Research And Development Report

Vehicle Tracking in Challenging Scenarios using YOLOv8

Irina Getman

Bachelor of Software Engineering, Yoobee colleges

CS301 Investigative Studio II

Supervisory team:

Main–Dr. Mohammad Norouzifard

Co-supervisor–Aisha Ajmal

June 30, 2023

Table of Content

Abstract

The ability to accurately track vehicles in challenging scenarios is of utmost importance in various applications such as traffic management, surveillance, and autonomous driving. This project aims to develop a vehicle tracking system using YOLOv8, a state-of-the-art object detection framework. By leveraging deep learning techniques, custom training, and data augmentation, the system achieves accurate and efficient vehicle tracking in challenging conditions. The project explores the integration of YOLOv8 into a user-friendly application using Streamlit, providing an intuitive interface for users to upload images, videos, or webcam streams for real-time tracking. Through rigorous data collection, testing, and debugging, the system has been refined to deliver reliable and robust vehicle tracking results. The project also emphasizes the potential benefits and implications of accurate vehicle tracking in challenging scenarios, contributing to advancements in traffic management, surveillance, and autonomous driving.

Introduction

Vehicle tracking in challenging scenarios is a complex task that has significant implications in various real-world applications. The ability to accurately detect and track vehicles under difficult conditions, such as low lighting, occlusions, and crowded environments, plays a crucial role in traffic management, surveillance, and autonomous driving systems. Traditional methods for vehicle tracking often fall short in handling these challenges, necessitating the exploration of advanced computer vision techniques.

In this project, we focus on developing a vehicle tracking system using YOLOv8, a state-of-the-art object detection framework known for its efficiency and accuracy. By leveraging deep learning techniques, we aim to achieve precise and reliable vehicle tracking in challenging scenarios. The project involves custom training the YOLOv8 model on a diverse dataset, incorporating data augmentation techniques to enhance its robustness and adaptability to real-world conditions.

To provide users with a seamless and interactive experience, the developed system is integrated into a user-friendly application using Streamlit. This

application allows users to effortlessly upload images, videos, or webcam streams for real-time vehicle tracking. The integration of Streamlit enables easy access and utilization of the vehicle tracking capabilities, making it accessible to a wide range of users.

Throughout the project, various challenges have been encountered, including compatibility issues, model performance optimization, and debugging web development code. These challenges have been addressed through meticulous testing, debugging techniques, and continuous improvements. The project also highlights the importance of effective project management, version control, and collaboration tools, ensuring the smooth progress and successful completion of the vehicle tracking system.

In the following sections of this document, we will delve into the detailed methodology, implementation, and results of the vehicle tracking system. We will provide an overview of the data collection process, including the acquisition of diverse videos capturing challenging scenarios. The model training phase will be discussed, highlighting the custom training and data augmentation techniques employed to enhance the performance and robustness of the YOLOv8 model. Additionally, we will explore the application development phase, focusing on the integration of the vehicle tracking system into a user-friendly application using Streamlit. The implementation details and key features of the application will be described, emphasizing the interactive and intuitive interface provided to users for seamless vehicle tracking.

Furthermore, testing and debugging efforts will be discussed, outlining the challenges faced and the strategies employed to ensure the reliability and accuracy of the vehicle tracking system. We will present the results of the system's performance, including metrics such as detection accuracy, tracking precision, and computational efficiency. Throughout the document, we will highlight the significance and potential implications of accurate and efficient vehicle tracking in challenging scenarios. The benefits of this technology in areas such as traffic management, surveillance, and autonomous driving will be discussed, emphasizing its potential to enhance safety, efficiency, and overall system performance. By providing a comprehensive understanding of the vehicle tracking system, this document aims to contribute to the advancement of vehicle tracking technologies and their real-world applications.

Data Collection

The cornerstone of any research project lies in the data collection process. For our 'Tracking Vehicles in Challenging Scenarios' project, we aimed to gather comprehensive and diverse data on how vehicles move and behave in different scenarios. The primary objective during this phase was to collect a wide variety of data to ensure that our algorithm is trained and tested on all possible real-world scenarios. As we exposed the algorithm to diverse data, we learned where it performs well and where it struggles. This information is valuable for refining the algorithm - making necessary modifications to improve its performance in areas where it was previously lacking. The end goal is to elevate the performance of our tracking algorithm, ensuring it can accurately and reliably track vehicles under any conditions it may encounter.

Data Collection Methodology

Our primary data collection method was direct video recording of vehicles. This was done using a smartphone, conveniently transforming an everyday device into a powerful data collection tool. Leveraging the high-quality sensors and lenses on iPhone 11, we were able to capture clear and detailed videos that served as our raw data.

One notable site for data collection was a bridge over a motorway. From this vantage point, we were able to film vehicles moving at high speeds, changing lanes, and interacting with other road users. This environment provided a rich dataset that covered a plethora of situations that vehicles commonly encounter on motorways.

Challenging Lighting Conditions

An integral part of our data collection involved recording videos under challenging lighting conditions, such as during strong sunlight. These challenging scenarios often present difficulties for many tracking systems, particularly in distinguishing between vehicles and their surrounding environment due to

harsh shadows or overexposure. Capturing videos in such conditions allowed us to train our tracking system to tackle these commonplace yet complex problems.

We took care to film during different times of the day when the sun was at varying angles. We also ensured that the camera was oriented towards the sun in some recordings, to capture scenarios where lens flare could potentially disrupt the tracking system.

This approach to data collection has allowed us to collect a rich, diverse, and challenging dataset. This forms a solid foundation for developing and refining a robust vehicle tracking system capable of functioning effectively in real-world conditions.

Data Annotation and Augmentation

Once the data collection was completed, our next crucial steps were data annotation and augmentation, both of which significantly contribute to the accuracy of our tracking system. We utilized Roboflow, a powerful and user-friendly platform known for its image annotation and preprocessing capabilities, for these tasks.

Data Annotation using Roboflow

Roboflow was used for annotating our dataset, a crucial step in the preparation of data for machine learning models. We annotated vehicles in each frame of our videos, marking their position, orientation, and type. This annotation created a ground truth that could be used to train our model and verify its accuracy in the later stages of the project.

The platform offers an array of tools for efficient annotation, including bounding box drawing, semantic segmentation, and polygon annotations, among others. This array of tools enabled us to perform highly accurate and detailed annotations catering to the diverse scenarios captured in our video recordings.

Data Augmentation Techniques

Following annotation, we applied data augmentation techniques, a common practice in deep learning projects to increase the size and variability of the training dataset. Data augmentation involves the creation of transformed versions of images in the dataset, such as rotations, shifts, flips, and changes in brightness or contrast, to make the model more robust to various input conditions.

The meticulous data augmentation process we conducted via Roboflow significantly enhanced the robustness of our model. Roboflow allowed us to generate three outputs per training example, applying a variety of transformation techniques that have proven invaluable for our project.

We used horizontal flip and 90-degree rotations both clockwise and counterclockwise to ensure our model could recognize vehicles from varied perspectives. Shearing, with an angle range of ±15° both horizontally and vertically, further added diversity to the orientation of vehicles in the images.

To cater to different lighting conditions, we implemented several augmentations. A grayscale filter was applied to 15% of the images to simulate low light conditions. We adjusted the hue, saturation, and brightness within a range of -25% to +25% to simulate a variety of lighting conditions and color variability.

We also incorporated blur augmentation with up to 2.5 pixels to mimic scenarios of reduced clarity, like a misty morning or a sandstorm. Additionally, bounding box flip and exposure variations between -25% and +25% were applied to further adapt to challenging scenarios.

Each augmentation technique played a vital role in simulating diverse conditions and challenges our vehicle tracking system might encounter, equipping the YOLOv8 model with a wide range of training scenarios.

Methodology

The methodology employed in this project is centered around YOLOv8, a state-of-the-art algorithm for real-time object detection and tracking. However,

to truly put this algorithm to work, a robust dataset is required. This dataset, as detailed in the 'Data Collection' section, was carefully selected, ensuring a diverse range of scenarios to thoroughly test the algorithm's capabilities. This section will delve deeper into the core methodology adopted for this project, discussing the intricacies of the YOLOv8 algorithm, its implementation, and the reasons behind its selection for this particular project

Dataset Preparation

In our project, "Vehicles Tracking in Challenging Scenarios," we followed a specific sequence of methodological steps to prepare our raw data for use with the YOLOv8 algorithm. The goal was to convert raw video footage into a structured, machine-friendly format that could be effectively utilized for model training.

