UNIVERSITY OF CRAIOVA
FACULTY OF AUTOMATION, COMPUTERS AND ELECTRONICS

DEPARTMENT OF COMPUTERS AND TECHNOLOGY INFORMATION

GPU-BASED VIDEO/IMAGE ANALYSIS APPLICATION

ALEXANDRU MANDA

SUPERVISOR

Asist. Dr. IVĂNESCU CONSTANTIN RENATO

JULY 2023

CRAIOVA

SUMMARY OF THE PROJECT

The application of machine learning to Deep Learning models has achieved notable success in the fields of computer vision (CV), natural language processing (text), and video/image recognition. This work aims to present a traffic analysis system that utilizes machine learning and computer vision technology to interpret traffic patterns.
Motivated by the desire to improve traffic management and planning, I chose to work on this project, which combines my interest in machine learning and computer vision. The system is based on the YOLOv8 model, a state-of-the-art object detection model known for its accuracy and speed.

I used a range of technologies, including Python, Flask, Docker, MongoDB, and OpenCV, to develop this system. During the development process, I encountered several challenges, such as optimizing the model's performance and ensuring accuracy under variable traffic conditions. However, I overcame these obstacles by fine-tuning the model's parameters and using a comprehensive dataset for training, which I created manually by annotating images in Roboflow and training the model using GPU resources on Google Colab.

The results of my project include the development of a system capable of analyzing traffic in real-time and providing actionable insights. The system includes two main scenarios: people counting and traffic analysis. Both scenarios can be used in real-time from various sources, such as webcams, YouTube links, UDP streams, and RTSP streams.

One of the unique aspects of my project is its ability to modify tracker parameters in real-time by providing updates from the MongoDB database, using the Abstract Factory Design Pattern. Through this project, I learned valuable lessons about applying machine learning and computer vision in a real-world context. This experience helped me improve my technical skills and gain a better understanding of the challenges and opportunities presented by AI technology in the transportation sector.

Keywords: traffic analysis, machine learning, computer vision, YOLOv8, traffic management.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The advent of technology has brought about a significant transformation in various sectors, including traffic management. The traditional methods of traffic management are gradually being replaced by more sophisticated and efficient systems. That's where Artificial Intelligence plays its role. In combination with GPU resources, it can efficiently provide powerful real-time streaming performance, since the CPU multithreading methods are far less performant than the GPU, since it has access to more GPU CUDA cores depending on the architecture and CUDA version.

The Traffic Analytics system we propose is built upon the YOLOv8 model, a state-of-the-art object detection model known for its speed and accuracy. The system is designed to analyze traffic patterns and provide useful insights that can be used to improve traffic management and planning by efficiently providing key metrics, such as: speed estimation, vehicle counting for each class (car, truck, bus, motorcycle).

1.1 Scope

The scope of this project encompasses several key areas in the field of traffic analysis using computer vision and machine learning techniques. The primary focus is on real-time traffic analysis, which involves the use of advanced techniques such as image segmentation, object detection, and object tracking. These techniques are applied in the context of a traffic analysis system, which aims to provide accurate and timely traffic data for improved traffic management and planning.

By leveraging the power of machine learning and computer vision, the system can analyze traffic in real-time, providing instant feedback and allowing for quick decision making. This can be particularly useful in situations where immediate action is required, such as in the case of traffic congestion, but also in case of vehicle counting purposes.

The project will dive into the specifics of these techniques, exploring how they work and how they can be applied to real-world traffic scenarios. This includes a detailed examination of the YOLO (You Only Look Once) model, a state-of-the-art object detection model that is particularly suited for real-time applications due to its speed and accuracy.

However, it's important to note that while the project covers a broad range of topics, it does not encompass all aspects of traffic analysis or computer vision. For instance, the project will not delve into the detailed mathematical and computational aspects of these models. Nor will it compare YOLO with every existing model in the field. These topics, while important, are beyond the scope of this project.

Moreover, while the project discusses the use of machine learning techniques for traffic analysis, it does not cover the full breadth of machine learning applications in this field. For example, the project does not cover the use of reinforcement learning for traffic signal control or the use of generative models for traffic prediction.

The limitations of the techniques discussed in this project will also be acknowledged. For instance, while the YOLO model is highly effective for object detection, it has its limitations. According to a study by Choi et al. [1], the YOLO model can struggle with small object detection and is sensitive to object scale.

Similarly, real-time traffic analysis presents its own set of challenges. Real-time traffic analysis requires high computational resources and can be affected by factors such as lighting conditions and occlusion events. That's why, in our system we make use of GPU resources to run inference of the YOLO model on the GPU. The project presents also the improvements of the GPU vs CPU multithreaded solutions with its own drawbacks and improvements.

In addition, the project will touch upon the potential of hardware acceleration for deep learning models. Hardware acceleration is a method of boosting the performance of certain processes, and in the context of our project, it can significantly speed up the training and inference process of deep learning models like YOLO. This is particularly relevant when dealing with real-time traffic analysis, where speed and efficiency are of utmost importance.

In conclusion, the scope of this project is to develop a real-time traffic analysis system using advanced computer vision and machine learning techniques, with a particular focus on the YOLO model. While the project does not cover all aspects of traffic analysis and computer vision, it aims to provide a comprehensive overview of the key techniques and challenges in this field. The project is designed to be a stepping stone towards more efficient and effective traffic management systems, contributing to safer and more convenient travel experiences for all road users.

[1] Choi, J., Chun, D., & Oh, H. (2020). Object Detection Based on YOLOv3 and Distance Measurement Using Depth Camera. In 2020 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1205-1207). IEEE.

1.2 Motivation

The motivation behind this project stems from the need for more efficient traffic management systems. As urban areas continue to grow, so does the need for effective traffic management. Traditional methods of traffic analysis are often unable to keep up with the increasing demand, leading to inefficient traffic management and planning [2].

The advent of machine learning and computer vision technologies presents an opportunity to revolutionize the way we approach traffic management. By harnessing these advanced technologies, we can develop systems capable of real-time traffic analysis, offering a significant improvement over traditional methods. Real-time analysis allows for immediate response to changing traffic conditions, leading to more efficient use of infrastructure and resources, and ultimately contributing to safer and more convenient travel experiences for all road users [3].

A crucial aspect of traffic management is the ability to accurately count and track vehicles. This is where the power of computer vision and deep learning models like YOLO (You Only Look Once) come into play. These models can detect and count vehicles in real-time, providing valuable data for traffic management systems. The ability to quantify vehicles on the road can lead to more accurate traffic predictions and better planning, which in turn can reduce congestion and improve overall traffic flow.

Moreover, with the rise of autonomous vehicles, the need for precise vehicle counting and tracking becomes even more critical. Autonomous vehicles rely heavily on accurate data about their surroundings to navigate safely. Therefore, the development of robust vehicle counting systems can contribute significantly to the advancement of autonomous driving technologies.

By developing a system that can analyze traffic in real-time, we can significantly improve traffic management and planning. The Traffic Analytics system is a step towards this goal. By leveraging the power of machine learning and computer vision, the system provides a more efficient and accurate way of analyzing traffic patterns, paving the way for improved traffic management and planning [4].

---

[2] Wang, Y., Zheng, L., & Xue, M. (2019). Transport efficiency of urban arterial road traffic: A case study of Shanghai. Sustainability, 11(3), 681

[3] Liu, H., Tian, Z., Li, Y., Zhang, M., & Bigham, J. (2020). Real-time and Predictive Traffic Management Using Artificial Intelligence: A Survey. arXiv preprint arXiv:2001.10993.

[4] Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767.

The potential impact of such a system is immense. Improved traffic management can lead to reduced congestion, lower emission levels, improved road safety, and enhanced urban planning. By exploring and developing these advanced traffic analysis techniques, this project contributes to the ongoing efforts to improve urban living and transportation.

The importance of traffic management systems is further highlighted by the increasing complexity of urban environments. With the rise of autonomous vehicles and smart city initiatives, the need for advanced traffic management systems that can handle complex traffic scenarios is more critical than ever.

Moreover, the advent of large-scale driving datasets has opened up new opportunities for traffic management. These datasets, which contain detailed information about traffic behaviors, can be used to create digital twins of complex traffic scenarios in simulation. This data-driven approach can provide more accurate and realistic traffic management solutions [5].

Furthermore, the use of advanced machine learning techniques, such as deep learning, has shown great promise in traffic management. Deep learning models can process large amounts of data and learn complex patterns, making them ideal for traffic forecasting. However, deploying these models in real-time applications can be challenging due to computational and memory constraints. Therefore, research into efficient model deployment techniques is an important aspect of this project [6].

In conclusion, the motivation for this project lies in the intersection of several important trends: the growing complexity of urban environments, the availability of large-scale driving datasets, and the potential of machine learning techniques for traffic management. By addressing these challenges, this project aims to contribute to the development of more efficient and effective traffic management systems.

[5] Haoyi Xiong et all "ScenarioNet: Open-Source Platform for Large-Scale Traffic Scenario Simulation and Modeling". Link

[6] Yu Zheng, Junbo Zhang, Dekang Qi. "Deep Spatio-Temporal Residual Networks for Citywide Crowd Flows Prediction" Link

1.3 Theme overview

The field of computer vision has seen significant advancements in recent years, particularly in the areas of image processing, object detection, and object tracking. It involves manipulating or processing an image to enhance it or extract useful information from it. This process can be as simple as adjusting brightness and contrast or as complex as identifying objects within an image, visual enhancement, noise removal, feature extraction and more. Image processing forms the backbone of computer vision, a field that enables computers to gain a high-level understanding from digital images or videos[7].

One of the critical aspects of image processing is image segmentation, which involves dividing an image into multiple segments or 'regions' of interest. This technique is essential in isolating specific areas of an image for further analysis or processing. Image segmentation has found applications in various fields, including medical imaging, autonomous driving, and surveillance systems[8]. It forms the basis of object detection and tracking, where objects within these regions are identified and followed over time by identifying it's underlying shape.

Object detection, on the other hand, is a computer vision technique for locating instances of objects in images or videos. This is a more complex task than image segmentation as it requires the correct classification of the object as well as accurately determining its location within the image.It is a key technology behind applications like video surveillance, image retrieval systems, and advanced driver assistance systems (ADAS). Object detection algorithms typically leverage machine learning or deep learning to produce meaningful results [9].

Object tracking, another crucial aspect of computer vision, is the process of locating a moving object (or multiple objects) over time using a camera. This is particularly challenging in situations where the tracked object changes shape or scale, moves rapidly, or when it is occluded by other objects and it has

[7] Gonzalez, R.C., Woods, R.E., 2002. Digital Image Processing (3rd Edition). Prentice-Hall, Inc., USA

[8] Pham, T. D., Xu, C., & Prince, J. L. (2000). Current methods in medical image segmentation. Annual review of biomedical engineering, 2(1), 315-337

[9] Redmon, J., Divvala, S., Girshick, R., Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788)

a variety of uses in several domains, including human-computer interaction, security and surveillance, medical imaging, and traffic control [10].

One of the popular libraries used for image processing and computer vision tasks is OpenCV (Open Source Computer Vision Library). OpenCV provides a robust and flexible framework for implementing a wide range of image processing and machine learning tasks. It was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products [11]. It has more than 2500 optimized algorithms for image and video analysis.

Deep learning has revolutionized the field of object detection, with models like YOLO (You Only Look Once) leading the charge. YOLO applies a single neural network to the full image, predicting bounding boxes and class probabilities directly from full images in one evaluation. This approach makes YOLO incredibly fast while still maintaining impressive accuracy [12].

However, deep learning models, including YOLO, require significant computational resources for training and inference. This requirement has led to the increasing importance of GPU processing in deep learning. GPUs, with their high parallel processing capabilities, are well-suited to the heavy computational needs of deep learning, making them a vital component of any deep learning setup [13]. This is because GPUs are designed to quickly render high-resolution images and video concurrently, making them ideal for image processing tasks .

The theme of this project, which involves person counting detection, speed estimation for vehicles, and vehicle detection, is a complex task that requires the integration of these various aspects of computer vision. The need for such a system is evident in various fields, including traffic management, surveillance, and autonomous driving.

The advancements in this field, powered by deep learning and GPU processing, have opened up new possibilities and applications that were previously unimaginable.

---

[10] Yilmaz, A., Javed, O., Shah, M., 2006. Object tracking: A survey. ACM computing surveys (CSUR), 38(4),

[11] Bradski, G., 2000. The OpenCV Library. Dr. Dobb's Journal of Software Tools.

[12] Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767.

[13] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. Queue, 6(2), 40-53.

2 ANALYSIS AND DESIGN OF THE APPLICATION-

2.1 Project directory structure

In the next chapters we will present the directory structure of the app for BackEnd services and FrontEnd implementation alongside with the basic configurations of the app.

2.1.1 Backend structure

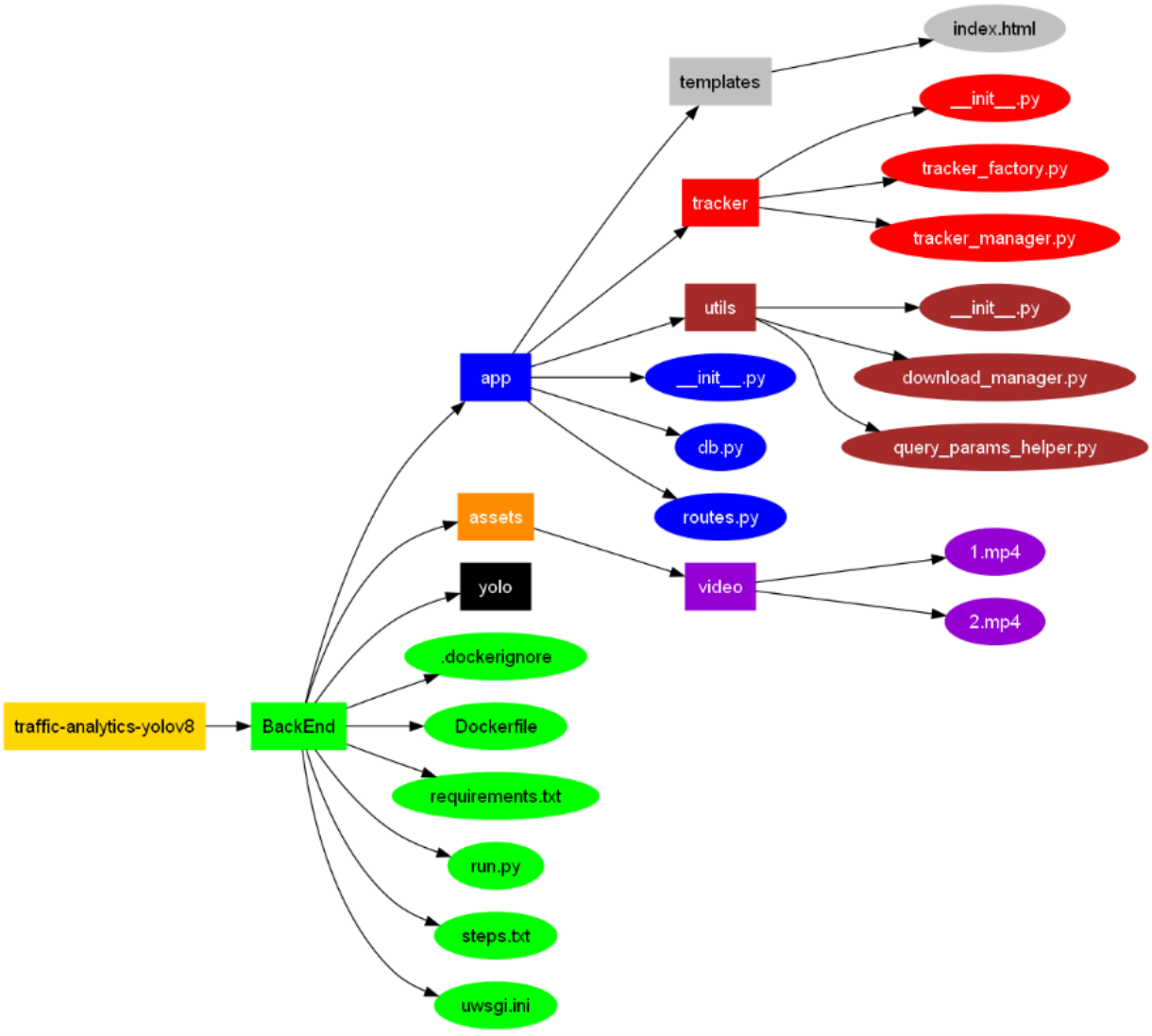

Figure 1 - Project structure backend

The project „traffic-analytics-yolov8" is composed into multiple submodules and directories which I will explain in detail in the next paragraphs.

The BackEnd application, built with Flask, communicates with the React-based FrontEnd via HTTP requests. The core logic for the YOLOv8 object detection system resides in the BackEnd, powering the real-time tracking functionalities of our application.

The tracker module in the BackEnd is a key component, housing classes that implement the factory pattern. This design allows dynamic loading of classes at runtime from MongoDB, offering flexibility to adapt to different requirements without code modification.

Moreover, this dynamic class loading enables us to alter the constructor parameters for each class, allowing us to customize class behavior based on specific application needs. This adaptability is a significant strength of our BackEnd application, ensuring a responsive and customizable user experience.

Additionally, the dynamic class loading feature provides a level of flexibility that is crucial in the ever-evolving landscape of technology. It allows us to keep up with changes and advancements without the need for extensive code revisions or system overhauls. This feature not only enhances the adaptability of our BackEnd application but also significantly reduces the time and resources required for system updates, leading to improved efficiency and productivity.

Furthermore, the ability to alter constructor parameters for each class gives us the power to fine-tune our application to meet specific user needs or to adapt to changing conditions. This level of customization ensures that our application remains user-centric, providing a tailored experience for each user. This focus on customization and user experience is a testament to our commitment to delivering a high-quality, user-friendly application that meets and exceeds user expectations.

