UNIVERSITÀ
DI SIENA
1240

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE E SCIENZE MATEMATICHE

Corso di laurea Magistrale in
Computer and Automation Engineering

SETUP OF A DEEP LEARNING-BASED SYSTEM FOR DETECTING QUEUES ON HIGHWAYS USING Z2 BOARD, EVALUATION OF PERFORMANCE AND POWER CONSUMPTION

Relatore:
Prof. Alessandro Mecocci
Prof. Sandro Bartolini

Candidato:
Vahid Farjood Chafi

Anno Accademico 2021 – 2022

Abstract

Artificial Intelligence plays an important role in shaping the current and future ways of mobility and transportation. In recent years, identifying vehicles and measuring their speed from pictures or videos have become crucial tasks for any ITS (Intelligent Transportation Systems) and it remains challenging. We can expect that using AI application, transport will be more efficient and safer. For instance, an ITS can predict a potential traffic, navigates drivers through another path, prevent potential accidents, it can also be used to control traffic lights in a much better way and decrease the possibility of a traffic jam.

So far, the field of traffic estimation is governed mostly by RADAR, LIDARS, GPS, and section speed measurements, but those techniques are not able to provide semantic information and they are usually expensive. Conversely, nowadays cameras are cheap and with the help of Deep Learning and computer vision techniques we are able to extract many useful information from videos or images captured by those cameras such as type of the vehicle or estimating distance between vehicles.

However, implementing Deep Learning models require a lot of resources and most projects have been tested on the servers in which there were enough GPUs and memories to run Deep Learning models. On the contrary, in this thesis we are interested in implementing Deep Learning models on small and cheap devices that are limited by the hardware and power supply and software dependencies/compatibilities.

The main goal of this project is to implement a Deep Learning model on an embedded device which is designed by University of Siena, and then install it on the highway with a small camera to detect the existence of a traffic queue. In addition, some other quantitative information such as number of vehicles, type and speed of the vehicle, density of the road or their lane number will be also collected and eventually transmitted to a server via network. Moreover, I also tried to assess the board's performance and its power consumption.

Acknowledgement

First and foremost, I have to thank my research supervisors, Prof. Alessandro Mecocci, Prof. Sandro Bartolini. This paper would have never been finished if it weren't for their help and committed participation in every step of the procedure. I sincerely appreciate all your help and patience over the past year.

Additionally, I would like to express my thanks to my instructors from the Mathematical and Engineering Department, particularly Prof. Marco Gori, Prof. Edmondo Trentin, Prof. Stefano Melacci, Prof. Marco Maggini, and Prof. Monica Bianchini. My first-year lecturer at Siena University was Prof. Marco Gori, I will always have fond recollections of his lessons because of the manner he taught and his excitement for the field of machine learning.

I started attending Siena University in September 2019 and have been quite productive there. Working with so many teachers has been an amazing experience. I owe a lot of the analyses in my thesis to my experience at Siena University. I have a huge number of individuals to thank for their patience and, at times, willingness to put up with me over the past three years to help me finish my dissertation, which required more than just academic assistance. The people who have supported me personally and professionally throughout my stay at the university are Pouya Hosseinzadeh, Saber Alilou, Mirco Mannino, Alexandru Marocico, Claudio Grassi, and Mehraveh Ala. My thanks and appreciation for their friendship go beyond words.

Finally, none of this would have been possible without my family, especially my wife. My family, who, despite my own little commitment to communication, provided them support through weekly phone conversations. To my wife: To say that we have had some ups and downs as a family over the last three years would be an understatement. You stopped me every time I was about to give up, and for that, I will always be thankful. This dissertation is evidence of the unwavering support and love you have given me.

Content

List of Figures

Chapter 1 - Introduction

1.1 Overview

With the continuous increase of the number of vehicles in the road, traffic management are becoming very difficult, and it requires better traffic surveillance system. Having more vehicles, would potentially raise up issues like congestion, air pollution, accidents on the road each year. Traffic monitoring with an intelligent transportation system provides solutions to different challenges, such as vehicle counting, speed estimation, accident detection, and assisted traffic surveillance [1]. Such a traffic monitoring system would require to properly detect vehicles that are on an image or video frame and localize their positions while they stay in the scene, that is called object detection; Object detection is one of the most important computer vision challenge that deals with detecting and localizing visual objects of a certain class (such as humans, animals, or cars) in digital images [2].

Afterward, we need to track the object in the subsequent frames to identify each vehicle properly. The problem gets more complex once the number of vehicle increases. Tracking multiple objects, also known as Multiple Object Tracking (MOT), is largely partitioned into locating multiple objects, maintaining their identities, and yielding their individual trajectories given an input [3]. Having properly detected and identified all vehicles in each frame, I consider the task of traffic estimation as a classification problem. I used three different features, namely, vehicle counting, road density, average flow speed, which were extracted from detection and tracking, to classify the level of a traffic queue on the highway. The level is categorize into: Low, Medium, and High.

All the mentioned steps will be run on a small, embedded board which has an ARMv7 architecture with a camera and with limited power consumption. The board is assumed to be installed on a bridge/pole above a highway and camera should be calibrated in a way that can capture top-down and rear-view images of vehicles. The board will be turned on every few minutes (e.g. about 5, 10 or more) and each time it takes two consecutive images with 1 second delay between them. Then these two images are used as the input of the object detection algorithm which will collect all the necessary information like the name or bounding boxes of the vehicles or center point of the object. Those bounding boxes with their corresponding center point, are fundamental elements in

tracking algorithm but tracking is achieved based on some assumptions, like a vehicle will remain in the same lane during two frames or the shape of a box is not incremental regarding to camera view.

Based on two assumptions, I investigate a simple tracking strategy, by checking if two center points of bounding boxes in two frames are in the same lane number, and the area of the first frame is larger than the area of the box on the second frame, then algorithm identifies the vehicle as the one in the previous frame. Once we identify all the vehicles, then we can start to extract some quantitative information.

Firstly, the corresponding lane number of each vehicle will be kept in each vehicle's attribute buffer. Secondly, approximate speed of each vehicle will be estimated based on its distance in two frames. A proper speed estimation algorithm requires a precise camera calibration and specifying a pre-measured ground plane distance on the road; that is require when we do not use any sensors like LIDAR.

Basically, dealing with this task, requires perspective projection; also, it would be necessary to deal with unknown distance from camera to the ground plane of the road and possibly with radial and tangential distortion [4]. In this project, speed estimation task is relaxed by defining ground plane distance of the road based on standard white line length on the road. According to United State Federal Highway Administration guideline, the broken lines should consist of 3 m (10 ft) line segments and 9 m (30 ft) gaps [5]. Therefore, we can calculate an approximate ground distance of a given image without camera calibration. Then, speed estimation task would be simply to calculate the distance of a vehicle between two frames divided by the number of frames per second (fps), and fps in this project is equal to one because we are using two consecutive frames with delay 1 second.

In order to have some quantitative information, we can use extracted information of each vehicle from previous tasks, such as their average speed, bonding box, center point, lane number, or area. Then we can calculate three important features, which are, number of vehicles, road's density, and road's average flow speed. Based on those three features, we can simply use a classifier, and predict traffic quality in three levels: low, medium, and high. The classifier that I used in this project is a linear Support Vector Machine (SVM) model.

Finally, all the results from detection and tracking and estimation modules will be saved in a text file with all details. This file is supposed to be sent to a server via network for later analysis and in the future, it might be feed to a more accurate and sophisticated model for traffic estimation task.

The layout of the entire pipeline is shown in Figure 1.

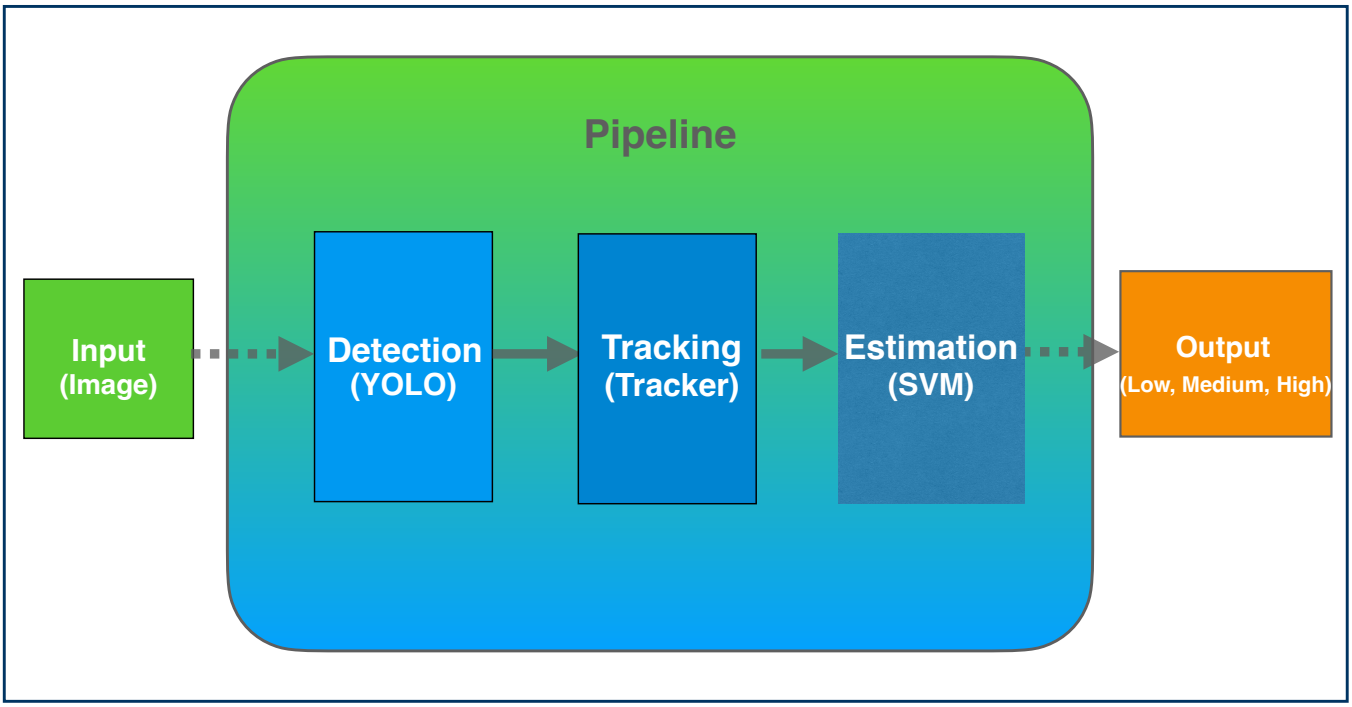


Figure 1: The full pipeline of program procedures

Chapter 2 - Device Specification

In this chapter, I will look at different part of the Z2 board which runs the final application on the highway. I will explain the hardware aspect of the board and some information related to the software aspect.

2.1 Hardware

The Z2 board is a single-board computer (SBC), which is an entire computer with all of the necessary components for operating, including an ARM Cortex-A series processor, memory, input/output (I/O) and other features that are required for a complete computer capability. Single-board computers are commonly used as educational systems, or for use as embedded systems. An embedded system is conventionally defined as a piece of computer hardware running software designed to perform a specific task. Examples of such systems might be TV set-top boxes, smart cards, routers, disk drives, printers, automobile engine management systems, MP3 players or photocopiers [6].

Embedded systems can contain very simple 8-bit microprocessors, such as an Intel 8051 or PIC micro-controllers, or some of the more complex 32 or 64-bit processors, such as the ARM family [6] which we have in the Z2 board.

2.1.1 Processor

The ARM architecture supports different implementations for a very wide range of performance points. The simplicity of the architecture provides solutions for small implementations, and it enables a very low power consumption. The Cortex-A series processors covered in this board is the family of Cortex-A5 model which conform to the ARMv7-A architecture [figure-1].

