Traffic Lights detection using YOLO8

Prepared By/

Ahmed Darwesh
Heba Mohsen
Dalia Elsayed
Faten Ahmad
Mohamed Abdel Naser

1. Introduction

In this project, we aim to develop a robust traffic light detection and classification system using the state-of-the-art YOLOv8 object detection algorithm and deploy it into a web application to analyze input video frames. The primary objective of this system is to accurately detect and classify traffic lights into four distinct classes: red, green, and yellow. Additionally, the system will include an “off" class to handle instances where a traffic light is not detected in the input video frame.

The system can provide valuable information to assist in decision-making processes, improve traffic flow, and enhance road safety. The use of YOLOv8, a powerful object detection model, enables the system to detect with high accuracy and performance suitable with the project use case.

The project's key goals include:

1- Developing a deep learning-based traffic light detection and classification system using YOLOv8
2- Training the model to accurately detect traffic lights into four classes: red, green, yellow and off .
3- Evaluating the system's performance on a test dataset and calculating KPI metrics like recall, precision mAP and mAP50-95.
4- Optimizing the system's efficiency and deployment feasibility for integration into a web application that analyzes uploaded videos using onnx.

2. Methodology

2.1 Data Collection

The dataset used in this project is the Traffic Light Detection Dataset from Roboflow. This dataset contains a variety of images of traffic lights under different conditions, including varying light, weather, and angles. The dataset was downloaded from Roboflow and subsequently uploaded to Dropbox for easy access and management throughout the project.

2.2 Data Preprocessing

Before training the model, the dataset underwent several preprocessing steps:

- **Annotation Conversion:** The annotations provided in the dataset were converted to the format required by YOLOv8.
- **Data Augmentation:** To increase the robustness of the model, various data augmentation techniques such as horizontal flipping, rotation, and scaling were applied to the images.
- **Normalization:** The pixel values of the images were normalized to a range of [0, 1] to facilitate better convergence during training.

2.3 Model Training

- **Model Configuration**: Setting up the YOLOv8 model, which includes defining the architecture, setting hyperparameters such as learning rate, batch size, and number of epochs, and specifying the input image size.

- **Training Process**: The model was trained on the preprocessed dataset using a high-performance GPU. The training involved optimizing the model weights to minimize the loss function, which measures the discrepancy between the predicted and actual bounding boxes of traffic lights. The Adam optimizer was used to update the model weights iteratively.

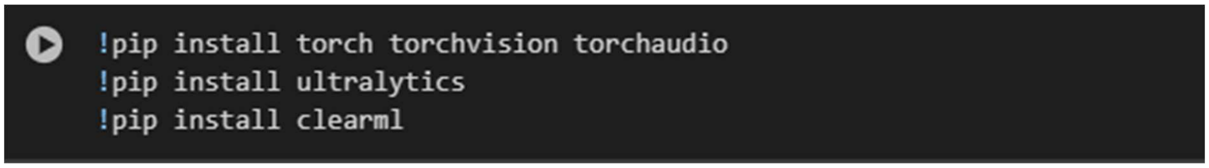
- **Validation**: The model's performance was validated on a separate validation set to ensure it generalizes well to unseen data. Metrics such as precision, recall, and mean Average Precision (mAP) were used to evaluate the model's accuracy.

3. Implementation

3.1 Environment Setup

3.1.1 Install Dependencies

First, we need to install the necessary libraries. Use the following commands to install PyTorch, Ultralytics, and ClearML.



```
!pip install torch torchvision torchaudio
!pip install ultralytics
!pip install clearml
```

1. **torch, torchvision, torchaudio**: These are PyTorch and related libraries for deep learning, computer vision, and audio processing.