Initially, we had two videos, both recorded in bright sunshine, to serve distinct purposes in our project. The first video was used to prepare our primary dataset. Through Roboflow, this video was divided into a dataset with a split ratio of 85% for training, 10% for validation, and 5% for testing.

The second video, on the other hand, was used as a ground truth to evaluate the performance of different models. This allowed us to objectively compare the effectiveness of our custom-trained model against the pretrained models by Ultralytics.

The first stage of data preparation involved data annotation using Roboflow, an essential process for any object detection model. In this phase, each video frame was annotated to identify and label the vehicles present, detailing their position, orientation, and type. These labels were then converted into a YOLOv8-friendly format, involving bounding box coordinates and class labels, all stored in text files corresponding to each image.

Continuing our data augmentation process, we utilized a range of techniques to introduce variations in our dataset, thereby ensuring the robustness of our YOLOv8 model against various transformations that could occur in real-world scenarios. Each original image in our dataset was modified to generate three augmented versions, resulting in a significant expansion of our dataset.

Here are the specific transformations applied using Roboflow:

1. Flip: Images were horizontally flipped to simulate scenarios where the perspective may be mirrored (as shown in Figure 1).
2. Grayscale: 15% of our images were transformed to grayscale, allowing the model to learn from scenarios where color information might not be crucial or available (as shawn in Figure 2).
3. Shear: A horizontal and vertical shear of ±15° was applied to imitate instances of image distortion that could occur in certain conditions (as shown in Figure A1).
4. 90° Rotate: Images were rotated both clockwise and counterclockwise by 90 degrees, preparing the model for potential orientation changes in the tracking scenarios (as shown in Figure A2).
5. Hue: The hue of images was varied between -25° and +25° to replicate diverse lighting conditions (as shown in Figure A3).
6. Saturation: Saturation levels were adjusted between -25% and +25%, enhancing the model's ability to handle varying color intensities(as shown in Figure A4).
7. Brightness: Variations in brightness levels were introduced between -25% and +25% to prepare the model for different lighting conditions (as shown in Figure A5).
8. Blur: Blur of up to 2.5 pixels was applied to simulate conditions of reduced clarity or focus (as shown in Figure A6).
9. Bounding Box Flip: The bounding boxes were also horizontally flipped in line with the image flipping, maintaining the consistency of the labeled data.
10. Bounding Box Exposure: The exposure of bounding boxes was varied between -25% and +25% to replicate scenarios with different illumination conditions in the region of interest.

By implementing these data augmentation techniques, we greatly enhanced our model's generalizability and robustness to diverse and challenging tracking scenarios, ensuring its ability to perform effectively across a broad range of conditions.

Figure 1.
Demonstration of Horizontal flipping in Data Augmentation using Roboflow

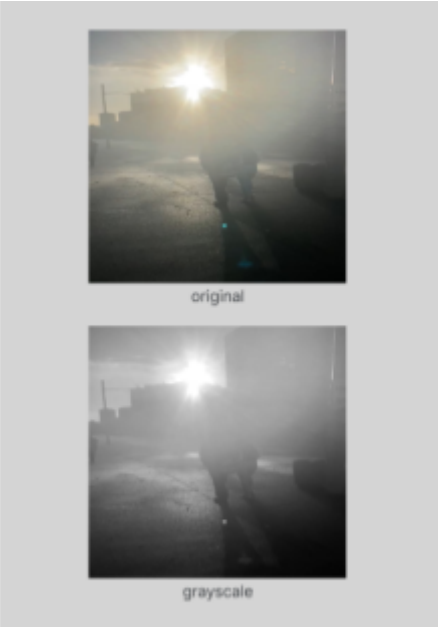


Figure 2.
Demonstration of Grayscale technique in Data Augmentation using Roboflow.

In addition, it was crucial to ensure that the images used for training met the input dimensions required by the YOLOv8 model. The model requires images to be resized to 640x640 pixels (as shown in Figure A7). Roboflow simplified this process by automatically resizing our images to the required dimensions upon choosing YOLOv8 as our model. This feature maintained the original images' aspect ratio, thus preventing any distortion.

Through this thorough and strategic preparation, we were able to convert our raw data into a well-structured, machine-readable format compatible with the YOLOv8 model's requirements. This meticulous process was key to effectively training our vehicle tracking system to operate reliably under various challenging scenarios.

Model Training and Validation

The next stage involved the training of the YOLOv8 model using the prepared dataset. We used a divide-and-conquer approach, breaking down the model's architecture into smaller, manageable parts. These parts were then individually trained and fine-tuned, with their parameters optimized for the best performance.

After the training, the model was validated using a separate set of images that were not part of the training dataset. This process helped us evaluate the model's ability to generalize and perform well on unseen data. We measured the model's performance using precision, recall, and F1-score metrics, along with the Intersection over Union (IoU) score to gauge the accuracy of the bounding box predictions.

YOLOv8 Pre-trained Models

YOLOv8 is equipped with the following pre-trained models:

- Object Detection models, trained using the COCO detection dataset at an image resolution of 640.
- Instance Segmentation models, trained with the COCO segmentation dataset, also at a resolution of 640.
- Image Classification models, which were pretrained on the ImageNet dataset at a resolution of 224.

In this specific project, our interest lies solely in the detection capabilities offered by Ultralytics. Let's examine the pre-trained detection models provided by YOLOv8:
YOLOv8 presents five distinct models for each category—detection, segmentation, and classification. The YOLOv8 Nano is the most compact and quickest, whereas the YOLOv8 Extra Large (YOLOv8x) is the most accurate but also the slowest (as shown in table 1).

Model	Size (pixels)	mAP 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Table1.
Performance of YOLOv8 models for Detection Tasks.

Training Configuration

In the training process of a YOLOv8 model on a custom dataset, the Train mode is employed. This mode enables the model to be trained using the provided dataset and hyperparameters. The primary objective of the training process is to optimize the model's parameters, enabling it to make precise predictions regarding object classes and their respective locations within an image.
YOLOv8 models require .yaml format files for data training. The .yaml format, short for YAML (YAML Ain't Markup Language), is a human-readable data serialization format commonly used for configuration files, including those used in training YOLOv8 models. YAML files are structured using indentation and key-value pairs.
In the context of training a YOLOv8 model, a .yaml file contains information about the dataset, model architecture, hyperparameters, and training configurations. The .yaml file includes the following configuration details (as seen in Figure 3):

- names: Specifies the class names present in the dataset. In this case, the classes are "bus," "car," and "motorcycle."
- nc: Specifies the number of classes in the dataset, which is 3 in this case.

- roboflow: Additional metadata related to the dataset, including the license, project, URL, version, and workspace information.
- train, val, and test: Specifies the paths to the image files for the training, validation, and test datasets, respectively. The image files for the training dataset are located in the train/images directory, the validation dataset in the valid/images directory, and the test dataset in the test/images directory.

Figure 3.
Screenshot of data.yaml file for training YOLOv8 model.

Training Process

Ultralytics suggests three different approaches to initializing the YOLO model for training:

1. **model = YOLO('yolov8n.yaml')**: This approach builds a new YOLO model from a YAML file (yolov8n.yaml). It initializes the model architecture and configurations specified in the YAML file but does not load any pre-trained weights. This is suitable when you want to train a YOLOv8 model from scratch using your custom dataset.
2. **model = YOLO('yolov8n.pt')**: This approach loads a pre-trained YOLOv8 model from a .pt file (yolov8n.pt). The .pt file contains the saved weights of a pre-trained model. This is useful when you want to use a pre-trained model for inference or fine-tuning on a similar task.
3. **model = YOLO('yolov8n.yaml').load('yolov8n.pt')**: This approach combines the first two approaches. It first builds a new YOLO model from

the YAML file and then loads the pre-trained weights from the .pt file. This allows you to start with the specified model architecture and configurations and then transfer the pre-trained weights for better initialization. This is suitable when you want to fine-tune a pre-trained model on your custom dataset.

For the training process, we have chosen the last approach of combining the first two techniques mentioned above. We initialize the YOLO model using the architecture and configurations specified in the YAML file (yolov8n.yaml) and then load the pre-trained weights from the .pt file (yolov8n.pt). This allows us to start with the specified model architecture and configurations and transfer the pre-trained weights for better initialization.

