TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

FINAL PROJECT: DIGITAL IMAGE PROCESSING COURSE

TRAFFIC SIGNS RECOGNIZE

Người hướng dẫn: **TS Trịnh Hùng Cường**

Người thực hiện: **Ngô Minh Khôi – 521H0254**

Khoá **: 25**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

FINAL PROJECT: DIGITAL IMAGE PROCESSING COURSE

TRAFFIC SIGNS RECOGNIZE

Người hướng dẫn: **TS Trịnh Hùng Cường**

Người thực hiện: **Ngô Minh Khôi – 521H0254**

Khoá **: 25**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024

Acknowledgment

I always feel deeply grateful and respectful towards Professor Trinh Hung Cuong. He is not only a dedicated teacher, but also a mentor who constantly motivates and inspires me to learn. Throughout my studies, he has continuously shared his extensive expertise, helping me grasp the key concepts and important applications. Thanks to his dedicated guidance, I have been able to overcome many challenges and successfully complete my assignments and projects.

In addition, he always takes the time to listen and patiently address my queries. He is not only concerned with imparting knowledge, but also constantly encourages and supports me to fully realize my potential. With his mentorship and support, I have made remarkable progress in my studies and developed the necessary skills.

I sincerely thank Professor Trinh Hung Cuong for his tremendous contributions. He is a shining example for me to follow, and I will always strive to learn and cultivate myself to become a person like him - a dedicated educator and a respected mentor.

THE PROJECT IS COMPLETED AT TON DUC THANG UNIVERSITY

I solemnly declare that this is the project product of our group and was guided by Dr. Trinh Hung Cuong. The research contents and results in this topic are truthful and have not been published in any form before. The data in the tables and figures used for analysis, comments, and evaluation were collected by the authors from various sources, which are clearly cited in the references.

In addition, the project also uses some comments, evaluations, and data from other authors and organizations, all of which are properly cited and referenced.

If any fraud is discovered, I will take full responsibility for the content of my project. The University of Technology and Education Ton Duc Thang is not involved in any copyright or intellectual property violations that I may have committed during the implementation process.

TP. Hồ Chí Minh, ngày tháng năm

Tác giả

(ký tên và ghi rõ họ tên)

Ngô Minh Khôi

INSTRUCTOR VERIFICATION AND EVALUATION SECTION

Confirmation from the instructor_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

The teacher's evaluation part marks the test_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

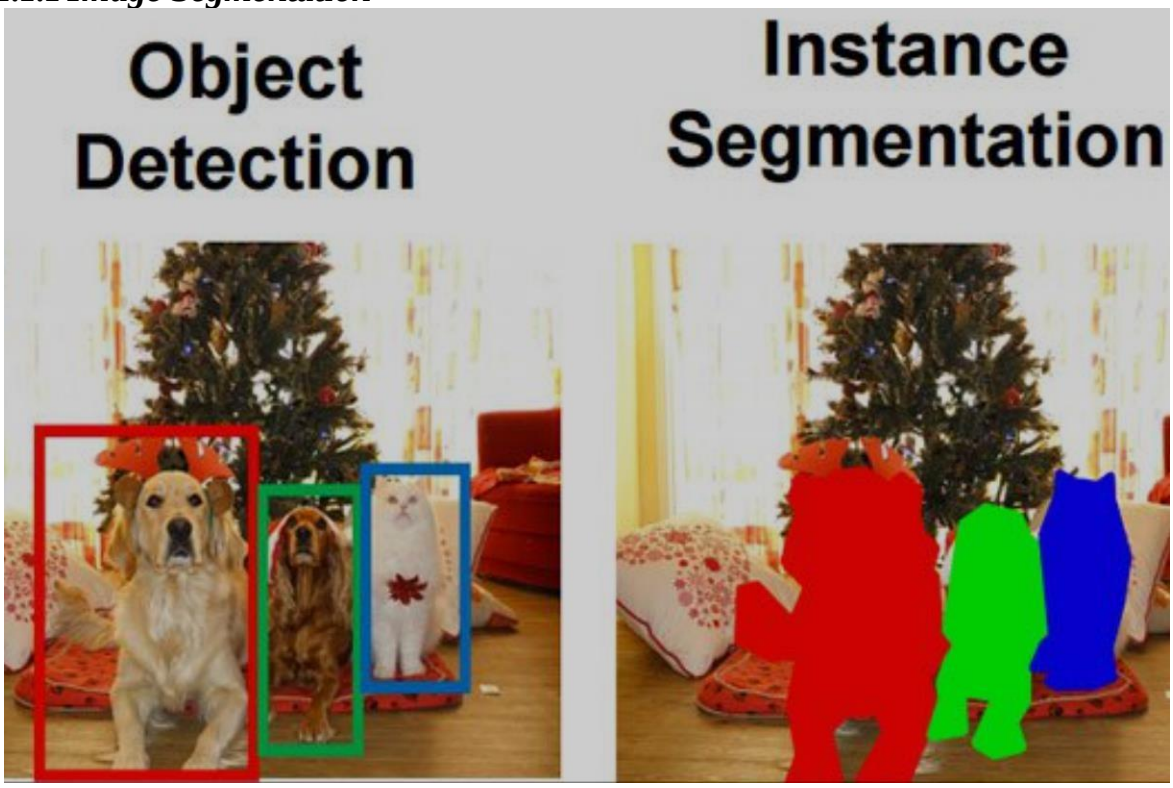
Table of contents

CHAPTER 01: SOLVING METHODS

1.1. Overview of Relevant Techniques and Algorithms

This project utilizes image processing and computer vision techniques for traffic sign recognition. The key techniques applied include:

1.1.1 Image Segmentation



- Separating the traffic sign from the background using color-based segmentation techniques.

- Image segmentation is a crucial step in the traffic sign recognition process outlined in the methodology. It involves separating the traffic sign from the background of the input image. The key aspects of the image segmentation approach include:

Converting the images to the HSV (Hue, Saturation, Value) color space, which makes it easier to apply color-based segmentation techniques.

Defining color thresholds to identify regions corresponding to the colors of traffic signs, such as red and blue.

Using these color thresholds to create masks that highlight the potential traffic sign areas within the image.

