DCO1008 - Processamento Digital de Sinais

Vinícius Dantas de Lima Melo - 2013020579
vinicius.gppcom@gmail.com

2017.1

1 Introdução

1.1 Objetivo

Esse projeto visa à implementação da convolução 2D para aplicar filtros em imagens. Foi-se requisitado que o algoritmo seja rodado contra as seguintes máscaras:

$$H_1 = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

1.2 Relevância

Filtros podem ser aplicados a imagens para acentuar algumas características delas, seja apra melhor observar contornos, suavizá-los, atenuar cores etc. Atualmente, por causa do grande uso de redes sociais, imagens são constantemente modificadas para que possam ser compartilhadas. Um aplicativo bastante famoso chamado Instagram, com valor de mercado de cerca de $35 bilhões de dólares, possui como uma de suas ferramentas mais famosas a aplicação de filtros em imagens antes de serem compartilhadas no aplicativo.

1.3 Convolução 2D

Convolução 2D é um método bastante utilizado para aplicar filtros a imagens. Definamos essa operação como uma que recebe uma imagem X[m,n] e uma função H chamada de máscara (ou de *kernel*). Assumiremos que essa imagem X[m,n] é um sinal 2D discreto e que possui amplitude x no intervalo [0,255] na escala de tons de cinza, em que o valor 0 representa o preto e o valor 255 representa o branco. Para isso, temos que a imagem resultante y é encontrada

por:

$$y[m,n] = \sum_{i=-k}^{k} \sum_{j=-k}^{k} X[m+i, n+j] H[i,j]$$

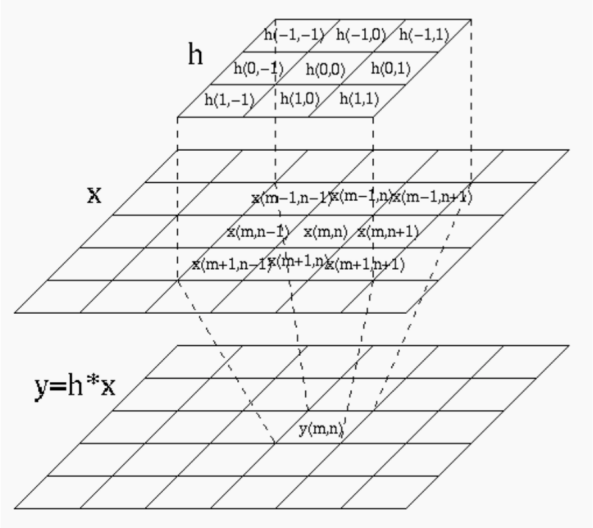

Figure 1: Representação da convolução 2D

1.4 Índice de similaridade estrutural (SSIM)

Para comparar o resultado da implementação realizada, foi utilizado como padrão o resultado da função *scipy.ndimage.convolve*, implementada como parte da biblioteca scipy, uma biblioteca de código aberto com ferramentas científicas para Python. Foi-se utilizada essa função sem explicitar argumentos além da imagem e da máscara, o que implica preenchimento de bordas com zero.
A implementação da Scipy usa a similaridade da operação de convolução com a operação de correlação [1] e, no final, usa uma implementação de correlação em C.
Com a finalidade de comparar as imagens, escolheu-se o SSIM ao invés do erro mínimo quadrático (MSE). MSE é um método amplamente utilizado em diversas áreas, mas SSIM é um método concebido com o propósito de comparar imagens.
Embora o SSIM tenha um performance inferior e sua computação demore um pouco mais (não tão perceptível para as pequenas imagens utilizadas nesse projeto) [2], apresenta um resultado no intervalo [0,1], o que é mais fácil de analisar do que o resultado do MSE, cujo valor pode assumir diversas ordens de grandeza.
Além disso, o SSIM é um método que compara as imagens em janelas, subdividindo a comparação para pequenas partes dela ao invés de comparar globalmente, como o MSE.
SSIM é um método utilizado para medir a similaridade entre duas imagens.

Esse método foi concebido para analisar imagens digitais que requerem uma comparação mais apurada, sendo utilizado, por exemplo, para predizer a qualidade imagens de televisão digital e imagens cinematográficas [3].
Para as análises presentes desse projeto, arrendodou-se o SSIM em cinco casas decimais.

1.5 Notações

Nesse projeto, será utilizado um asterisco (*) para denotar uma multiplicação ponto a ponto entre duas matrizes.

1.6 Expectativa

Espera-se que o algoritmo da convolução 2D seja devidamente implementado e que tenha um bom resultado quando comparado com o resultado padrão.

2 Desenvolvimento

2.1 Recursos usados

A implementação do código foi feita na linguagem de programação Python, utilizando a versão 3.5.2.
Foram utilizadas as bibliotecas Numpy (versão 1.12.0), Scipy (versão 0.18.1), e scikit-image (versão 0.12.3).
Para o versionamento do código, utilizou-se o sistema Git, hospedando o repositório no *Github*.

