Computer vision

Project Phase 1

Name: Ibrahim Ashraf Ibrahim 19P7817

AbdulRahman Ayman Mahmoud Abbas ElSherbiny 19P6458
Sohayla Ihab Abdelmawgoud Ahmed Hamed 19P7343

Contents

GitHub link: https://github.com/PerfectionistAF/SVHN-Recognition.git

1.0The methods we used through the project

1-Load dataset
2-Localize directory
3-Calling Canny edge
4-we passed the resulting image form canny image function to localize digits function

5-Inside the function Localize digits we will use estimate digit area
6-Getting percentage of intersection

2.0Here we will explain each function used in the code in details :

2.1The first function is(Loading DataSet)

Here it's code:

```
def loadDataSet(file_path: str):
    f = open(file_path, 'r')
    data = json.load(f)
    return data
```

2.1.1Explanation of the function :

The function takes in a file path to a dataset in JSON format and uses the json library to load it into a variable called data it then returns the variable

This function loadDataSet() takes in a file path as an argument and returns the contents of that file in JSON format.

Here are the steps it follows:

1. It opens the file specified in the file_path argument using the built-in open() function in read-only mode ('r').

2. It reads the contents of the file using the json.load() function, which converts the JSON-formatted data inside the file to a Python object.

3. It returns the loaded data as a Python object.

So, essentially this function loads a JSON dataset from a given file path and returns it in a format that can be used by a Python program

2.2-The second function used is (Localize Images and Compute Accuracy)

Here it’s code:

```
def LocalizeDir(dataset, showSteps):
    percents = []
    i = 0
    total = len(os.listdir('testImages'))
    for filename in os.listdir("testImages"):
        i += 1
        loadPercent = (i/total)*100
        sys.stdout.write(f"\rLoading: [{'=' * math.floor(loadPercent/10)}{'
' * (10 - math.floor(loadPercent/10))}] "
                         f"{round(loadPercent, 1)}%")
        f = os.path.join("testImages", filename)
        imgReal = cv2.imread(f)

        myOutput = LocalizeDigits(
            CannyEdge(imgReal, showSteps=showSteps))
        realOutput = []

        for box in dataset[int(filename.split(".")[0]) - 1]['boxes']:
            realOutput.append((int(box['left']), int(box['top']),
int(box['width']), int(box['height'])))

        percent = getIntersectionPercentage(myOutput, realOutput)
        percents.append(percent)

        if showSteps:
            for (x, y, w, h) in myOutput:
                cv2.rectangle(imgReal, (x, y), (x + w, y + h), (0, 255, 0),
2)

            cv2.imshow('Localized', imgReal)
            cv2.waitKey(0)
            cv2.destroyAllWindows()

    return percents
dataSet = loadDataSet('training.json')
percentages = LocalizeDir(dataSet, showSteps=False)
print(f"\n\nAccuracy is: {round(sum(percentages)/len(percentages), 1)}%")
```

2.2.1Explanation of the function :

The Function iterates through all images in the testImages folder, applies the CannyEdge and LocalizeDigits functions to each image, compares the resulting bounding boxes to the ground truth values, and calculates the intersection over union (IoU) percentage for each image. Finally, the function returns a list of IoU percentages for all images.

The LocalizeDir function takes two arguments: dataset, which is a list of training images with labeled bounding boxes, and showSteps, a boolean flag indicating whether or not to display intermediate steps during processing.Within the function, the number of test images is determined and the progress is displayed using a simple progress bar. For each test image, the cv2.imread function is used to read the image from disk and then passed through an edge detection function called CannyEdge. The resulting edges are then passed to another function called LocalizeDigits which identifies and localizes any digits in the image.The localized output is then compared against the ground truth bounding box annotations for the corresponding image in the dataset. This comparison is done using the getIntersectionPercentage function. The percentage overlap between the predicted and ground truth bounding boxes is computed and appended to a list of per-image accuracies.If showSteps is set to True, the localized digits are drawn on the input image using OpenCV's cv2.rectangle function and displayed. Finally, the function returns the list of per-image accuracies.Outside the function, the loadDataSet function is called to load the training dataset from a JSON file. Then, the LocalizeDir function is called on the test dataset, and the average accuracy across all test images is computed and printed.