Overall, the project is divided into several modules and packages that can be broadly classified as follows:

A. **BackEnd:** Contains all the server-side code for the application, including yolo implementation. It is divided into several packages:
B. **app:** The main application package which has the following sub-packages:
    a. **templates:** Contains index.html, the main template file used by the Flask app.
    b. **tracker:** Contains modules like tracker_factory.py and tracker_manager.py which presumably handle the tracking functionality in the application.
    c. **utils:** Contains utility modules like download_manager.py and query_params_helper.py.
C. **assets/video:** Contains video files that are used for processing or testing purposes.

Figure 2 - Yolo directory structure

D. **yolo:** Contains the YOLOv8 specific code and resources. It has several sub-packages and modules:

   a. **assets/Images:** Contains images used in the application.

   b. **assets/videos:** Contains video files used for object detection/tracking tasks.

   c. **base:** Contains base classes like base_counter.py and base_tracker.py.

   d. **classes:** Contains classes specific to the object detection tasks like line.py, person_count.py, speed.py, and total_classes_rectangle.py used to estimate the speed, count the objects crossed the lines and display the total count of the classes.

   e. **datasets:** Contains datasets used for object detection tasks like coco128 and highway-cars-object-detection-1 and highway-cars-object-detection-2 annotated manually in RoboFlow.

Top-level files like requirements.txt, run.py, and Dockerfile among others are also present in the BackEnd directory.

2.1.2 Frontend project structure

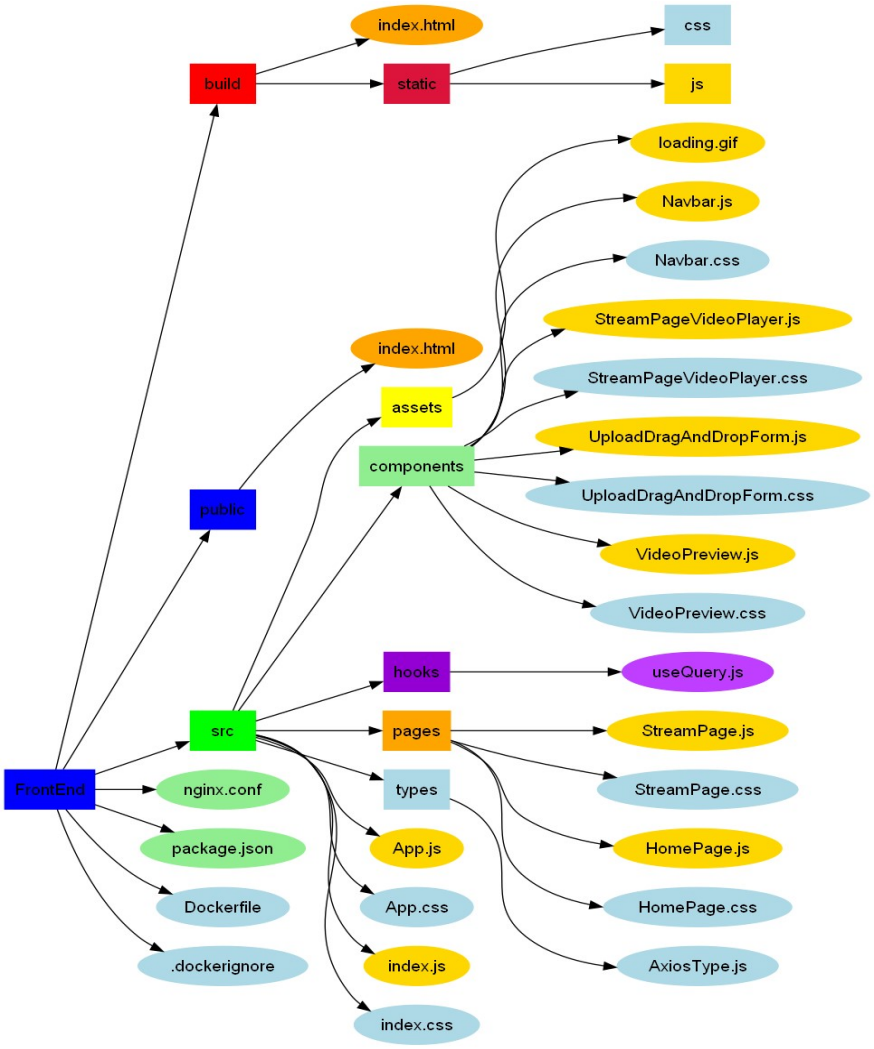

Figure 3 - Frontend project structure

The project structure is organized according to the standard conventions of a React project and is designed to promote clean code and easy navigation for developers.

In the root of the FrontEnd directory, there are two main subdirectories: public and src. The public directory contains static files that are not processed by Webpack, including the index.html file and favicon, logo and etc.

The src directory is where the majority of the application code resides. This includes JavaScript and CSS files, as well as any assets used by the application.

Inside src, there are five subdirectories:

- **Assets** – images, gif, videos served by Webpack/Nginx
- **Components** – reusable React components that can used in multiple places of the app
- **Hooks** – custom hooks which extracts functionality into one hook for simplicity of the code base
- **Pages –** contains components that represents the different pages of our application, built up by using components defined in the Components folder
- **Types –** here resides the AxiosType instance dictionary which we import it to use HTTP services via Axios library.

It's worth noting that the React framework has been designed from the start for gradual adoption. This means you can use as little or as much React as you need, whether you want to get a taste of React, add some interactivity to an HTML page, or start a complex React-powered app[14].

2.2 Product functions

- Provided access to a traffic analytics application where the user can input media source for analysis (youtube link, upload mp4 video or img, udp stream, rstp stream etc..).
- Traffic detection and analysis functionality, where the system can identify and count different types of vehicles in the video feed using the YOLOv8 model.
- Vehicle speed detection analysis, where the system estimates the speed by tracking the vehicle along the frame and calculates the distance traveled in the given time.
- Vehicle dynamic line counting, where the system can receive 2 line coordinates from the frontend and starts counting the total objects who passed those line.
- Person counting functionality, where the YOLOv8 model counts the persons present in the given source (image, video, udp stream, youtube link etc.)

The application will have the following functionalities:

- Traffic Counting (Total count and lines count)
- Vehicle speed estimation
- Person Tracker with counter

[14][REACT23] React Documentation, *Start a New React Project* ,2023, (Source: https://react.dev/learn/installation)

2.3 User characteristics

The main user will be the one who uses the application for traffic analysis. The user can be a traffic management authority, a researcher, or any individual interested in traffic analysis. The user should be able to input the media source and view the analysis results.

2.4 Functional Requirements

ID: FR1

Feature: Video Input

In order to analyze traffic from various video sources

A user should be able to input a video for analysis through different methods.

Scenario: Upload a video file

Given the user wants to analyze a video file

When the user uploads a video file

Then the system should be able to accept and process the video file for analysis

Scenario: Provide a YouTube link

Given the user wants to analyze a YouTube video

When the user provides a YouTube link

Then the system should be able to fetch and process the YouTube video for analysis

Scenario: Specify a UDP or RTSP stream

Given the user wants to analyze a live video stream

When the user specifies a UDP or RTSP stream

Then the system should be able to fetch and process the live stream for analysis

ID: FR2

Feature: Traffic Detection and Analysis

In order to provide users with detailed traffic analysis data

The system should be able to identify and count different types of vehicles in the video feed.

Scenario: Identify and count vehicles

Given a video input

When the system analyzes the video using the YOLOv8 model

Then the system should be able to identify and count different types of vehicles

ID: FR3

Feature: Vehicle Speed Detection

In order to provide users with additional traffic analysis data

The system should be able to estimate the speed of vehicles.

Scenario: Estimate vehicle speed

Given a video input with vehicles

When the system tracks their movement across frames

Then the system should calculate and provide the speed of the vehicles

ID: FR4

Feature: Dynamic Line Counting

In order to analyze traffic flow in specific areas of the video

The system should be able to receive two-line coordinates from the user and start counting the total objects that pass those lines.

Scenario: Count objects passing lines

Given two-line coordinates specified by the user on a video input

When the system starts its analysis

Then the system should count the total objects that pass those lines

And display them back on a stream in the frontend

ID: FR5

Feature: Person Counting

In order to analyze pedestrian traffic

The system should be able to count the number of persons present in the given source.

Scenario: Count persons in a video

Given a video input

When the system analyzes the video using the YOLOv8 model

Then the system should be able to count the number of persons present in the video.

2.5 Performance Requirements

The system's performance is crucial. It shall process video feeds and provide traffic analysis results in real-time or near real-time, irrespective of the video feed's size or complexity. This ensures users receive timely and accurate data.

A key feature is the information link, which shall be prominent and user-friendly. Accessing the information shall require a single click, facilitating quick and easy access to needed information.

The system's response time is highly efficient, it shall not exceed 7 seconds from request to response. This quick response time enhances the system's reliability, ensuring users can trust it to provide timely results.

The system shall be available 24/7, allowing users to access and analyze traffic data anytime. Exceptions are scheduled maintenance periods, during which the system may be temporarily unavailable. Any potential downtime is communicated to users in advance.

The user interface shall be responsive and user-friendly, enabling users to easily input video feeds, view traffic analysis results, and perform other actions without significant delays. This ensures effective and efficient user-system interaction.

Finally, the system is scalable and maintainable, capable of handling an increasing workload, whether processing larger video feeds or serving more users. This scalability ensures the system can adapt over time, meeting the needs of an expanding user base or increasing data volume.

2.6 UML Class design

In this chapter, we will explore UML class diagrams that represent the design of various components within the software system. These diagrams serve as blueprints for understanding the structure and interactions between different classes, facilitating effective communication among stakeholders and guiding the development process.

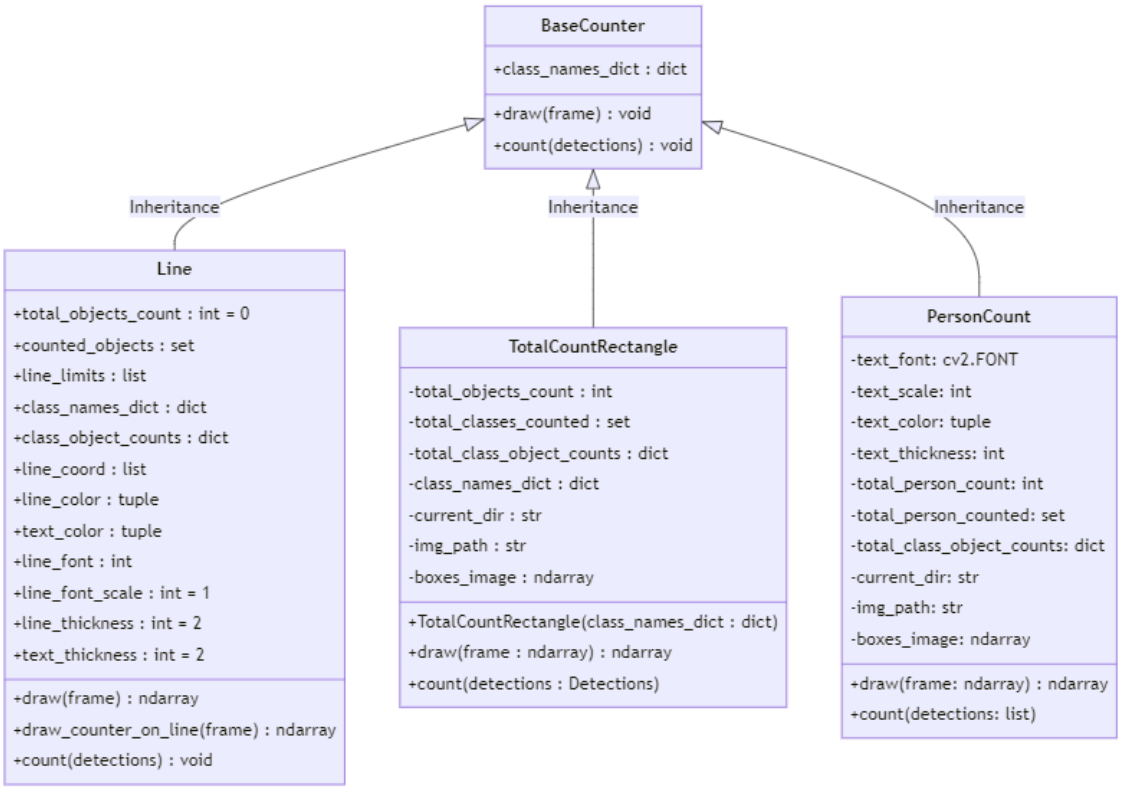

Figure 4 - Base Counter UML class design

The class diagram outlines the structure for object counting. The **BaseCounter** class provides common attributes and methods for counting. **PersonCount** extends **BaseCounter**, focusing on counting persons and tracking related information. **TotalCountRectangle**, another **BaseCounter** extension, represents a rectangular region for counting objects. It has attributes for total counts and class names. The **Line** class extends **BaseCounter**, representing a line used for counting objects. It includes attributes for total counts, line limits, and class names. This class diagram provides a foundation for implementing object tracking and counting functionalities.

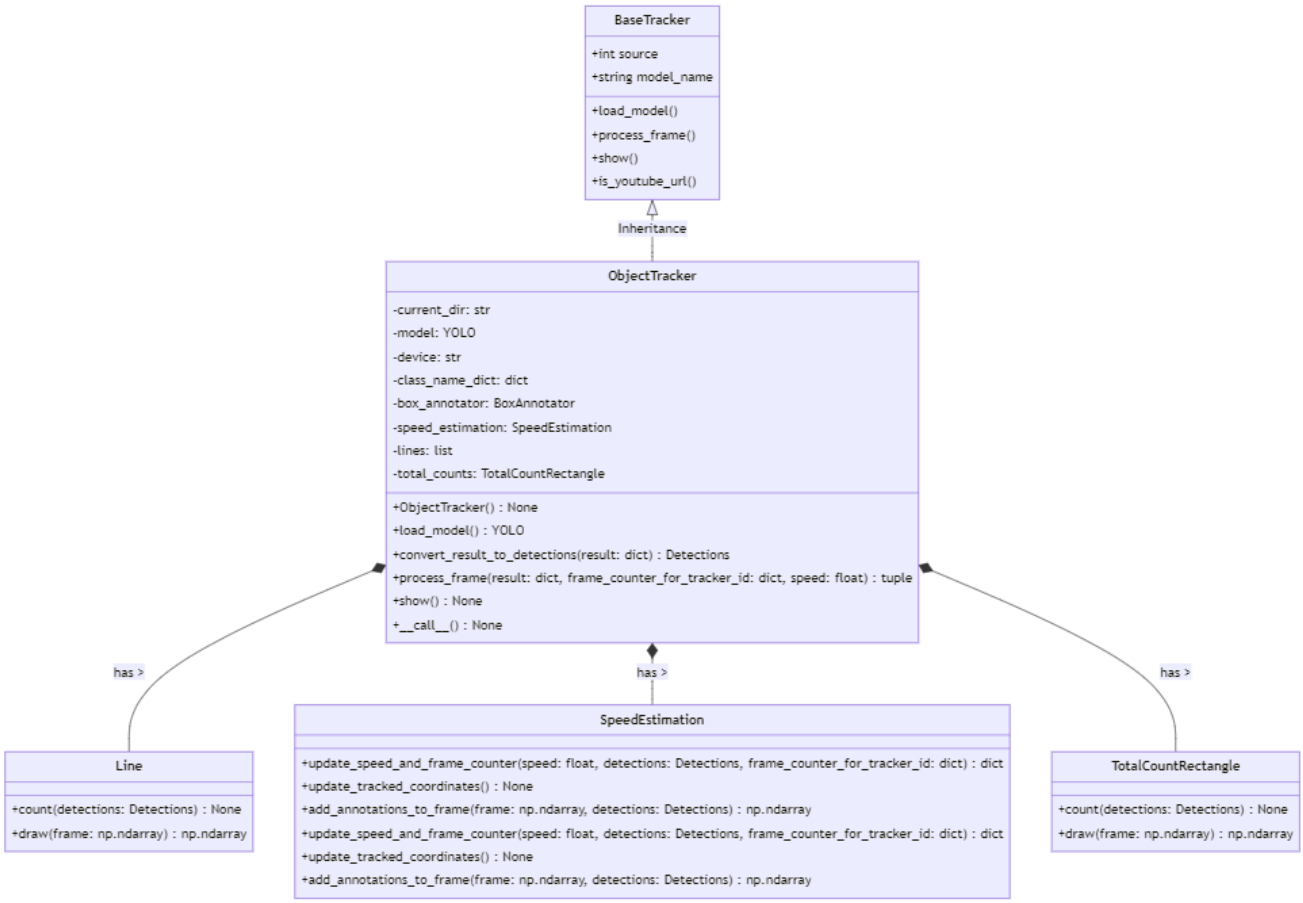

Figure 5 – Car Tracker UML diagram

The class diagram showcases the structure for object tracking and counting. The **BaseTracker** class is central, with attributes like source, **model_name**, and methods for model loading and frame processing. The **ObjectTracker** class extends **BaseTracker**, with attributes for model, device, **class_name_dict**, and methods for model loading, result conversion, frame processing, and result display. It associates with Line, **SpeedEstimation**, and **TotalCountRectangle** classes. **SpeedEstimation** updates speed and frame counter, tracks coordinates, and adds annotations. **Line** counts objects and draws lines on frames. **TotalCountRectangle** counts objects and draws count rectangles on frames. This diagram provides insight into class interactions for object tracking and counting.