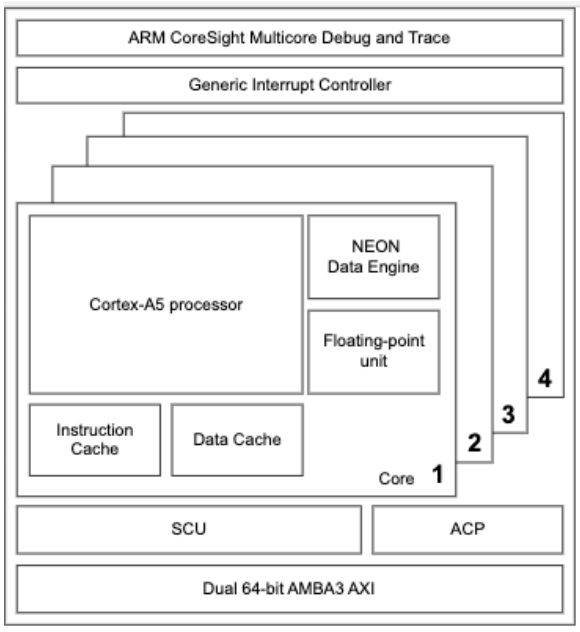


Figure 1: Cortex-A5 processor [6].

The Cortex-A5 processor has the following features:

- Full application compatibility with other Cortex-A series processors.
- Multiprocessing capability for scalable, energy efficient performance.
- Optional Floating-point or NEON units for media and signal processing.
- High-performance memory system including caches and memory management unit.
- High value migration path from older ARM processors. [6]

2.1.2 Memory

The Z2 board uses a 256 MB nominally memory, and in a normal condition half of the memory space is used by operating system and the remaining space (174MB) would be available for users. It has 7GB disk drive with at least 3.5GB available for the users. It can also be tuned to use swap memory, but I did not use swap memory.

2.1.3 Modules

Mainly, the Z2 board has two miniPCIe modules for different purposes for instance a VPU accelerator, Camera, Wireless module can be plugged in to it. However, depending on the type of module, it can be either directly connected to the board via miniPCIe, or it can be connected via a miniPCIe-To-Usb convertor, which is the case I used in this project. Therefore, I provide two miniPCIe-To-Usb convertors; one of them is used for a Usb-WiFi connector and another one for a Usb-Camera.

Below is the list of all the modules used in the Z2 board:

1. Two miniPCIe-To-Usb
2. USB-WiFi Connector (Realtek Wireless Lan)
3. USB-Camera (Logitech webcam HD C920)
4. Serial-To-Usb (for management via console)

The picture of the board with all the mentioned modules is illustrated in the figure 2.2.

2.1.4 Battery

In many embedded systems the power source is a battery, and programmers and hardware designers must take great care to minimize the total energy usage of the system. This can be done, for example, by slowing the clock, reducing supply voltage, or switching off the core when there is no work to be done.

The Z2 board, can use about 7.5V minimum battery up to 15V. In our case a 14.4V (nominal voltage) battery was used. Voltage and capacity of the battery can be tuned to accommodate the energy footprint of the application and meet the energy requirements in the field (e.g. harvesting with external solar panel). In this project, since I use two miniPCIe convertors, one for WiFi converters and one for Usb-Camera, I provided a 14.8V battery.

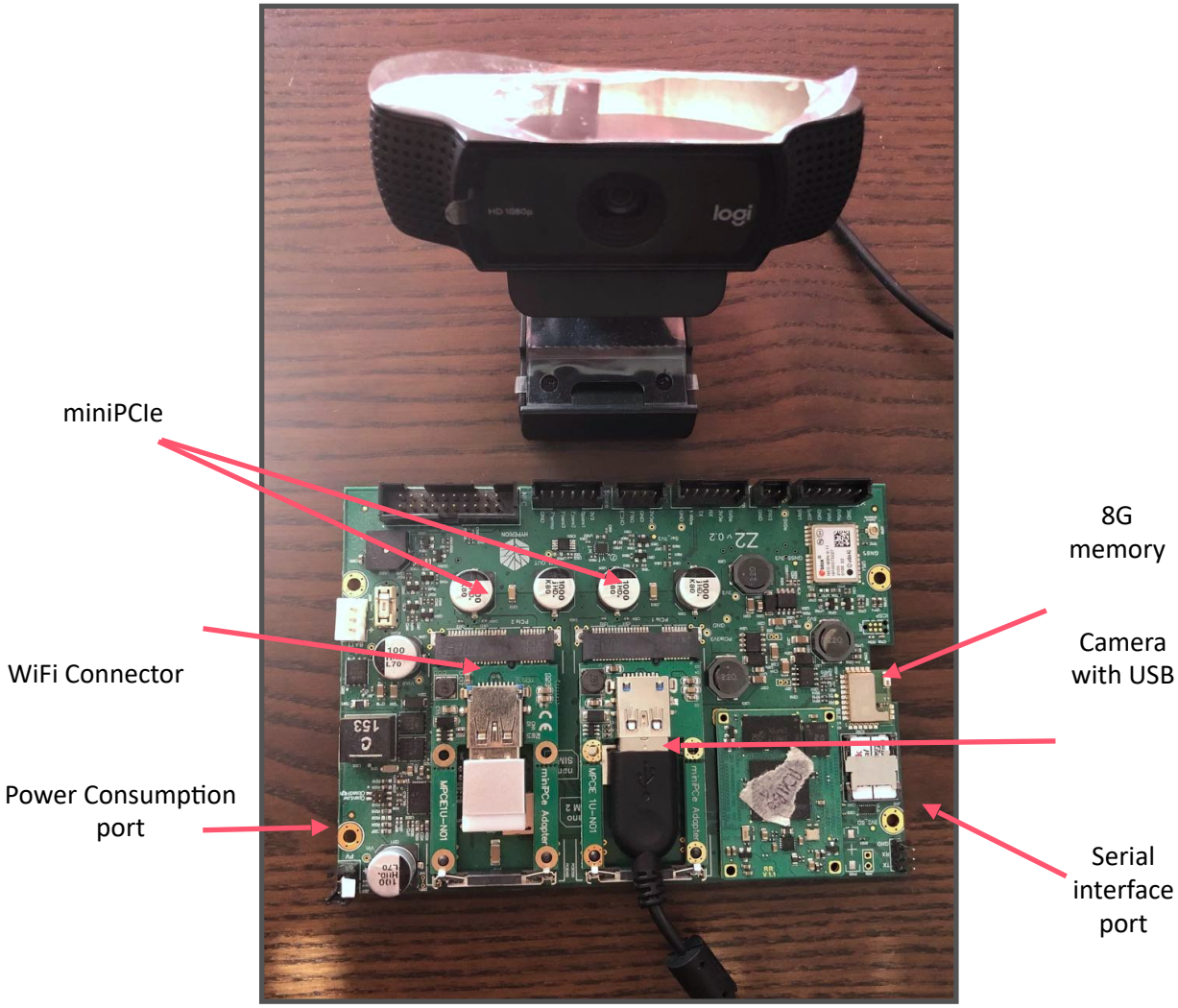


Figure 2: Z2 board with wifi and camera modules.

2.1 Software

In this section, we will look at different software aspects of the project that will be run on the Z2 board, such as type of operating system, build system, programming languages, etc. the Z2 board, just like Raspberry Pi, is a very cheap computer that runs Linux as its operating system, and it also provides a set of GPIO (general purpose input/output) pins, allowing you to control electronic components for edge computing, and explore the Internet of Things (IoT). Since it uses Linux as its operating system, as a result, it runs a suite of open-source software and packages too.

2.2.1 Operating System

Linux is the most used operating system for single-board computer and embedded system by far. The operating system used in Z2 is Linux Roadrunner 4.19.134 kernel distribution. It is a free operating system based on Debian GNU/Linux which is one of the oldest operating systems and it has access to online repositories that contain over 51,000 packages and free software [7].

2.2.2 Programming Language

C or C++ are ideal programming languages for a lot of embedded devices. That's in part because they are highly effective and "compiled" languages. Compiled languages are quick and stable because the computer (or embedded device) executes the code, without intermediate management layers. On the other side, when using an interpreted language like Python, the code is interpreted by a program, which introduces significant overhead as a counterpart for some flexibility. In fact, the code is executed by another interpreter software running on top of the device. There is a research paper that shows the energy footprint differences between various languages and, apart from the little differences among similar languages, which are irrelevant, the paper witnesses that interpreted languages can have orders of magnitude time and energy overhead compared to interpreted high-efficiency languages. However, they are much easier for programmers to learn, read and write; Python and Java are the best example of interpreted languages. Python is the most used language for Machine Learning, and it supports different variety of popular Deep Learning libraries.

Having that comparison in mind, I took the advantages of Modern C++ for the main program since its faster and its more efficient. However, for training the Deep Learning models I used used Python. So basically, I tried to train and fine-tune two computer vision Deep Learning models,

namely, YOLO and MobileNet SSD on Google Colab with python code, and saved the model files in the board. Then I used those saved model for inferencing inside my main program which is written in C++ code.

2.2.3 Application structure

Program is designed in a modular base in which each task is implemented in a different file(Figure-3). All the different files are handled by the main.cpp file which is the core part of the application. First, the command-line parser will parse input arguments and will set all those arguments into a Parameter object which has all the variable needed to store input arguments, such as input file path or model path. Then, based on our selected detection model, program detects all the vehicles on a given image. The result of the object detector model is a '*std::vector<std::vector<Vehicle>> vehicle_vector;'* which means we will have a buffer that holds two other buffers, one of them will keep the detected vehicles for image number one and the other buffer for image number two. After the detection task is done, the result will be sent to the tracker module in order to identify all the vehicles from those two images and calculate their distance and speed. Finally, those tracked vehicles with all their quantitative information that we collected them during the detection and tracking, will be used to estimate the traffic based on the SVM classifier model in the estimator module. At the end, the result with all the necessary information, will be saved on the disk for later use.

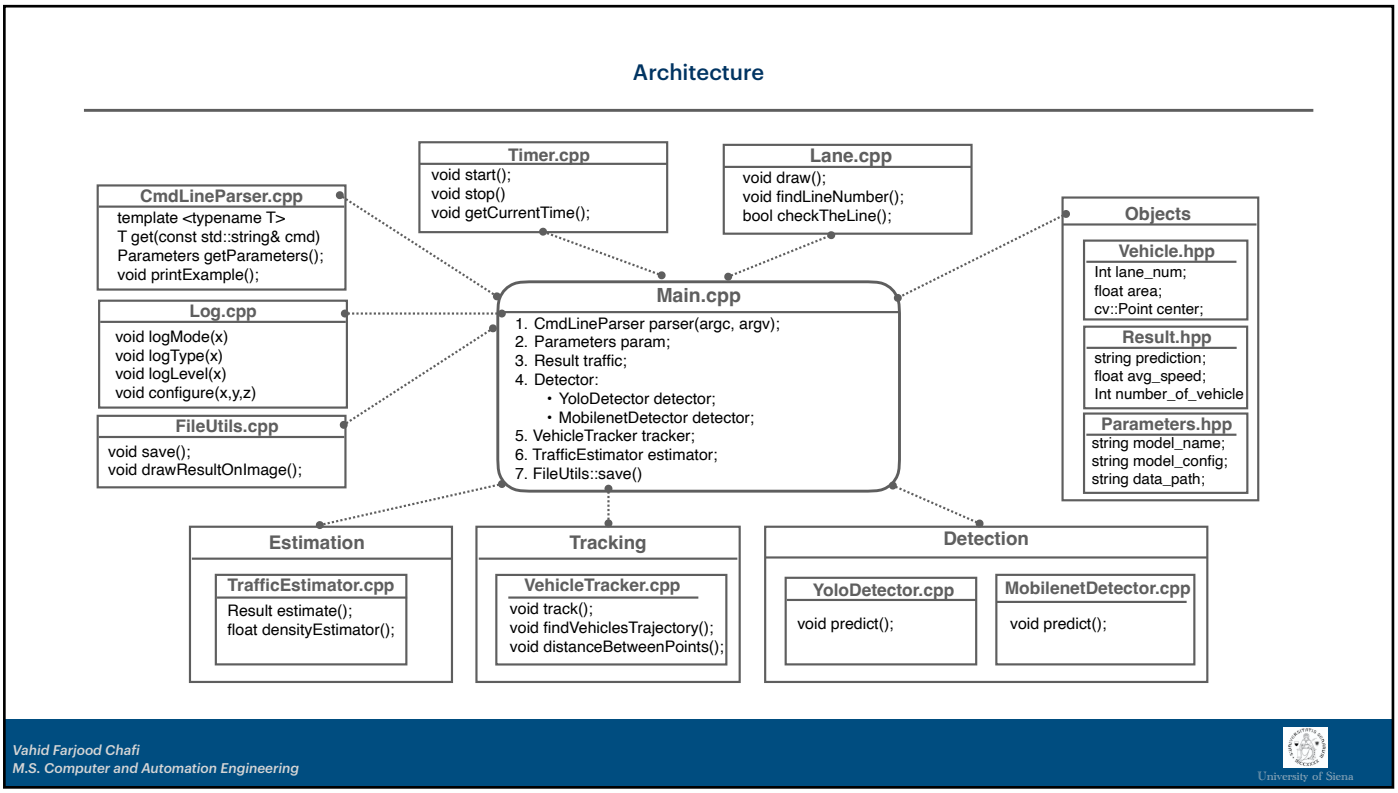


Figure 3: Software Architecture Diagram

I also provided a Log feature for the program in which we can define the log level and configure many other functionalities. The log can simply be shown either to the console, or to be saved into a file for later investigation.

