suas características é conhecido como classificador. Existem diversos desses algoritmos, e os mais populares em sistemas de Visão Computacional são os Bayesianos, o K-NN (*K-Nearest Neighbor*), a Lógica Nebulosa e as Redes Neurais Artificiais. Com esta diversidade de algoritmos com o mesmo propósito, surge o questionamento sobre qual é o melhor ou qual deve ser usado em

determinadas situações.

Os classificadores até podem ser avaliados quanto à velocidade de execução e à capacidade de fazer classificações corretas e outros parâmetros, entretanto, todos apresentam vantagens e desvantagens para cada problema específico de classificação. Consequentemente, fica a cargo do desenvolvedor decidir qual implementar.

A participação humana é fundamental não apenas para definir o algoritmo de classificação, mas também para definir as classes e as características consideradas no processo. Justamente por haver tantas variáveis envolvidas, compará-los a fim de definir qual classificador é melhor não é uma tarefa trivial. Por esse motivo, medir a eficiência de diferentes classificadores para solucionar um mesmo problema é um trabalho frequentemente desenvolvido por muitos pesquisadores.

Para que os classificadores consigam designar objetos em classes, sem que tenham sido explicitamente programados, eles precisam aprender alguns critérios para executar essa tarefa. Existem dois métodos de aprendizagem de máquina bastante usados para ensiná-los: a aprendizagem supervisionada e a não supervisionada. Nos tópicos a seguir, elas serão apresentadas.

Aprendizagem supervisionada

Nesse modelo de aprendizagem, o classificador é treinado para reconhecer padrões a partir de objetos já conhecidos. Um conjunto de características de objetos e suas respectivas classes são apresentados ao classificador. A partir desses dados, ele é treinado para identificar automaticamente padrões nessas informações,

tornando-se capaz de classificar novos objetos.

Justamente por necessitar de um agente externo, responsável por apresentar ao classificador dados para o treinamento, esse método é conhecido como supervisionado. Geralmente, os dados usados na etapa de treinamento são definidos por algum especialista no problema em questão.

A figura a seguir ilustra o processo de aprendizagem supervisionada. Observe que um conjunto de objetos está representado em um espaço de características bidimensional. Todos os objetos coloridos possuem classes definidas e indicadas pela legenda da imagem; eles representam a base de conhecimento do classificador.

No entanto, a imagem à esquerda exibe um elemento de classe desconhecida, representado na cor preta. Os classificadores, baseados nesse tipo de aprendizagem, são capazes de classificar esse elemento a partir das informações sobre as características dos elementos conhecidos (já classificados). A imagem à direita apresenta esse mesmo objeto, após o processo de classificação, definido em uma das classes.

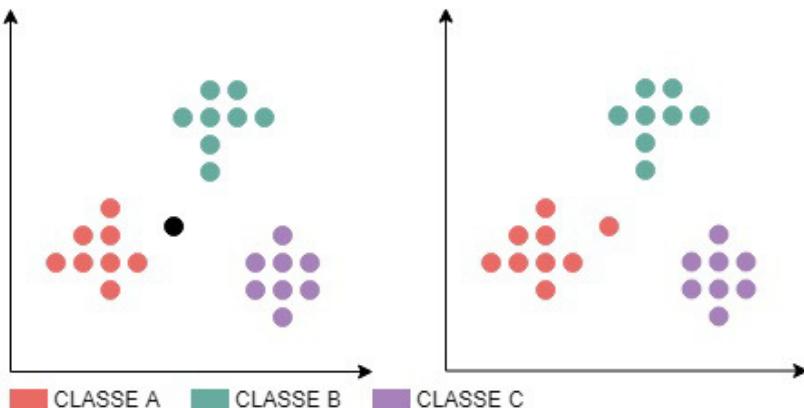


Figura 11.2: Aprendizagem supervisionada

Aprendizagem não supervisionada

Assim como na aprendizagem supervisionada, esse método necessita de uma lista de características de diversos objetos para o treinamento. A diferença é que as classificações desses objetos não precisam ser apresentadas, descartando a necessidade de um supervisor.

Os classificadores baseados em aprendizagem não supervisionada são capazes de identificar classes por si só, considerando o quão semelhantes são as características dos objetos. Consequentemente, esse método pode ser utilizado também para descobrir novos padrões.

A figura a seguir ilustra o processo de aprendizagem não supervisionada. Observe na imagem à esquerda que objetos desconhecidos, sem classe definida, estão representados em um espaço de características bidimensional. Os classificadores, baseados nesse tipo de aprendizagem, são capazes de definir esses

objetos em classes distintas, agrupando os que possuem características semelhantes.

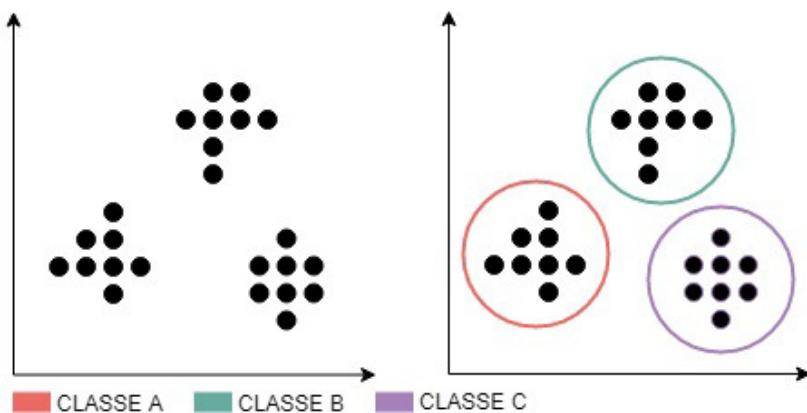


Figura 11.3: Aprendizagem não supervisionada

Um algoritmo de classificação baseado em aprendizagem supervisionada e frequentemente usado em sistemas de visão computacional é o *K-Nearest Neighbors* (K-NN). No próximo capítulo, ele será discutido do ponto de vista prático e teórico.

CAPÍTULO 12

CLASSIFICADOR K-NN

O algoritmo *K-Nearest Neighbors* (K-NN) – também conhecido como K Vizinhos Mais Próximos – é um dos algoritmos de classificação mais simples de ser compreendido. Por ser de fácil implementação e capaz de solucionar rapidamente problemas que envolvem reconhecimento de padrões, ele é amplamente utilizado.

Esse classificador opera por meio do método de aprendizagem supervisionada, ou seja, precisa ser previamente treinado por um especialista. Neste capítulo, o K-NN será discutido tanto do ponto de vista teórico quanto prático. No tópico a seguir, vamos detalhar sua lógica de funcionamento.

12.1 ALGORITMO K-NN

Com o propósito de demonstrar a flexibilidade do classificador K-NN e facilitar o entendimento desse algoritmo, considere o problema de classificação de peças baseado nas características de massa e área de cada uma delas. Nesse contexto, a figura a seguir ilustra a etapa de aprendizagem supervisionada desse classificador, na qual foram apresentados ao modelo 18 objetos distintos.

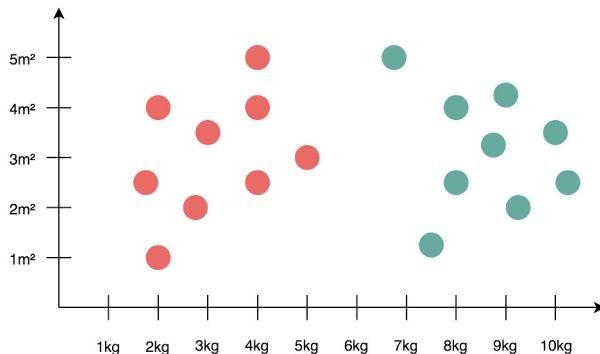


Figura 12.1: Etapa de aprendizagem

No esquema ilustrado, observe que as características de massa e área, de cada objeto apresentado, estão definidas em um espaço euclidiano bidimensional. Os objetos definidos à esquerda no espaço, na cor vermelha, indicam uma classe; os definidos à direita, na cor verde, indicam uma outra classe.

Note que os objetos da classe verde possuem um padrão de massa e área diferente dos objetos da classe vermelha. Portanto, eles encontram-se posicionados em regiões distintas e agrupados próximos aos seus semelhantes.