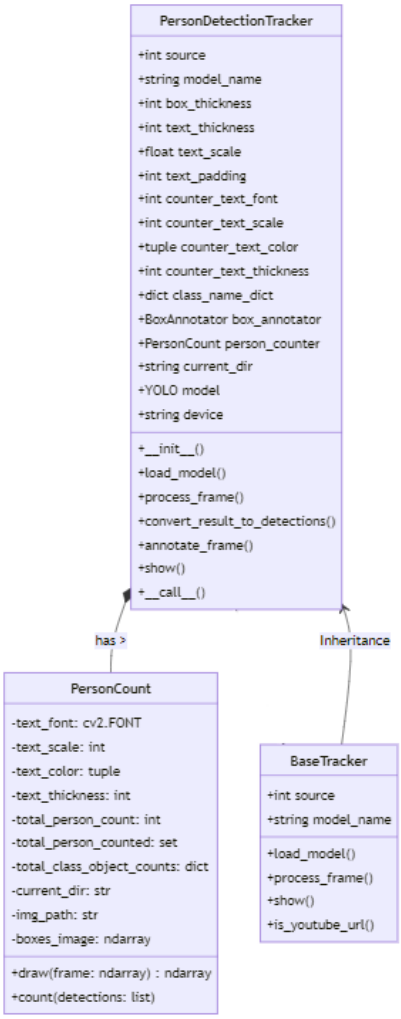

Figure 6 - PersonDetectionTracker UML diagram

The class diagram outlines the structure for person detection and tracking. The **BaseTracker** class, with attributes like source and **model_name**, serves as the base. It includes methods for model loading, frame processing, and result display. The **PersonDetectionTracker** class extends **BaseTracker**, focusing on person detection and tracking. It has attributes for source, **model_name**, **box_thickness**, **text_thickness**, and methods for model loading, frame processing, result conversion, frame annotation, result display, and process calling. It associates with the **PersonCount** class, which includes attributes for **text_font**, **text_scale**, **text_color**, **text_thickness**, **total_person_count**, and methods for drawing counts on frames and counting persons based on detections. This diagram provides an overview of class interactions in person detection and tracking systems.

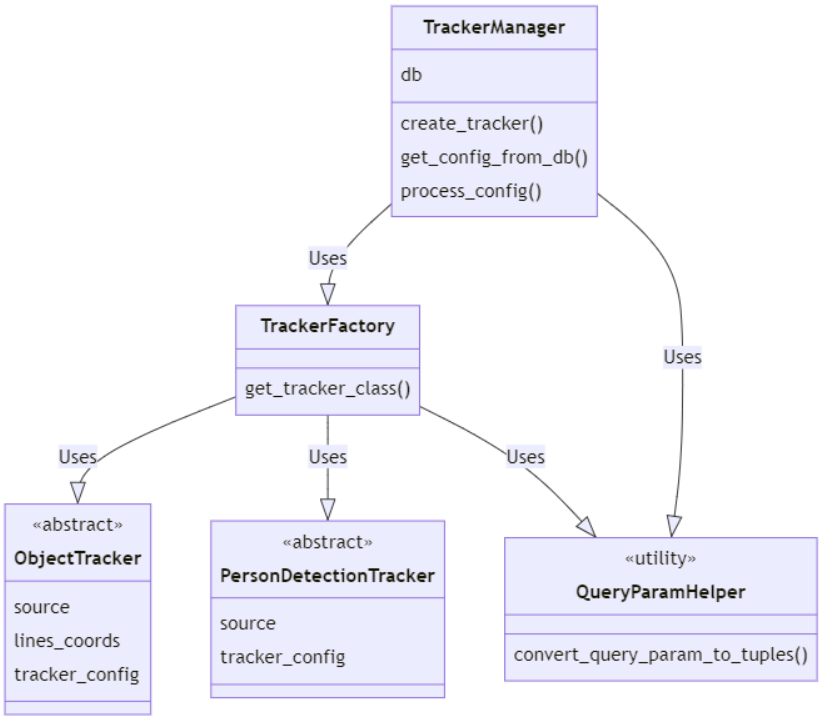

Figure 7 - Tracker Factory UML design

The class diagram showcases the structure for object tracking and person detection. The **TrackerFactory** class, central to the system, uses **ObjectTracker**, **PersonDetectionTracker**, and **QueryParamHelper** classes. It has a method, **get_tracker_class(),** for determining the suitable tracker class.

The **TrackerManager** class, utilizing **TrackerFactory** and **QueryParamHelper**, includes methods for creating the tracker, retrieving and processing the configuration, and a 'db' attribute for database interactions.

The **ObjectTracker** and **PersonDetectionTracker** are abstract classes serving as bases for object tracking and person detection, respectively. They include attributes essential for their respective tasks.

The **QueryParamHelper**, a utility class, has a method, **convert_query_param_to_tuples(),** for handling query parameters.

In summary, this diagram provides an overview of the class structure and relationships in object tracking and person detection, illustrating how the classes interact to achieve the desired functionalities.

3 TOOLS AND TECHNOLOGIES

3.1 Introduction to CUDA GPU Architecture

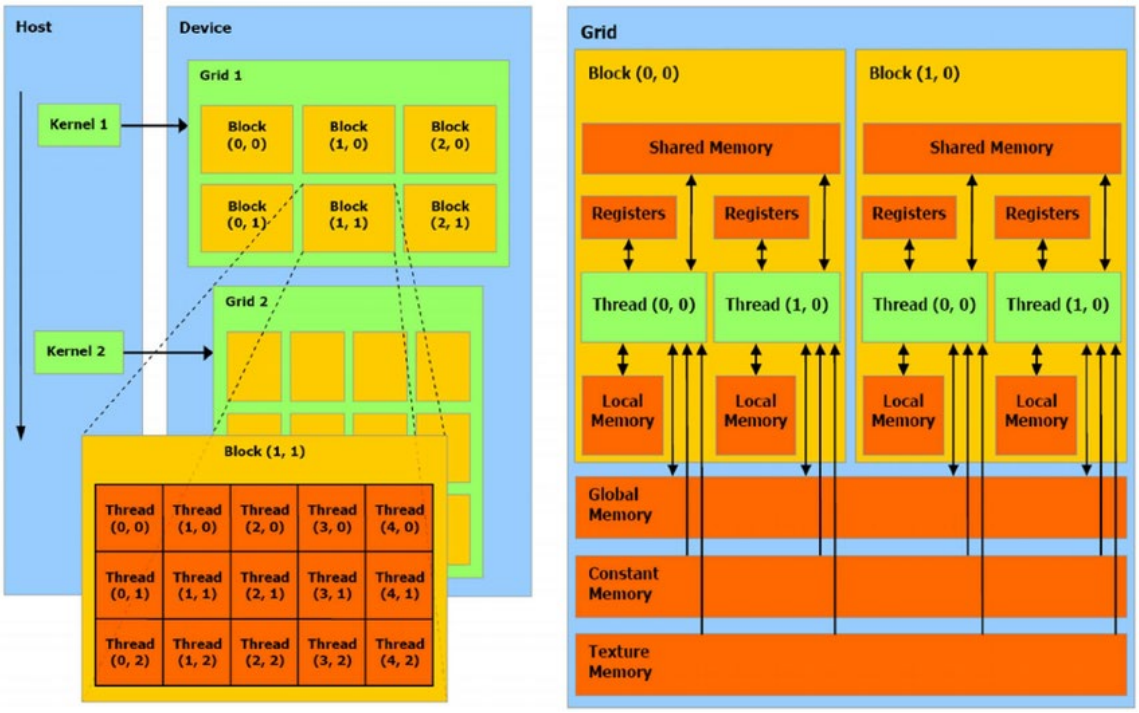

Figure 8 - CUDA Architecture (Source: [AARO23])

GPUs, with their high number of cores, are particularly well-suited for tasks that can be broken down into many smaller, independent tasks. For example, the GTX 1650 GPU used for testing our application has 896 CUDA cores, a number far greater than the 6 cores (with 12 logical processors) that a typical CPU might provide [15].

Image processing is one such task that benefits greatly from the parallel processing capabilities of GPUs. Images are represented as matrices of pixel values. In the case of grayscale images, this matrix is two-dimensional. For color images, which are typically represented in the RGB (Red, Green, Blue) color space, the image is represented as a three-dimensional matrix, with separate 2D matrices for each of the three-color channels.

[15] NVIDIA Corporation. "NVIDIA GeForce GTX 1650. – link: https://www.nvidia.com/en-gb/geforce/graphics-cards/gtx-1650/.

The matrix representation of images aligns well with the parallel processing capabilities of GPUs. Each pixel in the image can be processed independently, allowing for efficient computations on GPUs, which excel at performing the same operation on multiple data points simultaneously.

In the context of GPU processing, the host (CPU) triggers the execution of a kernel on the device (GPU), and transfers the image data from the host memory to the device memory. The computation is performed in the device's execution context and memory. Once the computation is complete, the results are transferred back from the device (GPU) to the host (CPU) for display or further processing.

However, transferring and processing images frame by frame can be inefficient due to the overhead of data transfer between the host and device. To maximize efficiency, images are often transferred in batches to the GPU, processed in parallel, and then the results are transferred back in batches.

Underneath the YOLO architecture, this is how images are processed on the GPU[16]. The CUDA programming model involves launching a kernel on the GPU, which operates on a grid of thread blocks. Each pixel is assigned to a specific thread within a block, and multiple blocks are contained within a grid. This structure allows for efficient parallel processing of the image data, leveraging the full power of the GPU.

Another bottleneck for performance in this whole workflow of communication from HOST to DEVICE and from DEVICE to HOST is RAM storage. In order to run the program at maximum capacity, as long as we transfer more images to the GPU and after processing transferring them back to the HOST memory, sometimes not even 32GB of RAM is not enough, but in the case that multiple images are transferred to the DEVICE and processing them, the time is increasing significantly.

In the next paragraphs I will provide some more experiments where processing was done on different datasets of images with different resolutions for the 4 main operators for image processing (Canny, Sobel, Robert, Laplacian Prewitt).

The comparison is based on CPU/GPU comparison of image processing using OpenCV[17].

---

[16] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640.

[17] Ivanescu, R. C., Nagy, R., Ofiteru, A. I., Comanescu, C., & Manda, A. (2022, November). Parallel vs Distributed Edge Detection for Large Medical Image Datasets. In 2022 E-Health and Bioengineering Conference (EHB) (pp. 1-4). IEEE.

Operators	1000 images (1060x780 px)	2000 images (1060x780 px)	4000 images (1060x780 px)	6000 images (1060x780 px)	8000 images (1060x780 px)	10000 images (1060x780 px)
Canny	0.974747	1.89261	3.95661	5.79486	7.83885	9.6521
Sobel	0.78975	1.30799	2.6249	3.70192	4.68741	8.3535
Laplacian	0.722378	1.25485	2.6418	3.74211	4.75799	5.98398
Prewitt	0.998004	1.70571	3.39869	4.95302	6.39718	7.79672
Roberts	0.996254	1.69704	3.3585	4.77709	6.16388	7.60922

Table 1 - GPU time processing for images

1000 images (1060x780 px)	2000 images (1060x780 px)	4000 images (1060x780 px)	6000 images (1060x780 px)	8000 images (1060x780 px)	10000 images (1060x780 px)
1.56802	3.58075	6.92068	9.91495	13.3869	16.5299
1.29007	2.45574	4.80471	7.16624	8.37861	11.661
0.904295	1.71551	3.22781	4.66849	6.63752	8.19946
1.22371	2.20067	4.34703	5.89519	7.71857	10.1342
1.56169	1.78521	4.93254	7.36306	9.21094	11.5188

Table 2 - CPU time processing for images

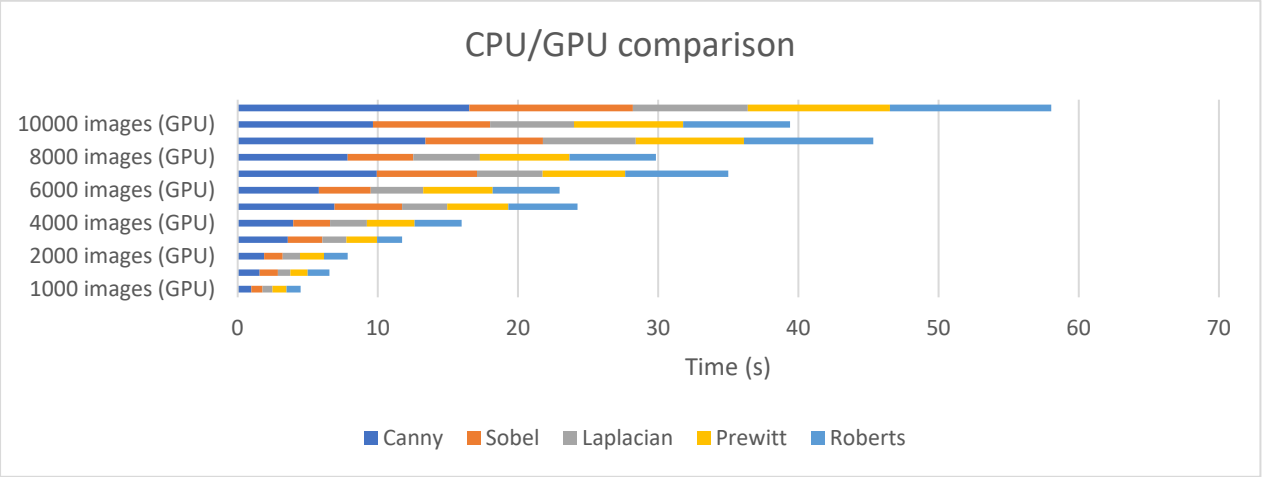

Figure 9 - CPU/GPU comparison

3.2 CNN Architecture and image operations

To begin with, the simplest way to understand CNN is by applying a filter on a black and white image. Let's take as an example the Gaussian filter.

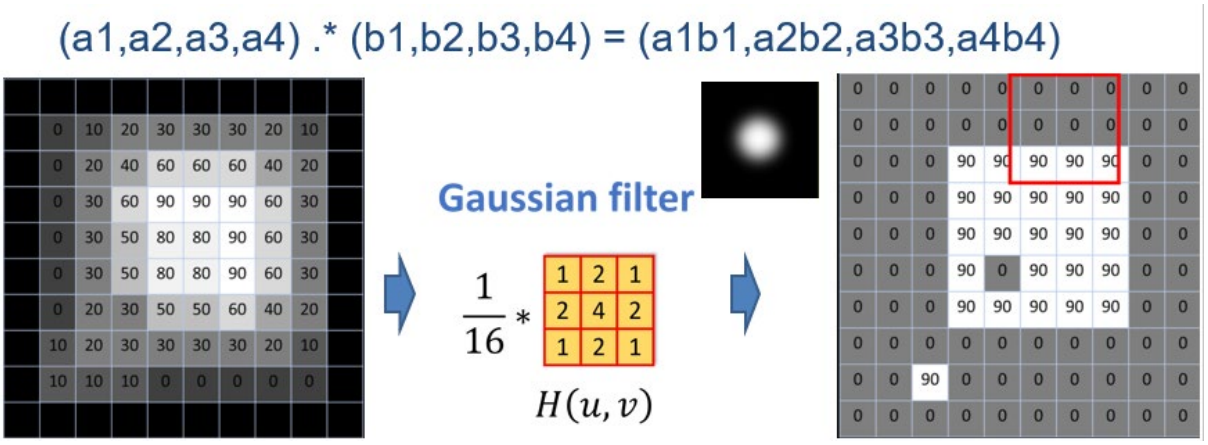

Figure 10 - Gaussian Filter Example

In this case, the input is the pixel values of an image, represented as a 2D matrix. We then apply a selected filter, which in our case is a 3x3 matrix filter multiplied by 1/16. This filter is traversed across the image, and matrix multiplication is applied to the original image. The resulting output image corresponds to the application of the Gaussian Filter.

In a broader context, key stages within a CNN include stride, pooling, and the application of the ReLU activation function. Ultimately, the outputs of the network predict the class of the detected object.

To give a more practical example applied on more image, we can take a reference to the following image:

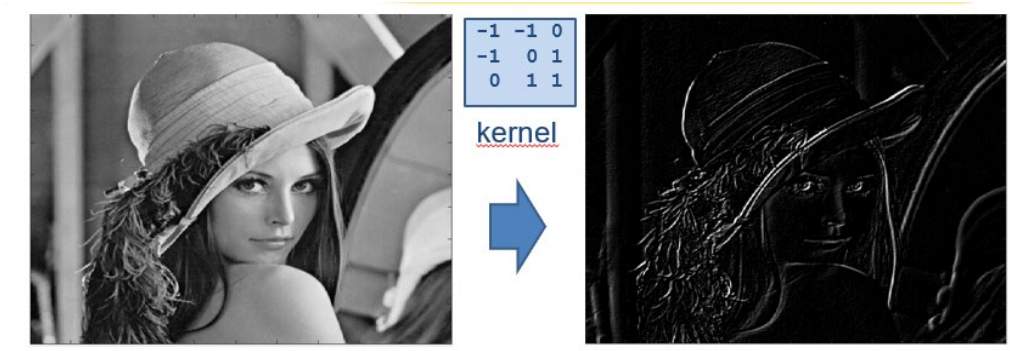

Figure 11 - Lena convolutional kernel filter

In case of Neural Networks, we have the same concept, but slightly different. The original image is processed by passing it through multiple layers.

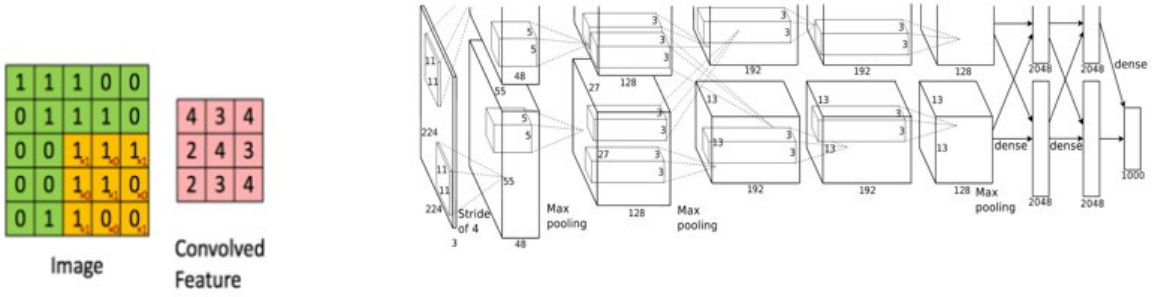

Figure 12 - CNN neural networks

We have the original image who will be passed through multiple steps of convolution. The stride is the part that tells the neural network how many frames to skip when multiplying the next iteration of the matrix. By each operation, the size of the image gets smaller and smaller containing the convolved feature of the image[18].

