IBM Innovation Center for Education

PHEMESOFT Software That Matters
TRUSTED DELIVERY PARTNER for IBM ICE

UPES

University Of Petroleum and Energy Studies

Final Project Report

on

Object Detection for Autonomous Vehicles

Made By:

Devashish Uniyal

Table of Contents

1.Background

1.1 Aim

The goal of this project is to create an efficient and accurate object detection system for autonomous cars based on the YOLO (You Only Look Once) approach. This system is designed to identify and classify numerous objects in real time, hence improving the safety and dependability of self-driving vehicles. The research focusses on using advanced deep learning techniques to interpret and analyse images acquired by the vehicle's cameras, which will provide essential information for navigation and obstacle avoidance. Additionally, a web interface is created to facilitate user interaction and visualization of the detection results.

1.2 Technologies.

To achieve its objectives, the initiative makes use of a variety of cutting-edge technology.

1.Deep Learning Frameworks: The object identification system is built upon the YOLO model, which was implemented using the Ultralytics YOLO package.
2.Computer Vision: OpenCV is used for image processing tasks such as reading images, drawing bounding boxes, and displaying findings.
online

3.Development: HTML, CSS, and JavaScript are used to construct a responsive and interactive online interface that displays detection results and allows users to engage.
4.Backend Framework: Flask, a lightweight Python web framework, is used to handle web requests and deliver the website. Python is the major programming language for model development, training scripts, and backend functionality. Frontend development involves the usage of HTML, CSS, and JavaScript.
5.Data Handling: YAML configuration files using python to create labels for training.

1.3 Hardware Architecture.

The hardware architecture for this project includes the following:

High-Performance Computing: A powerful GPU-enabled machine is used to train the YOLO model, enabling for faster processing and management of big datasets.
Camera Systems: The autonomous vehicle is equipped with high-resolution cameras that collect real-time images, which are subsequently processed by an object identification system.
Edge Computing Devices: In a deployment scenario, edge computing devices with adequate computational power can execute the learnt model for real-time object detection on the vehicle.
Web Server: A server is configured to host the Flask application, allowing users to access the web interface remotely.

1.4 Software Architecture.

The project's software architecture is intended to be modular, scalable, and efficient:

Model Training Pipeline: This pipeline involves data preprocessing, model initialisation, training, and evaluation. The model is trained to detect objects using a large dataset of labelled photos.
Detection Pipeline: This pipeline consists of loading the trained model, reading input photos, conducting the detection procedure, and outputting the results. The detection method involves drawing bounding boxes and labels around the discovered objects.
Logging and Monitoring: The system uses logging techniques to monitor the training and detection processes, offering insights and debugging information.
User Interface: A graphical user interface designed with OpenCV displays detection results, allowing users to see the object detection system's performance in real time.

User Interface: A graphical user interface, built using OpenCV, displays the detection results, allowing users to visualize the performance of the object detection system in real-time.

Web Interface: The web interface is developed using HTML, CSS, and JavaScript to provide a user-friendly platform for viewing detection results. Flask handles the backend operations, serving the web pages and managing user requests. This setup allows users to upload images, initiate object detection, and view results directly through their web browser.

2. System

2.1 Requirements

2.1.1 Functional requirements.

Object detection requires the system to effectively detect and classify items in real time using the YOLO paradigm.
Model Training: The system must support the development of the YOLO model on labelled datasets.
Image Processing: Before putting photos into the model, the system must resize and normalise them.
Result Visualisation: The system must give visual feedback by putting bounding boxes and labels around discovered objects in images.
Web Interface: The system must provide a web interface where users can upload photographs, initiate object detection, and view the results.
Logging: The system must record training and detection operations for monitoring and debugging.

2.1.2 User Requirements

Ease of Use: The web interface should be simple and straightforward, allowing people with little technical understanding to interact with the system.
Real-Time Detection: For autonomous vehicle scenarios, the system must detect objects in real-time or near-real-time.
Accessible Web Interface: Users should be able to use the web interface from a variety of devices, including PCs, laptops, and smartphones.
Feedback Mechanism: The system must offer clear and timely feedback on detection findings, such as confidence levels and object labels.

