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 ... 3
1.1 Project Description ... 3
1.2 Case Studies ... 4
1.3 How Do We Want to Solve This Problem ... 5
1.4 Short Plan with Some Big Milestones ... 6
1.5 Agreement ... 6
2. Mid-Term Report ... 7
2.1 Dataset Collection and Preprocessing ... 7
2.2 Color Segmentation & Detection Module ... 10
2.3 Train Traffic Sign Classifier ... 12
Conclusion: Mid-Term Milestone Reflection ... 16
3. Final Report ... 17
3.1 Traffic Light Classifier ... 17
3.2 Traffic Sign Recognition Further Adjustments ... 29
CONCLUSION ... 33

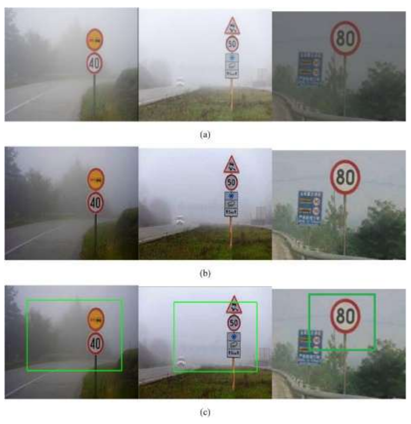
1. Initial Report

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

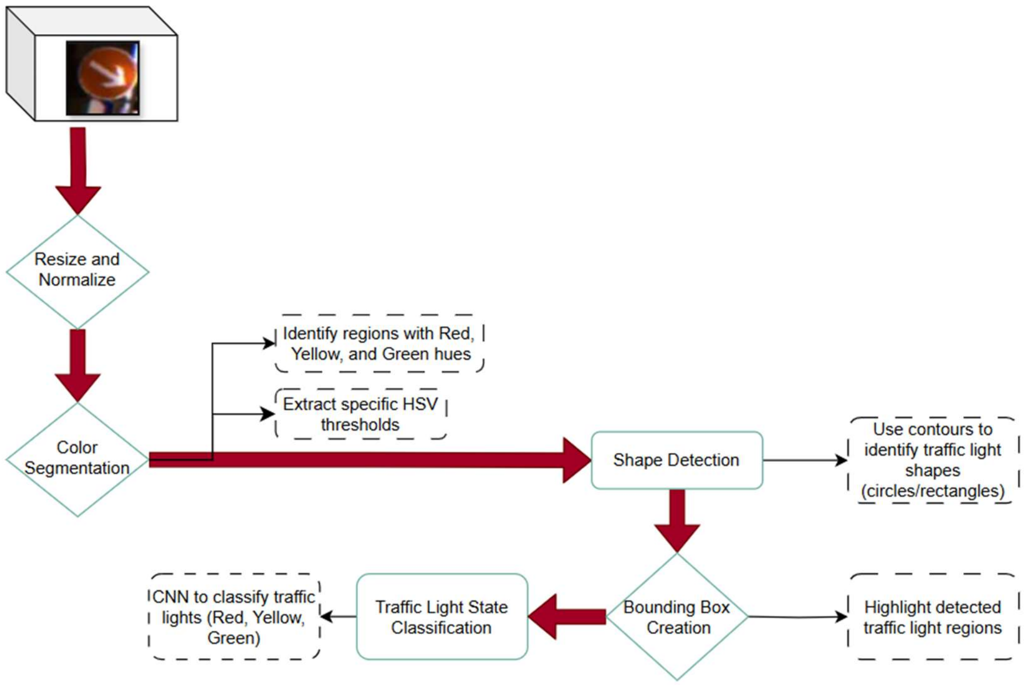
We, Yiğit Temiz, Azra Selvitop as a team agree to deliver the project within a defined timeline, within defined scope. Mr. Mateusz Cholewinski, PhD is confirming to grade it in an appropriate way, taking following document as a base. All changes, especially in timeline and scope, must be agreed on by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


Resize and Normalize
Color Segmentation
Identify regions with Red, Yellow, and Green hues
Extract specific HSV thresholds
Shape Detection
Use contours to identify traffic light shapes (circles/rectangles)
Bounding Box Creation
Highlight detected traffic light regions
Traffic Light State Classification
CNN to classify traffic lights (Red, Yellow, Green)

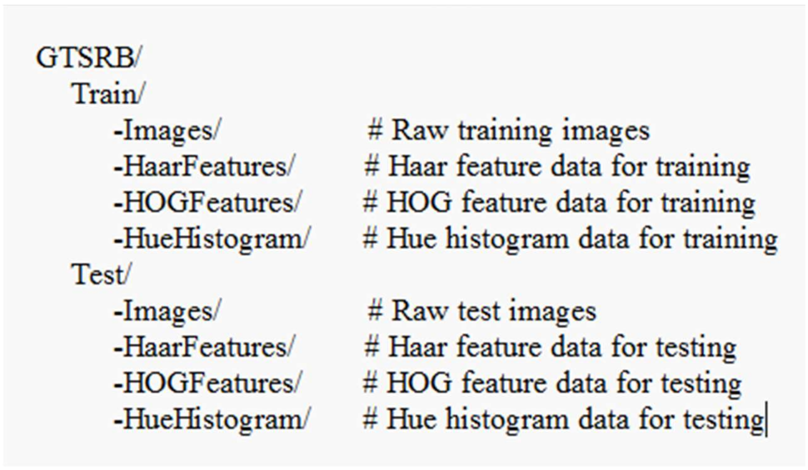
Steps:

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

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.

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.

```python
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.

Color segmentation code:

```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()
```

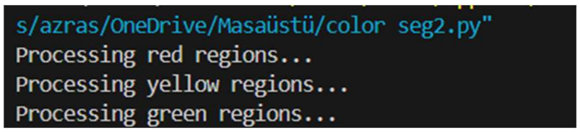
2. Detecting Shapes

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:

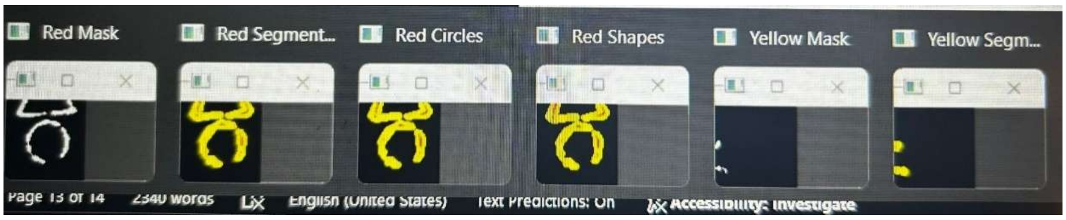
- **Triangles**: Detected by 3 vertices.
- **Rectangles**: Detected by 4 vertices.
- **Circles**: Detected by contours with high circularity.

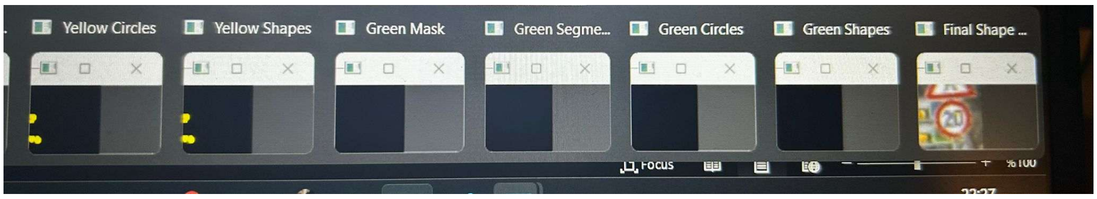
3. Visualizing the Results

To verify the module, we 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...
```





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.

