Princípios de Visão Computacional
3 de Junho de 2017

Contagem de vagas em estacionamentos

Daniel Marcos Botelho Barbosa
17/0052427
DanielM.B.Barbosa@hotmail.com

Gabriel Filipe Botelho Barbosa
12/0050935
gabrielfbbarbosa@gmail.com

Abstract

Este projeto se caracteriza como uma pesquisa sobre um método de contagem de vagas na área da visão computacional sem a utilização de inteligência artificial.

Várias das atividades no campo de visão computacional se apoiam em atividades menores. Com isso, é bastante notório que o processo de contagem de vagas em um estacionamento tem uma grande abrangência no campo de estudos, não apenas da visão computacional. Técnicas como análise de texturas, detecção de gradiente, processamento morfológico (limiarização, binarização, dilatação e erosão), extração de características e clusterização estão bem presentes neste trabalho.

As tarefas realizadas envolvem desde abrir uma imagem simples do tipo `BGR` *até as mais diversas técnicas para o desenvolvimento de uma solução da problemática. Para isso, a liguagem utilizada foi C++ com o padrão C++ 11 e a versão 3.2.0 do OpenCV.*

1. Objetivos

O objetivo principal o desenvolvimento desse projeto é chegar à uma solução sobre a precisão e o resultado do processo utilizado para a contagem das vagas. Além disso, a atividade como um todo objetiva a exploração dos conceitos aprendidos ao longo do curso de Princípios de Visão Computacional e ferramentas disponíveis na biblioteca em questão, OpenCV[4]. Com isso, tornar afim delas.

2. Introdução

Para se obter esse resultado da contagem, foi realizado um procedimento complexo desde detecção de gradiente até criação de algorítmos para determinar a similaridade de duas retas. O procedimento desta metodologia foi o seguinte:

Etapa 1 Abre uma imagem de estacionamento;

Etapa 2 Calcula as matrizes GLCM's;

Etapa 3 Aplica o algorítmo de detecção *Sobel Detector*[6];

Etapa 4 A partir de Sobel, aplica *Hough Line Transform*;

Etapa 5 A partir de Sobel, calcula a Transformada Probabilística de Hough.

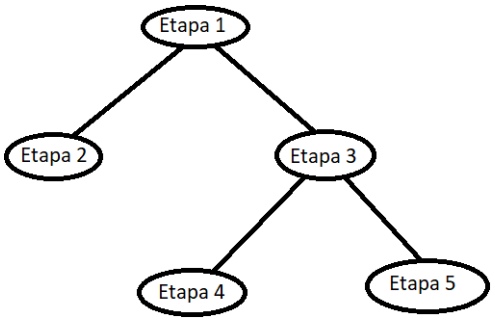

Figure 1. Diagrama em árvore da ordem e disposição das etapas realizadas.

2.1. Imagem

Os padrões de orientação e de pixelagem são tratados diferentemente pelo OpenCV.

2.1.1 Padrão RGB

Tanto imagens como vídeos são armazenados da mesma forma. A classe `cv::Mat` pode armazenar a matriz de pixels de uma imagem, bem como os frames de um vídeo.

Cada pixel tem seus canais armazenados, por padrão, na ordem Vermelho (*Red*), Verde (*Green*) e Azul (*Blue*) (`RGB`). No entanto, a biblioteca em questão utiliza um padrão diferente, definido como `BGR`, invertendo a posição do valor do pixel azul com o vermelho.

É interessante converter a imagem do tipo `BGR` para `gray scale` porque o algorítmo necessita. Trabalhando com níveis de cinza só existe uma única informação a ser extraída: a intensidade do pixel, de 0 a 255. Todavia, ao trabalhar com imagem colorida, as informações triplicam. Três canais não relacionados aumentam o nível de complexidade por não terem uma ordenação linear que seja fácil de colocar em uma matriz.

2.1.2 Coordenadas

O padrão de referências cartesianas computacionais é com origem no extremo noroeste da tela. Com eixo x crescendo para a direita e eixo y crescendo para baixo. No entanto, o padrão adotado pela biblioteca é situada com origem no extremo norte-oeste da tela, porém com eixo x crescendo para baixo e eixo y crescendo para a direita.

2.1.3 Acesso dos dados

Para realizar o acesso dos dados matriciais da classe `cv::Mat`, utiliza-se, recomendavelmente, pela documentação[1], o método `cv::Mat::at<type T> (cv::Point(j, i))`. Contudo, esse acesso garante o percorrimento para cada índice de acesso em toda a matriz. Com isso, o desempenho é comprometido. Para a retificação desse problema, pode-se utilizar o recurso de ponteiros. A biblioteca disponibiliza um tipo de dados unsigned char chamado `uchar * cv::Mat::data` em que aponta para o primeiro canal do primeiro pixel de uma matriz `cv::Mat`. No entanto, isso é para o caso de imagem colorida, como estamos trabalhando com níveis de cinza, não há preocupação com isso.