2.1.3 Environmental Requirements

Hardware requirements: The system must be deployable on hardware with adequate computational power, such as GPU-enabled computers for training and edge devices for real-time detection.
Software Compatibility: The system must work with the most recent versions of the required software libraries, such as OpenCV, Flask, and the Ultralytics YOLO library.
Network Requirements: The web interface must be accessible via a regular internet connection, and the backend server must manage several concurrent users efficiently.

2.2 Design & Architecture

The system is modularly structured, with discrete modules for model training, detection, image processing, and a web interface.
Backend Architecture: The Flask-based backend processes user requests, manages picture uploads, runs the detection model, and serves the results.
Frontend Architecture: The frontend is built with HTML, CSS, and JavaScript, resulting in a responsive and interactive user experience for uploading photographs and viewing detection results.
Model Integration: The YOLO model is incorporated into the backend, with functions for initialisation, training, and detection.
Logging Mechanism: A logging mechanism is used to monitor the system's operations, providing timestamps and

status messages for important events.
Database: If needed, a lightweight database (such as SQLite) can be used to store user input.

2.3 Implementation.

Setting up the Environment: Install the required libraries and tools, such as Python, OpenCV, Flask, and the Ultralytics YOLO library.
Model Training: Using a labelled dataset, train the YOLO model, specify training parameters, and store the model weights.
Image Processing routines: Add routines for reading, resizing, and normalising images.
Detection Functions: Create functions for applying the YOLO model to input photos, generating bounding boxes, and labelling found items.
Web Interface Development: Create HTML, CSS, and JavaScript files for the frontend, then configure Flask routes to handle picture uploads and serve detection results.
Integration and testing: Combine all components, test the system for functionality and performance, and make any necessary changes.
Deployment: Deploy the system on a suitable server and ensure it is accessible through the web interface and capable of handling real-time detection requests.

2.4 Testing

2.4.1 Test Plan Objectives.

Verify Functionality: Ensure that all functional requirements are met, such as object identification, model training, image processing, and result visualisation.
Ensure usability: Made sure the online interface is easy to use and gives an intuitive experience.
Validate performance: Evaluated the system's performance in real-time object detection scenarios.
Guarantee Security: Ensured that the system is free of potential threats and vulnerabilities.
Check Robustness: Determine our system's robustness under a variety of scenarios, such as heavy load and unexpected input.

2.4.2 Data Entry.

Test Data: To test the object detection system, use a variety of photos with varying objects, backgrounds, and lighting.
Input Validation: Check that the system accepts erroneous inputs correctly, such as non-image files or corrupted images.
Data Consistency: Ensure that the uploaded photos are consistently processed and displayed.

2.4.3 Security: Limit web access to authorised users.
Data Protection: Ensure that the system securely manages and stores user data and photos.
Vulnerability Assessment: Conduct security evaluations to detect and address potential vulnerabilities.

2.4.4 Test Strategy Unit Testing: Evaluate individual components, including model initialisation, image processing, and web routing.
Integration Testing: Ensure that the components function together effortlessly.
System Testing: Verify that the entire system fits the criteria.
User Testing: Conduct user testing to get input on the web interface's usability and functioning.

2.4.5 System Testing: Conduct end-to-end testing, from picture upload to detection result visualisation.
Regression Testing: Check that new modifications do not introduce any new bugs or issues.

2.4.6 Performance Test

Response Time: The time required to process and detect objects in photos.
Throughput: Determine the system's ability to handle several requests simultaneously.
Scalability: Determine whether the system can scale to meet additional load.

2.4.7 Security Test.
Conduct penetration tests to uncover and resolve security flaws.
Authentication and Authorisation: Ensure that the authentication and authorisation procedures are properly implemented.

2.4.8 Basic Test: Ensure all basic capabilities, including image upload, detection, and result presentation, work as expected.
User Interface: Ensure that the web interface elements are properly displayed and functional.

2.4.9 Stress and Volume Test.
Load testing involves putting the system under high load conditions to ensure stability.
Volume Testing: Determine whether the system can manage massive amounts of data and photos.