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.

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 CNN on the prepared GTSRB dataset. Involves building, training, and validating the model.

Steps Taken

1. Data Preparation

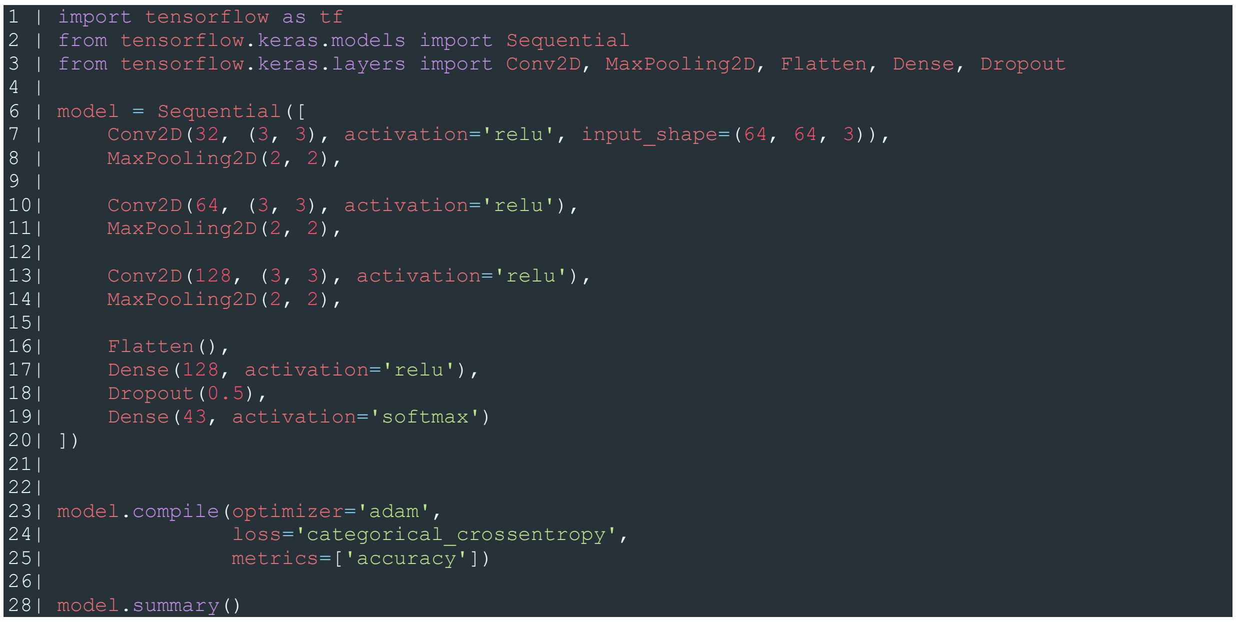
We utilized the **Augmented Images** dataset for training and the **Organized Test Images** for validation. Both datasets were preprocessed to ensure uniform image size and normalization:

- 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.

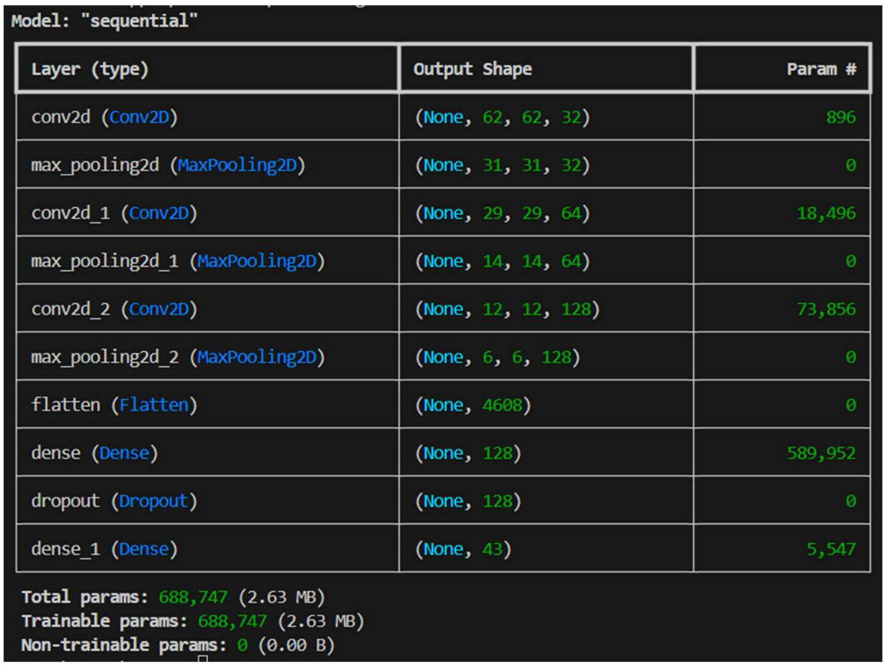
3. CNN Architecture

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

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')
])


