Sabancı Üniversitesi

EE 417

TERM PROJECT

Gürkan Talha Soylu 26883

Ahmet Enes Sılacı 22346

Contents

1. Importance of the Problem and Problem Definition

Image recognition technology has come a long way in recent years, but it still faces significant challenges. One of the main problems with image recognition is that it is a highly complex task that requires sophisticated algorithms and large amounts of data to train them. This is because images can be affected by various factors, such as lighting, angle, and perspective, which can make them difficult to interpret. Additionally, the recognition process must be able to accurately identify objects within an image, even if they are partially obscured or distorted. D

Despite these challenges, the importance of image recognition cannot be overstated since it is one of the 3 R problems of computer vision which can be seen in Figure 1. It is a critical technology for many industries, including transportation, security, and retail. In transportation, image recognition can be used for traffic monitoring, toll enforcement, and tracking of stolen vehicles. In security, it can be used to monitor public spaces and identify potential threats. In retail, it can be used for product recognition, inventory management, and customer tracking.

One specific application of image recognition is license plate recognition, which plays a crucial role in transportation and security. License plate recognition systems use image recognition technology to automatically identify and read license plates from vehicles. This technology can be used for a variety of purposes, such as monitoring traffic flow, enforcing toll payments, and tracking stolen vehicles. Additionally, license plate recognition systems can be integrated with other technologies, such as cameras and GPS, to provide real-time tracking of vehicles.

As the technology advances and more data becomes available, the accuracy and capabilities of image recognition systems will continue to improve, making it an essential tool for many industries. In addition, the increasing amount of data from cameras and other sources will be used to improve the recognition systems and make them more accurate. The future of image recognition is promising, and it will continue to play a vital role in improving safety, security, and efficiency in a variety of industries.

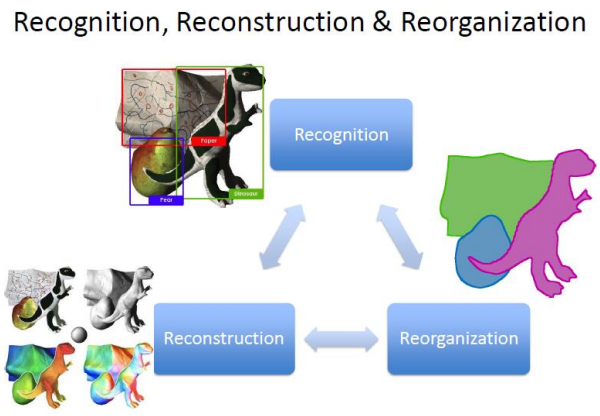

Figure 1

2. Problem Formulation and Solution Method

First and essential step of face recognition is plate detection. It requires a wide range of knowledge from image processing. Before we use our images, we need to pre-process them to be able to perform our task more precisely and get better results.

2.1 Gaussian Blurring:

First technique used for pre-processing is Gaussian Blurring. Gaussian Blurring is a widely used technique in image processing and computer vision, which is based on the mathematical concept of the Gaussian function. The Gaussian function is a bell-shaped curve that is defined by its mean and standard deviation. In image processing, the Gaussian function is used to create a kernel, which is a small matrix of numbers. The kernel is then convolved with the image. This process is done by multiplying each element of the kernel with the corresponding element of the image, and then summing the results. This effectively blurs the image by averaging the pixel values in the kernel's neighborhood.

The mathematical formula for a 1D Gaussian function can be seen in Figure 2 and corresponding functional graph can be seen in Figure 3. Since we are dealing with 2D matrices, we need to expand this formula into 2D. The 2D Gaussian function can be seen in Figure 4.