2.2 Implementação

A fórmula mostrada para definir a operação de convolução 2D apresenta, claramente, um problema de bordas: por exemplo, o que fazer para determinar y[0,0]? Nesse caso, teríamos:

$$y[0,0] = \sum_{i=-k}^{k} \sum_{j=-k}^{k} X[i,j]H[i,j]$$

Observando os índices do somatório, teríamos situações em que índices negativos aconteceriam. Como lidar com esses índices?
Para isso, devemos adicionar bordas às imagens antes de processá-la, uma forma comumente utilizada é preencher as bordas com zeros.
Nessa implementação, são apresentadas duas opções: ou preencher a borda com zeros ou preencher com o valor médio da submatriz que se deseja. Outras abordagens são discutidas em [1].
O objetivo de ambas abordagens adotadas é de não alterar o resultado esperado pela multiplação ponto a ponto seguida da soma. Esse objetivo será melhor

esclarecido com os exemplos seguintes. Analisando o canto superior de uma imagem (x[0,0]) em que queremos aplicar a máscara H_1, teríamos:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & X[0,0] & X[0,1] \\ e & X[1,0] & X[1,1] \end{bmatrix}$$

Para essa máscara H_1, pode-se perceber que estamos aplicando a média aritimétrica quando somarmos todos os elementos, pois teríamos:

$$y[0,0] = \frac{1}{9}(a+b+c+d+X[0,0]+X[0,1]+e+X[1,0]+X[1,1])$$

Vale notar que, caso as bordas fossem zeros, a média seria alterada. Para não alterar esse valor, deveríamos ter:

$$a = b = c = d = e = \frac{X[0,0]+X[0,1]+X[1,0]+X[1,1]}{4}$$

Dessa forma, decidiu-se por uma implementação que preencha as bordas com a média da submatriz existente. Porém, para permitir uma comparação dessa abordagem, também se implementou uma borda preenchida com zeros. Tem-se também que bordas preenchidas com zero fazem mais sentido para a máscara H_2. Para aplicá-la, temos, de forma similar:

$$\frac{1}{9}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & X[0,0] & X[0,1] \\ e & X[1,0] & X[1,1] \end{bmatrix}$$

Somando os elementos dessa multiplicação, tem-se:

$$y[0,0] = 0+b+0+d-4X[0,0]+X[0,1]+0+X[1,0]+X[1,1]$$

Para não alterar o resultado da operação, faz mais sentido preencher as bordas com zeros do que com a média.
Graças à simetria dessas máscaras em torno de sua diagonal principal, a mesma análise valeria para outros valores extremos de X.
Conforme analisado, a operação definida tenta acessar índices que estão além das dimensões de X. Dessa forma, foi-se criada uma classe chamada UnboundedArray, a qual herda da classe *numpy.ndarray*, implementada pela biblioteca numpy, sobreescrevendo seu método __getitem__, o qual é acessado quando se tenta acessar índices de uma array.
Sendo assim, a classe foi implementada da seguinte forma:

```
1  import numpy as np
2
3
4  class UnboundedArray(np.ndarray):
5      COLUMN = (-1, 1)
```

```
    LINE = (1, -1)

    def __new__(cls, input_array, *args, **kwargs):
        return np.asarray(input_array).view(cls)

    def __init__(self, input_array, *, padding='zero'):
        if self.ndim > 2:
            raise NotImplementedError('3+ dimensions not supported')
        if padding == 'mean':
            self.padding = 1
        elif padding == 'zero':
            self.padding = 0
        else:
            raise NotImplementedError(
                padding+' padding not implemented. Options are: mean, zero'
                )

    def _generate_bounds(self, arr, item):
        for axis, x in (
               (axis, x) for axis, x in enumerate(item) if isinstance(x, slice)
               ):
            p = -x.start if x.start < 0 else 0
            q = x.stop-self.shape[axis] if x.stop > self.shape[axis] else 0
            stack = (np.vstack, np.hstack)[axis]
            shape = arr.shape[not axis]
            _1dshape = (self.LINE, self.COLUMN)[axis]
            arr = stack((np.array(
                                    [[arr.mean()*self.padding]*(shape), ]*p
                                    ).reshape(_1dshape),
                         arr)
                        ) if p else arr
            arr = stack((arr,
                         (np.array(
                            [[arr.mean()*self.padding]*(shape), ]*q
                             ).reshape(_1dshape)))
                        ) if q else arr
        return arr

    def __getitem__(self, item):
        bounds = tuple(
                    slice(
                        max(x.start, 0),
                        min(x.stop, size), x.step
                        ) if isinstance(x, slice) else x for x, size
                    in zip(item, self.shape)
                    )
```

```
        arr = super().__getitem__(bounds)
        arr = np.array(arr)
        if arr.ndim == 1:
            if isinstance(bounds[0], slice):
                arr = arr.reshape(self.COLUMN)
            elif isinstance(bounds[1], slice):
                arr = arr.reshape(self.LINE)
        if bounds != item:
            arr = self._generate_bounds(arr, item)
        return arr
```

