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 highroad 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 raises 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 a video frame and localizes their positions while they stay in the scene that is called object detection. Object detection is one of the most important computer visions challenge that deals with detecting and localizing visual objects of a certain class (such as humans, animals, or cars) in digital images [2].

After the task of detection, 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 part, to classify the level of a traffic queue on the highroad. The levels are: Low traffic, Medium traffic, High traffic.

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 above a highroad and the camera should be calibrate in a top-down and rear-view position. The board will be turned on every 5 minutes. First it tries to take two consecutive images with 1 second delay between them. Then these two images will be 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 will be process for the task of tracking but 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 based on the camera view.

Having these two assumptions, I investigate a simpler tracking strategy by checking that if two center points of the 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 the algorithm will identify it as the same vehicle. Only after identifying all the vehicles, we can 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 their distance passed between two frames. A proper speed estimation algorithm requires a precise camera calibration and specifying a real distance on the road in case when we are not using LIDAR. Basically, dealing with this task requires perspective projection and different rotations of the camera; 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]. I tried to relax the task of speed estimation by defining the real distance of a 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]. Having said that, we can have an approximate real distance on image plane assuming without camera calibration. Then the task of speed estimation would be simply the distance of a vehicle between two frames divided by the number of frames per second which in this project fps is equal to one because we are using two consecutive frames with delay 1 second. Having that information, we can eventually estimate some useful information such as number of vehicles, road density, and average flow speed on the road. With that three information we can use a classifier to classify the traffic into three levels: Low, Medium, High. The classification task here can be seen as linearly separable problem in which we can use any simple classifier to predict between those three different levels of a traffic. The classifier that I used in this project is a Support Vector Machine (SVM) model which I trained it for our specific problem.

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.

Chapter 2 - Device Specification

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

2.1 Hardware

Z2 board is a single-board computer (SBC) which is a complete computer built on a single circuit board, with ARM Cortex-A series processor, memory, input/output (I/O) and other features that are requirement of a functional computer. 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 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 to have 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].

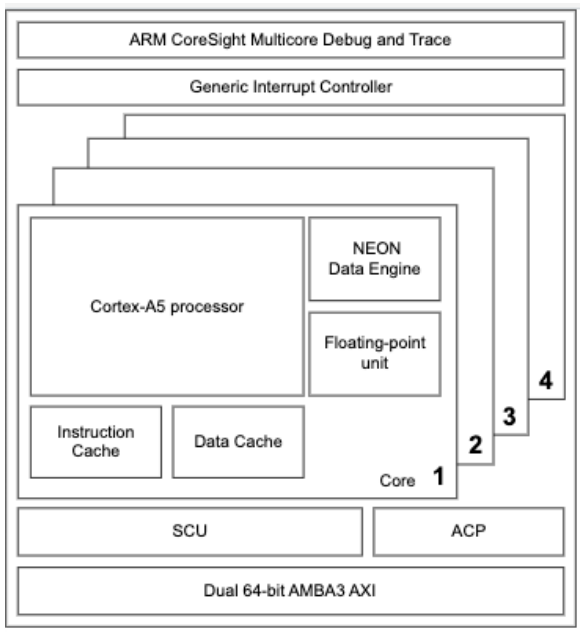


Figure 1: Cortex-A5 processor [6].

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 board uses a 244MB memory which normally the operation system uses half of it and eventually 174MB would be available for user. It has also 7GB disk drive with at least 3.5GB available for the user. It can also be tuned to use swap memory, but I do not use swap memory by default.

2.1.3 Modules

Mainly Z2 board has two miniPCIe modules for different purposes for instance a VPU accelerator, Camera, Wireless module can be plugged in to it. Of course, depends on the type of the 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.

Z2 board can use from a 5V up to 15V battery but it depends on how many modules or what type of modules is going to be used in addition to the board. In this project since I use two miniPCIe convertors and one WiFi connector plus one Usb-Camera, I provide a 14.8V battery.

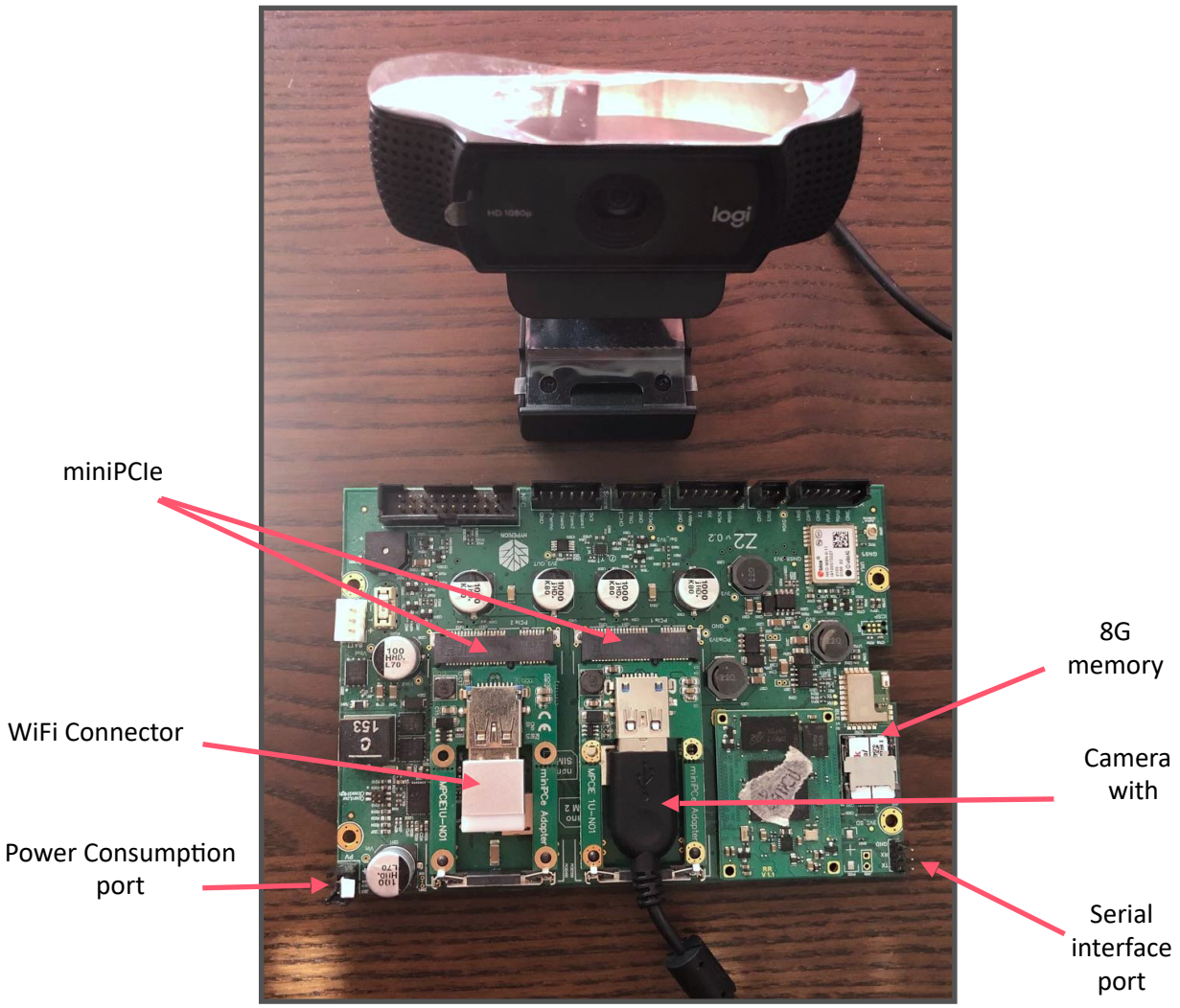


Figure 2: Z2 board with wifi and camera modules.

2.1 Software

In this section we are going to look at different software aspect of the project that runs on Z2 board such as type of operation system, build system, programming languages, etc. Z2 board just like Raspberry Pi is a very cheap computer that runs Linux as its operation 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 operation system as a result it runs a suite of open-source software and packages too.

2.2.1 Operation System

Linux is the most used operation system for single-board computer and embedded system by far. The operation 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++ would be the 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) directly translates the code without any further processing. On the other side, when using an interpreted language like Python, the embedded device does not directly translate the code. In fact, the code is executed by another interpreter software running on top of the device. Because they need to do additional processing, interpreted languages frequently run more slowly than compiled languages. However, they are much easier for programmers to learn, read and write; Python and Java are the best example of interpreter languages. Python is the most used languages 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 and used Python for training our deep learning model. 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.