2.2.4 Packages

Apart from many popular Deep Learning frameworks like TensorFlow or Pytorch, I used OpenCV as the main project library. The reason is that since I wanted to use C++ as the main language then I was limited to select Deep Learning libraries because most Deep Learning libraries are written for python and not C/C++. However, for some of those libraries, we also have C++ version like LibTorch, TensorFlow-Lite or Caffe, but unfortunately, most of them are not build for ARM processor, and those which are built for ARM processor, have a lot of compatibility issues and they are not light enough to be installed on the Z2 board as TensorFlow-Lite. Therefore, I used OpenCV with its DNN module which has capability of running Deep Learning inference on pre-trained models. OpenCV DNN module supports many popular Deep Learning frameworks such as: Caffe, TensorFlow, PyTorch, Darknet, and many others. So, for training and fine-tuning the models, I used PyTorch and TensorFlow on Google Colab, but for inference I used OpenCV in my main program.

2.2.5 Build System

The term build might refer to the process in which the source code is converted into a stand-alone software that could be run on a computer or the result of doing so [9]. The compilation process is one of the most important steps of building a software, where source code files will be converted into executable code. This process of building a software is usually controlled by a build tool [9].

CMake is a build tool that I used in my project to manage building of source code. It is widely used for the C and C++ languages, but it might also be used to build source code of other languages. CMake is an open-source, cross-platform family of tools designed to build, test and package software, but it is not a build system itself; it generates another system's build file. It is used to control the software compilation process using simple platform and compiler independent configuration files and generate native MakeFiles and workspaces that can be targeted in the compiler environment of your choice [10].

2.2.6 Cross-Compilation

The concept of cross-compiling is when you have a compiler that runs on platform A (the **host machine**) but generates executables files for platform B (the **target machine**) [11]. Usually, in embedded world the operating system and hardware of your current machine are always going to be different from your target machine which in our case it is the Z2 board with ARM processor. It is easy and takes a few moments to build a cross-compiler that can generate executable files for the target machine and by doing so you can save a lot of time that you might spend on fixing and debugging your code. On the contrary, it might take a while to build the same executable files on slower computers such as the Z2 board, and it could be even impossible for other resource constraints (e.g. memory).

So, I develop a virtual cross-compiling environment by installing the same Debian Linux version plus the same GCC and all the other necessary dependencies of the Z2 board on my laptop. Then, we can cross compile our code and every package that I needed for the program and at the end when everything works well I transfer the final executable program to the Z2 board and run it there without recompiling.

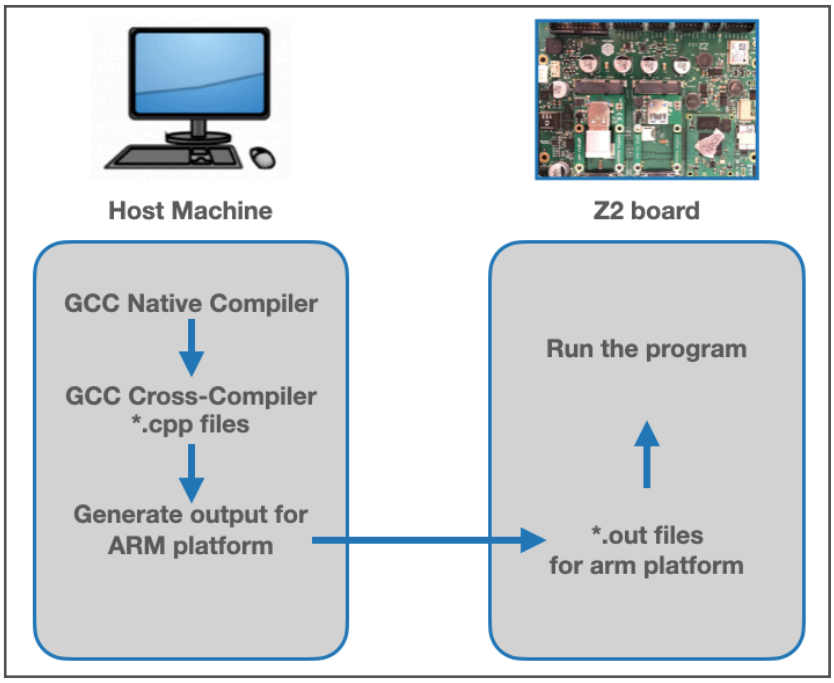


Figure 4: Cross-Compilation process from host machine to target machine.

You can find more information about how to cross compile your program by looking at these three useful websites below, which I also followed during my implementation:

5) http://exploringbeaglebone.com/chapter7
6) https://solarianprogrammer.com/2018/12/18/cross-compile-opencv-raspberry-pi-raspbian/
7) https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/

Chapter 3 - Deep Learning and Computer Vision

The focus of this chapter is on deep learning and the kind of algorithm employed in this project. I'll begin by describing the dataset I used to fine-tune the models, and then we'll look at the two separate object detection approaches used in this project, YOLO and MobileNet SSD. Finally, I'll go over how I put the tracking and estimate algorithms into practice to count the number of vehicles, the average flow speed, road's density, and determine whether a traffic jam is present.

3.1 Dataset

The field of Deep Learning primarily relies on data to train, validate, and test the models. Data is a key component of the field. One of the most significant areas of Artificial Intelligence is computer vision, and the data needed to train its model is either video or photos. When we have supervised learning models, the data must typically also be annotated. The technique of visually marking the location and type of items that the Deep Learning models should train to detect is known as image annotation for object detection. It is challenging to provide such datasets because it takes a lot of time and effort, especially if you are working on a particular case.

3.1.1 Traffic dataset

Traffic surveillance cameras in roads have been widely installed all over the world but traffic images and videos are rarely accessible publicly due to the copyright, privacy, and security issues. From image acquisition point of view, a traffic image or video frame dataset can be divided into three different categories: images taken by the vehicle camera, images taken by the traffic surveillance camera, and images taken by non-monitoring cameras [12]. The first type could have images of highway scenes and ordinary road scenes taken by a car used for automatic vehicle driving and can solve problems such as object detection and tracking. The third one, usually is a vehicle dataset taken by non-monitoring cameras with vehicle appearance covering the brands, models, and production years of the vehicles which usually is suitable for solving problems such as classification challenges. These two categories, the first and the third, are not suitable for traffic

estimation and we need to use a dataset taken by the surveillance cameras which is the category number two.

Due to security concerns, only a small number of traffic scene datasets are publicly available. As a result, I produced a brand new dataset by taking images from traffic videos (35 videos on various scenes and conditions), which is freely accessible online at:

https://www.pexels.com/

https://www.istockphoto.com/it/

Some sample images from dataset are shown below:

Figure 5: Images sample from dataset

As shown in figure-5, all images were collected in a up-down view on a bridge. Accordingly, the board should also be installed in such position where it can take picture in the same view.

I prepared the dataset at Roboflow [26] website, and it can be downloaded for different frameworks. Roboflow is a Computer Vision developer framework for better data collection to preprocessing, and model training techniques. It provides all of the tools needed to convert raw images into a custom computer vision dataset and use it to train the model. Roboflow supports object detection and classification models. It generates various annotation formats for different Deep Learning framework.

The dataset images were captured for different scenes, different times, and different lighting conditions. This dataset divides the vehicles into four categories: cars, buses, trucks and motorcycles. The label file is stored in a text document that contains name and the numeric code of the object category and the normalized coordinate value of the bounding box. This dataset has a total of 557 images with RGB format of 640*640 resolution. This dataset is divided into three parts: a training set with 87%, a validation set with 8%, and a test set with 5%.

This dataset is excellent for fine-tuning the vehicle detection models to detect vehicles in our unique traffic estimation task, but it is not applicable to all vehicle detection tasks used in different settings.

Due to the fact that I employed two separate object detection algorithms for this thesis, I had to create two distinct annotation formats for each model, notably for YOLOv5 and MobileNet SSD. Therefore, we must make sure to transform annotations to work with our models depending on the annotation tool that is utilized. Apparently, 27 distinct labeling formats are supported by Roboflow [26], and you can easily export them based on the target framework, such as to operate with YOLOv5 and MobileNet SSD.

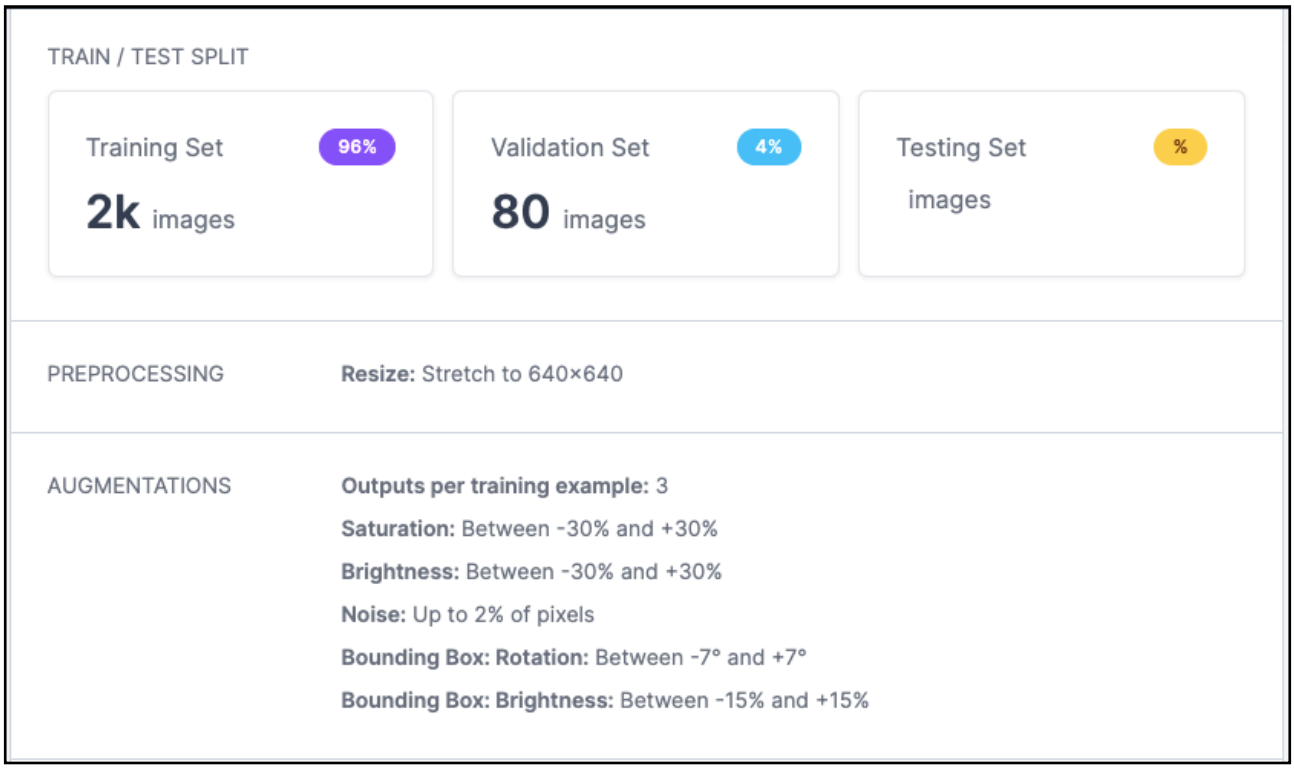


Figure 6: Dataset summary

3.2 Detection

To identify the existence of a traffic queue on a highway, first we need to detect all the vehicles on the road and then calculate other parameters. Object detection has drawn a lot of attention recently and it is one of the most fundamental and difficult issues in computer vision. Today, it is widely used in a variety of practical applications, including autonomous driving, robot vision, video surveillance, etc. The problem definition for object detection is to determine where objects are located in a given image(also called object localization) and which category each object belongs to (also called object

classification) [13] and labeling them with rectangular bounding box to indicate their positions.

