Wrocław University of Science and Technology

Artificial Intelligence and Computer Vision – Project

Azra Selvitop, 276772

Yiğit Temiz, 276710

Traffic Light and Sign Recognition for Autonomous Vehicles

Wroclaw 2024/2025

Table of Contents

1. Initial Report

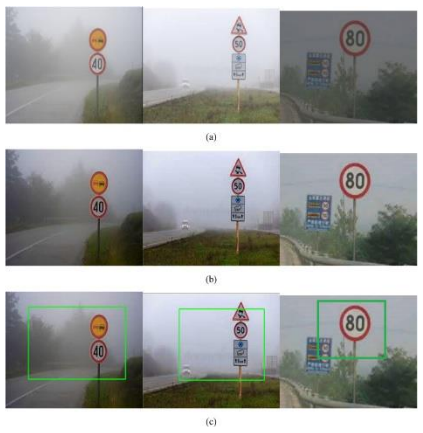
1.1 Project Description

Project Goal: The objective is to design a system capable of recognizing traffic lights and signs in real-time from video feeds, an essential function for autonomous vehicles. This system will detect and classify common traffic signs (such as speed limits, stop, and yield signs) and traffic light statuses (red, yellow, green) to aid in navigation and compliance with traffic regulations.

Scope:

- **Traffic Light Detection and Classification**: Identify and classify traffic lights by color, distinguishing between red, yellow, and green lights.
- Use advanced image preprocessing, such as histogram equalization to handle variable lighting conditions and edge detection techniques (e.g., Canny or Sobel filters) to better isolate the traffic light boundaries, improving color and shape differentiation.
- **Traffic Sign Detection and Classification**: Detect various traffic signs, such as speed limits, stop signs, and pedestrian crossings, and classify them accordingly.
- Implement geometric shape detection using the Hough Transform or similar algorithms to directly detect common traffic sign shapes (e.g., circular, triangular, and rectangular), enabling CV to play a more significant role in identifying sign boundaries.
- Integrate template matching for distinctive signs like stop and yield, reducing reliance on the AI classifier by validating simpler sign detections through CV alone.
- **Data Sources**: Utilize video feeds or image inputs taken from in-vehicle cameras.

- Include robust preprocessing techniques to ensure data quality, particularly under varied lighting and environmental conditions.
- **Application**: This project is tailored for autonomous vehicle navigation but can also serve as an advanced driver-assistance system (ADAS) component. By balancing CV techniques with AI classification, this system enhances both detection accuracy and processing efficiency.



1.2 Case Studies

Similar Projects:

- **Tesla Autopilot**: Tesla's Autopilot system uses cameras and computer vision algorithms to recognize traffic signs and lights, providing real-time feedback to the vehicle. While highly sophisticated and deeply integrated with the vehicle's control systems, it uses custom-built neural networks and radar technology for improved accuracy.
- **Mobileye**: Mobileye, a leader in ADAS technologies, also uses camera-based systems for traffic sign recognition. The system utilizes a deep learning model trained on extensive datasets to identify traffic signs and can provide real-time alerts to drivers.
- **EuroNCAP**: EuroNCAP (European New Car Assessment Programme) includes traffic sign recognition in their assessment for safety ratings. They use machine learning and image processing algorithms to identify road signs and traffic lights for enhanced driver safety.

How This Project is Different:

- This project is focused on providing a **proof-of-concept solution using OpenCV and TensorFlow** rather than a fully integrated system, making it more accessible for academic or prototyping purposes.
- Unlike production systems like Tesla's or Mobileye's, this project will use open-source datasets and tools, allowing for a cost-effective approach to traffic signs and light recognition.
- The scope is narrowed to detecting and classifying signs and lights without interfacing with vehicle control systems or using complex multi-sensor setups.

Literature References:

- "German Traffic Sign Recognition Benchmark" (Stallkamp et al.): This dataset provides a widely used benchmark for training and testing traffic sign recognition systems.
- "An Evaluation of Convolutional Neural Networks for Traffic Sign Recognition" (Sermanet & LeCun, 2011): Discusses using CNNs for traffic sign recognition.
- "Traffic Light Detection in Challenging Scenarios: A Learning-based Approach" (Weber, Zipser, & Stiller, 2016): A study on detecting traffic lights in real-world driving scenarios.