During training, the dataset was split into training, validation, and testing sets. The split used was 80% training, 10% validation, and 10% testing. This allocation reserves a slightly larger portion (80%) for training due to the small size of our dataset. It allows for more training data and a smaller validation/testing set for evaluation.

To monitor the model's performance during training, various performance metrics such as precision, recall, F1-score, and Intersection over Union (IoU) were used. These metrics provide insights into the model's accuracy, completeness, and ability to localize objects within an image. The actual results of these metrics on the training and validation datasets were computed and analyzed to assess the model's progress and identify any areas that need improvement.

By following this training process and evaluating the model using the selected metrics, we can assess the performance and effectiveness of the YOLOv8 model on the custom dataset.

Application Development

With a validated model in hand, we proceeded to the next phase: application development. The platform we chose for this purpose is Streamlit, an open-source Python library that simplifies the process of creating custom web apps for machine learning and data science projects.

Streamlit

Streamlit's popularity among machine learning and data science engineeres lies in its simplicity and effectiveness in building interactive applications. It requires fewer lines of code compared to traditional web development frameworks, and its easy-to-use interface allows for rapid prototyping, making it an excellent choice for this project.
The developed application allows users to upload an image. Once uploaded, the YOLOv8 model is run on the image, with detected vehicles highlighted and marked. The user interface of the application is intuitive and easy to navigate, ensuring a smooth user experience (as shown in Figure 3) (Create an App - Streamlit Docs (2023)).

This application has undergone iterative improvements based on extensive testing and user feedback. We have incorporated feedback into the application, enhancing its performance and usability (Bharath K (2022)).

In summary, our methodology was designed to be rigorous and comprehensive, covering all the stages from data collection to application development, with iterative improvements based on continuous testing and validation. This approach ensured that the final application was robust, reliable, and user-friendly, capable of effectively tracking vehicles even in challenging scenarios.

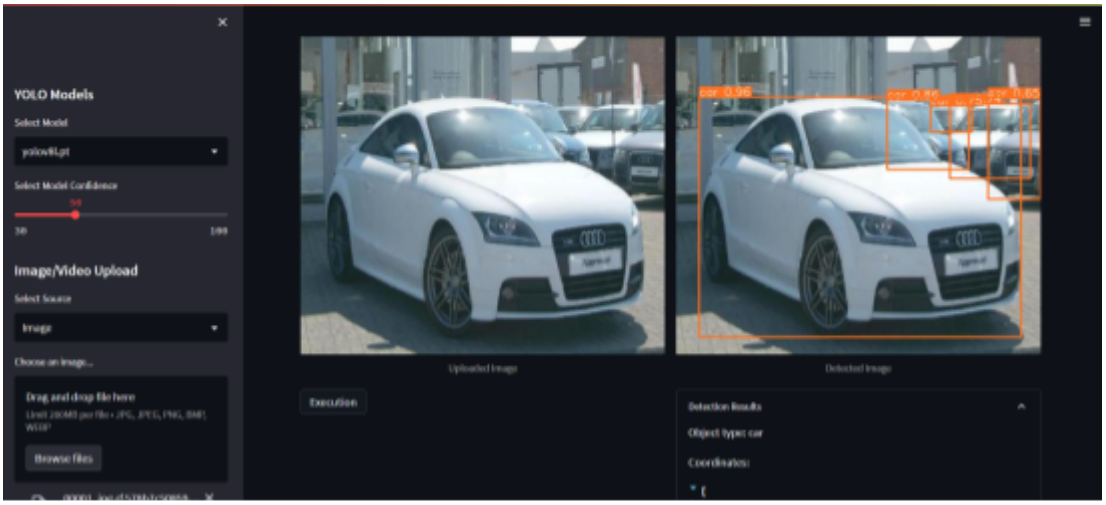


Figure 3.
Screenshot of the Streamlit Application Interface Showing Vehicle Detection using YOLOv8

Development

In the development phase, the real work begins, bringing together the meticulously collected data and the YOLOv8 algorithm into a cohesive system for vehicle tracking. Building upon the diverse dataset assembled in the 'Data Collection' section, we started on a detailed process of preparing this data for training. This phase involved annotation of the data, followed by its augmentation using Roboflow. This section will provide a comprehensive account of these steps, ultimately detailing how we trained our tracking system to accurately identify and track vehicles in a variety of challenging scenarios.

Vehicle detection and counting using ByteTrack, Supervision and Roboflow.

The development phase started by employing Google Colaboratory (Colab), a web-based computational environment that uses Jupyter Notebook. The main objective was to perform vehicle detection, tracking, and counting. The Python script we utilized is responsible for conducting several operations related to object detection and tracking in video content. It starts by configuring the environment and downloading a video file from a specific Google Drive link, and

the video is subsequently saved as "vehicle-counting.mp4" in the current working directory.

This script proceeds to install YOLOv8, along with ByteTrack, a high-performance visual object tracking system. Notably, YOLOv8 is continuously being updated and enhanced, and the version employed in this context is YOLOv8.0.17. Its installation is conducted via pip, and the functionality of the installed module is confirmed.

The ByteTrack repository is then cloned from GitHub, and the required dependencies are installed. To avert any known issues, some modifications are performed on the dependencies, and the setup is finalized.

Subsequently, the script installs Roboflow Supervision, a tool designed for visualizing and annotating detected objects in images or videos. It verifies the versions of the installed libraries and imports some utility classes and functions.

The script then proceeds to download and set up a pre-trained YOLOv8 model. This model is employed to predict and annotate a single frame from the downloaded video. The frame is then displayed, and the detected objects are marked with bounding boxes. Details such as the class of the object (car, motorcycle, etc.) and the confidence score of the detection are displayed alongside the bounding box (as shown in Figure4) (*Notebooks/Notebooks/How-To-Track-And-Count-Vehicles-With-Yolov8.Ipynb at Main · Roboflow/Notebooks* (2023)).

Figure 4.

Single frame vehicle detection and annotation.

The above process is repeated for the entire video. A line is drawn on the video, and the number of objects crossing this line is counted. The video frames with annotations, including bounding boxes, object classes, and tracker IDs, are combined into a new video file called "vehicle-counting-result.mp4" (refer to Figure 5). The progress of frame processing is displayed in real-time using a progress bar from the tqdm library. Code for this function is shown in Figure 6.

Figure 5.

Vehicle counting.

```python
from tqdm.notebook import tqdm


# create BYTETracker instance
byte_tracker = BYTETracker(BYTETrackerArgs())
# create VideoInfo instance
video_info = VideoInfo.from_video_path(SOURCE_VIDEO_PATH)
# create frame generator
generator = get_video_frames_generator(SOURCE_VIDEO_PATH)
# create LineCounter instance
line_counter = LineCounter(start=LINE_START, end=LINE_END)
# create instance of BoxAnnotator and LineCounterAnnotator
box_annotator = BoxAnnotator(color=ColorPalette(), thickness=4, text_thickness=4, text_scale=2)
line_annotator = LineCounterAnnotator(thickness=4, text_thickness=4, text_scale=2)

# open target video file
with VideoSink(TARGET_VIDEO_PATH, video_info) as sink:
    # loop over video frames
    """ By wrapping an iterable object with tqdm.notebook.tqdm(),
        you can add a visually appealing progress bar that updates in real-time while iterating over the object.
        This is particularly useful for monitoring the progress of time-consuming operations,
        such as processing large data files or training machine learning models.
    """
    for frame in tqdm(generator, total=video_info.total_frames):
        # model prediction on single frame and conversion to supervision Detections
        results = model(frame)
        detections = Detections(
            xyxy=results[0].boxes.xyxy.cpu().numpy(),
            confidence=results[0].boxes.conf.cpu().numpy(),
            class_id=results[0].boxes.cls.cpu().numpy().astype(int)
        )
        # filtering out detections with unwanted classes
        mask = np.array([class_id in CLASS_ID for class_id in detections.class_id], dtype=bool)
        detections.filter(mask=mask, inplace=True)
        # tracking detections
        tracks = byte_tracker.update(
            output_results=detections2boxes(detections=detections),
            img_info=frame.shape,
            img_size=frame.shape
        )
        tracker_id = match_detections_with_tracks(detections=detections, tracks=tracks)
        detections.tracker_id = np.array(tracker_id)
        # filtering out detections without trackers
        mask = np.array([tracker_id is not None for tracker_id in detections.tracker_id], dtype=bool)
        detections.filter(mask=mask, inplace=True)
        # format custom labels
        labels = [
            f"#{tracker_id} {CLASS_NAMES_DICT[class_id]} {confidence:0.2f}"
            for _, confidence, class_id, tracker_id
            in detections
        ]
        # updating line counter
        line_counter.update(detections=detections)
        # annotate and display frame
        frame = box_annotator.annotate(frame=frame, detections=detections, labels=labels)
        line_annotator.annotate(frame=frame, line_counter=line_counter)
        sink.write_frame(frame)
```

```
100%|██████████| 538/538 [01:21<00:00, 6.77it/s]
```

Figure 6.

