



**NUST COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING**



Perception Stack of a Self-driving car

A PROJECT REPORT

DE-40 (DEE)

Submitted by

NC Ali Akram

NC Mirza Ahmed Aftab

NC Muhammad Umer Ahsan

BACHELORS

IN

ELECTRICAL ENGINEERING

YEAR 2022

PROJECT SUPERVISOR

Dr. Fahad Mumtaz Malik

NUST COLLEGE OF

ELECTRICAL AND MECHANICAL ENGINEERING

PESHAWAR ROAD, RAWALPINDI

DECLARATION

We hereby declare that no portion of the work referred to in this Project Thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning. If any act of plagiarism is found, we are fully responsible for every disciplinary action taken against us depending upon the seriousness of the proven offence, even the cancellation of our degree.

1. Ali Akram

NUST ID: _____

Sign: _____

2. Mirza Ahmed Aftab

NUST ID: _____

Sign: _____

3. Muhammad Umer Ahsan

NUST ID: _____

Sign: _____

COPYRIGHT STATEMENT

- Copyright in text of this thesis rests with the student author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the author and lodged in the Library of NUST College of E&ME. Details may be obtained by the Librarian.
- This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.
- The ownership of any intellectual property rights which may be described in this thesis is vested in NUST College of E&ME, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the College of E&ME, which will prescribe the terms and conditions of any such agreement.
- Further information on the conditions under which disclosures and exploitation may take place is available from the Library of NUST College of E&ME, Rawalpindi.

ACKNOWLEDGEMENTS

First and foremost, we thank Allah Almighty, the Most Merciful, for the strength He has given upon us, the bounties He has bestowed upon us, and the well-being He has bestowed upon us. All of these contributed to the project's timely and successful completion, Alhamdulillah.

We wish to extend our sincere gratitude to our supervisor Dr Fahad Mumtaz malik for his continuous support and mentoring. Without his constant counselling and assurance, we would not have been able to make the progress we have.

This section would be incomplete without acknowledging the unwavering support of our families and friends and most importantly our seniors. They encouraged us to pursue our dreams and their endless faith in our abilities helped us achieve them. Their acceptance for our field of choice, confidence in us during our difficult moments and understanding of the strenuous work routine enabled us to reach the position we have. We have been able to achieve this important milestone of our professional lives today because of their love and support.

ABSTRACT

Over 1.3 million people die in car crashes each year, according to the WHO (World Health Organization), and it appears that most accidents are caused by driver error. Self-driving cars can improve safety by drastically reducing collisions and saving lives. Because making self-driving cars can remove human error, it gives some serious safety benefits to these sophisticated artificial intelligence systems.

The aim of our project is to develop algorithms for robotic perception. This includes main tasks of static and dynamic object detection, object tracking, depth estimation, collision avoidance, visual odometry and semantic segmentation for drivable surface area. This project is developed on the extensive knowledge of computer vision, deep learning and robotics which employs different AI algorithms to help the vehicle analyse the environment around it

We used and integrated number of state-of-the-art algorithms such as YOLO and VGG that gathers information around the vehicle using monocular and stereo cameras and process it in real time on jetson development kit and enables our vehicle to move efficiently.

TABLE OF CONTENTS

Contents

DECLARATION	1
COPYRIGHT STATEMENT	2
ACKNOWLEDGEMENTS	3
ABSTRACT	4
ACRONYMS	7
CHAPTER 1: INTRODUCTION	8
1.1 Overview	8
1.2 Motivation.....	8
1.3 Objectives	9
1.4 Approach.....	9
CHAPTER 2: Literature Review	10
2.1 Background	10
2.2 Deep Learning.....	10
2.2.1 Convolutional Neural Networks.....	11
2.2.2 Overview of Deep Convolutional Neural Networks.....	15
2.2.2.1 LENET	16
2.2.2.2 SINGLE SHOT DETECTORS	16
2.2.2.3 YOLO (YOU ONLY LOOK ONCE)	18
2.3 Hardware Selection	21
2.3.1 Device Chosen	22
2.3.2 NVIDIA JETSON XAVIER	22
2.3.3 Cameras.....	26
CHAPTER 3: Image Processing	28
CHAPTER 4: Camera Calibration	30
4.1 Camera Geometry and Image Formation	30