A representação dos objetos no espaço euclidiano, especificando a classe a que cada um pertence, exemplifica o treinamento do classificador. Após a etapa de aprendizagem, ilustrada na figura anterior, o classificador está pronto para reconhecer novos objetos.

A figura a seguir ilustra a etapa de classificação. Perceba que um novo objeto, representado na cor cinza e cuja classificação é desconhecida, foi inserido ao conjunto.

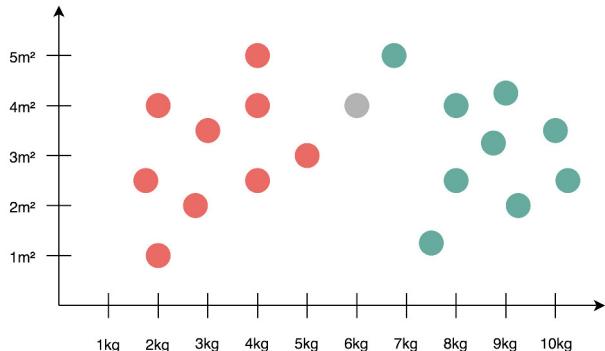


Figura 12.2: Etapa de classificação

Na figura anterior, observe que esse novo objeto possui valores de massa e área que podem nos confundir ao tentar classificá-lo como pertencente a uma das duas classes. Entretanto, esta questão pode ser rapidamente solucionada por meio do classificador K-NN.

Para classificá-lo, o K-NN calcula a distância euclidiana entre o objeto e os seus K vizinhos mais próximos. Considerando K igual a 5, a distância entre os 5 vizinhos mais próximos será verificada. A figura a seguir ilustra esse procedimento na etapa de classificação.

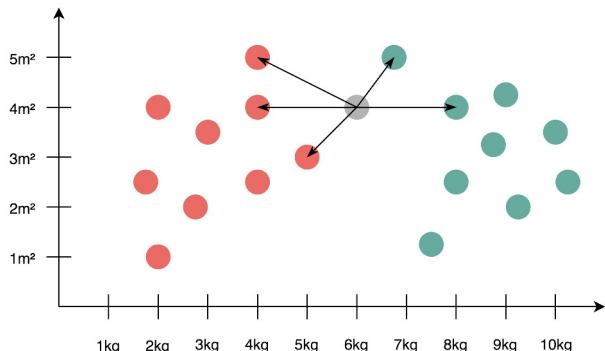


Figura 12.3: Distância entre objetos próximos

Observe que, dos cinco objetos mais próximos ao objeto não classificado, três pertencem à classe vermelha e dois à verde. A partir desse dado, o algoritmo K-NN define o objeto desconhecido como pertencente à classe vermelha, justamente por seus vizinhos mais próximos serem predominantemente pertencentes a essa classe.

Nesse exemplo, K foi considerado igual a 5, entretanto, K pode ser definido com qualquer valor inteiro positivo. Geralmente, são usados valores ímpares, pois qualquer valor par para K pode resultar em empate. Além disso, utilizamos somente duas características para classificar o objeto, todavia, não há limites para o número de características. Para representar mais de duas em um espaço euclidiano, serão necessárias mais dimensões.

Nesta seção, vimos a teoria sobre o funcionamento do classificador K-NN. Na seção a seguir, será exemplificado como utilizá-lo para solucionar um problema de classificação, por meio da linguagem Python e da biblioteca `scikit-learn`.

12.2 K-NN COM SCIKIT-LEARN

Um classificador K-NN pode ser implementado, com poucas linhas de código, pela biblioteca `scikit-learn`. Ela dispõe de métodos prontos para criar, treinar e validar esse classificador, inviabilizando o desenvolvimento de algoritmos específicos para executar essas tarefas. Nesta seção, a fim de solucionar um problema de classificação de objetos, vamos ver como aplicar cada um desses métodos para desenvolver e utilizar um classificador K-NN.

Conforme estudado no primeiro capítulo deste livro, a segmentação do objeto de interesse é uma das primeiras etapas comuns aos sistemas de Visão Computacional. Nessa etapa, o objeto de interesse (o qual desejamos classificar) será segmentado para que, em seguida, possamos extrair suas características. No tópico a seguir, este processo será exemplificado.

Segmentação do objeto de interesse

Com o intuito de criar o conjunto de dados usados na etapa de aprendizagem, devemos segmentar objetos distintos com classes conhecidas. A figura a seguir apresenta três imagens que representam objetos de uma classe, denominada Classe A.

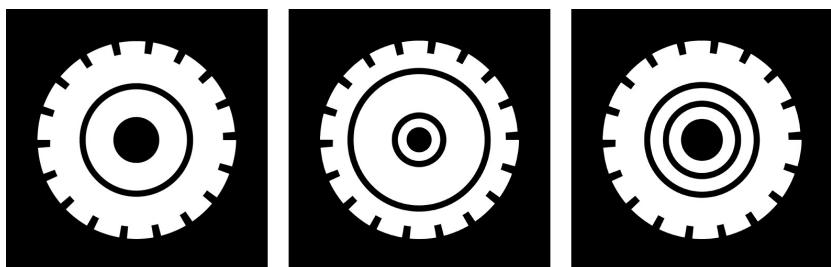


Figura 12.4: Objetos da Classe A

Para a classe de objetos apresentados nessa figura, seis imagens foram segmentadas. O mesmo procedimento foi realizado para uma segunda classe de objetos, denominada Classe B, apresentada na figura seguinte.

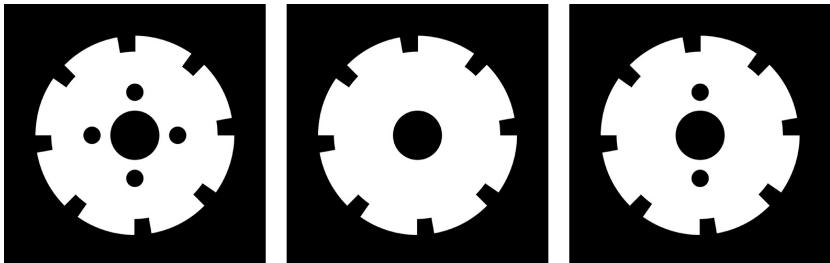


Figura 12.5: Objetos da Classe B

As características das doze imagens, seis referentes a objetos da Classe A e seis referentes da B, foram extraídas para o conjunto de dados de treinamento. Este assunto será abordado com detalhes e exemplificado no tópico a seguir.

Extração de características

Inúmeras características desses objetos poderiam ser usadas para classificá-los. Uma delas é a quantidade de dentes que compõe as engrenagens de cada classe. Observe que as engrenagens da Classe A possuem 17 dentes, diferente das da Classe B, que possuem 7 dentes.

A priori, mesmo parecendo uma característica satisfatória para classificá-las, contabilizar o número de dentes pode não ser uma tarefa trivial. As características referentes aos momentos invariantes de Hu podem representá-las melhor segundo vários aspectos, por exemplo, área e centro geométrico. Além disso, essas características são invariantes à escala e à rotação, facilitando a classificação do objeto. Por esse motivo, os momentos invariantes de Hu foram extraídos de cada objeto segmentado.

A tabela a seguir apresenta os sete momentos extraídos de cada

imagem. Os arquivos cujo nome iniciam com a letra A referem-se a fotografias de objetos da Classe A, bem como os que iniciam com B pertencem à Classe B.

Objeto	M1	M2	M3	M4	M5	M6	M7
A1.jpeg	3.10	11.98	14.68	16.86	-32.64	22.85	33.25
A2.jpeg	3.11	12.36	14.63	17.06	32.99	23.34	33.16
A3.jpeg	3.07	12.26	14.62	15.32	31.4	-22.69	-30.30
A4.jpeg	3.10	11.98	14.68	16.86	-32.64	22.85	33.25
A5.jpeg	3.11	12.36	14.63	17.06	32.99	23.34	33.16
A6.jpeg	3.07	12.26	14.62	15.32	31.4	-22.69	-30.30
B1.jpeg	3.12	11.72	14.76	15.33	32.08	21.40	30.37
B2.jpeg	3.14	11.78	14.83	15.59	-34.29	21.71	30.81
B3.jpeg	3.13	10.43	14.77	15.44	31.55	20.75	30.54
B4.jpeg	3.12	11.72	14.76	15.33	32.08	21.40	30.37
B5.jpeg	3.14	11.78	14.83	15.59	-34.29	21.71	30.81
B6.jpeg	3.13	10.43	14.77	15.44	31.55	20.75	30.54

Para o problema em questão, seis imagens de objetos de cada classe são suficientes para treinar o classificador. Em soluções mais complexas, com mais classes e características mais sutis, quanto maior for o conjunto de treinamento, mais eficiente será o classificador. No próximo tópico, veremos como criar e realizar o treinamento de um classificador K-NN.