2.4.10 Recovery Test: Evaluated the system's capacity to recover from errors such server crashes or network interruptions.
Data Integrity: We have ensured that the data is intact and consistent following recovery.

2.4.11 Test for Documentation Accuracy Ensure that all documentation, including user manuals and technical documentation, is correct and up to date.

2.4.12 User Acceptance Test
User Feedback: Collected feedback from users to verify the system meets their requirements and expectations.
Usability Testing: Conducted usability testing to detect faults.

2.4.13 System Assessment: Evaluate the complete system to confirm it meets requirements and is ready for deployment.
Final Review: Conducted a final assessment with teamates to ensure the system is ready for production use.

<u>**2.5Graphical User Interface (GUI)**</u>

Home Page

- Header: Displays the project title "Object Detection for Autonomous Vehicles".
- Navigation Bar: Includes links to Home, Upload Image, Live Detection, and Documentation.
- Main Content: Introduction to the project and brief instructions on how to use the system.
- Footer: Contact information and links to related resources.

Upload Image Page

- Header: Page title "Upload Image for Detection".
- Upload Form: Allows users to upload an image from their local device.
- Submit Button: Triggers the object detection process.
- Detection Results: Displays the uploaded image with bounding boxes and labels for detected objects.

Live Detection Page

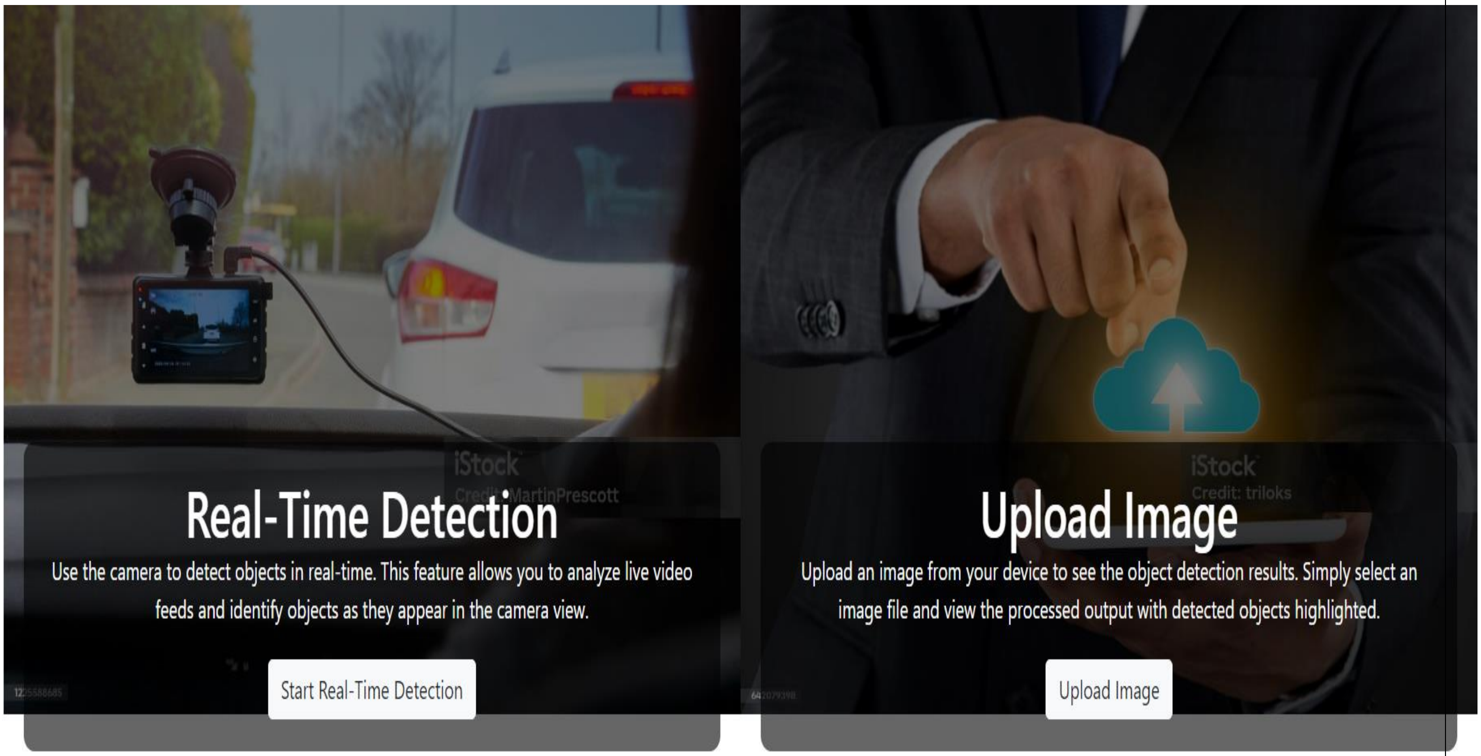
- Header: Page title "Live Object Detection".
- Live Stream: Displays real-time video feed from the camera.
- Start/Stop Button: Allows users to start or stop the live object detection.
- Detection Results: Shows bounding boxes and labels on the live video feed.