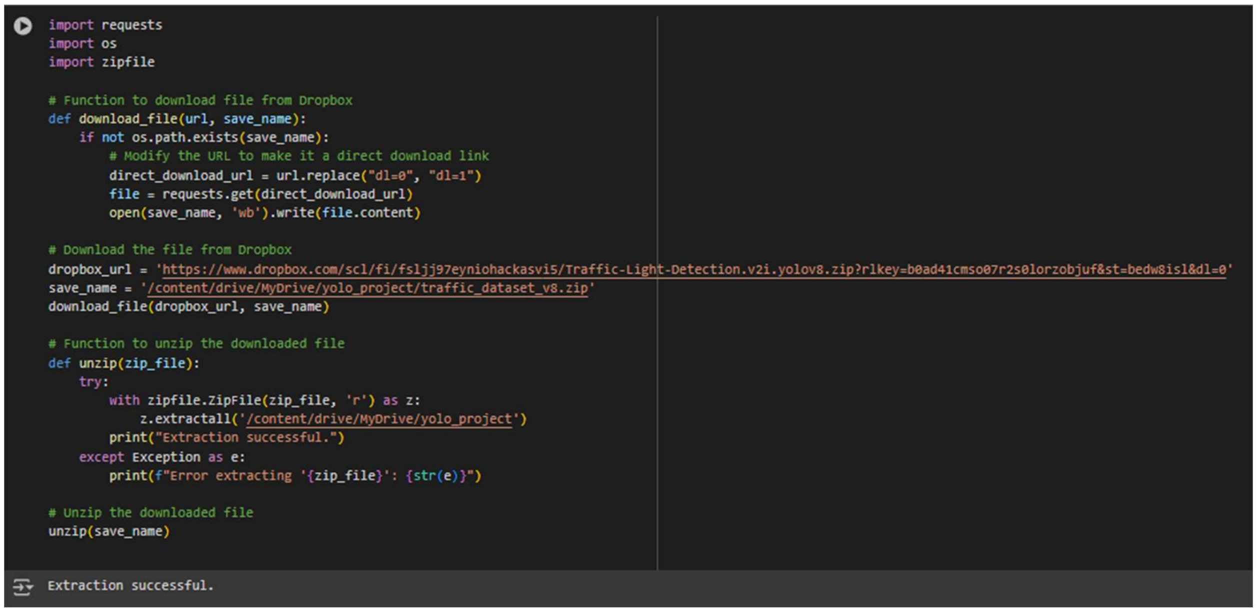
2. **ultralytics**: This is a package for working with YOLO (You Only Look Once) models for object detection.

3. **clearml:** This is a package for machine learning experiment management and MLOps.

3.2. Download and Prepare Dataset

3.2.1 Download Dataset

The following script downloads a dataset from Dropbox and saves it locally.



```
import requests
import os
import zipfile

# Function to download file from Dropbox
def download_file(url, save_name):
    if not os.path.exists(save_name):
        # Modify the URL to make it a direct download link
        direct_download_url = url.replace("dl=0", "dl=1")
        file = requests.get(direct_download_url)
        open(save_name, 'wb').write(file.content)

# Download the file from Dropbox
dropbox_url = 'https://www.dropbox.com/scl/fi/fsljj97eyniohackasvi5/Traffic-Light-Detection.v2i.yolov8.zip?rlkey=b0ad41cmso07r2s0lorzobjuf&st=bedw8isl&dl=0'
save_name = '/content/drive/MyDrive/yolo_project/traffic_dataset_v8.zip'
download_file(dropbox_url, save_name)

# Function to unzip the downloaded file
def unzip(zip_file):
    try:
        with zipfile.ZipFile(zip_file, 'r') as z:
            z.extractall('/content/drive/MyDrive/yolo_project')
        print("Extraction successful.")
    except Exception as e:
        print(f"Error extracting '{zip_file}': {str(e)}")

# Unzip the downloaded file
unzip(save_name)
```

```
Extraction successful.
```