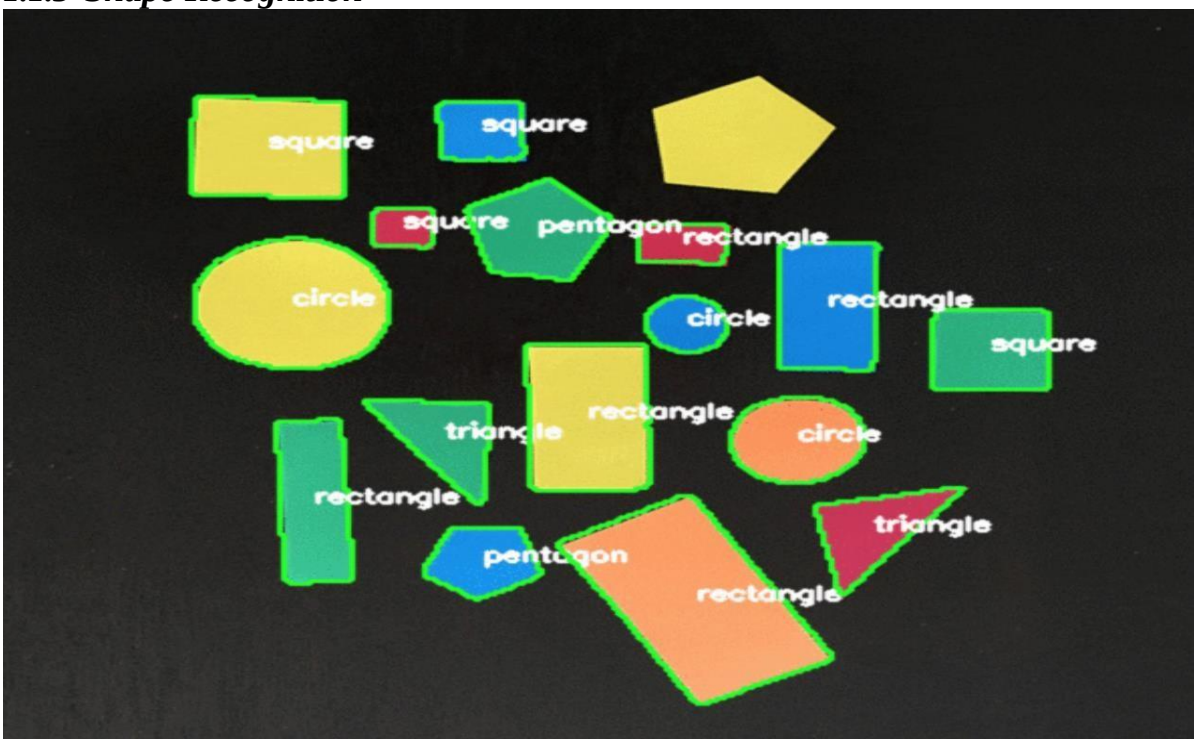
Applying morphological operations like dilation and erosion to filter out noise and refine the segmented regions.

The goal of this image segmentation process is to isolate the traffic signs from the surrounding environment, preparing them for the subsequent steps of shape determination, cropping, and final recognition.

1.1.2 Edge Detection

- Identifying the edges of the traffic sign using edge detection algorithms like Canny Edge Detection.
- Edge detection is an important step in the traffic sign recognition methodology outlined earlier. It involves identifying the edges of the traffic signs within the input images using edge detection algorithms such as the Canny edge detector.
-The edge detection process helps determine the boundaries and contours of objects in the image, including the traffic signs. This provides information about the shape and structure of the traffic signs, which is a critical foundation for the subsequent steps of shape identification and traffic sign classification.
By utilizing edge detection techniques like Canny, the approach is able to isolate and emphasize the significant edges of the traffic signs. This prepares the segmented regions for the shape analysis and final recognition stages of the traffic sign recognition pipeline.
The goal of this edge detection step is to extract the important structural information about the traffic signs from the input images, enabling their accurate identification and classification in the later processing phases.

1.1.3 Shape Recognition



- Determining the shape of the traffic sign using shape recognition techniques like Hough Transform.
- Shape recognition is another key component in the traffic sign recognition methodology. It involves determining the specific geometric shapes of the detected traffic signs, such as circles, triangles, rectangles,

- Some of the common shape recognition techniques used in this context include:

Hough Transform - This algorithm is able to identify the presence of particular shapes, like circles and lines, within the image data. It's effective at detecting the basic geometric forms often used in traffic sign design.

- Template Matching - By comparing the segmented traffic sign regions against a library of template shapes, this technique can classify the signs based on their overall shape profile.
- Contour Analysis - Examining the contours and boundaries of the segmented regions provides information about the underlying shape characteristics of the traffic signs.

-The goal of the shape recognition step is to extract the distinctive shape features of the detected traffic signs. This shape information is then used as an important attribute for the final traffic sign classification process, where the signs are identified based on a combination of color, shape, and other distinguishing characteristics.
-Accurately recognizing the geometric shapes of traffic signs is a critical component in enabling robust and reliable automated traffic sign detection and recognition systems.

1.1.4 Traffic Sign Classification

- Classifying traffic signs based on color, shape, and features using the OpenCV library and experimentation.
- The traffic sign classification stage is the final step in the traffic sign recognition pipeline. After segmenting the image to isolate the traffic signs and analyzing their shapes, the next task is to actually identify and classify the specific type of traffic sign detected.

For this classification step, OpenCV provides a range of computer vision techniques that can be leveraged:

Feature Extraction:

- OpenCV offers various feature descriptor algorithms like SIFT, SURF, ORB,that can extract distinctive visual features from the segmented traffic sign images.

These features capture information about color, texture, edges, and other characteristics that are unique to different traffic sign

Template Matching:

- OpenCV's template matching functions can also be used to compare the segmented traffic sign against a database of known traffic sign templates.
- This allows classification by finding the best matching template for the detected traffic sign shape and appearance.

The specific implementation of the traffic sign classification component will depend on factors like the size of the traffic sign dataset, the complexity of the sign types, and the desired level of ac bcuracy. But OpenCV provides a robust set of computer vision tools to tackle this challenge.

1.2. Detailed Explanation and Description of the Proposed Methodology or Approach

1.2.1. Create the Dictionaries for Traffic Sign

Two dictionaries,**'Red'** and **'Red_lower'**, are predefined to map binary patterns to specific traffic signs. These dictionaries facilitate the recognition of traffic signs based on the analysis of segmented image regions.

1.2.2. Create Four-Point Perspective Transformation

We wil design to perform a perspective transformation of an image based on four specified points. This transformation corrects the perspective of detected sign regions to a top-down view, making it easier to analyze the contents of the sign.

