Análise da complexidade de algoritmo de convolução aplicado em conjunto com Filtros de Gabor.

Gilmar Santos, Mauricio Menezes, Marcelo Henrique, Ricardo Azevedo, Tamires Farias

Curso de Bacharelado em Sistemas de Informação – Universidade do Estado da Bahia (UNEB) – Campus I (Salvador, Bahia), 2019.1

1. Introdução

O processamento de imagens é certamente uma área em crescimento no meio científico, sendo o tema comumente abordado na literatura. Uma das definições para processamento de imagens diz que "processar uma imagem consiste em transformá-la sucessivamente com o objetivo de extrair mais facilmente a informação nela presente".

Neste âmbito, outro conceito importante é o de filtragem de imagens, que consiste na aplicação de técnicas de transformação (operadores – máscaras) com o objetivo de corrigir, suavizar ou realçar determinadas características de uma imagem, dentro de uma aplicação específica, extraindo assim os atributos desejados. Esses atributos referem-se as informações presentes em uma imagem digital e são subdivididos em quatro categorias: espectrais, espaciais, de contexto e temporais. Uma das técnicas mais conhecidas para analise e processamento de imagens e consequentemente extração de seus atributos é a convolução.

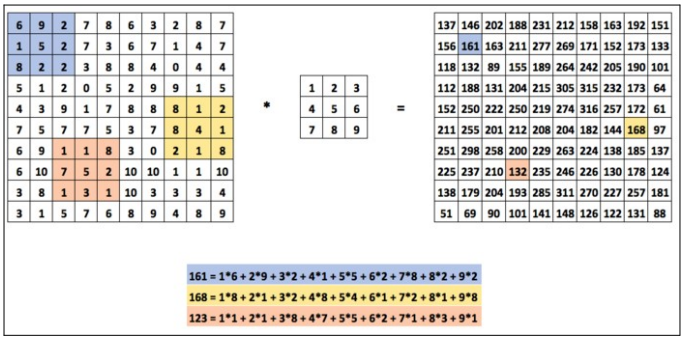
Um computador "enxerga" uma imagem como um conjunto de números, no caso, matrizes.

Então, computacionalmente falando, uma imagem é apenas uma matriz multidimensional, que tem uma largura (número de colunas) e uma altura (número de linhas), exatamente como uma matriz. Mas, diferentemente das matrizes tradicionais, as imagens também têm uma profundidade: o número de canais na imagem. Para uma imagem RGB padrão, por exemplo, tem-se uma profundidade de três, representando cada um dos canais vermelho, verde e azul, respectivamente.

Dado esse conhecimento, podemos pensar em uma imagem como uma grande matriz em conjunto com um núcleo - matriz convolucional - usada para desfoque, nitidez, detecção de borda e outras funções de processamento de imagem. Essencialmente, esse

minúsculo núcleo/kernel fica na parte superior da imagem grande e desliza da esquerda para a direita e de cima para baixo, aplicando uma operação matemática (isto é, uma convolução) em cada (x, y) coordenada da imagem original.

A Convolução, então, envolve a aplicação de um kernel sobre uma imagem, gerando uma nova imagem como resultada. O kernel também sendo uma matriz, na prática, “desliza” sobre toda a imagem, modificando-a. Para um dado deslocamento, os elementos do kernel são multiplicados pelos pixels da imagem, que, por fim, são somados e resultam em um elemento da imagem convolucionada:.



O custo computacional da convolução geralmente é muito alto, em uma imagem M x M onde se aplica um filtro N x N são realizadas M^2 x N^2 operações. Se pensarmos em exemplos que envolvem uma imagem de tamanho 512 x 512 e um kernel de 16 x 16, são necessárias nesse processamento 67.108.864 multiplicações.

Com base nessa dificuldade de implementação, este trabalho tem como objetivo o estudo da complexidade de um algoritmo de convolução, utilizado em conjunto com cinco diferentes filtros de gabor, para a extração de atributos de cinco imagens de tamanhos diferentes. Sendo que os filtros aplicados apresentam diferenciação quanto a frequência e orientação.

2. Descrição e validação do ambiente de testes

O ambiente escolhido foi o Sistema operacioanal Linux, distro Debian 9, com 8GB de memória RAM, com 500GB disponível em Armazenamento. A escolha foi por conhecer que no ambiente Open Source temos mais liberdade para alterar o ambiente, como bloqueio de serviços, criação de scrips para desabilitação de funcionalidades, remoção de parte gráfica para melhorar o desempenho, testes realizados mostram que influência do desempenho de outras tarefas do sistema operacional tiveram impácto inferior a 5%.