To explain further, the stride operator can be more easily understood by following the image bellow:

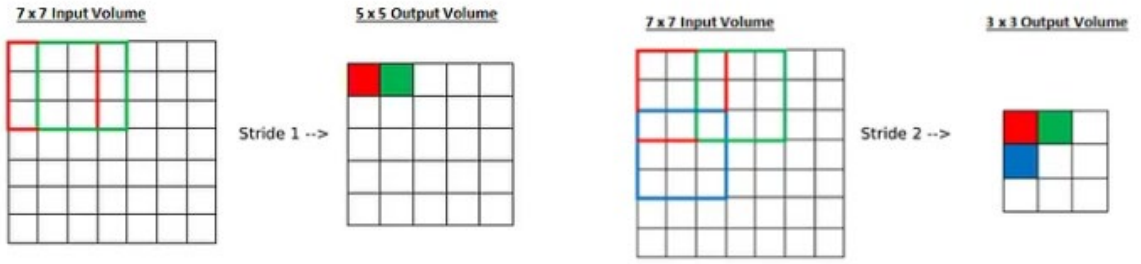

Figure 13 - Stride explanation

The stride parameter plays a crucial role in determining the size of the output feature map. A larger stride results in a smaller feature map, while a smaller stride results in a larger feature map. This is

[18] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. http://www.deeplearningbook.org

because a larger stride means the filter is moved more pixels at a time, covering fewer total positions, and thus producing a smaller output[19].

In addition to convolution and stride, other key operations in a CNN include pooling (also known as subsampling or down-sampling), which reduces the dimensionality of each feature map while retaining the most important information, and the application of the ReLU activation function, which introduces non-linearity into the model, allowing it to learn more complex patterns[20].

In the context of the YOLOv8 model used for our real-time traffic analysis system, the activation function plays a key role in the object detection process. When an image or video frame is passed through the network, it undergoes a series of convolutions, pooling operations, and non-linear transformations via the activation function. These operations extract features from the image that are then used to predict the presence and location of objects in the image.

The stride operation in the convolutional layer of YOLOv8 helps in reducing the spatial size of the 3D volume of feature maps, thus reducing the computational complexity of the network. This is particularly important for a real-time system like ours, where computational efficiency is key to achieving real-time performance.

Furthermore, the YOLOv8 architecture includes multiple convolutional layers with different filter sizes and strides, which allows the network to learn features at various scales and complexities. This is crucial for detecting objects of different sizes and shapes in the traffic scenes.

In summary, the combination of convolutional layers, stride operations, and the Mish activation function in the YOLOv8 architecture enables the model to effectively and efficiently detect and classify objects in real-time, making it an excellent choice for our traffic analysis system[21].

[19] Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285.

[20] Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In European conference on computer vision (pp. 818-833). Springer, Cham.

[21] Misra, D. (2019). Mish: A Self Regularized Non-Monotonic Neural Activation Function. arXiv preprint arXiv:1908.08681.

3.3 Introduction to YOLOv8

The YOLOv8 model, which stands for "You Only Look Once version 8", is a state-of-the-art object detection model that has gained significant attention in the field of computer vision due to its speed and accuracy[22]. Unlike traditional object detection models that scan an image multiple times at different scales and aspect ratios, YOLOv8 looks at the entire image only once, hence the name "You Only Look Once". This unique approach makes YOLOv8 significantly faster than other object detection models, making it ideal for real-time applications[23].

One of the key advantages of YOLOv8 is its ability to detect objects in real-time with high accuracy. This is achieved through a combination of convolutional neural networks for feature extraction and a custom layer for bounding box prediction. The model is trained to predict the class of an object and the coordinates of a bounding box around the object simultaneously, which allows it to detect and classify objects in an image in a single pass[24].

In the context of a real-time traffic analysis system, the speed and accuracy of YOLOv8 are crucial. The system needs to be able to detect and classify vehicles quickly and accurately to provide real-time traffic analysis. As found by Bochkoyskiy et. al., the high speed of YOLOv8 allows the system to process video feeds in real-time, while its high accuracy ensures that the traffic analysis is reliable.

[22] Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934.

[23] Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv preprint arXiv:1804.02767.

[24] Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934

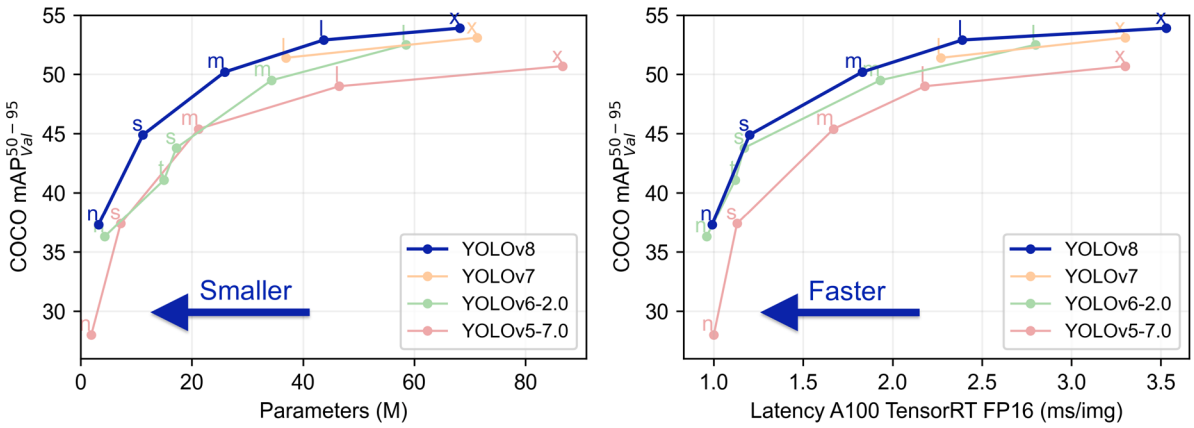

Figure 14 - YOLO versions performance comparison

As shown in Figure 1, the YOLOv8 model, trained on the COCO (Common Objects in Context) image dataset, outperforms other YOLO versions in terms of mean average precision. The COCO dataset, consisting of 330k images, is a large-scale object recognition and segmentation dataset which we use it in our application for detecting people in the frame[25].

YOLO provides different versions of weights: n, s, m, l, xl. Choosing a higher version involves a trade-off between the precision of the overall detection and the latency of the inference. For instance, the 'S' version in the chart yields a mAP with a confidence interval of 50-95% of 45.

The latest version of YOLO v8 provides support for in-built tracking functionalities by implementing ByteTrack and BoT-SORT which we are using in our project and we need it later for speed estimation task, since we want to keep track of the detected object alongside the frame, and if it disappears from the frame to be removed.

[25] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., ... & Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In European conference

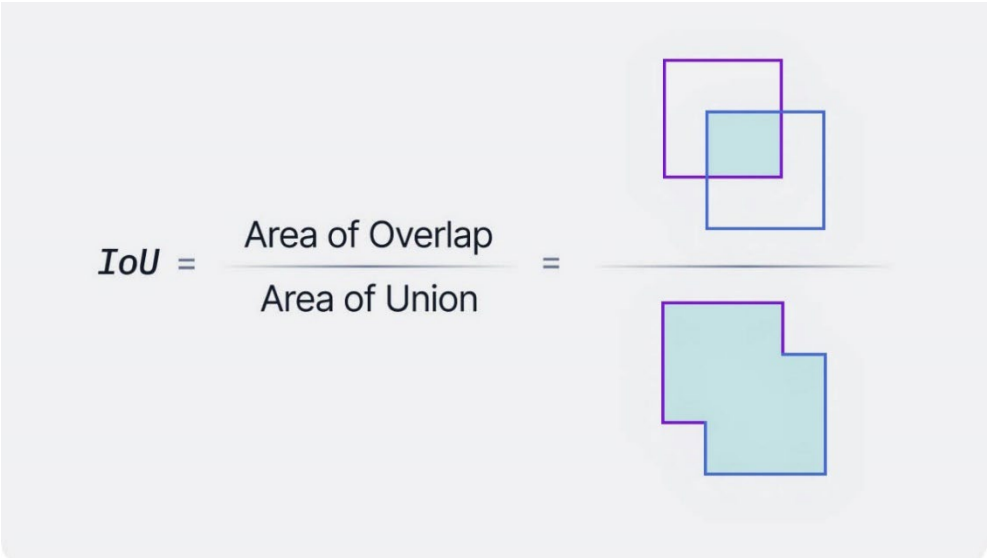

Figure 15 - Intersection over Union (Source: [ROH23])

Here, the confidence interval of 50-95% refers to the Intersection over Union (IoU) thresholds used to evaluate the model's performance. IoU is a metric used in object detection to measure the overlap between the predicted bounding box (from the model) and the ground truth bounding box (the actual position of the object in the image). The IoU value ranges from 0 to 1, where 1 means a perfect overlap (the predicted bounding box exactly matches the ground truth), and 0 means no overlap[26].

In the context of the mAP score, the IoU thresholds of 50-95% mean that the model's predictions are evaluated at different levels of precision, from a relatively medium overlap of 50% to a very strict overlap of 95%. A mAP score of 45 at these thresholds indicates that the YOLOv8 model is able to accurately detect and localize objects even under difficult conditions, further demonstrating its effectiveness for real-time traffic analysis applications.

Moreover, the model itself it's written as a python module and can be installed using the following command: "pip install ultralytics". It can also be downloaded separately under the ultralytics github page and the standard functionality can be overridden in order to provide custom functionality to our yolov8 model. In our project we are calling and instantiating the YOLO class as an installed python module and the speed detection, counting are built on top of that.

[26] Rosebrock, A. (2017). Intersection over Union (IoU) for object detection. PyImageSearch

3.4 Roboflow

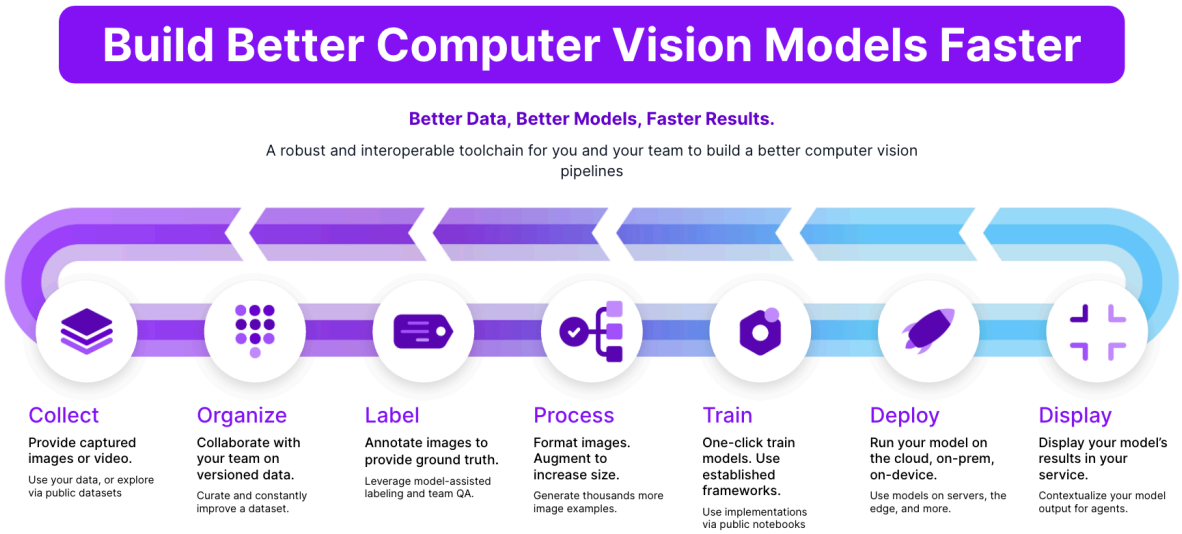

Figure 16 - Roboflow process (Source: https://docs.roboflow.com/)

In our project, Roboflow was used in order to collect, label and annotate the data for our custom dataset of highway vehicle detection. Roboflow gives flexibility in annotation and it is a valuable resource for computer vision models as it has export format which YOLOv8 accepts it.

Here it can be found my roboflow dataset titled "Highway Cars Object Detection," which is available at https://universe.roboflow.com/car-detection-smznf/highway-cars-object-detection.

The process of creating the dataset involved several key steps facilitated by Roboflow. Firstly I collected frames for the dataset by uploading a custom video with traffic monitor webcam. Roboflow created the dataset for me by sampling the video each 1 frame, resulting in 809 frames.

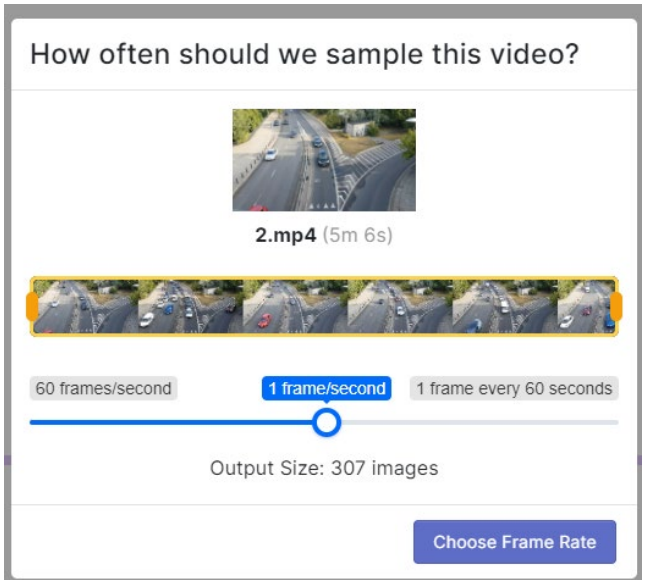

Figure 17 - Roboflow sampling from video

Next step involved in creating the dataset is to assign labeling jobs for annotating the dataset with labels for the classes we want to detect (car, bus, motorcycle, truck). The annotation process was made efficient and precise through Roboflow's annotation tools, such as object duplication and batch annotation. I paid close attention to detail, ensuring accurate and consistent annotations across the entire dataset.

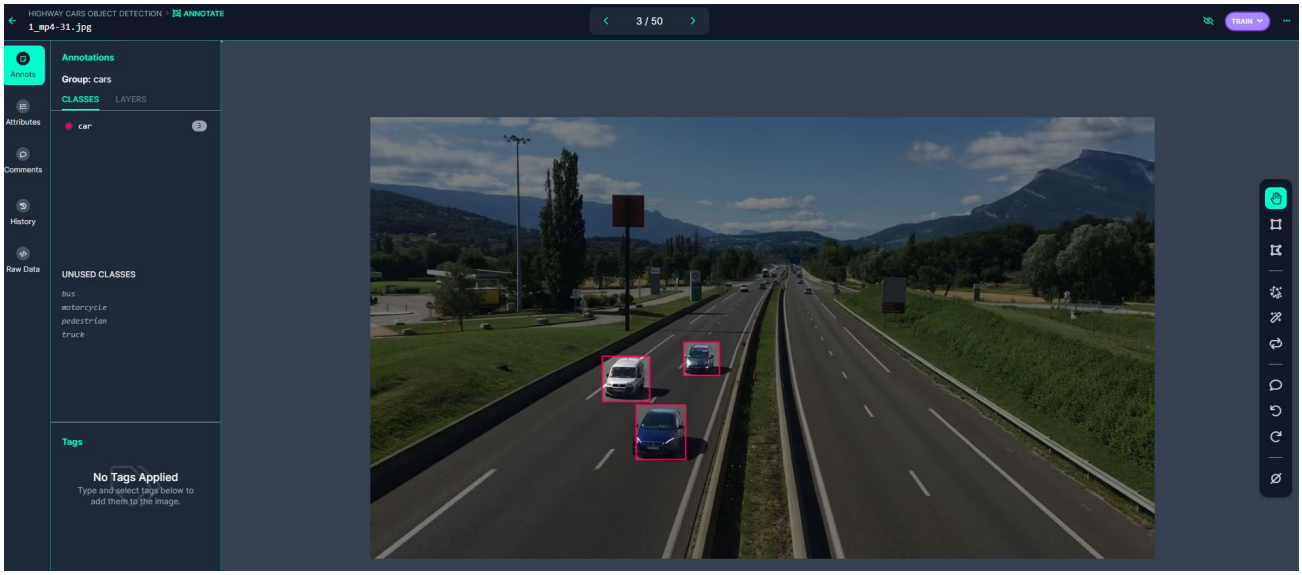

Figure 18- Annotation example roboflow

The good part is that we don't have to manually annotate all the 809 images at once. We can choose to annotate the model for about 40-50 images in the first version, train the model inside Roboflow and we can actually annotate it using the v1 model using the feature called Label Assist :

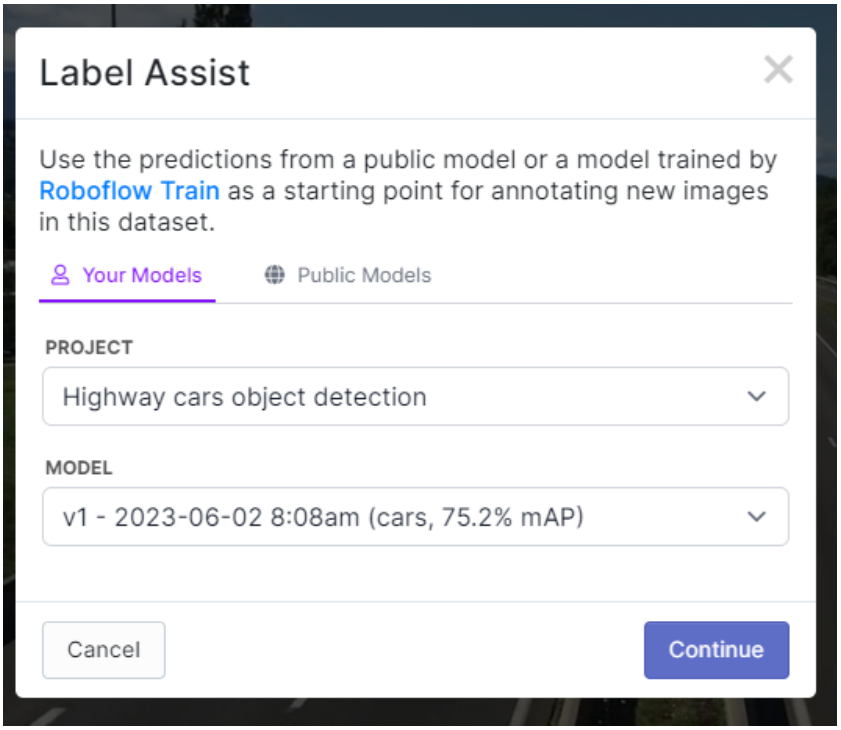

Figure 19 - Roboflow label assist example

Now when we try to annotate the image, it will automatically draw bounding boxes corresponding to each class we trained in the V1 model and we can just adjust it if something it's wrong.