Steps in Four-Point Perspective Transformation:
Compute the Summation and Difference of Points:
-Calculate the sum and difference of the coordinates of the points to identify the top-left, top-right, bottom-right, and bottom-left corners.
Calculate the Maximum Width and Height:
-Determine the width and height of the new perspective by calculating the distances between the identified points.
Define the Destination Points:
-Set the destination points for the transformed perspective based on the maximum width and height.
Compute the Perspective Transform Matrix:
-Use OpenCV's getPerspectiveTransform function to obtain the transformation matrix.
Apply the Perspective Warp:
-Apply the perspective warp to the image using OpenCV's warpPerspective function.

1.2.3. Traffic Sign Recognition

I wil create The function will identified the traffic sign processes the image to identify traffic signs by dividing the image into specific blocks and analyzing the fraction of white pixels in each block. This fraction is used to generate a binary pattern, which is then matched against the predefined dictionaries.

Steps in Traffic Sign Recognition:
-Invert Image Colors:
- Invert the colors of the image to highlight the regions of interest.

-Divide the Image into Blocks:
- Split the image into specific subregions (left, center, right, and top blocks).

-Calculate White Pixel Fractions:
- Compute the fraction of white pixels in each block.

-Generate Binary Pattern:
- Create a binary pattern based on whether the fraction of white pixels in each block exceeds a threshold.

-Match Pattern to Dictionary:
- Match the generated binary pattern against the entries in the predefined dictionaries to identify the traffic sign.

1.2.4. Traffic Sign Detection

I wil create The function cam find traffic sign orchestrates the detection of traffic signs by reading the image, applying color segmentation, performing morphological transformations, detecting contours, and identifying the largest contour that represents a potential traffic sign.

Steps in Traffic Sign Detection:

-Read and Resize Image:

- Load the image from the file and resize it to a standard dimension (250x250 pixels).

-Convert Image to HSV Color Space:

- Convert the image from BGR to HSV color space for easier color-based segmentation.

-Color Segmentation:

- Create masks to segment the image based on red or blue color ranges.

-Morphological Transformations:

- Apply morphological operations (opening and closing) to clean up the segmented mask.

-Contour Detection:

- Find contours in the cleaned mask.

-Identify Largest Contour:

- Determine the largest contour by area and check if it exceeds a certain threshold relative to the image area.

-Draw Contour and Transform Perspective:

- Draw the largest contour on the image and apply a perspective transformation to the region of interest.

-Recognize Traffic Sign:

- Use the transformed region to recognize the traffic sign by calling identifyTrafficSign.

-Add Annotation and Save Result:

- Annotate the original image with the detected traffic sign and save the annotated image to the output folder.

1.2.5 Display and Read Images Functions

I wil create the functions display image and read file image facilitate displaying images and processing multiple images in a folder, respectively.

Steps in Display and Read Images:

Display Images:

- Read all images from a specified folder and display them in a grid layout using Matplotlib.

Process Images in Folder:

- Read images from a folder, detect traffic signs using findTrafficSign, and save the results.

The final Summary

➔the methodology integrates several steps to detect and recognize traffic signs effectively:

-Defining Lookup Dictionaries: Predefined patterns for quick recognition.
-Perspective Transformation: Correcting the view of detected regions.
-Sign Recognition: Analyzing segmented regions to identify signs.
-Sign Detection: Detecting and processing traffic signs in images.
-Image Display and Processing: Managing and displaying image results.
-Main Execution: Orchestrating the overall process.

CHAPTER 02: EXPERIMENTAL STEP AND RESULT

2.1 Import the libraries we need

```
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt
```

2.2 Define two dictionaries

```
SIGNS_LOOKUP_RED = {
    (1, 1, 0, 1): 'Signs for tractors',
    (1, 0, 0, 1): 'No-horn sign',
    (0, 1, 1, 1): 'No go straight sign',
    (0, 1, 0, 1): 'No-taxi sign',
    (0, 1, 0, 0): 'No-parking sign',
    (0, 0, 1, 0): 'No-right-turn sign',
    (1, 1, 1, 1): 'Speed-Limit 20 km/h sign',
    (0, 0, 0, 1): 'No U-Turn sign',
    (0, 1, 1, 0): 'Height-Limit-3.7m sign',
    (0, 0, 1, 1): 'No Left Turn Sign',
}

SIGNS_LOOKUP_RED_LOWER = {
    (1, 0, 1, 0): 'Stop-sign',
    (1, 1, 1, 0): 'No-Entry Sign',
    (1, 1, 1, 1): 'No-cycling sign',
    (0, 0, 0, 0): 'No-Entry sign'
}
```

Purpose: To map binary patterns into specific, easily identifiable traffic signs.

Description: Two dictionaries, ‘SIGNS_LOOKUP_RED’ and ‘SIGNS_LOOKUP_RED_LOWER’, are predefined with keys representing binary patterns of segmented image regions and values being the names of traffic signs.

2.3 Transforming four specified points.

```
### function used to transform perspective by transforming four specified points.
def four_point_transform(image, pts):
    # Initialize an array to hold the ordered points (top-left, top-right, bottom-right, bottom-left)
    rect = np.zeros((4, 2), dtype="float32")

    # Sum the x and y coordinates and find the top-left (smallest sum) and bottom-right (largest sum) points
    s = pts.sum(axis=1)
    rect[0] = pts[np.argmin(s)]  # Top-left point
    rect[2] = pts[np.argmax(s)]  # Bottom-right point

    # Compute the difference between the x and y coordinates and find the top-right (smallest difference)
    # and bottom-left (largest difference) points
    diff = np.diff(pts, axis=1)
    rect[1] = pts[np.argmin(diff)]  # Top-right point
    rect[3] = pts[np.argmax(diff)]  # Bottom-left point

    # Optional: Apply a scaling ratio (not needed here, so ratio is 1)
    ratio = 1
    rect *= ratio

    # Unpack the ordered points
    (tl, tr, br, bl) = rect

    # Compute the width of the new image, which will be the maximum distance between the bottom-right and
    # bottom-left x-coordinates or the top-right and top-left x-coordinates
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))

    # Compute the height of the new image, which will be the maximum distance between the top-right and
    # bottom-right y-coordinates or the top-left and bottom-left y-coordinates
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
dst = np.array([
    [0, 0],
    [maxWidth - 1, 0],
    [maxWidth - 1, maxHeight - 1],
    [0, maxHeight - 1]],
    dtype="float32")
M = cv2.getPerspectiveTransform(rect, dst)
warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
return warped
```

Purpose: To correct the perspective of the detected traffic sign region, transforming it to a top-down view.

Input: An image and four points defining the region to be transformed.

Process:
First create the function name ‘four_point_tranform’

-Step 1 Initialization of Ordered Points:

- **‘react’** is initialized to store the coordinates of the four points in the order of top-left, top-right, bottom-right, and bottom-left.

