Traffic sign detection and classification

Filipe, Francisco
up201604601@fe.up.pt

Fernandes, Pedro
up201603846@fe.up.pt

Neto, Pedro
up201604420@fe.up.pt

April 2020

Contents

1 Description of the Proposed Global Solution

The implemented solution for this problem involves two main stages. These two stages are color detection and shape detection.

1.1 Color Detection

For the color detection stage, since it's only needed to find blue and red traffic signs, color masks were applied to the selected image. For that, it was necessary to define a range of colors detectable by the mask. Once these colors were defined, the procedure was very simple:

- Convert the original image from RGB to HSV.
- Create three different masks (red, blue and red+blue)
- Apply the masks to the original image

Once these steps were over, the color detection stage was complete and besides still having the original image, there were now three additional images containing the red, blue and red+blue components of the original image.

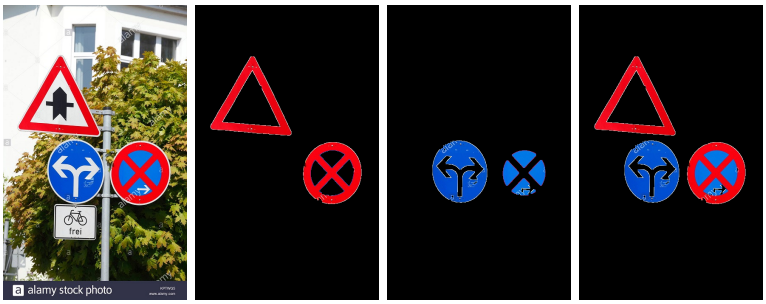

Figure 1: Resulting images of the color detection stage

1.2 Shape Detection

This stage is responsible for detecting the shapes within the original image. For that, it is necessary to use the information gathered in the color detection stage.

- Firstly, the previously obtained images are gray scaled.
- Afterwards, generate the contours of each gray scaled image.
- Moreover, the red+blue contour image is used to detect all the shapes within the image. For each shape, depending on the number of sides detected, it classifies it as a triangle, rectangle/square or circles.
- After detecting all the shapes, using the red and blue images and the contours found previously, it labels each shape as either blue or red.

Once this stage ends, the image analysis and detection of traffic signs is complete. The implemented solution also allows for the detection of stop signs, slanted signs, bad lighting conditions and several signs in one image .

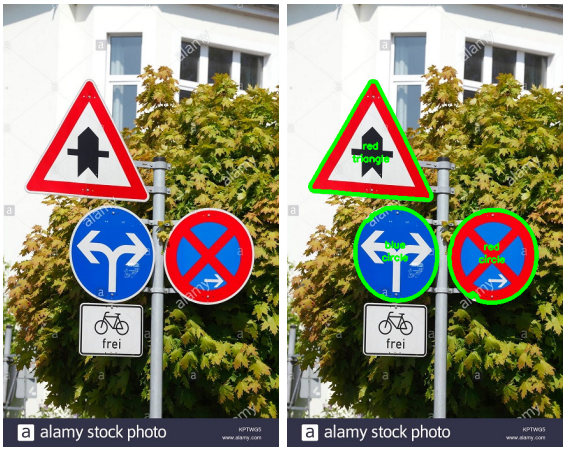

Figure 2: Results of the shape detection stage(original image vs result image)

2 Efficacy of the Used Methods

2.1 Main Problems

There are two problems that could be improved, even though they rarely interfere with the results.

- Since the program is built on a contour-based approach, the classification of shapes is made by counting the number of sides. This sometimes leads to false positives, such as considering any four sides connected a rectangle.
- For a shape to qualify as a circle, the requirement is to have a large number of "sides". One of the big parts of the project was qualifying blue circles, but by using this method, it sometimes happens that in pictures with low contrast, our code reads a piece of the sky as a blue circle.

2.2 Proposed Solutions

For the two problems above, we consider the following solutions:

- To fix improve the classification based on the number of sides, we could perhaps also consider inner angles to better read a shape.
- Using a different system to discover circles, such as *Hough Circles*, can fix these false positives.