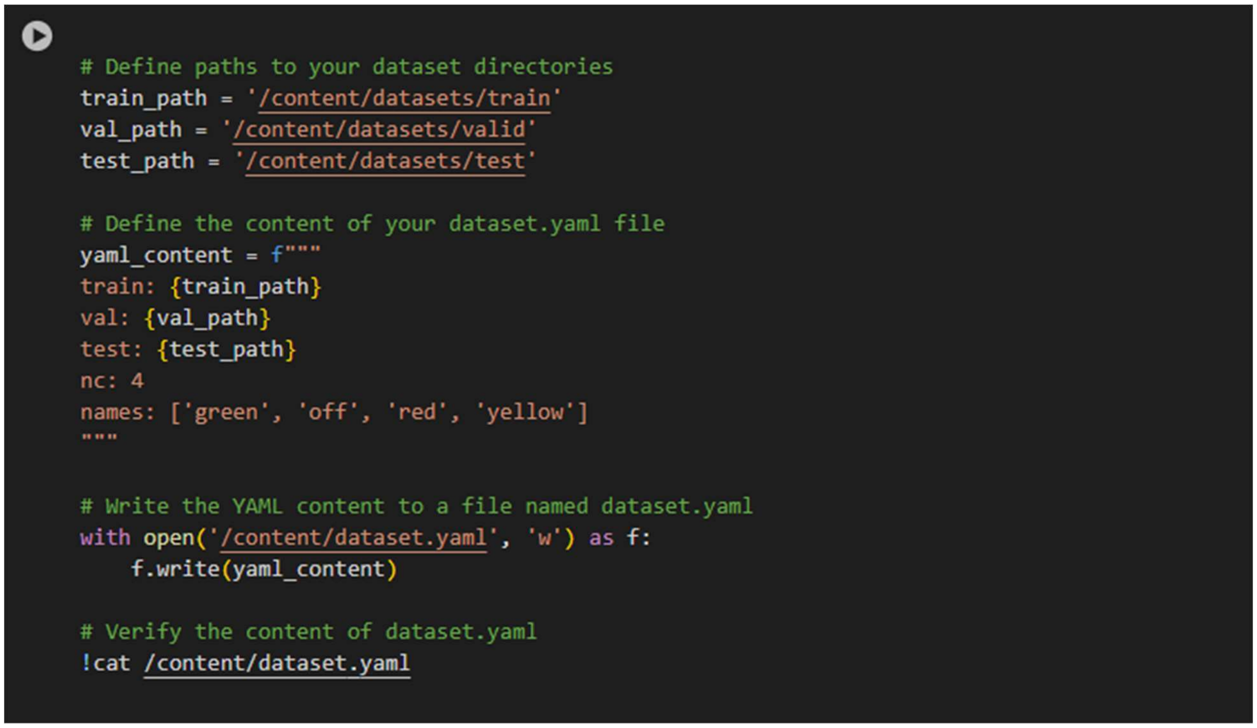
- **Imports**: The script imports the `requests`, `os`, and `zipfile` libraries.
- **Function to Download File**: The `download_file` function takes a URL and a save name, modifies the URL for direct download, retrieves the file content, and saves it locally.
- **Download File from Dropbox**: It uses the `download_file` function to download a ZIP file from a Dropbox link to a specified path.
- **Function to Unzip File**: The `unzip` function extracts the contents of the ZIP file to a specified directory.
- **Unzip the Downloaded File**: The script calls the `unzip` function to extract the downloaded ZIP file.

The final message "Extraction successful" indicates that the file extraction was completed without errors.

3.3 Prepare Data for Training

After downloading and extracting the dataset, ensure the data is organized in a suitable format for YOLOv8 training. The dataset should be structured with images and corresponding labels in separate directories.

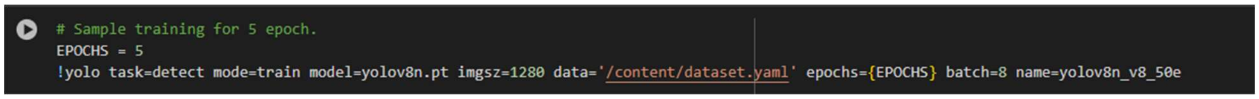
For training, we need the dataset YAML to define the paths to the images and the class names. this YAML file should be in the project root directory. We will name this file dataset.yaml .



```
# Define paths to your dataset directories
train_path = '/content/datasets/train'
val_path = '/content/datasets/valid'
test_path = '/content/datasets/test'

# Define the content of your dataset.yaml file
yaml_content = f"""
train: {train_path}
val: {val_path}
test: {test_path}
nc: 4
names: ['green', 'off', 'red', 'yellow']
"""

# Write the YAML content to a file named dataset.yaml
with open('/content/dataset.yaml', 'w') as f:
    f.write(yaml_content)

# Verify the content of dataset.yaml
!cat /content/dataset.yaml
```