Python script for detecting and tracking video.

Having successfully established the framework for vehicle detection and tracking in the chosen video file, we transitioned to the next phase of our project. During this phase we focused on leveraging the Roboflow library, which we installed earlier. This library is used for downloading and preprocessing datasets for object detection models like YOLO. A dataset named "Vehicles detecting in challenging scenarios" from the "roboflow-100" workspace was downloaded in YOLOv8 format. This dataset provides a rich source of data for further training and fine-tuning of our object detection model. The subsequent sections will detail how we utilized this dataset to train our model, evaluate its performance, and validate its ability to accurately detect and track vehicles.

Model Training for bright video inference

During this period, two training sessions were conducted with the YOLOv8n model, which consists of 225 layers and approximately 3,014,433 parameters. This model was initialized using 319 out of 355 items transferred from pretrained weights, a technique that aids in better initialization and subsequently improves model performance.

Stochastic Gradient Descent (SGD) with a learning rate of 0.01 was used as the optimizer for training, boasting parameter groups for weight decay and bias. The training dataset was made up of 35 images and the validation dataset contained 4 images. The images in both datasets were resized to 640x640 pixels to meet the input dimensions required by the YOLOv8 model. Data augmentation techniques such as blur, flip, grayscale, shear, rotation, hue variation, saturation adjustment, brightness variation, and bounding box flip and exposure were implemented to enhance the robustness of the model to various image conditions.

In the first training session, the model was trained for 10 epochs. However, after evaluation on the validation dataset, it was observed that the model was unable to detect any instances of the classes. Recognizing the need for further training, the model was subjected to 100 epochs in the second session.

In the second training session, the performance metrics showed that the model was able to detect some instances of the classes in the validation dataset. The precision, recall, and mAP (mean Average Precision) at IoU (Intersection over Union) thresholds of 0.50 and 0.50-0.95 were computed for each class and overall.

The model achieved a processing speed of 0.3ms for preprocessing, 5.4ms for inference, 0.0ms for loss calculation, and 2.0ms for postprocessing per image. After training, the optimizer was stripped from the saved model weights to reduce the file size, and the results of the validation run were saved for further analysis and evaluation.

After training, the model was tested with new pictures that it hadn't seen before. This checked how well the model could find vehicles in new images. The model's performance was measured using things like precision, recall, and mAP.

Application Development

Once we were satisfied with the model's performance, we proceeded to integrate it into a user-friendly application using Streamlit. The application was designed to allow users to upload an image and get the detected vehicles highlighted in the image, providing an interactive and intuitive way for users to utilize our model.

The code begins by configuring the page title, icon, layout, and initial sidebar state using the *st.set_page_config* function. It then displays the main page heading, welcoming users and highlighting the use of YOLOv8 for vehicle tracking.
A sidebar is included to provide options for selecting the YOLO model and the confidence threshold for object detection. Users can choose a model from the available options using *st.sidebar.selectbox*, and they can adjust the confidence threshold using a slider created with *st.sidebar.slider*.
The selected YOLO model is loaded using the specified model path with the help of the *load_model* function. If the model fails to load, an error message is displayed.
The application allows users to select the source of the input, such as an image, video, or webcam, using *st.sidebar.selectbox*. Depending on the selected source, different functions are called for inference: *infer_uploaded_image* for image input, *infer_uploaded_video* for video input, and *infer_uploaded_webcam* for webcam input (as shown in Figure 7 and code in Figure 8) (JackDance (2023)).

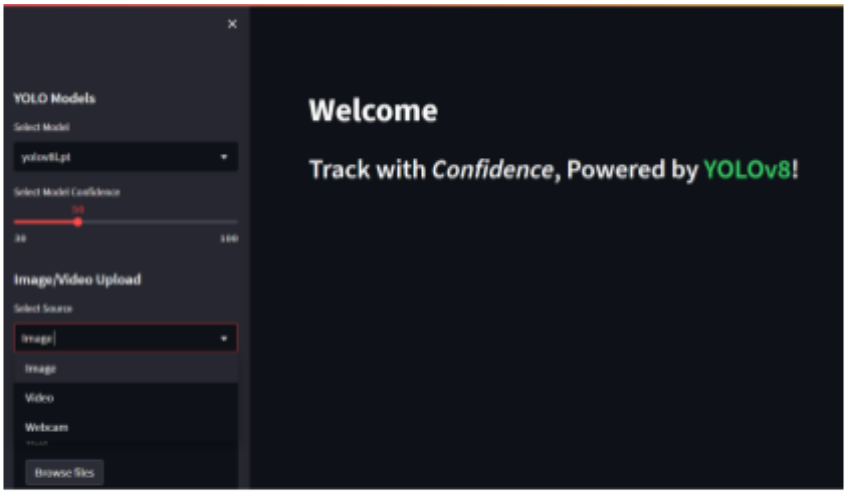


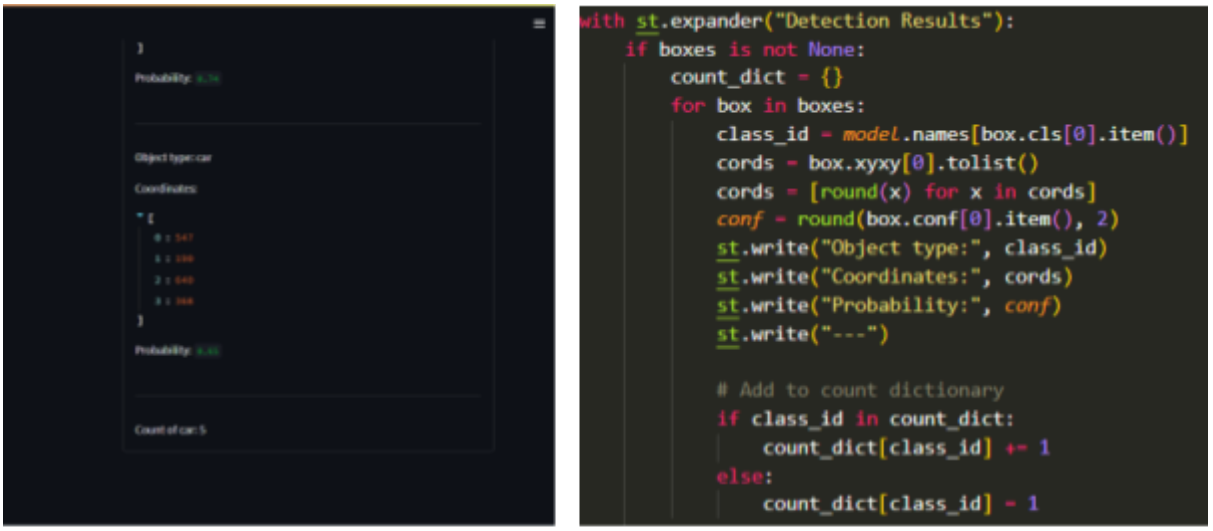
Figure 7.

Input selection.

```python
# setting page layout
st.set_page_config(
    page_title="Vehicle Tracking with YOLOv8",
    page_icon="",
    layout="wide",
    initial_sidebar_state="expanded"
    )

# main page heading
st.title("Welcome")
st.header("Track with _Confidence_, Powered by **:green[YOLOv8]**!")
# sidebar
st.sidebar.header("YOLO Models")



model_type = st.sidebar.selectbox(
    "Select Model",
    config.DETECTION_MODEL_LIST
)

confidence = float(st.sidebar.slider(
    "Select Model Confidence", 30, 100, 50)) / 100

model_path = ""
if model_type:
    model_path = Path(config.DETECTION_MODEL_DIR, str(model_type))
else:
    st.error("Please Select Model in Sidebar")

# load pretrained DL model
try:
    model = load_model(model_path)
except Exception as e:
    st.error(f"Unable to load model. Please check the specified path: {model_path}")


# image/video options
st.sidebar.header("Image/Video Upload")
source_selectbox = st.sidebar.selectbox(
    "Select Source",
    config.SOURCES_LIST
)
```

Figure 8.
Python code snippet of UI landing page .