3.2.1 A brief history of object detections

It is generally acknowledged that during the past 20 years, object detection has usually progressed through two historical periods: "traditional object detection period (before 2014)" and "Deep Learning-based detection period (after 2014)" [2]. The most popular algorithm in the early era was *Template matching* + *sliding windows*: going through all possible locations and with any scale in the image to see if any of that window contains a the template object or not. Later, some other algorithms such as HOG (Histogram of Oriented Gradient) [15] or DPM (Deformable Part Model) [16] were developed in which they used *Feature-Extraction* + *Classification* techniques to detect object in a given image.

With the increase of computing power, researchers started to pay more attention to Deep Learning algorithms. Deep learning object detection algorithms can be divided in two categories: One-Stage object detectors (like YOLO and SSD) and Two-Stage object detectors (like R-CNN family). One-stage detectors are usually faster, but their accuracy is relatively lower than that of two-stage detectors. First, I will describe two-stage detections models and then one-stage detection models. Figure-7 illustrates two different types of detector architectures.

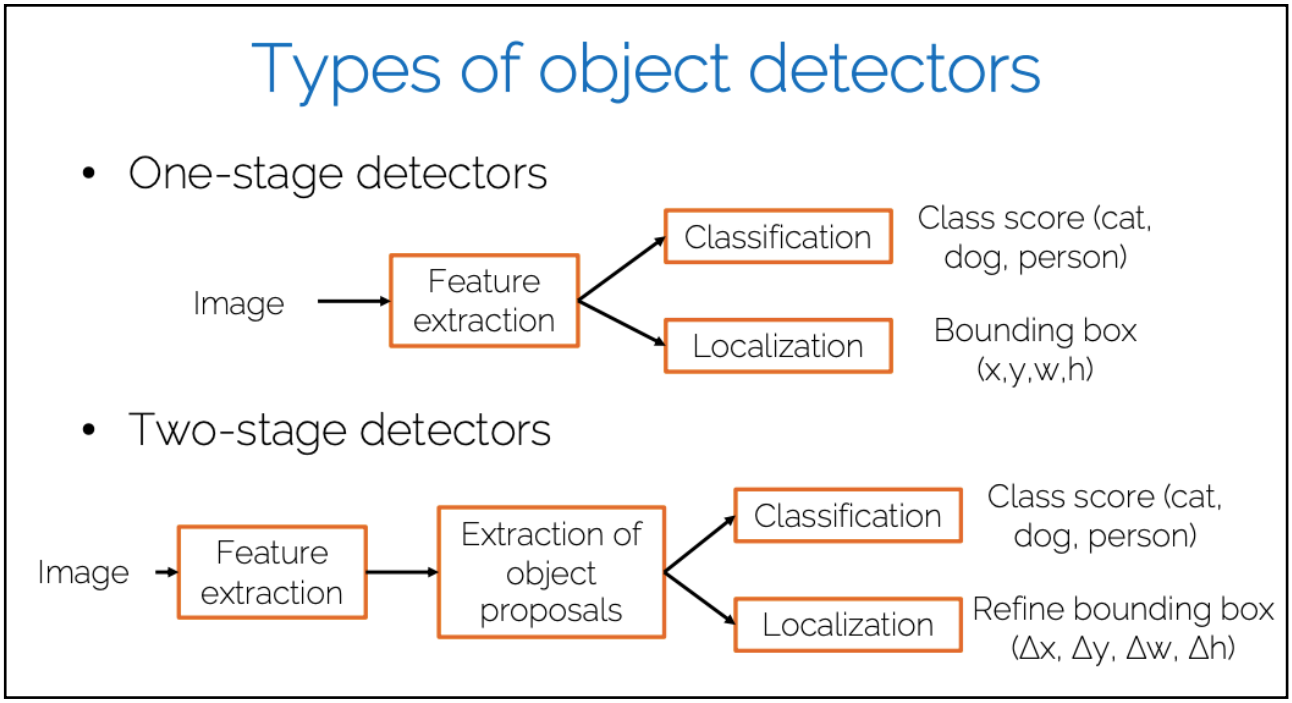


Figure 7: Types of object detectors

3.2.2 Two-stage object detector

There are two main steps in two-stage detector models: first step is to generate the region proposal boxes, which might have our objects in the image, and second step is divided into two layers, classification layer and localization layer. The classification layer is to classify those boxes as one of the ground truth objects, and at the same time localization is performed to adjust those boxes to properly fit the actual object.

The first and original two-stage detection is R-CNN which takes an input image, extracts around 2000 bottom-up region proposals, computes features for each proposal using a large convolutional neural network (CNN), and then classifies each region using a class-specific linear SVMs [14]. After the initial release, other variations, including Fast R-CNN and Faster R-CNN, were made. The main distinction is in the architecture, where the Fast R-CNN enables simultaneous training of the bounding box regressor and classifier in the same network configuration. By combining the majority of the individual building blocks of an object detection system, such as proposal detection, feature extraction, bounding box regression, etc., into a single, end-to-end learning framework, the Region Proposal Network (RPN) in Faster R-CNN enables almost cost-free region proposals. While two-stage detection models are accurate, they have been computationally intensive for embedded system, even with high-performance computers, too slow for real-time applications.

3.2.3 One-stage object detector

One-stage object detection models, apply a fully end-to-end neural network pipeline on an image, and it requires only a single pass through the neural network and predicts all the bounding boxes in one go. The first one-stage object detector was YOLO (You Only Look Once) [18], which divides the image into regions and predicts bounding boxes and probabilities for each region simultaneously. One-stage detectors reframe object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. Considering that, since one-stage detection models characterize detection as a regression issue, then there is no need for a complex pipeline, which makes them extremely fast. Moreover, using a fixed grid of detectors is the main idea which makes one-stage detectors powerful models.

They have a Base network that acts as a feature extractor, such as Inception or ResNet networks, but on edge devices it makes sense to use a small, fast architecture. On

top of the feature extractor, several additional convolutional layers are stacked. These are fine-tuned to learn how to predict bounding boxes and class probabilities for the objects inside these bounding boxes. This is the object detection part of the model or Head. The second famous one-stage detector in Deep Learning era is SSD (Single Shot Detector) [19]. The main contribution of SSD is the introduction of the multi-reference and multi-resolution detection techniques, which significantly improves the detection accuracy of a one-stage detector, especially for some small objects.

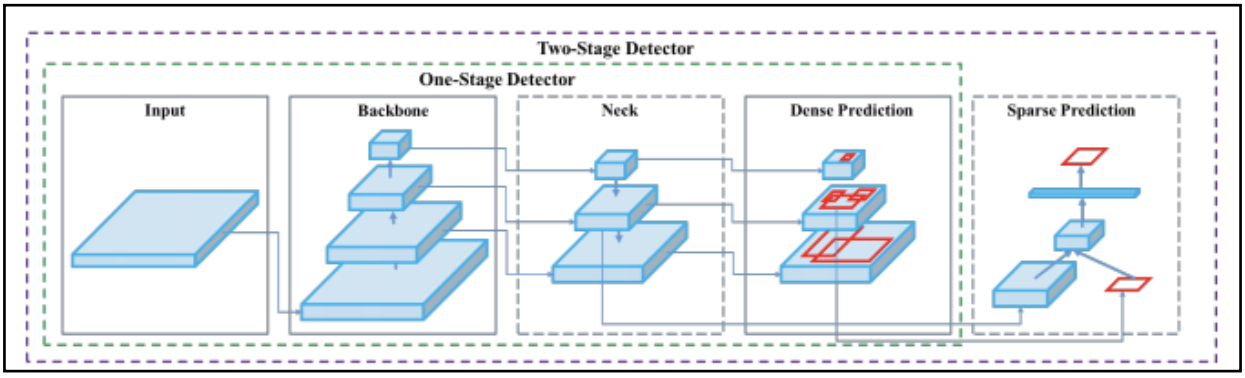


Figure 8: Overview of object detection procedure and architecture [20]

3.2.4 YOLOv5

The original YOLO (You Only Look Once) [18] was written by Joseph Redmon in a custom framework called Darknet. Now, there is a total of seven versions of the YOLO algorithms. The original YOLO model was the first object detection network to combine the problem of drawing bounding boxes and identifying class labels in one end-to-end differentiable network, hence it is the first one-stage object detection algorithm. The latter YOLO algorithms just borrowed or added advanced techniques or tricks to improve the performance.

The detection accuracy of YOLOv5 network [21] is quite high, and the inference speed is very fast, with the fastest detection speed being up to 140 frames per second. Moreover, the size of the weight file for YOLOv5 target detection network model is small, which is nearly 90% smaller than YOLOv4, indicating that YOLOv5 model is suitable for the embedded devices in real-time applications. As a result, the advantages of YOLOv5 network are, high detection accuracy, lightweight characteristics, and fast detection speed at the same time.

Since the accuracy, performance and lightweight aspect of the model are essential for traffic estimation, this study intends to improve the vehicle targets recognition network for the traffic estimation based on the YOLOv5 architecture.

The YOLOv5 model has five different architectures, namely, YOLOv5n [21], YOLOv5s [21], YOLOv5m [21], YOLOv5l [21] and YOLOv5x [21]. The main difference among them is the amount of feature extraction modules and convolution kernel in a specific location of the network is different. The size of the model and the amount of parameters in the four architectures increase in turn.

The YOLOv5n architecture consists of three components, Backbone network, Neck network and Head network.

1) **Backbone** - Backbone network is a convolutional neural network that aggregates different fine-grained images and creates image features. Specifically, the first layer of the Backbone network is the focus module, which is designed to reduce the calculation of the model and accelerate the training speed. On the third layer of the Backbone network, we have BottleneckCSP[23] module, which is designed to better extract the deep features of the image. The ninth layer of the Backbone network, is SPP (spatial pyramid pooling) [24] module, which is designed to improve the receptive field of the network by converting different size of feature map into a fixed-size feature vector.
2) **Neck** - The Neck network is a series of layers to mix and combine image features to pass them forward to the head network (prediction network). The feature extractor of the Neck network adopts a new FPN (feature pyramid networks) [25] structure, which enhances the bottom-up path. It improves the transmission of low-level features, and detection of objects with different scales. Therefore, the same target object with different sizes and scales can be accurately recognized.
3) **Head** - The Head network is mainly used for the final detection part of the model, which applies anchor boxes on the feature map from the previous layer, and outputs a vector with the category probabilities of the target objects, the object scores, and the position of the bounding box surrounding the object.

Figure-9 on the following page displays the whole Yolov5 model architecture pipeline.

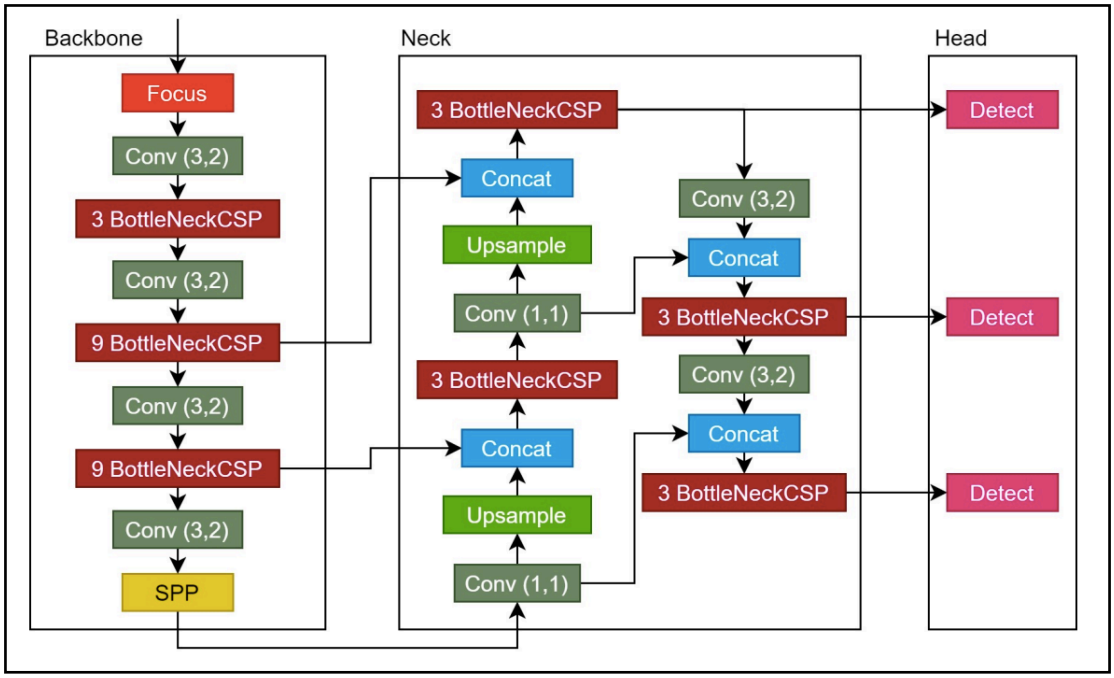


Figure 9: YOLOv5 architecture [22]

3.2.5 MobileNet SSD

The MobileNet [27] model is a lightweight deep neural network architecture designed for mobiles and embedded vision applications, and it is TensorFlow's first mobile computer vision model developed in 2017. The core layers of MobileNet are built on depth-wise separable filters. The depth-wise separable filters significantly reduce the number of parameters, comparing to the network with regular convolutions layers. As a result, we will have a light weight Deep Neural network.