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Figure 2

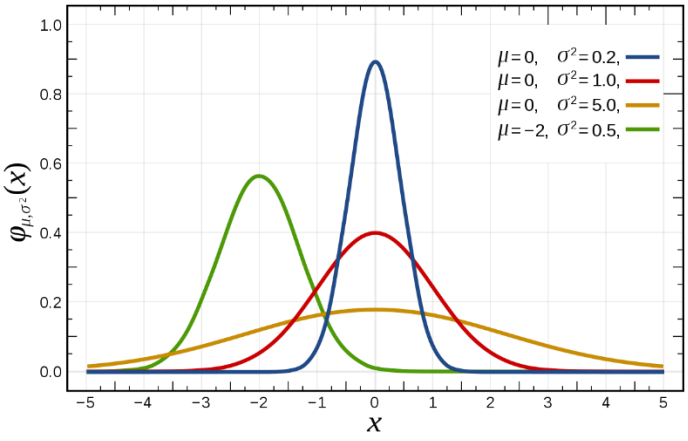

Figure 3

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

Figure 4

Where G(x,y) is the value of the function at position (x,y), π is the mathematical constant pi, σ is the standard deviation and e is the mathematical constant of Euler's number. The standard deviation, σ, controls the spread of the blurring effect. A larger value of σ results in a more extensive blurring effect, while a smaller value results in a more localized effect.

The kernel size is determined by the size of the neighborhood that the algorithm uses to average the pixel values. The larger the kernel, the more extensive the blurring effect. The kernel is obtained by sampling the 2D Gaussian function in a discrete grid, this is called the Gaussian kernel. The kernel is also symmetric, so it can be represented in a 1D array.

The convolution process can be implemented in various ways. The most common way is the spatial domain convolution where each output pixel is calculated as the sum of the element-wise product of the kernel and the corresponding region of the image. However, this process can be computationally expensive, especially for large images, and kernels. An alternative approach is to use the frequency domain convolution. This approach is based on the convolution theorem, which states that the convolution in the spatial domain is equal to the element-wise product in the frequency domain. The process of converting the image and the kernel to the frequency domain is called the Fourier Transform, and the operation of multiplying the two images in the frequency domain is called the point-wise product.

2.2 Image Sharpening:

Second technique used for pre-processing is Image Sharpening. Image sharpening is a technique used to enhance the details in an image by increasing the contrast of the edges. The process of sharpening an image can be mathematically represented by a convolution operation between the image and a kernel, also called a filter. The kernel is designed to enhance the edges in the image by emphasizing the intensity differences between adjacent pixels. Unsharp Masking is a widely used method for image sharpening, which is based on the difference between the original image and a blurred version of the image. The blurred version is obtained by convolving the original image with a low-pass filter, such as a Gaussian or a Box filter. The low-pass filter is designed to reduce the high-frequency details in the image, such as edges and textures, while preserving the low-frequency details, such as the overall structure of the image. The difference between the original image and the blurred version is then multiplied by a constant factor and added back to the original image.

The mathematical formula for the Unsharp Masking process can be represented as:

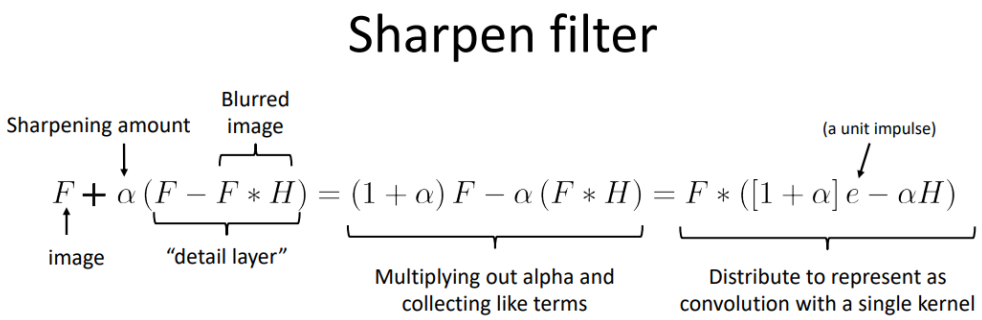

Figure 5

Where α is a constant that controls the amount of sharpening. The value of α is usually set between 0 and 1. A value of 0 will not produce any sharpening, while a value of 1 will produce the maximum sharpening. The term α*(Original image – Blurred image) is often called the Unsharp mask.