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

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

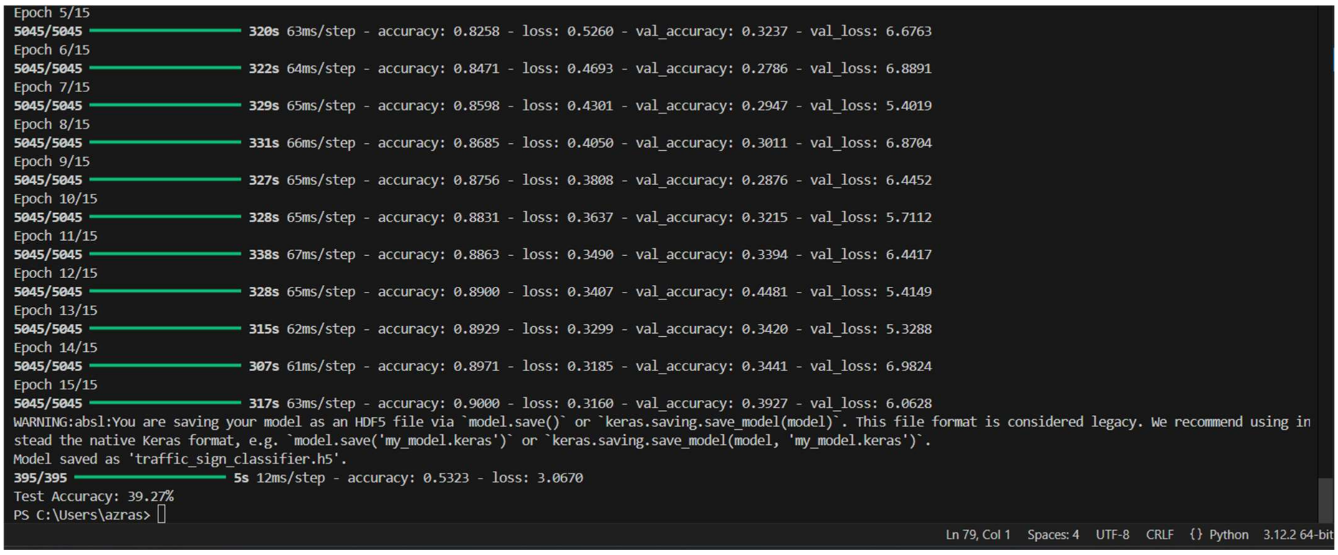
The training process included monitoring the validation accuracy to ensure the model generalized well.

```python
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


train_dir = 'C:/GTSRB/Train/Augmented_Images'
test_dir = 'C:/GTSRB/Test/Organized_Images'

img_height, img_width = 64, 64
batch_size = 32


train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

test_datagen = ImageDataGenerator(rescale=1./255)


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

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

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')
])
```

```python

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

epochs = 15

history = model.fit(
    train_generator,
    validation_data=test_generator,
    epochs=epochs
)
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.

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.

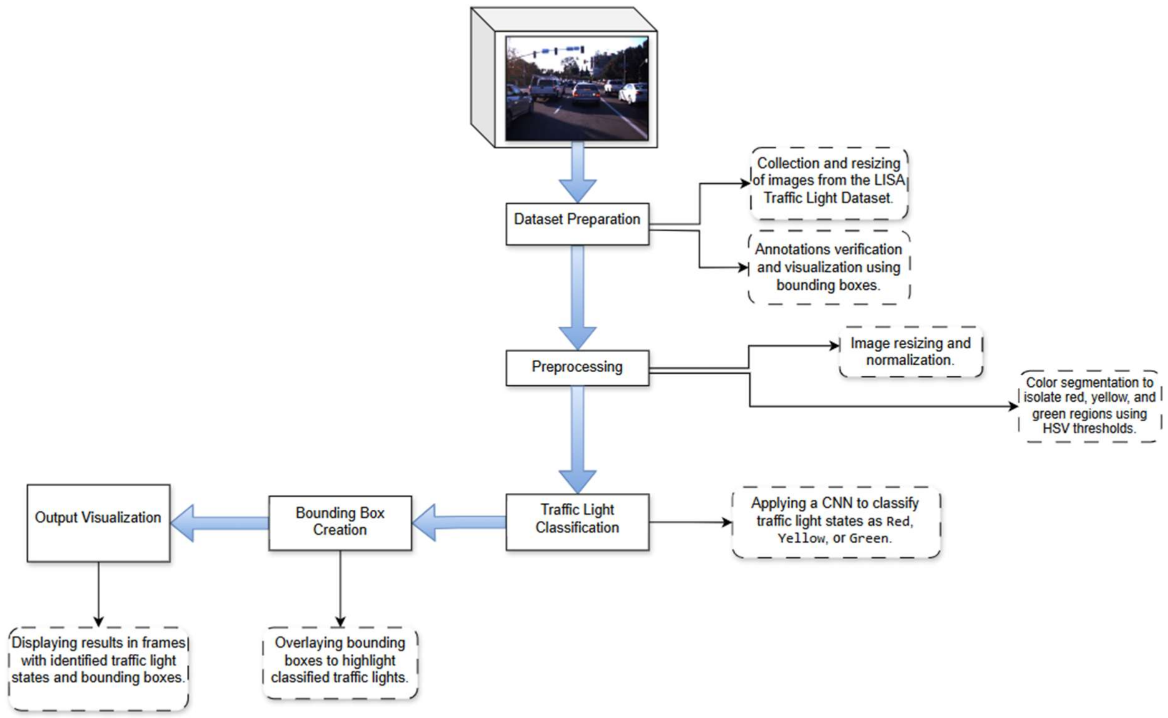
Even though our training process was not accurate enough or as expected we believe the Computer Vision part was completed successfully. The mid-term achievements lay a strong foundation for our traffic sign recognition system by ensuring a well-prepared dataset, integrating color and shape detection to enhance efficiency, and developing a baseline classifier ready for further optimization.

3. Final Report

In the final part of the report our plan was according to the 1.4. short plan:

- Train Traffic Light Classifier
- Traffic Sign Further Adjustments*
- Testing & Optimization
- Final Presentation and Report

*This part was included due to challenges faced in the Mid-term phase.


Dataset Preparation
Collection and resizing of images from the LISA Traffic Light Dataset.
Annotations verification and visualization using bounding boxes.
Preprocessing
Image resizing and normalization.
Color segmentation to isolate red, yellow, and green regions using HSV thresholds.
Traffic Light Classification
Applying a CNN to classify traffic light states as Red, Yellow, or Green.
Bounding Box Creation
Overlaying bounding boxes to highlight classified traffic lights.
Output Visualization
Displaying results in frames with identified traffic light states and bounding boxes.

3.1 Traffic Light Classifier

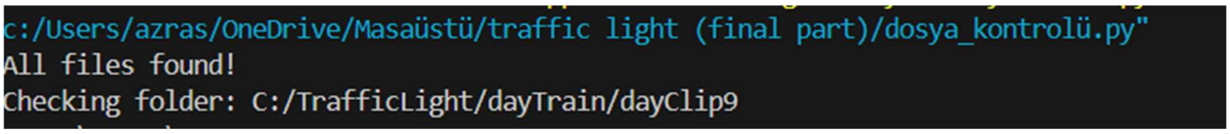
In this part of the job, we again gathered the dataset like it was done in the Traffic Sign Classifier. As our goal is to specify the traffic lights as red, green and yellow we used **LISA Traffic Light Dataset**.

Steps:

1) Dataset Preparation

After the installation part we analyzed the folders and decided to only use “dayClips”. The main reason for this problem was when we resized images to make the training part more executable. Unfortunately .jpg files were unreadable to train, so we decided to work on original pictures. In the meantime, we lost plenty of time figuring out this and we realized that also with this original size of the pictures were too large for our systems CPU/RAM to handle during training. So, we were focused on dayClips.

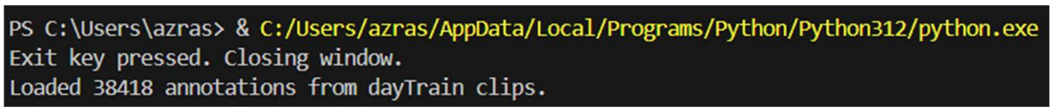
DayClips had 13 different folders and annotations so to see the total number of .jpg files and test if they are all executable we ran a control code.