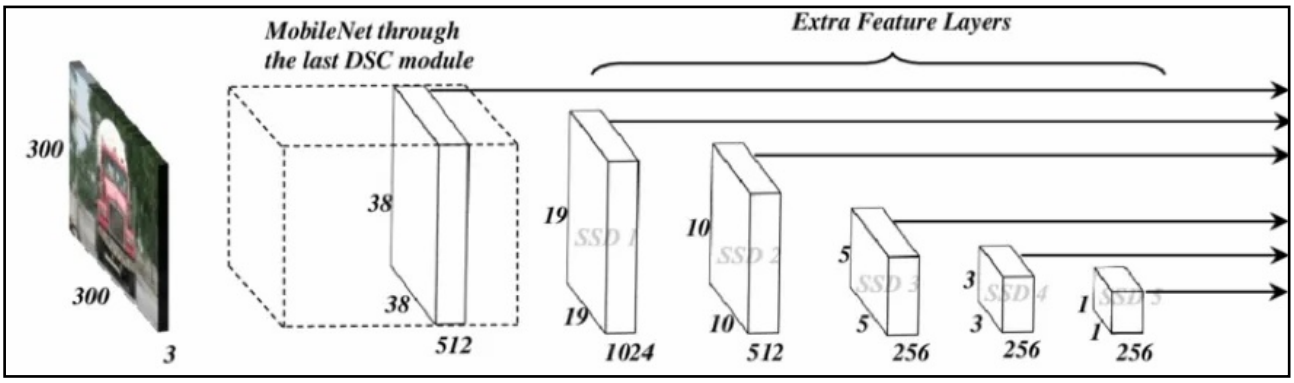


Figure 10: MobileNet Architecture [28]

Around the same time (2016), SSD[28], or single shot detector, was developed by the Google Research team to provide the need for models that could run on embedded devices without a significant trade-off in accuracy. Similar to YOLOv2, instead of using only a single

feature map, it performs classification at different scales. The architectural novelty allows it to predict objects at different scales, meaning feature maps are extracted at different layers for predictions. The SSD network is designed to be independent of the base network, and so it can run on top of any base networks, such as VGG, YOLO, or MobileNet.

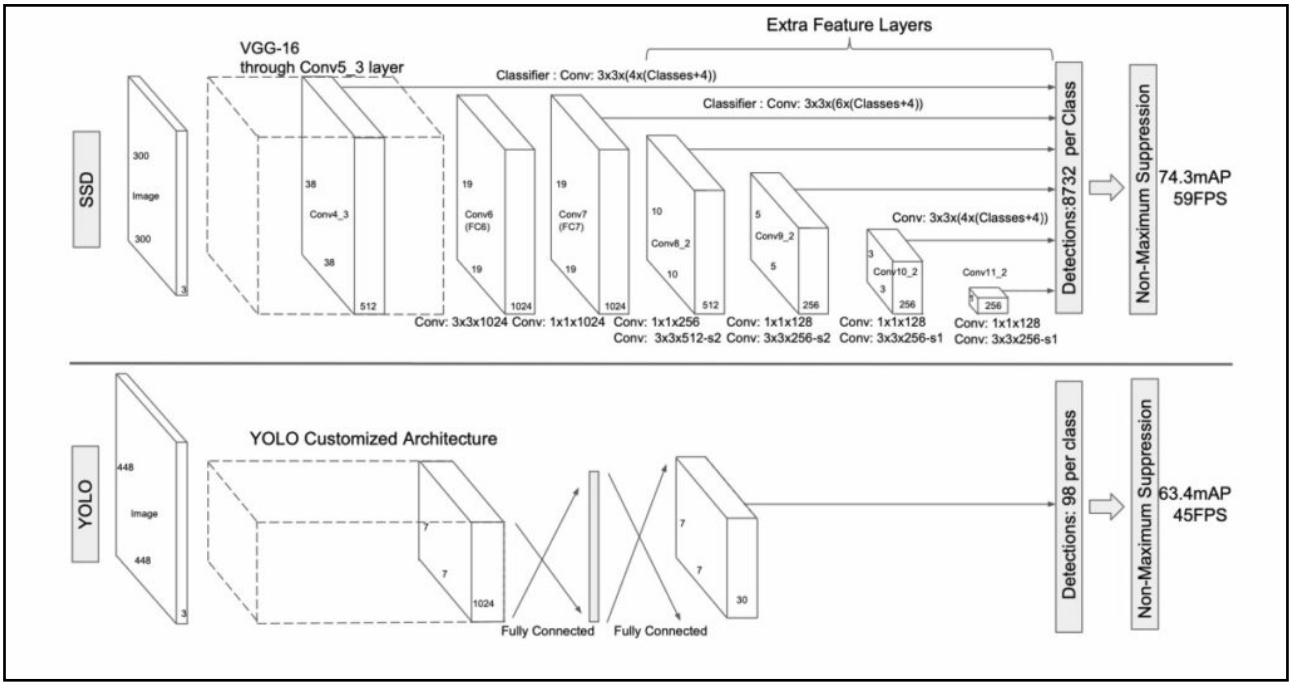


Figure 11: SSD vs YOLO architecture [28]

To further tackle the practical limitations of running high-resource and power-consuming neural networks on low-end devices in real-time applications, MobileNet was integrated into the SSD framework. So, when MobileNet is used as the base network in the SSD, it became **MobileNet SSD**.

MobileNet SSD is ideal for environments with constrained computation. It runs exceptionally well on CPUs, unlike costly hardware accelerators like GPUs. The MobileNetV2 [30] + SSD combination uses a variant called SSD Lite with depth-wise separable layers instead of regular convolutions for the object detection portion of the network, which makes it easier to get real-time results.

When MobileNetV2 is paired with SSD, the MobileNetV2 network is used as an effective feature extractor and it uses two features in its architecture: bottlenecks between layers, and shortcut connections. The bottlenecks encode the model's intermediate inputs and outputs while the inner layer encapsulates the model's ability to transform from lower-level concepts such as pixels to higher level descriptors such as image categories. The residual connections, enable faster training and better accuracy.

The extracted features maps from MobileNet network, are being taken from several different layers (resolutions) and being fed to SSD network for prediction (classification and localization layers).

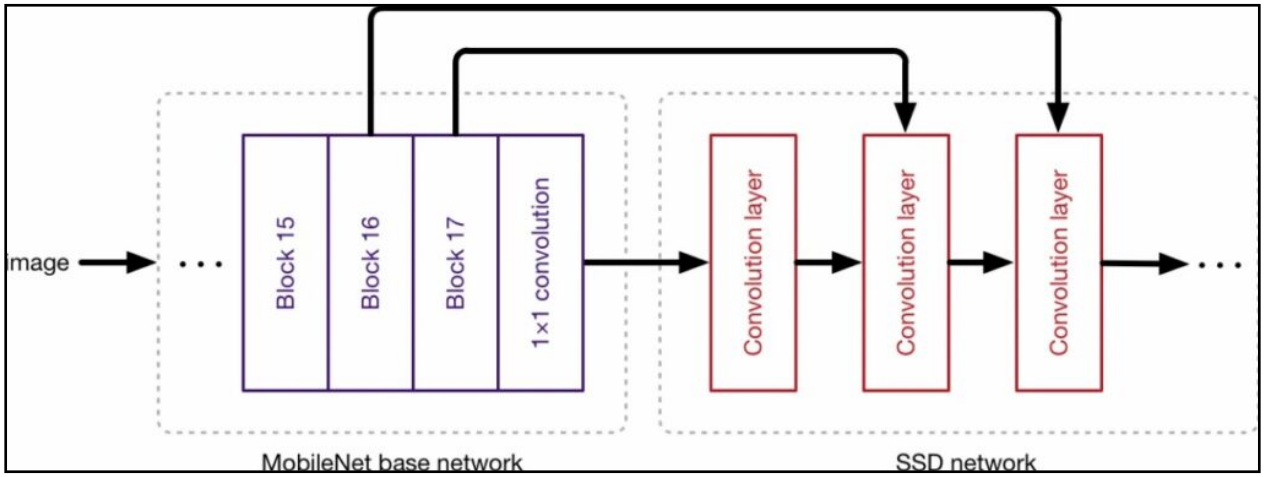


Figure 12: MobileNet SSD architecture [28]

3.2.6 Fine-Tuning

Pre-trained models are frequently utilized as the foundation for Deep Learning tasks in computer vision and natural language processing, because they save both time and money compared to developing neural network models from scratch, and because they perform vastly better on related tasks. The approach in which we use a pre-trained model for another related task, is called Transfer Learning. Transfer learning [31] is a research problem in machine learning (ML) that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them, to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task.

In this thesis I used two different pre-trained object detection models namely YOLOv5 and MobileNet SSD for recognition of the vehicles on the road and those models were trained on COCO [32] dataset which contains 80 different objects like car, motorcycles, human, etc. The distribution of the objects in COCO dataset is similar to the input data of the task of this thesis in which we want to detect vehicles. Therefore, it makes sense to use those pre-trained models for this project.

Here, in this project, the model is required to detect four classes, namely: car, bus, truck, and motorcycle. To do that, first we need to have our brand-new dataset which I describe it on section 3.1.1. Each pre-trained model needs their own specific dataset format to fine-tune them. For instance, YOLOv5 and MobileNet SSD require the dataset to be in the *Darknet* and *TensorFlow TFRecord* format respectfully. We can either prepare the dataset in their specific format manually, or do it automatically. For instance, I used Roboflow website to create dataset for both YOLO and MobileNet SSD. When you prepare a dataset in Roboflow website then you can easily export your dataset in any format depending on your target framework.

For YOLO architecture, I used YOLOv5n (Nano) pre-trained model, and compare to other YOLOv5 versions, Nano model is much faster, but less accurate than others. Since, we want to implement the model on an embedded device, a light and fast model, such as YOLOv5n, would be a good choice for us. You can find more information about how to fine tune YOLOv5 model at: https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data . Some of the parameters that I used for fine-tuning the YOLOv5n model are shown at Table 3.1.

Image Size	*640x640*
batch	*16*
epochs	*100*
Optimizer	*SGD(lr=0.01)*

Table 3.1 Hyper-parameters for fine-tuning YOLOv5n

Table 3.2 and 3.3 you can see that the loss is decreased from the first epoch to the last epoch:

box_loss	obj_loss	cls_loss
0.108	0.06257	0.03186
Images	Instances	P
110	1060	0.804

Table 3.2 First epoch

box_loss	obj_loss	cls_loss
0.01971	0.02584	0.002226
Images	Instances	P
110	1060	0.577

Table 3.3 Last epoch

Figure-13 Shows images from test set where the model tries to detect vehicle on unseen images after being fine-tuned.

Figure 13: Detection result for YOLOv5n model

Based on the pictures illustrated above, it is obvious that YOLOv5 performs well after fine-tuning the model, and it detects almost all the vehicles on the road.

For MoblieNet_SSD, there are two versions that differ in the size of input layer. One is for 320x320 resolution and the other one is for 640x640 resolution (I used 640x640). You can find more information at: https://github.com/tensorflow/models/tree/master/research/object_detection .