The Unsharp Masking process can also be represented as a convolution operation between the image and a kernel. The kernel is designed to enhance the edges in the image by emphasizing the intensity differences between adjacent pixels. An example kernel can be seen in Figure 5.

−1	−1	−1
−1	9	−1
−1	−1	−1

Figure 6

The Unsharp Masking process enhances the edges and fine details in the image by subtracting the low-frequency details, represented by the blurred version of the image, from the original image, and adding the result back to the original image. The constant factor, α, controls the amount of sharpening. A value of 0 will not produce any sharpening, while a value of 1 will produce the maximum sharpening. However, it is important to note that the Unsharp Masking process can produce over-sharpening or halo effect if the value of α is too high. This is why it is important to choose an appropriate value of α that balances the need to sharpen the image and the risk of over-sharpening. Also, the choice of the low-pass filter used for obtaining the blurred version of the image is important as the characteristics of the filter will have an impact on the final result.

2.3 Feature Extraction:

Pictures are full of information. Among the information pool of image, one information is mostly used in robotic vision, object detection and shape detection. This information is called edge and our object recognition application heavily relies on edge detection. There are many edge detection techniques but the one we used for our project is Canny Edge Detection since it is the most popular method and it proved its efficiency over years.

Technique was developed by John Canny in 1986, and it is known for its ability to detect edges with a high degree of accuracy and low level of noise. The Canny edge detector consists of several stages, including noise reduction, gradient computation, non-maximum suppression, and hysteresis thresholding.

The first stage is “Canny Enchancer” which includes noise reduction and gradient computation. The main steps of this stage can be seen in Figure 7. First, we perform image blurring or , in other words, noise reduction. For this purpose, Canny uses Gaussian Filter which uses mean as 0 and the standard deviation decision is left to performer. Second step of Canny Enhancer is edge enhancement. This step has 3 sub-steps. First sub-step is to calculating image gradients for both x and y axes. For this purpose, any type of gradient kernel can be user such as Sobel or Prewitt. After calculating gradient vertically and horizontally, we need to calculate edge strength as second sub-step. We calculate the edge strength like we calculate the hypotenuse of a triangle and we store resulting strength for each pixel as a separate matrix Es. As third sub-step, we need to find orientation of edge. This calculation is important because we will use the edge orientation to perform non-max suppression and hysteresis thresholding. After calculating orientation, we store it as separate matrix Eo.

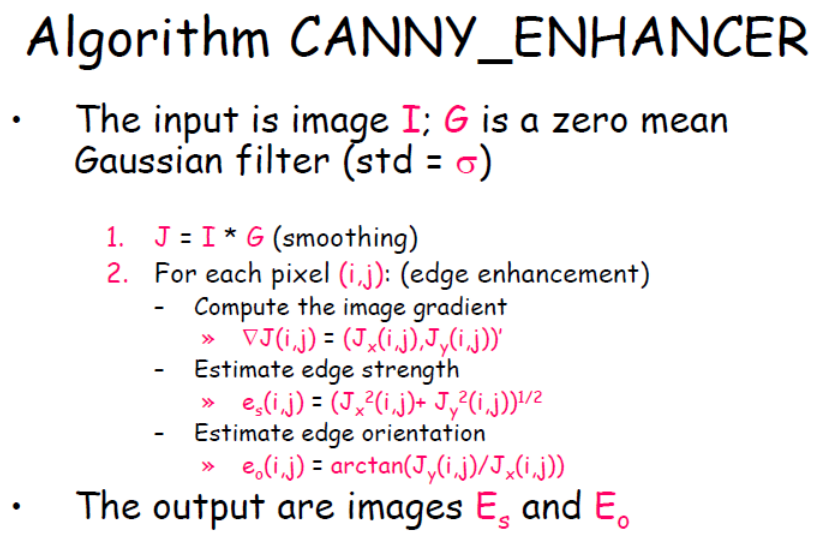

Figure 7

The second stage of Canny Edge Detection is “Non-max Suppression”. Sure, I can explain the non-maximum suppression step in a more detailed way. The non-maximum suppression step is an important part of the Canny edge detector algorithm. It is used to thin the edges and remove the pixels that are not a local maximum in the gradient magnitude. The idea behind this step is that, along an edge, the pixels with the highest gradient magnitude should be considered as the edge pixels, while the pixels with a lower gradient magnitude should be suppressed.