```
c:/Users/azras/OneDrive/Masaüstü/traffic light (final part)/dosya_kontrolü.py"
All files found!
Checking folder: C:/TrafficLight/dayTrain/dayClip9
```

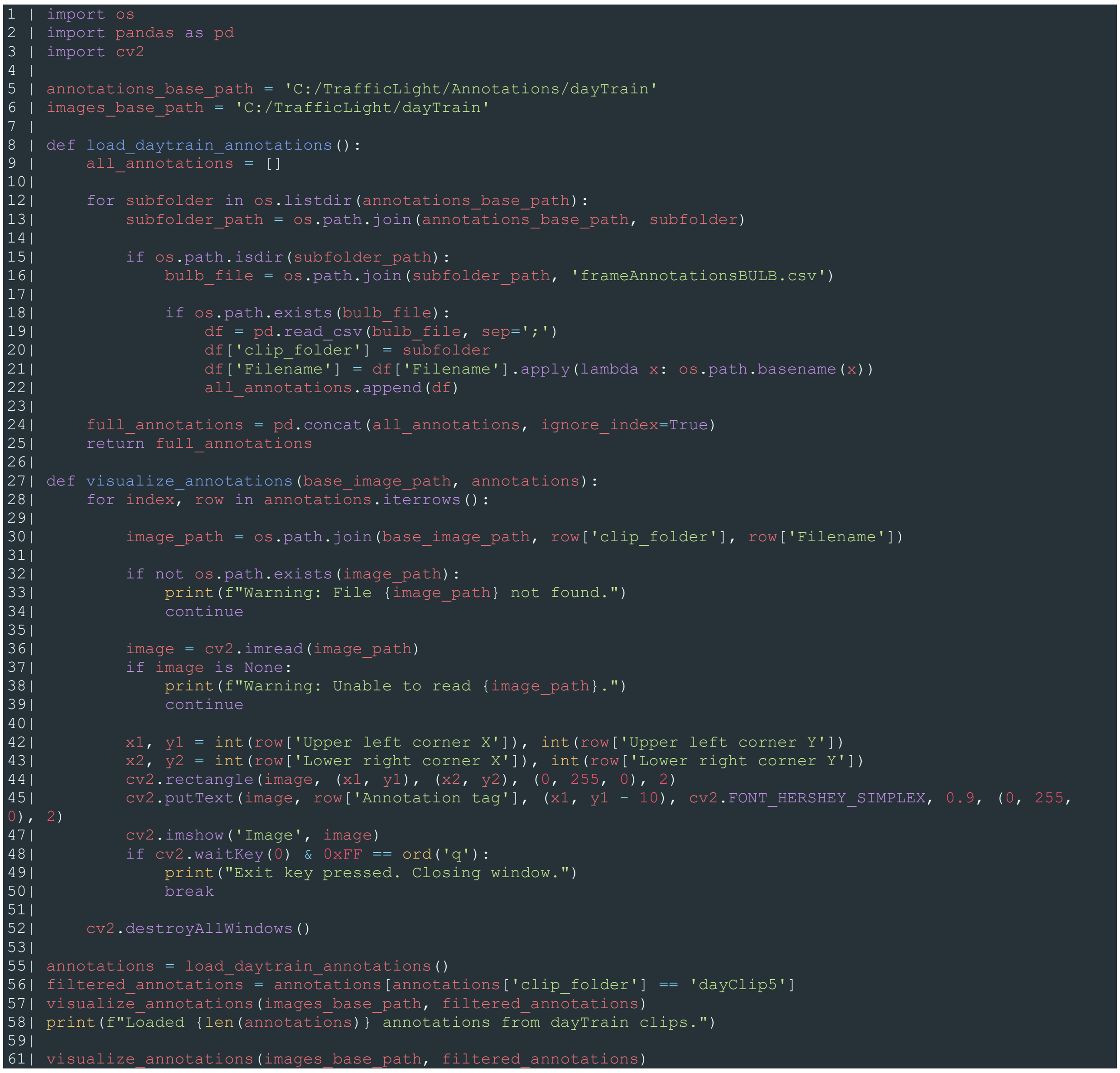
Fig.3.1. File checking output

In this part of the project, we chose to use *frameAnnotationsBULB.csv* as it provides annotations specifically for the illuminated bulbs (red, yellow, green signals), focusing on the actual traffic light indication, rather than the broader bounding boxes around the entire traffic light structure (as in *frameAnnotationsBOX.csv*). This decision ensures a more precise analysis of the lit-up signals for our task.



```
PS C:\Users\azras> & C:/Users/azras/AppData/Local/Programs/Python/Python312/python.exe
Exit key pressed. Closing window.
Loaded 38418 annotations from dayTrain clips.
```

Fig.3.2. Annotations checking output



```python
import os
import pandas as pd
import cv2

annotations_base_path = 'C:/TrafficLight/Annotations/dayTrain'
images_base_path = 'C:/TrafficLight/dayTrain'

def load_daytrain_annotations():
    all_annotations = []

    for subfolder in os.listdir(annotations_base_path):
        subfolder_path = os.path.join(annotations_base_path, subfolder)

        if os.path.isdir(subfolder_path):
            bulb_file = os.path.join(subfolder_path, 'frameAnnotationsBULB.csv')

            if os.path.exists(bulb_file):
                df = pd.read_csv(bulb_file, sep=';')
                df['clip_folder'] = subfolder
                df['Filename'] = df['Filename'].apply(lambda x: os.path.basename(x))
                all_annotations.append(df)

    full_annotations = pd.concat(all_annotations, ignore_index=True)
    return full_annotations

def visualize_annotations(base_image_path, annotations):
    for index, row in annotations.iterrows():

        image_path = os.path.join(base_image_path, row['clip_folder'], row['Filename'])

        if not os.path.exists(image_path):
            print(f"Warning: File {image_path} not found.")
            continue

        image = cv2.imread(image_path)
        if image is None:
            print(f"Warning: Unable to read {image_path}.")
            continue

        x1, y1 = int(row['Upper left corner X']), int(row['Upper left corner Y'])
        x2, y2 = int(row['Lower right corner X']), int(row['Lower right corner Y'])
        cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(image, row['Annotation tag'], (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
        cv2.imshow('Image', image)
        if cv2.waitKey(0) & 0xFF == ord('q'):
            print("Exit key pressed. Closing window.")
            break

    cv2.destroyAllWindows()

annotations = load_daytrain_annotations()
filtered_annotations = annotations[annotations['clip_folder'] == 'dayClip5']
visualize_annotations(images_base_path, filtered_annotations)
print(f"Loaded {len(annotations)} annotations from dayTrain clips.")

visualize_annotations(images_base_path, filtered_annotations)
```

Fig. 3.3. Code for loading and visualization

In this part, we aimed to visualize the annotations from the traffic light dataset step-by-step. Here's how we structured it:

Loading Annotations:

We created a function load_daytrain_annotations () that reads the annotation files (frameAnnotationsBULB.csv) for each video clip (dayClip1, dayClip2, ..., dayClip13) from the folder C:/TrafficLight/Annotations/dayTrain.

For each clip, we read the CSV file and added the corresponding subfolder name (dayClipX) as a new column clip_folder to know which folder the image belongs to.

We also removed unnecessary paths from the Filename column to keep only the image names (like dayClip1--00000.jpg).

Filtering Specific Clip:

Once all the annotations are combined into a single dataset (full_annotations), we filtered them to focus on one specific clip, dayClip7, to check the visualization for that part only. This helped us focus on a smaller set of images instead of visualizing all clips at once.

Visualizing Annotations:

In visualize_annotations(), we used OpenCV (cv2) to: Load the image files from C:/TrafficLight/dayTrain.

Draw bounding boxes using the coordinates from the annotation files (Upper left corner X, Y, Lower right corner X, Y) and display the type of annotation (like "Red Light", "Green Light") on the image.

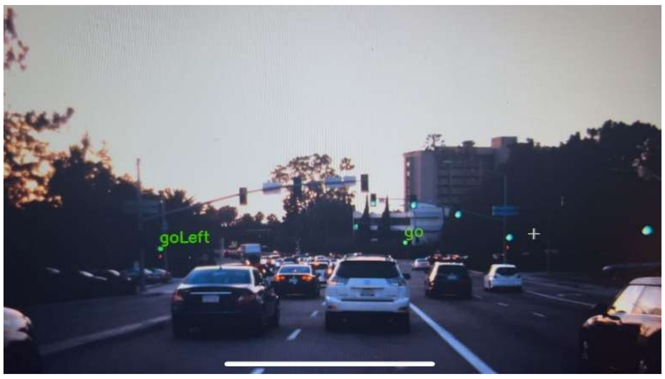


Fig.3.4. Output picture from dayClip5

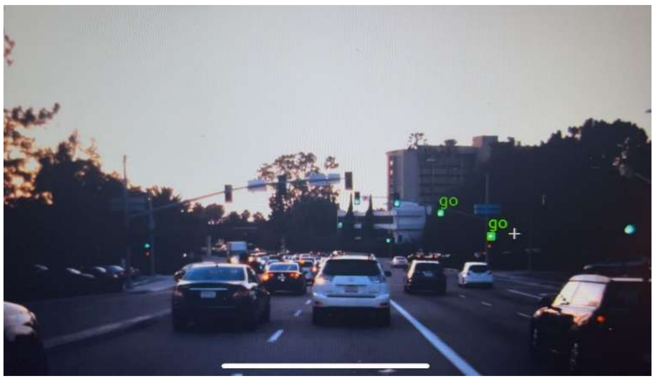


Fig .3.5. Output picture from dayClip5

Why we added the visualization:

We added the visualization step after loading the annotations to confirm that our data and bounding boxes are correct.
This step helps us:

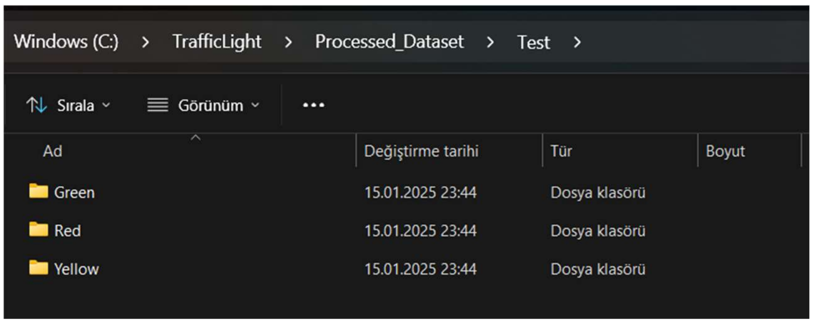
- Check if the annotations match the correct positions in the images.
- Confirm that the image paths and filenames are being read properly.
- Ensure the annotation tags ("Red Light", "Green Light", etc.) are displayed clearly.
- By visualizing a smaller set of images first (dayClip5), we can catch any mistakes before moving to the full dataset.

2) Preprocessing

In this step we aimed to extract and organize traffic light images into labeled folders based on their annotations. Here is what we did:

```python
import os
import cv2
import pandas as pd
from tqdm import tqdm

annotations_base_path = 'C:/TrafficLight/Annotations/dayTrain'
images_base_path = 'C:/TrafficLight/dayTrain'
processed_dataset_path = 'C:/TrafficLight/Processed_Dataset'

label_to_color = {
    'go': 'Green',
    'stop': 'Red',
    'warning': 'Yellow'
}

def extract_and_save_images():
    for clip_folder in os.listdir(annotations_base_path):
        annotations_file = os.path.join(annotations_base_path, clip_folder, 'frameAnnotationsBULB.csv')
        if not os.path.exists(annotations_file):
            print(f"Skipping {clip_folder}, no annotation file found.")
            continue

        annotations = pd.read_csv(annotations_file, sep=';')

        for index, row in tqdm(annotations.iterrows(), total=len(annotations), desc=f"Processing {clip_folder}"):
            label = row['Annotation tag'].strip().lower()
            if label not in label_to_color:
                continue  # Skip if it's not a recognized label

            color_label = label_to_color[label]
            image_path = os.path.join(images_base_path, clip_folder, os.path.basename(row['Filename']))

            if not os.path.exists(image_path):
                print(f"Image not found: {image_path}")
                continue
            image = cv2.imread(image_path)
            if image is None:
                print(f"Failed to read image: {image_path}")
                continue

            dest_folder = os.path.join(processed_dataset_path, 'Train', color_label)
            os.makedirs(dest_folder, exist_ok=True)

            dest_path = os.path.join(dest_folder, os.path.basename(image_path))
            cv2.imwrite(dest_path, image)

    print("Traffic light dataset extraction complete")
extract_and_save_images()
```

Fig. 3.6. Code for color separation in train and test folders



```
Ad        Değiştirme tarihi   Tür            Boyut
Green     15.01.2025 23:44    Dosya klasörü
Red       15.01.2025 23:44    Dosya klasörü
Yellow    15.01.2025 23:44    Dosya klasörü
```

Fig. 3.7. Output

Explanation to the code:

a) Define Paths and Labels:

We specified paths for the annotations, images, and the processed dataset.
A dictionary label_to_color was created to map annotation tags ('go', 'stop', 'warning') to their respective traffic light colors (Green, Red, Yellow).

b) Processing Annotations:

We looped through the annotation files in the C:/TrafficLight/Annotations/dayTrain directory, skipping any folder without the required CSV file (frameAnnotationsBULB.csv).

c) Filtering Relevant Data:

For each annotation, we checked if the Annotation tag matched one of the predefined labels ('go', 'stop', 'warning'). If it didn't, we skipped that row.

d) Locate and Save Images:

We used the annotation to find the corresponding image file in the C:/TrafficLight/dayTrain directory.
If the image existed and was readable, we saved it to the correct folder in C:/TrafficLight/Processed_Dataset/Train based on its label (Green, Red, or Yellow).

e) Organizing the Processed Dataset:

For each traffic light category, we ensured the destination folder existed before saving the image.

Why we included this step:
We added this process to organize our dataset into labeled categories (Green, Red, Yellow) for training our traffic light detection model. By grouping images based on their annotations, we ensured that the data was ready for the next steps, such as training or testing.
This step also functioned as a quality check, ensuring we only included relevant and correctly labeled images in our dataset.

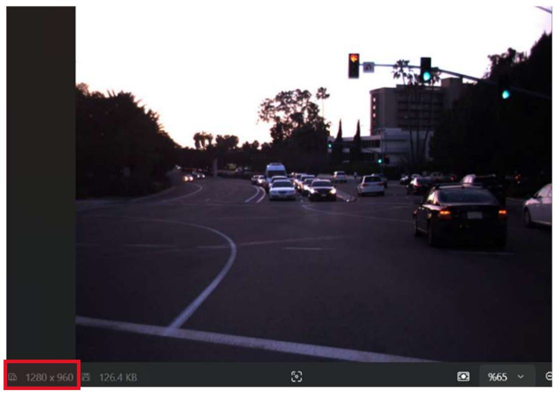
Splitting the data:

In this part, we split the processed dataset into **80% for training and 20% for testing**. The training set is used to teach the model to recognize traffic light states, while the test set checks how well the model performs on new, unseen images. This step helps us evaluate the model's ability to generalize and ensures it doesn't just memorize the training data.

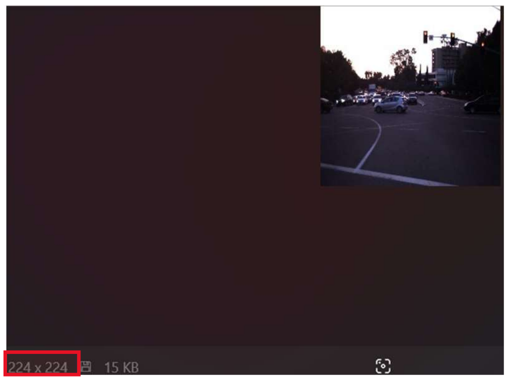
```python
import os
import shutil
import random

def split_train_test(base_path, split_ratio=0.2):
    for color in ['Red', 'Yellow', 'Green']:
        source_folder = os.path.join(base_path, 'Train', color)
        dest_folder = os.path.join(base_path, 'Test', color)
        os.makedirs(dest_folder, exist_ok=True)

        # List all images in the source folder
        all_images = os.listdir(source_folder)
        num_test_images = int(len(all_images) * split_ratio)

        # Randomly select images for the test set
        test_images = random.sample(all_images, num_test_images)

        for img_name in test_images:
            src_path = os.path.join(source_folder, img_name)
            dest_path = os.path.join(dest_folder, img_name)
            shutil.move(src_path, dest_path)

    print("Train-Test split completed!")