2.3-Inside the The second function (def LocalizeDir) there is another function used called (Canny Edge Detection) def CannyEdge(img, showSteps=False):

Here it’s code :

```
def CannyEdge(img, showSteps=False):

    # dst = cv2.fastNlMeansDenoisingColored(img, None, 10, 10, 7, 21)
    dst = cv2.bilateralFilter(img, 9, 75, 75)

    if showSteps:
        cv2.imshow('Noise Reduction', dst)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    gray = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)

    if showSteps:
        cv2.imshow('GrayScale', gray)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    thresh = cv2.Canny(gray, 50, 100)
    if showSteps:
        cv2.imshow('Canny', thresh)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    return thresh
```

First thing the function does os filter the image using a bilateral Filter the function then converts the filtered image to grayscale and applies the Canny algorithm to detect edges finally the function returns the image with canny edges

2.3.1Explanation of the function :

The given function CannyEdge(img, showSteps=False) is used to perform edge detection on an image using the Canny edge detection algorithm. Here's what the function does:It applies bilateral filtering or non-local means denoising on the input image to remove any noise present in the image.

If the showSteps parameter is set to True, it displays the filtered image using the OpenCV imshow() function and waits for a keystroke from the user before closing the window.The function then converts the filtered image to grayscale using the OpenCV cvtColor() function.

If the showSteps parameter is set to True, it displays the grayscale image using the imshow() function and waits for a keystroke from the user before closing the window.Finally, the function applies the Canny edge detection algorithm on the grayscale image with the threshold values of 50 and 100.

If the showSteps parameter is set to True, it displays the resulting edge-detected image using the imshow() function and waits for a keystroke from the user before closing the window.The function returns the resulting edge-detected image as output.

2.4- we passed the resulting image form canny image function to localize digits function :

Here it’s code:

```
def LocalizeDigits(img):
    contours, _ = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    minArea, maxArea = Estimate_digit_area(img.shape)
    finalContours = []
    for contour in contours:
        area = cv2.contourArea(contour)
        if minArea < area < maxArea:
            finalContours.append(cv2.boundingRect(contour))

    return finalContours
```

2.4.1Explanation of the function :

The function LocalizeDigits takes a grayscale image containing digits as input and performs the following operations to localize the digits:

1. Find the contours in the image using the cv2.findContours() function.
2. Estimate the minimum and maximum areas of the digit contours based on the image size using the Estimate_digit_area() function.
3. Iterate through each contour found in step 1 and check if its area falls within the estimated range of digit areas.
4. If the area of the contour is within the estimated range, add the bounding box of the contour to the finalContours list.
5. Return the finalContours list containing the bounding boxes of the digit contours.

2.5- Inside the function of localize digits we used a function called Estimating Digit Area

Here it's code :

```
def Estimate_digit_area(image_size):
    # Estimate the maximum and minimum sizes of the digits based on the
image size
    max_digit_height = int(image_size[0] * 0.8)  # assume maximum digit
height is 80% of the image height
    aspect_ratio = [0.38, 0.51, 0.54, 0.53, 0.55, 0.58, 0.53, 0.47, 0.57,
0.52]  # aspect ratio of digits 0-9
    # Assume maximum digit width is 90% of the image width, adjusted by the
maximum aspect ratio
    max_digit_width = int(image_size[1] * 0.9 * max(aspect_ratio))
    # Assume minimum digit height is 10% of the image height
    min_digit_height = int(image_size[0] * 0.1)
    # Assume minimum digit width is 10% of the image width, adjusted by the
minimum aspect ratio
    min_digit_width = int(image_size[1] * 0.1 * min(aspect_ratio))

    # Calculate the approximate maximum and minimum area of the digit
contours based on the estimated sizes
    max_digit_area = (max_digit_height * max_digit_width)
    min_digit_area = (min_digit_height * min_digit_width)

    return min_digit_area, max_digit_area
```

2.5.1Explanation of the function :

This function estimates the minimum and maximum areas of a digit contour in an input image based on the size of the image. The maximum digit height is assumed to be 80% of the image height, and the maximum digit width is assumed to be 90% of the image width adjusted by the maximum aspect ratio of digits 0-9. Similarly, the minimum digit height and width are assumed to be 10% of the image height and width, respectively, adjusted by the minimum aspect ratio of digits 0-9. These estimates are used to calculate the approximate minimum and maximum areas of a digit contour.