2.1. Descrição do código utilizado

O código utilizado na implementação do método de convolução utiliza linguagem Python juntamente com a biblioteca OpenCv, utilizada para o desenvolvimento de aplicativos na área de visão computacional.

As primeiras linhas do código referem-se aos pacotes Python utilizados. Dentre eles destaca-se o "cv2", pacote OpenCv que contém a função "GetGaborKernel" a qual será utilizada na geração dos diferentes filtros de Gabor utilizados na convolução:

```
2
3  # pacotes necessários
4  from skimage.exposure import rescale_intensity
5  import numpy as np
6  import argparse
7  import cv2
8  import pdb
9  import time
10 from scipy import stats
11
```

O método de convolução é definido a partir da linha 12, nomeado como "convolve" recebe dois parâmetros: a imagem, já em escala de cinza, e o filtro gabor a ser utilizado. A imagem é representada como uma matriz NumPy, e suas dimensões espaciais, isto é, sua largura e altura são definidas nas linhas 16 e 17, "image.shape" retorna estas dimensões, tanto para a imagem quanto para o filtro Gabor.

No processo de deslizar a matriz convolucional através de uma imagem, aplicando a convolução e armazenando a saída, a imagem gerada será diminuída. Isso acontece por que o filtro é posicionado em torno de uma cordenada central (x,y) da imagem de entrada. Isso implica na não existência de filtros centrais para os pixels que estão ao longo da borda da imagem. Sendo assim, a diminuição da dimensão espacial é simplesmente um "efeito colateral" da aplicação de convoluções às imagens. Na maioria dos casos, para obter uma imagem de saída com as mesmas dimensões da imagem de entrada aplica-se um preenchimento, onde pixels são replicados ao longo da borda. No código, essa replicação é feita nas linhas 23 a 26:

```
11
12 # método de convulsão
13 def convolve(image, kernel):
14         # dimensões espaciais da imagem, e do kernel
15         # retorna a altura e largura da imagem em matriz
16         (iH, iW) = image.shape[:2]
17         (kH, kW) = kernel.shape[:2]
18
19         # alocar memória para a imagem de saída, tendo o cuidado de
20         # "acoplar" as bordas da imagem de entrada para que o espaço
21         # tamanho (ou seja, largura e altura) não sejam reduzidos
22         # replicando os pixels para que a img de saída tenha o mesmo tamanho
23         pad = (kW - 1) // 2
24         image = cv2.copyMakeBorder(image, pad, pad, pad, pad,
25                 cv2.BORDER_REPLICATE)
26         output = np.zeros((iH, iW), dtype="float32")
27
28         # faça um loop sobre a imagem de entrada, "deslizando" o kernel
29         # cada (x, y) - coordena da esquerda para a direita e de cima para
30         # inferior
31
```

A seguir, a parte do código utilizada na análise quantitativa, as linhas 35 e 36 contém os dois laços responsáveis por "deslizar" na imagem de entrada, o kernel/filtro utilizado. Sendo que, o sentido é da esquerda para a direita e de cima para baixo percorrendo 1 pixel por vez.

A linha 39 extrai a região de interesse (ROI) da imagem usando o fatiamento da matriz NumPy. O roi será centralizado em torno das coordenadas atuais (x, y) da imagem, e também terá o mesmo tamanho do kernel, o que é crítico para o próximo passo. A convolução é realizada na Linha 42, efetuando a multiplicação por elementos entre o ROI e o kernel, seguido pela soma das entradas na matriz. O valor de saída k, na linha 46, é então armazenado na matriz de saída nas mesmas coordenadas (x, y), relativas à imagem de entrada:

```
32
33          # Parte Utilizada na Analise Quantitativa
34          #=============================================================================#
35          for y in np.arange(pad, iH + pad): # Complexidade n
36              for x in np.arange(pad, iW + pad): # Complexidade n
37                  # extrai a area de interesse da imagem ROI, pegando o
38                  # *centro* das coordenadas (x, y)atuais
39                  roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1] # 6 Operações
40
41                  #cálculo para convolução
42                  k = (roi * kernel).sum() # 9 + 8 + 1 Operações
43                  #print(k)
44                  #armazena o valor envolvido na saída (x, y) -
45                  #coordenada da imagem de saída
46                  output[y - pad, x - pad] = k # 3 Operaçõoes
47          # =============================================================================#
48
49          #Contagem FInal: 3O(n² . (9³ + 18))
```