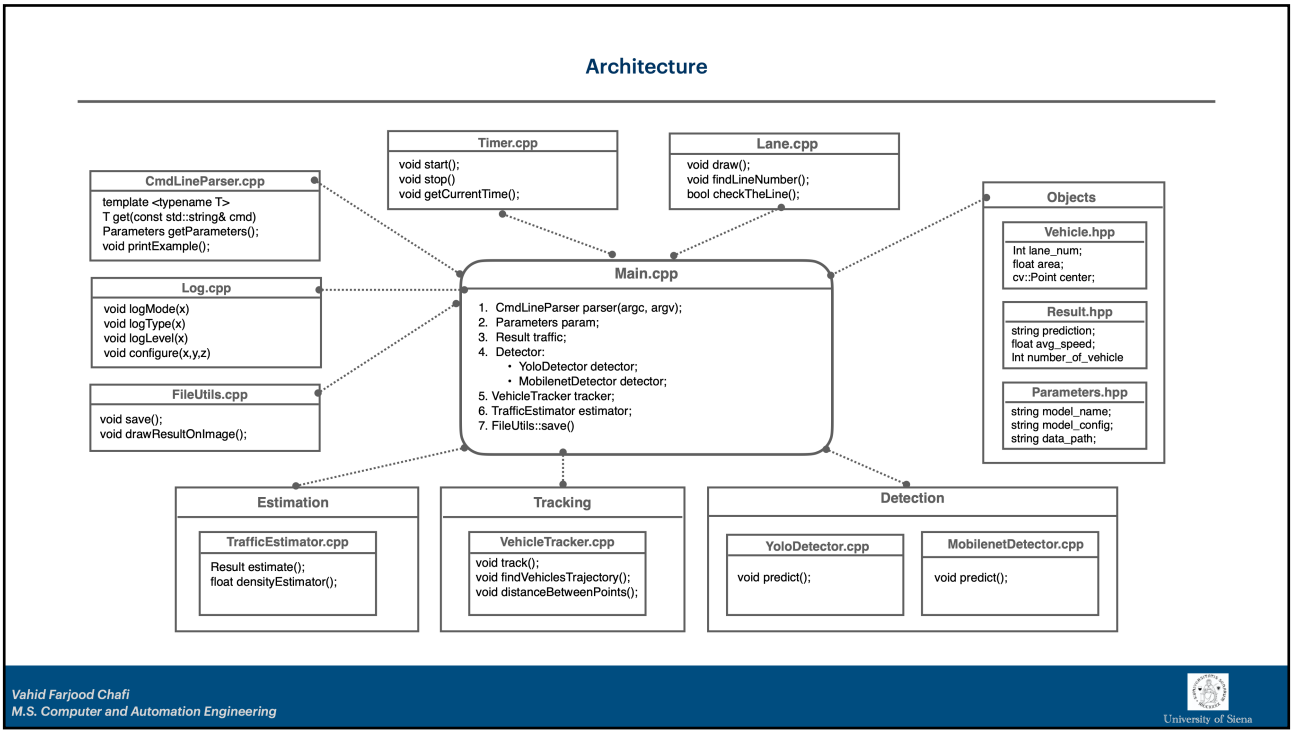


Figure 3: Software Architecture Diagram

As you can see on the above picture (Figure-3), program is designed in a modular base in which each task is implemented in a different file. All the different files are handled by the main.cpp file which is the core part of the application. First, command-line parser will parse input arguments and will set all those arguments into a Parameters 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 will start to detect all the vehicles on a given image which in our project we are aim at processing only two images. The result of object detector model is a '*std::vector<std::vector<Vehicle>> vehicle_vector;*' which means we will have a buffer that holds two other buffers that one of them is holding detected vehicles of image number one and the other buffer for image number two. After detection task is done, the result will be sent to 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 detection and tracking task, will be used to estimate the traffic based on SVM Classifier model in Estimator module. At the end, the result with all the necessary information will be saved on the disk for later use. 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 to the console or be saved into a file for later investigation.

2.2.3 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, there is

C++ version of some of them like LibTorch, TensorFlow-Lite or Caffe but most of them are not build for ARM processors, and those other which are built for ARM professor have a lot of compatibilities issues and they are not light enough to be installed on Z2 board such as TensorFlow-Lite. Therefore, I used OpenCV with its DNN module which has capability for 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.

2.2.4 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 files. 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].

2.2.5 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 its 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 Z2 board.

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 I can cross compile my 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.

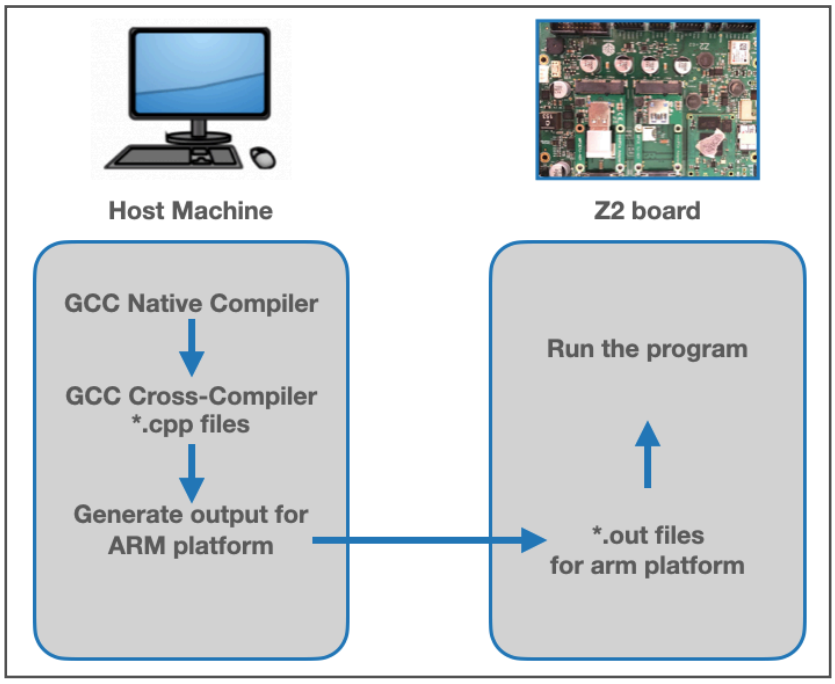


Figure 4: Cross-Compilation process from host machine to target machine.

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

This chapter is mainly about deep learning and what kind of algorithm is being used in this project. I will start explaining first looking at dataset which I used it for fine-tuning the models; then we will look at two different object detection algorithms namely YOLO and MobileNet SSD which are used in this project. Finally, I will describe how I implement tracking and estimation algorithm in order to count vehicles and predict the existence of a traffic queue, respectively.

3.1 Dataset

Data is a crucial component of Deep Learning, and the field is heavily depended on data to train, validate, and test the models. Computer Vision is one of the most important fields in AI and the data that it required to train its model is either video or images followed by image annotation. Image annotation means the process of visually indicating the location and type of objects that the deep learning model should learn to detect. Providing such datasets requires a lot of effort and time which makes it difficult especially when you are working on a specific scenario.

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.

Unfortunately, few datasets are released publicly in traffic scenes due to security issues. Therefore, I created a brand-new dataset by extracting pictures from traffic videos (35 videos on different scene and condition) which is publicly available on the internet 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 in 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 not universal for vehicle targets using in a variety of areas, but it is suitable for fine-tuning object detection models in order to detect vehicles in our specific traffic estimation task.

Since I used two different object detection algorithms in this thesis, I needed to produce two different annotation formats for each model namely for YOLOv5 and MobileNet SSD. So, depending on the annotation tool we use, we will need to make sure to convert