# Run the split (80% training, 20% testing)
processed_dataset_path = 'C:/TrafficLight/Processed_Dataset'
split_train_test(processed_dataset_path)
```

Resizing the images:

In this code, we resized all the images in the "Train" and "Test" folders to a uniform size of 224x224 pixels. This ensures consistency in image dimensions, which is important for the model during training and testing. By resizing, we prepare the images to match the input size required by the model.



Original



Resized version

3) Traffic Light Classification

We built a Convolutional Neural Network (CNN) to classify traffic lights into three classes: Red, Yellow, and Green. The goal was to train the model using images and evaluate its accuracy.

a) Data Preparation

We used “ImageDataGenerator” to preprocess images. For training, we added data augmentation (e.g., rotation, zoom) to make the model robust. For testing, we only normalized pixel values to improve consistency.

b) Model Design

Layers:

- Conv2D and MaxPooling2D for extracting features like shapes and colors.
- Flatten to convert 2D features into a vector.
- Dense for final classification with a softmax activation for probabilities

c) Compilation

- Optimizer: adam for efficient learning.
- Loss Function: categorical_crossentropy for multi-class classification.
- Metric: accuracy to measure performance.

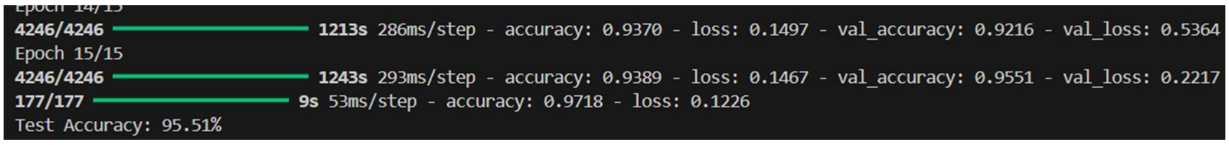
d) Checkpoints

- We used ModelCheckpoint to save the best version of the model based on validation loss.

e) Training and Evaluation

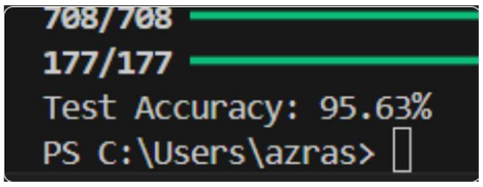
- First we trained for 10 epochs and evaluated accuracy on the test set.Later we trained it again with 15 epochs which took us 7 hours approximately to finish it.

First Training



```
Epoch 14/15
4246/4246 ━━━━━━━━━━━━━━━━━━━━ 1213s 286ms/step - accuracy: 0.9370 - loss: 0.1497 - val_accuracy: 0.9216 - val_loss: 0.5364
Epoch 15/15
4246/4246 ━━━━━━━━━━━━━━━━━━━━ 1243s 293ms/step - accuracy: 0.9389 - loss: 0.1467 - val_accuracy: 0.9551 - val_loss: 0.2217
177/177 ━━━━━━━━━━━━━━━━━━━━ 9s 53ms/step - accuracy: 0.9718 - loss: 0.1226
Test Accuracy: 95.51%
```

Second Training



```
708/708 ━━━━━━━━━━━━━━━━━━━━
177/177 ━━━━━━━━━━━━━━━━━━━━
Test Accuracy: 95.63%
PS C:\Users\azras> ▯
```

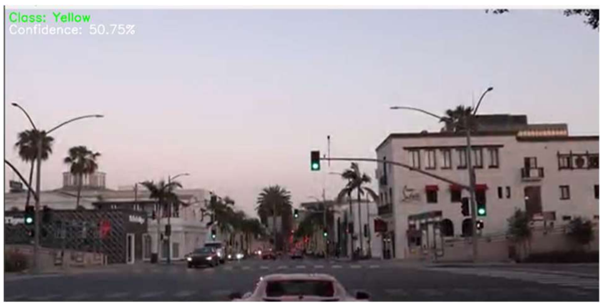
Luckly we were able to train our model at accuracy 95.63%. Way better than we had on traffic sign model. (But having 15 epoch)

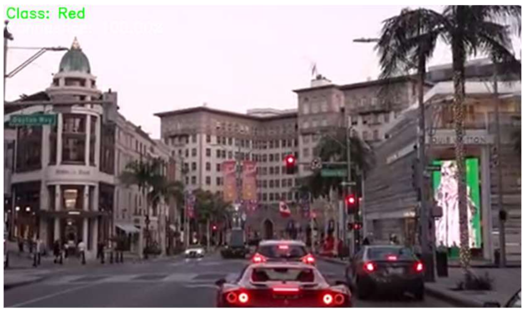
```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint


train_dir = 'C:/TrafficLight/Processed_Dataset/Train'
test_dir = 'C:/TrafficLight/Processed_Dataset/Test'

img_height, img_width = 224, 224
batch_size = 15

train_datagen = ImageDataGenerator(
    rescale=1.0 / 255.0,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=False
)
test_datagen = ImageDataGenerator(rescale=1.0 / 255.0)

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

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
model = Sequential([
    Input(shape=(img_height, img_width, 3)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(3, activation='softmax')
])
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

checkpoint = ModelCheckpoint(
    'best_traffic_light_model.keras',
    save_best_only=True,
    monitor='val_loss',
    mode='min'
)
epochs = 10
history = model.fit(
    train_generator,
    validation_data=test_generator,
    epochs=epochs,
    callbacks=[checkpoint]
)
test_loss, test_accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

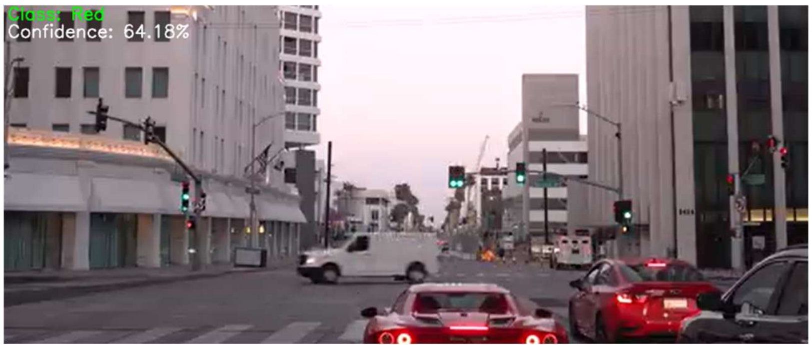
4) Testing the Trained Model

So, finally, this step was facing the truth. After all the training, trying to find the most suitable image scale, etc., we put our trained model in a test run. We tried it on both images and video frames.