Ao trabalhar com imagens, normalmente encontram-se valores de pixels que pertencem ao intervalo [0, 255]. No entanto, ao aplicar convoluções, facilmente pode ocorrer de obter valores que estão fora desse intervalo. Para trazer a imagem de saída de volta ao intervalo [0, 255], aplica-se a função rescale_intensity da scikit-image na linha 52. Também é necessário, converter a imagem em um tipo de dados inteiro de 8 bits na linha 53, anteriormente, a imagem de saída era um tipo de ponto flutuante para lidar com valores de pixel fora do intervalo [0, 255]:

```
50
51          # redimensionar a imagem de saída para estar no intervalo [0, 255]
52          output = rescale_intensity(output, in_range=(0, 255))
53          output = (output * 255).astype("uint8")
54          # return a imagem de sáida
55          return output
56
```

Nas Linhas 54 e 55, são definidos os filtros de Gabor utilizados. A função “GetGaborKernel” responsável por retornar um tipo de filtro Gabor recebe os seguintes parâmetros:

- Ksize: Tamanho do filtro retornado.
- Sigma: Desvio padrão do envelope gaussiano.
- Theta: Orientação das faixas normal para paralela de uma função Gabor.
- Lambd: Comprimento de onda do fator senoidal.
- Gama: Proporção espacial.
- Psi: Deslocamento de fase.
- Ktype: Tipo de tipo de coeficiente de filtro. Pode ser CV_32F ou CV_64F

Todos os filtros são agrupados no “kernelBank”, linhas 80 a 85, e possuem tamanho 21x21, porém, diferem-se quanto aos outros parâmetros. As linhas 89 e 90 carregam a imagem do disco e a convertem em escala de cinza.

convolutions.py

```
57  # argumento de imagem
58  ap = argparse.ArgumentParser()
59  ap.add_argument("-i", "--image", required=True,
60          help="path to the input image")
61  args = vars(ap.parse_args())
62
63  # kernels
64  # parametros: ksize, sigma, theta, lambda
65  gabor = cv2.getGaborKernel((21,21), 5, 1, 10, 1, 0, cv2.CV_32F)
66  mauricio = cv2.getGaborKernel((21,21), 15, 1, 10, 1, 0, cv2.CV_32F)
67  marcelo = cv2.getGaborKernel((21,21), 25, 2, 20, 1, 0, cv2.CV_32F)
68  tamires = cv2.getGaborKernel((21,21), 35, 2, 15, 1, 0, cv2.CV_32F)
69  ricardo = cv2.getGaborKernel((21,21), 8, 1, 4, 3, 0, cv2.CV_32F)
70  gilmar = cv2.getGaborKernel((21,21), 3, 3, 6, 3, 0, cv2.CV_32F)
71
72
73  # construir o banco de kernel, uma lista de kernels que vamos
74  # aplicar usando a função `convole` e a Função filter2D do OpenCV
75
76  kernelBank = (
77          ("maricio", mauricio),
78          ("ricardo", ricardo),
79          ("tamires", tamires),
80          ("marcelo", marcelo),
81          ("gilmar", gilmar)
82  )
83
84  # carrega a imagem de entrada e convert para escala de cinza.
85  image = cv2.imread(args["image"])
86  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Por fim, inicia-se o loop de aplicação dos kernels. Cada Kernel será aplicado à imagem de entrada já em escala de cinza. O método de convolução é chamado na linha 94. Os cálculos de tempo de execução assim como de variância e desvio padrão são executados a partir da linha 98. Como verificação de integridade, também se utilizou a função “cv2.filter2D”, que é uma função de aplicação de filtros da biblioteca OpenCv. A comparação é feita em nível de segurança do funcionamento da função “Convolve”. Finalmente, as linhas 115-120 exibem as imagens de saída na tela:

```
89  for (kernelName, kernel) in kernelBank:
90          print("[INFO] aplicando {} kernel".format(kernelName))
91          tempoInicial = time.time()
92          # aplicando função convolve
93          convoleOutput = convolve(gray, kernel)
94          tempoFinal = time.time()
95          tempoExecucao = str(tempoFinal - tempoInicial)
96          # aplicando função2D do openCVopencvOutput = cv2.filter2D(gray, -1, kernel)
97          tempoFinal = time.time()
98          tempoExecucao = str(tempoFinal - tempoInicial)
99          print(tempoExecucao)
100         print("\n")
101         print("Media")
102         print(convoleOutput.mean())
103         print("Variancia")
104         print(convoleOutput.var())
105         print("Desvio padrao")
106         print(convoleOutput.std())
107
108         print("\n")
109
110
111         # mostrar imagens de saída
112         cv2.imshow("original", gray)
113         cv2.imshow("{} - convole".format(kernelName), convoleOutput)
114
115         #cv2.imshow("{} - opencv".format(kernelName), opencvOutput)
116         cv2.waitKey(0)
117         cv2.destroyAllWindows()
118
```

2.2. Análise Quantitativa

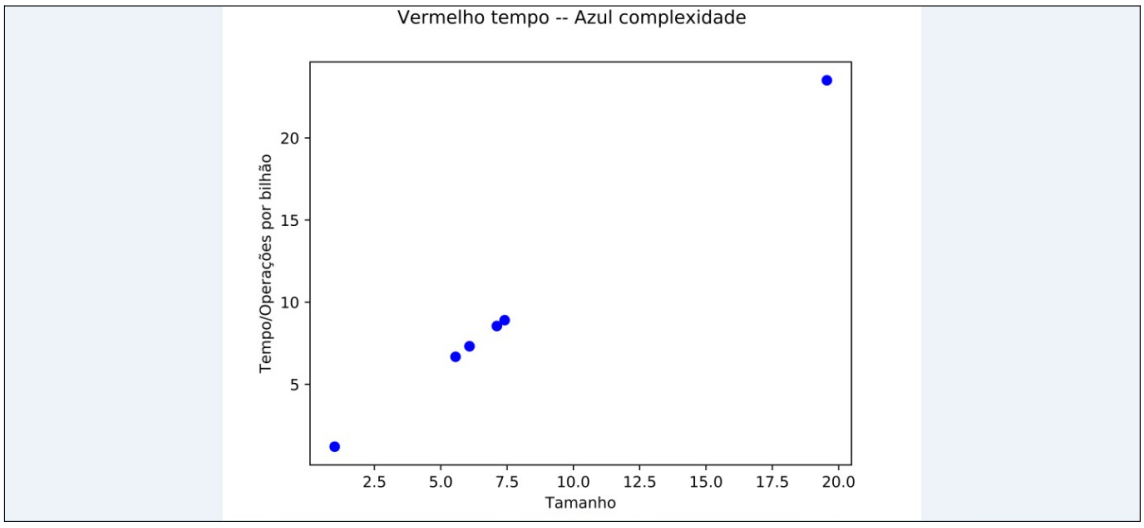
```
32
33          # Parte Utilizada na Analise Quantitativa
34          #=============================================================================#
35          for y in np.arange(pad, iH + pad): # Complexidade n
36                  for x in np.arange(pad, iW + pad): # Complexidade n
37                          # extrai a area de interesse da imagem ROI, pegando o
38                          # *centro* das coordenadas (x, y)atuais
39                          roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1] # 6 Operações
40
41                          #cálculo para convolução
42                          k = (roi * kernel).sum() # 21 + 20 + 1 Operações
43                          #print(k)
44                          #armazena o valor envolvido na saida (x, y) -
45                          #coordenada da imagem de saida
46                          output[y - pad, x - pad] = k # 3 Operações
47          # =============================================================================#