Result Page

- Header: Page title "Result".
- Content: Show the output produces by the model.

3Snapshots of Project

UI

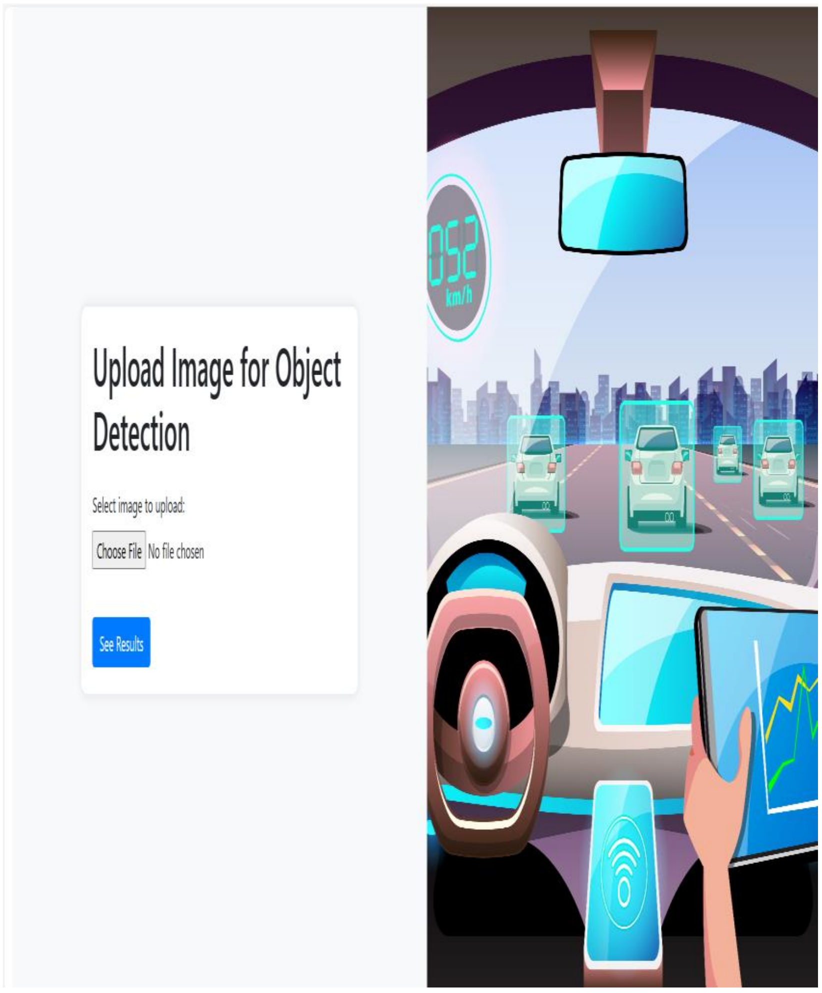


Real-Time Detection

Use the camera to detect objects in real-time. This feature allows you to analyze live video feeds and identify objects as they appear in the camera view.