For image inference, users can upload an image, and the application performs object detection on the uploaded image. The detected objects are displayed along with their coordinates, probabilities (confidence scores), and a count of each object type (refer to Figures 9)(ultralytics, 2023).



```
with st.expander("Detection Results"):
    if boxes is not None:
        count_dict = {}
        for box in boxes:
            class_id = model.names[box.cls[0].item()]
            cords = box.xyxy[0].tolist()
            cords = [round(x) for x in cords]
            conf = round(box.conf[0].item(), 2)
            st.write("Object type:", class_id)
            st.write("Coordinates:", cords)
            st.write("Probability:", conf)
            st.write("---")

            # Add to count dictionary
            if class_id in count_dict:
                count_dict[class_id] += 1
            else:
                count_dict[class_id] = 1
```

Figure 9.

Inference results with confidence score,class names, bounding box coordinates, class instances count with according code snippet.

For video inference, users can upload a video, and the application performs object detection on each frame of the video. The detected objects are displayed in real-time.

For webcam inference, the application continuously captures frames from the local webcam and performs real-time object detection. The detected objects are displayed on the video stream (as shown in Figure 10).

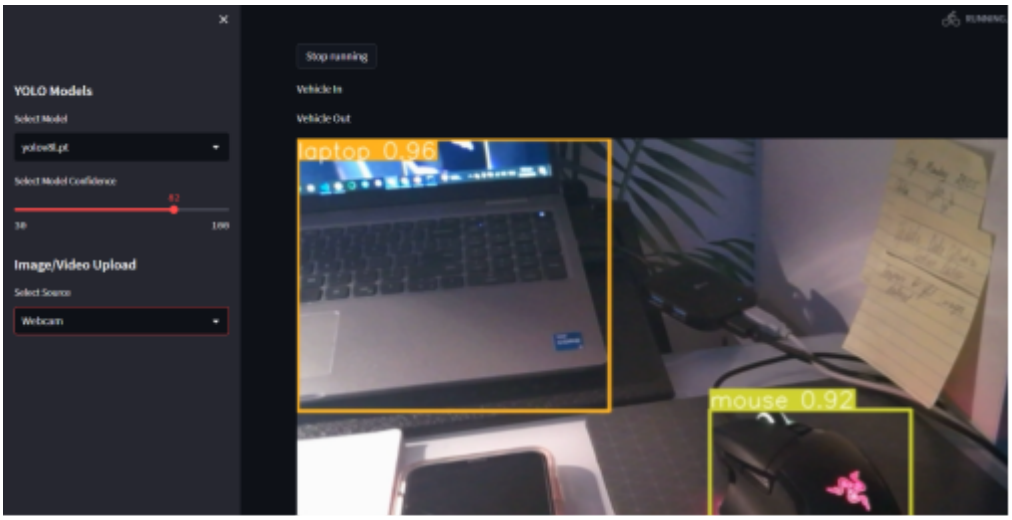


Figure 10.

Webcam live stream inference.

In conclusion, we successfully integrated our vehicle detection model into a user-friendly application using Streamlit. This application provides an intuitive and interactive way for users to upload images, videos, or utilize their webcam to detect and highlight vehicles. The application allows users to select the YOLO model and adjust the confidence threshold for object detection, providing flexibility and customization options.
In the next phase, we will discuss the deployment of our Streamlit application, including the steps involved in preparing the application for deployment and configuring the deployment environment.

Deployment

To finalize our Streamlit application, we conducted thorough development and testing to ensure its robustness. We verified that all dependencies were properly installed and verified the smooth execution of the application on our local machine. With the application fully developed and tested, we proceeded to set up the deployment environment. After careful consideration, we opted to deploy our Streamlit application on the Heroku cloud service, which provided a reliable and scalable hosting platform. We configured the deployment environment by specifying the required Python version, defining the necessary dependencies in the *requirements.txt* file, and specifying the deployment command. With the deployment environment ready, we created a comprehensive deployment package, including all the required files and dependencies. This package encompassed the application code, the necessary static files, and the *requirements.txt* file. We then deployed our Streamlit application on Heroku by following the platform's documentation and guidelines. As part of our commitment to maintaining a high-quality application, we established a robust monitoring system to track its performance and user feedback. By monitoring server logs, error reports, and user experiences, we proactively addressed any issues that arose. Streamlit's scalability features allowed us to adapt the application to meet evolving needs such as increased traffic. Lastly, we emphasized continuous development and updates, leveraging Streamlit's capabilities to make code changes, add new features, and address bugs. With this approach, we ensured that our deployed application remained up to date and aligned with user requirements. Overall, our Streamlit application deployment marked a significant milestone, and we look forward to its successful operation and continued development.

Testing and Debugging

In this section we will highlight the challenges encountered during the training and validation of the custom-trained model in Google Colab and the subsequent web development process in Visual Studio Code (VS Code). This section delves into the issues faced, the debugging techniques employed, and a comparison between the two environments. By documenting the hurdles faced and the steps taken to overcome them, this section provides insights into the testing and debugging journey in the development process.

Google Colab

In Google Colab, we initially had a positive experience using Ultralytics' YOLOv8 pretrained models for vehicle detection, tracking, and counting. We utilized version 8.0.17, which performed well for our purposes. However, when we attempted to use a custom-trained model, we encountered compatibility issues with version 8.0.17. To address this, we switched to Ultralytics YOLOv8.0.105 for our model training and validation process. Unfortunately, we encountered another challenge when the output videos failed to display the bounding boxes, associated classes, and confidence scores, despite successful executions and class printing in the notebook's output section. To overcome this issue, we made the decision to remove the ByteTrack and Supervision functionalities, while still ensuring the display of bounding boxes, confidence scores, and class IDs.

Visual Studio Code (VS Code)

During the web development process in VS Code, we encountered several challenges while debugging the code. Here are some of them we faced during app development phase:

- Syntax Errors such as
  - Missing colons

    ```
    def infer_uploaded_webcam (conf, model)
    ```
    instead of
    ```
    def infer_uploaded_webcam (conf, model):
    ```
  - Incorrect indentation

    ```
    if boxes is not None:
    count_dict = {}
    ```
    instead of
    ```
    if boxes is not None:
        count_dict = {}
    ```

Mismatched quotes

```
st.write("Probability:', conf)
```

- Error loading video: *too many values to unpack (expected 4)*

  Troubleshooting solution:

  - Verify the video capture: Ensure that the video capture object (cap) is properly initialized and opened. You can add a check after cap = cv2.VideoCapture(tfile.name) to verify if the capture was successful:

    ```
    if not cap.isOpened():
        raise ValueError("Failed to open video capture.")
    ```

  - Check the returned values: Add some print statements to check the values returned by cap.read():

    ```
    ret, frame = cap.read()
    print(ret, frame)
    ```

- Unsupported image type

  Troubleshooting solution:

  The Ultralytics YOLO model's predict method expects the file path of the video or image file to be processed. Thus, we need to save the uploaded file to a temporary file and then pass the file path to the predict method:

  ```
  tfile = tempfile.NamedTemporaryFile(delete=False)
  tfile.write(source_video.read())
  video_path = tfile.name
  ```

- Missing or Incorrect Installation

  Troubleshooting solution:

  The "supervision" module was not installed in my virtual environment.

- Incorrect import statement: *from PIL import Image, imageio*

  Troubleshooting solution:

  Imageio module imported from the following command (not part of PIL library):

  ```
  import imageio
  ```

To fix the bugs encountered during debugging, we made necessary modifications to the code or configuration.

Debugging techniques

- Exception Handling

  Enable debugging to catch any exceptions that might occur during execution. For example, if an exception is thrown when performing a detection process, we can examine the exception message:

```
except Exception as ex:
    st.write("Error occurred during inference.")
    st.write(ex)
```

- Print Statements

  We inserted print statements strategically throughout the code to display the values of variables, execution flow, or any relevant information:

```
st.write(f"The code was executed {execution_count} times.")
```

By utilizing these debugging techniques, making necessary code modifications, we were able to identify and resolve the bugs encountered during Streamlit application development in VS Code.