Treinamento do classificador

O primeiro passo para criar um classificador K-NN é importar a biblioteca `scikit-learn`. Em seguida, criar uma lista contendo

as características dos objetos para a etapa de treinamento. Cada característica nessa lista é representada em uma outra lista interna, nesse exemplo, contendo sete elementos. O código a seguir exemplifica a importação da biblioteca e a definição da lista de características.

```
from sklearn.neighbors import KNeighborsClassifier

características = [
[3.10, 11.98, 14.68, 16.86, -32.64, 22.85, 33.25],
[3.11, 12.36, 14.63, 17.06, 32.99, 23.34, 33.16],
[3.07, 12.26, 14.62, 15.32, 31.40, -22.69, -30.30],
[3.10, 11.98, 14.68, 16.86, -32.64, 22.85, 33.25],
[3.11, 12.36, 14.63, 17.06, 32.99, 23.34, 33.16],
[3.07, 12.26, 14.62, 15.32, 31.40, -22.69, -30.30],
[3.12, 11.72, 14.76, 15.33, 32.08, 21.40, 30.37],
[3.14, 11.78, 14.83, 15.59, -34.29, 21.71, 30.81],
[3.13, 10.43, 14.77, 15.44, 31.55, 20.75, 30.54],
[3.12, 11.72, 14.76, 15.33, 32.08, 21.40, 30.37],
[3.14, 11.78, 14.83, 15.59, -34.29, 21.71, 30.81],
[3.13, 10.43, 14.77, 15.44, 31.55, 20.75, 30.54]
]
```

Uma lista contendo a classificação desses objetos também precisa ser criada. Os valores que indicam a classe dos objetos, na lista de classificações, devem obedecer à ordem da lista de características. Dessa maneira, o primeiro elemento da lista de classificações indica a classe do primeiro objeto da lista de características.

Estamos trabalhando com peças que podem ser classificadas em Classe A ou B. Na lista de classificações, o valor 0 representa a Classe A, e o valor 1, a Classe B. A linha de código exemplifica a definição da lista de características:

```
classificacoes = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Com os dados de treinamento definidos e organizados em

listas, a próxima etapa é criar o classificador. O método `KNeighborsClassifier`, da biblioteca `scikit-learn`, nos permite criar um classificador K-NN requerendo um único parâmetro. Esse parâmetro é justamente o valor de K, que define o número de vizinhos considerados.

Para obtermos bons resultados com poucos dados apresentados no conjunto de treinamento, não é recomendado trabalhar com valores de K superiores a 3. Relembrando a teoria estudada na seção anterior: a fim de evitar que empates ocorram, o valor de K deve ser preferencialmente ímpar. Nessa solução, K será definido como 3:

```
knn = KNeighborsClassifier(3)
```

O treinamento do classificador K-NN é realizado pelo método `fit`, que requer dois parâmetros: o primeiro deles é a lista de características, o segundo, a lista de classificações. A linha de código a seguir exemplifica a chamada ao método de treinamento.

```
knn.fit(caracteristicas, classificacoes)
```

A execução desse método treinará o classificador K-NN para reconhecer novos objetos, entretanto, antes de utilizá-lo, é importante verificar a precisão do classificador para a execução dessa tarefa. Esse assunto será detalhado a seguir.

Validação do treinamento

Assim como definimos um conjunto de dados para a etapa de treinamento, outro conjunto similar pode ser definido para a etapa de validação. Esta consiste em analisar a precisão do classificador para reconhecer novos objetos.

A validação pode ser realizada por meio do método `score` da biblioteca `scikit-learn`. Esse método requer dois parâmetros: o primeiro é a lista de características que será utilizada nesse procedimento, o segundo, a lista de classificações.

A execução desse método compara a resposta esperada, indicada na lista de classificações, com a resposta fornecida pelo classificador. O resultado retornado é um valor que varia de 0 a 1, indicando a porcentagem de acerto do classificador. O valor 1 indica 100% de acerto, já valor 0 indica o oposto. A linha de código a seguir exemplifica o uso do método `score`:

```
print knn.score(caracteristicas, classificacoes)
```

O resultado dessa execução será:

```
1.0  
[Finished in 0.5s]
```

A execução do método `score` retornou o valor 1, indicando que a classificação correta foi definida para todos os objetos testados. O valor 1 foi retornado, pois estamos trabalhando com um número muito pequeno de amostras. Em um cenário real, podemos assumir que, para qualquer valor maior que 0.8, o classificador apresenta uma assertividade satisfatória.

No tópico a seguir, será demonstrado como a classificação de um objeto desconhecido pode ser indicada pelo K-NN.

Classificação de um objeto

A classificação de um objeto desconhecido pode ser obtida por meio do método `predict`, da biblioteca `scikit-learn`. Esse método requer como parâmetro apenas a lista de características do

objeto. O resultado retornado por ele é a classe à qual o objeto pertence.

O código a seguir exemplifica esse procedimento:

```
objetoDesconhecido = [  
    3.17, 11.84, 14.91, 16.22, -33.21, 22.38, 31.78  
]  
  
print knn.predict(objetoDesconhecido)
```

O resultado dessa execução será:

```
[1]  
[Finished in 0.5s]
```

O valor 1, retornado pela execução do código, indica a classificação do objeto desconhecido. Na etapa de treinamento, esse valor foi definido para representar a Classe B e, consequentemente, esse objeto pertence a essa classe.

Na figura seguinte, um mesmo modelo de engrenagem é apresentado em duas imagens distintas. No exemplo a seguir, a fim de determinar a classificação desses objetos, estas imagens serão submetidas ao classificador K-NN.

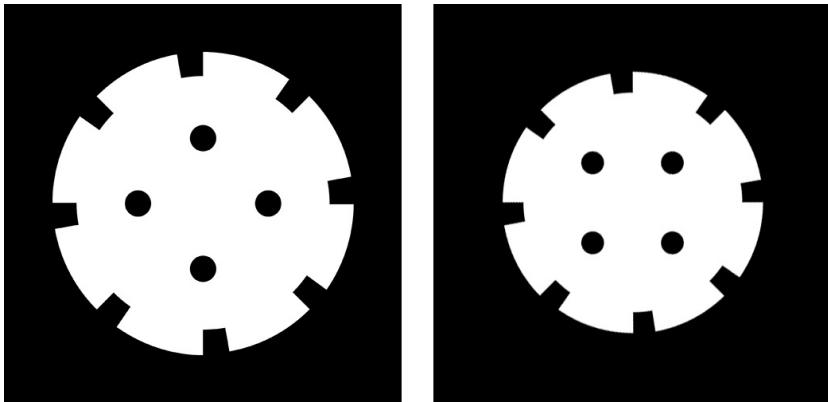


Figura 12.6: Objetos desconhecidos

```
knn = KNeighborsClassifier(3)
knn.fit(caracteristicas, classificacoes)

objetos = [
[3.17, 11.84, 14.91, 16.22, -33.21, 22.38, 31.78],
[3.17, 11.82, 14.91, 16.21, -33.38, 22.33, 31.77]
]

print knn.predict(objetos)
```

O resultado da execução será:

```
[1 1]
[Finished in 0.5s]
```

Considerando as características dos objetos representadas nas duas imagens, o classificador K-NN definiu-os como pertencentes à Classe B. Observe que uma lista de objetos também pode ser definida como parâmetro para o método `predict`, dessa forma, a classificação para cada um deles será retornada em um vetor.

Para facilitar o entendimento, o código exemplificado nesta seção foi apresentado em partes. Contudo, se você quiser conferi-lo com todas essas partes unificadas, ele está disponível no

repositório no GitHub, em <https://github.com/felipecbarelli/livro-visao-computacional>.

Neste capítulo, a última etapa de um sistema de Visão Computacional foi detalhada e exemplificada. A partir de então, você já está capacitado para desenvolver sistemas capazes de enxergar e interagir com o ambiente ao seu redor.

Para pôr em prática os conhecimentos adquiridos até então, uma sugestão seria desenvolver um sistema de Visão Computacional capaz de identificar gestos. A mesma lógica para classificar peças, estudada neste capítulo, pode ser empregada na resolução deste problema. A figura a seguir ilustra como a segmentação dos gestos, considerando a mão como objeto de interesse, pode ser realizada.



Figura 12.7: Segmentação dos gestos

No próximo capítulo, veremos um algoritmo, nativo da biblioteca OpenCV, capaz de detectar objetos em imagens.

CAPÍTULO 13

ALGORITMO HAAR CASCADE

Desenvolvido por Paul Viola e Michael Jones em 2001, o Haar Cascade é um eficiente algoritmo de busca de objetos em imagens. Denominado originalmente como algoritmo de Viola-Jones, ele utiliza uma abordagem baseada em aprendizado de máquina, no qual são usadas imagens positivas e negativas para o treinamento. Imagens positivas são imagens que contêm o objeto de interesse representado; do contrário, as negativas são as que não contêm o objeto representado.

Neste capítulo, o funcionamento do Haar Cascade será apresentado em alto nível, sem que detalhes matemáticos sobre sua implementação sejam abordados. Em seguida, será demonstrado como utilizá-lo, na prática, através de métodos nativos da biblioteca OpenCV.

Para que o conceito de imagens positivas e negativas fique ainda mais claro, imagine que o Haar Cascade seja usado para detectar veículos por meio de imagens capturadas pela câmera de um radar. Nesse cenário, será necessário um conjunto de imagens de veículos transitando na estrada (imagens positivas), bem como um conjunto de imagens da estrada vazia, sem nenhum veículo

transitando (imagens negativas).

Na figura a seguir, a imagem à esquerda representa uma imagem positiva, e a imagem à direita, uma negativa:



Figura 13.1: Imagem positiva e negativa

Com o propósito de extrair características do objeto de interesse, o Haar Cascade utiliza máscaras denominadas *Haar Features*. Estas funcionam como filtros que percorrem um conjunto de imagens, que representam o objeto de interesse segmentado, efetuando operações para extraír características do objeto. A figura a seguir ilustra algumas dessas máscaras.

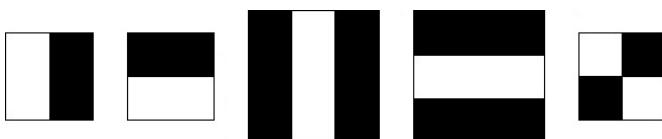


Figura 13.2: Haar Features

A figura seguinte mostra a convolução a partir das máscaras apresentadas na figura anterior, do processo de extração de

características em imagens segmentadas dos veículos.

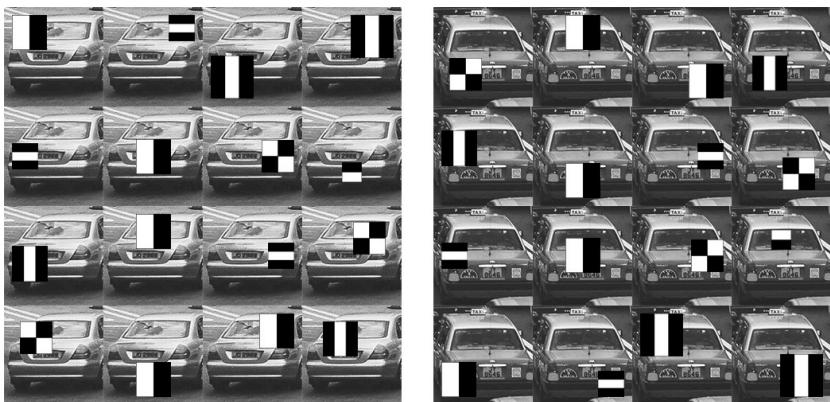


Figura 13.3: Extração de características

Observe que diferentes máscaras sobrepõem o objeto de interesse. Dessa maneira, cada uma delas efetua a subtração da soma dos pixels sobrepostos pelo retângulo branco da soma dos pixels sobrepostos pelo retângulo preto. O resultado dessa operação é uma característica do objeto representado na imagem.

Na prática, o que determina as características do objeto é a variação de luminosidade, principalmente das regiões de borda que apresentam variações abruptas na intensidade do tom de cinza dos pixels. As máscaras apresentadas na figura anterior capturaram essas variações em diferentes amplitudes e direções, permitindo que o objeto seja detectado de forma ágil e precisa.

Com as características extraídas de um objeto, a partir de várias imagens de objetos pertencentes à mesma classe, a detecção deste objeto também é realizada através das máscaras. Elas percorrem cada imagem tentando localizar regiões que caracterizam o objeto. Na figura a seguir, a imagem à esquerda ilustra o processamento de

uma máscara em uma determina região da cena capturada. A imagem à direita apresenta os objetos detectados pelo algoritmo Haar Cascade.

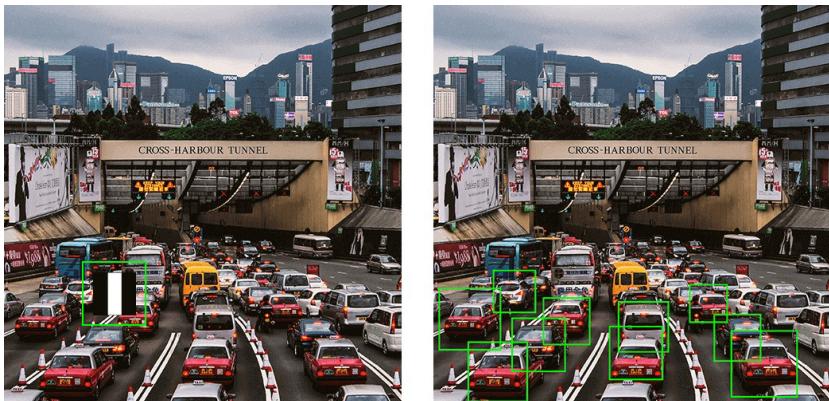


Figura 13.4: Algoritmo Haar Cascade

Uma vez que a variação da luminosidade determina as características dos objetos representados na imagem, melhores resultados serão obtidos em ambientes com iluminação controlada, onde não exista variação da luz sobre os objetos fotografados ou filmados.

IMPORTANTE

É válido ressaltar que o Haar Cascade não é um classificador, e sim um algoritmo para extração de características e detecção de objetos em imagens. Diferentes classificadores são utilizados pelo Haar Cascade nesse procedimento.

A biblioteca OpenCV possui métodos nativos que nos permitem usar o Haar Cascade de forma simples e objetiva. Na próxima seção, será exemplificado como é possível detectar faces humanas em uma imagem através desses métodos.

13.1 DETECÇÃO DE OBJETOS

Para que possamos identificar faces humanas representadas em uma imagem, precisamos do arquivo Haar Cascade. Ele contém os valores que representam as características do objeto. Os dados armazenados nesse arquivo são formatados em XML, e alguns exemplos podem ser obtidos no repositório: <https://git.io/vNjTs>.

Esse repositório contém arquivos que nos permitem detectar diferentes objetos, por exemplo, faces, olhos, bocas, parte superior ou inferior de um corpo humano etc. No exemplo a seguir, será demonstrado como detectar faces humanas e, para isto, o arquivo `haarcascade_frontalface_default.xml` será utilizado.

Para facilitar, no código apresentado no exemplo, este arquivo foi renomeado para `frontalface.xml`. Esse mesmo código apresenta como o arquivo de características deve ser carregado usando o método `CascadeClassifier`. Observe que basta indicá-lo como parâmetro no método.

