Trabalho #3:

MO443 - Introdução ao Processamento de Imagem Digital

Edson Riberto Bollis
RA: 144660
edsonbollis@gmail.com

I. Especificação dos Problemas

"A análise de documentos pode ser empregada na conversão automática da informação contida nas imagens de documentos em texto editável. A digitalização de documentos tem sido amplamente utilizada na preservação e disseminação de informação em formato eletrônico. Um problema frequente que ocorre no processo de digitalização é o desalinhamento do documento, ou seja, o posicionamento do papel com uma inclinação diferente do eixo do digitalizador. A correção da inclinação é fundamental para o adequado funcionamento de sistemas de reconhecimento ótico de caracteres.

Dois algoritmos para detecção e correção de inclinação de documentos devem ser implementados neste trabalho, um baseado em projeção horizontal e outro baseado na transformada de Hough" [5]

As tarefas necessárias para o desenvolvimento do algoritmo são as seguintes:

1) Implementar uma técnica de análise de rotação de textos utilizando projeção horizontal;
2) Implementar uma técnica de análise de rotação de textos utilizando transformada de Hough;
3) Utilizar um Reconhecedor Ótico de Caracteres (Optical Character Recognition – OCR) para analisar o texto após sua rotação.

II. Entrada de dados

O script para calcular todos os problemas enunciados na Seção I foi desenvolvido utilizando Python e seu código precisou de um arquivo chamado **alinhar.py**. O programa foi feito para aceitar imagens em tons de cinza no formato RGB do tipo PNG (*Portable Network Graphics*). Então, para executarmos seus cálculos, devemos chamar por linha de comando o arquivo e mais os parâmetros como no exemplo:

```
alinhar.py imagem_entrada.png modo imagem_saida.png
```

Os parâmetros de chamada são descritos como:

- alinhar.py: programa que corrige o ângulo de inclinação e reconhece o texto contido em uma imagem digitalizada;
- imgagem_entrada.png: caminho da imagem no formato PNG que terá o ângulo de seu texto corrigido e texto extraído;
- modo: 1 se quiser a correção do ângulo do texto da imagem de entrada utilizando a técnica de histograma horizontal e 2 se quiser utilizar a transformada de Hough;
- img_saida.png: caminho da imagem de saída com o ângulo corrigido pelo algoritmo escolhido.

Obs: A entrada de dados foi testada por imagens cedidas pelo professor e, no desenvolvimento do trabalho, por algumas da internet. Todas as imagens testadas estão na pasta **./trabalho_3/img/**.