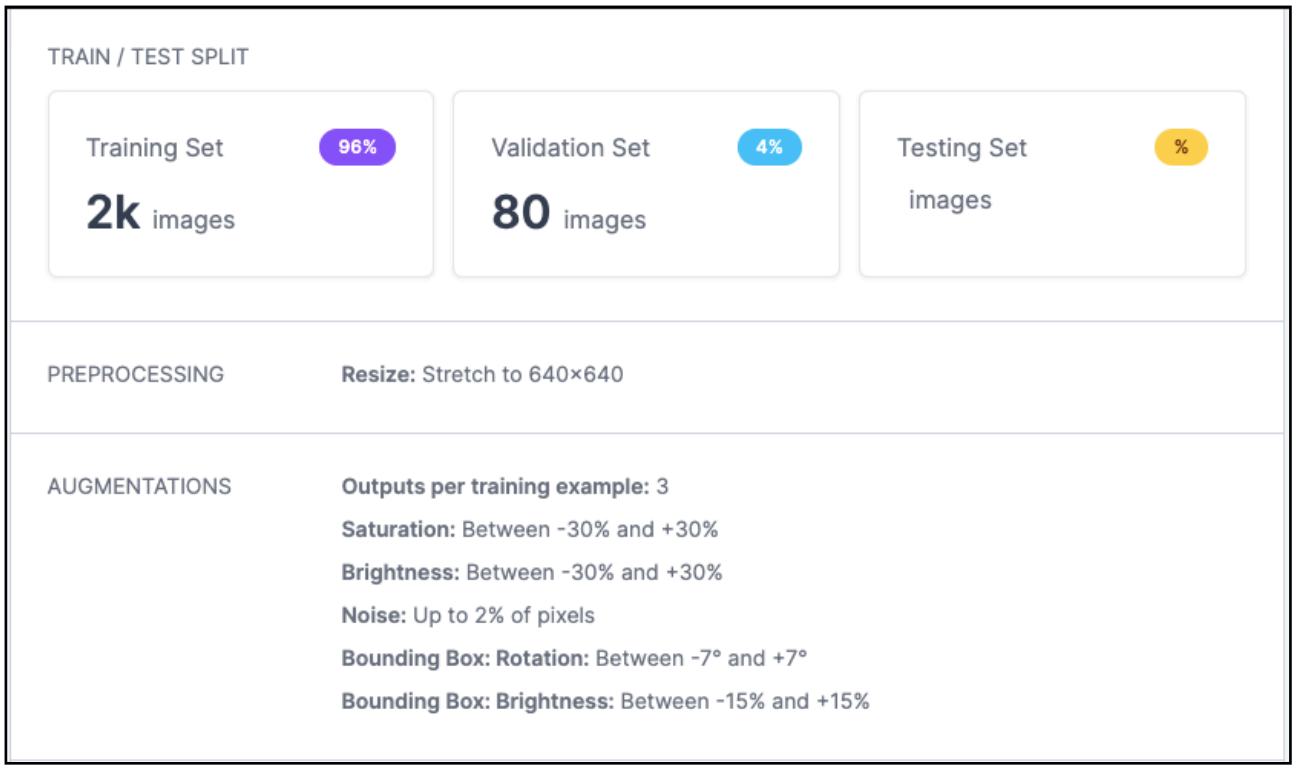


Figure 6: Dataset summary

annotations to work with our models. Roboflow [26] allows you to input 27 different labeling formats and export them to work with YOLOv5 and MobileNet SSD.

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 utilized 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 show the confidences of existence.

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 to two categories: One-Stage object detectors (like YOLO and SSD) and Two-Stage object detectors (like R-CNN family). While the algorithm of the one-stage detectors are faster, but the accuracy is relatively lower than that of the two-stage detectors which is relatively more accurate but slower (see figure-7). First, I will look at two-stage detections models and then will describe one-stage detections models.

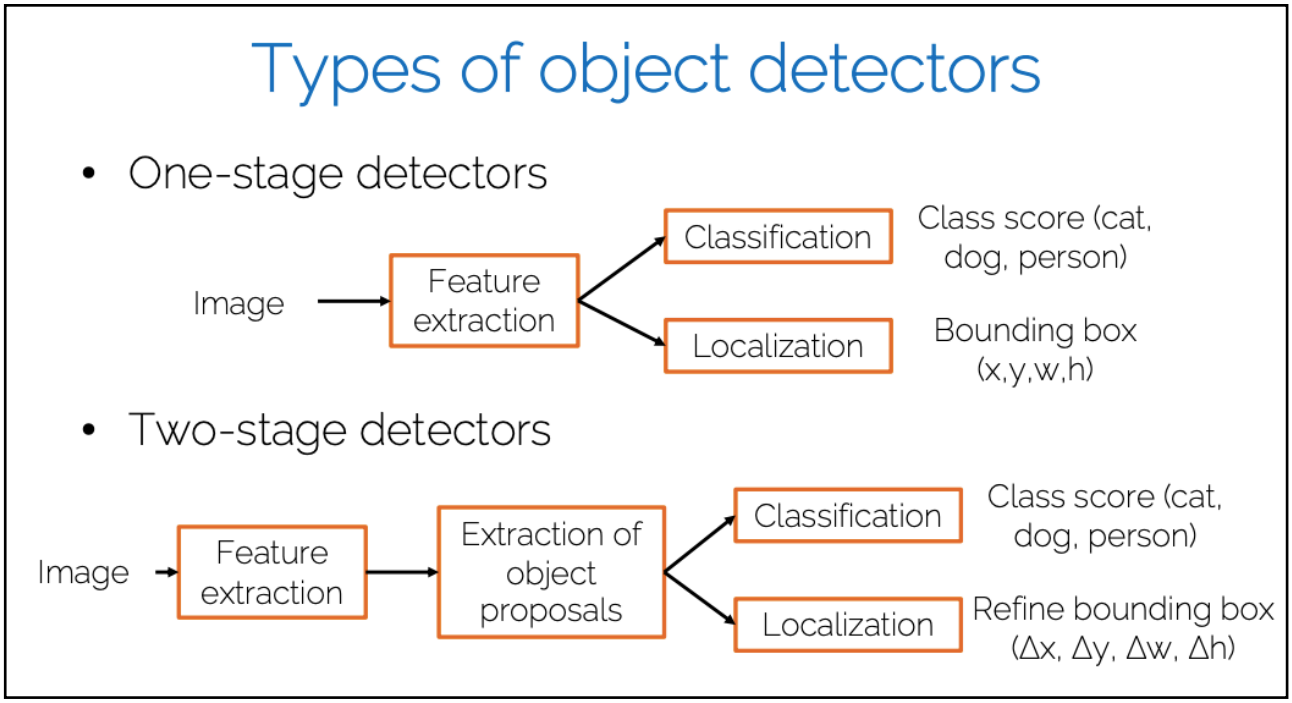


Figure 7: Types of object detectors

3.2.2 Two-stage object detector

Basically, two stage detectors used two step approach: first it generates the candidate boxes (which might have our objects to detect) and then will try to classify those boxes as one of the objects and at the same time adjust those boxes to properly fit the actual object which is bounding box regression.

the first and original two-stage detection is R-CNN which takes an input image, extracts around 2000 bottom-up region proposals, computes feature for each proposal using a large convolutional neural network (CNN), and then classifies each region using class-specific linear SVMs [14]. There are many different versions of R-CNN family after the first version, namely Fast R-CNN, Faster R-CNN. The main difference is in the architecture in which the Fast R-CNN gives us the ability to simultaneously train a detector and a bounding box regressor at the same time and under the same network configurations, and in Faster R-CNN the main change is the introduction of Region Proposal Network (RPN) that enables nearly cost-free region proposals by integrating most individual blocks of an object detection system, e.g. Proposal detection, feature extraction, bounding box regression, etc., into a unified, end-to-end learning framework.

While two-stage detections models are accurate, these approaches have been too computationally intensive for embedded systems and, even with high-end hardware, too slow for real-time applications. Therefore, my preference was to go for a one-stage object detection model such as YOLO or SSD.

3.2.3 One-stage object detector

One-stage object detections on the other hand, apply a single neural network to the full image. 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. Since they frame detection as a regression problem, then there is no need for a complex pipeline which make them extremely fast. Using a fixed grid of detectors is the main idea that powers one-stage detectors.

They have a Base network that acts as a feature extractor such as Inception or ResNet , 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.

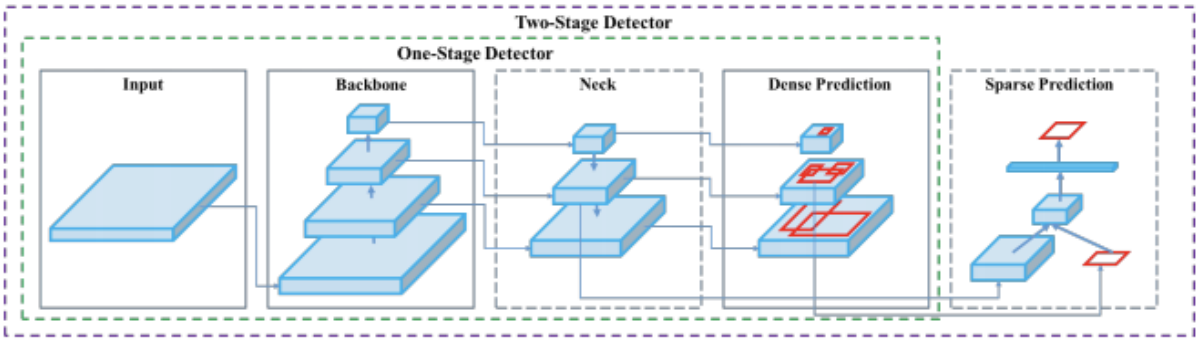


Figure 8: Overview of object detection procedure and architecture [20]

3.2.4 YOLOv5

The original YOLO(You Only Look Once) [18] was written by Joseph Redmon in a custom framework called Darknet. Until now, there is a total of seven versions of the YOLO algorithm. They are YOLO, YOLOv2, YOLOv3, YOLOv4, YOLOv5, YOLOV6, YOLOv7. 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. I used YOLOv5 in my thesis since at the time of starting this project YOLOv5 was the latest version.