```
import numpy as np
import cv2

# Carrega arquivos de características
cascadeFace = cv2.CascadeClassifier("frontalface.xml")

imagemOriginal = cv2.imread("selecao.jpg")
imagem = cv2.cvtColor(imagemOriginal, cv2.COLOR_BGR2GRAY)
```

Além do arquivo contendo as características do objeto de interesse, evidentemente precisamos carregar a imagem na qual as faces serão detectadas. A figura a seguir apresenta a imagem selecao.jpg utilizada neste exemplo.



Figura 13.5: Seleção brasileira. Foto: Lucas Figueiredo/CBF

Com o intuito de encontrar as faces presentes na imagem anterior, o método `detectMultiScale` deve ser usado. Ele requer quatro parâmetros obrigatórios. O primeiro deles é a imagem, convertida em tons de cinza, em que o objeto de interesse será detectado. O segundo, denominado `scaleFactor`, indica o fator de escala, um número qualquer maior que 1.0. Esse fator possibilita que a imagem carregada seja reduzida, facilitando a detecção do objeto.

O terceiro parâmetro, denominado `minNeighbors`, define a

quantidade mínima de vizinhos que cada candidato a retângulo (objeto de interesse detectado) deve possuir para ser, de fato, considerado um objeto de interesse. Para compreender melhor a importância desse parâmetro, observe a figura a seguir.

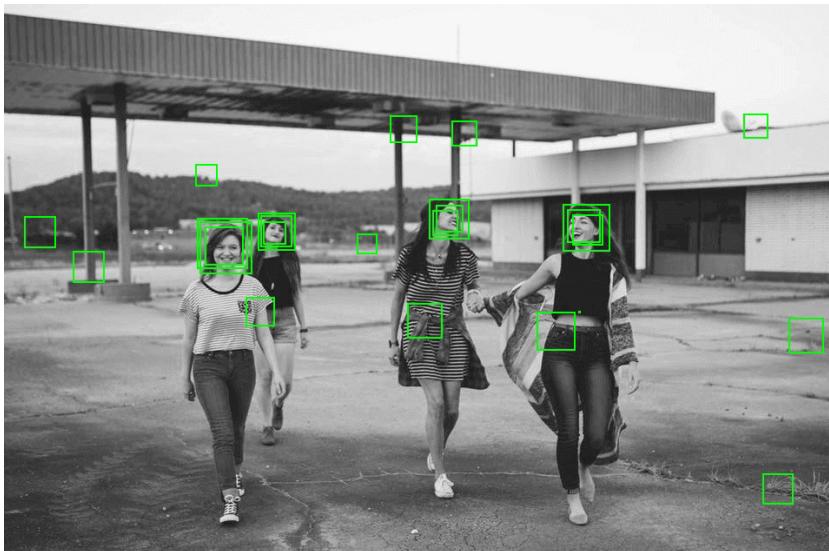


Figura 13.6: Objetos verdadeiros e falsos detectados

Na figura anterior, perceba que falsos objetos de interesse foram detectados, entretanto, os verdadeiros foram demarcados por mais de um retângulo. É justamente a quantidade mínima desses retângulos que é definida pelo parâmetro `minNeighbors`.

Na prática, ele tem o objetivo de evitar a ocorrência de falsos positivos, ou seja, impedir que objetos não desejados sejam detectados como objetos de interesse. Quanto maior o valor desse parâmetro, menor a chance de erro, porém, maior a chance de não serem detectados objetos de interesse reais.

O quarto parâmetro, denominado `minSize`, indica o tamanho mínimo do retângulo para que retângulos menores sejam ignorados. A tabela a seguir apresenta resumidamente cada um dos parâmetros citados.

Parâmetro	Descrição
<code>image</code>	Imagem em tons de cinza.
<code>scaleFactor</code>	Fator de escala para redução da imagem.
<code>minNeighbors</code>	Quantidade mínima de vizinhos.
<code>minSize</code>	Tamanho mínimo do retângulo que demarca o objeto.

O código exemplificado a seguir processa a imagem da seleção brasileira e detecta as faces representadas na fotografia.

```
import numpy as np
import cv2

cascadeFace = cv2.CascadeClassifier("frontalface.xml")

imagemOriginal = cv2.imread("selecao.jpg")
imagem = cv2.cvtColor(imagemOriginal, cv2.COLOR_BGR2GRAY)

faces = cascadeFace.detectMultiScale(imagem, 1.3, 5, (30,30))

# Desenha um retângulo nas faces detectadas
for (x,y,w,h) in faces:
    cv2.rectangle(imagemOriginal, (x,y), (x+w, y+h), (0,0,255), 2
)

# Exibe o total de faces detectadas
print len(faces)

cv2.imshow("Resultado", imagemOriginal)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

A figura a seguir apresenta o resultado da execução do código

anterior. Observe que a face de todos os jogadores foi detectada corretamente e indicada por um retângulo verde.

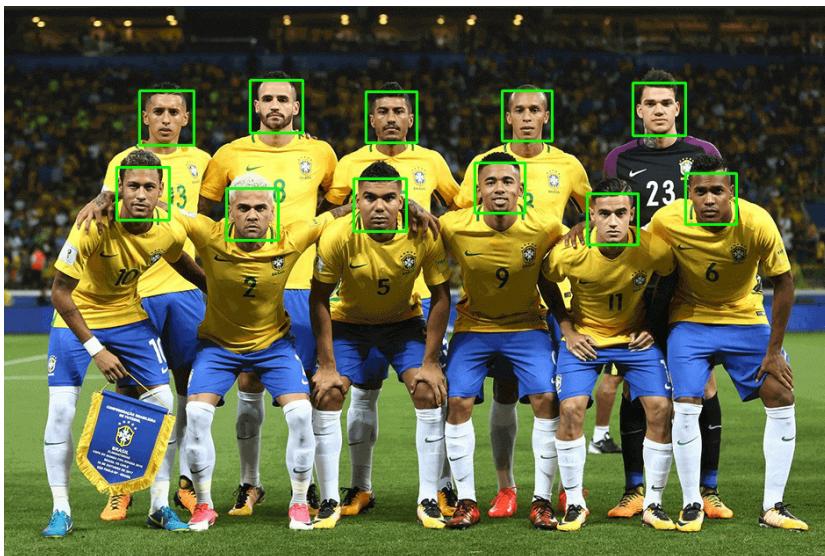


Figura 13.7: Seleção brasileira com faces detectadas

Assim como em outros métodos estudados neste livro, não há regras ou dicas para que os valores dos parâmetros do método `detectMultiScale` sejam definidos. Eles são definidos por tentativa e erro até que um resultado satisfatório seja encontrado para a detecção do objeto desejado.

Nesta seção, foi apresentado como detectar objetos em imagens a partir de arquivos de características já existentes. Caso você tenha interesse em aprender a criar arquivos com características extraídas de outros objetos, recomendo a leitura do tutorial escrito por Mahdi Rezaei, do departamento de Ciência da Computação da Universidade de Auckland, disponível em: <https://git.io/vAxeV>.

Diferentes bibliotecas e métodos para manipulação digital de imagem e reconhecimento de padrões foram apresentados nesta obra. Agora, após ter concluído a leitura de todo o conteúdo abordado, você está apto para desenvolver os primeiros sistemas de Visão Computacional.

No próximo capítulo, vamos exemplificar pequenos programas que demonstram aplicações básicas desse tipo de sistema. Estes podem ser utilizados como base para o desenvolvimento de aplicações mais complexas.

APLICAÇÕES DA VISÃO COMPUTACIONAL

Diferentes técnicas e métodos para processamento de imagens digitais foram apresentados até então, entretanto, os sistemas de Visão Computacional geralmente operam o processamento de imagens capturadas em tempo real ou em vídeos. Para exemplificar como essas técnicas podem ser aplicadas a um cenário real, este capítulo demonstra algoritmos que realizam tarefas comuns atribuídas a esses sistemas.

Todos os algoritmos exemplificados nas próximas seções operam processando imagens carregadas a partir de vídeos, simulando uma captura em tempo real. Mesmo assim, os códigos podem ser facilmente manipulados para processar imagens capturadas a partir de uma webcam.

Os vídeos usados nos exemplos estão disponíveis no repositório no GitHub, bem como os códigos dos programas. Caso você precise processar imagens capturadas a partir de uma webcam, o capítulo sobre aquisição de imagens detalha como implementar essa alteração.