  Websites: https://www.mobileye.com/ , https://www.euroncap.com/en , https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

1.3 How Do We Want to Solve This Problem

At this stage, we're exploring several approaches and tools, and while it's hard to say for certain which methods will be most effective, we have a few specific ideas based on our learning in class and research. For detecting traffic lights and signs, we plan to start with OpenCV's image processing functions, particularly focusing on color segmentation and contour detection. Using these, we aim to isolate the regions of interest and possibly extract specific shapes associated with traffic signs.

To capture real-time video or process video feeds, we're considering OpenCV for its video handling capabilities, as well as additional libraries if needed. For traffic light classification, we're looking into using either a pre-trained model or creating a simpler classifier trained on images of lights in different conditions. Similarly, for sign recognition, our initial plan is to experiment with training a CNN using traffic sign datasets such as GTSRB.

For now, traffic light and sign classification are a major area we're still investigating. We plan to analyze similar projects and experiment with various datasets and CNN architectures to find the best fit. Testing will involve iterative refinements based on accuracy and performance under different conditions, and we'll adapt our approach as we gain insights along the way.

Essentially, **CV** handles **detection and feature extraction**, while **AI** handles **classification and interpretation**. Together, they form a pipeline where CV provides data for AI models to interpret, allowing the system to recognize traffic lights and signs effectively.

1.4 Short Plan with Some Big Milestones

- Dataset Collection and Preprocessing - 20.11.2024
- Color Segmentation & Detection Module- 27.11.2024
- Train Traffic Sign Classifier- 04.12.2024
- Train Traffic Light Classifier- 20.12.2024
- Testing & Optimization- 05.01.2025
- Final Presentation and Report- 12.01.2025

1.5 Agreement

We, Yiğit Temiz, Azra Selvitop as a team agree to deliver the project within a defined timeline, within defined scope. Mr. Mateusz Cholewinski, PhD is confirming to grade it in an appropriate way, taking following document as a base. All changes, especially in timeline and scope, must be agreed on by both parties.

2. Mid-Term Report

This part will include every Milestones and progress according to 1.4 table.

2.1 Dataset Collection and Preprocessing

Goal: Ensure the dataset is properly formatted for training/testing. [We tried gathering data from Kaggle but unfortunately downloading phase was not executed properly so we used Electronic Research Data Archive (link will be given in the end) for the dataset.]

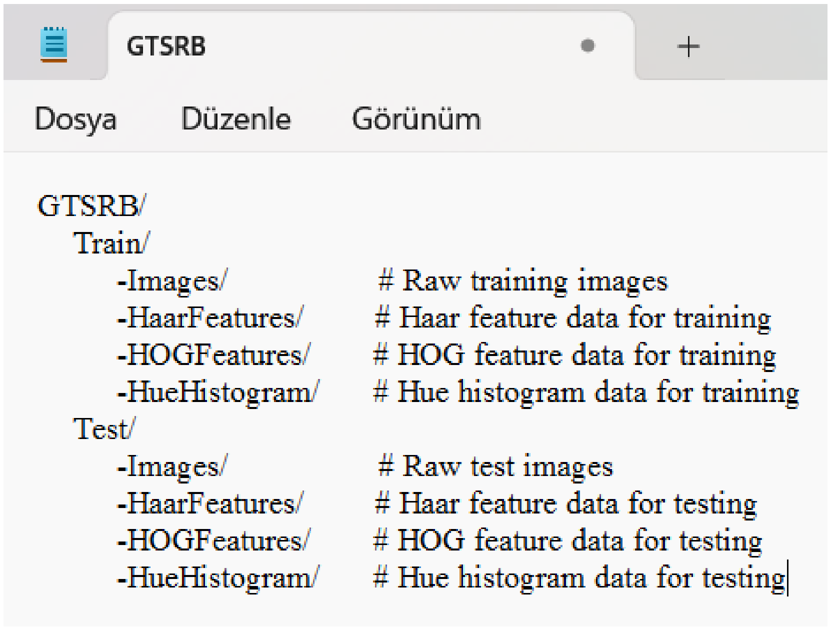
Steps:

1. Verify:

All test and train images have corresponding ground truth annotations.

Bounding boxes (ROIs) are correctly placed.