The detection accuracy of YOLOv5 network [21] is quite high, and the inference speed is very fast, with the fastest detection speed being up of 140 frames per second. Moreover, the size of the weight file of YOLOv5 target detection network model is small, which is nearly 90% smaller than YOLOv4, indicating that YOLOv5 model is suitable for deployment to the embedded devices to implement real-time detection. Therefore, the advantages of YOLOv5 network are its 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 architecture contains five different architectures, specifically namedYOLOv5n [21], YOLOv5s [21], YOLOv5m [21], YOLOv5l [21] and YOLOv5x [21], respectively. The main difference among them is that the amount of feature extraction modules and convolution kernel in the specific location of the network is different. The size of models and the amount of model parameters in the four architectures increase in turn.

The YOLOv5s framework mainly consists of three components, including: Backbone network, Neck network and Head network.

1) **Backbone** - Backbone network is a convolutional neural network that aggregates different fine-grained images and forms 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. Then on the third layer of the backbone network is the BottleneckCSP[23] module, which is designed to better extract the deep features of the image. The final output of the Bottleneck module is the addition of the output of this part and the initial input through the residual structure. After that on the ninth layer of the Backbone network is SPP module (spatial pyramid pooling)[24], which is designed to improve the receptive field of the network by converting any size of feature map into a fixed-size feature vector.
2) **Neck** - The neck network is a series of feature aggregation layers of mixed and combined image features, which is mainly used to generate FPN (feature pyramid networks)[25], and then the output feature map is transmitted to the head network (prediction network). Since the feature extractor of this network adopts a new FPN structure which enhances the bottom-up path, the transmission of low-level features is improved, and the detection of objects with different scales is enhanced. 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 output from the previous layer, and outputs a vector with the category probability of the target object, the object score, and the position of the bounding box surrounding the object.

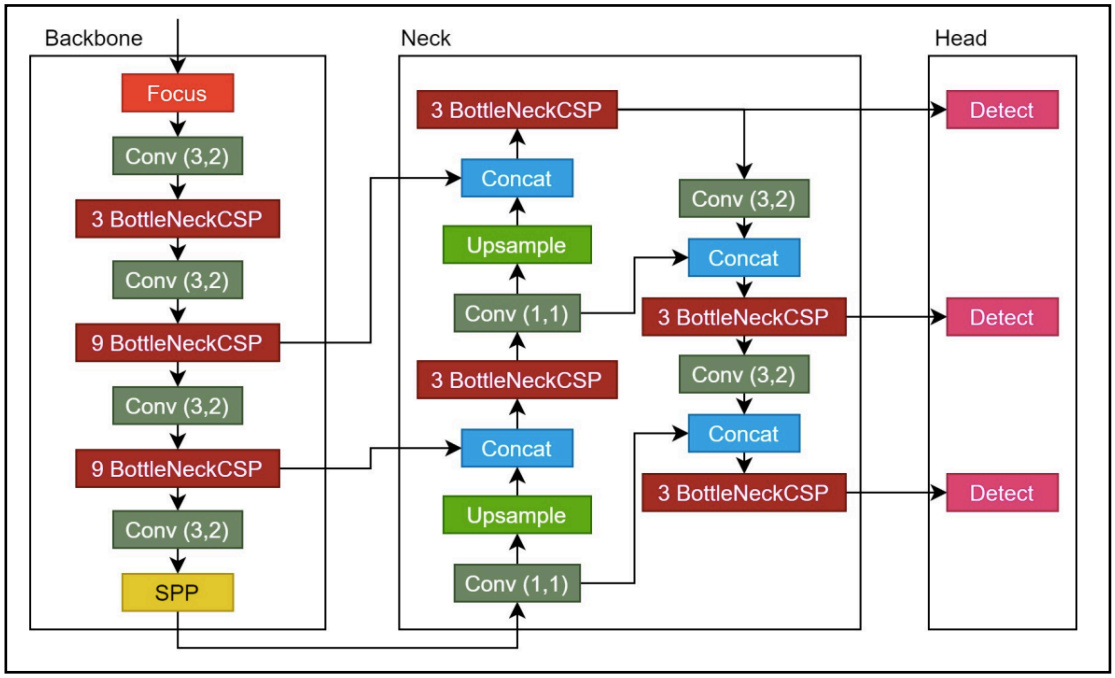


Figure 9: YOLOv5 architecture [22]

3.2.5 MobileNet SSD

As the name applied, 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 first layer, which is a full convolution, is an exception. Depth-wise separable filters significantly reduce the number of parameters when compared to the network with regular convolutions with the same depth in the nets. This results in light weight deep neural networks.

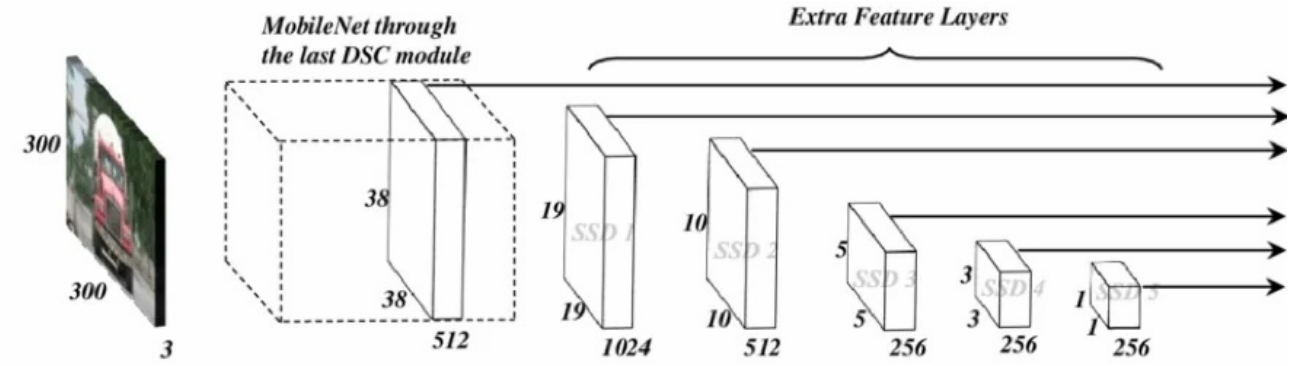


Figure 10: MobileNet Architecture [28]

Around the same time (2016), SSD[28], or single shot detector, was also developed by the Google Research team to cater to the need for models that can run real-time on embedded

devices without a significant trade-off in accuracy. It is very similar to YOLOv2, instead of using only a single feature map, what it does is it performs classification at different scales. So, during reducing the feature maps in convolution we do classification at each convolution layers. The main idea is that if your object is very small then it will get lost at the end layers (within those convolution and pooling layers). So, the last 7x7 feature map does not have enough resolution to detect small object, therefore we detect objects at different layers(scale).

SSD 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.

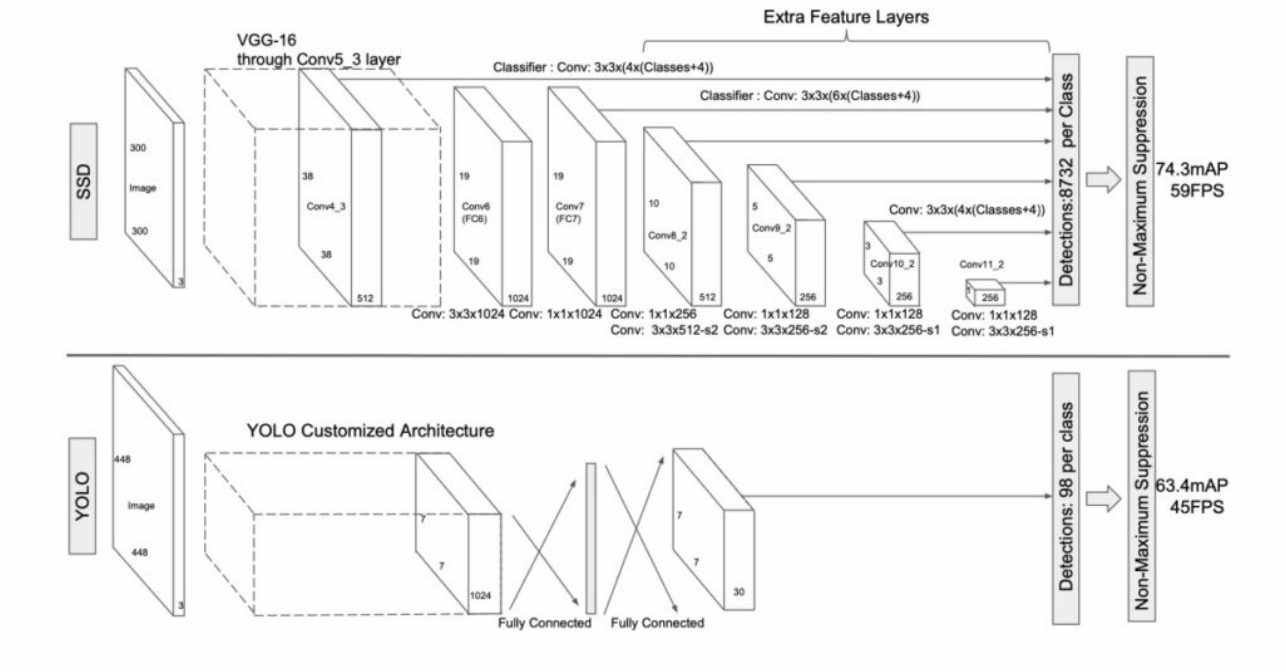


Figure 11: SSD vs YOLO architecture [28]