The process of non-maximum suppression starts by iterating through each pixel in the gradient magnitude image and checking its gradient direction. The gradient direction is computed during the gradient computation step, and it can have one of four possible values: 0, 45, 90, or 135 degrees. Once the gradient direction of a pixel is known, the algorithm compares the gradient magnitude of that pixel with the gradient magnitude of the pixels in its neighborhood. The neighborhood of a pixel is defined by the pixels that are in the same direction as the gradient direction of the pixel.

For example, if the gradient direction of a pixel is 0 degrees, the algorithm will compare the gradient magnitude of that pixel with the gradient magnitude of the pixels to the left and right of it. If the gradient magnitude of the pixel is greater than the gradient magnitude of its neighbors, the pixel is considered as an edge pixel, and its value is preserved. Otherwise, the pixel is suppressed. All possible directions can be seen in Figure 8.

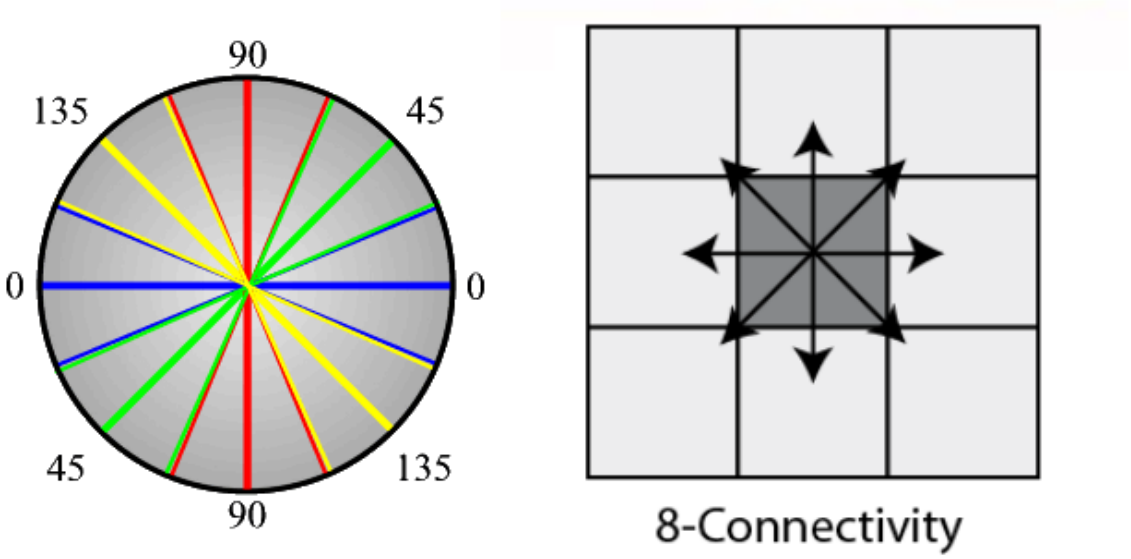

Figure 8

The same process is repeated for all other possible gradient directions (45, 90, and 135 degrees) and for all pixels in the gradient magnitude image. By the end of this step, the edge pixels are thinned, and the pixels that are not a local maximum in the gradient magnitude are removed.

It is important to note that the non-maximum suppression step can also be affected by the size of the neighborhood, which can be defined by the kernel size. A larger kernel size will result in a larger neighborhood, and this can make the edges thinner and more accurate. However, a larger kernel size can also increase the computation time and the risk of missing small edges.

Hysteresis thresholding is the final step of the Canny edge detector algorithm and it is used to remove the remaining pixels that are not part of an edge. This step is based on the idea that the edges in an image have a high gradient magnitude, while the non-edge pixels have a low gradient magnitude. The hysteresis thresholding step applies two threshold values, t_high and t_low, to the gradient magnitude image.

