Assignment 1

Mustafa Emre Başar, 21726999
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b21726999@cs.hacettepe.edu.tr

March 27, 2021

1 Introduction

In this assignment we were supposed to implement plate detector using Hough transform. The steps we should follow were ;

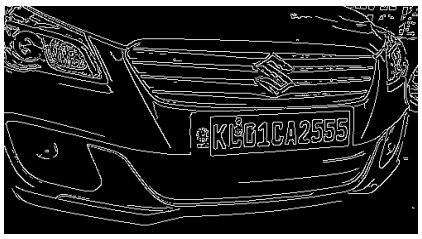
- Edge detection
- Line detection with Hough transform
- Plate detection
- IOU calculation

Detailed explanation of steps are examined in further sections.

2 Edge Detection

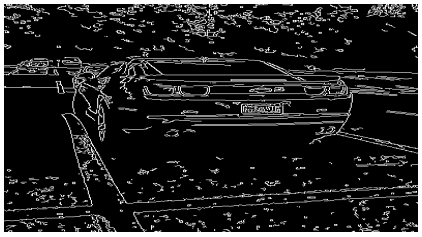
I used Canny Edge detection since it pans out a better result in aspects of error and noise reduction. I used 75 and 150 threshold values since we are trying to find a rectangular kind of shape. I aimed to find major edge of the rectangle with bigger threshold and the minor edge falls between the values of 75 and 150 and since they are connected to major edge they will be counted as an edge also. I didn't want to give both threshold values a small or high value since it might end up with a lot of unnecessary or no values at all. Some example results can be seen below;

Edges;

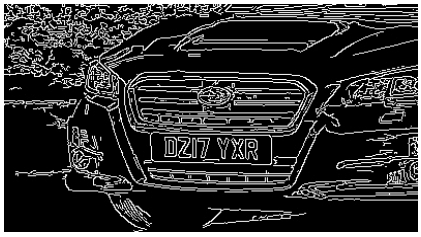


(a) Caption1

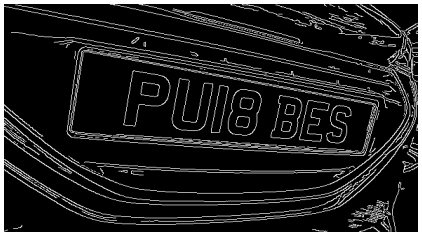
(b) Caption 2



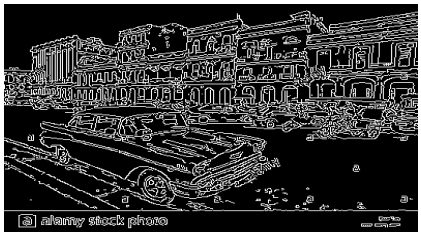
(c) Caption 2



(d) Caption 2

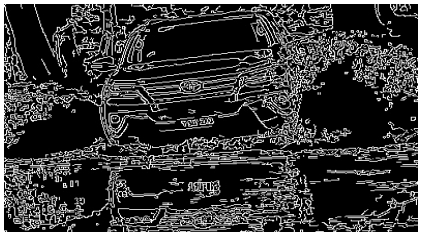


(e) Caption 2



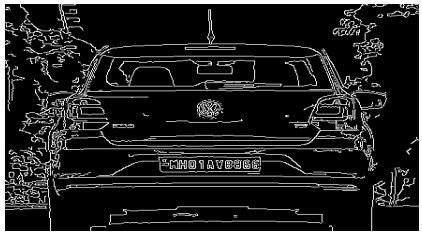
(f) Caption 2

(g) Caption 2



(h) Caption 2

(i) Caption 2



(j) Caption 2

3 Hough Transform

First thing I did in Hough transform was to create an array called Theta that stores the degrees from 0 to 179. Later I used that array to calculate sin and cos of these degrees.

```
theta = np.zeros(shape=(180,),dtype=int)
for i in range(180):
    theta[i] = i
cos = np.cos(theta*(pi/180))
sin = np.sin(theta*(pi/180))
```

After that I calculated rho range that is the maximum distance at which a line can be located furthest from the center. I initialized an array called accumulator according to that rho range. Its' size is the two times of the rho range since a line can be located on both negative or positive side of the center.

```
rho_range = round(math.sqrt(edge.shape[0]**2 + edge.shape[1]**2))
accumulator = np.zeros((2 * rho_range, len(theta)), dtype=np.uint8)
```