Uma das diferenças entre trabalhar com um único arquivo de

imagem e sequências de imagens está relacionada ao tratamento de ruídos de oclusão de cena. Além disso, muitas vezes o objeto de interesse pode não estar presente na cena em um dado instante. Nessas situações, a tentativa de extrair características dele pode resultar na obtenção de dados incorretos, ou provocar a interrupção da execução do programa.

Para evitar essas complicações, estude os algoritmos apresentados nas próximas seções. O primeiro deles trata sobre reconhecimento de objetos e figuras geométricas planas, em imagens processadas a partir de um vídeo.

14.1 RECONHECIMENTO DE OBJETOS

Uma das abordagens para reconhecer objetos em imagens é extrair a quantidade de vértices que ele possui. Esta técnica pode ser aplicada com o intuito de reconhecer figuras geométricas em imagens. Nesta seção, será exemplificado como reconhecer figuras geométricas (que representam um quadrado, um triângulo ou um círculo) em frames de uma captura de vídeo.

A priori, pode parecer uma questão simples, já abordada no capítulo sobre extração de características. No entanto, o propósito desse exemplo é apresentar como os frames, que representam a troca de uma figura por outra, podem ser desconsiderados.

A figura seguinte apresenta dois frames distintos do mesmo vídeo processado, cada um deles representando um objeto diferente:



Figura 14.1: Reconhecimento de objetos

O código exemplificado a seguir demonstra como reconhecer um objeto, representado por uma figura geométrica, considerando o número de vértices que ele possui.

```
import cv2
import numpy as np

indiceFrame = totalVerticesAnterior = 0
valoresMedidos = np.zeros(7)
video = cv2.VideoCapture("formas-geometricas-480.mov")

while True:
    ret, frameRGB = video.read()
    frameCinza = cv2.cvtColor(frameRGB, cv2.COLOR_RGB2GRAY)
    tipo = cv2.THRESH_BINARY
    ret, frameBinarizado = cv2.threshold(frameCinza, 127, 255, tipo
)

    valorMedioAtual = int(cv2.mean(frameBinarizado)[0])

    if valorMedioAtual != 0:
        if valorMedioAtual == int(cv2.mean(valoresMedidos)[0]):
            contornos, hierarquia = cv2.findContours(
                frameBinarizado, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
            )
            objeto = contornos[0]
            perimetro = cv2.arcLength(objeto, True)
            poligono = cv2.approxPolyDP(objeto, 0.03 * perimetro, True)
            totalVertices = len(poligono)
```

```

if totalVertices != totalVerticesAnterior:
    totalVerticesAnterior = totalVertices
    if totalVertices == 3:   print "triangulo"
    elif totalVertices == 4: print "quadrado"
    elif totalVertices > 7:  print "circulo"

valoresMedidos[indiceFrame] = valorMedioAtual
if indiceFrame == 6:
    indiceFrame = -1

indiceFrame += 1
cv2.imshow("Video", frameRGB)
if cv2.waitKey(1) & 0xFF == ord("q"):
    break

video.release()
cv2.destroyAllWindows()

```

No código exemplificado, observe como os frames que não contêm objetos representados, bem como os que representam objetos ainda em movimento, foram desconsiderados. A fim de realizar este procedimento, primeiro o algoritmo verifica se a média dos valores dos pixels do frame processado é igual a zero, representando uma imagem completamente preta. Se essa condição for verdadeira, o frame é desconsiderado. Do contrário, se a condição indicar que há objetos representados, o algoritmo verifica se o frame atual é igual aos últimos sete frames processados.

Se essa condição for verdadeira, indicando que o objeto de interesse não está em movimento, o total de vértices do objeto é obtido; do contrário, o frame é desconsiderado. Sempre que um novo valor é obtido para o total de vértices, o nome da figura geométrica correspondente (que representa o objeto exibido) é apresentado ao usuário.

Repare que todos os procedimentos executados no algoritmo, assim como todos os métodos usados, foram estudados em capítulos anteriores. Na próxima seção, será demonstrado como reconhecer e contabilizar objetos considerando suas cores.

14.2 RECONHECIMENTO DE CORES

A cor predominante de um objeto é outra característica que pode ser utilizada para reconhecê-lo. Em ambientes industriais, é comum o uso de esteiras seletoras, equipadas com sistemas de Visão Computacional, usadas para contabilizar e armazenar objetos de cores diferentes em caixas distintas.

O procedimento de contabilizar objetos de cores variadas, transportados por uma esteira, será exemplificado no código apresentado nesta seção. Um vídeo que demonstra tampas de três cores distintas (vermelha, azul e verde) sendo transportadas será usado como exemplo.

A seguir, veja a figura que apresenta dois frames distintos desse vídeo, cada um deles demonstrando uma tampa diferente.

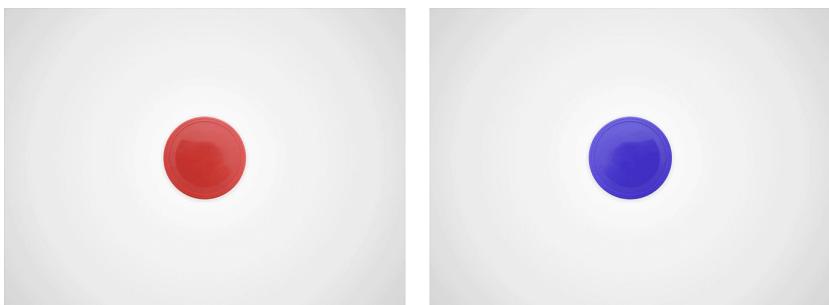


Figura 14.2: Reconhecimento de cores

O código exemplificado demonstra como contabilizar as tampas, exibidas no vídeo, considerando a cor de cada uma delas:

```
import cv2
import numpy as np

ultimaTela = indiceFrame = 0
totalAzul = totalVerde = totalRosa = 0
video = cv2.VideoCapture("objetos-coloridos-480.mov")

while True:
    indiceFrame += 1
    ret, frameRGB = video.read()
    valorMedioRGB = cv2.mean(frameRGB)
    maiorValor = npamax(valorMedioRGB)
    indiceCanal = np.argmax(valorMedioRGB)

    if indiceFrame == 30:
        indiceFrame = 0
        if valorMedioRGB[0] == valorMedioRGB[1]:
            ultimaTela = 0
        else:
            if ultimaTela == 0:
                ultimaTela = 1
                if indiceCanal == 0:
                    print "tampa azul"
                    totalAzul += 1
                elif indiceCanal == 1:
                    print "tampa verde"
                    totalVerde += 1
                elif indiceCanal == 2:
                    print "tampa rosa"
                    totalRosa += 1

    cv2.imshow("Video", frameRGB)
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

video.release()
cv2.destroyAllWindows()
```

No código anterior, observe que uma tampa só é contabilizada após o sistema verificar que não havia nenhuma delas no último

frame processado. Esse procedimento garante que uma mesma tampa não seja contabilizada mais que uma vez durante a sua passagem.

A variável `ultimaTela` é responsável por este controle e, quando definida como `0`, indica que não havia nenhum objeto no último frame; caso definida como `1`, indica o oposto. Sempre que uma tampa é identificada, ela é contabilizada e sua cor correspondente é exibida ao usuário.

Uma outra variável importante nesse algoritmo é a `indiceFrame`, responsável por controlar quando um frame será processado. É interessante que nem todos os frames sejam processados, pois, assim que um objeto entra cena (parcialmente representado), não há informações suficientes para que sua cor predominante seja definida. No código exemplificado, a cada 30 frames fornecidos pelo vídeo, apenas um é processado.

Na próxima seção, será demonstrado como contabilizar o total de objetos representados em um frame de uma captura.

14.3 CONTAGEM DE OBJETOS

Contabilizar objetos em imagens de um vídeo também é uma tarefa frequentemente atribuída aos sistemas de Visão Computacional. Este procedimento pode ser usado, por exemplo, para contabilizar o total de carros em uma garagem, ou o total de vagas disponíveis.

Nesta seção, será feito um algoritmo para desempenhar essa tarefa. Um vídeo que apresenta tampas coloridas, entrando e saindo de cena, será utilizado para isso. A figura a seguir apresenta

dois frames distintos desse vídeo, cada um deles representando uma quantidade diferente de tampas.