Some of the parameters that I used for fine-tuning the MobileNet SSD model are shown at Table 3.2

Image Size	*640x640*
batch	*16*
epochs	*80000*
Optimizer	*SGD(lr=0.004)*

Table 3.4 Hyper-parameters for fine-tuning MobileNet_SSDv2

Table 3.5 and 3.6 you can see that the loss is decreased from the first epoch to the last epoch:

```
loss = 26.261942, step = 1
```

Table 3.5 First epoch

```
loss = 0.24381836, step = 78000 (26.958 sec)
```

Table 3.6 Last epoch

Figure-14 Shows images from test set where the model tries to detect vehicle on unseen images after being fine-tuned.

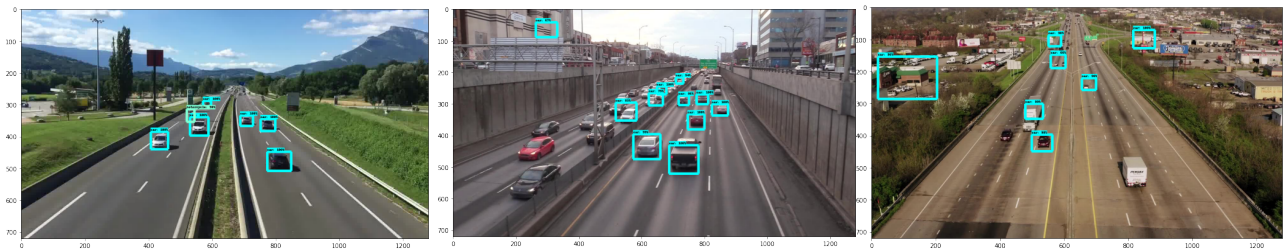


Figure 14: Detection result for MobileNet SSD model

Even though I have repeatedly attempted to enhance the MobileNet SSD model by fine-tuning it with various hyper-parameters, it still does not yield a suitable outcome, and I was unable to obtain a good result after evaluating the model. In Figure-3.8, you can see that the model can detect most vehicles in the road but there are some vehicles that the model couldn't detect. Moreover, there are some areas outside of the road that the model detected them as cars, for instance a part of the building

3.2.7 Inferencing

OpenCV is one of the best computer vision libraries, and it is designed in C++ and uses C++ as its primary interface, however, it also retains a more limited but still robust older C interface. There are bindings in Python, Java and MATLAB/OCTAVE. Although, It is mainly aimed at real-time computer vision, it also has functionalities for running Deep Learning inference (DNN module) as well.

Since OpenCV 3.1, the DNN module has been included in the library to enable forward pass (inferencing) using deep networks that have already been trained with a number of popular deep learning frameworks, such as Caffe, TensorFlow, PyTorch. The fact that the OpenCV DNN module is highly optimized for Intel processors is one of its many wonderful features. When using inference on real-time videos for object detection and image segmentation applications, we can get a good frame rate per second. When we use a model that has been pre-trained using a certain framework, we frequently receive greater FPS with the DNN module.

We can load a YOLO model using *cv::dnn::readNetFromONNX(model_path)* or for MobileNet SSD using *cv::dnn::readNetFromTensorflow(model_path, model_config).* After loading the model, we can then give the input image to the model, begin the forwarding phase through our pre-trained model, and obtain an output cv::Mat object.

The output must then be post-processed to help suppress superfluous boxes and aid in selecting the best possible boxes, such as using Non-Maximum Suppression (NMS), determining confidence levels. Since we employ two input images, this process, from the forwarding phase to post-processing, occurs twice, and the results from each run are stored in a buffer.

After the post-processing step, the detection module's job is complete, and the output is prepared to be sent to the following layer in the program pipeline for tracking and estimating.

3.3 Tracking

This project's objective is to identify any queues on the highway, but more significantly, we want to gather some quantitative data, such as the number of vehicles or average flow speed, and send it to a central server that can use it to anticipate traffic flow more precisely. We have to first identify every vehicle, then track them to count them, determine their average speed, and gather more data in order to extract some quantitative information. The plan is to take two consecutive images of the highway at a rate of one frame per second, then use one of our object detection models to find vehicles in each image. Afterward, in order to count and estimate the speed of each vehicle from those two images, we must be able to identify each one and assign it a label. Otherwise, we won't be able to tell whether the vehicle in the first image is the same as the vehicle in the second image, and as a consequence, all vehicles will be counted as new objects, which is incorrect. Additionally, we must identify each vehicle from frame to frame in order to determine their distance traveled between two frames in order to estimate their average speed.

3.3.1 Tracking algorithm

Object tracking is the process of identifying each vehicle in a series of frames. Accordingly, when we are dealing with multiple objects in a series of frames, the process is known as multiple object tracking (MOT), which is described as the procedure of recognizing multiple objects in a single frame and retrieving their identity information over consecutive frames [3]. When developing MOT approaches, two major issues should be considered. One is how to measure similarity between objects in frames, the other one is how to recover the identity information based on the similarity measurement between objects across frames.

While there are some Deep Learning object tracking algorithms, such as Multi-Domain Net (MDNet) and Generic Object Tracking Using Regression Networks (GOTURN), I choose not to use them in this thesis due to hardware resource constraints. Instead, the tracking task is accomplished by making some assumptions. The following assumptions are made:

1) In the course of two frames, no vehicle could travel from one lane to another.

2) The area of the bounding box shouldn't be increased in the following frames according on the camera view (top-down and vehicles' back view).

Using the following three pieces of information about boxes, we can now track each object from consecutive images, relying upon those assumptions:

- **Which lane number they belong to.** Will specify their lane number in both images..
- **How close their centers are on the consecutive frames.** Given a two frames, we can assume that the vehicle cannot suddenly move from one corner of the image to another. Therefore, the center of the vehicle that's been identified in the second frame should be somewhat close to the center of the identical vehicle in the first frame.
- **The area of the boxes.** The following frame should see a little reduction in the bounding box's area (according to the camera view).

With that knowledge, we can put them together to create a confidence to check if two boxes represent the same vehicle. In fact, a lot of tracking algorithms employ a model for predicting interior movement. It uses a movement model to forecast the upcoming position based on how the car has previously moved. Vehicles normally move in a predictable direction rather than at random. So, this technique really helps match the detections to the right track and that could be a future work for this thesis.

All the detected vehicles from detection algorithm are stored in a C++ Vector object and each element of the vector is a Centroid object which holds each vehicle attributes. Those attributes are essential for tracking and other calculations. Figure 15 is all the attributes that are being saved for each detected vehicle:

```
struct Centroid
{
    int id;
    float conf;
    float area;
    cv::Point center;
    cv::Rect box;
    std::vector<cv::Rect> box_history;
    std::vector<cv::Point> position_history;
    cv::Point next_position;

    std::string name;
    bool under_tracking;
    int lane_num;
    float distance{0.0};
    float speed{0.0};
};
```

Figure 15: Centroid object class

Having those attributes for each detected vehicle will help us to compare each vehicle between two images and eventually identify them. We will go through a loop to compare each vehicle from the first frame with all others from the second frame.

3.3.2 Lane detection

Assuming that no car can cross its lane in less than one second, I must first determine each vehicle's lane number before I can begin tracking. As a result, by just comparing vehicles on the same lane number, we can reduce the scope of the tracking problem. Therefore, it would be much simpler for us to state that the car that was identified on lane number one at frame t1 will most likely be present in lane number one again at frame t2.

We can select the location of interest for the camera installation (it should be on top of a bridge above a highway) and capture a test photo from the road. The back side of the vehicle should be seen in a top-down view from the camera. The next step is to manually determine each lane on the image space. It implies that we have a vector of pixels for each lane that corresponds to the actual lane on the road. We need to complete this assignment in advance. An illustration of a camera view with lane delineation is shown in Figure 16. Once lane pixels have been established, all that is left to do is determine the location of each vehicle center point and assign the associated lane number. A line iterator object class in OpenCV enables you to iterate through each pixel of a specified line.

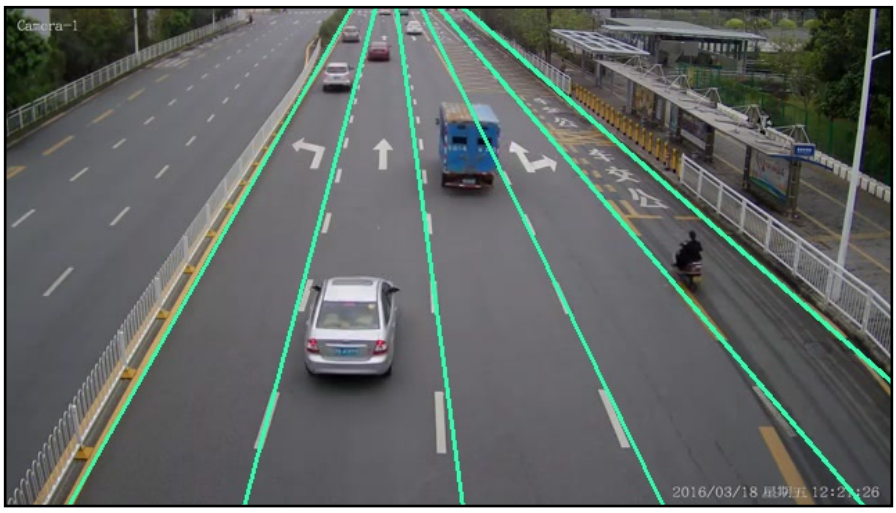


Figure 16: Camera view with lane separation

Therefore, you can iterate over a line and determine whether or not the point of interest is situated between two lines. I utilized the "cv::LineIterator" object class to identify the vehicle lane numbers.

3.3.3 Distance between points

After defining the lane number for each vehicle, we must determine how close each vehicle in the frame is to the identically numbered vehicles in the second frame. To accomplish this, we can calculate the distance between the first vehicle in frame-1 to every other vehicle in frame-2 within the same lane, and then pick the shortest distance among them. We carry out this procedure for each vehicle that is already in that lane number and each time we compare the least distance and keep the shortest one. The same procedure is repeated for the remaining lanes.

To determine the distances between two locations in a two-dimensional image, we can apply the Euclidean distance formula. It can be derived through using A (x1, y1) as the vehicle's center point in frame-1, and B (x2, y2) as vehicle's center point in frame-2. Then, the distance "d" between the two centers, A and B, is:

$$d = \sqrt{[(x2 - x1)2 + (y2 - y1)2]}$$

3.3.4 Area of a bounding box

The process for calculating area is the same as the one we used to calculate distance, with the exception that in this case we compare each vehicle's area rather of computing their distance. The vehicle's area in the first frame of a particular lane number will therefore be compared to the vehicle's area in the second frame of the same lane number, and if the area is greater than the one in the second frame, the vehicle may be a candidate to be chosen as the same vehicle. According to the assumption we made regarding the camera view, that would be valid for most of the time.

3.4 Estimation

I covered the detection, tracking, and collection of some relevant information about each vehicle in sections 3.2 and 3.3 of this chapter, but now we need to apply certain approaches to determine traffic flow on the highway based on that information. Estimating the number of vehicles on the road, figuring out the average flow speed, and calculating the road density are the three most significant elements we can extract from that information. Then, using these three features, we can apply a straightforward classifier to forecast the volume of traffic on a particular road.

3.4.1 Vehicle counting

One of the important inputs of the estimator model is the number of vehicles that are currently on the road. This number can be easily determined by counting the number of vehicles that we were able to successfully track during the tracking process. In other words, once the tracking algorithm has completed its work, we will get a C++ vector in which every single element represents a tracked vehicle. Therefore, all we need to know is the magnitude of our vector, which will tell us how many vehicles we managed to capture in those two frames.

3.4.2 Speed estimation

It is difficult and requires several factors to estimate speed based on a video stream. In this project, I simplified the requirements and estimated a vehicle's average speed, which would be adequate for our particular traffic estimating model.

Speed estimation can be accomplished, by mapping the relation between pixel distance and actual distance, the speed is estimated through linking the corresponding vehicles distance between the current frame and the previous frame divided by time. Since there is a one-second lag between these two subsequent images, the time interval here is one second, consequently, a vehicle's average speed is only the distance it traversed.