- **First outputs after the 1st training (10 epoch):**



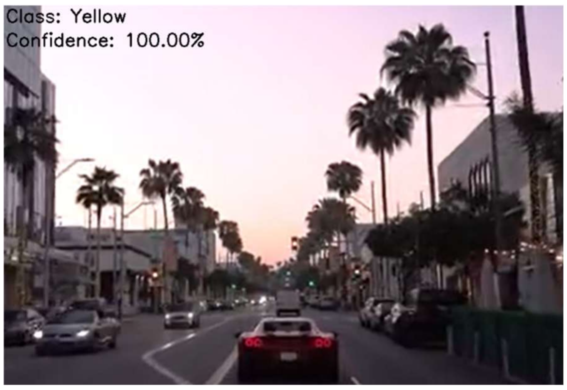


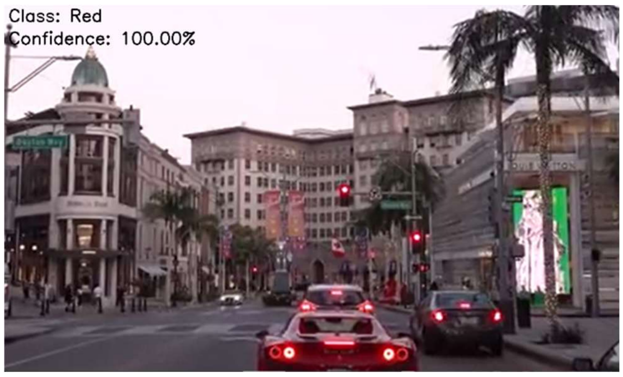


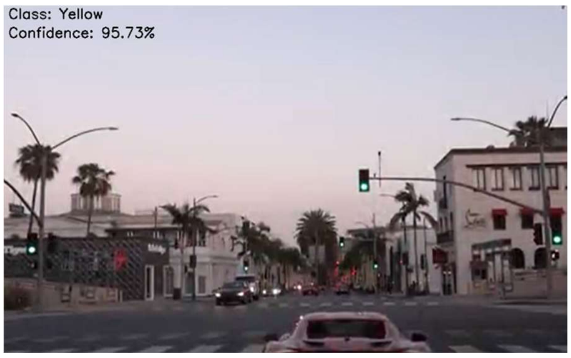
We observed that the red-light predictions were highly accurate with 100% confidence. However, predictions for yellow and green lights were weaker, likely due to limited data or insufficient training epochs.

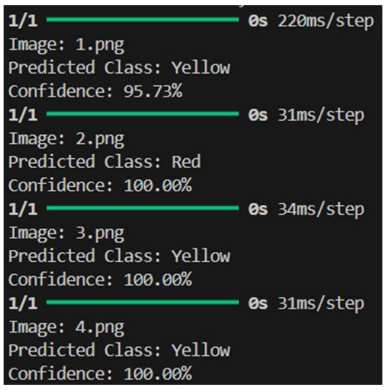
- **Second outputs after the 2nd training (15 epoch):**

After training with 15 epochs, the yellow-light predictions improved significantly, reaching 100% confidence. This was a big improvement from the earlier 50% confidence or incorrect predictions. Unfortunately, green-light predictions did not improve much, as we ran out of time to focus on them.









```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 220ms/step
Image: 1.png
Predicted Class: Yellow
Confidence: 95.73%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
Image: 2.png
Predicted Class: Red
Confidence: 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
Image: 3.png
Predicted Class: Yellow
Confidence: 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
Image: 4.png
Predicted Class: Yellow
Confidence: 100.00%
```

<u>What We Changed:</u>

To improve the model, we increased the training epochs from 10 to 15 for better learning. We also applied more data augmentation, including rotation, brightness variation, and zoom, to make the model more robust. Additionally, we calculated and applied class weights to handle the imbalance in training data, ensuring better attention to underrepresented classes like yellow and green lights.

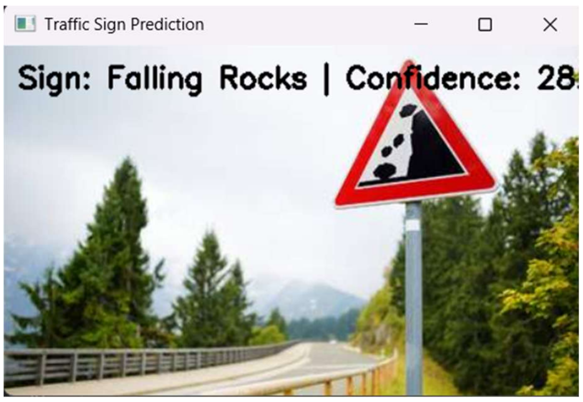
Summary for the Traffic Light Recognition:

In this project, we built a traffic light recognition model using CNN. The process started with preparing the dataset, including augmenting images to improve model generalization. We faced challenges with yellow and green light recognition, so we increased training epochs, used data augmentation, and applied class weights. These changes significantly improved accuracy for all classes.

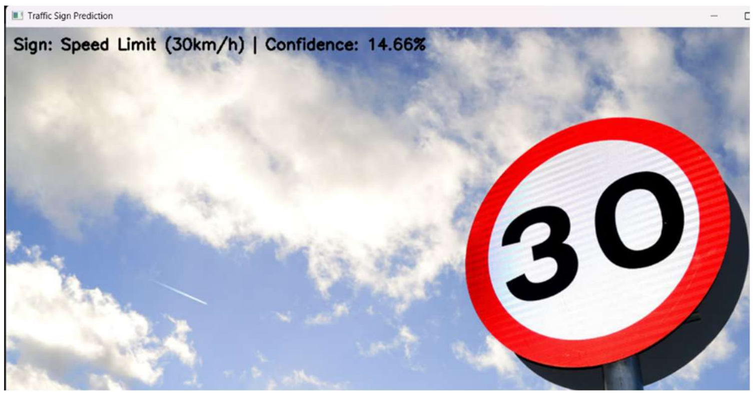
This project taught us how to handle unbalanced datasets, apply advanced training techniques, and analyze model performance. It was a challenging yet rewarding experience that improved our understanding of machine learning workflows and strengthened our problem-solving skills in AI.

3.2 Traffic Sign Recognition Further Adjustments

Testing the trained model:

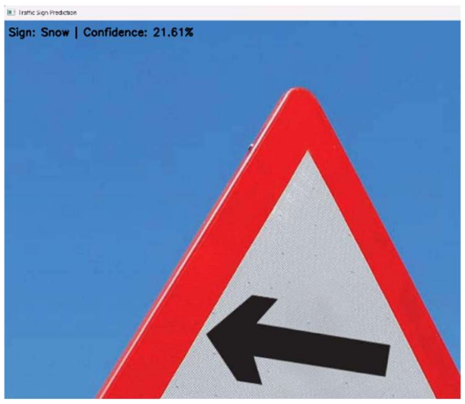


Sign: Falling Rocks | Confidence: 28

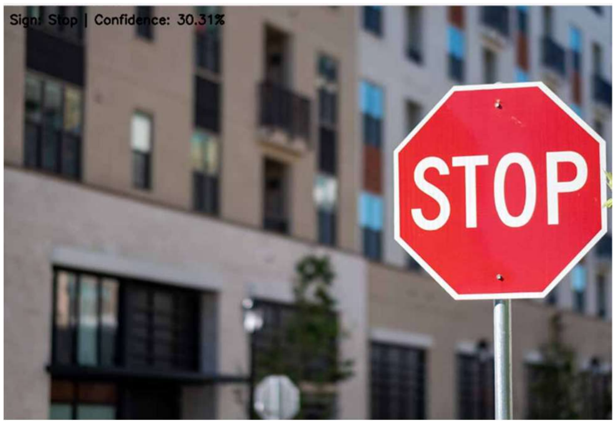


Sign: Speed Limit (30km/h) | Confidence: 14.66%

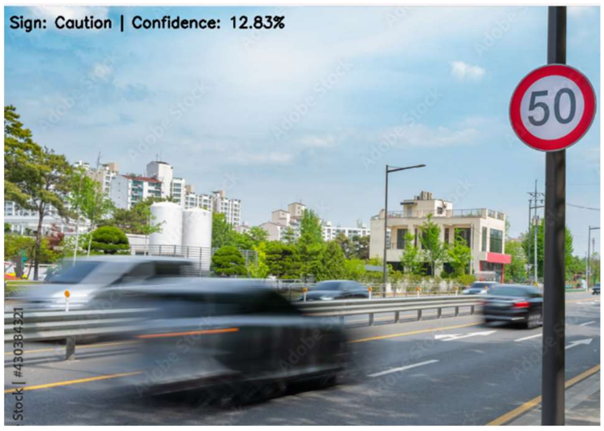
In the final part of the project, we tested our traffic sign recognition model on individual images. The model was able to classify traffic signs and display their names along with the confidence percentages on the images. We mapped the class indices to readable sign names for better understanding and used OpenCV to display the predictions. However, the model's confidence for some predictions was low, highlighting areas for improvement.