3 Status of the Proposed Method

3.1 Degree of Fulfillment of the Aims

For this project, we aimed for an improved version of a sign detector. Besides complying to all the basic version features, we added the following improvements through the use of these techniques:

1. Dealing with more than one traffic signs in an image - by starting to discover all the relevant red and blue contours and organizing them into an array, we could later iterate through it and classify each contour into a shape

2. Recognition of a STOP sign, differentiating it from a circular sign - the STOP sign has an octagonal shape and was recognized as an eight point circle. To distinguish it, we created an adaptable threshold which allows typical circles not to fall into an eight point circle unless it is indeed an octagonal shape.

3. Poorly illuminated images - poorly illuminated images are pre-processed. The pre-processing involves two different strategies to improve the image quality: improve the contrast by using the CLAHE techinque or improve the lighting by converting the color from RGB to YUV and then apply an histogram equalization to the Y channel.

4. Correct interpretation of slanted signs - due to a system of classification by counting the corners, even slanted signs can be classified correctly as long as all the edges are visible

4 Performance

As previously stated, the shape detection is based on the contours found in the masked images. If, by any chance, the image contains an object whose color lies inside the color mask spectrum, that object will be analysed by the shape detection algorithm and, most likely, classified as a traffic sign (even though it isn't one)

Moreover, the performance is also affected by the quality and size of the images. Small images proved to be hard to analyse.

Despite these edge cases where the program has trouble detecting traffic signs, it proved to have a good accuracy overall. The following images illustrates the behaviour of the program in various scenarios.

Figure 3: Recognition of a STOP sign

Figure 4: Recognition of a red circular sign with bad lighting

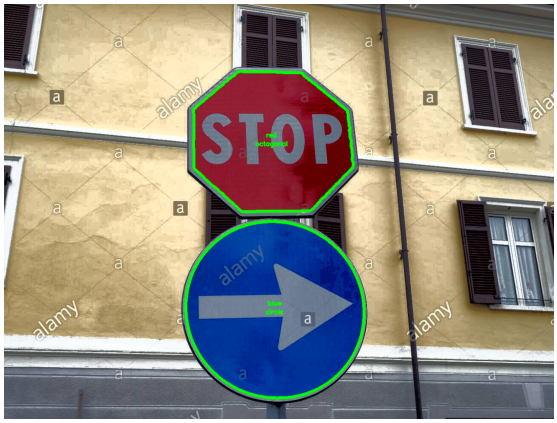

Figure 5: Recognition of a more than one sign with bad lighting conditions

Appendices

A Running Instructions

This section is destined to anyone who wants to run the project. Once you've downloaded the project, open the command line and navigate to the 'src' folder of the project. Once inside that folder, to run the program enter the following commands:

```
$ pip3 install -r requirements.txt
$ python3 main.py [-h] [-f FILE] [-q QUALITY]
```

Where:

- **-h:** Opens the help dialog with the running instructions.
- **-f FILE:** If present, the program will try to analyse the given image (FILE is the relative path to the image file). Otherwise the program will try to access the computer's webcam, take a picture and proceed with the analysis of that picture.
- **-q QUALITY:**A boolean attribute that specifies whether or not the image has a good quality. If not present the program assumes that the image is of a good quality,

B Code

- main.py - Entry point to the program. Contains functions to read images from files and cameras attached to the computer. The main function reads the image from the given source, and provides it to the color and shape detection functions.
- color_detection.py - Contains functions to perform color detection and segmentation, specifically for red and blue colors. Also includes image processing functions to fix contrast and lightning using histogram equalization.
- shape_detection.py - Contains shape detection functions using contours, and utilitary functions to write the color and shape of the signs on the image.