This way, we speed up the process of annotating the dataset. Roboflow's technology demonstrated excellent performance and scalability, effortlessly handling the large-scale dataset of highway images. It provided a reliable and efficient platform for annotating thousands of images, even in complex projects like traffic-analytics.

Overall, Roboflow proved to be an indispensable tool for dataset creation and manual annotation in my traffic-analytics YOLOv8 project. Its user-friendly interface, annotation flexibility, efficient annotation tools, quality control mechanisms, collaboration features, export compatibility, and scalability capabilities made it the ideal choice for this endeavor. The "Highway Cars Object Detection" dataset created through Roboflow serves as a valuable resource for training robust car detection models in traffic-analytics applications.

Even tough we can access and test our model in real time, there is no way that we can export the model weights that were trained inside the roboflow platform. So for that part we will use Google Collab GPU resources to train the model, since it's more efficient and fast to do it like so.

Figure 20 - Roboflow training results for the model

As we can see, Roboflow can split the dataset between training, validation and testing set of images automatically.

To export the dataset in accepted YOLOv8 format, we have two possibilities: either we install the roboflow python package and with the API key we can download the dataset, or we can download directly the dataset as a .zip file into our computer:

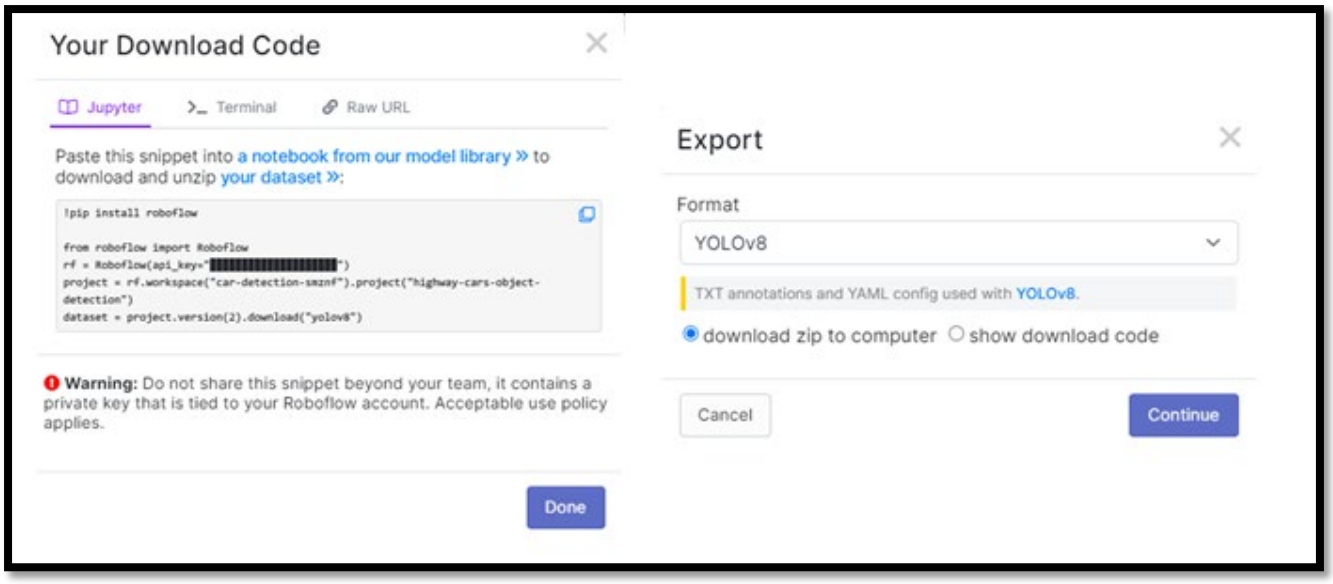

Figure 21- Roboflow export dataset options

In the end, if we download the dataset, it should be a folder with the following structure:

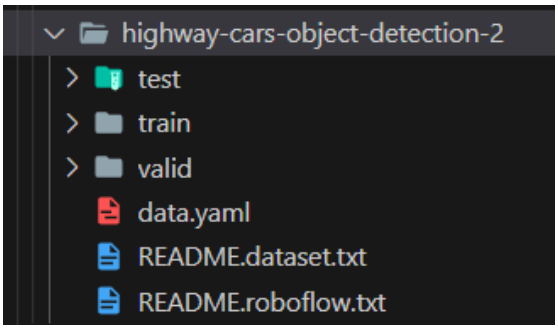

Figure 22 - Roboflow dataset format

By providing **data.yaml** we can train the dataset with directly with yolov8 in python by installing ultralytics python package. This step will be explained in detail in the Google Collab section.

3.5 Docker

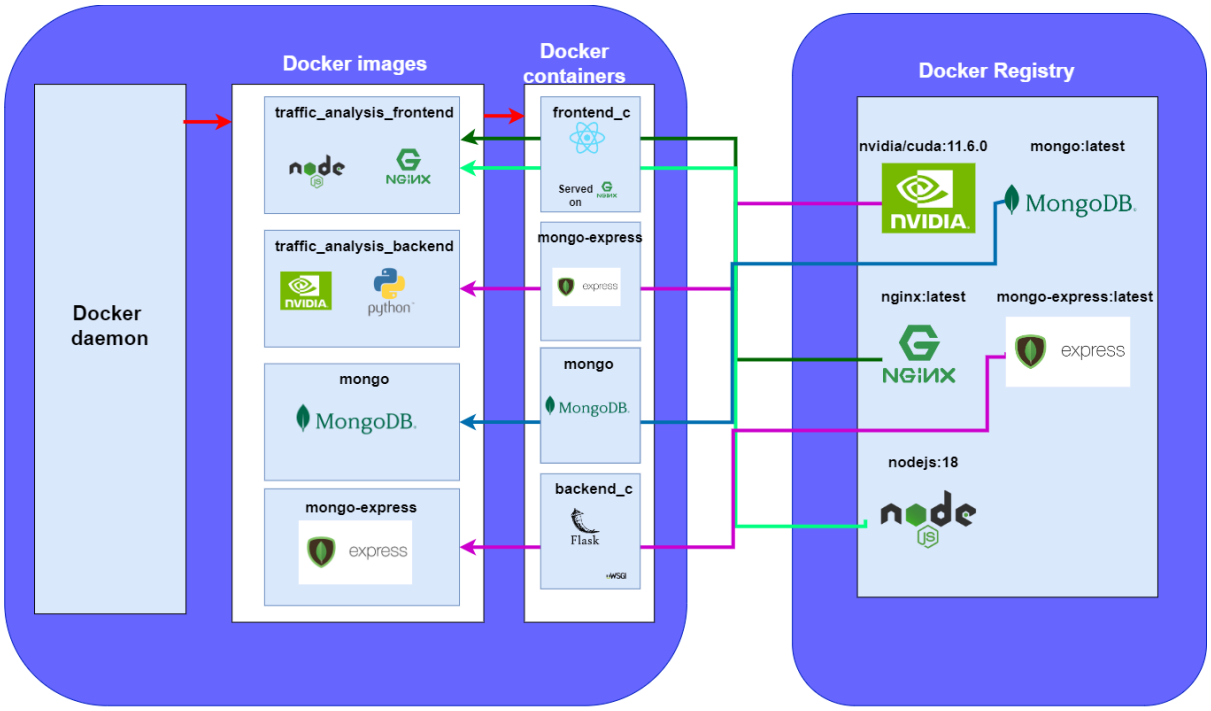

Figure 23- Docker architecture of the application

Using Docker within Windows Subsystem for Linux 2 (WSL2) provides the flexibility and power of a Linux environment while retaining the convenience of working on a Windows machine. This setup is incredibly beneficial for development and testing, as it allows us to work in the same environment in which our application will eventually run.

Our Docker configuration ensures that the code is modular, scalable, and isolated, making it ready for production right out of the box. One of the greatest advantages of Docker is its ability to reproduce the same environment across different machines, reducing the "it works on my machine" problem.

However, Docker does have some limitations, and one of them is its inability to directly access the host's USB peripherals. In our case, this meant we had to find a workaround for accessing the device's webcam. We achieved this by using FFmpeg to stream the webcam feed through UDP (User Datagram Protocol), thus bypassing Docker's limitation.

Moreover, we also faced challenges with displaying GUI applications, specifically using OpenCV's imshow function within a Docker container. Docker doesn't support GUI applications out of the box, so we leveraged VcXsrv, an X server for Windows, to display the output of imshow. This setup required

providing the local IP of the computer and starting the DISPLAY server on port 0. As a result, imshow outputs can be displayed in a window corresponding to the VcXsrv server.

While these workarounds do add some complexity to our Docker setup, they enable us to leverage Docker's benefits while overcoming its limitations. This flexibility and adaptability are key reasons why we chose Docker for our project.

The frontend of our application, which is a Node.js application, is built using the **node:18** Docker image, providing the necessary Node.js runtime.

```
FROM node:18 AS builder

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

# Build the frontend application
RUN npm run build

# Use Nginx as the final image
FROM nginx:latest

# Remove default Nginx configuration
RUN rm /etc/nginx/conf.d/default.conf

# Copy the built frontend files to Nginx
COPY --from=builder /app/build /usr/share/nginx/html

# Copy the custom Nginx configuration file
COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80 for HTTP and 443 for HTTPS (if needed)
EXPOSE 80
EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]
```

Figure 24 - Frontend dockerfile

The Dockerfile above is for the frontend service. It begins from the Node.js base image and includes the necessary build steps. It then compiles the Node.js application and prepares it for serving.

Once our Node.js application is built, we use Nginx, a powerful web server, to serve static files and handle client requests. This is accomplished through the **nginx:latest** Docker image. After removing

the default Nginx configuration, we copy the compiled Node.js files and our custom Nginx configuration to the container.

Our backend service is Python-based and performs computationally heavy operations. For this reason, we use the **nvidia/cuda:11.6.0-base-ubuntu20.04** Docker image. This image allows us to leverage Nvidia's CUDA platform and GPUs to enhance computational performance.

```
FROM nvidia/cuda:11.6.0-base-ubuntu20.04
RUN apt update
RUN apt-get install -y python3 python3-pip


WORKDIR /app

COPY requirements.txt .

RUN pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu116


RUN apt-get update && export DEBIAN_FRONTEND=noninteractive \
    && apt-get -y install --no-install-recommends \
    cmake \
    gcc g++ \
    libavcodec-dev \
    libswscale-dev \
    libavformat-dev \
    libgstreamer-plugins-base1.0-dev \
    libgstreamer1.0-dev \
    libpng-dev \
    libopencv-dev \
    libjpeg-dev \
    libopenexr-dev \
    libtiff-dev \
    libwebp-dev \
    wget \
    qtbase5-dev \
    qtchooser \
    qt5-qmake \
    qtbase5-dev-tools \
    libtbb-dev \
    libgphoto2-dev \
    ffmpeg \
    usbutils \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

RUN pip install -r requirements.txt

RUN pip install uwsgi
```

Figure 25 - Backend Dockerfile config

The Dockerfile above is for our backend service. It's based on a CUDA-enabled Ubuntu image and includes the necessary steps to prepare the Python environment, install the dependencies, and set up the application to run using uWSGI.

We store application data in MongoDB, a flexible and scalable NoSQL database. For this, we use the **mongo** Docker image. By containerizing MongoDB, we ensure that our database runs in a consistent environment and scales when necessary.

To manage MongoDB databases and collections, we've incorporated Mongo Express into our Docker setup. The **mongo-express** Docker image provides a user-friendly web interface to manage our data.

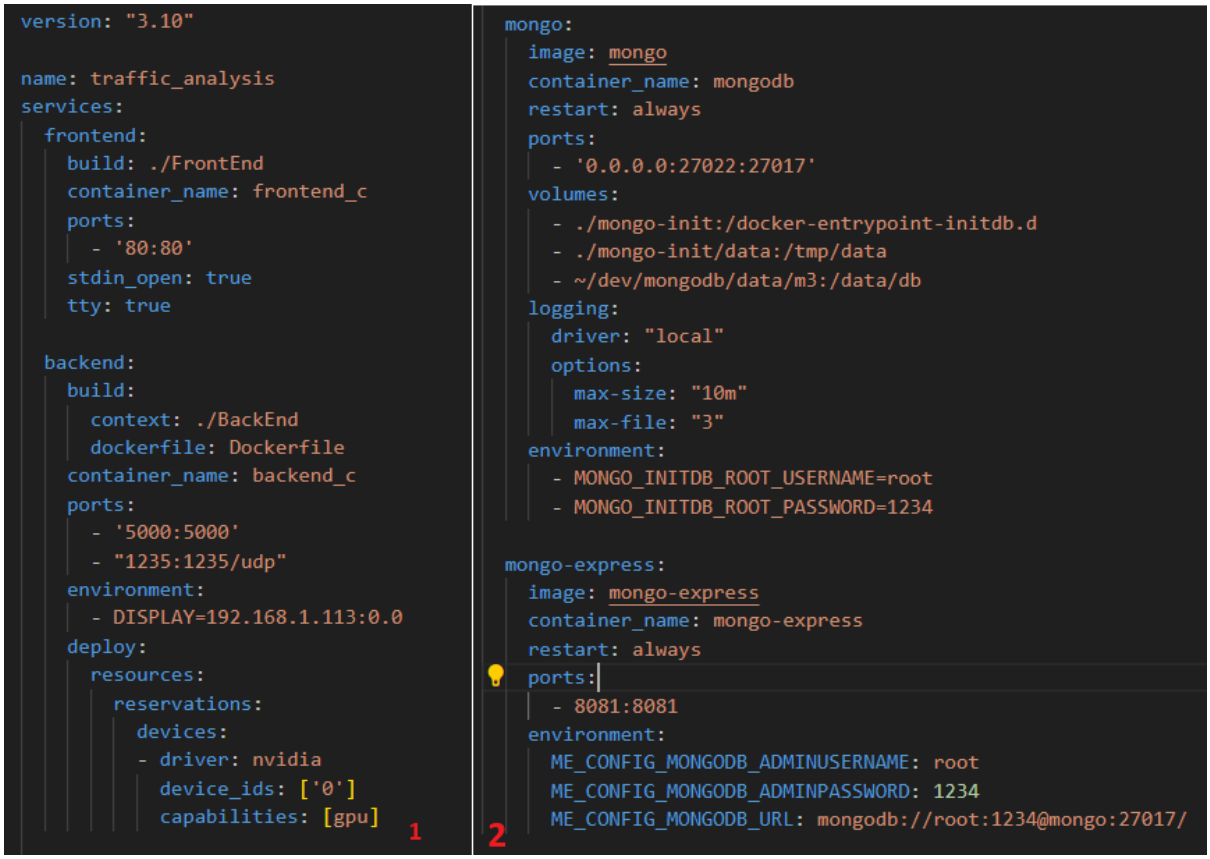

Figure 26 - Docker Compose file

The screenshot above shows our Docker Compose configuration. It defines each service, their dependencies, build context, and environment variables. Docker Compose makes it easy to manage and reproduce our multi-container Docker applications.

Docker provides a robust, scalable, and efficient environment for our application. As we continue developing our project, we'll further refine and optimize our Docker setup.

3.6 FFMPEG – streaming webcam through UDP

To allow webcam with Docker, we need to use ffmpeg for streaming the webcam data through UDP. For that purpose, we need to install the latest ffmpeg implementation and add it to the system environment path (in case of windows system).

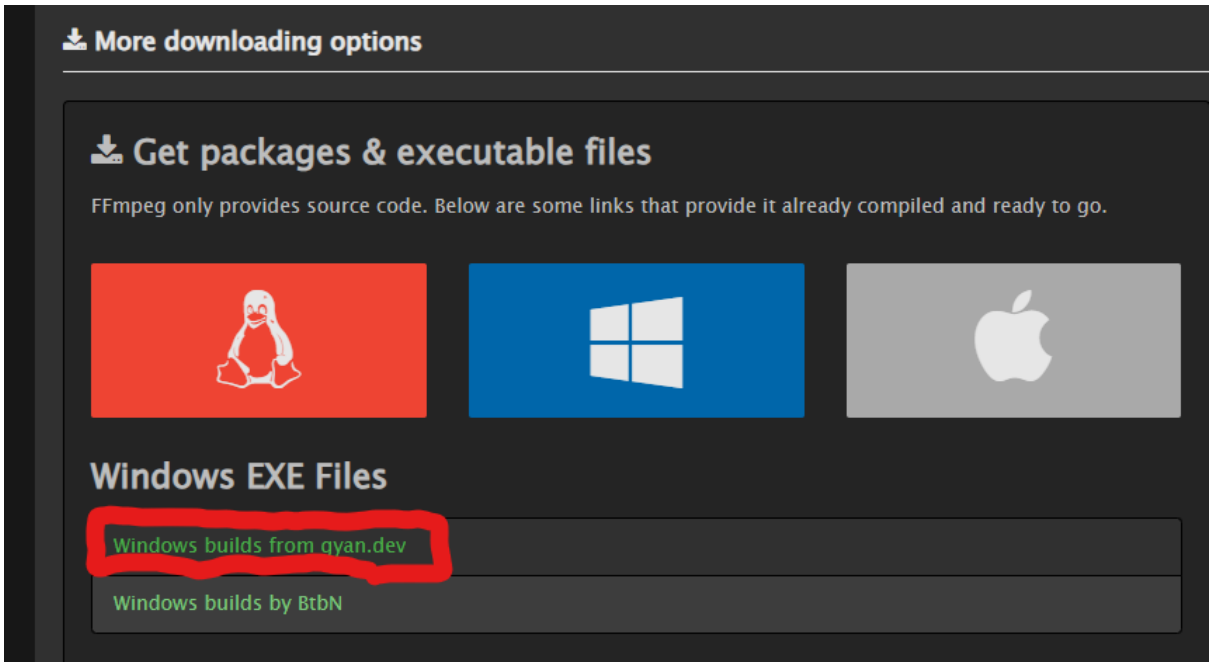

Figure 27 - Download method for ffmpeg

After this, we are prompted to another webpage where we need to download the **ffmpeg-git-full.7z** archive :

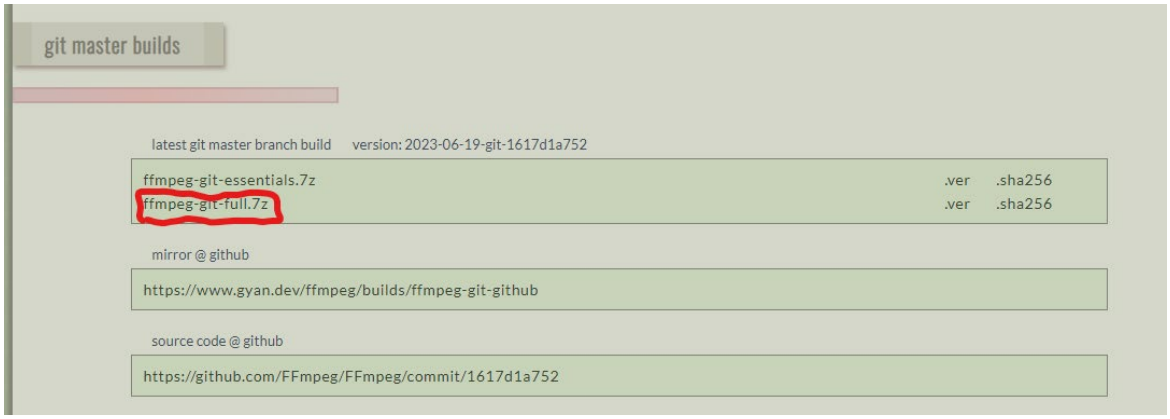

Figure 28 - FFMPEG correct 7z to download

Afterwards, we need to copy, extract the folder to the C:/ folder and add it to the system env path so that we can access it from the Windows CMD.