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 compute. It runs exceptionally well on CPUs instead of costly hardware accelerators like GPUs. The MobileNetV2[30] + SSD combination uses a variant called SSD Lite that uses depth-wise separable layers instead of regular convolutions for the object detection portion of the network, which

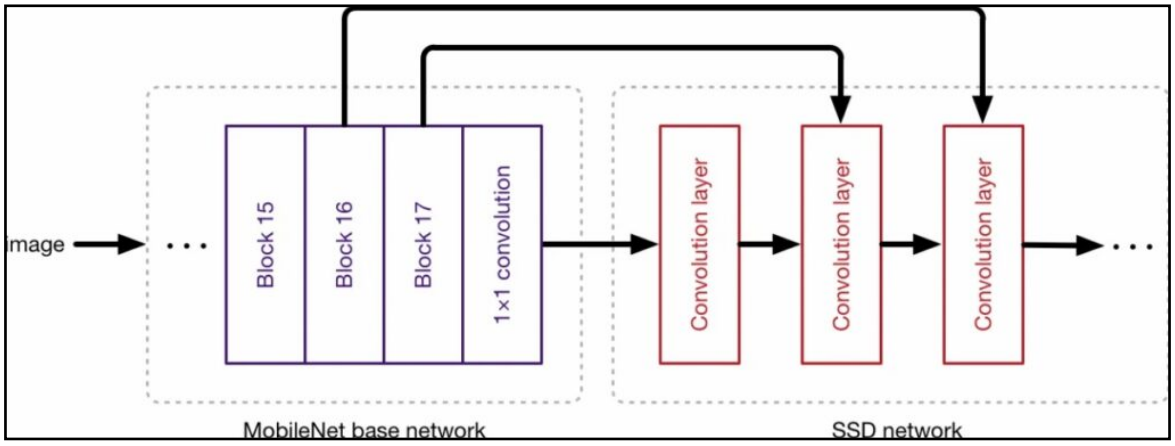


Figure 12: MobileNet SSD architecture [28]

makes it easier to get real-time results. When paired with SSD the job of the MobileNetV2 network is to convert the pixels into features that describe what's in the image and pass them onto the next layers. In this case the next layers can be considered as the ones from SSD. Here not only the outputs of the last MobilenetV2 layer are forwarded to SSD, but also some previous layers in order to get high- and low-level features.

3.2.6 Fine-Tuning

It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems. This approach 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 where we want to detect vehicles, therefore it makes sense to use those pre-trained models in order to ease training phase of the project.

Of course, we don't want the model detects 80 different classes but what we want is just 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 format manually or do it automatically by using Roboflow website which that's the case I used in this project. When you prepare a dataset in Roboflow website then you can easily export your dataset in any format depends on your target model framework you are using.

For YOLOv5 model I used YOLOv5n (Nano) pre-trained model which comparing to other version of YOLOv5 models, Nano model is much faster but less accurate than others which make sense in our project with an embedded device we need a very light and fast model. You can find more information about how to fine tune YOLOv5 model at: https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data .

Table 3.1 is all the parameters that I used for fine-tuning the YOLOv5n model

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 lost 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

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-tuned the model and can detect almost all the vehicles on the road.

On the other side, for MoblieNet_SSD there are only two versions of it and the difference is the size of input image that they accept. One of them 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 .

Table 3.2 is all the parameters that I used for fine-tuning the MobileNet SSD model

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 lost 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

Despite of many times I tried to improve the MobileNet SSD model by fine-tuning it with different hyper-parameters, still it does not produce a satisfactory result and I was not able to get a good result after testing the model. Although you can see it could detect vehicle into some degree as is shown below in figure-3.8 but when I used it on Z2 board the result was worth even.

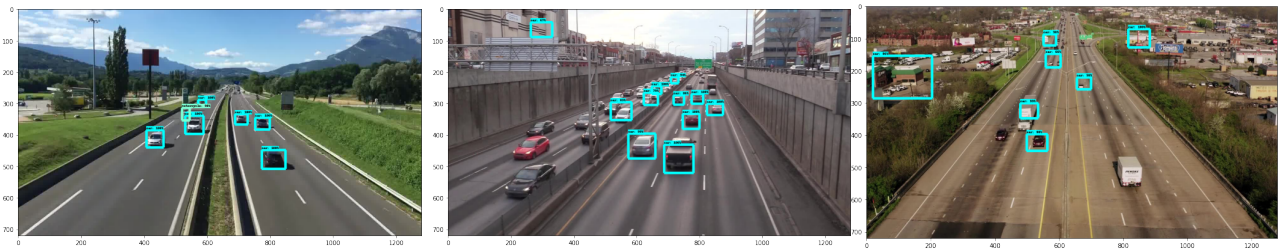


Figure 14: Detection result for MobileNet SSD model

If you consider the images shown above which are just three examples of many other that I tried to test the model, then you will realize that there are some vehicles that the model couldn't detect them. Moreover, there are some areas outside of the road that the model detects them as a car, for instance a part of the building.

3.2.7 Inferencing

We all know OpenCV as one of the best computer vision libraries out there. Additionally, it also has functionalities for running deep learning inference(DNN module) as well. The best part is supporting the loading of different models from different frameworks using which we can carry out several deep learning functionalities. The feature of supporting models from different frameworks has been a part of OpenCV since version 3.3 and the version that I used in this project is version 4.5.5.

One of the OpenCV DNN module's best things is that it is highly optimized for CPU. We can get good FPS when running inference on real-time videos for object detection and image segmentation applications. We often get higher FPS with the DNN module when using a model pre-trained using a specific framework. When we have the fine-tuned models, then we can use OpenCV DNN module to import these models and use them for model inferencing.

We can load a YOLO model by using *cv::dnn::readNetFromONNX(model_path);* or for MobileNet SSD using *cv::dnn::readNetFromTensorflow(model_path, model_config);* when

we loaded the model, then we can pass the input image to the model, start forwarding phase through our fine-tuned model and get an *cv::Mat* object as an output. Afterward, we need to post-process the output like applying Non-Maximum Suppression (NMS) or checking the confidence or any other kind of threshold that we have.

The job of detection module is finished after post-processing procedure and the output is ready to send to the next layer in the program pipeline for further processing which is tracking and estimation.

3.3 Tracking

The goal of this project is to detect the existence of a queue on the highroad but more importantly we aim to extract some quantitative information like number of vehicle or average speed, etc., from the road and then send them to a centralized server which could predict the traffic flow more accurately based on that information. To extract that quantitative information, I needed to first detect all the vehicles and then track them to count and estimate their average speed plus some more information. So, the idea is that we are going to capture two sequential images from the highway (fps is one frame per second), then run one of our object detections models to detect vehicles in both images. After that we need a way to identify each vehicle and put a label on them from those two pictures to count and estimate their speed otherwise we won't be able to determine whether the vehicle in the first image is the same on the second picture or not. If we don't track them, then it would cause to count all vehicles as a new object which is not correct. Moreover, to estimate their average speed, we must calculate their distance that they passed between two frames, so we need to identify each vehicle from frame to frame.

3.3.1 Tracking algorithm

We need to identify each vehicle in each consecutive frames and the task is called object tracking; and in case where we have more than one object in a picture then we call it Multiple Object Tracking (MOT) which is defined as to discover multiple objects in individual frames and recover the identity information across continuous frames, i.e., trajectory, from a given sequence [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.

Although there are some Deep Learning object tracking algorithms like Multi-Domain Net (MDNet) or Generic Object Tracking Using Regression Networks (GOTURN) but in this thesis since we are limited by hardware resources, I skip using deep learning algorithms and instead tracking task is done by making some assumptions and using computer vision techniques. Assumptions are as follow:

1) A vehicle cannot cross from one lane to another lane during two frames.

2) Shape of the box is not incremental based on the camera calibration (top-down and rear-view).

Now, based on those assumptions we can track each object from consecutive frames using three pieces of information about boxes:

- **Which lane number they belong to.** Will identify their lane number for both images.
- **How close their centers are on the consecutive frames.** Given a decent frame, we can assume that the vehicle cannot suddenly move from one corner of the image to another – which means that the centers of the detections of the same vehicle on consecutive frames must be close to each other.
- **The area of the boxes.** The area of the given bounding box should be decreased by a small amount in the next frame (according to the camera view).

Having that information, we can combine them into a measure of how likely is it that two boxes represent the same vehicle. In fact, many tracking algorithms use an internal movement prediction model. It remembers how the car moved previously and predicts the next location based on a movement model. Vehicles typically do not move randomly but rather go in a consistent direction. So, this technique really helps match the detections to the right track but since in this project I am going to use only two consecutive frames, then it does not make sense to use an internal prediction model because for prediction you need to have some history of the object at least 2 or more.