B.1 main.py

```
import argparse
import numpy as np
import cv2
import color_detection
import shape_detection
```

```
6
7
8  def readImageFromFile(fileName):
9      return cv2.imread(fileName, cv2.IMREAD_COLOR)
10
11
12 def readImageFromCamera():
13     cap = cv2.VideoCapture(0)
14     if not (cap.isOpened()):
15         print("Could not open video device")
16         return None
17     ret, frame = cap.read()
18     cap.release()
19     return frame
20
21
22 if __name__ == "__main__":
23     # Parse CLI arguments
24     parser = argparse.ArgumentParser(description='Process a road
       image.')
25     parser.add_argument(
26         '-f', '--file', help="Read the image from a file.", type=
       str, required=False)
27     parser.add_argument(
28         '-q', '--quality', help="Estimated quality of the image.
       True indicates good quality, false otherwise.", type=str, nargs
       ="?", const="True", default="True", required=False)
29     args = parser.parse_args()
30
31     image = readImageFromFile(
32         args.file) if args.file else readImageFromCamera()
33
34     if image is None:
35         print("Image not found!")
36         quit()
37
38     if not args.quality == "True":
39
40         # Color correction
41         contrast_image = color_detection.fixBadContrast(image)
42         lighting_image = color_detection.fixBadLighting(image)
43
44         # Color detection in both regular and bad lighting
45         contrastColorRes, contrastBlueRes, contrastRedRes =
       color_detection.colorDetection(
46             contrast_image, "bad")
47         lightingColorRes, lightingBlueRes, lightingRedRes =
       color_detection.colorDetection(
48             lighting_image, "bad")
49
50         # Shape detection in both regular and bad lighting
51         contrast_regular_signs = shape_detection.shapeDetection(
52             contrast_image, contrastColorRes, contrastRedRes,
       contrastBlueRes)
53         lighting_regular_signs = shape_detection.shapeDetection(
54             lighting_image, lightingColorRes, lightingRedRes,
       lightingBlueRes)
```

```
55
56          cv2.imshow('Fixed Contrast', contrast_regular_signs)
57          cv2.imshow('Fixed Lighting', lighting_regular_signs)
58      else:
59          regColorRes, regBlueRes, regRedRes = color_detection.
    colorDetection(
60              image, "regular")
61
62          regular_signs = shape_detection.shapeDetection(
63              image, regColorRes, regRedRes, regBlueRes)
64
65          # Print Results
66          cv2.imshow('Regular', regular_signs)
67
68      cv2.waitKey(0)
69      cv2.destroyAllWindows()
```

B.2 color_detection.py

```
1  import cv2
2  import numpy as np
3
4  lighting = {
5      'regular_lower_blue' : np.array([100, 160, 50]),
6      'regular_upper_blue' : np.array([135, 255, 255]),
7      'regular_lower_red1' : np.array([0,180,120]),
8      'regular_upper_red1' : np.array([10,255,255]),
9      'regular_lower_red2' : np.array([170,180,120]),
10     'regular_upper_red2' : np.array([180,255,255]),
11     'bad_lower_blue' : np.array([100, 150, 50]),
12     'bad_upper_blue' : np.array([140, 255, 255]),
13     'bad_lower_red1' : np.array([0,70,70]),
14     'bad_upper_red1' : np.array([10,255,255]),
15     'bad_lower_red2' : np.array([170,70,70]),
16     'bad_upper_red2' : np.array([180,255,255])
17 }
18
19
20 def fixBadContrast(image):
21     img = image.copy()
22
23     lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
24
25     clahe = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))
26     lab[...,0] = clahe.apply(lab[...,0])
27
28     contrast = cv2.cvtColor(lab, cv2.COLOR_LAB2BGR)
29
30     return contrast
31
32
33 def fixBadLighting(image):
34
35     img_yuv = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)
36
37     # equalize the histogram of the Y channel
38     img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])
```

```
39
40      # convert the YUV image back to RGB format
41      lighting = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
42
43      return lighting
44
45
46  # Apply image segmentation, in order to get a result with only blue
        and red colors
47  def colorDetection(image, type):
48      hsvImg = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
49
50      lower_blue = lighting[type + '_lower_blue']
51      upper_blue = lighting[type + '_upper_blue']
52
53      lower_red = lighting[type + '_lower_red1']
54      upper_red = lighting[type + '_upper_red1']
55
56      mask1 = cv2.inRange(hsvImg, lower_red, upper_red)
57
58      # upper mask (170-180)
59      lower_red = lighting[type + '_lower_red2']
60      upper_red = lighting[type + '_upper_red2']
61
62      mask2 = cv2.inRange(hsvImg, lower_red, upper_red)
63
64      # join my masks
65      redMask = mask1+mask2
66
67      # Threshold the HSV image to get only blue colors
68      blueMask = cv2.inRange(hsvImg, lower_blue, upper_blue)
69
70      redMask = cv2.morphologyEx(redMask, cv2.MORPH_OPEN, np.ones
        ((3,3),np.uint8))
71      redMask = cv2.morphologyEx(redMask, cv2.MORPH_DILATE, np.ones
        ((3,3),np.uint8))
72
73      blueMask = cv2.morphologyEx(blueMask, cv2.MORPH_OPEN, np.ones
        ((3,3),np.uint8))
74      blueMask = cv2.morphologyEx(blueMask, cv2.MORPH_DILATE, np.ones
        ((3,3),np.uint8))
75
76      # Bitwise-AND mask and original image
77      redRes = cv2.bitwise_and(image, image, mask=redMask)
78
79      # Bitwise-AND mask and original image
80      blueRes = cv2.bitwise_and(image, image, mask=blueMask)
81
82      mask = blueMask + redMask
83
84      res = cv2.bitwise_and(image, image, mask=mask)
85
86      return res, blueRes, redRes
```