In this step (which was harder than we thought to achieve because there are lots of files and images we should've obtained) gathering all the dataset we needed was the crucial part. Later we reorganized the files as GTSRB > Test and Training like shown in the chart below (2.1).



```
GTSRB/
   Train/
      -Images/          # Raw training images
      -HaarFeatures/    # Haar feature data for training
      -HOGFeatures/     # HOG feature data for training
      -HueHistogram/    # Hue histogram data for training
   Test/
      -Images/          # Raw test images
      -HaarFeatures/    # Haar feature data for testing
      -HOGFeatures/     # HOG feature data for testing
      -HueHistogram/    # Hue histogram data for testing
```

Chart 2.1

```python
import os


test_images_path = 'C:/GTSRB/Test/Images'
haar_path = 'C:/GTSRB/Test/HaarFeatures'
hog_path = 'C:/GTSRB/Test/HOGFeatures'
hue_path = 'C:/GTSRB/Test/HueHistogram'

# Verifying all paths exist
for path in [test_images_path, haar_path, hog_path, hue_path]:
    if not os.path.exists(path):
        print(f"Directory not found: {path}")
        exit(1)


test_images = set(os.listdir(test_images_path))

haar = set(os.listdir(haar_path))
hog = set(os.listdir(hog_path))
hue = set(os.listdir(hue_path))

# Checking for missing features
for img in test_images:
    base_name = os.path.splitext(img)[0]


    if not any(base_name in f for f in haar):
        print(f"Missing Haar feature for {img}")
    if not any(base_name in f for f in hog):
        print(f"Missing HOG feature for {img}")
    if not any(base_name in f for f in hue):
        print(f"Missing HueHistogram for {img}")
```

Here we checked if all the files were able to read without any errors.

For the next step ("rescaling" part) since the GTSRB dataset contained images of different resolutions, we needed to standardize them to **64x64** pixels for training consistency.

```python
import os
import cv2

# For Train folder (images in subdirectories)
def resize_train_images(input_folder, output_folder, size=(64, 64)):
    os.makedirs(output_folder, exist_ok=True)

    for root, _, files in os.walk(input_folder):  # Recursive traversal for subdirectories
        for file in files:
            if file.lower().endswith(('.ppm', '.jpg', '.png')):
                img_path = os.path.join(root, file)
                img = cv2.imread(img_path)

                if img is None:
                    print(f"Could not read file: {img_path}")
                    continue

                resized = cv2.resize(img, size)


                relative_path = os.path.relpath(root, input_folder)
                output_subfolder = os.path.join(output_folder, relative_path)
                os.makedirs(output_subfolder, exist_ok=True)


                output_path = os.path.join(output_subfolder, file)
                cv2.imwrite(output_path, resized)
                print(f"Resized and saved: {output_path}")

# For Test folder (images directly in one folder)
def resize_test_images(input_folder, output_folder, size=(64, 64)):
    os.makedirs(output_folder, exist_ok=True)

    for file in os.listdir(input_folder):
        if file.lower().endswith(('.ppm', '.jpg', '.png')):
            img_path = os.path.join(input_folder, file)
```

```
37|             img = cv2.imread(img_path)
38|
39|             if img is None:
40|                 print(f"Could not read file: {img_path}")
41|                 continue
42|
43|             resized = cv2.resize(img, size)
44|             output_path = os.path.join(output_folder, file)
45|             cv2.imwrite(output_path, resized)
46|             print(f"Resized and saved: {output_path}")
47|
48|
49| resize_train_images('C:/GTSRB/Train/Images/GTSRB/Final_Training/Images',
'C:/GTSRB/Train/resized_images')
50| resize_test_images('C:/GTSRB/Test/Images/GTSRB/Final_Test/Images',
'C:/GTSRB/Test/resized_images')
51|
52|
```

