Trabalho 2

Processamento Digital de Imagem

Ieremies Vieira da Fonseca Romero

Introdução

O processamento digital de imagens é uma área de estudo fundamental em Ciência da Computação, que visa o tratamento e manipulação de imagens digitais por meio de técnicas e algoritmos computacionais. A análise e processamento de imagens digitais é uma tarefa complexa e multidisciplinar, que envolve conhecimentos em matemática, estatística, computação gráfica, processamento de sinais e outras áreas correlatas.

O objetivo deste trabalho é apresentar uma série de processamentos básicos em imagens digitais, abordando desde a leitura e escrita de imagens até operações de filtragem. Para o desenvolvimento das operações, será empregada a vetorização de comandos, que permite processar as imagens de forma mais eficiente e rápida, através da aplicação de operações vetoriais em vez de operações matriciais.

O programa

Neste trabalho, utilizamos as bibliotecas `numpy` 1.24.1 e OpenCV 4.7.0.72 utilizado via `cv2`.

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

Há duas formas de utilizar o código deste projeto:

script.py Executa todas as funções em todas as imagens e salva o seus respectivos resultados.

iterativo utilizando `python -i funcs.py`, é possível interagir com cada uma das funções no terminal.

Cada uma das questões do enúnciado é resolvido por uma função contida no arquivo `funcs.py`. Todas as funções possuem como primeiro parâmetro a imagem ou janela sob a qual ela irá atuar.

Para leitura e escrita das imagens, utilizaremos as seguintes funções do `cv2`.

```
# read the pgm image
cv2.imread('in.pgm', -1) # TODO why -1?
cv2.imwrite('out.pgm', img)
```

Para criar os histogramas, utilizaremos função `histogram` da biblioteca numpy.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('img/sonnet.pgm', -1)
hist, _ = np.histogram(img, bins=256, range=[0, 255])
plt.hist(hist, bins=256, range=[0,255], histtype='stepfill
plt.show()
```

Global e Otsu

Os primeiros dois métodos que apresentaremos são chamados de globais. Estes utilizam o mesmo limiar para toda a imagem.

O método global recebe um certo valor de limiar, (`threshold`, por padrão 128).

```
def lin_global(img, threshold=128):
    # todos os pixels começam com preto
    res = np.zeros_like(img)
    # pixels com valor abaixo do limiar
    # são definidos como branco
    res[img < threshold] = 255
    return res
```

Figura 1: Método global com limiar padrão (128) aplicado à imagem "baboon.pgm" com 53.85% dos pixels em preto.

Figura 2: Método de Otsu aplicado à imagem "baboon.pgm" com 40.89% dos pixels em preto.

Já para o método de Otsu calcula o limiar baseado na variância intraclasse. Neste caso, utilizaremos a função `cv2.threshold` com método `cv2.THRESH_OTSU`.

```
def otsu(img):
    _, res = cv2.threshold(img, 0, 255,
                           cv2.THRESH_BINARY
                           + cv2.THRESH_OTSU)
    return res
```

Local

Todas as funções de limiarização locais realizam um processo similar:

- Percorrer a imagem selecionando as "janelas" de um tamanho específico (por padrão, 25).
- Para cada janela, determinar o limiar via algum cálculo, a depender do método (passado pelo parâmetro `f`)
- O valor do pixel central da janela é determinado em comparação do limiar do passo anterior.

Assim, podemos resolver os métodos de limiarização locais utilizando a função `local` descrita a seguir.

```
def local(img, f, window_size=25):
    # dimensões da imagem original
    height, width = img.shape
    # tamanho da janela
    w = window_size // 2
    # nova matriz para o resultado
    res = np.zeros_like(img)

    for i in range(height):
        for j in range(width):
            # limites da janela deslizante
            i_min = max(i - w, 0)
            i_max = min(i + w + 1, height)
            j_min = max(j - w, 0)
            j_max = min(j + w + 1, width)

            # A janela em questão
            window = img[i_min:i_max,
                         j_min:j_max]
```

```
            # Utiliza a função indicada
            # para calcular o limiar
            threshold = f(window)

            # em todos os casos, se o
            # valor do pixel é menor
            # que o limiar, ele é co-
            # lorido de branco
            if img[i,j] < threshold:
                res[i,j] = 255
    return res
```

Assim, é possível utilizar a função invocando apenas `local(img, bernsen)`.

É importante perceber que, diferentemente do Trabalho 1, quando estamos tratando de limiarização, não se faz necessário realizar o "padding" para as bordas. Nesse caso, utilizamos apenas janelas que cabem inteiramente na imagem.

Figura 3: Método de Bernsen aplicado à imagem "baboon.pgm" com 52.01% dos pixels em preto.

Bernsen

O método de Bernsen utiliza o contraste dentro da janela para determinar o limiar. Podemos calculá-lo a partir da média entre o maior e o menor valores na janela.

```
def bernsen(window):
    # valor máximo e mínimo
    # dentro da janela
    max_val = int(np.max(window))
    min_val = int(np.min(window))

    return (max_val + min_val) // 2
```

É importante pontuar nesse momento o uso da função `int`. Esta é utilizada para indicar ao python o uso de um valor inteiro e impedir o *overflow* que ocorreria na soma do `return`. Esse processo irá se repetir daqui pra frente

Niblack e Sauvola e Pietaksinen

O método de Niblack utiliza as métricas estatísticas de média e desvio padrão para calular o limiar baseado na fórmula $\mu(x,y)+k\sigma(x,y)$, onde $\mu(x,y)$ é a média das intensidades na janela e $\sigma(x,y)$ o desvio padrão nela. Assim, utilizando as funções `numpy.mean` e `numpy.std`, podemos calculá-lo da seguinte forma.