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

To start tracking I first need to identify each vehicle's lane number and the reason is that based on our assumption in which no car can cross their lane within one second, therefore we can narrow down the tracking problem space by comparing only vehicles that are on the same lane number. So, it would be much easier for us to say that the detected vehicle on the lane number one in frame t_1 will be highly likely again in lane number one in the next coming frame t_2.

We can choose the place of our interest where we are going to install the camera (it should be on top of a highway on a bridge) and take a sample picture from the road. The camera will capture back side of the vehicles from a top-down view. After that, we must determine each lane manually on the image space. It means for each lane we have a vector of pixels corresponding to the real lane on the road. This is a task that we need to do it in advance. Figure 16 is an example of the camera view with the lane drawing.

Having defined lane pixels in advance, then we just need to check where each vehicle center point is located and assign its corresponding lane number. In OpenCV there is a line iterator object class which provides you the ability to go through each pixel of a given line.

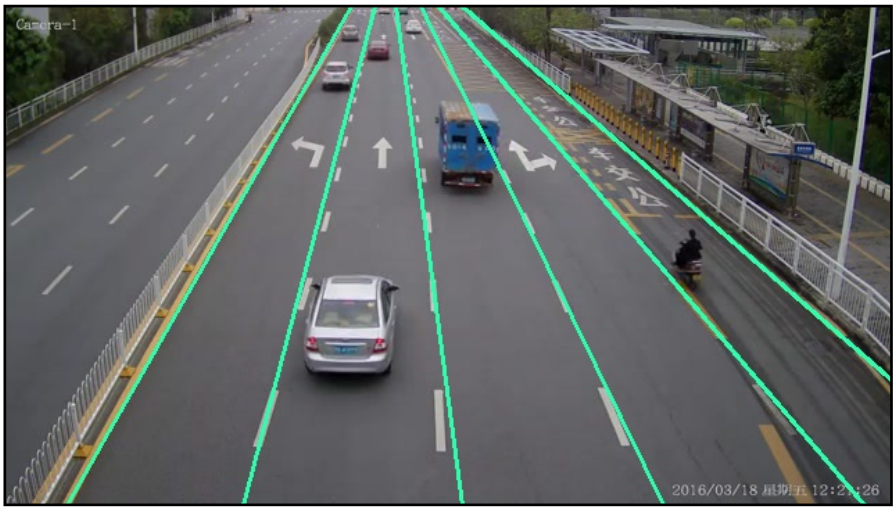


Figure 16: Camera view with lane separation

So you can iterate over a line and check if the point of your interest is located between two line or not. This object class is called “cv::LineIterator” and I used it in order to identify each vehicle lane number.

3.3.3 Distance between points

The distance formula which is used to find the distance between two points in a two-dimensional plane is also known as the *Euclidien distance* formula. To derive the formula, let us consider two points in 2D plane A (x_1, y_1), and B (x_2, y_2). Assume that 'd' is the distance between A and B:

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

So, for any point given in the 2-D image, we can apply the2D distance formula or the Euclidean distance formula in order to get the distance between two point or in our case distance between two vehicles. We can use this distance to check if the vehicle is close enough to the vehicle of our interest or not. And if the vehicle is in the same lane number but its not the closest one then probably it is a different vehicle but staying in the same lane number.

3.3.4 Area of a bounding box

After lane calculation, we can then compare a vehicle from the same lane number that if its area is larger than the area of a vehicle on the same lane number in the next frame, then probably that’s the same object. If you consider the view of the camera (rear view) you will realize that the size of a car should decrease by each frame because it will get further and farther as the time of capturing frames goes on.

The area of a square and a rectangle shape is simply the product of its two adjacent sides. Which means for a 2D bounding box we have:

$$area = Length \times Width$$

3.4 Estimation

In previous sections of this chapter (section 3.2 and section 3.3), I explained how to detect, track and collect some useful information about each vehicle but now we need to impose some techniques in order to estimate traffic flow on the highway based on that

information. The most important features that we can extract from that information would be estimating the number of existing vehicles on the road, estimating average flow speed, and calculating the density of the road. Then, with these three features we can use a simple classifier to predict traffic level on a given road.

3.4.1 Vehicle counting

The number of vehicles is simply calculated after we identified and tracked all the existing vehicle from those two frames. It means, after the job of tracking algorithm is done we will have a C++ vector that each one of its elements is a tracked vehicle without duplication. So, we just need to get the size of our vector and that's the number of vehicles we captured during those two frames.

3.4.2 Speed estimation

Speed Estimation based on video stream without any sensor is not an easy task and need many considerations. In this project I tried to relax the task and estimate an approximate speed of a vehicle, which would be sufficient for our traffic estimation system.

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, 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.
Also, we should consider that the cameras are aligned with the horizon. 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, by mapping the relation between pixel distance and actual distance, the speed is estimated by linking the corresponding vehicles distance between the current frame and the previous frame and divide it by time which here the time is 1 since the delay between two consecutive frames is one second.

3.4.3 Density estimation

Road density is a fundamental characteristic for estimating the traffic level. It tells you how significant the congestion of cars on the road is. If the density reaches its maximum,

the flow drops to zero, forming a traffic jam. I calculate density as the sum of all tracked vehicles area **a** that occupy a segment of the road of a total area length **L**, and then divide these two values and multiply them by hundred to get density percentage: (**density = a / L * 100).**

Sum of all tracked vehicles area can be simply calculated by summing up all the detected vehicle areas. Total road length can be calculated based on the total area of the drawn lanes on the image. If we take all the outer lanes left most line, right most line, top line and down line, then the type of the shape we get is a trapezoid. A trapezoid is a 4-sided geometrical shape with two sides parallel to each other. These two sides(**a** and **b**) are called the bases of the trapezoid. The other two sides(**c** and **d**) are called legs. **h** is the height of the trapezoid[33].

To find the area of a trapezoid (**A**), we need to find the length of each base (**a** and **b**), find the trapezoid's height (**h**), then substitute these values into the trapezoid area formula: **A = (a + b) × h / 2**.

A grid of total covered road length area is shown on below picture (figure-3.10). I provided a method called *FileUtils::drawGrid(cv::Mat image, Lane& lines);* in FileUtils.cpp in which you can use it in order to draw a grid on the picture.

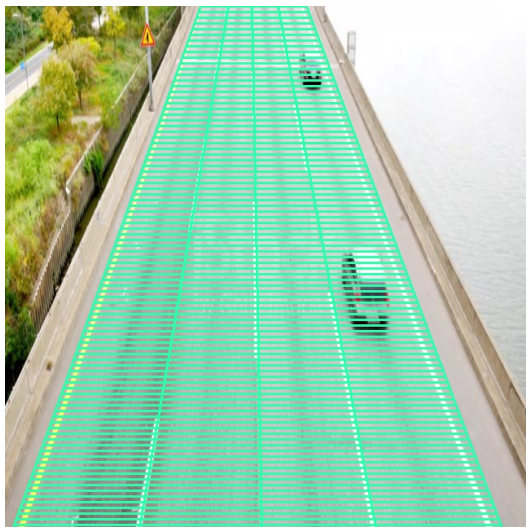


Figure 17: Road total length coverage

3.4.4 Traffic estimation

Finally, when we extract all the necessary information from the given input image, then we can use a classifier for estimating the traffic level. This classifier will use that three information as its input features and will predict the level of the traffic. For simplicity, I defined the quality of a traffic flow in three different levels: Low level, Medium level, High level. The classifier can be any type of decision maker algorithms whether it uses machine

learning or a simple classifier such as a Bayesian Decision classifier.

In this project, I used SVM algorithm as the final classifier. I trained this svm classifier with a brand-new dataset. The dataset contains 90 samples data with corresponding class label (Low, Medium, High). Each one of those sample data is an array of 3 values namely: Number of vehicles, Density of the road and Average flow speed. These three important features are actually that quantitative information that we extracted from our detection and tracking procedure, therefore, we have to train our classifier with input data that have those three features.

So, I created a dataset by extracting those features from different sequential pictures of the real-world traffic video downloaded from the internet. The extraction procedure was down by running the program without classifier but each time I compared the result with the real images and then If the extracted features was reasonable, I saved those features (number of vehicles, density, average flow speed).

Having that pre-trained model, then we can attach the classifier at the end of our program pipeline in which the extracted features from detection and tracking modules, can be sent to this classifier as an input and after processing them by this SVM classifier, it can finally predict the corresponding traffic class label. The architecture of the entire pipeline is shown in figure-18

For the future, we might devise a deep learning algorithm in server side to make the final decision but since this is the first version of this work I assume there is no server yet and I try to estimate the traffic level based on a simple SVM classifier. All the informations including svm prediction result will be stored in a file for later use.