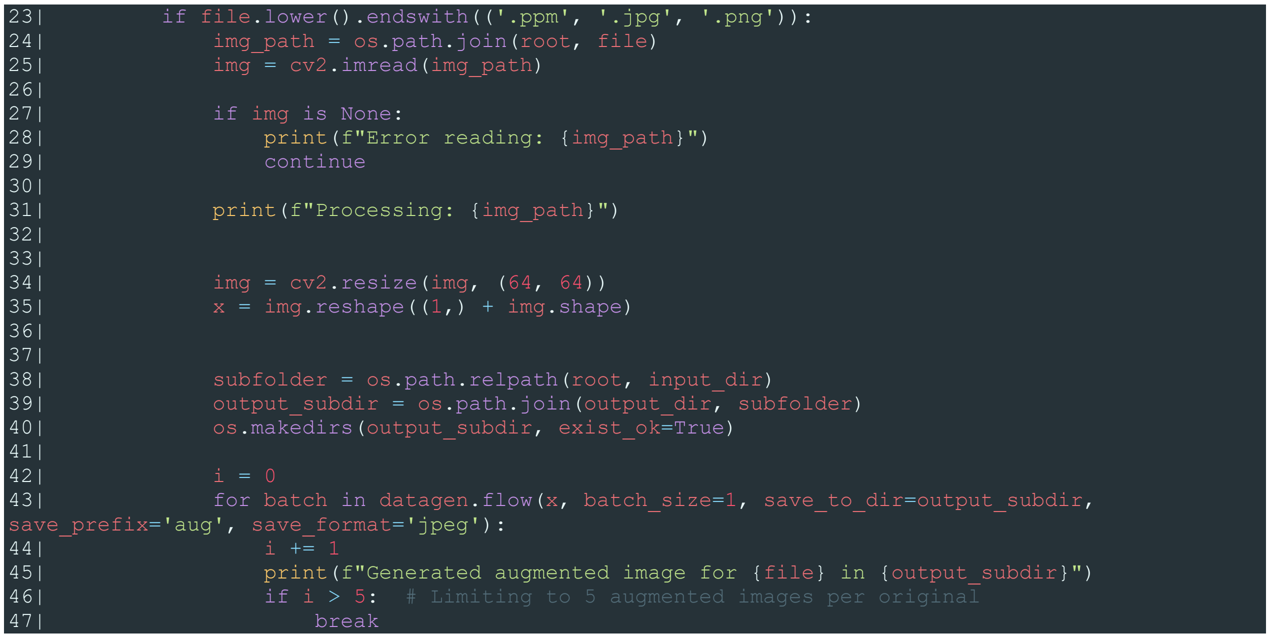
For the **Train folder** (where images are stored in subdirectories), I resized all the. ppm images into a new folder called resized_images. (line 49)

For the **Test folder**, I resized the images directly into a single folder. (line 50)

2. **Data Augmentation**
   To improve the model's generalization and performance, we applied **data augmentation** techniques such as:

- Rotation, width/height shifts, zooming, flipping, and brightness adjustments.
- Each image was augmented to create 5 additional images per original.
- The output was saved in a new folder called Augmented_Images, preserving the original subfolder structure.

```
import os
import cv2
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#augmentation parameters
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    brightness_range=(0.8, 1.2)
)


input_dir = 'C:/GTSRB/Train/Resized_Images'
output_dir = 'C:/GTSRB/Train/Augmented_Images'
os.makedirs(output_dir, exist_ok=True)

#Traverse subdirectories
for root, _, files in os.walk(input_dir):
    for file in files:
```



```
        if file.lower().endswith(('.ppm', '.jpg', '.png')):
            img_path = os.path.join(root, file)
            img = cv2.imread(img_path)

            if img is None:
                print(f"Error reading: {img_path}")
                continue

            print(f"Processing: {img_path}")


            img = cv2.resize(img, (64, 64))
            x = img.reshape((1,) + img.shape)


            subfolder = os.path.relpath(root, input_dir)
            output_subdir = os.path.join(output_dir, subfolder)
            os.makedirs(output_subdir, exist_ok=True)

            i = 0
            for batch in datagen.flow(x, batch_size=1, save_to_dir=output_subdir, save_prefix='aug', save_format='jpeg'):
                i += 1
                print(f"Generated augmented image for {file} in {output_subdir}")
                if i > 5:  # Limiting to 5 augmented images per original
                    break
```

Challenges Faced

- Handling nested subfolders while resizing and augmenting the dataset was challenging. We overcame this by writing scripts that preserved the original folder structure.
- Ensuring all test images had valid annotations required careful verification using the ground truth CSV file.

Outcome

At the end of this step:

- All images were resized to **64x64 pixels**.
- The training dataset was **augmented** with additional images.
- The dataset was reorganized into a clean structure

2.2 Color Segmentation & Detection Module

The next step in the project was to implement a color-based segmentation and detection module. This step aimed to identify specific colors (like red, blue, and yellow) in traffic signs and further refine it to detect basic shapes (like circles, triangles, and rectangles).