48
49          #Contagem FInal: O(n² . (21² + 18))
50
```

2.3 Classificação da Complexidade

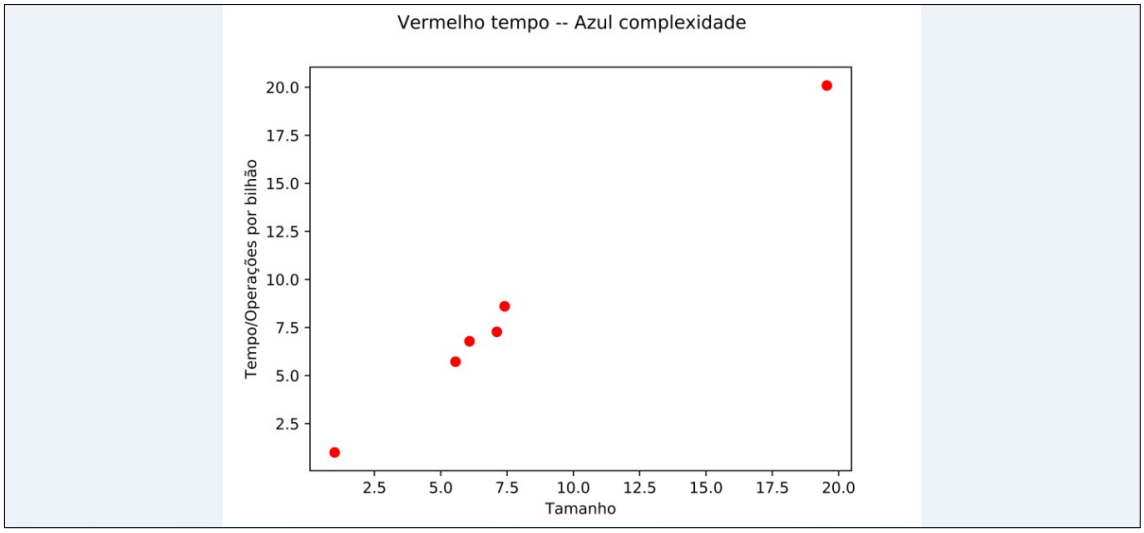
Como o código utilizado para a convolução, é formado pelo aninhamento de repetições contadas em k níveis, dependendo do tamanho do problema, no caso, o tamanho das imagens, nesse caso a complexidade da estrutura aninhada será da ordem n*k. No código como existem duas repetições o algoritmo apresenta complexidade $O(n^2)$.

2.4 Resultados Teóricos

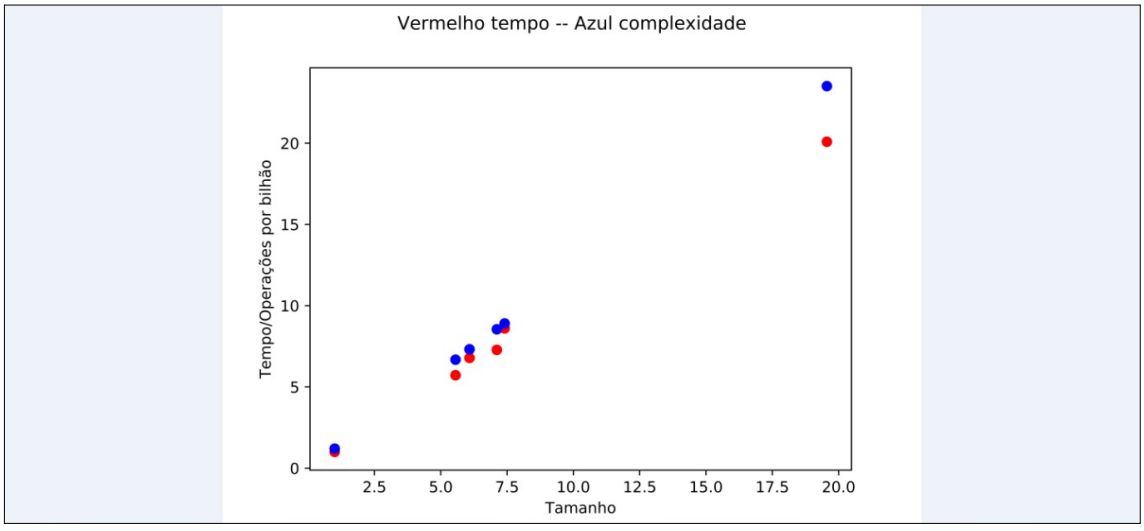
Para a melhor visualização dos resultados teóricos foi plotado no gráfico as operações por tamanho da imagem. Para que as medidas se tornassem sem unidade de medida todos os números foram divididos pela menor unidade de sua categoria. O tamanho foi normalizado multiplicando-se a altura pela largura e dividindo pela sua menor unidade.



2.5 Resultados Experimentais



2.4 Comparação Teoria- Prática



3. Conclusão

O algoritmo é custoso computacionalmente para imagens muito grandes apesar de ter uma complexidade $O(n^2)$ e necessita de computadores com grande capacidade de processamento. Pode-se automatizar o processo de carregamento e extração de features e filtros. Dentre os pontos fortes está a sofisticada capacidade de extrair características relevantes para a análise de imagens.

A implementação foi importante para a fixação dos conceitos abordados na matéria através de um problema prático e real, podemos perceber que otimizar uma solução,ou um algoritmo é um passo complexo e necessita de bastantes critérios como o particionamento do problema em problemas menores e utilização de técnicas para otimizar a solução, para implementação estudamos a bibliografia disponível sobre convolução e filtros de gabor e implementamos o algoritmo de convolução com a ajuda das bibliotecas OpenCV e Numpy.

Referências

PISTORI, Hemerson. Bancos de Filtros Gabor. 2015. (26m49s). Disponível em: <https://www.youtube.com/watch?v=rJ28FGLXFH4>. Acesso em: 03 jul. 2015.

DAVID, Karas. Algorithms for Efficient Computation of Convolution. Cap 8 1996

Portal Elo 7, Google Analytics. Disponível em: < https://elo7.dev/convolucao/>. Acesso em 19 de setembro de 2019.