The pixels with a gradient magnitude greater than t_high are considered as strong edges and are retained in the final edge map. The pixels with a gradient magnitude less than t_low are considered as non-edges and are removed from the final edge map. The pixels with a gradient magnitude between t_low and t_high are considered as weak edges, and they are retained in the final edge map if they are connected to a strong edge pixel.

The process of hysteresis thresholding can be mathematically represented as follows:

```
If (Gradient_magnitude > t_high)
    Edge_map(i, j) = 1;
else if (Gradient_magnitude < t_low)
    Edge_map(i, j) = 0;
else if (Gradient_magnitude >= t_low && Gradient_magnitude <= t_high)
    Edge_map(i, j) = check_neighbors(i, j);
```

The "check_neighbors" function in the above representation is used to check if a pixel with a gradient magnitude between t_low and t_high is connected to a strong edge pixel. This is done by iterating through the 8-neighbors of a pixel and checking if any of them have a gradient magnitude greater than t_high. If at least one neighbor has a gradient magnitude greater than t_high, the pixel is considered as part of an edge and is retained in the final edge map, otherwise, it is removed.

The choice of threshold values t_high and t_low is important and can affect the results of the Canny edge detector. A higher value of t_high will result in fewer edges being detected, while a lower value will result in more edges being detected. Similarly, a higher value of t_low will result in more non-edges being retained, while a lower value will result in fewer non-edges being retained.

2.4 Finding the Contours:

Fourth technique is findContours function provided by OpenCV library. The findContours function is a part of the OpenCV library and is used to detect contours in an image. Contours are defined as the boundaries of an object in an image, and they can be represented as a sequence of points. The findContours function can detect contours in both binary and grayscale images. The findContours function takes in the following parameters:

source image: This is the image in which the contours need to be detected.

contour retrieval mode: This parameter specifies the type of contours that need to be detected. There are four types of contours:

CV_RETR_EXTERNAL: retrieves only the extreme outer contours.

CV_RETR_LIST: retrieves all of the contours without any hierarchical relationships.

CV_RETR_CCOMP: retrieves all of the contours and organizes them into a two-level hierarchy: external contours and holes.

CV_RETR_TREE: retrieves all of the contours and reconstructs a full hierarchy of nested contours.

contour approximation method: This parameter specifies the method used to approximate the contours. There are three types of approximation methods:

CV_CHAIN_APPROX_NONE: stores absolutely all the contour points.

CV_CHAIN_APPROX_SIMPLE: compresses horizontal, diagonal, and vertical segments and leaves only their end points.

CV_CHAIN_APPROX_TC89_KCOS: applies one of the flavors of the Teh-Chin chain approximation algorithm.

The first step in the findContours function is to convert the image into a binary image using a thresholding method. This is done by applying a threshold value to the image such that all pixels with a value greater than the threshold are set to 255 (white) and all pixels with a value less than the threshold are set to 0 (black).

The second step is to perform a connected component analysis on the binary image. This is done by iterating through all the pixels in the image and checking if they are connected to any other pixels with the same value (255 for white pixels and 0 for black pixels). The connected components are then represented as a sequence of points, which form the contours.

The third step is to perform a contour simplification. This is done by approximating the contours with a simpler representation. The most common method used for this is the Douglas-Peucker algorithm, which is a recursive algorithm that approximates a contour with a new set of points. The algorithm starts with the first and last point of the contour, and it

iteratively adds points that are farther than a certain distance (epsilon) from the line segment that connects the first and last point. The findContours function returns a list of contours, where each contour is represented as a NumPy array of (x,y) coordinates of the contour points.

2.5 Bounding Box Drawing:

Fifth technique is boundingRect function provided by OpenCV library. The cv2.boundingRect() function in the OpenCV library is used to calculate the minimum area rectangle that encloses a set of points. This rectangle is represented by the top-left corner coordinates (x, y) and the width and height of the rectangle. The points can be represented as an array of x and y coordinates, or as a contour. The function takes in a single argument, which is the set of points or contour, and returns a tuple of four values representing the (x, y) coordinates of the top-left corner and the width and height of the rectangle. This function is useful for finding the smallest rectangle that encloses an object in an image, for example, for object detection or image segmentation tasks. It is often used as a preprocessing step before applying more complex algorithms.