After all of the steps, we just need to identify our webcam that we need to use following the command:

ffmpeg -list_devices true -f dshow -i dummy

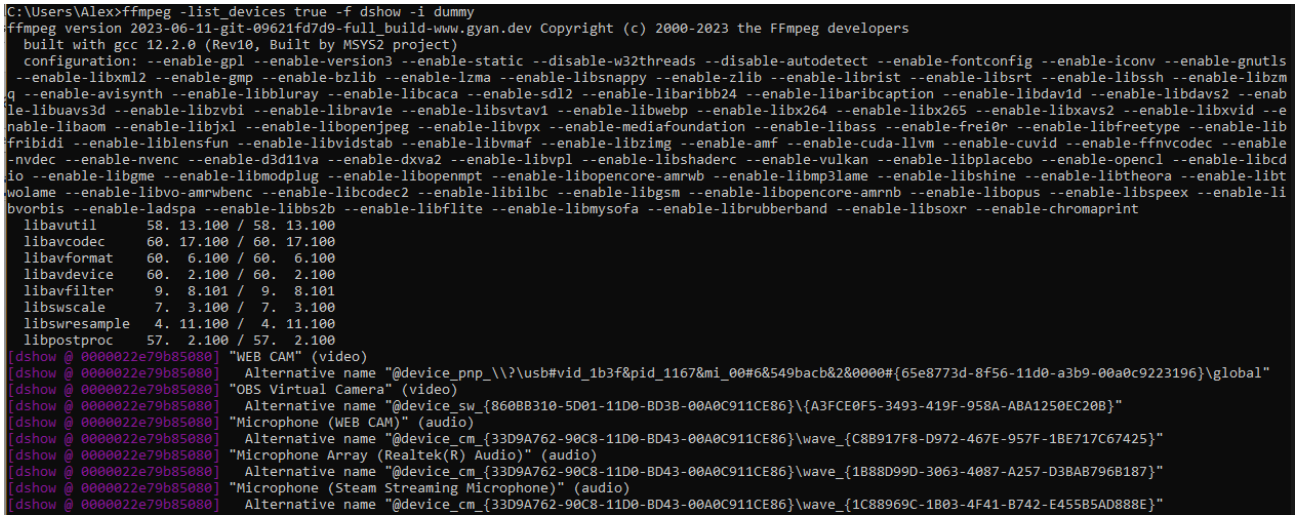

Figure 29 - FFMPEG list webcam command

As we can see, for me I have the name of the webcam to be “WEB CAM” which I will use for streaming. This command will start to capture video from the webcam and stream it to the specified udp ip address and name of the port which we will use in our app to access the webcam:

ffmpeg -f dshow -framerate 30 -i video="WEB CAM" -vcodec mpeg4 -q 12 -f mpegts udp://127.0.0.1:1235.

For docker to work and listen to the port 1235, we have to configure inside the docker-compose.yaml of the backend service the port to map, like in this image bellow:

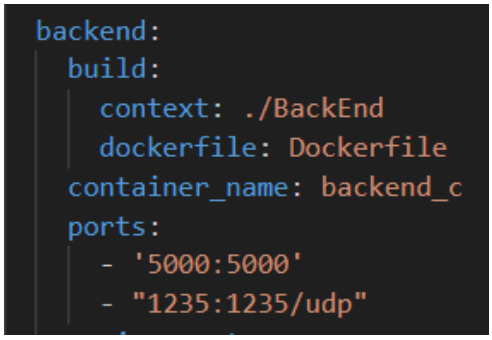

Figure 30 - Docker configuration of udp streaming port 1235

The configuration of ffmpeg with docker I was following using this git repository on internet: https://github.com/i99dev/Docker-with-WebCam.

3.7 VcXsrv – display server

To be able to test the yolo trackers inside the docker containers without the need of the frontend, we have to use a display server, like the one we have available from **Xming X Server (Link to download here : https://sourceforge.net/projects/xming/)**

The steps for configuration of VcXsrv is to launch the server and make sure to select no access control, so that Docker will be able to connect to it.

Now we have to create an environment variable for Docker in order to connect to the Display server, and we have to install the libraries needed for it to function. This is the docker-compose.yaml configuration we need to do in order to work with the display :

Figure 31 - Display env variable

3.8 Webpack

To begin with, let's see what webpack it's doing under the hood. It's providing us a development server with hot reload, so while we change the js code we are getting page refresh to see the changes in real time and it performs also bundling of all the js libraries, css files, .js, .jsx, images and it gives back main.js, and the corresponding .css file which we can later pass onto nginx configuration. If the application is built with npx create-react-app, then webpack by default will save the bundle in the build folder after running npm run build. Let's understand more easily what webpack does by looking at the following image:

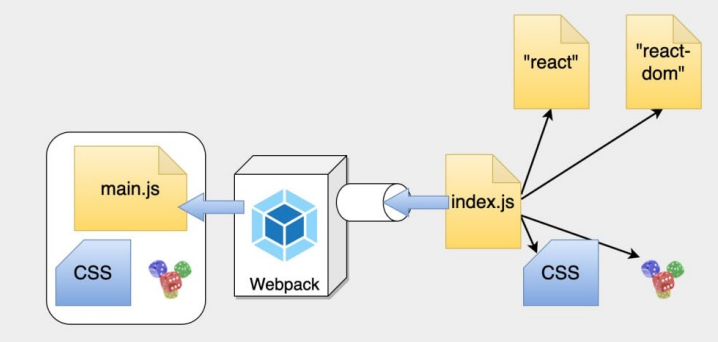

Figure 32- Webpack example

3.9 NGINX

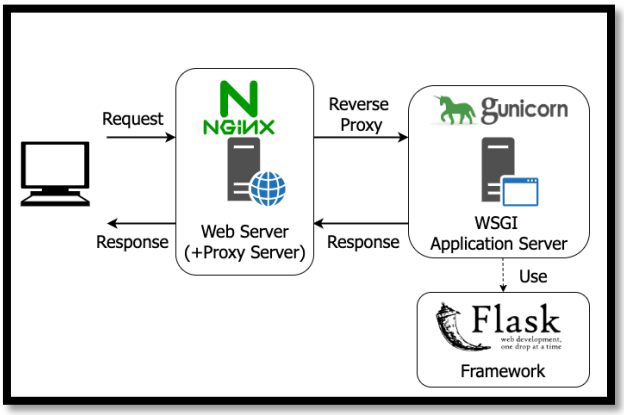

Figure 33 – [KRUP22] Nginx example

As we can see, nginx it serves as a production web server who is handling the requests and response from Client (React), it's configuring a reverse proxy to point to the uWSGI server and then to the flask application that it's being runned on the server redirecting it to the correct route.

4 PROJECT IMPLEMENTATION

4.1 Flask

The flask application is configured in the app/__init__.py module file by creating an application factory which will handle the db init, Blueprints configuration for our routes and CORS related issues.

```python
import eventlet
eventlet.monkey_patch()
import os

from flask_cors import CORS
from flask import Flask




def create_app():
    app = Flask(__name__)
    current_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

    app.config['UPLOAD_VIDEO_FOLDER'] = os.path.join(current_dir, 'assets', 'video')
    app.config['UPLOAD_IMAGE_FOLDER'] = os.path.join(current_dir, 'assets', 'image')

    CORS(app, resources={r"/*": {"origins": "*"}})

    from . import db
    app.teardown_appcontext(db.close_db)

    from . import routes
    app.register_blueprint(routes.bp)

    return app
```

Figure 34 - Flask app factory

As we can see, we setup the directory for uploading images and videos in the application config, alongside with registering the blueprint and **teardown_appcontext(db.close_db).** The database is configured in the flask global variable as a singletone, so every HTTP request has a db connection which we also need to register the teardown context, so that after each request we close the connection.

```python
from app import create_app

HOST_NAME = "0.0.0.0"
PORT = 5000
app = create_app()

if __name__ == '__main__':
    app.run(debug=True,port=PORT,host=HOST_NAME)
```

Figure 35 - Flask app run module

Afterwards, we create the application in the **run.py** by calling the **create_app()** from the app module. So the app will run from the BackEnd folder module with all the configurations necessarry.

```python
from flask import g
from pymongo import MongoClient

DATABASE_NAME="trackerDb"

def get_db():
    if 'db' not in g:
        g.db = MongoClient("mongodb://localhost:27017/").trackerDb
    return g.db

def close_db(error=None):
    db = g.pop('db', None)
    if db is not None:
        db.client.close()
```

Figure 36 - Db configuration

Flask provides us directly with the global context of the application, which we can inject the db connection and connecting to our database with the name trackerDb. We want to keep the db connection only when we request information from frontend to backend. So in that case, we will create a new connection to the db when we use it, and pop it from the g module when we finish the request.

In our application, we do separation of concerns by implementing Blueprints provided by Flask.

Flask Blueprints serve as an important structural element in building Flask applications, especially when the application size and complexity increases beyond a simple single file or module-based structure. They act as a tool for organizing Flask applications into distinct components or modules, each with their

own set of functionalities, templates, and static files, thus promoting modularity, reusability, and a clear separation of concerns[27].

A Blueprint can be visualized as a mini-application that's pluggable into the main Flask application. These mini-applications can have their own routes, error handlers, static files, and templates, which are later incorporated into the main application while retaining their unique identity. Blueprints can be registered multiple times on the application with different URLs, thereby providing a convenient method to replicate a set of operations across different parts of the application.

```python
bp = Blueprint('video', __name__, url_prefix="/api")

logging.basicConfig(level=logging.INFO)

@bp.route('/', methods=['GET'])
def index():
    """Returns the index page."""
    return render_template('index.html')
```

Figure 37 - Blueprints example

One of the key advantages of using Blueprints is the ability to refactor a large Flask application into smaller, more manageable components. For example, in a blogging application, one might create separate blueprints for operations related to 'posts', 'users', and 'comments', each with its own routes, view functions, error pages, and static files. This approach not only makes the codebase more maintainable and understandable but also ensures that changes to one component do not inadvertently affect the others. This separation of concerns becomes increasingly important as the application scales up and the development team grows.

Furthermore, Flask Blueprints help facilitate the creation of reusable application components. For instance, if you've created a Blueprint for user authentication, this Blueprint can potentially be reused across multiple Flask applications, saving development time and ensuring consistency in how user authentication is handled across projects[28].

---

[27] [ROBERT14] https://exploreflask.com/en/latest/organizing.html

[28] [PALL10] Pallets, *Blueprints and Views*, 2010 (Source: https://flask.palletsprojects.com/en/2.3.x/tutorial/views/)

The tracker factory class is responsible for getting the correct class based on the task provided, and it's achieving that using a dictionary class map like the following:

```
    """
    @staticmethod
    def get_tracker_class(task):
        """
        Returns the class of the tracker based on the task type.

        Args:
            task (str): The task type. Valid options are "traffic" and "person".

        Returns:
            The class of the corresponding tracker if the task type is valid, otherwise None.
        """
        tracker_type_map = {
            "traffic": ObjectTracker,
            "person": PersonDetectionTracker,
        }

        return tracker_type_map.get(task)
```

Figure 38 - Tracker class map

Then the responsibility of getting the parameters, processing and validation of the parameters from the db is at the **TrackerManager** class, which we will use as main method **create_tracker(self, task,source,line1,line2)** and based on the type of task, we will get the correct tracker from yolo module inside our flask application. Parameters line1, line2 are optional, since the person tracker doesn't require any lines passed from the frontend as coordinates. Since we will have a GET request with query parameters inside the link, the location will be in the following format: line1=(x1,y1)(x2,y2)&line2=(x1,y1)(x2,y2). So, we need to process them to have a list of lines as a tuple. In the above image we can see the implementation of the **TrackerManager** class:

```python
def create_tracker(self, task, source, line1, line2):
    """
    Creates and returns a tracker object.

    Args:
        task (str): The type of the task (e.g., "traffic", "person").
        source (str): The video source for the tracker.
        line1, line2: Additional parameters for the tracker.

    Returns:
        An instance of a tracker object.
    """
    TrackerClass = TrackerFactory.get_tracker_class(task)

    if not TrackerClass:
        return None

    if task == "traffic":
        tracker_config = self.get_config_from_db(task)
        lines_coords = []
        param_helper = QueryParamHelper()

        if line1 and line2:
            lines_coords.append(param_helper.convert_query_param_to_tuples(line1))
            lines_coords.append(param_helper.convert_query_param_to_tuples(line2))

        return TrackerClass(source=source, lines_coords=lines_coords, **tracker_config)
    else:
        tracker_config = self.get_config_from_db(task)
        return TrackerClass(source=source, **tracker_config)
```

Figure 39 - Create tracker method of TrackerManager

Based on the 2 types of tasks, we setup correspondingly the parameters we pass to the function. After this we pass the parameters of the constructor as keyword arguments after proper validation.

The Factory pattern is implemented in this code to encapsulate the object creation logic for different types of trackers, fostering code flexibility, reusability, and maintainability. This approach enables the easy introduction of new types of trackers without needing to modify the code that uses these objects. Consequently, complexity is concealed, and the code remains simple, easier to understand, and manage, promoting a higher degree of code reusability.

4.2 Stream feature

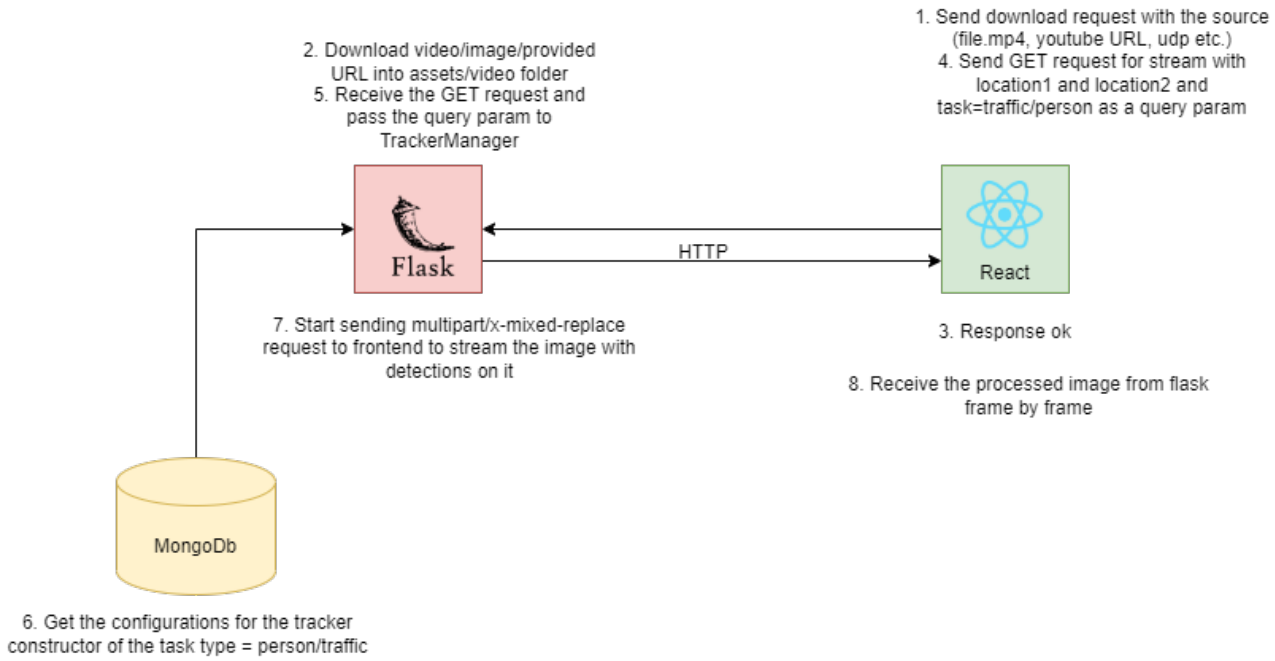

Figure 40 - Main workflow of the system

As we can see from the above image, we have the following workflow of communication between the backend and the frontend.

1. Download request from FrontEnd to BackEnd system
2. The received file from the frontend will be saved into assets/video folder
3. The response of the download request is ok
4. GET request for stream
5. Receiving the request at the backend
6. Getting the configuration from the DB
7. Start sending the video back to frontend

First, we send a download request from React with the source of the video/image we want to detect which according to the requirements it can be either an accepted video format (.mp4 mostly), a YouTube URL link, UDP stream link for camera streaming with FFMPEG and also RSTP protocol.