3.4 Train YOLOv8 Model

Use the YOLOv8 package to load a pretrained model and train it on your dataset. This involves specifying paths to training and validation images and labels, defining the number of classes, and providing class names. We use YOLO8 Nano Training , the smallest in the YOLOv8 family. This model has 3.2 million parameters and can run in real-time, even on a CPU.



```
# Sample training for 5 epoch.
EPOCHS = 5
!yolo task=detect mode=train model=yolov8n.pt imgsz=1280 data='/content/dataset.yaml' epochs={EPOCHS} batch=8 name=yolov8n_v8_50e
```

Here are the explanations of all the command line arguments that we are using:

- **task**: Whether we want to detect, segment, or classify on the dataset of our choice.
- **mode**: Mode can either be train, val, or predict. As we are running training, it should be train.
- **model**: The model that we want to use. Here, we use the YOLOv8 Nano model pretrained on the COCO dataset.
- **imgsz**: The image size. The default resolution is 640.
- **data**: Path to the dataset YAML file.
- **epochs**: Number of epochs we want to train for.
- **batch**: The batch size for data loader. You may increase or decrease it according to your GPU memory availability.
- **name**: Name of the results directory for runs/detect.

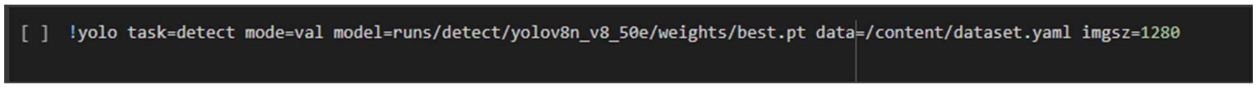
In our Training Model :

- We are training the model for 5 epochs, which will stay the same for all the models.
- As discussed earlier, we are training with a 1280 which is higher than the default 640.
- The batch size is 8.

3.5 Evaluate and Test the Model

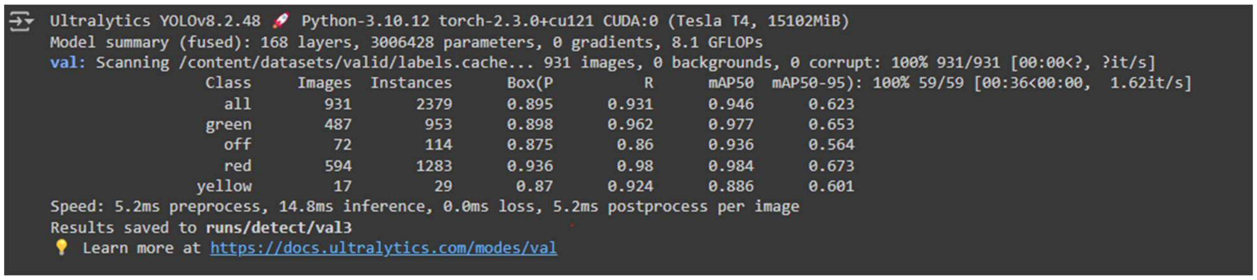
5.1 Evaluate the Model

Evaluate the trained model to check its performance on the validation set , so we evaluate the best model .



```
!yolo task=detect mode=val model=runs/detect/yolov8n_v8_50e/weights/best.pt data=/content/dataset.yaml imgsz=1280
```

And get the following result on the validation set.



```
Ultralytics YOLOv8.2.48 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 3006428 parameters, 0 gradients, 8.1 GFLOPs
val: Scanning /content/datasets/valid/labels.cache... 931 images, 0 backgrounds, 0 corrupt: 100% 931/931 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 59/59 [00:36<00:00,  1.62it/s]
                   all        931       2379      0.895      0.931      0.946      0.623
                 green        487        953      0.898      0.962      0.977      0.653
                   off         72        114      0.875       0.86      0.936      0.564
                   red        594       1283      0.936       0.98      0.984      0.673
                yellow         17         29       0.87      0.924      0.886      0.601
Speed: 5.2ms preprocess, 14.8ms inference, 0.0ms loss, 5.2ms postprocess per image
Results saved to runs/detect/val3
💡 Learn more at https://docs.ultralytics.com/modes/val
```

We can describe this results in details :

- **Precision and Recall (P, R)**: These metrics (0.895 precision, 0.931 recall for all classes) suggest that the model is correctly identifying and localizing objects with high accuracy.