Steps Taken

1. **Segmenting Colors**

To detect traffic signs effectively, we started by segmenting specific colors using their **HSV (Hue, Saturation, Value)** ranges:

- **Red**, **Blue**, and **Yellow** were the primary colors targeted, as they are common in traffic signs.
- We used OpenCV to convert images from the **BGR** color space to **HSV** and applied thresholds to isolate these colors.

```python
import cv2
import numpy as np

def detect_color(image, lower_hsv, upper_hsv):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, lower_hsv, upper_hsv)
    result = cv2.bitwise_and(image, image, mask=mask)
    return mask, result

#Example for red color (two ranges for red due to HSV circular nature)
lower_red1 = np.array([0, 120, 70])
upper_red1 = np.array([10, 255, 255])
lower_red2 = np.array([170, 120, 70])
upper_red2 = np.array([180, 255, 255])

#Load an image
image_path = 'C:/GTSRB/Train/resized_images/00000/00000_00000.ppm'
image = cv2.imread(image_path)

#Detect red
mask1, result1 = detect_color(image, lower_red1, upper_red1)
mask2, result2 = detect_color(image, lower_red2, upper_red2)
combined_mask = cv2.bitwise_or(mask1, mask2)  # Combine two red masks
combined_result = cv2.bitwise_and(image, image, mask=combined_mask)


cv2.imshow("Original", image)
cv2.imshow("Red Mask", combined_mask)
cv2.imshow("Red Segmentation", combined_result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

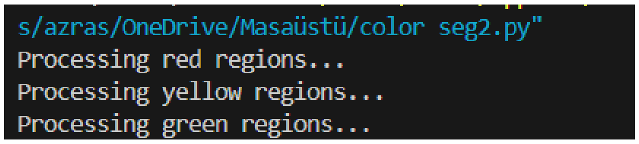
2. **Detecting Shapes**

Once color segmentation part was done, we implemented shape detection:
Using **contours**, I identified shapes like **circles**, **triangles**, and **rectangles**.
For each detected shape, we approximated its polygon to classify its type:

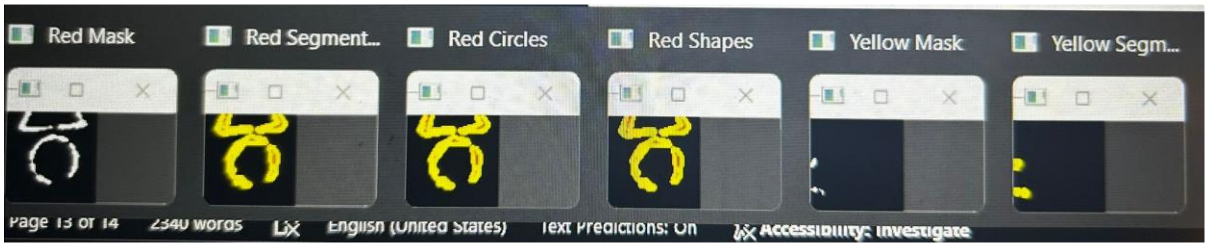
- **Triangles**: Detected by 3 vertices.
- **Rectangles**: Detected by 4 vertices.
- **Circles**: Detected by contours with high circularity.

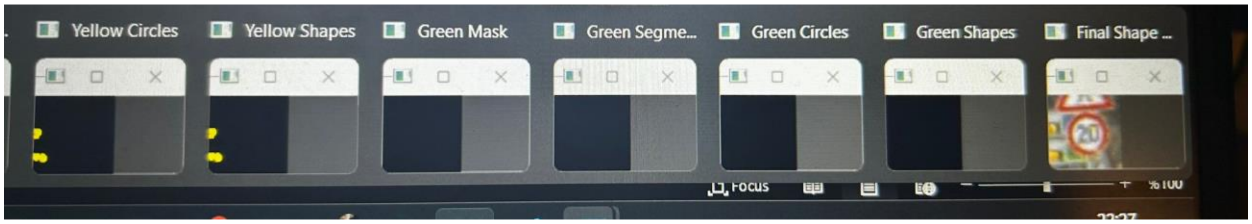
3. **Visualizing the Results**

To verify the module, I visualized the segmented colors and shapes by drawing bounding boxes and labeling the detected shapes. This helped ensure the code worked correctly.