4.1.1 Pinhole Camera Model.....	30
4.1.2 Central Projection as Linear Mapping using Homogeneous Co-ordinates	31
4.1.3 Principal Point Offset	31
4.1.4 Rotation and Translation of camera.....	32
4.2 Camera Distortions.....	33
4.2.1 Radial Distortions:	33
4.2.2 Tangential Distortions:	33
4.3 Camera calibration methodology.....	33
CHAPTER 5:	35
Lane Detection for Autonomous steering	35
5.1 Computer Vision Approach	35
5.2 Deep Learning Approach.....	38
CHAPTER 6 :	39
Traffic Signs & Light Detections.....	39
1. YOLO V4	39
2. Why we went for YOLO V4.	39
3. Dataset	40
4. Training with Google Collab	40
5. Setting up DARKNET Environment.....	40
6. Modifying YOLO V4 architecture	41
7. Creating YOLO V4 backup weights in Drive	42
• Visualizing training results.....	43
CHAPTER 7: Object Detection	45
7.1 introduction.....	45
7.2 MODES AND TYPES OF OBJECT DETECTION.....	45
7.3 Basic working structure.....	45
7.4 Results.....	46
CHAPTER 8: 3D Object Detection	47
8.1. Introduction	47
8.2. Training Scheme and Loss Function	48
8.3. Inference of Yolo3D object Detection.....	48
CHAPTER 9: Depth Estimation	49

9.1. Monocular Camera using deep learning Approach	49
9.2. Stereo Camera using computer vision Approach.....	50
Cv2.StereoSGBM function parameters:	51
CHAPTER 10: Hardware Optimization	53
10.1. Tensor Rt	53
10.2. ONNX.....	54
CHAPTER 11: Robotic Operating System (ROS)	56
11.1. Overview:	56
11.2. Tools :	56
11.2.1. rviz	57
11.2.2. rosbag	57
11.2.3. catkin	57
11.2.4. rosbash	57
11.2.5. roslaunch	57
Conclusion	58
References.....	58

ACRONYMS

- TSR** : Traffic Sign Recognition
- CV** : Computer Vision
- GTSDDB** : German Traffic Sign Detection Benchmark.
- CNN** : Convolutional Neural Network.
- DNN** : Deep Neural Network.
- YOLO** : You Only Look Once.
- SSD** : Single Shot Detector.

CHAPTER 1: INTRODUCTION

1.1 Overview

In self-driving automobiles, perception refers to how the vehicle perceives its surroundings. It is both the most important and the most difficult thing. Because we have eyes, hearing, and human intelligence, sensing the environment around us is a simple process for humans, but it is the most difficult and complex work for cars. We have eyes to see the surroundings, ears, noses, tactical sensors, internal sensors that can monitor muscle defluxion with those sensors, and we can perceive the environment around us with those sensors. We can accomplish a lot of things thanks to all of these sensors, and the brain has made it all possible. Our brain is constantly processing information. Most of our brain is dedicated to the perception i.e. for visual perception and the sub-conscious so that we can know where we are in the world. When coming to car they have somewhat different sensors they have cameras instead of eyes they also have some magic sensors like radar and lidar which can help in measuring the raw distances. so instead of knowing something in-front of me these sensors tell the exact distance in centimeters. So here the complex task involved is to take the huge amount of data from sensors and use the computer intelligence to evaluate data and make something meaning of it.

1.2 Motivation

Smart driving is the upcoming trend, and the most basic level of autonomy is based on environment perception and driver information. In the world of Artificial Intelligence and advancement in technologies, many researchers and big companies like Tesla, Uber, Google, Mercedes-Benz, Toyota, Ford, Audi, etc. are working on autonomous vehicles and self-driving cars. So, for achieving accuracy in this technology, the vehicles should be able to interpret traffic signs and make decisions accordingly. Since the world is shifting towards AI, ML and CV (Computer Vision) technique hence our main motivation

for this project is that it finds its application in ADAS (Advanced Driver Assistance System) in Pakistan. This is important because a prompt response to real-time traffic events can prevent road accidents.