However, the actual distance and the camera settings have a direct impact on the accuracy of the speed estimation method. I assume the camera recording traffic should be static, which holds for the 2018 AI City Challenge too. Since I do not know the camera settings and have to obtain establishing actual distance on the road, I calculate the real distance on the road by considering the white lane length based on United State Federal Highway Administration guideline [5] and transfer it into image plane. This approach will assume that the broken lines consist of 3m (10 ft) line segments and 9m (30 ft) for the gaps.

Moreover, the fact that the cameras are pointed at the horizon should also be taken into account. As such, vehicles traversing a frame from bottom to up will appear to be moving slower towards the top of the screen, as they reach the horizon, even though they may be driving at a constant speed in reality. Therefore, there should be a scale factor to naturalize the speed based on the vehicle position on the image, and should work in such a way, when a vehicle is located at the bottom of an image, the scale factor should be smaller than when the vehicle is located at the horizon points. I provided a scaling factor for speed

estimation that is based on the position of the vehicle at the y-axis divided by the height of the image. If the vehicle is near the top or at the horizon, its y coordinate will be small, and the scale factor will produce a high result. In the opposite case, when the vehicle is at the bottom of the image, the value of 'y' will be high, which will result in a smaller scale factor.

3.4.3 Density estimation

Road density is important for estimating traffic levels. The congestion of cars on the road is a significant problem. If the density of traffic reaches its maximum, the flow of traffic decreases to zero, resulting in a traffic jam. I calculate density by summing up all the area of tracked vehicles that occupy a particular segment of the road, and dividing this sum by the total area of the road. Then I multiply this value by 100 to get the density percentage.

The sum of all tracked vehicle areas can be calculated by summing up all the detected vehicle areas which are within the road coverage. The total length of the road on the image can be calculated based on the total area of the lanes. If you take all the outer lanes of the leftmost line, rightmost line, top line, and bottom line, the type of shape you get is an irregular polygon. There appears to be an irregular polygon in some of the pictures in our project, with its number of sides varying. In most cases, the geometric polygon shape should have at least 4, 5, or 6 sides to cover the road for this particular project.

To find the area of an irregular polygon, we can apply Green's theorem, which can be used to determine the area and centroid of plane figures simply through integrating over the perimeter. There is a method for finding the area directly by OpenCV library, based on Green's theorem. All we need to do is to input the vertices (corners) of the shape as a *cv::contour* object to the *cv::contuourArea()* function, and as an output it will return the calculated area of those points. The only constraint is that we need to pass the coordinates either in clockwise or counter-clockwise, starting at any vertex.Figure 17 displays a grid representing the total length of covered roads.

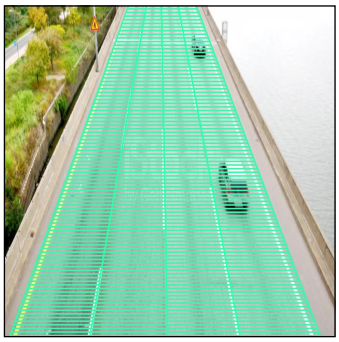


Figure 17: Road total length coverage

3.4.4 Traffic estimation

After extracting all the necessary information from the input image, we can use a classifier to estimate the traffic level. This classifier will use the three information it has as input features and predict how much traffic is present. Traffic flow can be classified into Low, Medium, and High levels. There are a variety of decision-making algorithms that can be used to classify data, including machine learning methods and simple classifiers.

A Support Vector Machine (SVM) was used as the project's final classifier. I used a brand new dataset to train the SVM classifier, which contained 100 samples of data with relevant class labels. Each sample dataset has three values: the number of vehicles, the density of the road, and the average flow speed. These three properties were the quantitative data that we had obtained from our detection and tracking technique, thus we had to train our classifier using input data that contained them.

Since there wasn't a publicly available dataset for this particular situation, I made my own by manually extracting those properties from the real-world traffic film I downloaded from the internet using a series of subsequent images. The properties of each pair of consecutive images had to be manually examined and noted, such as the number of cars that showed in both images or the amount of flow density a group of vehicles would occupy in a given frame. The dataset extraction process was sped up during dataset generation by running the program without a classifier, however each time I compared the results with actual images, and if the derived features were reasonable, I kept those features (number of vehicles, flow density, average flow speed).

After we have trained our SVM classifier with the relevant dataset, then we can load it to the main program, and the extracted features from the detection and tracking units can be fed into this classifier as an input. After it has processed those features, the SVM classifier can at last predict the related traffic class label.

3.5 Performance

Speed, accuracy, and power consumption were the three factors I used to assess performance. The time it takes the program to complete a cycle, which includes taking two sequential photos, feeding them as input into the models, and saving the result to the disk, can be used to measure speed. The same cycle must be run in order to assess the power consumption and accuracy, but with a few factors in mind. To measure the energy consumed during a cycle, we should use a standard multimeter, or some other appropriate instrument in the case of power usage. In the case of accuracy, we need to take into account more distinct components, which will be detailed in the future sections, because the program pipeline consists of three essential components: detecting, tracking, and estimating.

3.5.1 Speed

I utilized a C++ standard library called *std::chrono* to determine the speed by estimating the total running time as well as the amount of time spent on each individual component of the application. Calculated time for detection includes loading the model, loading two photos, and using one of the detection models to process those two images. In most cases, the overall detection time for the YOLO model was 52 seconds, whereas the total detection time for MobileNet is 72 seconds. Typically, tracking and estimating times are quite quick less than 0.01 seconds. As a consequence, the software takes an overall total of 53 seconds for the YOLO model and 73 seconds for the MobileNet SSD model to run one complete cycle from reading the input images to writing the output.

Pipeline	YOLO	MobileNet_SSD
Model loading time	1s 31ms	7s 541ms
Loading time for two images	0s 430ms	0s 330ms
Detection time for two images	50s 938ms	61s 983ms
Tracking time	0s 37ms	0s 34ms
Estimation time	0s 11ms	0s 16ms
Saving the result	0s 667ms	0s 662ms
Total time	**53s 114ms**	**70s 566ms**

Table-3.7 Speed comparison between YOLO and MobileNet SSD

The computation times for two images for YOLO and MobileNet SSD are compared in Table-3.7. We can observe that there are 20 seconds between these two models. You see that the loading time for the Yolo model is around 0.5 seconds, whereas the loading time for the model for the MobileNet SSD is approximately 7.5 seconds.

3.5.2 Power Consumption

Measuring power consumption requires hardware-specific devices, hence the Z2 board power usage cannot be accurately measured by just some software. The relevant hardware must be present, and the easiest option is to use an oscilloscope or multimeter with data logging capabilities. This can be done by measuring the current drawn from the entire system along with the voltage of the power supply.

Intuitively, if we want to measure the power consumption of only a specific application, we should first measure the baseline power consumption of the Z2 board when it's idle. Then we can run the program and measure the energy it consumed. Once we know these numbers, we can subtract them to find the amount of energy used by the particular application we are considering.

It appears that when the camera takes two shots, we can measure the power usage of the camera independently of the board as a whole. Since the camera is only required at the start of the application, we can control its usage to reduce our power consumption by being aware of how much power it will use and when to turn the module on and off.

3.5.3 Accuracy

Since there were three different compute modules and each of them had a different impact on the quality of the output, measuring the correctness of the entire pipeline was quite challenging. The tracking algorithm will function with high accuracy if we correctly identify all the vehicles in the two images. If we train the SVM model with more input data, based on actual traffic flows, the estimation classifier would be even more accurate. Since there was no suitable dataset to examine the algorithms against, I tried to do so using some actual experiments and evaluated the algorithms' correctness based on observing the real evidence. I evaluated them using 135 test samples from 19 different highways, using two consecutive images for each test sample. A rough assessment of the program quality was shown in Table-3.8 utilizing two alternative detection methods with 135 test samples.

Pipeline	YOLO	MobileNet_SSD
Detection	0,89	0,65
Tracking	0,92	0,73
Estimation	0,93	0,47
Total	**0,91**	**0,62**

Table-3.8 Accuracy comparison between YOLO and MobileNet SSD

I should point out that the entire pipeline is reliant on detection models, and as you can see from the table, the YOLO model performed significantly better than MobileNet SSD.

Over the course of those 135 test samples, the evaluation process was entirely automatic. Based on the discrepancy between the actual number of vehicles on the ground and the number of vehicles predicted through models, the detection and tracking algorithms were assessed. Since estimator is a classification model, I employed a confusion matrix to assess the model's accuracy. Table-3.9 and Table-3.10 display the estimator's results as a confusion matrix, and the evaluation's outcomes can be found on the following page.

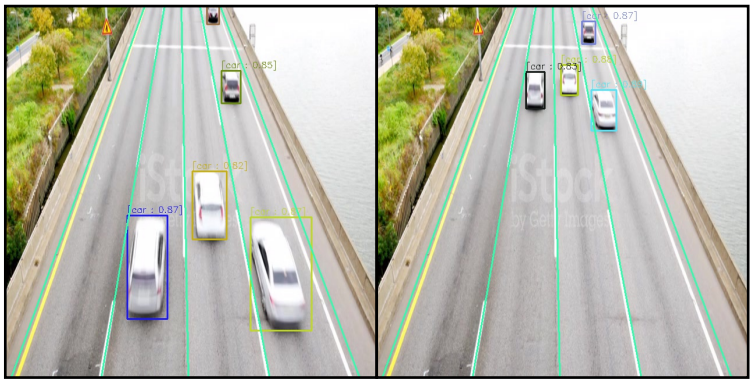
	Prediction		
[135]	Low	Medium	High
Low	45	4	0
Medium	2	38	1
High	0	2	43
Recall	**0,92**	**0,93**	**0,96**
Precision	**0,96**	**0,86**	**0,98**
Total Acc	**0,93**		

Table-3.9 Confusion matrix for estimator based on YOLO model

	Prediction		
[135]	Low	Medium	High
Low	44	5	0
Medium	21	19	1
High	12	33	0
Recall	**0,90**	**0,46**	**0,00**
Precision	**0,57**	**0,33**	**0,00**
Total Acc	**0,47**		

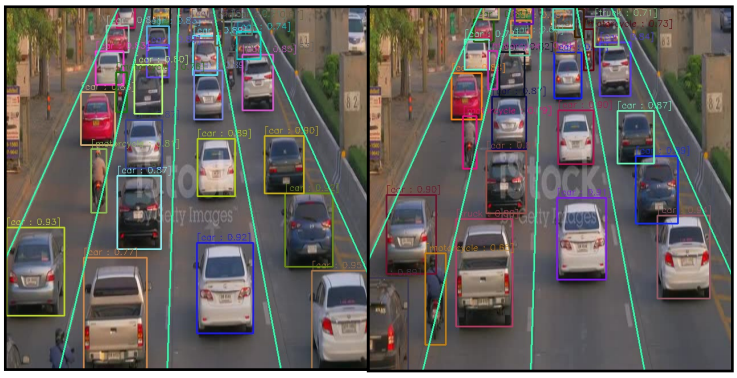
Table-3.10 Confusion matrix for estimator based on MobileNet SSD model

Frame 1 Frame 2 Result:



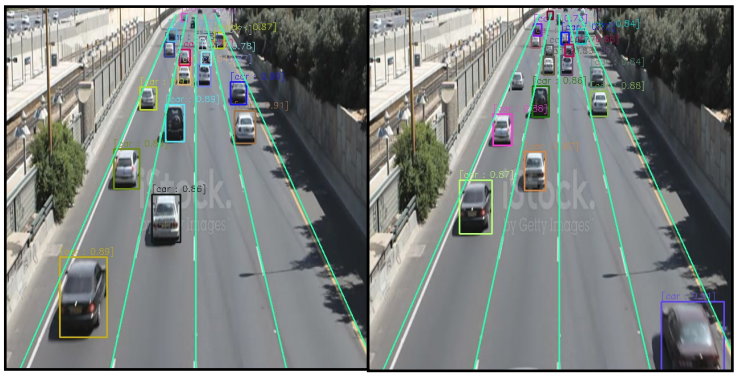
Output for YOLO:
• Traffic Status: Low
• Number of cars: 5
• Flow density: 18%
• Flow average speed: 114
• Highest flow is in lane number: 4

Frame 1 Frame 2 Result:



Output for YOLO:
• Traffic Status: High
• Number of cars: 29
• Flow density: 53%
• Flow average speed: 21
• Highest flow is in lane number: 3

Frame 1 Frame 2 Result:



Output for YOLO:
• Traffic Status: Medium
• Number of cars: 17
• Flow density: 19%
• Flow average speed: 18
• Highest flow is in lane number: 1

Frame 1 Frame 2 Result:

Output for YOLO:
• Traffic Status: High
• Number of cars: 30
• Flow density: 70%
• Flow average speed: 4
• Highest flow is in lane number: 2

Chapter 4 - Experiments

I had to conduct a real experiment in order to obtain a final test for the program. a test that would provide results similar to those of a real-world installation of the device on a highway. So, I described how and where I actually experimented with the project in this chapter.