After that an array called peak_values is declared. It holds indices of peak pixels in the given edges image. Since numpy.where returns the X and Y indices as two separate array I gathered them in an array called coordinates as pairs.

```
peak_values = np.where(edge == 255)
coordinates = list(zip(peak_values[0],peak_values[1]))
```

After that I iterated through this coordinates array and for each coordinate at each theta, a rho value is calculated according to the following rule;

$$\rho = xcos\theta + ysin\theta \tag{1}$$

```
for p in range(len(coordinates)):
    for t in range(len(theta)):
        rho = int(round(coordinates[p][1] * cos[t]
              + coordinates[p][0] * sin[t]))
        accumulator[rho, t] += 1
```

This rho value and theta are used as indices to vote up the corresponding place in accumulator array by one.

After these operations accumulator array stores the the line data in the image but since we got complex images the first result was a mess as shown below;

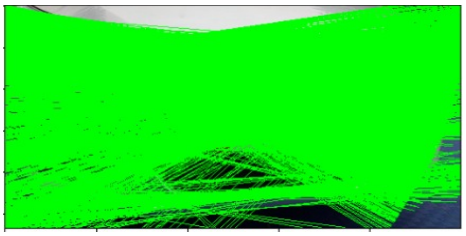


Figure 2: Hough for Cars0

First thing I did was to limit to theta values. the limitation are 0-10 and 170-180 for vertical lines , 85-95 for horizontal lines.

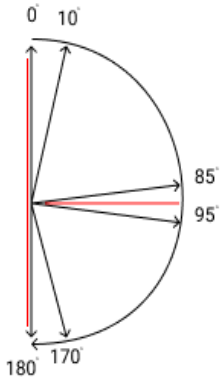


Figure 3: Thetas for Line Selection

```
coordinates = []
    for i in range(0,len(accumulator)):
        for j in range(0,len(accumulator[i])):
            if( j in range(0,10) or j  in range(170,180)):
                if(accumulator[i][j] > 40):
                    coordinates.append((i,j))
            if (j in range(85, 95) ):
                if (accumulator[i][j] > 100):
                    coordinates.append((i, j))
```

After that, to reduce the intensity of the lines in small regions, I used rho value. From all the lines I picked every nth line.Here n is the gap value which means before pick the line I first check if there is already a line picked with similar theta and that has a rho value in range of (n - 50 ,n+ 50).

```
final_coordinates = []
    final_coordinates.append(coordinates[0])
    for i in coordinates:
        addable = True
        theta = i[1]
        if(theta in range(0,10) or theta
in range(170,180) or theta  in range(85,95)):
            iter = len(final_coordinates)
            for j in range(0, iter):
                elm = final_coordinates[j]
                elm_rho = elm[0]
                if(abs(theta - elm[1]) > 70 ):
                    pass
                elif((i[0] - 30 <= elm_rho <= i[0]+ 30)):
                    addable = False
            if(addable):
                final_coordinates.append(i)
```

These modifications leads to a more proper results as shown below;

Figure 4: Result

Later I classified the lines as vertical and horizontal and picked two lines from each, 4 at total. The selection was nearly arbitrary, I end up with this conclusion by trial and error. The logic behind it, as far as e have seen most of the plates located in the mid sections of the image so I tried to pick the lines around the middle, for both vertical and horizontal.

```
if (len(recVer) == 2):
    line1 = recVer[0] line2 = recVer[1]
elif (len(recVer) > 2):
    line1 = recVer[len(recVer)//2 - 1]
    line2 = recVer[len(recVer)//2 + 1]
if (len(recHor) == 2):
    line3 = recHor[0] line4 = recHor[1]
elif (len(recHor) > 2):
    line3 = recHor[len(recHor) //2 - 1]
    line4 = recHor[len(recHor) //2 + 1]
```

I append these lines into an array called recCoord.I iterate through this array and convert each line from Polar to Cartesian coordinates and after that findLines[1] method I got the line equation coefficients of each line. Coefficients are used by a method called intersection[2]. This method is used to find intersection points of correct pairs of lines and later I draw the rectangle by using these two intersection points.

```
for i in range(0, len(recCoord)):
    rho = recCoord[i][0]
    theta = recCoord[i][1]
    a = np.cos(theta*(pi/180))
    b = np.sin(theta*(pi/180))
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    line = findLine( (x1,y1),(x2,y2))
    lines.append(line)
if(len(lines) == 4):
    i1 = intersection(lines[0],lines[2])
    i2 = intersection(lines[1],lines[3])
```

Here since vertical and horizontal lines appended respectively, I know which lines should be paired in order to find intersection points.