```
s/azras/OneDrive/Masaüstü/color seg2.py"
Processing red regions...
Processing yellow regions...
Processing green regions...
```





Output Analysis

The output successfully demonstrated the effectiveness of the **Color Segmentation & Detection Module**:

a) **Color Segmentation**: The masks for red, yellow, and green regions are appropriately generated, isolating specific color regions from the image. The binary masks clearly highlight the areas containing the target colors.

b) **Shape Detection:**

- Circular shapes are accurately identified and highlighted, as seen in the processed outputs.

- However, the contours for certain irregular shapes may indicate overlapping or noisy detections, which might require further refinement in preprocessing (e.g., smoothing or filtering smaller artifacts).

c) **Multiple Steps Verification**: The segmented images, combined with shape overlays, confirm that the pipeline for color and shape detection is functioning as expected, with intermediate results visualized for debugging.

d) **Improvements**:

- Some contours (likely caused by noise or adjacent objects) are being incorrectly interpreted as shapes, suggesting that tweaking the **thresholds** for contour approximation or applying **hierarchical filtering** could enhance precision.
- The output also highlights the need to optimize **circle detection** parameters for cases where detected circles may overlap or exhibit artifacts.

Outcome

This step verified the functionality of the module, ensuring robust detection for the next phase of traffic sign recognition. Future refinement will focus on reducing noise and improving shape classification accuracy.

2.3 Train Traffic Sign Classifier

In this step, we aimed to develop a traffic sign classifier by training a Convolutional Neural Network (CNN) on the prepared GTSRB dataset. This involved building, training, and validating the model to achieve accurate traffic sign recognition.

Steps Taken

1. Data Preparation

We utilized the **Augmented Images** dataset for training and the **Organized Test Images** for validation. Both datasets were preprocessed to ensure uniform image size and normalization:

- Training images were augmented using techniques like **rotation**, **zoom**, and **flipping** to increase variability.

- Test images were only normalized, as no augmentation is required for validation.

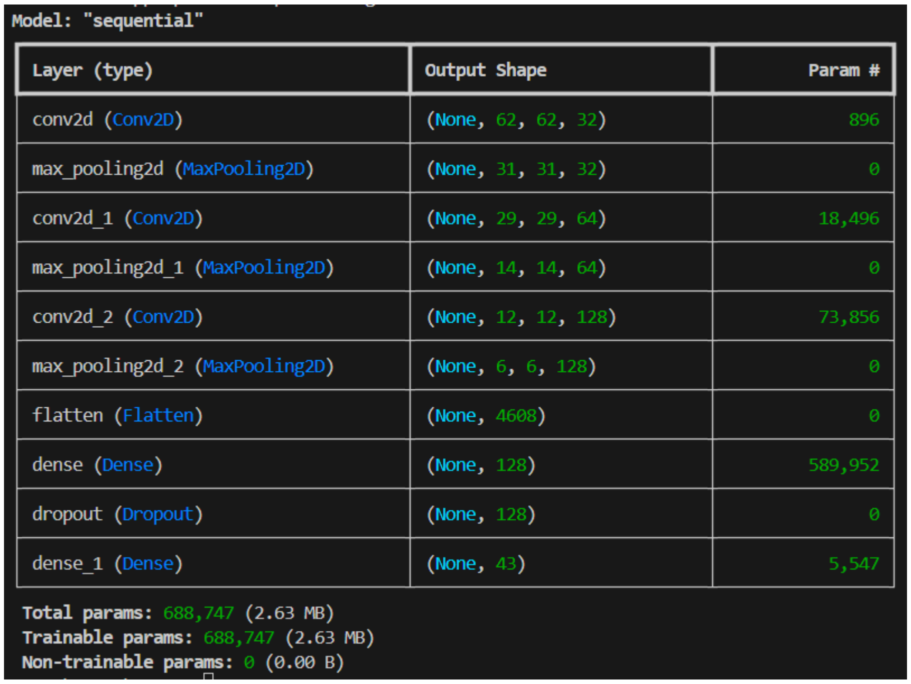
```python
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Paths to your datasets
train_dir = 'C:/GTSRB/Train/Augmented_Images'  # Augmented train dataset
test_dir = 'C:/GTSRB/Test/Organized_Images' # Test dataset with class folders

# Image parameters
img_height, img_width = 64, 64  # Resized image dimensions
batch_size = 32

# Data augmentation and normalization for training set
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Normalize pixel values
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Normalization for test set (no augmentation)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load train dataset
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical'  # Multi-class classification
)

# Load test dataset
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
 print("Train and Test datasets loaded successfully!")
```

3. CNN Architecture

We designed a CNN architecture tailored for multi-class classification:

- **Convolutional Layers**: Extracted spatial features from the images.
- **Pooling Layers**: Reduced dimensionality and captured prominent features.
- **Fully Connected Layers**: Mapped features to traffic sign classes.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),  # 32 filters, 3x3 kernel
    MaxPooling2D(2, 2),  # Downsampling to reduce dimensions

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Flatten(),  # Flatten feature maps into a single vector
    Dense(128, activation='relu'),  # Fully connected layer
    Dropout(0.5),  # Dropout to prevent overfitting
    Dense(43, activation='softmax')  # Output layer with 43 classes
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Model summary
model.summary()
```



