HISTOGRAM OF ORIENTED GRADIENTS

MILJKOVIC DIMITRIJE

22.12.2022.

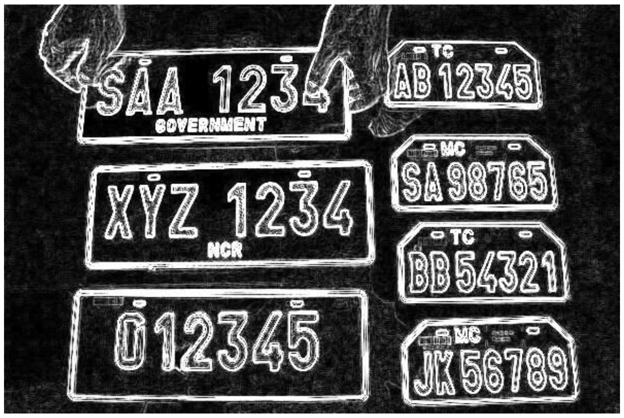
INTRODUCTION

Histogram of oriented gradients(HOG) is used in the process of object detection in digital images. Function **detect** compares two images(which have gone through some kind of image segmentation) using HOG and returns their similarity in range of [0,1], where greater value indicates higher similarity. Code will be stored in folder with this document for detailed analysis. Program is made in Processing and written in Java.

SOBEL FILTERS

Before I could make HOG function I had to make function that applies sobel filters(horizontal and vertical) on image to extract thicker edges and shapes. Sobel will be used to calculate magnitude of the gradient vector in pixel. It uses 3x3 kernel because when gradient vector is being computed neighborhood of pixel is also taken in account.

Image before application of Sobel



After application of Sobel

```
float[][] sobel1 = {{  -1,  0,  1   },
                   {    -2,  0,  2  },
                   {    -1,  0,  1  }};
```

vertical Sobel filter

```
float[][] sobel2 = {{  -1,  -2,  -1  },
                    {    0,   0,   0   },
                   {     1,   2,   1   }};
```

horizontal Sobel filter

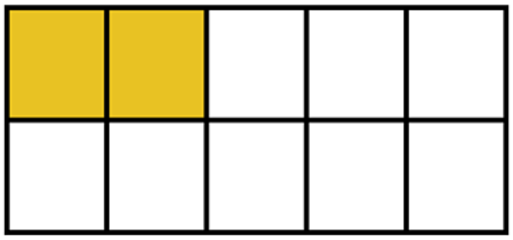
ARGUMENTS

Function **detect** which returns image similarity has following arguments:

```
float detect(PImage img1,PImage img2,int b[],int n)
```

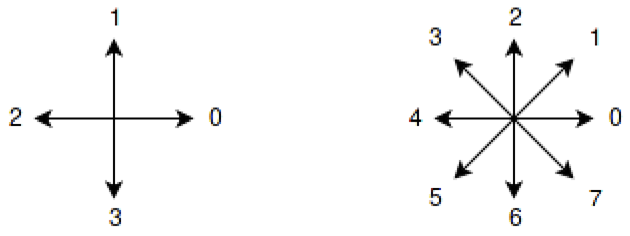
img1, img2 – pictures being compared

b[] – number of image segments(blocks), first element represents number of rows(N) and second element represents number of columns(M) image will be divided by.



2x5 block size

n – chaincode directions, set of elementary vectors, e.g. 4 or 8 on pictures below.



FEATURE VECTOR

Function is moving block by block until whole image is computed. For each block sum of feature vectors is calculated representing pixels in the observed block:

```
float movey=ceil((1.0*img1.height/b[0]));
float movex=ceil((1.0*img1.width/b[1]));

for(int i=1;i<img1.height-1;i=i+(int)movey)    //moving of blocks
  for(int j=1;j<img1.width-1;j=j+(int)movex){

    float[] b1= new float[n];    //used for summing up feature vectors in this block
```