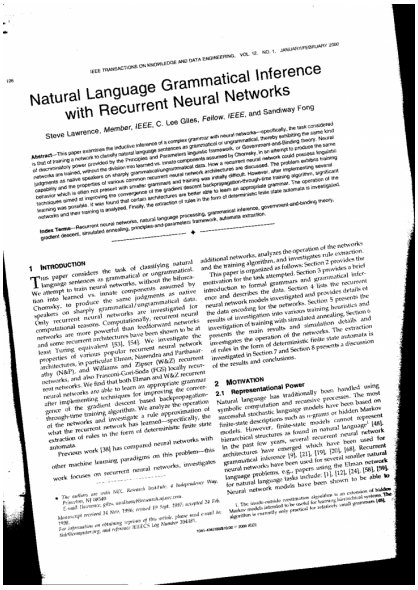
III. Código e Decisões Tomadas

A. Pré-processamento e Segmentação

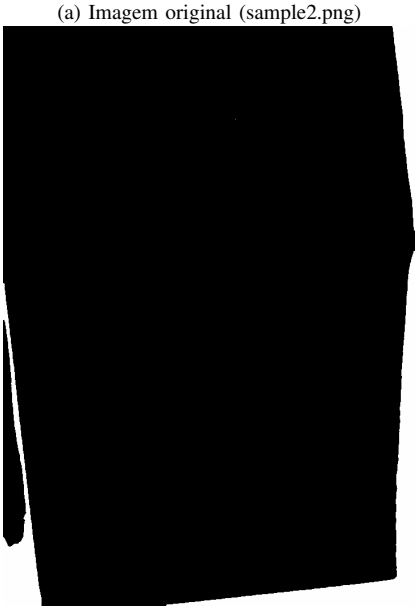
A imagem de entrada foi carregada pelo **Scipy.misc** [3] em uma matriz $I_{m \times n}$ onde m representa o número de linhas e n o número de colunas. Utilizamos o atributo *flatten = True* para carregar a imagem diretamente em tons de cinza e logo após efetuamos sua exibição em tela.

Posteriormente, utilizamos a função **skimage.filters.threshold_local** [2] com filtros que calculam a mediana de uma vizinhança cuja maior distância entre o pixel central da máscara e qualquer outro pixel é 9 (máscara de tamanho 19 $\times$ 19) e subtraímos do pixel centrado na imagem resultante o valor 0,1 obtendo, como resultado, uma matriz com valores menores que zero em regiões em que a mediana da vizinhança é igual a zero. Então, segmentamos a imagem em duas regiões, valores menores que zero e valores maiores que zero. Através desse cálculo, conseguimos detectar grandes regiões conexas totalmente pretas, como mostrado na Figura 1, para que pudéssemos extraí-las da imagem original após seu tratamento. Ao final, transformamos essa imagem em sua imagem invertida para que os valores pretos se tornassem brancos, ou seja, valores 0 tornaram-se valores 1 para que pudéssemos contabilizar mais facilmente o número de pixels significativos da imagem.

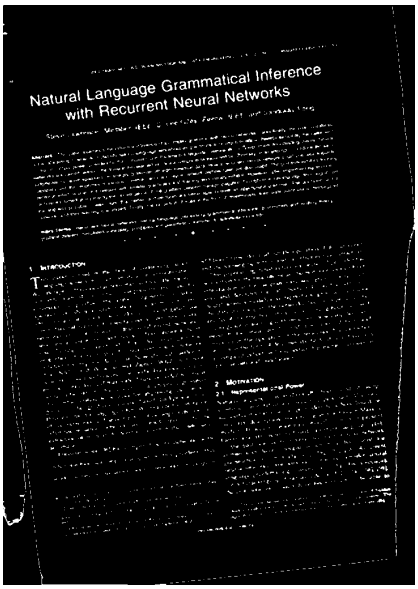
Da mesma forma, segmentamos localmente a imagem original para separar as regiões com texto das regiões sem texto. Para isso, usamos uma distância de tamanho igual a 3 na limiarização local, pois as letras são bem menores que grandes regiões conexas. Após calcular as fronteiras locais e homogeneizar essas regiões com a mediana, calculamos a imagem inversa e pegamos os valores em tons de cinza maiores que a metade da profundidade total, ou seja, fizemos um afinamento das letras. Depois, removemos as grandes regiões conexas e obtivemos a figura a ser enviada para as técnicas utilizadas neste trabalho. Podemos ver as imagens resultantes desses cálculo também Figura 1. Obs: No exercício foi pedido o uso de detectores de bordas, porém não consegui melhorar meus resultados com eles.



(a) Imagem original (sample2.png)



(b) Segmentação dos grandes elementos conexos



(c) Imagem binária pré-processada

Figura 1. Etapas do pré-processamento

B. Implementação da Projeção Horizontal

A primeira implementação do algoritmo de projeções horizontais ou histogramas de projeções horizontais utilizou diretamente a imagem gerada e descrita na Seção III-A, não obtendo bons resultados com imagens cujo número de linhas de texto fosse bem maior que o número de caracteres por linha. Então, como parte do trabalho, tivemos que dividir a matriz de entrada do algoritmo em matrizes $M^i_{m/3 \times n/3}, i = 1..9$, gerando um conjunto de matrizes S ao qual adicionamos a imagem original, totalizando 10 matrizes.