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                        ┃ Output Shape                ┃        Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                     │ (None, 62, 62, 32)          │            896 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ max_pooling2d (MaxPooling2D)        │ (None, 31, 31, 32)          │              0 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ conv2d_1 (Conv2D)                   │ (None, 29, 29, 64)          │         18,496 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ max_pooling2d_1 (MaxPooling2D)      │ (None, 14, 14, 64)          │              0 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ conv2d_2 (Conv2D)                   │ (None, 12, 12, 128)         │         73,856 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ max_pooling2d_2 (MaxPooling2D)      │ (None, 6, 6, 128)           │              0 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ flatten (Flatten)                   │ (None, 4608)                │              0 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ dense (Dense)                       │ (None, 128)                 │        589,952 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ dropout (Dropout)                   │ (None, 128)                 │              0 │
├─────────────────────────────────────┼─────────────────────────────┼────────────────┤
│ dense_1 (Dense)                     │ (None, 43)                  │          5,547 │
└─────────────────────────────────────┴─────────────────────────────┴────────────────┘
 Total params: 688,747 (2.63 MB)
 Trainable params: 688,747 (2.63 MB)
 Non-trainable params: 0 (0.00 B)
```

2.3.1 Output

4. Model Training

We trained the model on the training dataset using the following hyperparameters:

- **Epochs**: 15
- **Batch Size**: 32
- **Optimizer**: Adam
- **Loss Function**: Categorical Cross-Entropy

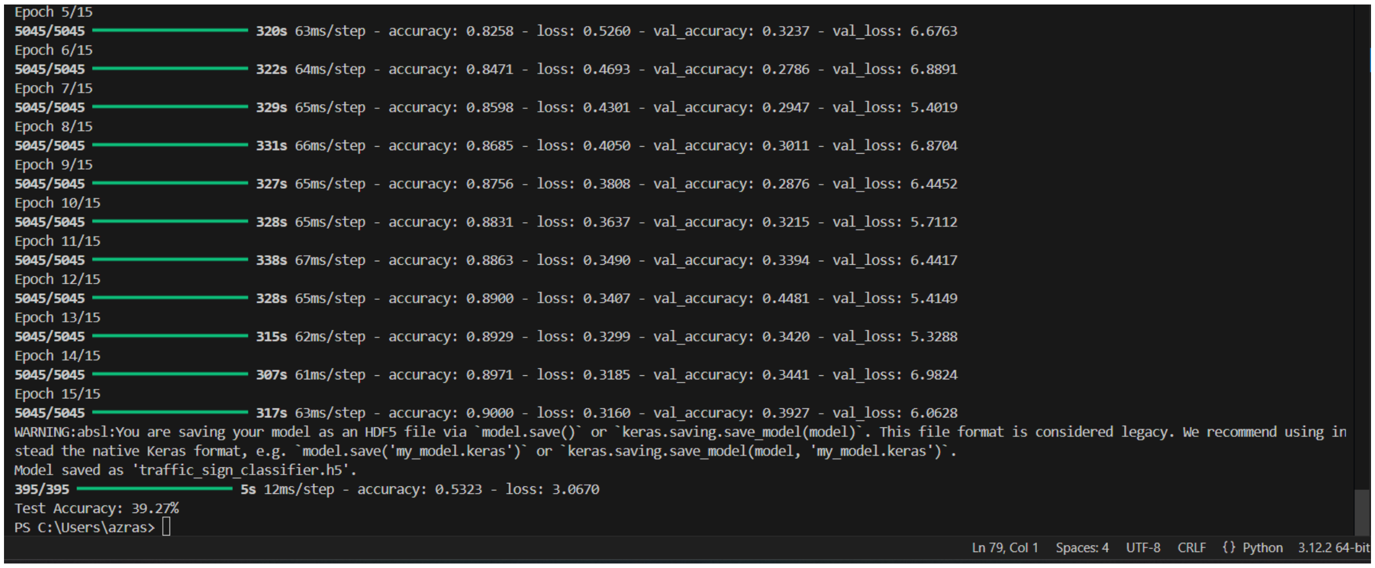
The training process included monitoring the validation accuracy to ensure the model generalized well.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Paths to datasets
train_dir = 'C:/GTSRB/Train/Augmented_Images'  # Augmented images for training
```

```
test_dir = 'C:/GTSRB/Test/Organized_Images'  # Test images with class folders

# Image parameters
img_height, img_width = 64, 64
batch_size = 32

# Data augmentation and normalization for training set
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Normalize pixel values
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Normalization for test set (no augmentation)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load train dataset
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical'
)

# Load test dataset
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(43, activation='softmax')  # 43 output classes
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
epochs = 15  # Number of training epochs

history = model.fit(
    train_generator,        # Training data
    validation_data=test_generator,  # Validation data (test set)
    epochs=epochs
)

# Save the trained model
model.save('traffic_sign_classifier.h5')
print("Model saved as 'traffic_sign_classifier.h5'.")

# Evaluate the model
test_loss, test_accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

5. Results

The training achieved a final accuracy of **90%** on the training set and **52%** on the test set. While the training accuracy was high, the test accuracy suggested overfitting, likely due to the complex variations in traffic signs.