4.2.1 Frontend

Let's see the frontend implementation of the stream page with the upload and stream functionality:

```
return (
    <div className="stream-page-container">
        <div>
            <UploadDragAndDropForm
                onFileSelect={handleFileSelect}
                task={selectedTask}
                handleUrl={handleUrl}
            />

            <br></br>
            <h3>Select the tracker type </h3>
            <select className="task-select" value={selectedTask} onChange={handleSelectChange}>
                {tasks.map(taskOption =>
                    <option key={taskOption} value={taskOption}>
                        {taskOption === 'person' ? '👤 Person' : '🚗 Traffic'}
                    </option>)
                }
            </select>
        </div>

        {/* Only render VideoPreview if task is 'traffic' */}
        {selectedTask === tasks[0] && !linesPresent && isStreamStopped &&
            <VideoPreview
                key={selectedTask}
                file={selectedFile}
                task={selectedTask}
                onLineDrawing={handleLineDrawing}

            />
        }
        {!isStreamStopped &&
            <StreamPageVideoPlayer
                url={url}
                isStreamStopped={isStreamStopped}
            />
        }
    </div>
);
```

Figure 41 - Stream page

This is the main component for our **StreamPage** which handles the functionality for choosing the **VideoPreview** in case the task is 'traffic', since we want to be able to draw the lines on the video and the backend will receive them as coordinates for the 2 lines.

The main logic is that after we upload the video by drag and drop, or we provide the URL we want to access the detection, we then form query parameters with the source, location1, location2 (if the task is

traffic) and the task type which is either traffic or person. We navigate to the page with the query parameters, and in the **StreamVideoPagePlayer** we are receiving the stream from the backend.

```
return (
    <form id="form-file-upload" onDragEnter={handleDrag} >
        <input ref={inputRef} type="file" id="input-file-upload" multiple={false} onChange={handleChange} />
        <label id="label-file-upload" htmlFor="input-file-upload" className={dragActive ? "drag-active" : ""}>
            <div>
                <p><FaUpload /> Drag and drop Image / Video File </p>
                <p>OR</p>
                <p><FaYoutube /> Paste Youtube / Image URL</p>
                <div id="source-url-container">
                    <div id="input-group">
                        <input
                            ref={urlRef}
                            type="url"
                            id="source-url"
                            onChange={handleUrlChange}
                            placeholder="Paste a link.."
                            style={inputValid ? { borderColor: "green" } : { borderColor: "red" }}
                        />
                        <button id="clear-button" onClick={clearUrlInput}><FaTimesCircle /></button>
                    </div>
                </div>

                {!inputValid && <p className="validation-message" style={{ color: "red" }}>{validationMessage}</p>}

                <div id="buttons-area">
                    <button className="upload-button" onClick={onButtonClick}>
                        <FaFileUpload className="icon" /> Upload a file
                    </button>
                    <button id="webcam-button" onClick={onWebcamButtonClick}>
                        <FaCamera className="icon" /> Webcam
                    </button>
                </div>

            </div>
        </label>
        {dragActive && <div id="drag-file-element" onDragEnter={handleDrag} onDragLeave={handleDrag} onDragOver={handleDrag} onDrop={handleDrop}></div>}
    </form>
);
```

Figure 42 - Upload drag and drop component

Here we take the video from the user by either youtube link, udp stream or uploading an mp4 video, and after that we build the URL to redirect by updating the parent component state (StreamPage).

```
const handleFile = (file) => {
    const formData = new FormData();
    formData.append("video", file);
    const newUrl = `/stream?source=${file.name}&task=${task}`;
    Axios.axiosForm.post('/save_video', formData)
    .then((response)=>{
        if(response.status === 200){
            if (inputRef.current) {
                inputRef.current.value = "";
            }
            console.log("Video uploaded successfully ! ");
            onFileSelect(file);
            handleUrl(newUrl);
            // navigate(newUrl)
        }
    })
    .catch(e=>console.log(e));
};
```

Figure 43 - Handle file frontend

```
return (
    <div>
        {(file || isVideoLoaded) && (
            <div style={{ position: 'relative' }}>
                {file && (
                    <video ref={videoRef} style={{ display: 'none' }} autoPlay muted loop />
                )}
                {isVideoLoaded && (
                    <ReactPlayer
                        ref={videoRef}
                        width="100%"
                        height="100%"
                        playing
                        controls
                    />
                )}
                <canvas
                    ref={canvasRef}
                    width={dimensions.width}
                    height={dimensions.height}
                    style={{ border: '1px solid black' }}
                    onMouseDown={handleMouseDown}
                    onMouseUp={handleMouseUp}
                    onMouseMove={handleMouseMove}
                />
                <button style={{
                                position: 'absolute',
                                bottom: '10px',
                                right: '10px',
                                zIndex: 1,
                                fontSize: '16px',
                                padding: '10px 20px',
                                border: 'none',
                                borderRadius: '5px',
                                color: '#FFFFFF',
                                backgroundColor: '#ff6347',
                                cursor: 'pointer',
                                transition: 'background-color 0.3s ease'
                                }}
                        onClick={() => setLines([])}>

                        Clear Lines
                </button>
            </div>
        )}
    </div>
);
```

Figure 44 - Video Preview component

The **VideoPreview** component it's responsible for drawing the lines on top of the video in real time. So, the user will have to keep pressed left click for drawing the first line, and when released then the end point it's done for the first line. After 2 lines are drawn, we can update the parent with the array of line coordinates and it will redirect to the correct URL of the stream page.

```
  return (
    <div className="stream-container">
      {isLoading && <img src={loadingSpinner} className="loading-spinner" alt="Loading..." />}
      <img
        ref={imgRef}
        className="stream-img"
        alt="Wait for detection .."
        style={{display: isLoading ? 'none' : 'block'}}
        onLoad={handleImageLoad}
      />
      <div className="button-container">
        <button onClick={handleRetry} className="retry-button">Retry</button>
        <button onClick={handleStop} className="stop-button">Stop Stream</button>
      </div>
    </div>
  );
};
```

Figure 45 - Stream video page component

This is a React component for a video player in a streaming page of a web application. The component uses several hooks from React to manage state and side effects.

The **StreamPageVideoPlayer** function component receives two props: **url** and **isStreamStopped**. It uses the **useState** and **useRef** hooks to manage the state of the loading status and a reference to the image element, respectively.

The component defines several helper functions, such as **handleRetry** to retry loading the video stream, **handleStop** to stop the video stream, and **handleImageLoad** to set the loading state to false when the image is loaded.

The **useEffect** hook is used to perform side effects in response to changes in the component's props. It updates the **src** attribute of the image element whenever the **isStreamStopped** prop or the **url** prop changes. When the component is unmounted, it clears the **src** attribute of the image element to stop the video stream.

The component's render method returns a JSX element that includes an image element for displaying the video stream, a loading spinner that is displayed while the video is loading, and two buttons for retrying and stopping the video stream.

Overall, this component is responsible for managing the video stream in the streaming page, including loading the video, displaying a loading spinner, and providing retry and stop controls.

4.2.2 Backend

```python
from flask import Response, jsonify, request, Blueprint, render_template, make_response, send_from_directory
from flask import current_app
import os
import logging
import re

try:
    from .utils.download_manager import DownloadManager
    from .tracker.tracker_manager import TrackerManager
except:
    from utils.download_manager import DownloadManager
    from tracker.tracker_manager import TrackerManager

bp = Blueprint('video', __name__, url_prefix="/api")
MIME_TYPE = 'multipart/x-mixed-replace; boundary=frame'
INVALID_TASK_MSG = 'Invalid task type'
HTTP_BAD_REQUEST = 400
HTTP_OK = 200

logging.basicConfig(level=logging.INFO)

@bp.route('/', methods=['GET'])
def index():
    """Returns the index page."""
    return render_template('index.html')

@bp.route('/save_video', methods=['POST'])
def save_video():
    """Saves a video to the server."""
    file = request.files['video']
    filename = file.filename
    save_path = os.path.join(current_app.config['UPLOAD_VIDEO_FOLDER'], filename)
    file.save(save_path)
    response_data = {'message': f'Video {filename}  saved successfully', 'filename': filename}
    response = make_response(response_data, HTTP_OK)
    return response
```

Figure 46 - Save video implementation Flask

The **save_video** function is associated with the '/save_video' route and responds to HTTP POST requests. When a POST request is made to this route, the function retrieves the video file from the request using **request.files['video']**.

The filename of the video file is extracted and a save path is created by joining the filename with the 'UPLOAD_VIDEO_FOLDER' directory, which is specified in the application's configuration.The video file is then saved to the specified path on the server using the **file.save** method.

After the video file is saved, the function creates a response data dictionary that includes a success message and the filename. This dictionary is used to create a response using the **make_response** function, with a status code of HTTP_OK (200).

This endpoint is typically used in applications that need to receive and store video files from clients.

```python
@bp.route('/stream', methods=['GET'])
def video():
    """Streams a video based on the source and task provided in the query parameters."""
    source = request.args.get('source')
    line1 = request.args.get('line1')
    line2 = request.args.get('line2')
    task = request.args.get('task')
    upload_folder = current_app.config['UPLOAD_VIDEO_FOLDER']

    if source is None or task is None:
        logging.error("Required query parameters 'source' and 'task' not provided.")
        return jsonify({'error': 'Required query parameters not provided'}), HTTP_BAD_REQUEST

    source = "0" if source == "0" else (source if re.match(r'^(https?|udp)://', source) else os.path.join(upload_folder, source))

    print(source)
    tracker_manager = TrackerManager()
    tracker = tracker_manager.create_tracker(task, source, line1, line2)

    if tracker:
        return Response(tracker(), mimetype=MIME_TYPE)
    else:
        return jsonify({'error': INVALID_TASK_MSG}), HTTP_BAD_REQUEST
```

Figure 47 - Stream get request providing the inference

The **video** function is associated with the '/stream' route and responds to HTTP GET requests. When a GET request is made to this route, the function retrieves the source, line1, line2, and task parameters from the request's query parameters using **request.args.get**.

The source parameter is expected to be either a URL (http or udp), the string "0" (which represents the default camera), or a filename of a video file stored in the 'UPLOAD_VIDEO_FOLDER' directory on the server. The task parameter is expected to be a string that specifies the type of tracking task to perform.

If either the source or task parameters are not provided, the function logs an error message and returns a JSON response with an error message and a HTTP_BAD_REQUEST (400) status code.

If the source and task parameters are provided, the function creates a **TrackerManager** object and uses it to create a tracker for the specified task and source. If the tracker is successfully created, the function returns a streaming Response with the tracker's output and a MIME_TYPE (typically 'multipart/x-mixed-replace').

If the tracker cannot be created (for example, if an invalid task is specified), the function returns a JSON response with an error message and a HTTP_BAD_REQUEST status code. This endpoint is typically used in applications that need to perform real-time video tracking tasks.

```python
@bp.route('/download', methods=['GET'])
def download():
    """Downloads a YouTube video and sends it as a file response."""
    youtube_url = request.args.get('url')
    save_path = current_app.config['UPLOAD_VIDEO_FOLDER']

    if youtube_url is None:
        logging.error("Required query parameter 'url' not provided.")
        return jsonify({'error': 'Required query parameter not provided'}), HTTP_BAD_REQUEST

    filename = DownloadManager.download_youtube_video(youtube_url, save_path)

    if filename:
        file_path = os.path.join(save_path, filename)
        response = make_response(send_from_directory(save_path, filename, as_attachment=True))
        response.headers['Content-Disposition'] = f'attachment; filename="{filename}"'
        response.headers['Access-Control-Expose-Headers'] = 'Content-Disposition'
        return response
    else:
        return 'Failed to download the YouTube video'
```

Figure 48 - Download request for YouTube video

This Python function, **download**, is a Flask route at '/download' that responds to HTTP GET requests. It downloads a YouTube video based on the 'url' query parameter and sends it as a file response. The video is saved in the server's 'UPLOAD_VIDEO_FOLDER'.

If the 'url' parameter is missing, it returns an error message and a HTTP 400 status code. If the 'url' is provided, it uses the **DownloadManager** to download the YouTube video and save it to the specified path.

If the download is successful, it creates a file response with the downloaded video and sets the 'Content-Disposition' header to specify the filename in the response. It also sets the 'Access-Control-Expose-Headers' to allow the 'Content-Disposition' header to be read by the client. If the download fails, it returns a failure message.

4.3 YOLO implementation

```python
class ObjectTracker(BaseTracker):
    def __init__(self,
                 source=0,
                 model_name="yolo-highway-v2.pt",
                 lines_coords=[],
                 line_color=(0, 0, 255),
                 text_color=(255, 255, 255),
                 line_font=cv2.FONT_HERSHEY_SIMPLEX,
                 line_font_scale=1,
                 line_thickness=2,
                 text_thickness=2,
                 box_thickness=2,
                 box_text_thickness=1,
                 box_text_scale=0.5,
                 box_text_padding=2
                 ):
        """
        Initializes the ObjectTracker.

        Args:
            source (str or int): The path to the video source or camera index. Default is 0 (camera).
            lines_coords (list): The coordinates of the counting lines. One example is [(268,435),(592,447)],
                                 [(600,450),(950,450)].
        """
        super().__init__(source,model_name)
        self.current_dir = os.path.dirname(os.path.abspath(__file__))
        self.model = self.load_model()

        #! Check the available device (CPU or GPU)
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        print("Using Device: ", self.device)

        #? Get the class names from the model
        self.class_name_dict = self.model.model.names

        #% Initialize the box annotator for drawing bounding boxes and labels
        self.box_annotator = sv.BoxAnnotator(
                color=sv.ColorPalette.default(),
                thickness=box_thickness,
                text_thickness=box_text_thickness,
                text_scale=box_text_scale,
                text_padding=box_text_padding)

        #% Create instances of Line, TotalCountRectangle, and SpeedEstimation classes
        self.speed_estimation = SpeedEstimation(class_name_dict=self.class_name_dict,box_annotator=self.box_annotator)
        self.lines = [Line(self.class_name_dict,
                           line_coord=coords,
                           line_color=line_color,
                           text_color=text_color,
                           line_font=line_font,
                           line_font_scale=line_font_scale,
                           line_thickness=line_thickness,
                           text_thickness=text_thickness) for coords in lines_coords]
        self.total_counts = TotalCountRectangle(self.class_name_dict)
```

Figure 49 - Object Car Tracker implementation

The **ObjectTracker** class in the traffic analysis system uses machine learning models for real-time object tracking and analysis. It initializes with parameters like video source, model name, and line coordinates. The class uses a pre-trained model loaded onto the CPU or GPU. It incorporates helper classes like **BoxAnnotator** for drawing bounding boxes, **SpeedEstimation** for speed calculation, **Line** for counting lines, and **TotalCountRectangle** for displaying total object counts. Thus, **ObjectTracker** provides a comprehensive framework for real-time object tracking in video feeds.

```python
def load_model(self):
    """
    Loads the YOLO model.

    Returns:
        YOLO: The loaded YOLO model.
    """
    model_path = os.path.join(
        self.current_dir, "models",self.model_name)
    model = YOLO(model=model_path)
    model.fuse()
    return model

def convert_result_to_detections(self, result):
    """
    Converts the YOLO result into Detections format.

    Args:
        result: The result from YOLO model.

    Returns:
        Detections: The converted detections.
    """
    detections = sv.Detections(
        xyxy=result.boxes.xyxy.cpu().numpy(),
        confidence=result.boxes.conf.cpu().numpy(),
        class_id=result.boxes.cls.cpu().numpy().astype(int),
        # tracker_id=result.boxes.id.cpu().numpy().astype(int)
    )
    if result.boxes.id is not None:
        detections.tracker_id = result.boxes.id.cpu().numpy().astype(int)
    return detections
```

Figure 50 - Load model and detections functions for tracker

The **load_model** function in the **ObjectTracker** class loads the YOLO model from the specified path, fuses the model for optimization, and returns it. The **convert_result_to_detections** function transforms the output from the YOLO model into a format suitable for further processing. It takes the model's result as input and returns a **Detections** object, which includes bounding box coordinates, confidence scores, class IDs, and tracker IDs. This function ensures the model's output is in a standardized format, facilitating subsequent steps like object tracking and speed estimation.

```python
def process_frame(self, result, frame_counter_for_tracker_id, speed):
    """
    Processes a single frame of the video stream.

    Args:
        result: The result of the YOLO model's track method for the current frame.
        frame_counter_for_tracker_id: A dictionary storing the frame counter for each object tracker.
        speed: The current speed value.

    Returns:
        The annotated frame with object detections and counters.
    """
    frame = result.orig_img

    detections = self.convert_result_to_detections(result)

    #? Count objects and update counts for each object (rectangle and lines)
    for line in self.lines:
        line.count(detections)
    self.total_counts.count(detections)

    #? Add object annotations to the frame
    frame_counter_for_tracker_id =  self.speed_estimation.update_speed_and_frame_counter(speed,
                                    detections,frame_counter_for_tracker_id)
    self.speed_estimation.update_tracked_coordinates()
    frame = self.speed_estimation.add_annotations_to_frame(frame, detections)

    #% Draw lines and total counts
    for line in self.lines:
        frame = line.draw(frame)
    frame = self.total_counts.draw(frame)

    return frame, frame_counter_for_tracker_id
```

Figure 51 - Process frame method of object tracking

The **process_frame** function in the **ObjectTracker** class is responsible for processing a single frame of the video stream. It takes as input the result of the YOLO model's track method for the current frame, a dictionary storing the frame counter for each object tracker, and the current speed value. The function first converts the YOLO model's result into a **Detections** object. It then counts the detected objects and updates the counts for each object. The function also updates the speed and frame counter for each tracked object and adds object annotations to the frame. Finally, it draws the counting lines and total counts on the frame and returns the annotated frame along with the updated frame counter dictionary. This function is crucial for real-time object tracking and speed estimation in the video stream.

```python
def __call__(self):
    """
    Runs the object tracking and counting process.
    """
    frame_counter_for_tracker_id = {}
    speed = 0

    if (self.is_youtube_url(str(self.source))):
        video = pafy.new(self.source)
        best = video.getbest(preftype="mp4")
        cap = cv2.VideoCapture(best.url)
    else:
        cap = cv2.VideoCapture(process_source(self.source))

    assert cap.isOpened()
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

    while True:
        ret, frame = cap.read()
        assert ret
        for result in self.model.track(source=frame,persist=True, stream=True, agnostic_nms=True, tracker="bytetrack.yaml"):

            frame, frame_counter_for_tracker_id = self.process_frame(result, frame_counter_for_tracker_id, speed)
            ret, buffer = cv2.imencode('.jpg', frame)
            frame = buffer.tobytes()
            yield (b'--frame\r\n' b'Content-Type: image/jpeg\r\n\r\n' + bytearray(frame) + b'\r\n')
```

Figure 52- Call method for car object tracker

The **__call__** method in the **ObjectTracker** class is the primary driver of the object tracking and counting process. It initializes a dictionary, **frame_counter_for_tracker_id**, to track the frame count for each object tracker and a **speed** variable for speed estimation. The method then checks the video source. If it's a YouTube URL, it uses the **pafy** library to fetch the best quality stream. If not, it assumes it's a local video file and opens it directly using OpenCV's **cv2.VideoCapture** function. It then sets the frame width and height to 1280x720 pixels, standardizing the video input size for the object detection model.