1)Sobre um domínio de 180 graus, representando metade da circunferência (-90 à 90), dividimos seu espaço em 9 segmentos com 20 graus cada e calculamos, sobre o início de cada intervalo, o histograma horizontal com maior pico dentre todos os histogramas (a função objetivo foi escolhida como o máximo de cada histograma em cada ângulo). Ao escolher um intervalo, dobramos sua extensão e usamos seu antigo ponto inicial como ponto central. Esse intervalo, foi dividido novamente em oito intervalos com cinco graus cada. 2)Efetuamos os mesmos cálculos do passo anterior sobre os pontos iniciais de cada novo intervalo e escolhemos o intervalo, cujo valor da aplicação da função objetivo em seu ponto inicial, foi o maior dentre eles. 3) No terceiro passo, dobramos outra vez o tamanho do intervalo atual para centralizá-lo em seu antigo ponto inicial e dividimos sua extensão em 10 intervalos de 1 grau cada. Finalizamos o processo ao escolher o maior valor da função objetivo aplicada aos dez pontos iniciais de cada intervalo atual.

Após efetuarmos o cálculo do ângulo para as 10 matrizes do conjunto S, escolhemos o ângulo que apareceu com maior frequência como resultado de nossos cálculos. Fizemos os cálculos da criação do histograma horizontal 27 vezes para cada matriz, o que resultou em 270 processamentos do histograma horizontal e 270 rotações. Descartamos, para o cálculo da frequência, as rotações que produziram os valores dos ângulos de -90 e 90 graus. Podemos ver na Figura 3 que conseguimos resultados muito bons para as imagens de saída.

C. Implementação da Transformada de Hough

A implementação da transformada de Hough usou a função **skimage.transform.hough_line** [2], que implementa o acumulador para a transformada da reta no espaço de parâmetros em coordenadas polares. Então, passamos como entrada a matriz resultante da Seção III-A, que já é binária, e recebemos a matriz de acumulção, imagem da Figura 2. Podemos notar na Figura 2 que existe um conjunto de regiões mais claras que estão alinhadas verticalmente na direção de um ângulo em comum. Isso representa um conjunto de retas com raios diferentes que têm o mesmo ângulo de inclinação, o que quer dizer que as linhas dos textos estão alinhadas sobre um mesmo ângulo.

Para calcular o ângulo de correção, tentamos calcular o maior pico do histograma vertical (ângulos) e assim obter a reta representada pelos pontos mais claros da figura 2, porém para cada ponto da imagem original, cria-se uma curva senoidal que se estende por todos os ângulos do domínio (0

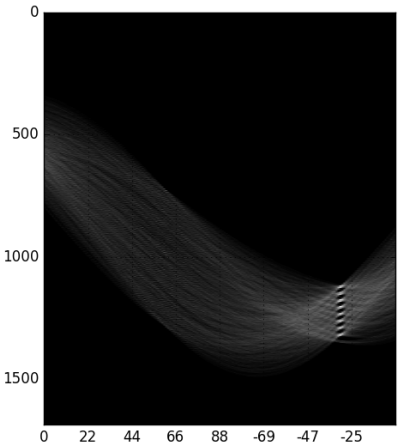


Figura 2. Representação visual do espaço parametrizado por retas na forma polar da transformada de Hough

à 180) e assim a soma dos pontos da matriz de acumulação são sempre iguais para cada coluna. Então, quanto mais claro maior o valor do ponto de acumulação, usamos a estratégia de verificar na matriz de acumulação o valor máximo dentre seus elementos e escolher sua coluna (ângulo) como valor de correção (função objetivo: ângulo cujo valor é máximo dentre os elementos da matriz de acumulação). A Figura 4 mostra o resultado do cálculo do ângulo de correção pela transformada de Hough. Obs: No final, precisamos corrigir em 90 graus o ângulo escolhido que é retornado pela matriz de acumulação.