```
def niblack(window, k=-0.2):
    # obter o valor médio e o desvio
    # padrão dentro da janela
    mean_val = int(np.mean(window))
    std_dev = int(np.std(window))

    # calcular o limiar usando a
    # fórmula de Niblack
    return mean_val + k * std_dev
```

Já o método de Sauvola e Pietaksinen utilizam uma ideia similar mas com uma fómula diferente que tenta compensar a má iluminação de documentos:

$$\mu(x,y)\left[1+k\left(\frac{\sigma(x,y)}{R}-1\right)\right]$$

```
def sauvola(window, k=0.5, R=128):
    mean_val = int(np.mean(window))
    std_dev = int(np.std(window))
```

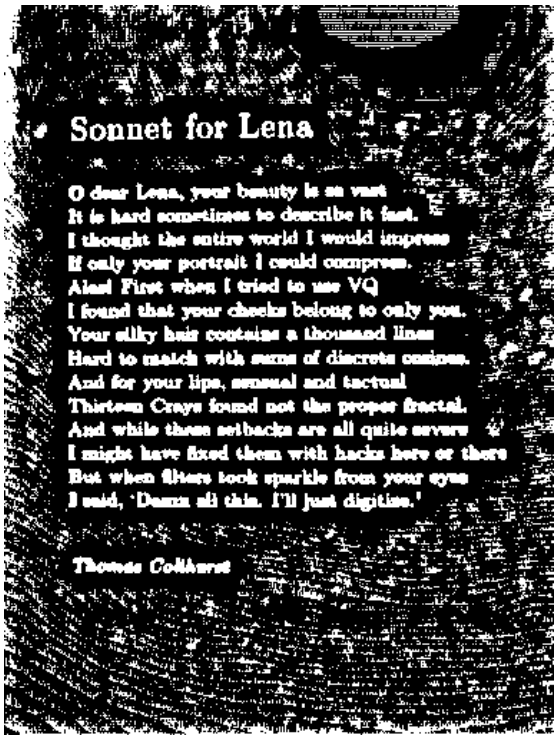

Figura 4: Método de Niblack aplicado à imagem "sonnet.pgm" com 77% dos pixels em preto.

```
    # fórmula de Sauvola e Pietaksinen
    aux = (1 + k * ((std_dev / R) - 1))
    return mean_val * aux
```

Phansalskar, More e Sabal

O método de Phansalskar, More e Sabal pode ser utilizado em imagens de baixo contraste. Similarmente à notação usada nos dois métodos anteriores, temos que nosso limiar é igual à

$$\mu(x,y)\left[1 + p\exp(-q\ \mu(x,y)) + k\left(\frac{\sigma(x,y)}{R} - 1\right)\right]$$

Como sugerido, por padrão, utilizaremos os valores de `p` como 2, de `q` como 10, `k` como 0.25 e `R` como 0.5. Calcular o limiar agora se trata apenas de resolver a fórmula (utilizaremos `aux` para representar o termo mais a direita).

```
from math import exp
def phansalskar(window, p=2, q=10,
                k=0.25, R=0.5):
    mean_val = int(np.mean(window))
    std_dev = int(np.std(window))
    # componente a direita da fórmula
    aux = 1 + p * exp(-q * mean_val)
    aux += k * ( ( std_dev / R ) - 1 )
    # média vezes o componente a direita
    return mean_val * aux
```

Contraste

O método do contraste nos pede que atrelemos o valor de preto (objeto) àqueles pixels que estão mais próximos do maior valor e fundo (branco) aqueles mais próximo do menor valor. Assim, isso equivale a dizer que o limiar é a média entre o maior e o menor valores na janela.

```
def contraste(window):
    img_min = int(np.min(window))
    img_max = int(np.max(window))
    # média
    return (img_max + img_min) // 2
```

Figura 5: Método de contraste aplicado à imagem "peppers.pgm" com 54.18% em preto.

Por fim, outros dois métodos mais simples de determinar o limiar é pela média e mediana dos valores na janela. Para o método da média, utilizamos a função `numpy.mean` e um valor constante a ser reduzido de cada limiar (`cte`, por padrão, 5).

```
def media(window, cte=5):
    return int(np.mean(window)) - cte
```

Já para a mediana, apenas a função `numpy.median` será suficiente.

```
def mediana(window):
    return np.median(window)
```

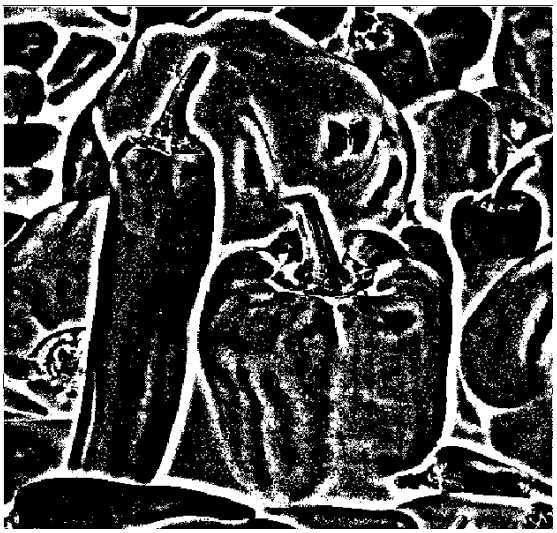

Figura 6: Método de média aplicado à imagem "peppers.pgm" com 74.00% em preto.

Figura 7: Método de mediana aplicado à imagem "peppers.pgm" com 54.63% em preto.