-Step 2 Sum and Difference of Points:

- **‘s’** computes the sum of the x and y coordinates. The point with the smallest sum is the top-left (because it’s the closest to the origin), and the point with the largest sum is the bottom-right.
- **‘diff ’** computes the difference between the x and y coordinates. The point with the smallest difference is the top-right, and the point with the largest difference is the bottom-left.

-Step 3 Scaling the Coordinates:

- The points are scaled by a ratio. Here, the ratio is set to 1, so there is no actual scaling applied.

-Step 4 Unpacking the Ordered Points:

- The ordered points are unpacked into individual variables’ **tl** ‘(top-left), **tr** (top-right), ‘**br’** (bottom-right), and ‘**bl’** (bottom-left).

-Step 5 Computing the Dimensions:

- The width of the new image is computed as the maximum distance between the bottom points (**widthA**) and the top points (**widthB).**
- The height of the new image is computed as the maximum distance between the right points (**heightA**) and the left points (**heightB**).

-Step 6 Defining Destination Points:

- **‘dst’** defines the coordinates of the destination points for the warped image. These points form a rectangle of dimensions ‘**maxWidth’** by’ **maxHeight’.**

-Step 7 Perspective Transform Matrix:

- **‘M’** is the perspective transform matrix computed using OpenCV's ‘**getPerspectiveTransform’** function, which takes the source and destination points.

-Step 8 Applying Perspective Transformation:

- The perspective transformation is applied using OpenCV's **‘warpPerspective’** function, which warps the original image to the new perspective based on matrix M.

-Step 9 Returning the Warped Image:

- Finally, the function returns the warped image, which has the corrected perspective.

2.4 Traffic sign recognization function

```python
###  Traffic sign recognition function
def identifyTrafficSign(image, mask_red, blue):
    # Invert the colors of the image (this operation reverses black to white and white to black, etc.)
    image = cv2.bitwise_not(image)

    # Divide the image dimensions by 10 to get the size of the blocks
    (subHeight, subWidth) = np.divide(image.shape, 10)
    subHeight = int(subHeight)  # Convert to integer
    subWidth = int(subWidth)  # Convert to integer

    # Define blocks from the image
    leftBlock = image[4 * subHeight:9 * subHeight, subWidth:3 * subWidth]  # Block on the left side
    centerBlock = image[4 * subHeight:9 * subHeight, 4 * subWidth:6 * subWidth]  # Center block
    rightBlock = image[4 * subHeight:9 * subHeight, 7 * subWidth:9 * subWidth]  # Block on the right side
    topBlock = image[2 * subHeight:4 * subHeight, 3 * subWidth:7 * subWidth]  # Block on the top

    # Calculate the fraction of the white pixels in each block
    leftFraction = np.sum(leftBlock) / (leftBlock.shape[0] * leftBlock.shape[1])
    centerFraction = np.sum(centerBlock) / (centerBlock.shape[0] * centerBlock.shape[1])
    rightFraction = np.sum(rightBlock) / (rightBlock.shape[0] * rightBlock.shape[1])
    topFraction = np.sum(topBlock) / (topBlock.shape[0] * topBlock.shape[1])

    # Combine the fractions into a tuple
    segments = (leftFraction, centerFraction, rightFraction, topFraction)
    print(segments)  # Print out the segment fractions
```

```python
    # Determine the threshold and lookup dictionary based on the color (blue or red)
    if blue:
        THRESHOLD = 100  # Threshold for detecting blue traffic signs
        SIGNS_LOOKUP = SIGNS_LOOKUP_BLUE  # Lookup dictionary for blue signs
    else:
        THRESHOLD = 196  # Threshold for detecting red traffic signs
        SIGNS_LOOKUP = SIGNS_LOOKUP_RED  # Lookup dictionary for red signs

        # Adjust the threshold and lookup dictionary if all segments are below the initial threshold
        if leftFraction < THRESHOLD and centerFraction < THRESHOLD and rightFraction < THRESHOLD and topFraction < THRESHOLD:
            print('Lower')
            THRESHOLD = 80  # Lower threshold for red traffic signs
            SIGNS_LOOKUP = SIGNS_LOOKUP_RED_LOWER  # Lookup dictionary for red signs with lower threshold

    # Convert the fractions into binary values (1 if above threshold, 0 if below)
    segments = tuple(1 if segment > THRESHOLD else 0 for segment in segments)
    print(segments)  # Print out the binarized segments

    # Check if the binarized segments match any entry in the lookup dictionary
    if segments in SIGNS_LOOKUP:
        return SIGNS_LOOKUP[segments]  # Return the identified traffic sign
    else:
        return None  # Return None if no match is found
```

First we will create the function definition name ‘indentifyTrafficsign’

Parameters:
image: The input image in which traffic signs need to be identified.

mask_red: A mask that highlights red regions in the image.
blue: A boolean flag indicating whether the traffic signs to be identified are blue (True) or red (False).
I will explained the code step by step:

Step 1: Color Inversion

```
# Invert the colors of the image (this operation reverses black to white and white to black, etc.)
image = cv2.bitwise_not(image)
```

-***Purpose:*** Inverts the colors of the image. This means black pixels become white and white pixels become black. This step is useful for enhancing the visibility of features in the image, especially when detecting white traffic signs against a dark background.

Step 2: Image Inversion
-***Purpose:*** Divides the image dimensions by 10 to determine the size of each block for segmentation.
-***Explaination***

- **'image.shape'** gives the dimensions of the image (height, width).
- **'np.divide(image.shape, 10)'** divides each dimension by 10, creating a grid of blocks.
- The resulting **'subHeight'** and **'subWidth'** are the heights and widths of each block in the grid.

Step 3: Block Extraction

```
# Define blocks from the image
leftBlock = image[4 * subHeight:9 * subHeight, subWidth:3 * subWidth]  # Block on the left side
centerBlock = image[4 * subHeight:9 * subHeight, 4 * subWidth:6 * subWidth]  # Center block
rightBlock = image[4 * subHeight:9 * subHeight, 7 * subWidth:9 * subWidth]  # Block on the right side
topBlock = image[2 * subHeight:4 * subHeight, 3 * subWidth:7 * subWidth]  # Block on the top
```

-***Purpose:*** Calculates the fraction of white pixels in each block.
-***Explaination***

- leftBlock extracts a block from the left side of the image.
- centerBlock extracts a block from the center of the image.
- rightBlock extracts a block from the right side of the image.
- topBlock extracts a block from the top center of the image.
- These blocks are chosen to isolate the regions where traffic signs are typically found.