The method enters a loop, reading frames from the video one by one. For each frame, it applies the YOLO model's **track** method, performing object detection and returning a result that includes the original image and detected bounding boxes. This result is passed to the **process_frame** method, which converts the result into a suitable format for counting, counts the objects, updates the speed and frame counter, and adds annotations to the frame. The processed frame is encoded into JPEG format and converted to bytes, allowing the method to produce a stream of processed frames for real-time video display with object tracking and counting annotations.

4.4 Demo

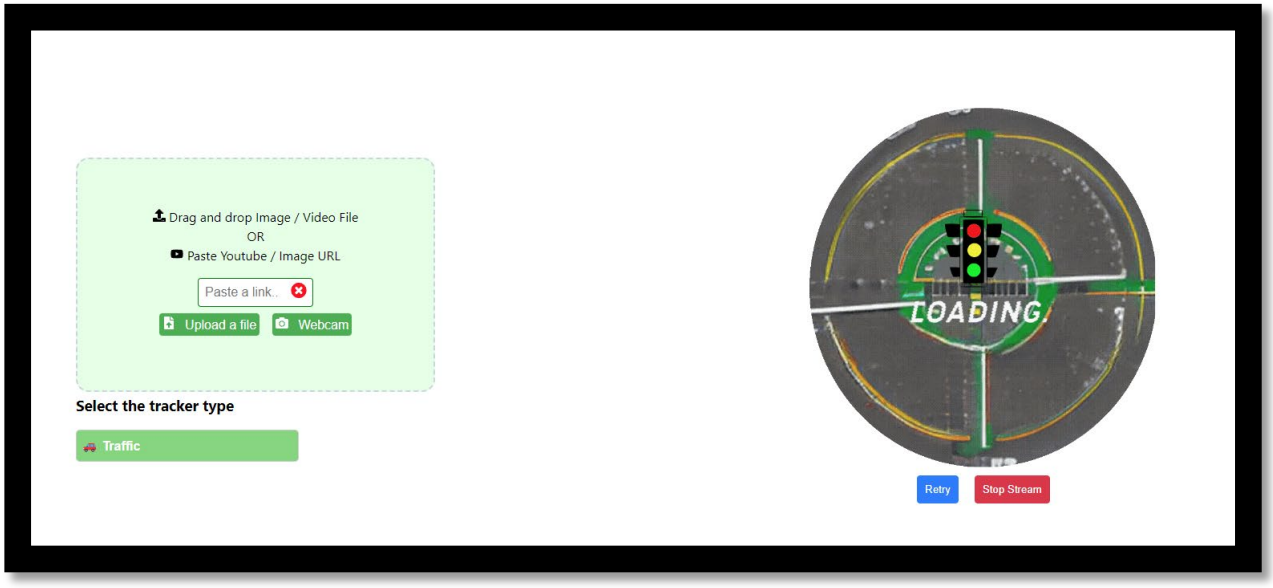

Figure 53 - User interface of the stream page

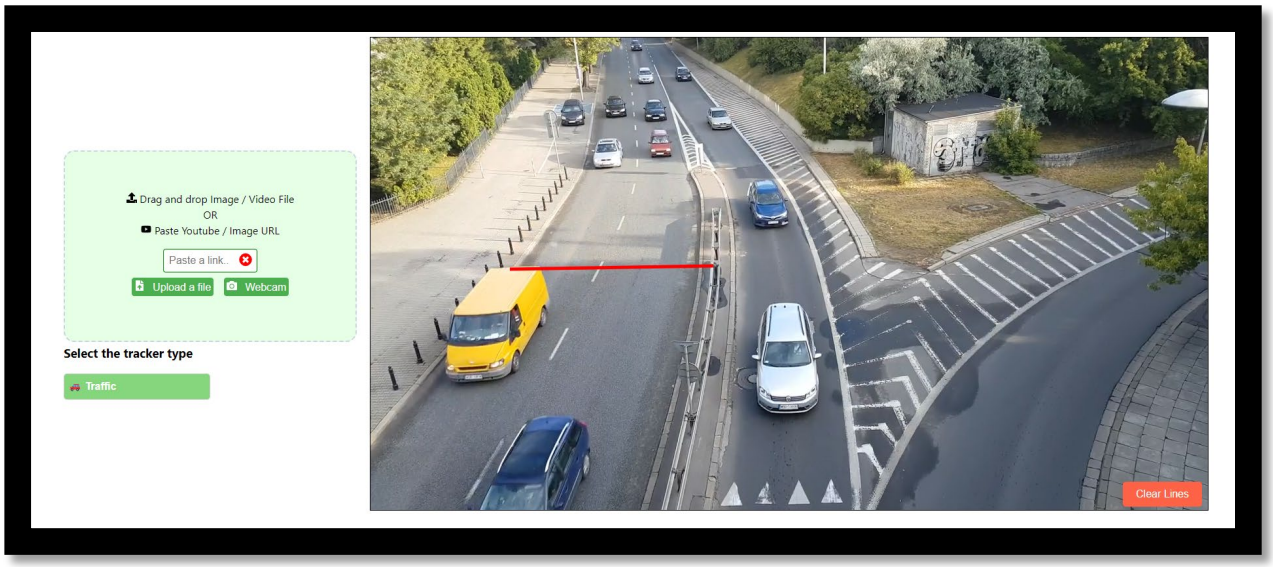

Figure 54 - Selection of lines positions on top of the video

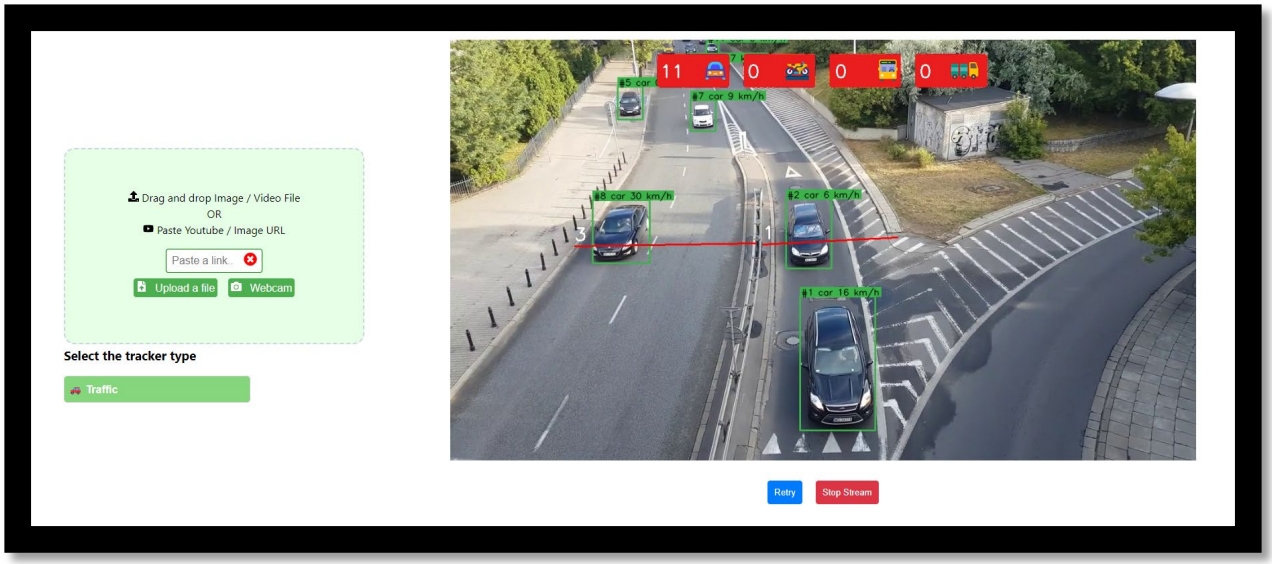

Figure 55 - Display count, speed and tracks of the vehicles

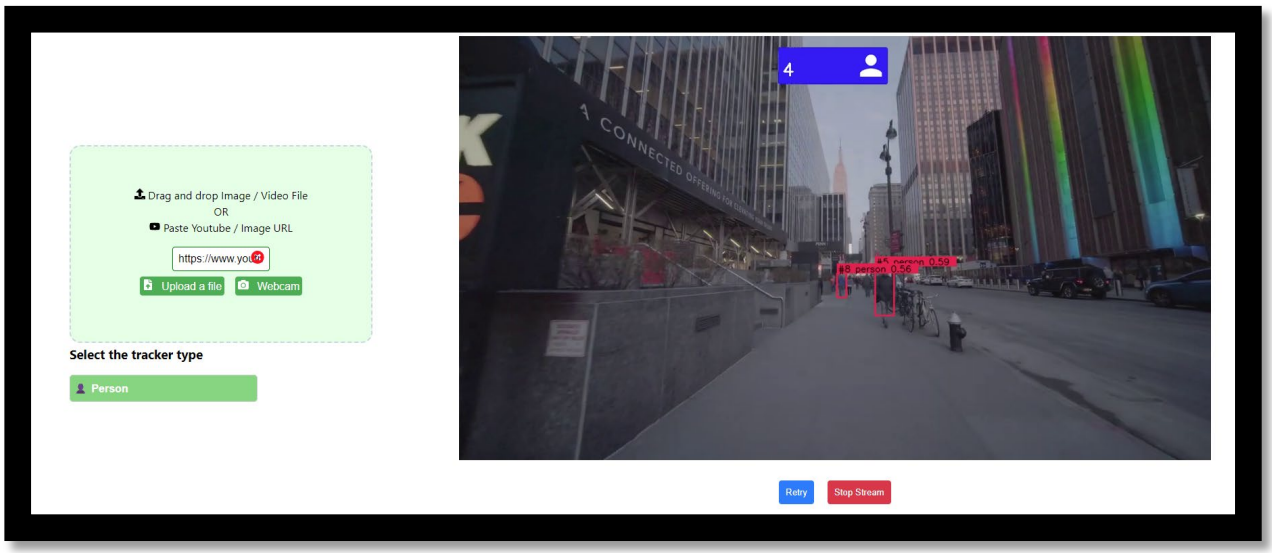

Figure 56 - Display people count from YouTube video

5 CONCLUSION AND FUTURE WORK

The journey of the Traffic Analytics Application project has been a comprehensive exploration into the field of traffic management and object detection. This exploration was not just a simple task, but a deep dive into the intricacies of these fields, with the aim of developing an application capable of real-time object detection and traffic analysis. The project has not only met this objective but exceeded expectations, delivering a robust and efficient application that stands as a testament to the power of modern technology.

The motivation behind the project was a desire to enhance traffic management and contribute to the development of smart cities. This was not just a lofty goal, but a practical response to the challenges faced by modern urban environments. By leveraging the power of machine learning and computer vision, the application interprets traffic patterns in a way that traditional systems cannot. The heart of this interpretation lies in the YOLOv8 model, a cutting-edge model renowned for its accuracy and speed in object detection. This model, combined with our application, provides a powerful tool for traffic analysis.

The application itself is split into two primary components: the backend and the frontend. The backend, the powerhouse of the application, is responsible for the core functionalities. It harnesses the power of technologies such as CUDA GPU architecture and YOLO V8 to perform complex computations and object detection tasks. The frontend, on the other hand, is where users interact with the application. Developed using React, it provides an intuitive and user-friendly interface that makes the power of the backend accessible to users.

The assembly of the application was facilitated by a suite of modern development tools. Docker, a platform that automates the deployment of applications inside lightweight containers, was employed for containerization. FFmpeg, a leading multimedia framework, was used for video streaming. Webpack and Nginx were used for serving the frontend, ensuring a smooth and responsive user experience. The Flask application configuration, a micro web framework written in Python, was instrumental in efficiently handling HTTP requests.

Despite the successful completion of the project, the journey is far from over. There are opportunities for further enhancement and expansion that promise to take the application to new heights. One potential area of future work is the integration of Optical Character Recognition (OCR) technology. The incorporation of OCR would enable the application to detect and read license plates, thereby

providing more detailed traffic data. This would not only enhance the functionality of the application but also open new avenues for personalized traffic management and security applications.

Another potential area for improvement is the communication method of the application. The current method, while functional, could be replaced with websockets to facilitate more efficient and real-time communication. This enhancement would not only improve the application's performance but also provide users with real-time updates, thereby enhancing the overall user experience.

The application's potential for improvement is not limited to the integration of OCR technology and the transition to websockets. There are several other areas where the application could be enhanced to provide a more robust and user-friendly experience.One such area is the implementation of an event-based system in the user interface, facilitated by websockets. This would allow the application to store key metrics of the system in the database and display them in real-time using dynamic charts. This real-time visualization of data would provide users with immediate insights into traffic patterns and system performance, enhancing the overall user experience.

In addition to real-time data visualization, the application could also be upgraded to provide real-time notifications to users when certain events occur. For instance, the application could alert users when vehicles exceed the speed limit. These notifications could be displayed in real-time and also stored in the database for future reference. This would not only provide immediate feedback to users but also create a historical record of traffic violations for further analysis.

Another potential area for improvement is the speed estimation algorithm. The current algorithm calculates speed by measuring the distance in pixels per meter and dividing it by time. This is achieved by using the interval of frames to compute the distance and time. While this method is functional, it could be enhanced to provide more accurate speed estimations. This could involve the use of more advanced algorithms or the integration of additional data sources to improve the accuracy of speed measurements.

In conclusion, the Traffic Analytics Application project has been a successful endeavor in leveraging technology for traffic management. The project has not only achieved its objectives but also laid the groundwork for future improvements. The potential integration of OCR technology and the transition to websockets are exciting prospects that promise to make the application even more robust and versatile in the future. As we look ahead, the journey of the Traffic Analytics Application project continues, with the promise of more innovations and improvements in the field of traffic management. This is not the end, but rather the beginning of a new chapter in the story of the Traffic Analytics Application.

6 BIBLIOGRAPHY

1) Choi, J., Chun, D., & Oh, H. (2020). Object Detection Based on YOLOv3 and Distance Measurement Using Depth Camera. In 2020 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1205-1207). IEEE.
2) Wang, Y., Zheng, L., & Xue, M. (2019). Transport efficiency of urban arterial road traffic: A case study of Shanghai. Sustainability, 11(3), 681
3) Liu, H., Tian, Z., Li, Y., Zhang, M., & Bigham, J. (2020). Real-time and Predictive Traffic Management Using Artificial Intelligence: A Survey. arXiv preprint arXiv:2001.10993.
4) Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767.
5) Haoyi Xiong et all "ScenarioNet: Open-Source Platform for Large-Scale Traffic Scenario Simulation and Modeling". Link
6) Yu Zheng, Junbo Zhang, Dekang Qi. "Deep Spatio-Temporal Residual Networks for Citywide Crowd Flows Prediction" Link
7) Gonzalez, R.C., Woods, R.E., 2002. Digital Image Processing (3rd Edition). Prentice-Hall, Inc., USA
8) Pham, T. D., Xu, C., & Prince, J. L. (2000). Current methods in medical image segmentation. Annual review of biomedical engineering, 2(1), 315-337
9) Redmon, J., Divvala, S., Girshick, R., Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788)
10) Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640.
11) Ivanescu, R. C., Nagy, R., Ofiteru, A. I., Comanescu, C., & Manda, A. (2022, November). Parallel vs Distributed Edge Detection for Large Medical Image Datasets. In 2022 E-Health and Bioengineering Conference (EHB) (pp. 1-4). IEEE.
12) Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. http://www.deeplearningbook.org
13) Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285.
14) Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In European conference on computer vision (pp. 818-833). Springer, Cham.

15) Misra, D. (2019). Mish: A Self Regularized Non-Monotonic Neural Activation Function. arXiv preprint arXiv:1908.08681.
16) Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934.
17) Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv preprint arXiv:1804.02767.
18) Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934
19) Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., ... & Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In European conference
20) Rosebrock, A. (2017). Intersection over Union (IoU) for object detection. PyImageSearch

7 WEB REFERENCES

- [ROH23] – Rohit Kundu, YOLO: *Algorithm for Object Detection Explained [+Examples], 2023,* available at the address: https://www.v7labs.com/blog/yolo-object-detection
- [AARO23] - Aaron Morrison, *NVIDIA CUDA ARCHITECTURE, 2023*, available at the address: https://cloud2data.com/nvidia-cuda-architecture-3/
- [ROBERT14] Robert Picard, *Organizing your project*, 2023 (Source: https://exploreflask.com/en/latest/organizing.html)
- [PALL10] Pallets*, Blueprints and Views*, 2010 (Source: https://flask.palletsprojects.com/en/2.3.x/tutorial/views/)
- [REACT23] React Documentation, *Start a New React Project* ,2023, (Source: https://react.dev/learn/installation)
- [KRUP22] Krupa Bhimani, *Serve Python App on Nginx,*2022, (Source: https://blog.devgenius.io/serve-python-app-on-nginx-6bc57ceaed4c)
- [OBAID23] Obaid Al Teneiji, *Docker with Webcam* (Source: https://github.com/i99dev/Docker-with-WebCam)