Upload Image

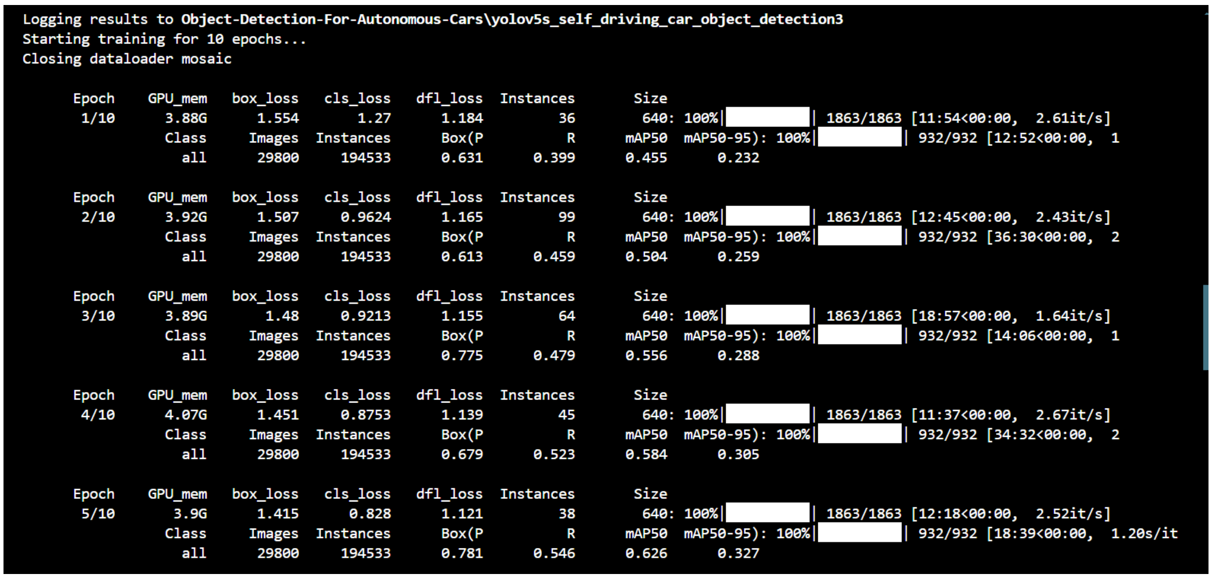
Upload an image from your device to see the object detection results. Simply select an image file and view the processed output with detected objects highlighted.



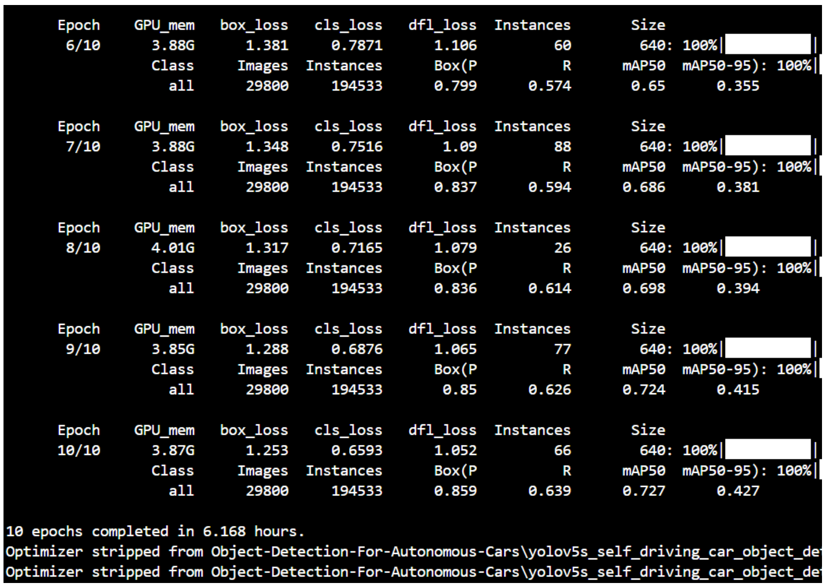
Upload Image for Object Detection

Select image to upload:

Training Progress

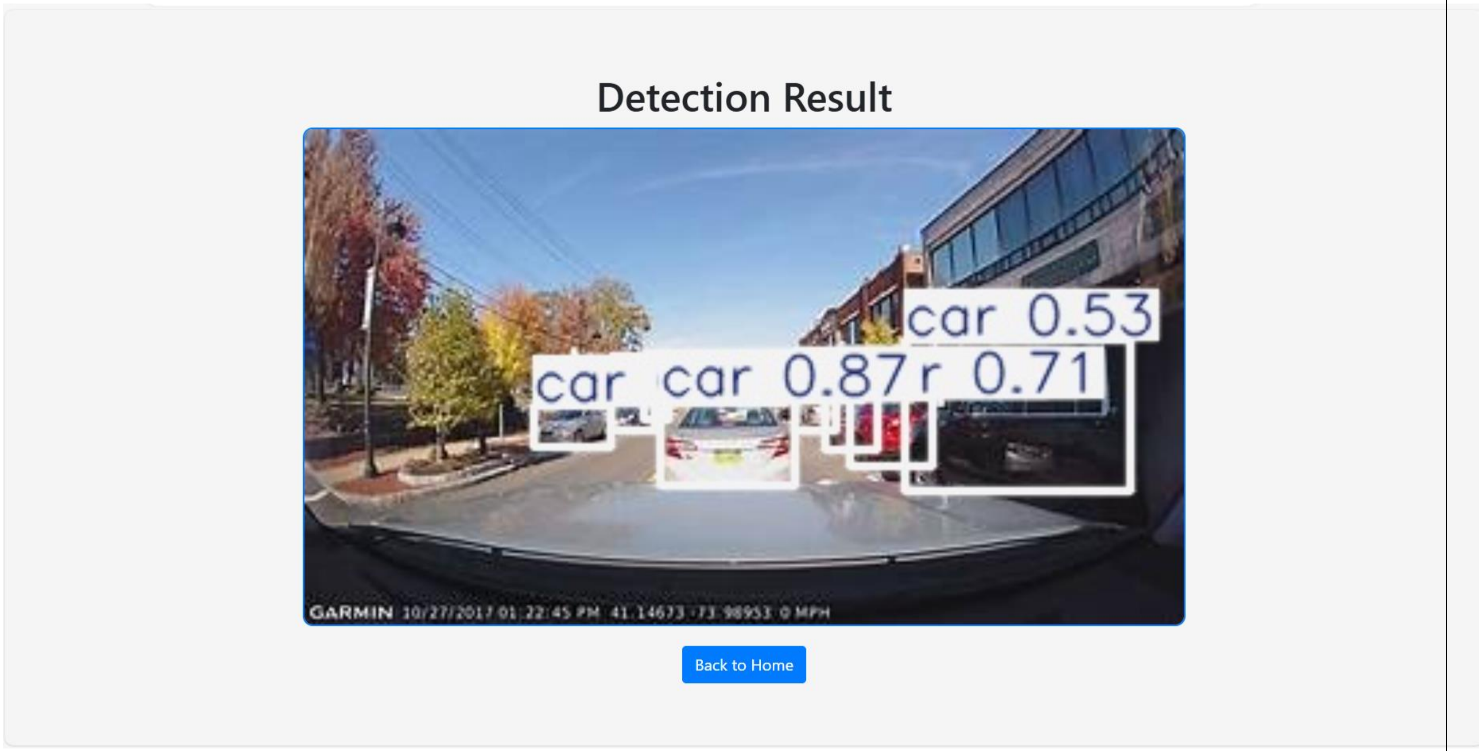


```
Logging results to Object-Detection-For-Autonomous-Cars\yolov5s_self_driving_car_object_detection3
Starting training for 10 epochs...
Closing dataloader mosaic

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/10      3.88G      1.554       1.27      1.184         36        640: 100%|██████████| 1863/1863 [11:54<00:00,  2.61it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 932/932 [12:52<00:00,  1
                   all      29800     194533      0.631      0.399      0.455      0.232

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       2/10      3.92G      1.507     0.9624      1.165         99        640: 100%|██████████| 1863/1863 [12:45<00:00,  2.43it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 932/932 [36:30<00:00,  2
                   all      29800     194533      0.613      0.459      0.504      0.259

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       3/10      3.89G       1.48     0.9213      1.155         64        640: 100%|██████████| 1863/1863 [18:57<00:00,  1.64it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 932/932 [14:06<00:00,  1
                   all      29800     194533      0.775      0.479      0.556      0.288

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       4/10      4.07G      1.451     0.8753      1.139         45        640: 100%|██████████| 1863/1863 [11:37<00:00,  2.67it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 932/932 [34:32<00:00,  2
                   all      29800     194533      0.679      0.523      0.584      0.305

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       5/10       3.9G      1.415      0.828      1.121         38        640: 100%|██████████| 1863/1863 [12:18<00:00,  2.52it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 932/932 [18:39<00:00,  1.20s/it
                   all      29800     194533      0.781      0.546      0.626      0.327
```



```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       6/10      3.88G      1.381     0.7871      1.106         60        640: 100%|          |
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all      29800     194533      0.799      0.574       0.65      0.355

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       7/10      3.88G      1.348     0.7516       1.09         88        640: 100%|          |
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all      29800     194533      0.837      0.594      0.686      0.381

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       8/10      4.01G      1.317     0.7165      1.079         26        640: 100%|          |
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all      29800     194533      0.836      0.614      0.698      0.394

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       9/10      3.85G      1.288     0.6876      1.065         77        640: 100%|          |
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all      29800     194533       0.85      0.626      0.724      0.415

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      10/10      3.87G      1.253     0.6593      1.052         66        640: 100%|          |
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all      29800     194533      0.859      0.639      0.727      0.427

10 epochs completed in 6.168 hours.
Optimizer stripped from Object-Detection-For-Autonomous-Cars\yolov5s_self_driving_car_object_de
Optimizer stripped from Object-Detection-For-Autonomous-Cars\yolov5s_self_driving_car_object_de
```