Step 4: Fraction Calculation

```
# Calculate the fraction of the white pixels in each block
leftFraction = np.sum(leftBlock) / (leftBlock.shape[0] * leftBlock.shape[1])
centerFraction = np.sum(centerBlock) / (centerBlock.shape[0] * centerBlock.shape[1])
rightFraction = np.sum(rightBlock) / (rightBlock.shape[0] * rightBlock.shape[1])
topFraction = np.sum(topBlock) / (topBlock.shape[0] * topBlock.shape[1])
```

***-Purpose*:** Calculates the fraction of white pixels in each block.

-Explanation:

- **'np.sum(leftBlock)'** sums up the pixel values in the leftBlock.
- **'leftBlock.shape[0] * leftBlock.shape[1]'** calculates the total number of pixels in the leftBlock.
- The fraction is the ratio of the sum of white pixels to the total number of pixels in the block.
- This process is repeated for the centerBlock, rightBlock, and topBlock.

Step 5: Storing and Printing Fractions

```
# Combine the fractions into a tuple
segments = (leftFraction, centerFraction, rightFraction, topFraction)
print(segments)  # Print out the segment fractions
```

***-Purpose*:** Stores the fractions in a tuple and prints them for debugging.

-Explanation:

- **'segments'** is a tuple containing the fractions of white pixels for the left, center, right, and top blocks.
- **'print(segments)'** outputs the fractions to the console for debugging purposes.

Step 6: Threshold and Lookup Table Selection

```
# Determine the threshold and lookup dictionary based on the color (blue or red)
if blue:
    THRESHOLD = 100  # Threshold for detecting blue traffic signs
    SIGNS_LOOKUP = SIGNS_LOOKUP_BLUE  # Lookup dictionary for blue signs
else:
    THRESHOLD = 196  # Threshold for detecting red traffic signs
    SIGNS_LOOKUP = SIGNS_LOOKUP_RED  # Lookup dictionary for red signs

    # Adjust the threshold and lookup dictionary if all segments are below the initial threshold
    if leftFraction < THRESHOLD and centerFraction < THRESHOLD and rightFraction < THRESHOLD and topFraction < THRESHOLD:
        print('Lower')
        THRESHOLD = 80  # Lower threshold for red traffic signs
        SIGNS_LOOKUP = SIGNS_LOOKUP_RED_LOWER  # Lookup dictionary for red signs with lower threshold
```

***-Purpose*:** Determines the threshold and lookup table based on the color of the traffic sign (blue or red).

-Explanation:

- If blue is True, a lower threshold of 100 is used and '**SIGNS_LOOKUP_BLUE'** lookup table is selected.
- If blue is False, a higher threshold of 196 is used initially and '**SIGNS_LOOKUP_RED'** lookup table is selected.
- If the fractions for all blocks are below the initial threshold, it prints '**Lower**', reduces the threshold to 80, and uses a different lookup table '**SIGNS_LOOKUP_RED_LOWER.'**

Step 7: Binarization of Segments

```
# Convert the fractions into binary values (1 if above threshold, 0 if below)
segments = tuple(1 if segment > THRESHOLD else 0 for segment in segments)
print(segments)  # Print out the binarized segments
```

-***Purpose:*** Converts the fractions into binary values based on the threshold.

-***Explanation:***

- Each fraction in segments is compared to the **THRESHOLD**.
- If a fraction is greater than the threshold, it is converted to 1 (white), otherwise to 0 (black).
- This binarized segments tuple is printed for debugging.

2.5 Traffic sign detection function

```
### Traffic sign detection function
def findTrafficSign(image_path, output_folder, blue):
    print(image_path)  # Print out the image paths
    bb = 'box'
    lower_red1 = np.array([0, 70, 50])
    upper_red1 = np.array([10, 255, 255])
    lower_red2 = np.array([170, 70, 60])
    upper_red2 = np.array([180, 255, 255])
    lower_blue = np.array([90, 110, 80])
    upper_blue = np.array([255, 255, 255])

    # Read image
    frame = cv2.imread(image_path)
    frame = cv2.resize(frame, (250, 250))  # Resize the image
    frameArea = frame.shape[0] * frame.shape[1]

    # Convert to HSV
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    kernel = np.ones((3, 3), np.uint8)

    # Segmentation based on color
    if blue:
        mask = cv2.inRange(hsv, lower_blue, upper_blue)
    else:
        mask_red1 = cv2.inRange(hsv, lower_red1, upper_red1)
        mask_red2 = cv2.inRange(hsv, lower_red2, upper_red2)
        mask = cv2.bitwise_or(mask_red1, mask_red2)

    # Transform morphology
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

```
# Edge detection
cnts, _ = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
largestArea = 0
largestRect = None
if len(cnts) > 0:
    for cnt in cnts:
        rect = cv2.minAreaRect(cnt)
        box = cv2.boxPoints(rect)
        box = np.int0(box)
        sideOne = np.linalg.norm(box[0] - box[1])
        sideTwo = np.linalg.norm(box[0] - box[3])
        area = sideOne * sideTwo
        if area > largestArea:
            largestArea = area
            largestRect = box

# Condition: If the area is greater than a threshold
if largestArea > frameArea * 0.02:
    cv2.drawContours(frame, [largestRect], 0, (0, 0, 255), 3)
    output_folder_bb = output_folder + bb
    if not os.path.exists(output_folder_bb):
        os.makedirs(output_folder_bb)
    output_folder_bb = os.path.join(output_folder_bb, os.path.basename(image_path))
    cv2.imwrite(output_folder_bb, frame)

    # Perspective transformation
    warped = four_point_transform(mask, [largestRect][0])

    # Traffic sign recognition
    detectedTrafficSign = identifyTrafficSign(warped, mask, blue)

    # Adding text to an image
    text_position = (30, 30)
    cv2.putText(frame, detectedTrafficSign, text_position, cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
```

```
# Save output image
if not os.path.exists(output_folder):
    os.makedirs(output_folder)
output_path = os.path.join(output_folder, os.path.basename(image_path))
cv2.imwrite(output_path, frame)
```

Next I will create the function name **‘findTrafficsign’**

```
### Traffic sign detection function
def findTrafficSign(image_path, output_folder, blue):
```

The first step (step 1) : Initial Setup