Weekly Progress

We have diligently tracked the progress of our project week by week, ensuring its smooth execution and timely completion. In this section, we will provide a detailed overview of our project management activities, challenges encountered, and plans for the upcoming weeks. Each week is presented in a tabular format, highlighting the week number, corresponding dates, completed activities, challenges faced, and the planned activities for the following week. Additionally, a Gantt chart visualizes the project timeline (refer to Figure 11), providing a comprehensive view of the project's progress and milestones. Through meticulous project management and effective planning, we have navigated challenges and remained on track to achieve our project goals (refer to Tables 2-9).

Table 2.

Week 1 project management.

Week 1 (8/05 - 14/05)	
Objectives	Data Collection
Activities completed	1. Filmed 10 videos during different times of a day: down, very bright sun lights, sunset. 2. Getting familiar with Roboflow functionalities
Challenges	**Solutions**
1. Consistency & Quality in Data Collection 2. Familiarizing with Roboflow functionalities	1. Maintain consistent camera settings, angles, distances. 2. Watching tutorials, reading relevant documentation.
Next Week's objectives	1. Start testing pre-trained YOLOv8 models 2. Collected data to be annotated 3. Learn about different augmentation techniques 4. Learn about evaluation metrics from YOLOv8 models' performance

Table 3.

Week 2 project management.

Week 2 (15/05 - 21/05)	
Objectives	1. Data cleaning & preprocessing 2. Roboflow notebooks
Activities completed	1. Collected data was annotated 2. Roboflow dataset ready for YOLOv8 inference was created 3. Roboflow notebook executed with ByteTrack, Supervision functionalities performed well.
Challenges	**Solutions**

Evaluation metrics	`print(f"map50-95: {metrics.box.map}")`
	`print(f"map50: {metrics.box.map50}")`
	`print(f"map75: {metrics.box.map75}")`
	At this point I managed to get only mAP metrics to be obtained to evaluate the performance.

Next Week's objectives	1. Train a YOLOv8 on one bright video dataset 2. Test & evaluate new model on the second bright video

Table 4.

Week 3 project management.

Week 3 (22/05 - 28/05)	
Objectives	Model Training
Activities completed	1. A new model was compiled and trained in two sessions on bright video 2. Custom-trained model was tested and evaluated on the second bright video
Challenges	**Solutions**
1. Custom Trained Model Prediction Function 2. Compatibility Issues with YOLOv8 Version (the bounding boxes, confidence scores, and associated classes failed to be displayed).	1. Issue stemmed from the compatibility between the YOLOv8 version used in the previous notebook and our current mode.Upgrading our Ultralytics YOLOv8 version to the most recent one 2. Scratch off the Bytetrack and Supervision libraries, which were causing compatibility conflicts
Next Week's objectives	Retest new model to ensure model's consistency

Table 5.

Week 4 project management.

Week 4 (29/05 - 4/06)	
Objectives	Model Validation
Activities completed	Acquiring Probability (Confidence Score), Bounding Box Coordinates, and Associated Classes from Inference
Challenges	**Solutions**
1. Extracting Bounding Box Coordinates 2. Difficulties understanding Ultralytics Documentation	1. Previously, the bounding box coordinates were displayed as tensor vectors, to fix that,I researched and experimented with different methods to extract the bounding box coordinates correctly 2. I explored alternative sources such as online forums, community discussions, and relevant code repositories
Next Week's objectives	Getting familiar Streamlit platform

Table 6.

Week 5 project management.

Week 5 (5/06 - 11/06)	
Objectives	Performance tuning App Development
Activities completed	1. Reassessing the dataset by creating new versions with a broader range of data augmentation techniques. 2. Testing the custom-trained model on the updated dataset to evaluate its performance and accuracy. 3. Cloning a repository and studying it to understand its structure and make necessary modifications to meet specific requirements.
Challenges	**Solutions**
Understanding the structure, dependencies, and functionality of the cloned repository	Performing analysis of the repository's documentation, source code, and related resources, seeking guidance from relevant documentation
Next Week's objectives	Streamlit App

Table 7.

Week 6 project management.

Week 6 (12/06 - 18/06)	
Objectives	Building the Streamlit app
Activities completed	1. Implementing the functionality for uploading images and performing object detection 2. Adding YOLOv8 models to the project 3. Displaying the results (the detected objects, their associated class names, confidence scores, and bounding box coordinates, class instances count) 4. Testing and debugging the application
Challenges	**Solutions**
Failed to push VS Code to GitHub repo due to the large file size	Created a .gitignore file to exclude heavy files, such as actual models themselves, virtual environment files, compiled files (ipython_config.py)
Next Week's objectives	Video files inference

Table 8.

Week 7 project management.

Week 7 (19/06 - 25/06)	
Objectives	Testing & Debugging
Activities completed	1. Debugging Process 2. Code Modification 3. Testing Iterations 4. Documentation and Reporting
Challenges	**Solutions**
Video inference takes very long time to be processed	1. Use smaller model 2. Downgrading the resolution 3. Downsizing number of frames per second
Next Week's objectives	Presentation slides and Documentation

Table 9.

Week 8 project management.

Week 8 (26/06 - 30/06)	
Objectives	Project wrap-up
Activities completed	1. Final Testing and Bug Fixing: 2. Documentation and Report Writing. 3. Code Refactoring and Optimization. 4. Project Presentation. 5. Project Evaluation and Lessons Learned.
Challenges	**Solutions**
1. Time Constraints 2. Documentation Complexity 3. Code Refactoring Challenges 4. Evaluation and Lessons Learned	1. Prioritized tasks ,created a detailed schedule to meet deadlines. 2. The documentation process was broken down into smaller sections. Outlined the report to ensure coherence. Seeked feedback from supervisor. 3. Identified areas of improvement. 4. Reviewed project objectives, measured achieved results, and solicited feedback from mentor . Documented insights, challenges and recommendations for future projects.

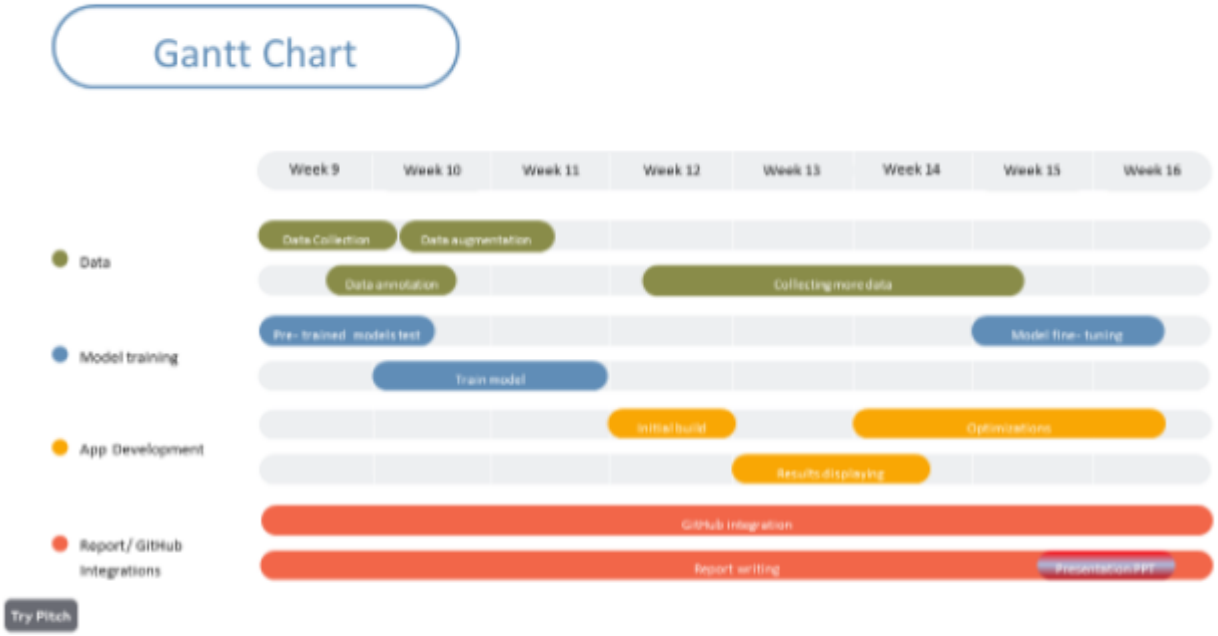


Figure 11.