B.3 shape_detection.py

```
1  import cv2
```

```
import numpy as np
import imutils


def writeText(img, text, size, x, y):
    textsize = cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX, size
    , 2)[0]
    # get coords based on boundary
    textX = int((x - (textsize[0] / 2)))
    cv2.putText(img, text, (textX, y),
                cv2.FONT_HERSHEY_SIMPLEX, size, (0, 255, 0), 2)


def detectShape(c):
    # Compute perimeter of contour and perform contour
    approximation
    shape = ""
    peri = cv2.arcLength(c, True)
    # approx = cv2.approxPolyDP(c, 0.04 * peri, True)
    approx = cv2.approxPolyDP(c, 0.011 * peri, True)

    if len(approx) > 4 and len(approx) < 8:
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)

    # Triangle
    if len(approx) == 3:
        shape = "triangle"

    # Square or rectangle
    elif len(approx) == 4:
        (x, y, w, h) = cv2.boundingRect(approx)
        ar = w / float(h)

        # A square will have an aspect ratio that is approximately
        # equal to one, otherwise, the shape is a rectangle
        shape = "square" if ar >= 0.95 and ar <= 1.05 else "
    rectangle"

    #Stop
    elif len(approx) == 8:
        shape = "octagonal"

    # Otherwise assume as circle or oval
    else:
        shape = "circle"

    return shape


def shapeDetection(image, colorRes, redRes, blueRes):
    img = image.copy()
    _, _, v = cv2.split(colorRes)
    _, _, redV = cv2.split(redRes)
    _, _, blueV = cv2.split(blueRes)

    # Find contours and detect shape
    cnts = cv2.findContours(v, cv2.RETR_EXTERNAL, cv2.
```

```
    CHAIN_APPROX_SIMPLE)
    cnts = cnts[0] if len(cnts) == 2 else cnts[1]

    blueCnts = cv2.findContours(
        blueV, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    blueCnts = blueCnts[0] if len(blueCnts) == 2 else blueCnts[1]

    redCnts = cv2.findContours(
        redV, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    redCnts = redCnts[0] if len(redCnts) == 2 else redCnts[1]

    centers = []
    for c in cnts:
        area = cv2.contourArea(c)

        # Ignore very small areas
        if area > 100:        # Identify shape
            shape = detectShape(c)

            if shape != "":
                # Find centroid and label shape name
                M = cv2.moments(c)
                cX = int(M["m10"] / M["m00"])
                cY = int(M["m01"] / M["m00"])
                centers.append([cX, cY])
                writeText(img, shape, 0.5, cX, (cY + 10))
                cv2.drawContours(img, [c], 0, (0, 255, 0), 6)

    writeColor(img, blueCnts, centers, "blue")
    writeColor(img, redCnts, centers, "red")

    return img

# Label the sign's color
def writeColor(img, cnts, centers, color):
    for c in cnts:
        area = cv2.contourArea(c)

        if area > 100:        # Identify shape
            M = cv2.moments(c)
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])

            if [cX, cY] in centers:
                writeText(img, color, 0.5, cX, (cY - 10))
```