```
def findTrafficSign(image_path, output_folder, blue):
    print(image_path)  # Print out the image paths
    bb = 'box'
    lower_red1 = np.array([0, 70, 50])
    upper_red1 = np.array([10, 255, 255])
    lower_red2 = np.array([170, 70, 60])
    upper_red2 = np.array([180, 255, 255])
    lower_blue = np.array([90, 110, 80])
    upper_blue = np.array([255, 255, 255])
```

- ***Purpose:*** Print the image path and define color ranges for red and blue in the HSV color space.
- ***Explanation:***

'print(image_path)' outputs the image path to the console for debugging.
bb is a string used later for naming the bounding box directory.
lower_red1, upper_red1, lower_red2, upper_red2 define two ranges for red color to account for the hue wrapping in HSV color space.
lower_blue, upper_blue define the range for blue color.

Step 2: Image Reading and Resizing

```
# Read image
frame = cv2.imread(image_path)
frame = cv2.resize(frame, (250, 250))  # Resize the image
frameArea = frame.shape[0] * frame.shape[1]
```

- ***Purpose:*** Create a binary mask to segment the image based on the specified color.
- ***Explanation:***

- If blue is True, create a mask for blue color using **lower_blue** and **upper_blue**.
- If blue is False, create two masks for red color (mask_red1 and mask_red2) and combine them using **cv2.bitwise_or.**

Step 3: Color Conversion

```
# Convert to HSV
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
kernel = np.ones((3, 3), np.uint8)
```

Purpose*:* Convert the image from BGR to HSV color space and define a kernel for morphological operations.
Explanation:

- **'cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)'** converts the image to HSV color space, which is more suitable for color-based segmentation.
- kernel is a 3x3 matrix of ones used for morphological transformations.

Step 4: Color Segmentation

```
# Segmentation based on color
if blue:
    mask = cv2.inRange(hsv, lower_blue, upper
else:
    mask_red1 = cv2.inRange(hsv, lower_red1, upper_red1)
    mask_red2 = cv2.inRange(hsv, lower_red2, upper_red2)
    mask = cv2.bitwise_or(mask_red1, mask_red2)
```

(variable) upper_red1:

Purpose: Create a binary mask to segment the image based on the specified color.
Explanation:

- If blue is True, create a mask for blue color using '**lower_blue'** and '**upper_blue**.'
- If blue is False, create two masks for red color **(mask_red1 and mask_red2**) and combine them using **'cv2.bitwise_or'**.

Step 5: Morphological Transformations

```
# Transform morphology
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

-Purpose: Clean up the mask using morphological operations.
-Explanation:

- **cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)** performs an opening operation to remove small noise.
- **cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)** performs a closing operation to close small holes in the mask.

Step 6: Contour Detection

```
# Edge detection
cnts, _ = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
largestArea = 0
largestRect = None
if len(cnts) > 0:
    for cnt in cnts:
        rect = cv2.minAreaRect(cnt)
        box = cv2.boxPoints(rect)
        box = np.int0(box)
        sideOne = np.linalg.norm(box[0] - box[1])
        sideTwo = np.linalg.norm(box[0] - box[3])
        area = sideOne * sideTwo
        if area > largestArea:
            largestArea = area
            largestRect = box
```

-Purpose: Detect contours and find the largest bounding rectangle.
-Explanation:

- **cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)** finds contours in the mask.
- Initialize largestArea to zero and largestRect to None.
- Iterate through the contours, find the minimum area rectangle for each contour, and calculate its area.
- Keep track of the largest rectangle found.

Step 7: Condition Check and Drawing Contours

```
# Condition: If the area is greater than a threshold
if largestArea > frameArea * 0.02:
    cv2.drawContours(frame, [largestRect], 0, (0, 0, 255), 3)
    output_folder_bb = output_folder + bb
    if not os.path.exists(output_folder_bb):
        os.makedirs(output_folder_bb)
    output_folder_bb = os.path.join(output_folder_bb, os.path.basename(image_path))
    cv2.imwrite(output_folder_bb, frame)
```

-Purpose: Check if the largest rectangle's area exceeds a threshold and save the image with drawn contours.
-Explanation:

- If **'largestArea'** is greater than 2% of '**frameArea'**, draw the largest rectangle on the image.
- Create the output directory if it does not exist.
- Save the image with the drawn rectangle in the output folder.

Step 8: Perspective Transformation and Traffic Sign Recognition

```
# Perspective transformation
warped = four_point_transform(mask, [largestRect][0])

# Traffic sign recognition
detectedTrafficSign = identifyTrafficSign(warped, mask, blue)
```

-Purpose: Apply perspective transformation and recognize the traffic sign.
-Explanation:

- **four_point_transform(mask, [largestRect][0])** applies a perspective transformation to get a top-down view of the largest rectangle.
- **identifyTrafficSign(warped, mask, blue)** identifies the traffic sign from the transformed image.

Step 9: Adding Text and Saving Output

```
        # Adding text to an image
        text_position = (30, 30)
        cv2.putText(frame, detectedTrafficSign, text_position, cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    # Save output image
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    output_path = os.path.join(output_folder, os.path.basename(image_path))
    cv2.imwrite(output_path, frame)
```

-Purpose: Add the detected traffic sign text to the image and save the final output.
-Explanation:

- **cv2.putText(frame, detectedTrafficSign, text_position, cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)** adds the detected traffic sign text to the image.
- Create the output directory if it does not exist.
- Save the final image in the output folder.

2.6 Display image in the folder

```
### Function to display images in a folder
def display_images_in_folder(folder_path, num_cols=3):
    image_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp'))]
    num_images = len(image_files)
    num_rows = (num_images + num_cols - 1) // num_cols
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 5 * num_rows))
    fig.set_facecolor('black')  # Set the background of the figure to black
    axes = axes.ravel()
    for idx, image_file in enumerate(image_files):
        image_path = os.path.join(folder_path, image_file)
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        axes[idx].imshow(image)
        axes[idx].axis('off')
        axes[idx].set_facecolor('black')  # Set the background of the axis to black

    for i in range(idx + 1, len(axes)):
        axes[i].axis('off')
        axes[i].set_facecolor('black')  # Set the background of the axis to black
    plt.tight_layout()
    plt.show()
```

I will create the function name **‘ display_image_in_folder’**

```
def display_images_in_folder(folder_path, num_cols=3):
```

-folder_path: The path to the folder containing images.
-num_cols: The number of columns to use when displaying images. Default is 3.

I will explain the code detail

Step 1: List Image Files

```
image_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp'))]
```

-Purpose: List all image files in the specified folder.

- Explanation:

- **os.listdir(folder_path)** lists all files in the folder.
- The list comprehension filters files to include only those with image extensions **('.png', '.jpg', '.jpeg', '.gif', '.bmp').**

Step 2: Calculate Number of Rows

```
    num_images = len(image_files)
    num_rows = (num_images + num_cols - 1) // num_cols
```

-Purpose: Calculate the number of rows needed to display all images.