Project timeline gantt chart.

Discussion

Model training and validation

The two training sessions provided different results. In the initial session, the model was unable to detect any instances of the classes in the validation dataset. This could be due to the limited number of epochs for which the model was trained. In contrast, during the second session, with the model trained over 100 epochs, it was able to detect some instances in the validation set. This demonstrates the importance of providing adequate training for the model to learn and generalize.

The precision, recall, and mAP were computed for the validation dataset and provided insights into the model's performance. Notably, the 'motorcycle' class had the highest precision and mAP, suggesting that the model was particularly effective in identifying motorcycles.

The model's processing speed for various stages, including preprocessing, inference, loss calculation, and postprocessing, was also noted. This information

is important for understanding the model's efficiency and potential for real-time applications.
The validation process, carried out with a separate set of images, assessed the model's ability to generalize and detect vehicles in unseen images. This step is crucial to evaluate the model's practical applicability. The metrics used in this stage, such as precision, recall, and mAP, helped understand the strengths and weaknesses of the model.

Comparing pre-trained largest model and custom-trained

Model Size and Complexity
The first model has 168 layers and about 3 million parameters. The second model, on the other hand, is significantly larger with 268 layers and roughly 68 million parameters.

Dataset
The first model has been trained and validated on a dataset with 40 images, containing 18 instances of 3 classes: 'bus', 'car', and 'motorcycle'. The second model was trained and validated on a dataset with 128 images and includes 929 instances of a significantly larger variety of classes (over 60 classes are listed, including 'person', 'bicycle', 'car', and many others).
Performance
The first model achieved an average precision (AP) at IoU > 0.50 (mAP50) of 0.54 and an average precision from IoU 0.50 to 0.95 (mAP50-95) of 0.223 across all classes. The second model achieved a mAP50 of 0.977 and a mAP50-95 of 0.93 across all classes.
Inference Speed
The first model has an inference speed of 4.7ms per image, while the second model has an inference speed of 22.2ms per image.
In summary, pretrained model is larger, trained on a more diverse dataset, has better performance metrics, but is slower in terms of inference speed compared to our model. It's important to note that these differences do not inherently make one model better than the other, as the best model to use would depend on the specific use-case, requirements, and constraints.

During the preparation of my model for demonstration on the Streamlit app, I conducted performance evaluations focusing on detecting bright light images, which formed the basis of my model's training. To my delight, the results were remarkable. The performance of my pre-trained model proved to be nearly on par with the largest YOLOv8 model, and in some instances, it even surpassed expectations by delivering significantly superior outcomes. This observation highlights the effectiveness of my model in accurately detecting objects in challenging scenarios, particularly those characterized by bright lighting conditions. The promising performance of the pre-trained model reinforces its suitability for a wide range of practical applications, showcasing its potential impact in real-world scenarios.

Conclusion

In conclusion, the project "Vehicle Tracking in Challenging Scenarios using YOLOv8" has been an exciting and insightful journey. Throughout this project, we have explored and implemented various techniques and methodologies to tackle the complex task of vehicle tracking in challenging scenarios. The use of YOLOv8, along with custom training and data augmentation techniques, has allowed me to achieve accurate and efficient object detection and tracking.

The project's success has been driven by the rigorous data collection process, which involved filming multiple videos under different lighting conditions and scenarios. This diverse dataset has proven invaluable in training and validating the custom YOLOv8 model. The integration of Streamlit into the application development phase has provided a user-friendly and interactive platform for users to upload images, videos, and webcam streams for real-time vehicle tracking.

Throughout the project, we encountered various challenges that required careful problem-solving and debugging. These challenges ranged from issues with model compatibility and video processing to debugging code for web development. However, with perseverance and the use of effective debugging techniques, we were able to overcome these challenges and ensure the smooth functioning of the application.

The project has not only allowed us to gain practical experience in deep learning and computer vision but also taught us valuable lessons in project management, testing, and debugging. It has reinforced the importance of thorough documentation, version control, and collaboration tools like GitHub.

The initial hi-fi prototyping has been successfully implemented, with some minor differences in the interfaces that have actually improved the user experience. While the original prototype intended to display a processed image in a separate tab or window, the new design has taken a different approach. Now, the processed image is displayed face to face with the original image, allowing for a better comparison and analysis of the detection results. This change not only streamlines the user flow but also enhances the overall friendliness and intuitiveness of the interface. Users can easily assess the effectiveness of the detection algorithm by visually comparing the two images side by side. This modification in the design contributes to a more user-centric and engaging experience, empowering users to make informed decisions based on the detection outcomes. Figures 12 and 13 provide a vizual comparison.

.

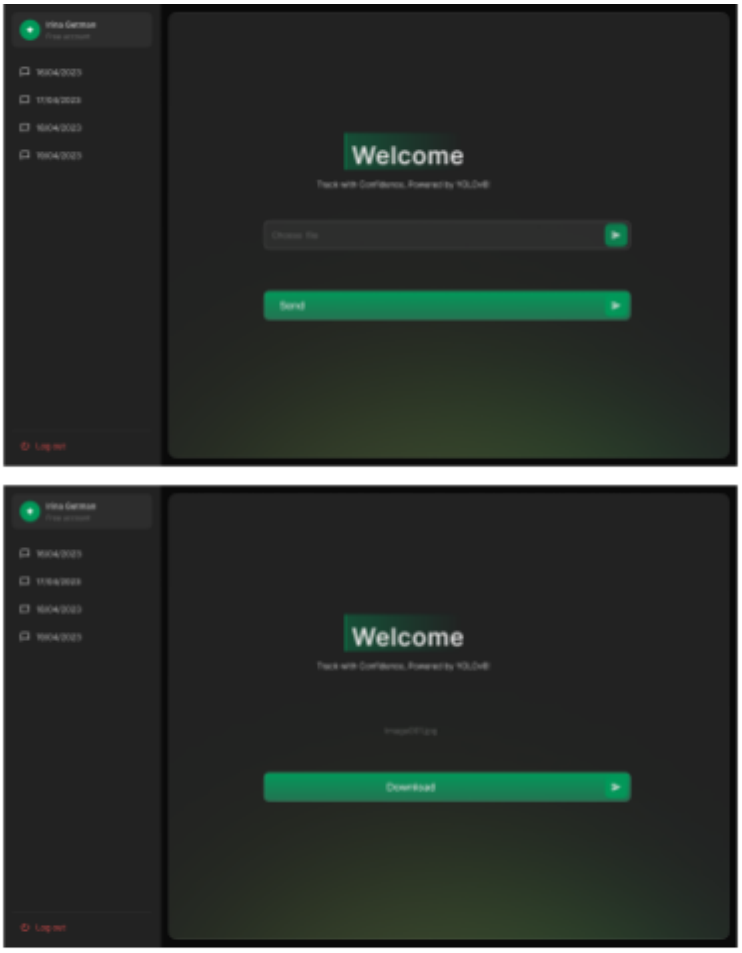


Figure 12.

Initial UI proposal.