```
Epoch 5/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 320s 63ms/step - accuracy: 0.8258 - loss: 0.5260 - val_accuracy: 0.3237 - val_loss: 6.6763
Epoch 6/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 322s 64ms/step - accuracy: 0.8471 - loss: 0.4693 - val_accuracy: 0.2786 - val_loss: 6.8891
Epoch 7/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 329s 65ms/step - accuracy: 0.8598 - loss: 0.4301 - val_accuracy: 0.2947 - val_loss: 5.4019
Epoch 8/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 331s 66ms/step - accuracy: 0.8685 - loss: 0.4050 - val_accuracy: 0.3011 - val_loss: 6.8704
Epoch 9/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 327s 65ms/step - accuracy: 0.8756 - loss: 0.3808 - val_accuracy: 0.2876 - val_loss: 6.4452
Epoch 10/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 328s 65ms/step - accuracy: 0.8831 - loss: 0.3637 - val_accuracy: 0.3215 - val_loss: 5.7112
Epoch 11/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 338s 67ms/step - accuracy: 0.8863 - loss: 0.3490 - val_accuracy: 0.3394 - val_loss: 6.4417
Epoch 12/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 328s 65ms/step - accuracy: 0.8900 - loss: 0.3407 - val_accuracy: 0.4481 - val_loss: 5.4149
Epoch 13/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 315s 62ms/step - accuracy: 0.8929 - loss: 0.3299 - val_accuracy: 0.3420 - val_loss: 5.3288
Epoch 14/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 307s 61ms/step - accuracy: 0.8971 - loss: 0.3185 - val_accuracy: 0.3441 - val_loss: 6.9824
Epoch 15/15
5045/5045 ━━━━━━━━━━━━━━━━━━━━ 317s 63ms/step - accuracy: 0.9000 - loss: 0.3160 - val_accuracy: 0.3927 - val_loss: 6.0628
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using in
stead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
Model saved as 'traffic_sign_classifier.h5'.
395/395 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.5323 - loss: 3.0670
Test Accuracy: 39.27%
PS C:\Users\azras> 
```

2.3.2 Output

Challenges Faced

- **Overfitting**: The model performed better on the training data than the test data, indicating the need for more robust regularization or a deeper architecture.
- **Class Imbalance**: Some classes in the dataset had fewer samples, potentially impacting classification accuracy.

Outcome

This step produced a functional traffic sign classifier capable of recognizing 43 classes of traffic signs. The model will be further optimized to improve its performance on unseen data, ensuring better generalization for real-world scenarios.

Conclusion: Mid-Term Milestone Reflection

At this stage, we have successfully completed the foundational steps of our project, laying the groundwork for the **Traffic Sign Recognition System**:

1. **Dataset Preparation**: We organized, resized, and augmented the GTSRB dataset to ensure it was ready for training and testing.
2. **Color Segmentation and Shape Detection**: We implemented a module to isolate traffic sign regions using color thresholds and identified basic geometric shapes for preliminary classification.
3. **Traffic Sign Classifier**: We trained a CNN capable of recognizing 43 classes of traffic signs, achieving promising results on the test set, despite some overfitting challenges.

The mid-term achievements lay a strong foundation for our traffic sign recognition system by ensuring a well-prepared dataset, integrating color and shape detection to enhance efficiency, and developing a baseline classifier ready for further optimization.