When inside block, Sobel filters are used to calculate magnitude of gradient vector and gradient orientation of **each pixel**:

```
for (int y = i; (y < i+movey) && (y < img1.height-1); y++) //moving by pixels in block
  for (int x = j; (x < j+movex) && (x < img1.width-1); x++) {
    float gx=0,gy=0;      //magnitude of gradient vector
    float teta=0;
    //float[] a=new float[n];
    for (int ky = -1; ky <= 1; ky++)      //application of sobel filters
      for (int kx = -1; kx <= 1; kx++) {
       int index = (y + ky) * img1.width + (x + kx);
       float r = brightness(img1.pixels[index]);
       gx += sobel1[ky+1][kx+1]*r;
       gy += sobel2[ky+1][kx+1]*r;
      }

if(gy>=0)teta=atan2(gy,gx);    //gradient orientation
  else teta=atan2(gy,gx)+2*PI;
```

Way to decompose gradient vector is to project it on first right elementary vector. Decomposing gradient vector we obtain feature vector **a** which has **n** elements (number of chaincode directions)

```
//a[(int)(teta*n/(2*PI))]=abs(gx)+abs(gy);  feature vector of pixel

b1[(int)(teta*n/(2*PI))]+=abs(gx)+abs(gy);  //sum of feature vectors in block
```

whose all elements but one are 0, and the non-zero element is equal to the magnitude of gradient vector. Since all other elements are 0 we can immediately add feature vector to sum **b1** which also has **n** elements.

After we finish computing whole block we put sum in array C that represents all feature vectors of image. Size of array C is number of blocks * chaincode directions(n).

```
for(int bi=0;bi<n;bi++){
  c1[block*n+bi]=b1[bi];
  powc1+=b1[bi]*b1[bi];
}
block++;
```

Once last block of image 1 is computed function moves on next image and repeats the process.

```
block=0; //new picture
```

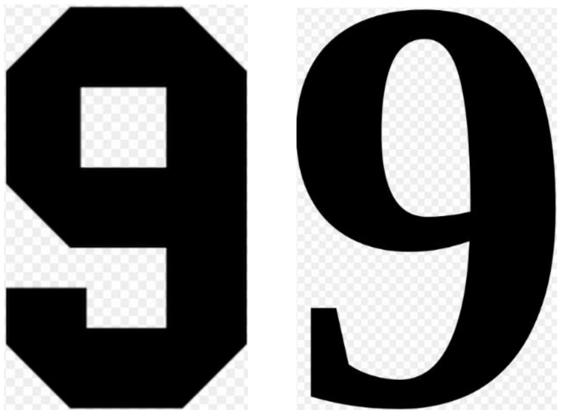
NORMALISATION AND COMPARISON

Finally image is represented by feature vector(array C) of constant dimension **n** x **N** x **M** and **does not depend** on the image dimensions, meaning we can compare images of different dimensions. Before we can compare images we have to normalize their feature vectors(array C). Each element of array C is divided by sum of all elements squared individually. After normalization scalar product and sum of all elements squared individually are being calculated so we can compare images using cosine similarity.

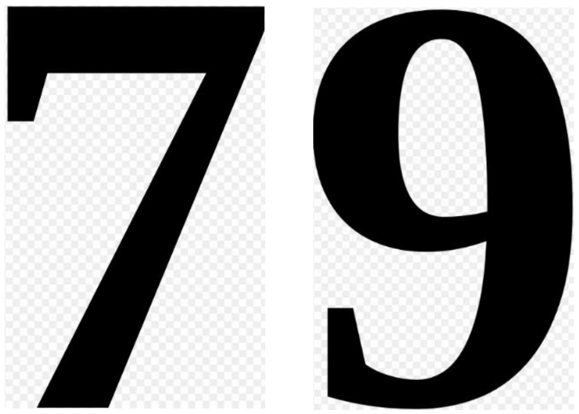
```
for(int i=0;i<block*n;i++){
    c1[i]=c1[i]/powc1;      // normalisation
    c2[i]=c2[i]/powc2;
    skalar+=c1[i]*c2[i];      //scalar product
    pownorm1+=c1[i]*c1[i];      //sum of all elements squared
    pownorm2+=c2[i]*c2[i];      //individually after normalisation
    }

return skalar/(sqrt(pownorm1)*sqrt(pownorm2)); //cosine similarity
```

Example of comparison:

We are going to compare these images with parameters b[]=3x2 and n=4



```
0.9464745
```



```
0.73678744
```

Conclusion:

If we had a database where we could compare our image with the entire database, the one with the highest value would correspond to the shape of our image.