Sign: Snow | Confidence: 21.61%



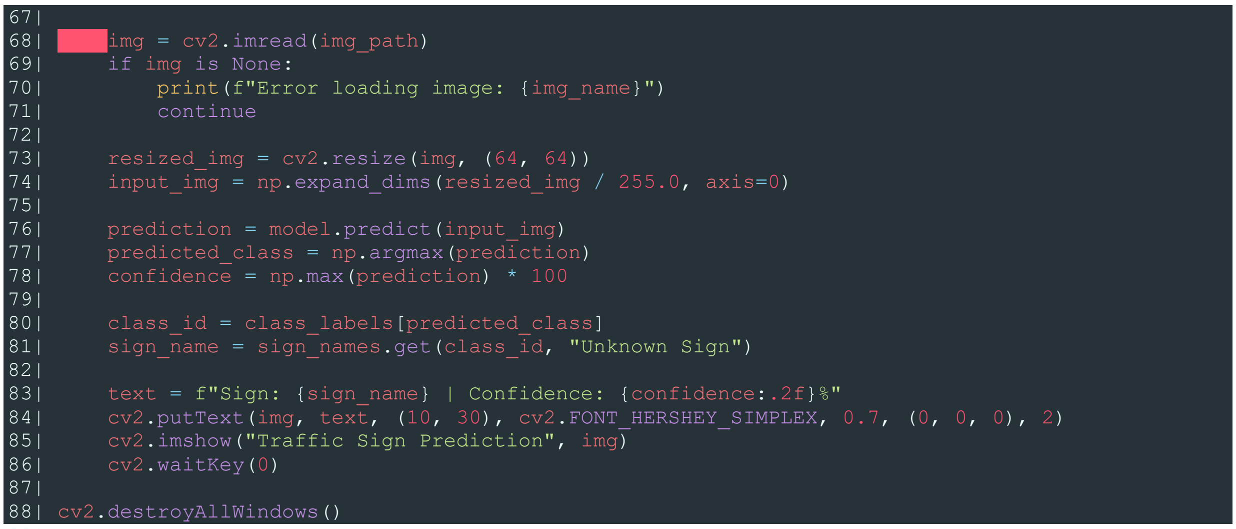
Sign: Stop | Confidence: 30.31%



Sign: Caution | Confidence: 12.83%

Training model testing code:

```python
import tensorflow as tf
import numpy as np
import cv2
import os
import json

model = tf.keras.models.load_model('traffic_sign_classifier.h5')
print("Model loaded successfully.")

with open('class_indices.json', 'r') as f:
    class_indices = json.load(f)

class_labels = {v: k for k, v in class_indices.items()}

test_images_dir = 'C:/GTSRB/testing_images'

sign_names = {
    "00000": "Speed Limit (20km/h)",
    "00001": "Speed Limit (30km/h)",
    "00002": "Speed Limit (50km/h)",
    "00003": "Speed Limit (60km/h)",
    "00004": "Speed Limit (70km/h)",
    "00005": "Speed Limit (80km/h)",
    "00006": "Speed Limit (90km/h)",
    "00007": "Speed Limit (100km/h)",
    "00008": "Speed Limit (120km/h)",
    "00009": "Speed Limit (50km/h)",
    "00010": "Speed Limit (50km/h)",
    "00011": "Speed Limit (50km/h)",
    "00012": "Speed Limit (50km/h)",
    "00013": "Falling Rocks",
    "00014": "Stop",
    "00015": "No parking",
    "00016": "Speed Limit (50km/h)",
    "00017": "Speed Limit (50km/h)",
    "00018": "Caution",
    "00019": "Right hand curve ",
    "00020": "Left hand curve",
    "00021": "Right reverse bend",
    "00022": "Hump or rough road",
    "00023": "Slippery Road",
    "00024": "Lane ends",
    "00025": "Road work ahead",
    "00026": "Traffic signals ahead",
    "00027": "Pedestrian crossing ahead",
    "00028": "Children",
    "00029": "Bicycle",
    "00030": "Snow",
    "00031": "Deer",
    "00032": "End of all bans",
    "00033": "Turn right",
    "00034": "Turn left",
    "00035": "Go straight",
    "00036": "Straight and right turn",
    "00037": "Straight and left turn",
    "00038": "Children",
    "00039": "Children",
    "00040": "Circular arrow",
    "00041": "Circular arrow",
    "00042": "Circular arrow",

}

for img_name in os.listdir(test_images_dir):
    img_path = os.path.join(test_images_dir, img_name)

```



```

    img = cv2.imread(img_path)
    if img is None:
        print(f"Error loading image: {img_name}")
        continue

    resized_img = cv2.resize(img, (64, 64))
    input_img = np.expand_dims(resized_img / 255.0, axis=0)

    prediction = model.predict(input_img)
    predicted_class = np.argmax(prediction)
    confidence = np.max(prediction) * 100

    class_id = class_labels[predicted_class]
    sign_name = sign_names.get(class_id, "Unknown Sign")

    text = f"Sign: {sign_name} | Confidence: {confidence:.2f}%"
    cv2.putText(img, text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
    cv2.imshow("Traffic Sign Prediction", img)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

To improve, we can:

- Collect more diverse and high-quality data for training to enhance robustness.
- Increase training epochs or fine-tune hyperparameters for better accuracy.
- Use advanced techniques like transfer learning with pre-trained models.
- Apply better image preprocessing techniques to reduce noise and improve feature extraction.
- Optimize the architecture by adding more layers or experimenting with different loss functions.

CONCLUSION

This project was an ambitious attempt to build a traffic sign and light recognition system for autonomous vehicles using AI and computer vision. Throughout the process, we encountered numerous challenges, particularly in the AI component. Dataset preparation was time-intensive, requiring extensive organization, resizing, and augmentation. Balancing class distributions and dealing with low confidence predictions for certain classes, like yellow lights and complex traffic signs, proved especially difficult.

Despite these issues, we successfully implemented both traffic sign and light classifiers. While the traffic light model achieved high accuracy after additional training and adjustments, the traffic sign model highlighted areas for improvement, such as overfitting and limited generalization. Our efforts taught us valuable lessons in dataset handling, CNN architecture optimization, and iterative testing.

Moving forward, we recommend collecting more diverse data, experimenting with transfer learning, and refining the model with advanced preprocessing techniques. This project strengthened our problem-solving skills and deepened our understanding of the complexities in developing AI-driven solutions for real-world applications.