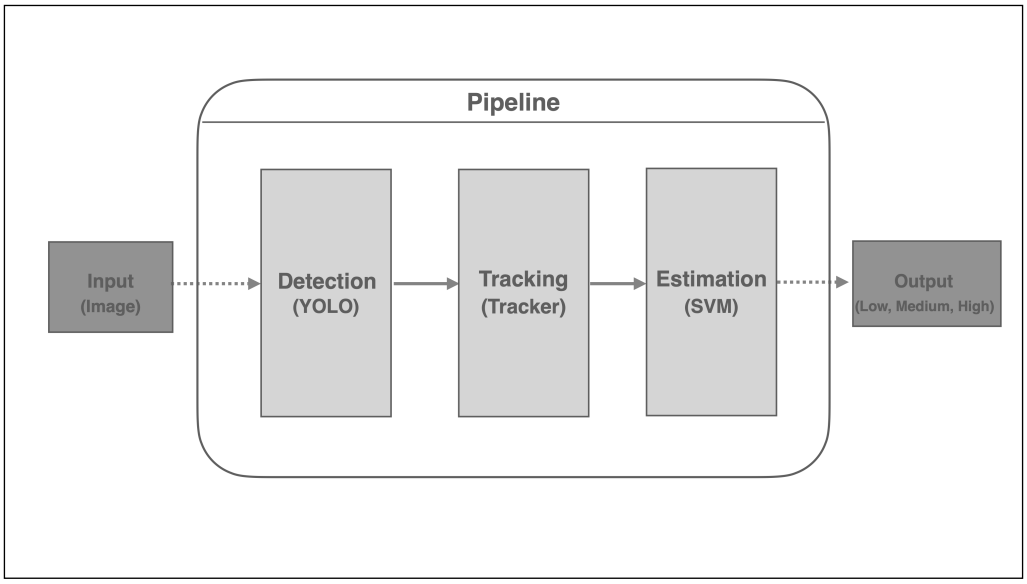


Figure 18: Entire Pipeline of program procedure

3.5 Performance

I considered the performance in terms of three different criteria: Speed, Accuracy and Power consumptions. Speed can be easily estimated by considering how long the program takes running to give us an output. The same is also for power consumption in the way that we need to use a wall outlet power meter or usage monitor device to physically measure the amount of energy it spent during one cycle of running the program. On the other side, for accuracy, we need to take into account some more considerations since there are three main different modules in the program pipeline, detection, tracking and estimation.

3.5.1 Speed

To calculate the speed, I used a C++ standard library called std::chrono for overall running time and for calculating time spent for each different parts of the program. Computed time for detection is for loading the model, loading two images and processing those two images with one of the detection models. The total detection time was usually around 52 seconds for YOLO model versus total detection time with MobileNet is 72 seconds. For tracking and estimation time, it is usually very fast and its less than 0.01 second. Therefore, the overall time of running the program from reading the input image up to writing the result takes 53 seconds for YOLO model and 73 seconds for MobileNet SSD model which we can see there is 20 seconds differences between these two models.

Table-3.7 shows the speed comparison between YOLO and MobileNet SSD computation time for processing two images.

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

Notice that loading time for yolo model is less than half a second but on the contrary, for MobileNet SSD loading time for the model is around 7,5 seconds.

3.5.2 Accuracy

Measuring the accuracy of the whole pipeline is very difficult since there were three different computation modules and each of them has different effect on the quality of the result. For detection, I used Mean Average Precision(mAP) to measure model quality based on my dataset once I fine-tuned the model. For tracking, it directly depends on the quality of object detection models. So, if we properly detect all the vehicles of two pictures, then tracking algorithm will work with a high accuracy. The estimation classifier would be even more accurate if we train the model with more input data in which is provided based on real traffic flows. Since there was no proper dataset to evaluate the algorithms, I tried to evaluate them based on some real experiment and analyzed their correctness based on observing the real evidence. I evaluate them on 10 different roads and for each road I used 4 sequential images which means two different tests per road. Table-3.8 illustrated an approximate evaluation of the program quality using two different detection algorithms with 20 test samples.

Pipeline	YOLO (mAP@.5)	MobileNet_SSD (mAP@.5)
Detection	0,766	0,226
Tracking	95%	15%
Estimation	98%	20%

Table-3.8 Accuracy comparison between YOLO and MobileNet SSD

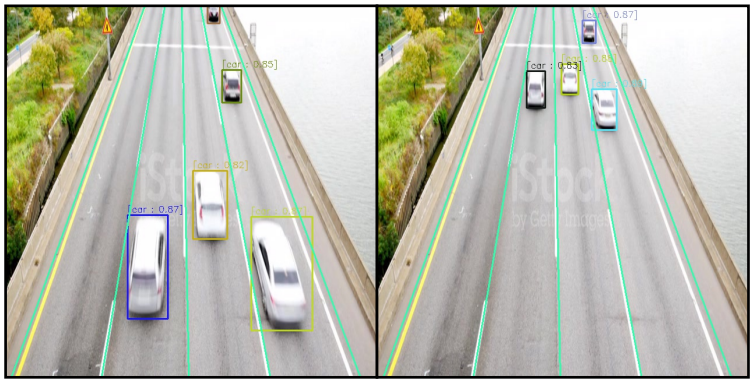
I must mention that the entire pipeline is heavily dependent on detection models and as you can see from the table YOLO model outperformed MobileNet SSD by far. However, it only has some error if there is a heavy traffic with a lot of occlusions or if it gets dark in the road. Even though, with some small error on YOLO detection process, the result still will be correct since the SVM model predicts the result based on probability distribution of the dataset. That's why in almost all most cases, 98% the test was correct and that two percents incorrect prediction is either when we did not define the lines properly or the dataset for training the SVM classifier was not enough.

3.5.3 Power Consumption

Computing power consumption needs some hardware specific equipment, and we cannot accurately measure the power consumption of the Z2 board by just software. There needs to be some hardware in place and the easiest option would be to use an oscilloscope or multimeter with data logging capabilities if they are available to us. This can be done by measuring the current drawn from our entire system along with the voltage of the power supply. It is usually wise to measure the voltage instead of just assuming it will remain at its expected value. Of course, If we are going to measure the energy consumption only for our specific application, then in this case, we may need to first measure the base energy consumed by Z2 board when it is idle. Then we can measure the energy consumed by running the program. Once we have those values, we can subtract them and see what the difference is.

Below, you can find some examples that I tried to test the model with. Those image samples have never seen by the model and they are extracted from videos which had a completely new location.

Frame 1 Frame 2 Result:



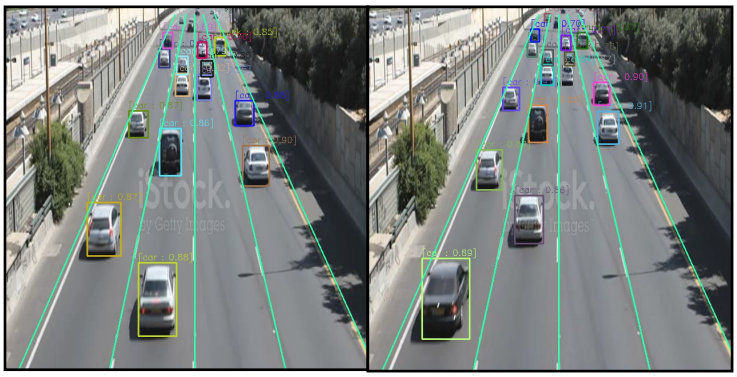
Output for YOLO:
• Traffic Status: Low
• Number of cars: 5
• Flow density: 28%
• Flow average speed: 114
• Highest flow is in lane number: 4

Frame 1 Frame 2

Result:

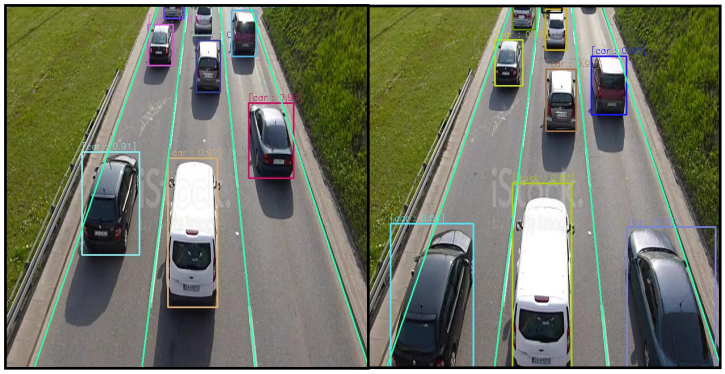
Output for YOLO:
• Traffic Status: High
• Number of cars: 33
• Flow density: 133%
• Flow average speed: 19
• Highest flow is in lane number: 3

Frame 1 Frame 2



Output for YOLO:
• Traffic Status: Medium
• Number of cars: 16
• Flow density: 24%
• Flow average speed: 21
• Highest flow is in lane number: 1

Frame 1 Frame 2



Output for YOLO:
• Traffic Status: Medium
• Number of cars: 9
• Flow density: 95%
• Flow average speed: 47
• Highest flow is in lane number: 2

Frame 1 Frame 2

Output for YOLO:
• Traffic Status: High
• Number of cars: 33
• Flow density: 129%
• Flow average speed: 8
• Highest flow is in lane number: 2

Chapter 4 - Experiments

To have a final test for the program, I needed to do a real experiment. An experiment that would show a result as if the device could be installed in a real location on highways. So, in this chapter, I explained how and where I did some real experiment with the project.