Choosing the rectangle;

Figure 5: Hough for Cars0

Here is the mentioned findLine and intersection methods;

```
def findLine(p1, p2):
    A = (p1[1] - p2[1])
    B = (p2[0] - p1[0])
    C = (p1[0]*p2[1] - p2[0]*p1[1])
    return A, B, -C
```

```
def intersection(L1, L2):
    D  = L1[0] * L2[1] - L1[1] * L2[0]
    Dx = L1[2] * L2[1] - L1[1] * L2[2]
    Dy = L1[0] * L2[2] - L1[2] * L2[0]
    if D != 0:
        x = Dx / D
        y = Dy / D
        return int(x),int(y)
```

4 IOU

I defined a function called readXmlandCalculate that reads the corresponding XML file and calculate the IOU value. The function shown below;

```
def readXmlandCalculate(input,area,i1,i2 ):

file_name = input + ".xml"
full_file = os.path.join('annotations', file_name)

root = et.parse(full_file).getroot()
xmin = int(root.findall('object/bndbox/xmin')[0].text)
ymin = int(root.findall('object/bndbox/ymin')[0].text)
xmax = int(root.findall('object/bndbox/xmax')[0].text)
ymax = int(root.findall('object/bndbox/ymax')[0].text)

actual_area = (xmax - xmin) * (ymax - ymin)

overlap1_x = max(xmin,i1[0])
overlap1_y = max(ymin,i1[1])

overlap2_x = min(xmax,i2[0])
overlap2_y = min(ymax,i2[1])


overlap_area = abs((overlap2_x-overlap1_x))*abs((overlap2_y-overlap1_y))

if(overlap_area > actual_area):
    overlap_area = 0

result = overlap_area / (area + actual_area - overlap_area)
if(result < 0):
    result = 0
print("IOU = ", result)
```

The average IOU value was : 0.034557.

Since the picking lines was based on assumptions in my implementation, success rate is quite low. I believe that using Hough Transformation was not the right method for such a task.

5 Good Examples

(a) Cars 9, IOU : 0.34426

(b) Cars 12, IOU : 0.18547

(c) Cars 14, IOU : 0.15100

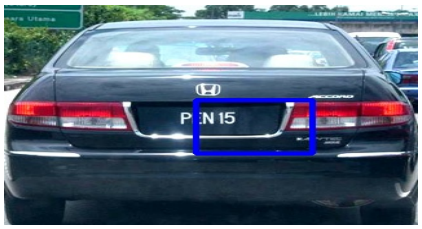
(d) Cars 25, IOU : 0.20678

(e) Cars 39, IOU : 0.30373

(f) Cars 43, IOU : 0.28600

(g) Cars 96, IOU : 0.20890

(h) Cars 102, IOU : 0.38196



(i) Cars 144, IOU : 0.35496

(j) Cars 147, IOU : 0.20277

6 Bad Examples

(a) Cars 3

(b) Cars 22

(c) Cars 34

(d) Cars 35

(e) Cars 134

These 5 example are not the only ones as there are more failed outputs. The reason behind this faults is mostly because the images are very different from each other and as I mentioned before, as I pick the lines predictive, it nearly impossible to work properly for every image. Also since plate areas are small compare to the whole image, edge detection thresholds are low and that leads to noises in the images and that makes it even harder to detect the correct line.

7 Conclusion

To conclude, as I said before, I think Hough Transform is quite incapable for this job. Learning methods should be included for such a task.

8 References

[1] https://stackoverflow.com/questions/20677795/how-do-i-compute-the-intersection-point-of-two-lines
[2] https://stackoverflow.com/questions/20677795/how-do-i-compute-the-intersection-point-of-two-lines

Both references, refers to the same page