Com essa clase, fica-se extremamente simples implementar a convolução 2D, que foi implementada como parte do seguinte script:

```
#!bin/python3
import scipy
import numpy as np
from skimage.measure import compare_ssim
import matplotlib.pylab as plt


from array import UnboundedArray


def rgb2gray(rgb):
    r, g, b = rgb[:, :, 0], rgb[:, :, 1], rgb[:, :, 2]
    # Scipy implements: r * 299/1000 + g * 587/1000 + b * 114/1000
    # that is the ITU-R 601-2 luma transform
    # Note there is a subtle difference for the r factor, which is
    # matlab's (NTSC/PAL) implementation
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray


def to_image(arr):
    return scipy.misc.toimage(arr, cmin=0)


def convolve2d(image, kernel):
    filtered = np.zeros(image.shape)
    for (i, j), x in np.ndenumerate(image):
        filtered[i, j] = image[i-1:(i+1)+1, j-1:(j+1)+1].dot(kernel).sum()
    return filtered

web_lena = scipy.misc.imread('lena.png').astype(float)
sigaa_lena = rgb2gray(scipy.misc.imread('lena_web.png', mode='RGB')).astype(float)
```

```
34
35  kernel1 = 1/9*np.ones((3, 3))
36  kernel2 = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
37
38  for k, kernel in enumerate((kernel1, kernel2)):
39      for im, lena in enumerate((web_lena, sigaa_lena)):
40          lenas_name = ('web', 'sigaa')[im]+('_a', '_b')[k]+'_filtered_lena'
41
42          sc = to_image(scipy.ndimage.filters.convolve(lena, kernel))
43          sc.save('scipy_'+lenas_name+'.png')
44          for padding in ('zero', 'mean'):
45              image = UnboundedArray(lena, padding=padding)
46
47              filtered = convolve2d(image, kernel)
48              filtered = to_image(filtered)
49              filtered.save(padding+'_'+lenas_name+'.png')
50
51              ssim = round(
52                  compare_ssim(
53                      scipy.misc.fromimage(filtered), scipy.misc.fromimage(sc)
54                      ), 5
55                  )
56              print(padding+'_'+lenas_name+'\n    SSIM '+str(ssim))
```

Rodou-se o script com duas entradas: a imagem da Lena disponibilizada pelo professor no SIGAA e uma versão colorida baixada da internet e transformada em tons de cinza através da função rgb2gray, a qual foi definida no mesmo script.

Figure 2: Lena em preto e branco disponibilizada no SIGAA

Figure 3: Lena colorida baixada da internet

3 Resultados

A nível de ilustração, apenas será exibida a lena que obteve o maior ISSM para cada máscara.

3.1 Máscara H_1

	Com zeros	Com valor médio
Internet	0.99491	1.0
Sigaa	0.99503	0.99999

Figure 4: Lena com aplicação da máscara H_1

Conforme discutido anteriormente, a máscara H_1 consiste da aplicação da média aos pixels. O filtro da média é um filtro passa-baixa, já que reduz altas variações entre valores próximos.
Na imagem, pode-se perceber o efeito do filtro como uma suavização dos contornos.

3.2 Máscara H_2

	Com zeros	Com valor médio
Internet	0.4959	0.49596
Sigaa	0.65289	0.65295

Figure 5: Lena com aplicação da máscara H_2

Com base na observação da imagem, vê-se que a aplicação da máscara H_2, diferente da máscara H_1, tem o efeito de um filtro passa-alta, realçando os contornos da imagem após sua aplicação.

4 Conclusão

Com base nos valores do ISSM, pôde-se comparar as saídas do algoritmo implementado com a implementação considerada como padrão.
Para a máscara H_1, obteve-se valores bem próximos de 1, percebendo-se valores maiores (embora com diferença apenas a partir da terceira casa decimal), para o preenchimento das bordas com o valor médio, conforme esperado.
Já para a máscara H_2, valores baixos foram encontrados, obtendo-se valor maior para a imagem baixada do SIGAA, o que se deu porque a implementação da scipy foi rodada contra essa imagem. Quebrando expectativas, o melhor valor foi encontrado preenchendo as bordas com o valor médio, assim como com a máscara H_1.
Contudo, a possível causa para esses resultados baixos para a máscara H_2 não é a implementação da convolução em si, mas a normalização dos valores realizada para que a imagem fosse salva. Percebe-se, então, que a convolução 2D não apresenta grande dificuldade, mas a normalização de seu resultado sim.

References

[1] David Jacobs *Class Notes for CMSC 426* Fall 2005 em `http://www.cs.umd.edu/~djacobs/CMSC426/Convolution.pdf`.

[2] Adrian Rosebrock *How-To: Python Compare Two Images* em `http://www.pyimagesearch.com/2014/09/15/python-compare-two-images/`.

[3] Structural Similarity *Wikipedia* em `https://en.wikipedia.org/wiki/Structural_similarity`.