4 Conclusion



5 Further Research or Development

1 Model Improvement

Advanced Models: To improve accuracy and efficiency, investigate and integrate more advanced object detection models such as YOLOv7, EfficientDet, and Transformers.

Model Optimisation: Improve the present model's performance on embedded devices and low-end hardware.

Transfer Learning: Use transfer learning techniques to tailor the model to specific environments or conditions, increasing detection accuracy.

2 Dataset Expansion.

Diverse datasets: Expand the dataset to incorporate more diverse and complex settings, such as changing weather,

different times of day, and congested urban areas.
Synthetic Data: Use synthetic data generating techniques to generate more training data, improving the model's capacity to generalise.
3 Real-time Processing.
Edge Computing: Investigate edge computing systems for doing real-time object detection directly on the vehicle, lowering latency and dependency on cloud services.
Hardware Acceleration: Use hardware accelerators like as GPUs, TPUs, or FPGAs to accelerate processing and enable real-time detection on low-powered devices.
4 Integration of Autonomous Systems
Sensor Fusion: Integrate data from various sensors, such as LiDAR, radar, and GPS, to improve object detection accuracy and gain a more complete picture of the vehicle's surroundings.
Decision-Making Algorithms: Create and integrate algorithms that employ object detection data to make intelligent driving decisions, hence improving the overall safety and efficiency of autonomous cars.

Reference

- Rosebrock, A. (2019). *Deep Learning for Computer Vision with Python*. PyImageSearch.

- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.

- Anderson, J. M., Kalra, N., & Stanley, K. D. (2014). *Autonomous Vehicle Technology: A Guide for Policymakers*. Rand Corporation.

- Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. *arXiv preprint arXiv:1804.02767*.

- Bojarski, M., et al. (2016). End to End Learning for Self-Driving Cars. *arXiv preprint arXiv:1604.07316*.

- Zhang, K., et al. (2018). Image-Based Object Detection System for Self-Driving Cars Application. *IEEE Transactions on Intelligent Transportation Systems*.

- ISO. (2018). ISO 26262: Road Vehicles - Functional Safety. International Organization for Standardization.