D. Rotação e Aplicação da OCR

Recebendo o ângulo calculado no algoritmo de projeção horizontal ou no algoritmo da transformada de Hough, utilizamos o método **scipy.ndimage.interpolation.rotate** [3] para rotacionar a imagem. Depois, exibimos a imagem em tela e passamos essa imagem para a OCR ***Tesseract*** [4] extrair seu texto. Porém, notamos que a biblioteca Scipy não gerava bons resultados para o uso do *Tesseract* e, por isso, passamos a utilizar a biblioteca **OpenCV** [1] para calcular a imagem resultante da rotação utilizando o métodos de interpolação cúbica. Assim, o *Tesseract* passou a retornar textos com menos erros, porém não obtivemos um resultado ótimo para textos com letras muito pequenas. Podemos verificar esse resultado nos textos a seguir gerados pelo *Tesseract* aplicado as Figuras 3a (texto com muitos caracteres com fonte pequena) e 4b (texto com poucos caracteres com fonte grande).

Figura 3a:

```
.2. m rm...» umoumwwucun mum-u; m. .1 up,
Natural Language Grammatical Inference
with Recurrent Neural Networks
5.... Lawrence. Member, 1555.
c Lee sues. Fnliow .525 am Sanmway Vang
"mm/m5 W ......m in: mm. ......"
.. "m. Wm... M... "ID-WWW .... .... cum!!!"
9m ... ".....  mm mm. ......
"Wyn"... W... (1 mm»... mam, "mm... ......
2....
(Continua...)
```

Figura 4b:

```
Our last argument is how we want to
approximate the  contour. We use
cv2.CHAIN_APPRDX_SIMPLE to compress
horizontal, vertical, and diagonal
segments into their end-
points only, This saves both computation
and memory. If we wanted all the points
along the contour, without com-
pression, we can pass in cv2.
CHAIN_APPRDX_NUNE; however,
be very sparing when using this
function. Retrieving all
points along a contour is often
unnecessary and is wasteful
of resources.
```

IV. Saída de Dados

O programa desenvolvido gera 1 imagem, a de saida, na pasta escolhida pelo usuário e 3 imagens na pasta de execução. As imagens geradas são: **[out].png** (Figura 3a / imagem corrigida); **out_big_componen.png** (Figura 1b / imagem de grandes elementos conexos); **pre_processed.png** (Figura 1c / imagem de entrada dos algoritmos); e a **Figura hough_t.png** (Figura 2 / gerada se escolhemos o método da transformada de Hough).

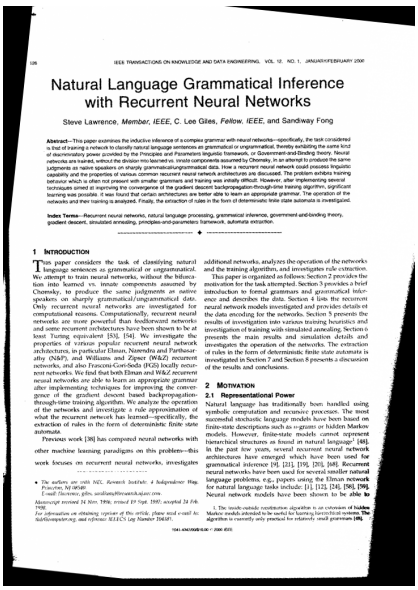
Ele também imprime em tela os valores de retorno de cada matriz do método de projeção horizontal, se escolhido, e o valor do ângulo final de correção. Antes de terminar, ele aplica o *Tesseract* sobre a imagem **[out].png** gerada e exibe o texto de retorno no terminal.

V. Resultados

Os resultados do trabalho foram satisfatórios ao que se refere a correção do ângulo de inclinação quando aplicados sobre as imagens oferecidas pelo professor, porém os algoritmos não conseguiram corrigir corretamente algumas imagens com grande variação de tons de cinza, como na Figura 5. Em relaçã da aplicação da OCR, o resultado não foi tão satisfatório, pois em vários casos que podemos ler a imagem final corrigida, a OCR não conseguiu reconhecer o texto de saída dos nossos algoritmos.