Mathematically, the function calculates the rectangle by finding the minimum and maximum values of the x and y coordinates of the points, and then using these values to define the width and height of the rectangle. It can be represented as:

rect = (minX, minY, maxX-minX, maxY-minY)

Where minX,minY, maxX and maxY are the minimum and maximum values of X and Y coordinates respectively.

2.6 Thresholding:

Sixth technique is thresholding. The thresholding operation is used to convert a grayscale image into a binary image, where all pixels with a value greater than a threshold value are set to 255 (white) and all pixels with a value less than the threshold value are set to 0 (black). The function cv2.threshold() takes in four arguments:

-The first argument is the input image

-The second argument is the threshold value,

-The third argument is the maximum value that should be used for pixels that are greater than the threshold value

-The fourth argument is a flag that specifies the thresholding method to be used

In our implementation, binary and otsu thresholding techniques are used to increase detection quality of region of interests (roi).

To briefly explain, Binary thresholding is a simple image processing method that converts a grayscale image into a binary image. In a binary image, each pixel is assigned a value of either 0 or 255 based on whether the pixel's intensity value is above or below a specified threshold value.

To briefly explain, Otsu's thresholding is a method for automatically calculating the optimal threshold value to use for binary thresholding an image. Otsu's method is based on the idea of maximizing the variance between two classes of pixels in an image (foreground and background) by choosing a threshold value that separates the image's histogram into two peaks. Otsu's method is particularly useful when the image's background and foreground have different intensities. It can be used to improve the accuracy of object detection or image segmentation algorithms by providing a more precise separation of the object from the background.

2.7 Optical Character Recognition (OCR):

The last technique used is optical character recognition. To do so, we utilized pytesseract library of python. pytesseract.image_to_string() is a function from the PyTesseract library, which is a wrapper for the Tesseract OCR engine, an open-source Optical Character Recognition (OCR) library developed by Google. The function pytesseract.image_to_string() is used to extract text from an image. It takes in two main arguments:

-The first argument is the image from which text needs to be extracted. The image can be passed as a file path, an object of the PIL Image class or a numpy array.

-The second argument is the language in which the text is written. It is optional, If not passed it will use the tesseract's built-in language 'eng' (English) as default.

The function returns a string containing the extracted text. The Tesseract OCR engine uses a combination of techniques to recognize characters in an image, including neural networks, statistical models, and image processing. The neural network is trained on a large dataset of images and their corresponding text, called a training dataset. The network learns to recognize patterns in the images that correspond to the text. The statistical model is used to analyze the shapes and patterns of the characters in the image, and to match them to the characters in the training dataset. The image processing techniques are used to pre-process the image before it is passed to the OCR engine. This includes image cropping, image thresholding, image resizing, image enhancement and many more to improve the quality of the image to make it more readable for the OCR engine. The OCR engine uses these techniques to identify the characters and words in the image, and outputs the recognized text.

In terms of mathematical background, the OCR engine uses deep learning algorithms such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) to extract features from images and to classify the characters in the image. CNNs are used to extract features from the image, by applying convolutional filters to the image and down-sampling the image through pooling layers. The filters are learned from the training dataset and are optimized to detect specific features in the image. RNNs are used to process sequences of data, such as the sequences of characters in a word. They are used to analyze the context of the characters in the image, and to recognize patterns in the sequences of characters. The OCR engine uses these deep learning algorithms to recognize characters in the image and to output the recognized text. It is important to note that the accuracy of the OCR engine is highly dependent on the quality of the image passed, so before using this function it is recommended to preprocess the image to improve the text recognition, this could include image cropping, image thresholding, image resizing and image enhancement.