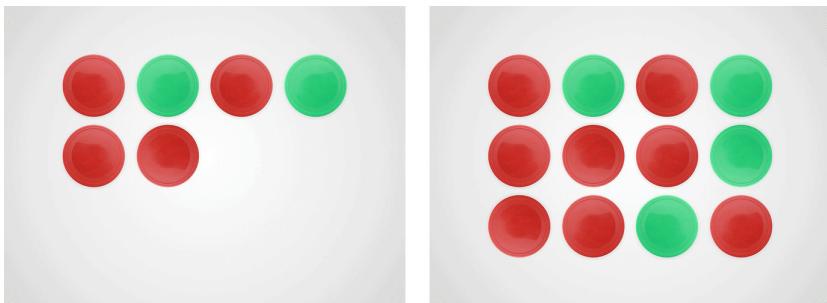


Figura 14.3: Contagem de objetos

O código exemplificado a seguir demonstra como contabilizar – desconsiderando a cor – o total de tampas representadas em cada instante na cena.

```
import cv2
import numpy as np

totalAnterior = 0
video = cv2.VideoCapture("contagem-de-objetos-480.mov")

while True:
    ret, frameRGB = video.read()
    frameCinza = cv2.cvtColor(frameRGB, cv2.COLOR_RGB2GRAY)
    tipo = cv2.THRESH_BINARY_INV
    ret, frameBinarizado = cv2.threshold(frameCinza, 200, 255, tipo
    )

    contornos, hierarquia = cv2.findContours(
        frameBinarizado, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
    )

    totalAtual = len(contornos)
    if totalAtual != totalAnterior:
        totalAnterior = totalAtual
        print totalAtual
```

```
cv2.imshow("Video", frameRGB)
if cv2.waitKey(1) & 0xFF == ord("q"):
    break

video.release()
cv2.destroyAllWindows()
```

No código exemplificado, observe que o total de objetos foi fornecido pelo método `findContours`, o mesmo estudado no capítulo sobre extração de características. Esse método nos retorna conjuntos de pontos que representam os contornos dos objetos detectados. O total de conjuntos indica o total de objetos na cena. Comparando o frame atual com o anterior, quando o total de objetos é alterado, o novo valor é exibido ao usuário.

Na próxima seção, será demonstrado como calcular a distância entre dois objetos, considerando o centro geométrico de cada um deles.

14.4 DISTÂNCIA ENTRE OBJETOS

A tarefa de calcular a distância entre dois pontos é atribuída constantemente aos sistemas de Visão Computacional. Um exemplo são os sistemas de controle de nível por imagem. Alguns deles obtêm o nível do reservatório pela distância entre uma boia, que flutua sobre o líquido armazenado, e a base do reservatório.

Neste exemplo, vamos ver como obter a distância em pixels entre dois objetos representados em um frame. O vídeo utilizado aqui apresenta duas tampas em cena, uma na cor verde e outra na cor vermelha. A tampa verde permanece parada durante toda a cena capturada; diferente da tampa vermelha, que se movimenta próxima à tampa verde. A figura a seguir apresenta dois frames

distintos do mesmo vídeo, demonstrando a variação da distância entre os objetos.

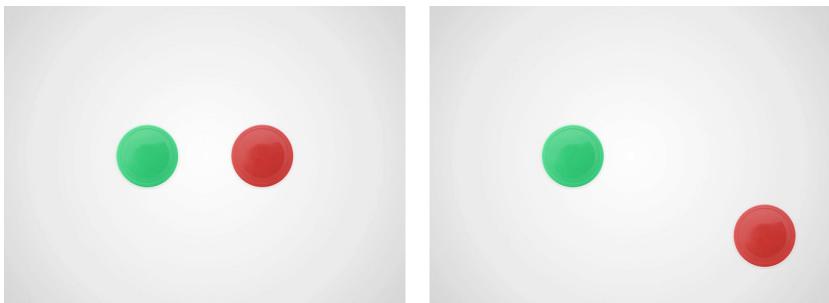


Figura 14.4: Movimentação do objeto

Considerando o centro geométrico de cada um dos dois objetos de interesse, o código seguinte demonstra como obter a distância entre esses dois pontos:

```
import cv2
import numpy as np

video = cv2.VideoCapture("movimentacao-de-objetos-480.mov")

while True:
    ret, frameRGB = video.read()
    frameCinza = cv2.cvtColor(frameRGB, cv2.COLOR_RGB2GRAY)
    tipo = cv2.THRESH_BINARY_INV
    ret, frameBinarizado = cv2.threshold(frameCinza, 200, 255, tipo
    )

    contornos, hierarquia = cv2.findContours(
        frameBinarizado, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
    )

    if len(contornos) == 2:
        momentosC1 = cv2.moments(contornos[0])
        momentosC2 = cv2.moments(contornos[1])
        C1x = momentosC1["m10"] / momentosC1["m00"]
        C1y = momentosC1["m01"] / momentosC1["m00"]
```

```

C2x = momentosC2["m10"] / momentosC2["m00"]
C2y = momentosC2["m01"] / momentosC2["m00"]

distancia = int(np.sqrt(((C2x - C1x)**2) + ((C2y - C1y)**2)))
print distancia

cv2.imshow("Video", frameRGB)
if cv2.waitKey(1) & 0xFF == ord("q"):
    break

video.release()
cv2.destroyAllWindows()

```

No código exemplificado, observe que os pontos referentes ao centro geométrico dos dois objetos são obtidos por meio dos momentos de cada um deles. Em cada frame processado, a distância entre esses dois pontos é calculada por uma equação matemática e, em seguida, o resultado é exibido ao usuário.

Os frames que possuem apenas um único objeto representado são desconsiderados, e o cálculo da distância é realizado apenas quando os dois objetos estão presentes na cena. A quantidade de objetos presentes nela é definida pelo total de contornos detectados pelo método `findContours`.

Na próxima seção, será demonstrado como acompanhar a movimentação de um objeto em capturas de vídeo.

14.5 RASTREAMENTO DE OBJETOS

Rastrear objetos em imagens é uma tarefa que possui uma infinidade de aplicações em sistemas de Visão Computacional. Ela pode ser aplicada, por exemplo, para rastrear carros em uma rodovia a fim de controlar o tráfego, ou rastrear a movimentação de um alvo visando acertá-lo com um projétil.

Existem inúmeras técnicas e algoritmos para rastrear objetos em vídeo. Alguns algoritmos apenas detectam a posição do objeto de interesse a cada frame processado, outros tentam predizer a posição do objeto no frame seguinte. Assim como existem inúmeros algoritmos, há também diferentes abordagens para detectar a movimentação do objeto.

Uma delas consiste na subtração de dois frames para eliminar o fundo da imagem, e foi apresentada no capítulo sobre pré-processamento de imagens. Aqui, será demonstrado como obter a posição de um objeto, identificado por sua cor, a cada frame processado.

O vídeo apresentado na seção anterior, que exibe duas tampas de cores diferentes, será usado novamente neste exemplo. O código demonstra como seguir e obter a posição da tampa vermelha em cada frame processado:

```
import cv2
import numpy as np

video = cv2.VideoCapture("movimentacao-de-objetos-480.mov")

while True:
    ret, frameRGB = video.read()
    frameHSV = cv2.cvtColor(frameRGB, cv2.COLOR_BGR2HSV)
    tomClaro = np.array([160, 100, 100])
    tomEscuro = np.array([200, 255, 255])
    frameSegmentado = cv2.inRange(frameHSV, tomClaro, tomEscuro)

    elementoEstruturante = np.ones((10, 10), np.uint8)
    frameSegmentado = cv2.morphologyEx(
        frameSegmentado, cv2.MORPH_CLOSE, elementoEstruturante
    )

    contornos, hierarquia = cv2.findContours(
        frameSegmentado, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
    )
```

```

if len(contornos) >= 1:
    objeto = contornos[0]
    momentos = cv2.moments(objeto)
    x = momentos["m10"] / momentos["m00"]
    y = momentos["m01"] / momentos["m00"]
    print("posicao: %d, %d" % (x, y))
    x, y, w, h = cv2.boundingRect(objeto)
    vertice1 = (x - 10, y - 10)
    vertice2 = (x + w + 10, y + h + 10)
    cv2.rectangle(
        frameRGB, vertice1, vertice2, (0, 255, 0), 2
    )

cv2.imshow("Video RGB", frameRGB)
if cv2.waitKey(1) & 0xFF == ord("q"):
    break

video.release()
cv2.destroyAllWindows()

```

Nesse código, observe que, a cada frame processado, a tampa na cor vermelha é segmentada e, em seguida, uma borda verde é inserida na imagem para destaca-la. Assim como no código exemplificado na seção anterior, a posição do objeto observado é definida a cada frame por seu centro geométrico.