2.6 The last function used is Getting percentage of intersection :

Here it’s code:

```
def getIntersectionPercentage(myOutput, realOutput):
    global HarshAccuracy
    img1Temp = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)
    img1 = copy.deepcopy(img1Temp)
    img2 = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)
    allPercents = []

    for (x, y, w, h) in realOutput:
        cv2.rectangle(img1, (x, y), (x + w, y + h), 255, 2)

    for (x, y, w, h) in myOutput:
        cv2.rectangle(img2, (x, y), (x + w, y + h), 255, 3)

    interSection = cv2.bitwise_and(img1, img2)
    allPercents.append((np.sum(interSection == 255) /
                        (np.sum(img1 == 255) + np.sum(img2 == 255) -
np.sum(interSection == 255))) * 100)

    for shift in [[5, 0], [-5, 0], [0, 5], [0, -5]]:
        img1 = copy.deepcopy(img1Temp)
        for (x, y, w, h) in realOutput:
            cv2.rectangle(img1, (x + shift[0], y + shift[1]), (x + w +
shift[0], y + h + shift[1]), 255, 2)
        interSection = cv2.bitwise_and(img1, img2)
        if HarshAccuracy:
            allPercents.append((np.sum(interSection == 255) /
                                (np.sum(img1 == 255) + np.sum(img2 == 255)
- np.sum(interSection == 255))) * 100)
        else:
            allPercents.append((np.sum(interSection == 255) /
                                (np.sum(img1 == 255))) * 100)

    return max(allPercents)
```

2.6.1Explanation of the function :

The function starts by initializing an empty list called 'allPercents', which will be used later to store the intersection percentages calculated in each iteration.

It then draws rectangles on two blank images 'img1' and 'img2', one for each set of rectangular regions, using the cv2.rectangle() function. These rectangles represent the detected regions in 'realOutput' and 'myOutput'.

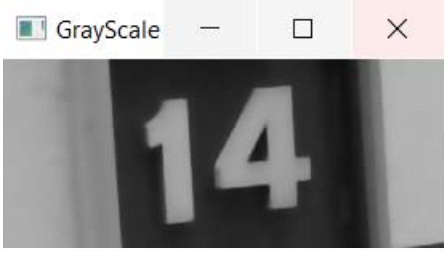
Next, it performs a bitwise AND operation between 'img1' and 'img2' to obtain the intersection between the two sets of rectangles. The result is stored in 'intersection'.

Then, the function calculates the intersection percentage by dividing the number of pixels that are white in the intersection area by the total number of pixels that are white in both areas. The resulting value is multiplied by 100 to get the percentage. The intersection percentage is appended to the 'allPercents' list.

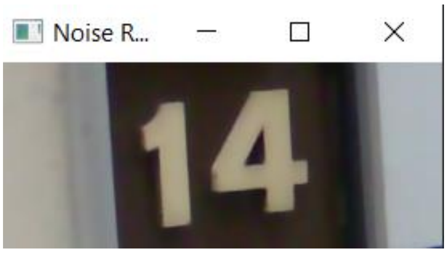
After that, the function creates four shifted versions of the 'img1' image, each shifted by 5 pixels in different directions. It then repeats the above process for each shifted image to calculate the intersection percentage, which is added to 'allPercents' list.

Finally, the function returns the maximum intersection percentage found in all iterations. This indicates the best match between the two sets of rectangular regions.

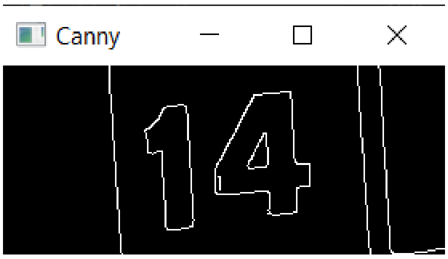
3.0Output snippets:



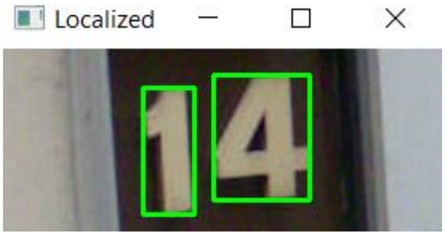
Grey scale image 1



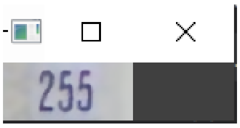
noise reduction 1



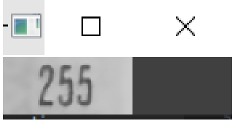
Canny detection 1



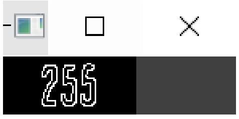
Localized image 1



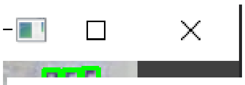
Grey scale image 2



noise reduction 2



Canny detection 2



Localized image 2