OUR Implementation:

We first read images from my directory. After reading each image, we converted them to gray scale image for ease of computation. Then, we applied Gaussian Blur in order to reduce possible noise in the images. We used 3x3 Gaussian kernel having 0 as standard deviation. Having applied Gaussian Blur, there is a possibility of some features to be blurred. In order to overcome this possible problem, we applied a sharpening filter to the image. After sharpening, we applied Canny edge detection algorithm to extract edges from image because the way we perform detection and recognition is based on edges and contours. We used 25 as lower threshold and 125 as higher threshold. We relatively keep thresholds small in order to increase number of detected edges. Then, we needed to find contours which possibly hold the license plate and text inside them. We performed all these steps to facilitate the detection and recognition process.

After extracting contours from detected edges, we needed to eliminate some of the contours because small contours possibly do not contain any text. During testing and implementing the code, we noticed that the possible license plate texts were found in contours having relatively larger areas. We first sorted contours in descending order with respect to their bounding box area and selected top 10 contours.

After selecting candidate contours, we needed to extract texts from each candidate and select possible license plate text. First, we calculated contour are with “cv2.contourArea()” function and compare it with MIN_AREA which is the minimum area allowed for a contour to have. Then, by utilizing “cv2.boundingRect()” function, we calculated bounding box corresponding to given contour. we extracted this bounding box from gray-scaled image as region of interest. In order to optimize detection, we performed binary thresholding combined with OTSU thresholding.

Having region of interest extracted and processed, we were able to feed region of interest to optical character recognition (ocr) function. To be able to perform ocr, we uitilized “pytesseract.image_to_string()” function. If detected text turns out to be empty, we passed the iteration and continue with next contour. If text contains many lines, we store each line in a list. Then, we remove spaces from every list element. After that, we iterated through each element to first remove unallowed characters in license plate and check for length of remaining text. At the end of this process, we selected the longest text as possible license plate text and corresponding bounding box. In the final, we drew the bounding box on the image with detected text on the left top of bounding box and saved to directory. We performed these tasks for each image in our dataset.

Note: In order to perform OCR, we first installed “Tesseract OCR” to our computers via .exe file inside project folder. Then, we provided path of executable of Tesseract OCR to code. After completing these 2 steps, we were able to perform optical character recognition.

3. Results

Figure 9

Figure 10

Figure 11

Figure 12

Figure 13

Figure 14

Figure 15

Figure 16

Figure 17

Figure 18

Figure 19

Figure 20

Figure 21

Figure 22

Figure 23

Figure 24

4. Discussion

Figures 9 – 24 display the original images of our dataset and the results of each test image. In some cases, the detection works as we desired. For example, Figures 23 and 24 are the example of desired detection. The code was able to perfectly localize license plate and OCR algorithm was able nearly perfectly able to recognize text whereas it only added 1 extra letter to the beginning. Desired text was “34ES1716” and what we detected was “H34ES1716”. The additional “H” in front of the plate was possibly caused by country code “TR” in the beginning. In addition to Figure 23 and 24, Figures 19 and 20 can be given as desired results. The license plate contains “WOR 516K” and we detected “WOR S16K”. First of all, the localization of license plate is as desired, text recognition worked nearly perfectly and the only mistake was recognizing “5” as “S” which is understandable since their shapes are similar and after blurring, it is possibly harder to distinguish them.

We can give more examples to near perfect or “okay” detection. For example, Figures 9 and 10 are good example for this type detection. The desired text was “MH12DE1433” and we detected it as “MH1Z2DE1433”. There is an additional “Z” character which is possibly caused by the fact that shapes of “2” and “Z” are similar and after Gaussian blur, they possibly became indistinguishable. In addition to Figures 9 and 10, Figures 13 and 14 can be given as “okay” example. First of all, localization of license plate was perfect and bounding box fitted original license plate like its shadow. The desired text was “AG2951”and we detected it as “BRAG2951”. This was possibly caused by country code “DK” at the beginning of license plate.

If we want to see some, 100 % detection and recognition, we can look at Figures 11-12 and 15-16. In these examples, our algorithm worked perfectly both for license plate localization and optical character recognition. In Figures 11-12, the desired text was “34FC6302” and we detected “34FC6302”. In Figures 15-16, desired text wad “06CD1171” and we detected “06CD1171” as desired.