A figura seguinte ilustra dois frames processados pelo código anterior. Repare que a borda verde segue o objeto em toda a sua trajetória.

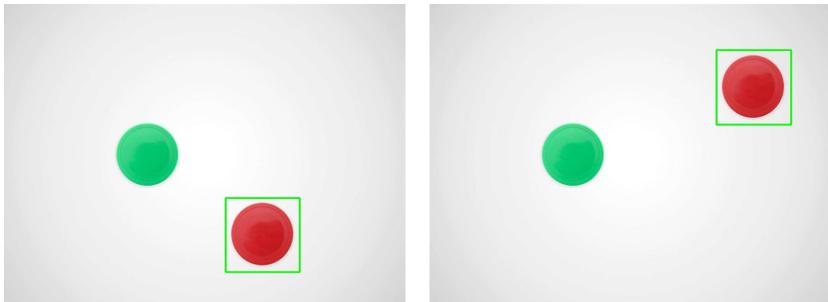


Figura 14.5: Rastreamento do objeto

Armazenando a posição do objeto a cada frame processado em uma lista ou em um vetor, você pode implementar uma rotina para obter a direção para qual o objeto está se movendo. Além disso, é possível obter também a distância em pixels que ele se movimentou em um dado instante. Essas duas tarefas serão sugeridas como exercícios de fixação na última seção. Esses algoritmos estão disponíveis no GitHub para consulta.

Nesta seção, foi apresentado um dos métodos mais simples de rastrear um objeto em uma captura de vídeo, realizando sua detecção frame a frame. Existem outros mais eficientes e complexos como o Meanshift, o Camshift e o TLD. Tanto no Meanshift quanto no Camshift, o rastreamento ocorre pela forma e aparência do objeto.

O TLD é um algoritmo capaz de rastrear, aprender e detectar um objeto, sua sigla significa: *Track, Learn and Detect* – justamente o que ele se propõe a fazer. Esse algoritmo é capaz de aprender em tempo real a aparência do objeto, para então detectá-lo e rastreá-lo.

Na próxima seção, será exemplificado como sistemas de Visão

Computacional, capazes de reconhecer caracteres, conseguem ler textos em imagens.

14.6 RECONHECIMENTO DE CARACTERES

Em sistemas de Visão Computacional, os algoritmos de reconhecimento de caracteres permitem que a entrada de dados (formatados em texto) seja automatizada. Um exemplo no qual essa tecnologia pode ser aplicada são os sistemas de controle de entrada e saída de veículos.

Alguns desses sistemas são capazes de realizar automaticamente a leitura das placas dos automóveis, descartando a mão de obra humana para realizar essa tarefa. Outros exemplos de aplicações são os sistemas capazes de ler automaticamente displays de balanças, multímetros e termômetros, sem que esses equipamentos estejam conectados ao sistema.

Desenvolver algoritmos para reconhecimento de caracteres pode ser um tanto quanto trabalhoso. Felizmente, existe uma biblioteca chamada `Python-tesseract`, que torna essa tarefa extremamente simples de ser realizada.

O método `image_to_string` dessa biblioteca retorna em uma string todos os caracteres detectados em uma imagem. O único parâmetro requerido por esse método é justamente a imagem desejada. Para utilizar a `Python-tesseract`, é necessário a importação da biblioteca `Pillow`, uma outra biblioteca para manipulação de imagens.

O código exemplificado aqui demonstra como realizar a leitura de caracteres exibidos em um vídeo. A figura a seguir apresenta

dois frames do vídeo que será usado no exemplo. Em cada um deles, um valor numérico está representado em seis caracteres.

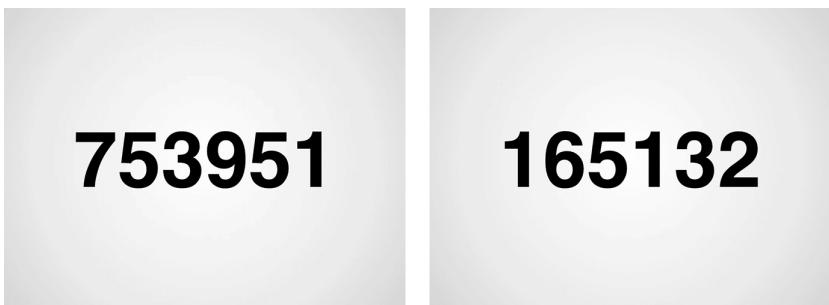


Figura 14.6: Reconhecimento de caracteres

O código exemplificado demonstra como utilizar o método `image_to_string` para obter os valores exibidos no vídeo.

```
from PIL import Image
import cv2
import pytesseract

indiceFrame = 0
valorAnterior = 0
video = cv2.VideoCapture("reconhecimento-de-caracteres-480.mov")

while True:
    indiceFrame += 1
    ret, frameRGB = video.read()
    imagem = Image.fromarray(frameRGB)

    if indiceFrame == 15:
        indiceFrame = 0
        valorAtual = pytesseract.image_to_string(imagem)
        if valorAtual != valorAnterior:
            valorAnterior = valorAtual
            print valorAtual

    cv2.imshow("Video", frameRGB)
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break
```

```
video.release()  
cv2.destroyAllWindows()
```

Nesse código, observe que uma variável chamada `indiceFrame` foi declarada. Esta é utilizada para controlar a frequência que a leitura será realizada. Uma vez que cada número exibido no vídeo permanece em cena durante dois segundos, não há necessidade de processar todos os frames. Nesse exemplo, a cada 15 frames exibidos, apenas um é processado a fim de obter o texto representado.

Esta seção finaliza o estudo sobre Visão Computacional e Processamento de Imagens guiado por esta obra. A partir de todo o conhecimento adquirido no decorrer da leitura, você está apto para desenvolver os primeiros sistemas de Visão Computacional.

Os exemplos apresentados aqui podem ser utilizados como base para o desenvolvimento dos seus primeiros programas. Todos os algoritmos, imagens e vídeos apresentados estão disponíveis no GitHub: <https://github.com/felipecbarelli/livro-visao-computacional>. Caso tenha sugestões de otimização ou melhoria para os códigos, não deixe de fazer um *pull-request* lá.

CAPÍTULO 15

REFERÊNCIAS BIBLIOGRÁFICAS

Livros e artigos

ALBUQUERQUE, Marcio Portes de; ALBUQUERQUE, Marcelo Portes de; CHACON, Germano Teixeira; GASTARDELLI, Elton; MORAES, Fernanda D.; OLIVEIRA, Gabriel. *Aplicação da técnica de momentos invariantes no reconhecimento de padrões em imagens digitais*. Centro Brasileiro de Pesquisas Físicas, Out. 2011. Disponível em: <http://www.cbpf.br/~chacon/mhu/>.

BACKES, André; SÁ JUNIOR, Jarbas. *Introdução à Visão Computacional usando MATLAB*. Alta Books, 2016.

BALLARD, Dana H.; BROWN, Christopher M. *Computer Vision*. Prentice Hall, 1982

GONZALEZ, Rafael; WOODS, Richard. *Processamento digital de imagens*. Pearson, 2007.

REDDY, Kishore K.; SOLMAZ, Berkan; YAN, Pingkun; AVGEROPOULOS, Nicholas G.; RIPPE, David J.; SHAH, Mubarak. *Confidence guided enhancing brain tumor segmentation*

in multi-parametric MRI. 2012. Disponível em:
http://crcv.ucf.edu/papers/ISBI_2012.pdf.

SOLOMON, Chris; BRECKON, Toby. *Fundamentos de processamento digital de imagens*. Grupo GEN, 2013.

Outras referências

Documentação oficial da biblioteca OpenCV

- <http://docs.opencv.org/master>
- <http://opencv-python-tutroals.readthedocs.io>