-Explanation:

- **num_images = len(image_files)** counts the total number of image files.
- **num_rows = (num_images + num_cols - 1) // num_cols** calculates the number of rows required by dividing the number of images by the number of columns and rounding up.

Step 3: Create Figure and Axes

```
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 5 * num_rows))
fig.set_facecolor('black')  # Set the background of the figure to black
axes = axes.ravel()
```

Purpose: Create a figure and a grid of subplots to display images.

Explanation:

- **plt.subplots(num_rows, num_cols, figsize=(15, 5 * num_rows))** creates a figure with a grid of subplots, where each subplot will display an image. The figure size is adjusted based on the number of rows.
- **fig.set_facecolor('black')** sets the background color of the figure to black.
- **axes = axes.ravel()** flattens the 2D array of axes into a 1D array for easy iteration.

Step 4: Display Images in Subplots

```
for idx, image_file in enumerate(image_files):
    image_path = os.path.join(folder_path, image_file)
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    axes[idx].imshow(image)
    axes[idx].axis('off')
    axes[idx].set_facecolor('black')  # Set the background of the axis to black
```

Purpose: Read and display each image in the subplots.
Explanation:

- The loop iterates over **image_files** with enumerate to get both the index **(idx)** and the file name **(image_file).**
- **image_path = os.path.join(folder_path, image_file)** constructs the full path to the image file.
- **image = cv2.imread(image_path)** reads the image from the file.
- **image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)** converts the image from BGR (OpenCV default) to RGB (matplotlib default).
- **axes[idx].imshow(image)** displays the image in the corresponding subplot.
- **axes[idx].axis('off')** hides the axes for a cleaner look.
- **axes[idx].set_facecolor('black')** sets the background color of the axis to black.

Step 5: Handle Empty Subplots

```
for i in range(idx + 1, len(axes)):
    axes[i].axis('off')
    axes[i].set_facecolor('black')  # Set the background of the axis to black
plt.tight_layout()
plt.show()
```

Purpose: Hide any remaining subplots that do not contain images.
Explanation:

- The loop iterates over the remaining axes starting from **idx + 1** (the index after the last image).
- **axes[i].axis('off')** hides the axes.
- **axes[i].set_facecolor('black')** sets the background color of the axis to black.

Step 6: Final Adjustments and Display

```
plt.tight_layout()
plt.show()
```

Purpose: Adjust the layout and display the figure.
Explanation:

- **plt.tight_layout()** adjusts the padding between and around subplots to minimize overlap.
- **plt.show()** displays the figure.

2.7 READ IMAGE AND MAIN FUNCTION

```
### Function to read images from a folder and detect traffic signs
def read_image(folder_red_path, output_folder, blue=False):
    image_files = os.listdir(folder_red_path)
    for image_file in image_files:
        if image_file.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp')):
            image_path = os.path.join(folder_red_path, image_file)
            findTrafficSign(image_path, output_folder, blue)

### Main function
def main():
    folder_path = 'image_red'
    output_folder = 'image_output'
    bb = 'box'
    output_folder_bb = output_folder + bb
    read_image(folder_path, output_folder)
    display_images_in_folder(output_folder_bb)
    display_images_in_folder(output_folder)

if __name__ == '__main__':
    main()
```

First I will create the function name **'read_image'**

```
### Function to read images from a folder and detect traffic signs
def read_image(folder_red_path, output_folder, blue=False):
```

folder_red_path: The path to the folder containing images.
output_folder: The folder where the output images will be saved.
blue: A boolean flag to indicate whether to detect blue traffic signs. Default is False.

List Image Files

```
    image_files = os.listdir(folder_red_path)
```

-Purpose: List all files in the specified folder.
-Explanation:
'os.listdir(folder_red_path) 'returns a list of all files and directories in the specified folder.

Loop Through Image Files

```
    for image_file in image_files:
```

-Purpose: Iterate over each file in the folder.

Filter Image Files

```
if image_file.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp')):
```

-Purpose: Check if the file is an image.

-Explanation:

- **image_file.lower()** converts the file name to lowercase to make the check case-insensitive.
- **.endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp'))** checks if the file name ends with any of the specified image extensions.

Construct Image Path

```
image_path = os.path.join(folder_red_path, image_file)
```

-Purpose: Create the full path to the image file.

-Explanation:

- ***os.path.join(folder_red_path, image_file)*** combines the folder path and the image file name to create the full file path.

Detect Traffic Signs

```
    findTrafficSign(image_path, output_folder, blue)
```

-Purpose: Call the findTrafficSign function to detect traffic signs in the image.

-Explanation:

findTrafficSign(image_path, output_folder, blue) calls the function to process the image and detect traffic signs based on the provided parameters.

```
### Main function
def main():
    folder_path = 'image_red'
    output_folder = 'image_output'
    bb = 'box'
    output_folder_bb = output_folder + bb
    read_image(folder_path, output_folder)
    display_images_in_folder(output_folder_bb)
    display_images_in_folder(output_folder)

if __name__ == '__main__':
    main()
```

folder_path: The path to the folder containing the input images. In this case, it is set to '**image_red**'.

output_folder: The path to the folder where the processed images will be saved. It is set to '**image_output**'.
bb: A string 'box' used to create a subfolder for storing images with bounding boxes.

Create Output Folder for Bounding Box Images

```
bb = 'box'
output_folder_bb = output_folder + bb
```

Purpose: Concatenate the output_folder and bb to create the path for the subfolder where images with bounding boxes will be saved.
Explanation:
output_folder_bb will be '**image_outputbox**'.

```
if __name__ == '__main__':
    main()
```

=> ***Purpose:*** Ensure that the main function is called only when the script is executed directly, not when it is imported as a module.
-Explanation:
if __name__== '__main__': checks if the script is being run directly.
main() calls the main function to start the program.