Desse modo, como a imagem, neste ponto, está em níveis de cinza, o retorno desse método `at<unsigned char>(j, i)`, em cada pixel (i, j), será um valor de 0 a 255 indicando a intensidade de pixel para cada pixel analisado. Então é só comparar com os pixels da vizinhança.

3. Trabalhos relevantes

4. Metodologia proposta

análise de texturas, detecção de gradiente, processamento morfológico (binarização e limiarização), extração de características e clusterização

4.1. Análise de texturas

As medidas de textura de co-ocorrência de nível de cinza têm sido a força de trabalho da textura da imagem desde que foram propostas por Haralick[8] na década de 1970.

Uma matriz de co-ocorrência, ou distribuição de co-ocorrência,[7] é uma matriz que é definida sobre uma imagem para ser a distribuição de valores de pixel co-ocorrentes (valores de tons de cinza ou cores) a um dado deslocamento. Esta matriz aproxima a distribuição de probabilidade conjunta de um par de pixels.

Ela representa a relação entre distância e relação de angulação espacial sobre uma sub-relação de imagem de uma região específica e de tamanho específico. Ou seja, com essa matriz de coocorrência é possível detectar, de certa forma, a textura de objetos capturados em uma imagem. Neste projeto terão quatro matrizes finais que serão obtidas pelas direções 0º, 45º, 90º e 135º, em ambos sentidos.

4.2. Detecção de gradiente

O pré-processamento da imagem foi realizado com o algorítmo de Sobel.

Sobel O detector de bordas de Sobelfootnotesize [6] se basea em algumas alterações e adaptações do operador matemático Laplaciano[2] para um espaço dimensional discreto e bidimensional[3]. O efeito desse operador no espectro matemático é o mesmo quando aplicado numa imagem, ou seja, ele ameniza variações de baixa frequência e atenua variações de alta frequência.[5]

Ele realiza esse procedimento realizando uma convolução nos eixos da imagem usando um *kernel* direcional e realizando a média dos resultados. Ele é, geralmente, quadrado com tamanho ímpar, comumente 3, onde a soma de todos os elementos é 0.

-1	0	1
-2	0	2
-1	0	1

Figure 2. Operador Sobel horizontal para imagens bidimensionais

Além disso, se considerarmos o *kernel* utilizado para computador o gradiente horizontal (Figure 2), ou seja, na coordenada x, todos os elementos da coluna central será 0. Todos os *kernels* não são nada além de uma rotação ao redor do pixel central de algum outro.

4.3. Processamento morfológico

4.3.1 Limiarização

4.3.2 Binarização

A binarização realizada após as detecções serem finalizadas foi trivial, mas merece ser rapidamente explicada pois o valor final não foi 0 ou 1. Para que fosse possível se visualizar a binarização, ela foi escalada de forma que 1 seja 255 mas que não possua nenhum outro número entre 255 e 0. Ou seja, todo pixel maior que um *threshold* determinado seria setado como 255, e 0 caso contrário.

4.3.3 Dilatação

4.3.4 Erosão

4.4. Extração de características

4.5. Clusterização

5. Resultados

6. Conclusões

6.1. Cálculo de precisão

Uma forma simples de se calcular o quão preciso um detector de borda é, quando comparado a um *ground truth*, computar quantos pixels obtidos são iguais e dividir pela quantidade total de pixels existente. Portanto, uma imagem idêntica ao *ground truth* resultaria em precisão 1. Já uma imagem oposta teria precisão 0. A equação 1 é que foi utilizada para se obter a precisão de cada detector, por imagem.

$$precision = PixelsHit/NumPixels \quad (1)$$

7. Metodologia Empregada

Como todos os detectores de borda precisam de uma imagem em escala de cinza e levemente suavizada, esse foi a primeira etapa. A biblioteca OpenCV possue funções para cada um desses operadores, logo, detectar as bordas passou a ser algo extremamente trivial como 5 ou menos linhas de código, como demonstrado no código em Listing 1.

```
Mat gradientX, gradientY;
Sobel( blurred, gradientX, S_DEPTH, 1, 0,
       S_KSIZE, S_SCALE, S_DELTA, BORDER_DEFAULT
    );
Sobel( blurred, gradientY, S_DEPTH, 0, 1,
       S_KSIZE, S_SCALE, S_DELTA,
       BORDER_DEFAULT );
convertScaleAbs( gradientX, gradientX );
convertScaleAbs( gradientY, gradientY );

Mat sobel;
addWeighted( gradientX, 0.5, gradientY, 0.5,
             0, sobel );
```

Listing 1. Obtenção das bordas por Sobel. Esse foi o maior procedimento dos três.