4.1 Overview

There are two different experiments in different locations. For the first experiment I used my cellphone to capture images and then transferred them into Z2 board to test it. However, for the second experiment all the task from capturing images up to the final estimation happened by the project itself. Meaning that we used even its own camera and power supply to test the entire workflow.

All the pictures were taken on a bridge where the camera was setup to capture up-down and rear view of vehicles. It happened during a good weather condition within day light, so no occlusions.

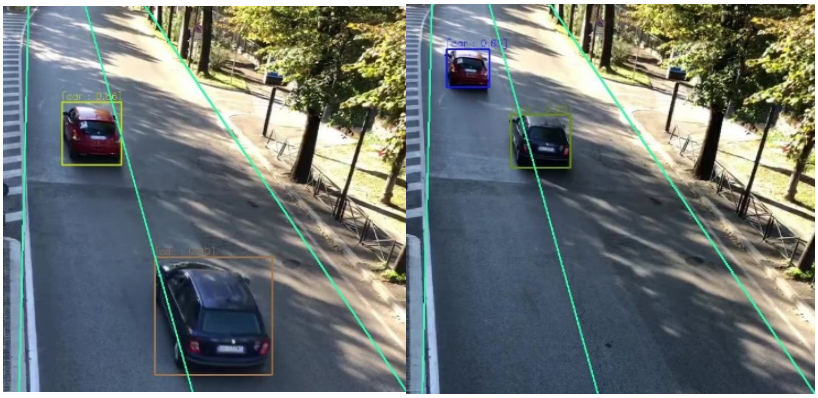
The first experiment was done on a specific street in urban area of Rimini city with a low traffic flow. The third experiment was done on a highway in Siena with low, medium, and high traffic flow.

4.1.1 Rimini experiment

Since it was the first experiment, I used my cellphone to capture images without taking the device to the location. So, I captured a video for 5 minutes with a low traffic level. Afterward I started to extract some sequential images from it to make it input sample data. Based on those images, first I defined its lanes on the picture and created line.txt file which is a requirement for the tracking and estimation algorithm. Then I test the model with 3 input data which means each input data had 2 sequential images (with one second delay between each image).

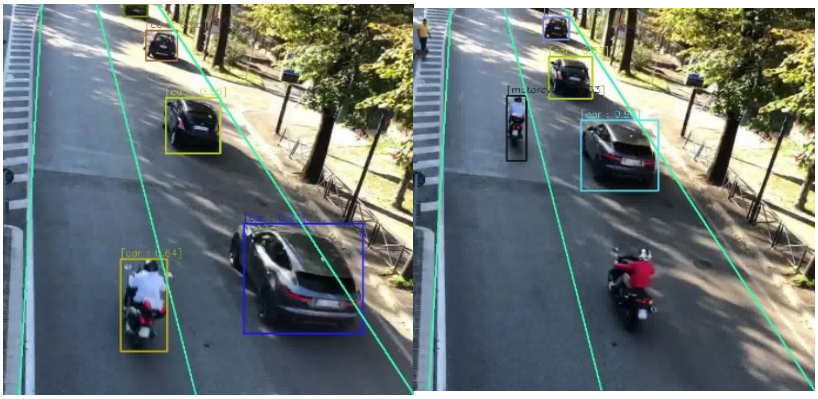
Below you can see two different examples for each detection models:

Frame 1 Frame 2



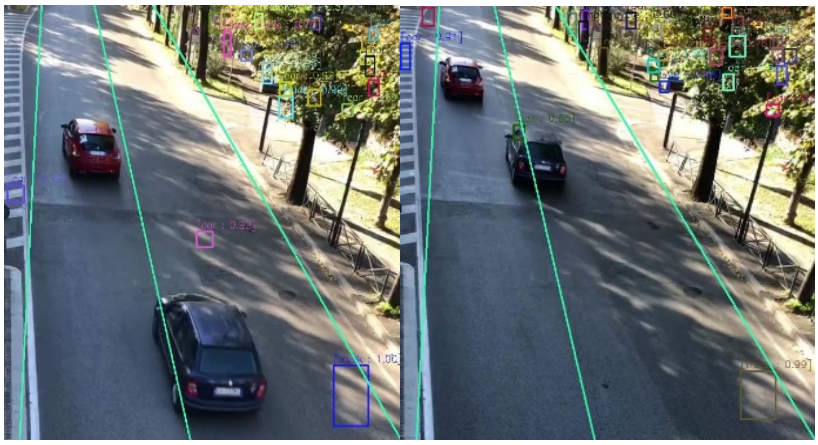
Output for YOLO:
• Traffic Status: Low
• Number of cars: 2
• Flow density: 33%
• Flow average speed: 107
• Highest flow is in lane number: 2

Frame 1 Frame 2



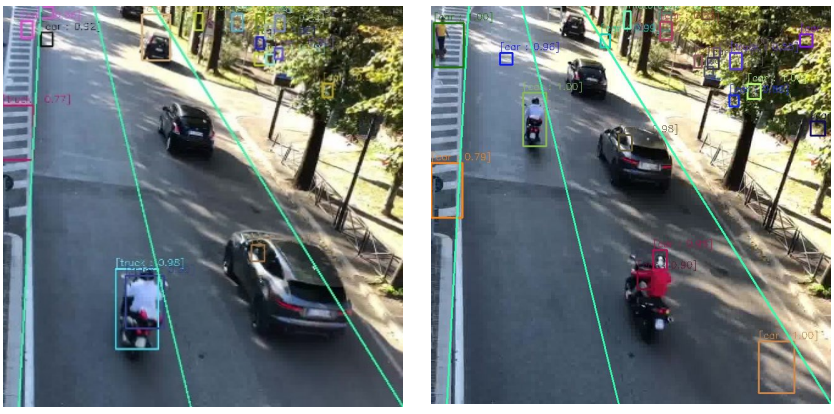
Output for YOLO:
• Traffic Status: Low
• Number of cars: 5
• Flow density: 39%
• Flow average speed: 84
• Highest flow is in lane number: 2

Frame 1 Frame 2



Output for MobileNet SSD:
• Traffic Status: Medium
• Number of cars: 24
• Flow density: 10%
• Flow average speed: 45
• Highest flow is in lane number: 0

Frame 1 Frame 2



Output for MobileNet SSD:
• Traffic Status: Medium
• Number of cars: 22
• Flow density: 26%
• Flow average speed: 91
• Highest flow is in lane number: 0

As you can see on the above pictures, the YOLO model performed very good, and it detects all the objects which results in a correct estimation that is low traffic. But on the other side, MobileNet SSD did not perform well, and it detects incorrectly shadows and other things as vehicles, which estimate incorrectly medium traffic.

4.1.2 Siena experiment

This final experiment,

Chapter 5 - Conclusion

In this work, I have conducted an experimental thesis in which two modern object detection algorithms namely YOLOv5n and MobileNet_SSDv2 were used for the automatic traffic estimation problem on highways. These object detection models with the tracking algorithm were used as a feature extractor for estimator algorithm which is a SVM classifier. There were 3 main features extracted from detection/tracking algorithms and those are Number of vehicles, Density of the road, and Average flow speed of the road. Based on those features, the SVM model classifies the traffic flow in three levels: Low, Medium, High. By this approach, we start from a problem which was originally non-linearly separable to a linearly separable problem in which at the end of the day with a simple classifier we can predict the existence of a traffic queue with its corresponding quality level.

The first part of the program pipeline which is detection, is the most important part of the whole program and it has a direct impact on the result. Therefore, the result is heavily depending on the accuracy of the detection models. Based on this experiment, YOLOv5 model performed satisfactorily, achieving both high marks in speed and accuracy. However, MobileNet_SSDv2 model did not performed well, achieving both low in speed and accuracy. Possible explanations could be due to different object sizes, illumination conditions, image perspective, partial occlusion, complex background, that MobileNet SSD suffers from. Another possible explanation could be due to fine-tuning procedure which I might needed to try to change model parameters maybe or fine-tune the model for with more training data and more epochs.

Future work includes extending this analysis to additional detection models, notably the YOLO (Y*ou Only Look Once*) family of models, or FOMO (Fear of Missing Out) model. Improving tracking and estimation algorithms will also provide achieving a better result. Apart from improving algorithms, we also need to prepare a more comprehensive dataset for fine-tuning detection and estimation models.

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