2.8. Result

My result each image have 2 output the first ouput name **'image_outputbox'** I will show the image **with rectangular** and the second ouput will be the final result name **'image_output'** I will show the image were **drawn rectangular and the content of prohibit sign**

Input

Name of input folder: **image_red**

'No taxi sign'

'No U-Turn sign'

'No Left Turn Sign'

'No-car sign'

'Height-Limit-3.7m sign'

‘No-horn sign’

'Speed-Limit 20 km/h sign'


'No-Entry sign'

'No go straight sign'

'No-right-turn sign'

‘No-parking sign’

‘Signs for tractors’

'No-cycling sign'

'Stop-sign'

'No-Entry Sign'

1. First Result

-output name: **image_outputbox**

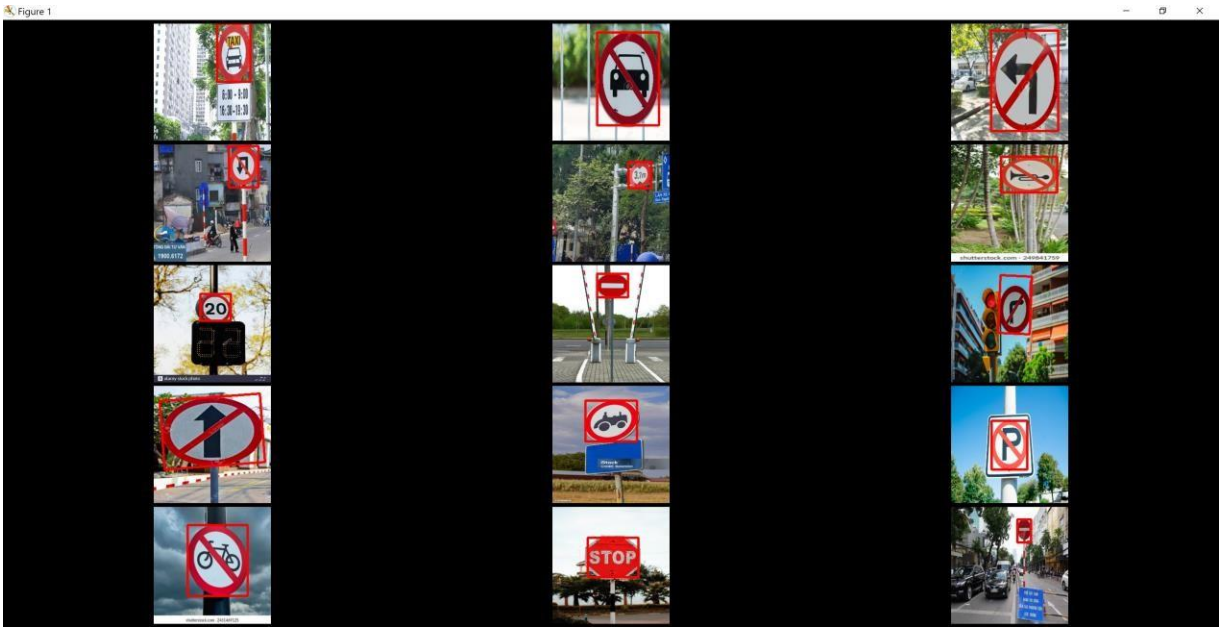
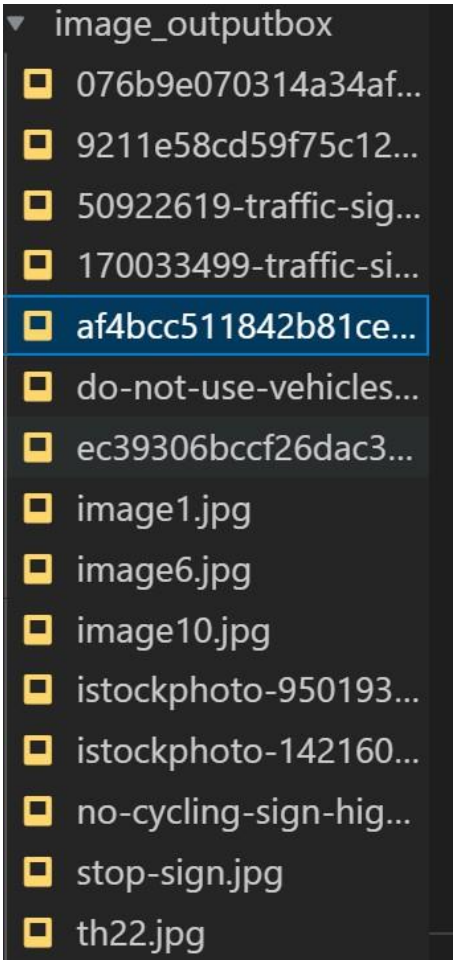


Figure 1
STOP


image_outputbox
076b9e070314a34af...
9211e58cd59f75c12...
50922619-traffic-sig...
170033499-traffic-si...
af4bcc511842b81ce...
do-not-use-vehicles...
ec39306bccf26dac3...
image1.jpg
image6.jpg
image10.jpg
istockphoto-950193...
istockphoto-142160...
no-cycling-sign-hig...
stop-sign.jpg
th22.jpg

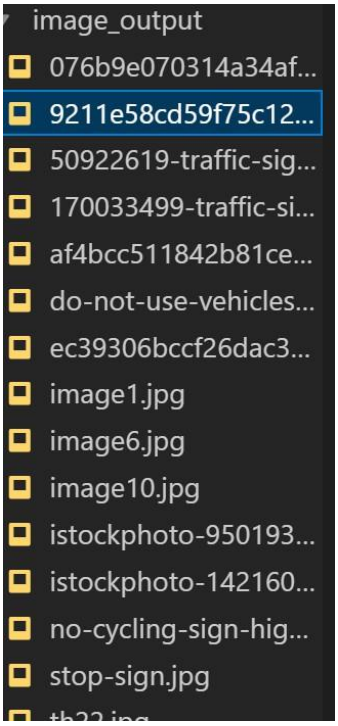
Or you can click on each image in the output to see the result


20
alamy stock photo
P
iStock
Credit: Baioncici

2. Final Result

-output name : **image_output**

After close the tab show first result the final result will be shown



Or you can click on each image in the output to see the result


No-taxi sign
TAXI
6:00 - 9:00
16:30-19:30
No U-Turn sign
TỔNG ĐÀI TƯ VẤN
1900.6172
No Left Turn Sign
No-taxi sign
Height-Limit-3.7m sign
3.7m
No-horn sign
shutterstock.com · 249841759


Speed-Limit 20 km/h sign
20
alamy stock photo
No-Entry Sign
No-go straight sign
No-right-turn sign
No-parking sign
P
Signs for tractors
iStock
Credit: Baioncici

3.REFERENCE

1. https://numpy.org/
2. https://opencv.org/
3. https://www.geeksforgeeks.org/opencv-python-tutorial/

4. https://medium.com/@nimritakoul01/image-processing-using-opencv-python-9c9b83f4b1ca
5. https://www.semanticscholar.org/paper/Traffic-sign-detection-and-recognition-using-OpenCV-Shopa-Sumitha/0c6added736a9cc4931acdd4039f80882e7c0de3