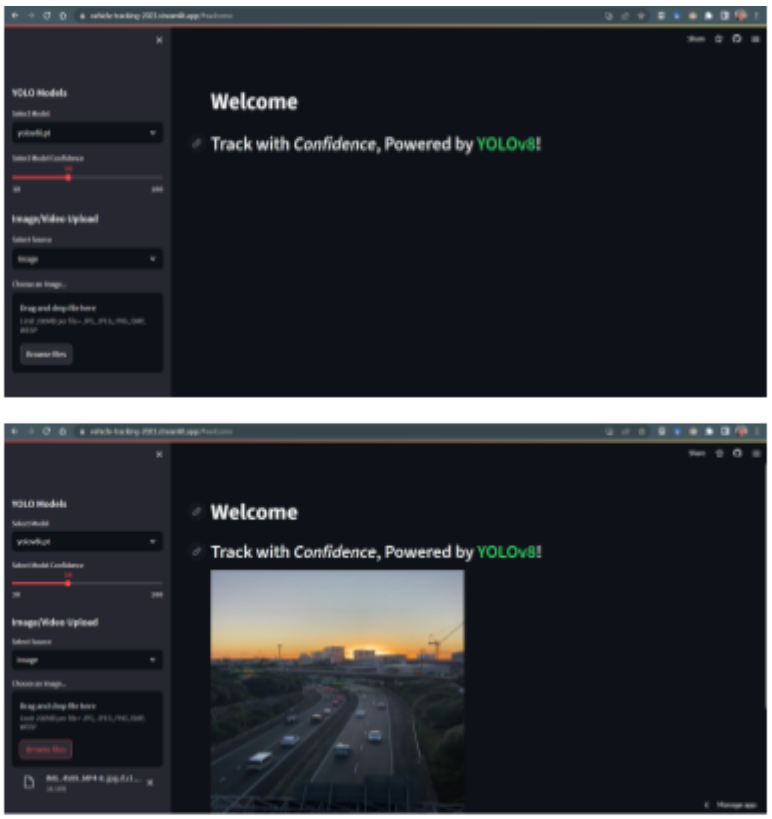


Figure 13.
Streamlit UI landing page and 'uploaded image' page.

In conclusion, the project "Vehicle Tracking in Challenging Scenarios using YOLOv8" has provided a solid foundation in computer vision and deep learning techniques. It has been a rewarding experience, and we are confident that the knowledge and skills gained from this project will serve as a strong stepping stone for future endeavors in the field of computer vision and object tracking.

Next steps

Looking ahead, there are several areas that could be explored to further enhance the user experience and expand the system's capabilities. One potential area is the implementation of user logging and session management. This would allow users to log in and access their previous sessions, providing a seamless workflow and facilitating the review of past work.

Another valuable addition would be the development of a function to record detected videos and enable playback with the displayed results. This feature

would enable users to revisit and analyze video footage along with the associated detection outcomes, providing a comprehensive understanding of the tracking performance over time.

In addition, future improvements can be made in the area of results visualization. Enhancing the visual representation of the detection outcomes, such as through the use of interactive visualizations or advanced graphical techniques, would enable users to better interpret and analyze the tracked objects.

Visualization

In order to enhance the user experience and provide a comprehensive visual representation of the vehicle tracking process, several visualization techniques can be implemented in future steps. These may include:

Progress Bar: Enhancing the progress bar during video inference to visualize the progress of object detection on each frame. This visual indicator will give users a sense of how much of the video has been processed.

Graphical Representation of Count Distribution: Creating a bar chart or line graph to represent the count distribution of different object types over time. This visualization can show the fluctuations and trends in vehicle counts, allowing users to analyze patterns and changes in traffic.

Heatmap of Vehicle Density: Generating a heatmap to visualize the density of vehicles in different regions of the video frames. This visualization can provide insights into high-traffic areas or congestion patterns.

These visualization techniques can greatly enhance the understanding and analysis of the vehicle tracking process, providing users with valuable insights and a more interactive experience.

Furthermore, as part of future improvements, optimizing the custom-trained model's performance can be explored. This can include fine-tuning the model's hyperparameters, exploring more efficient network architectures, or utilizing hardware acceleration techniques such as GPU processing. By optimizing the

model's performance, the challenge of video inference taking a long time to process can be addressed, resulting in faster and more efficient vehicle tracking.

In summary, the addition of proposed functions, advanced visualizations and the optimization of the custom-trained model's performance are promising avenues for further improvement in the vehicle tracking system. These enhancements will not only enhance the user experience but also provide more accurate and efficient results, making the system more robust and effective in real-world scenarios.

References

Bharath K. (2022, September 27). *Beginners Guide to Streamlit for Deploying your Data Science Projects*. Medium; Towards Data Science. https://towardsdatascience.com/beginners-guide-to-streamlit-for-deploying-your-data-science-projects-9c9fce488831

Create an app - Streamlit Docs. (2023). Streamlit.io. https://docs.streamlit.io/library/get-started/create-an-app

JackDance. (2023, May 22). *JackDance/YOLOv8-streamlit-app: 🔥🔥🔥 Use streamlit framework to increase yolov8 front-end page interaction function*. GitHub. https://github.com/JackDance/YOLOv8-streamlit-app

McDonald, A. (2022, May 23). *Getting Started With Streamlit Web Based Applications*. Medium; Towards Data Science. https://towardsdatascience.com/getting-started-with-streamlit-web-based-applications-626095135cb8

Nielsen, N. (2023). YOLOv8 Model Training and Deployment for Object Detection with Real-time Webcam [YouTube Video]. In *YouTube*. https://www.youtube.com/watch?v=TRMZCsBfX78

notebooks/notebooks/how-to-track-and-count-vehicles-with-yolov8.ipynb at main · roboflow/notebooks. (2023). GitHub. https://github.com/roboflow/notebooks/blob/main/notebooks/how-to-track-and-count-vehicles-with-yolov8.ipynb

Surya Remanan. (2023, March 31). *Damaged Car Parts Detection using YOLOv8n: From Data to Deployment*. Paperspace Blog; Paperspace Blog. https://blog.paperspace.com/damaged-car-parts-detection-using-yolov8n-an-end-t/

ultralytics. (2023, May 20). *How to get names of predicted objects · Issue #2716 · ultralytics/ultralytics*. GitHub. https://github.com/ultralytics/ultralytics/issues/2716

Appendix

Figure A1.
Demonstration of Shear Technique in Data Augmentation using Roboflow.

Figure A2.
Demonstration of 90° Rotation Technique in Data Augmentation using Roboflow.

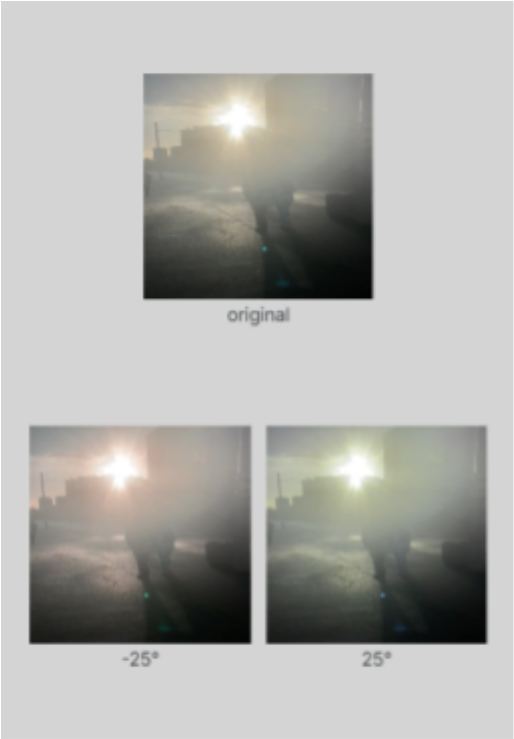


Figure A3.
Demonstration of Hue Technique in Data Augmentation using Roboflow.

Figure A4.
Demonstration of Saturation Technique in Data Augmentation using Roboflow.

Figure A5.
Demonstration of Brightness Technique in Data Augmentation using Roboflow.

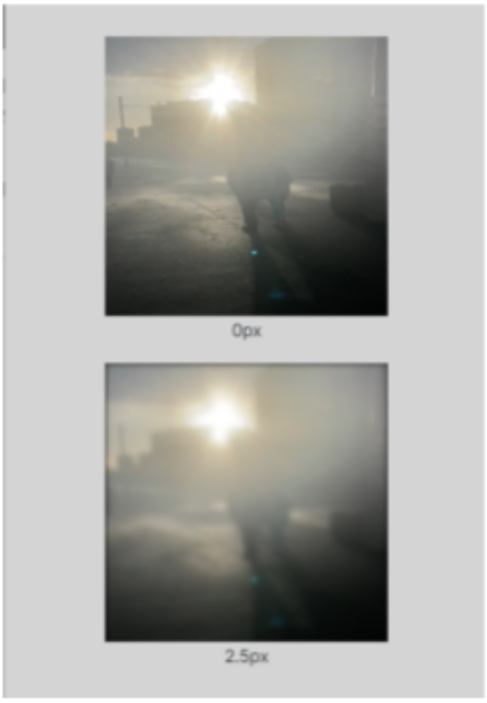


Figure A6.
Demonstration of Blurring Technique in Data Augmentation using Roboflow.

Figure A7.

Comparison of Original and resized images for YOLOv8 training and testing.