Referências

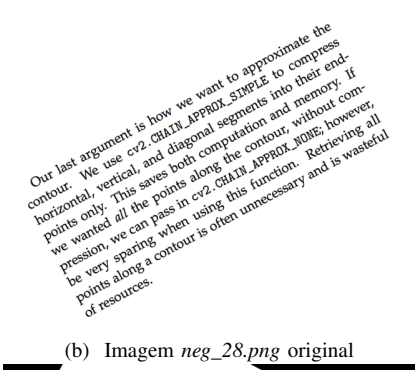
[1] Opencv:welcome to opencv documentation! https://docs.opencv.org/2.4/index.html. Acesso em 27/03/2018.
[2] Scikit-image: skimage 0.13.1 docs. http://scikit-image.org/docs/stable/. Acesso em 27/03/2018.
[3] Scipy: Documentation for core scipy stack projects. https://www.scipy.org/docs.html. Acesso em 27/03/2018.
[4] Tesseract: Home. https://github.com/tesseract-ocr/tesseract/wiki. Acesso em 15/05/2018.
[5] Hélio Pedrini. Trabalho 3. Unicamp: Introducão ao Processamento Digital de Imagem (MC920 / MO4431, 2018.



Natural Language Grammatical Inference with Recurrent Neural Networks

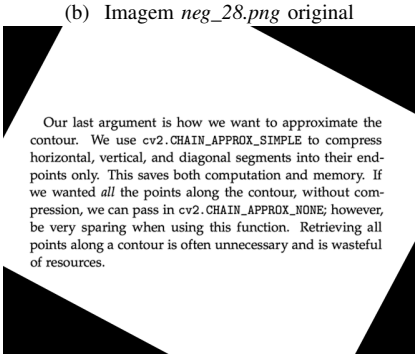
Steve Lawrence, *Member, IEEE*, C. Lee Giles, *Fellow, IEEE*, and Sandiway Fong

(a) Imagem *sample2.png* corrigida (original: Figura 1a



Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we can pass in cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

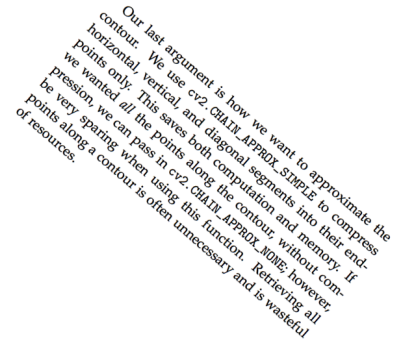
(b) Imagem *neg_28.png* original



Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we can pass in cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

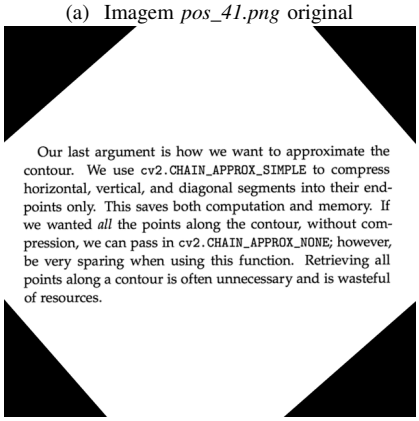
(c) Imagem *neg_28.png* corrigida

Figura 3. Resultados da projeção horizontal



Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we can pass in cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

(a) Imagem *pos_41.png* original



Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we can pass in cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

(b) Imagem *pos_41.png* corrigida

Figura 4. Resultados da transformada de Hough

Noisy image to test Tesseract OCR

(a) Imagem de entrada com maior variação dos tons de cinza

Noisy image to test Tesseract OCR

(b) Imagem corrigida

Figura 5. Resultado da aplicação da transformada de Hough com pequeno erro de alinhamento devido a variação dos tons de cinza da imagem de entrada