4.1 Overview

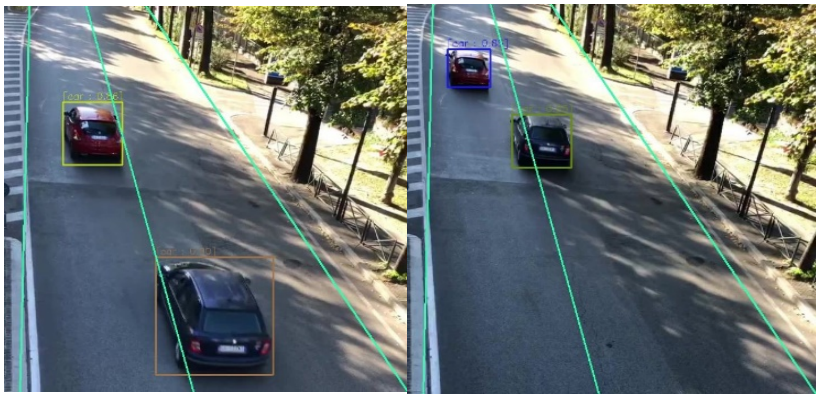
There are two various experiments taking on in various places. I tested the first experiment by taking pictures using my cellphone and then transferring them to the Z2 board. For the second experiment, however, the project itself was responsible for every task, from image capture through final evaluation. i.e., we tested the full procedure using even its own camera and power source.

All of the photos were taken above a bridge over a highway, in which a camera was set to record the back side, and top of moving vehicles. The first experiment was conducted on a particular street in a densely populated region of Rimini. The second experiment was conducted on a highway in Siena with varying traffic levels, low, medium, and high.

4.1.1 Rimini experiment

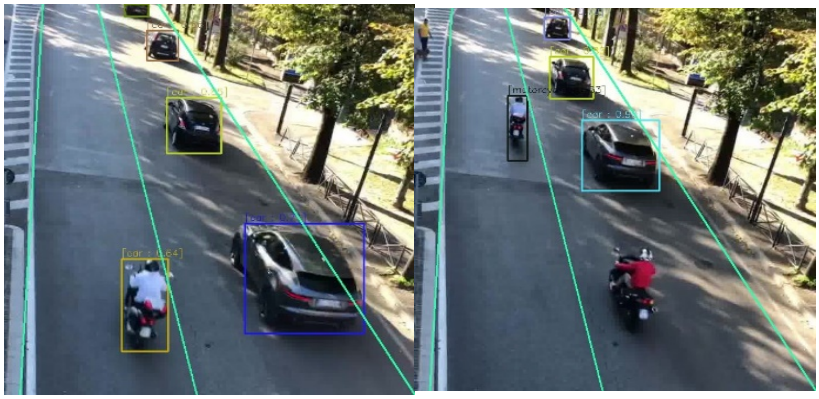
I didn't bring the Z2 board because it was my first effort; instead, I used my cellphone to take shots of the scene. Then, for five minutes, I shot a video with low traffic. I next started to extract several sequential images from it to give it some sample input to enter. Based on those images, I first produced the line.txt file which is required for the tracking and estimation procedure indicated the object's lanes on the image. When testing the model, I use two input samples that each contain two consecutive images (with one second delay between each image). You can observe two distinct examples for each detection model on the following page.

Frame 1 Frame 2



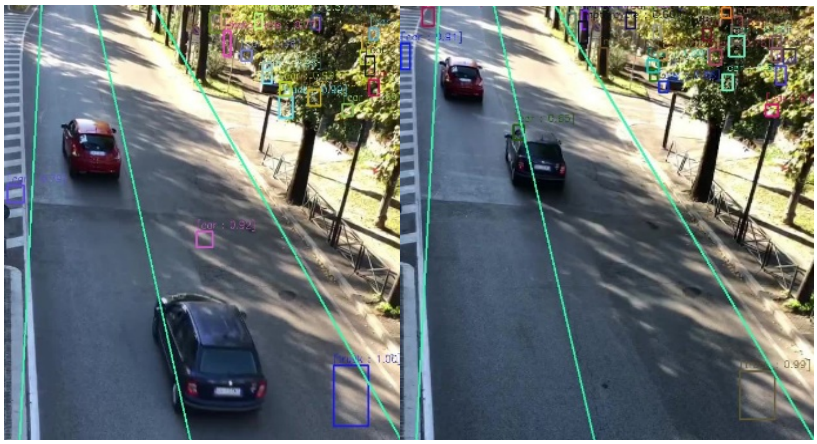
Output for YOLO:
• Traffic Status: Low
• Number of cars: 2
• Flow density: 18%
• Flow average speed: 107
• Highest flow is in lane number: 2

Frame 1 Frame 2



Output for YOLO:
• Traffic Status: Low
• Number of cars: 5
• Flow density: 21%
• Flow average speed: 84
• Highest flow is in lane number: 2

Frame 1 Frame 2



Output for MobileNet SSD:
• Traffic Status: Medium
• Number of cars: 3
• Flow density: 2%
• Flow average speed: 3
• Highest flow is in lane number: 2

Frame 1 Frame 2

Output for MobileNet SSD:
• Traffic Status: Medium
• Number of cars: 8
• Flow density: 9%
• Flow average speed: 86
• Highest flow is in lane number: 0

As you can see from the images above, the YOLO model performed admirably. It accurately estimated that there was little traffic flow because it detected all the objects. On the other hand, MobileNet SSD underperformed, mistakenly classifying shadows and other objects as vehicles but estimating light traffic correctly.

4.1.2 Siena experiment

The final experiment,

Chapter 5 - Conclusion

In this work, we aim to predict the traffic flow conditions in order to identify queues and potential bottlenecks. We intend to accomplish that by employing a full harvesting device, and we need to use sub-sampling of the traffic flow in order to achieve this goal because we cannot use an acquisition device that runs continuously (2 images 1 sec sampling rate). We agree that there are roughly three levels of traffic conditions. Even though a more precise classification could be the next goal for future work, this is sufficient.

The most crucial component of the entire program is detection, which comes first in the pipeline and directly affects the outcome. The accuracy of the detection models thus has a significant impact on the outcome. Based on this test, the YOLOv5 model scored admirably, scoring highly for both speed and accuracy. However, the MobileNet SSDv2 model did poorly, attaining low levels of accuracy and speed. Different object sizes, lighting circumstances, image perspective, partial occlusion, and complicated backgrounds all seem to have contributed to the MobileNet SSD's weak performance. Another argument can be that the model has to be improved with additional training data.

This research will be expanded to incorporate more detection models in the future, particularly the YOLO (You Only Look Once) family of models and the FOMO (Fear of Missing Out) model. Gaining a better outcome will also be possible with improved tracking and estimating algorithms. In addition to developing better algorithms, we also need to construct a larger dataset so that detection and estimation models may be improved.

Bibliography

[1] Cheng-Jian Lin, Shiou-Yun Jeng, Hong-Wei Lioa, "A Real-Time Vehicle Counting, Speed Estimation, and Classification System Based on Virtual Detection Zone and YOLO", Mathematical Problems in Engineering, vol. 2021, Article ID 1577614, 10 pages, 2021. https://doi.org/10.1155/2021/1577614
[2] Zou, Z., Shi, Z., Guo, Y. and Ye, J., 2019. Object detection in 20 years: A survey. arXiv preprint arXiv:1905.05055 https://doi.org/10.48550/arXiv.1905.05055
[3] Luo, W., Xing, J., Milan, A., Zhang, X., Liu, W. and Kim, T.K., 2021. Multiple object tracking: A literature review. Artificial Intelligence, 293, p.103448. https://doi.org/10.1016/j.artint.2020.103448
[4] Sochor, J., Juránek, R., Špaňhel, J., Maršík, L., Široký, A., Herout, A. and Zemčík, P., 2018. Comprehensive data set for automatic single camera visual speed measurement. IEEETransactions on Intelligent Transportation Systems, 20(5), pp.1633-1643. https://ieeexplore.ieee.org/abstract/document/8356199
[5] [Online]. Federal Highway Administration https://mutcd.fhwa.dot.gov/htm/2003r1/part3/part3a.htm
[6] [Online]. ARM Developer https://developer.arm.com/documentation/den0013/d/Introduction/Embedded-systems
[7] [Online]. Linux https://en.wikipedia.org/wiki/Debian
[8] [Online].Pros, Cons, and Comparisons of Popular Languages https://www.qt.io/embedded-development-talk/embedded-software-programming-languages-pros-cons-and-comparisons-of-popular-languages#:~:text=For%20many%20embedded%20systems%2C%20C,language%20is%20fast%20and%20stable.
[9] [Online]. Build https://www.techopedia.com/definition/3759/build
[10] [Online]. CMake https://cmake.org/
[11] [Online]. Cross-Compiler https://wiki.osdev.org/GCC_Cross-Compiler
[12] Song, H., Liang, H., Li, H. et al. Vision-based vehicle detection and counting system using deep learning in highway scenes. Eur. Transp. Res. Rev. 11, 51 (2019). https://doi.org/10.1186/s12544-019-0390-4
[13] Zhao, Z.Q., Zheng, P., Xu, S.T. and Wu, X., 2019.Object detection with deep learning: A review. IEEE transactions on neural networks and learning systems, 30(11), pp.3212-3232.
[14] Girshick, R., Donahue, J., Darrell, T. and Malik, J., 2014.Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedingsof the IEEE conference on computer vision and pattern recognition (pp. 580-587).
[15] Dalal, N. and Triggs, B., 2005, June.Histograms of oriented gradients for human detection. In 2005IEEE computer society conference on computer vision and pattern recognition(CVPR'05) (Vol. 1, pp. 886-893). Ieee.
[16] Felzenszwalb, P.F., Girshick, R.B. and McAllester, D., 2010, June.Cascade object detection with deformable part models. In 2010 IEEE Computer society conference on computer vision and pattern recognition (pp. 2241-2248). Ieee.
[17] (Online). ObjectDetection https://dvl.in.tum.de/slides/cv3dst-ss20/1.ObjectDetection-Introduction.pdf
[18] Redmon, J., Divvala, S., Girshick, R. and 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).
[19] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y. and Berg, A.C., 2016, October. Ssd:Single shot multibox detector. In European conference on computer vision (pp. 21-37). Springer, Cham.
[20] (Online). https://blog.roboflow.com/yolov5-improvements-and-evaluation/
[21] (Online). Github repository. https://github.com/ultralytics/yolov5/

[22] Benjumea, A., Teeti, I., Cuzzolin, F. and Bradley, A., 2021. YOLO-Z: Improving small object detection in YOLOv5 for autonomous vehicles. arXiv preprint arXiv:2112.11798.
[23] Wang, C.Y., Liao, H.Y.M., Wu, Y.H., Chen, P.Y., Hsieh, J.W. and Yeh, I.H., 2020. CSPNet: Anew backbone that can enhance learning capability of CNN. In Proceedingsof the IEEE/CVF conference on computer vision and pattern recognitionworkshops (pp. 390-391).
[24] He, K., Zhang, X., Ren, S. and Sun, J., 2015.Spatial pyramid pooling in deep convolutional networks for visual recognition. IEEEtransactions on pattern analysis and machine intelligence, 37(9), pp.1904-1916.
[25] Lin, T.Y., Dollár, P., Girshick, R., He, K., Hariharan, B. and Belongie, S., 2017.Feature pyramid networks for object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2117-2125).
[26] (Online). https://roboflow.com/
[27] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. Mobilenets:Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.
[28] (Online). https://xailient.com/
[29] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y. and Berg, A.C., 2016, October. Ssd: Single shot multibox detector. In European conference on computer vision (pp. 21-37). Springer, Cham.
[30] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.C., 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510-4520).
[31] (Online). Wikipedia https://en.wikipedia.org/wiki/Transfer_learning
[32] (Online). Coco. https://cocodataset.org/
[33] (Online). omnicalculator.com/everyday-life/traffic-density