However, there were also some bad tests. For example, Figures 21 and 23 are the first pair of bad detection example. First of all, license plate localization was too slopy, bounding box was too large and because of this problem, optical character recognition did not work as desired and gave us wrong text.

When we consider the overall performance of our algorithm, we cannot deny the fact that our algorithm requires many more updates. First of all, it heavily relies on mid-quality or high-quality images of license plates. For example, if we use low quality or farther images of license plate, our algorithm possibly detects unrelated regions as license plates. Second of all, the OCR algorithm we have utilized, needs to be updated or at least optimized. However, we believe that, with slight updates, our algorithm can be used to meet at least personal requirements.

5. Appendix

```
import cv2
import pytesseract
import numpy as np

#Path of tesseract ocr exe file
pytesseract.pytesseract.tesseract_cmd = 'C:\\Program Files\\Tesseract-
OCR\\tesseract.exe'

#Min area parameter for comparing contour area
MIN_AREA = 100

#Function to calculate area of given contour
def getArea(cnt):
    [x, y, w, h] = cv2.boundingRect(cnt)
    return w*h


for i in range(1,9):

    # Load the image
    image = cv2.imread(f"deneme{i}.jpg")

    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to the image
    gray = cv2.GaussianBlur(gray, (3,3), 0)

    # Define the sharpening kernel
    kernel = np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])

    # Apply the kernel to the image using a convolution
    gray = cv2.filter2D(gray, -1, kernel)

    # Apply Canny edge detection to the image
    edges = cv2.Canny(gray, 25, 125)

    # Find contours in the image
    contours, _ = cv2.findContours(edges, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)

    plateText = ""
    allowedChars = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    bbox_holder = []
    area = 0
```

```
    # Iterate through the contours and filter for top 10 contours if there are
more than 10
    # Too many contours ma lead to false detections
    if len(contours) > 10:
        contours = sorted(contours, key=getArea, reverse=True)
        contours = contours[:11]

    # Iterate through the contours and filter for rectangular ones
    for cnt in contours:

        #Pass the iteration if contour has smaller area than desired
        if cv2.contourArea(cnt) < MIN_AREA:
            continue
        [x, y, w, h] = cv2.boundingRect(cnt)
        if w/h > 5 or h/w > 5:
            continue
        #cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
        roi = gray[y:y + h, x:x + w]

        # Apply Otsu thresholding
        _, roi = cv2.threshold(roi, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)

        # Run Tesseract OCR on the region of interest
        text = pytesseract.image_to_string(roi,
                config='--psm 11')

        #Pass the iteration if there is not detected text in contour
        if text == "":
            continue

        #Remove newline characters if there is any
        if "\n" in text:
            text = text.split("\n")

        #Remove white spaces
        for txt in text:

            txt = txt.replace(" ", "")
            txt = txt.replace( "\t", "")

            for ch in txt:
                if ch not in allowedChars:
                    txt = txt.replace(ch, "")

            if len(txt) > len(plateText):
                plateText = txt
                bbox_holder = [(x, y), (x + w, y + h), (0, 255, 0), 2]
                area =  w * h
```

```
        print(text)
        print("----------------------------------------")

    if bbox_holder != []:
        cv2.rectangle(image, bbox_holder[0], bbox_holder[1], bbox_holder[2],
bbox_holder[3])
        cv2.putText(image,  plateText, bbox_holder[0],
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (36,255,12), 2)

    # Show the original image with rectangles around the license plates
    print("Possible Plate Text: ", plateText)
    cv2.imwrite(f"./results/result{i}.jpg", image)
    #cv2.imshow("License Plate", image)
    #cv2.waitKey(0)
```

6. References

https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html

https://en.wikipedia.org/wiki/Canny_edge_detector

https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

https://docs.opencv.org/4.x/da/d0c/tutorial_bounding_rects_circles.html

https://en.wikipedia.org/wiki/Gaussian_blur

https://en.wikipedia.org/wiki/Unsharp_masking