Para o cálculo da precisão de cada um deles, o loop foi unificado para que somente em uma iteração já fosse possível computador todos os hits. Para evitar problemas com artefatos de compressão, a imagem *ground truth* também foi binarizada antes desse cálculo.

```
int sobelHit = 0;
int cannyHit = 0;
int laplaceHit = 0;
for ( int j = 0; j < image.rows; ++j ) {
 for ( int i = 0; i < image.cols; ++i ) {
  uchar gtPixel = gt.at<uchar>( j, i );
  sobelHit += (thresSobel.at<uchar>( j, i ) ==
    gtPixel) ? 1 : 0;
  cannyHit += (thresCanny.at<uchar>( j, i ) ==
    gtPixel) ? 1 : 0;
  laplaceHit += (thresLaplace.at<uchar>( j, i )
    == gtPixel) ? 1 : 0;
 }
}

float numPix = image.rows * image.cols;
float precSobel = sobelHit/numPix;
float precCanny = cannyHit/numPix;
float precLaplace = laplaceHit/numPix;
```

Listing 2. Obtendo precisão dos detectores

Abaixo, no Listing 3, está as configurações usadas para cada detector de borda. Importante lembrar que eles não foram modificados dependendo da imagem utilizada. Na próxima seção isso será discutido o efeitos dessa decisão.

```
// Sobel
#define S_SIZE Size(3,3)
#define S_DEPTH CV_16S
#define S_SCALE 1
#define S_DELTA 0
#define S_KSIZE 3

// Canny
#define C_THRES 30
#define C_RATIO 3
#define C_KSIZE 3

// Laplace
#define L_SIZE Size(3,3)
#define L_DEPTH CV_16S
#define L_SCALE 1
#define L_DELTA 0
#define L_KSIZE 3

// Binary Threshold
#define BIN_THRES 60
```

Listing 3. Parâmetros passados para os detectores

8. Resultados

Após realizar o teste com cada uma das imagens e se obter a precisão para cada detector, a tabela 1 compila todos eles de forma organizada para se poder comparar as diferenças de uma forma não-empírica.

Detectores	Imagens					
	46	140	208	212	217	221
Sobel	0.902598	0.823201	0.834766	0.825176	0.901705	0.904328
Canny	0.967755	0.901883	0.873035	0.841835	0.938034	0.943913
Laplace	0.859735	0.756041	0.731960	0.821448	0.882921	0.858020

Table 1. Precisão obtida com cada detector por imagem

E a seguir estão as imagens-exemplo de como cada um de seus detectores se comportou.

9. Conclusões

Ao final, observando a tabela 1, é possível perceber que o detector de bordas de Canny possue a maior precisão dentre os três. Entretanto, o resultado obtido nesse experimento

Figure 3. Imagem Original

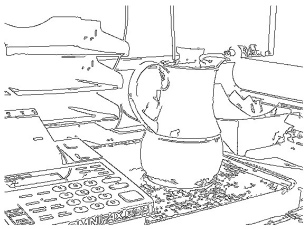

Figure 4. Ground Truth

Figure 5. Sobel

Figure 6. Sobel Binarizado

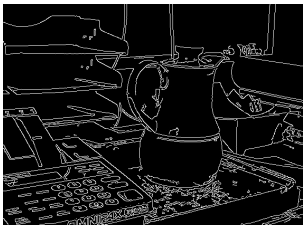

Figure 7. Canny

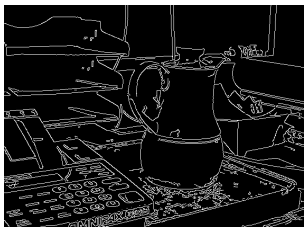

Figure 8. Canny Binarizado

Figure 9. Laplace

Figure 10. Laplace Binarizado

pode estar enviesado. Isso porquê, ao se observar a imagem *ground truth*, é possível perceber que foi gerada usando um detector de borda Canny, porém com seus parâmetros otimizados para a imagem. Isso significa que o detector de Canny, nesse experimento, provavelmente sempre obterá um resultado melhor.

E isso leva ao segundo ponto de que, nesse experimento, os parâmetros de cada detector não são ajustados para os valores ótimos de cada imagem. Isso fornece uma competição mais cega mas talvez resulte em uma comparação não tão justa, visto que não representa o caso de uso real da maioria das aplicações e ótimo de cada um deles. Esse ponto poderia ser mitigado ao tentar encontrar os parâmetros de cada detector, por imagem, que resultem na maior precisão possível para eles.

Considerando todas as ressalvas, o detector de Canny ainda assim pode ser considerado o melhor para a maioria dos casos, se o objetivo é precisão.

References

[1] Devdocs. *http://devdocs.io/*.

[2] Laplace operator. *http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html*.

[3] Laplacian edge detector. https://de.mathworks.com/matlabcentral/answers/uploaded_files/51086/Edge%20detection%20-%20Laplace%20edge%20detection%20algorythm.pdf. Accessed: 20/06/2017 - 21/06/2017.

[4] Opencv 3.2.0 documentation. *http://docs.opencv.org/3.2.0/*.

[5] Sobel derivatives. *http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html*.

[6] Sobel edge detector. http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm. Accessed: 20/06/2017 - 21/06/2017.

[7] M. Hall-Beyer. The glcm tutorial. *http://www.fp.ucalgary.ca/mhallbey/tutorial.htm*, Fevereiro, 2007.

[8] R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(6):610–621, Dezembro, 1973.