- **mAP (mean Average Precision)**:

    - **mAP50**: 0.946 is quite good, indicating that on average, the model's detections are highly accurate when considering an IoU threshold of 0.5.
    - **mAP50-95**: 0.623 across the range of IoU thresholds (0.5 to 0.95) is reasonable, though higher values are generally preferred for more robust performance.

- **Class-specific Performance**:

Classes like green (mAP50: 0.977), red (mAP50: 0.984) show very high precision and recall, indicating strong performance for these categories.

Classes with fewer instances, like yellow, also show respectable performance but could potentially benefit from more training data.

- **Speed**:

Inference speed (14.8ms per image) is relatively fast, which is beneficial for real-time or large-scale deployment scenarios.

So the evaluation results demonstrate strong performance in terms of accuracy and speed for the YOLOv8.2.48 model. Precision and recall metrics indicate robust identification and localization of objects across various classes. The model achieves an overall mAP50 of 0.946, showcasing high accuracy at an IoU threshold of 0.5, with specific classes such as green and red achieving even higher mAP values of 0.977 and 0.984, respectively. However, the mAP50-95 of 0.623 suggests room for improvement across the full range of IoU thresholds. The inference speed of 14.8ms per image ensures efficient processing, suitable for real-time applications.

4. Requirements

4.1 Hardware

- **GPU:** A high-performance GPU, such as NVIDIA Tesla or RTX series, is recommended for efficiently training the YOLOv8 model.
- **CPU:** A multi-core CPU to handle data preprocessing and other computational tasks.
- **Memory:** At least 16 GB of RAM for managing large datasets and model training processes.

4.2 Software

- **Python:** The primary programming language used for developing and training the model. Version 3.7 or higher is recommended.
- **Libraries:** Essential libraries include:
    - **TensorFlow/Keras:** For building and training the deep learning model.
    - **OpenCV:** For image processing tasks such as reading, preprocessing, and augmenting images.
    - **NumPy:** For numerical operations and handling arrays.
    - **Matplotlib/Seaborn:** For visualizing data and model performance metrics.
    - **Pandas:** For data manipulation and analysis.
- **Roboflow:** For accessing and managing the Traffic Light Detection Dataset.
- **Dropbox:** For storing and accessing the dataset conveniently throughout the project.
- **Jupyter Notebook:** For interactive development, experimentation, and documentation of the project workflow.

5. Conclusion

In this project, we have successfully developed a robust traffic light detection and classification system using the YOLOv8 object detection algorithm. The system is capable of accurately detecting and classifying traffic lights into four distinct classes: red, green, and yellow, as well as an “off" class.

The model achieves an overall mAP50 of 0.946, showcasing high accuracy at an IoU threshold of 0.5.

However, the Model needs more improvement to improve the results mAP50-95 of 0.623 much more by increasing the training dataset samples that support color yellow class and off class so the data set shall be more balanced, and this shall be a future work.

Model Conversion and Deployment:

To facilitate the deployment and integration of the traffic light detection system into real-world applications, we have explored the use of the Open Neural Network Exchange (ONNX) format.

The process of deploying the traffic light detection system using ONNX involves the following steps:

1- ONNX Model Conversion:

After training the YOLOv8 model, we exported the model in the ONNX format. This conversion process ensures that the model can be easily integrated into a wide range of deployment environments, including web applications.

2- ONNX Runtime Integration:

To leverage the ONNX model, we have integrated the ONNX Runtime into the traffic light detection web application that support uploading a video and analyze it into frames to detect a traffic light and their classification.

6. References

- **YOLOv8 Documentation:** Detailed documentation on the YOLOv8 model and its implementation. YOLOv8 Documentation

- **Roboflow Dataset:** The Traffic Light Detection Dataset used in this project. Roboflow Traffic Light Detection Dataset

- **TensorFlow Documentation:** Comprehensive documentation on TensorFlow, the deep learning framework used. TensorFlow

- **OpenCV Documentation:** Official documentation for OpenCV, used for image processing tasks. OpenCV

- **Keras Documentation:** Documentation for Keras, the high-level neural networks API used with TensorFlow. Keras