1.3 Objectives

- Object Detection
- Lane Keeping
- Collision Avoidance
- Traffic sign and light detection
- Depth Estimation
- Tracking
- Segmentation
- ROS Integration

1.4 Approach

Our group will work on perception in self-driving cars that uses a combination of high-tech sensors and cameras, combined with state-of-the-art software to process and comprehend the environment around the vehicle, in real-time. This includes to implement both traditional and deep learning approach such as Lane keeping assistance, 2D and 3D object detection and tracking, Traffic signs and traffic light detection, collision avoidance system and behavior cloning. We will be using USB-Camera for visual recognition of objects and Stereo-Camera for Stereo Vision and depth estimation.

We did our project based on one of the latest deep learning models (YOLOV4) and accomplish a comparison across the proposed algorithms.

CHAPTER 2: Literature Review

2.1 Background

Perception has benefited an outsized number of realistic applications, like driver assistance system, autonomous vehicles, and intelligent mobile robots since they need delivered the present state of the environment into various systems. However, there are a couple of difficulties for computers to acknowledge, which are mainly from two aspects: One is said to the complex scene, the opposite is about unbalanced class frequencies within the datasets .

As for the problem of real-world scenes, traffic signs are always well designed for drivers to simply read and recognize the signs during the driving time, including vivid colors, strong and bolded words, also as various specific and simplified shapes, it is a difficult task to style the features combined with contaminated conditions. For instance, the conditions are with weak illumination, small-size signs in scenes, partial occlusions, rotations and physical damages. All those factors will have an enormous impact on the performance of computer algorithms to perceive environment.

2.2 Deep Learning

Deep learning is a machine learning technique used to build artificial intelligence (AI) systems. It is based on the idea of artificial neural networks (ANN), designed to perform complex analysis of large amounts of data by passing it through multiple layers of neurons.

There is a wide variety of deep neural networks (DNN). Deep convolutional neural networks (CNN or DCNN) are most commonly used to identify patterns in images and video. DCNNs have evolved from traditional artificial neural networks, using a three-dimensional neural pattern inspired by the visual cortex of animals.

Deep convolutional neural networks are mainly focused on applications like object

detection, image classification, recommendation systems, and are also sometimes used for natural language processing.

The strength of DCNNs is in their layering. A DCNN uses a three-dimensional neural network to process the red, green, and blue elements of the image at the same time. This considerably reduces the number of artificial neurons required to process an image, compared to traditional feed forward neural networks.

Deep convolutional neural networks receive images as an input and use them to train a classifier. The network employs a special mathematical operation called a “convolution” instead of matrix multiplication.

The architecture of a convolutional network typically consists of four types of layers: convolution, pooling, activation, and fully connected.

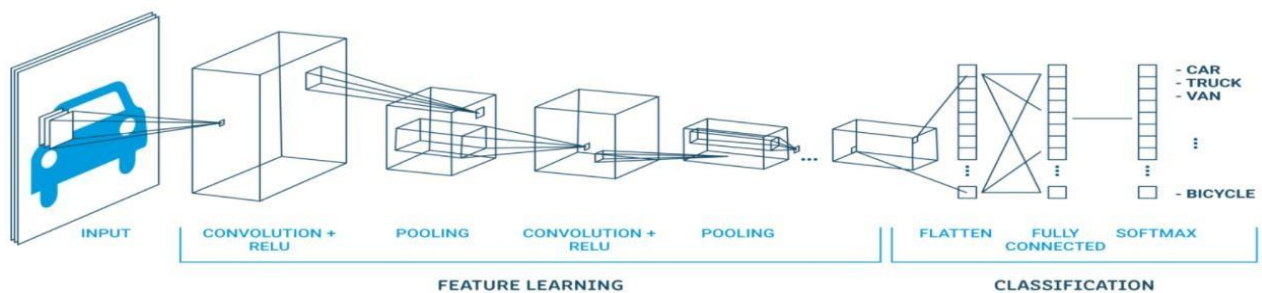


Figure 1 Architecture

2.2.1 Convolutional Neural Networks

Neural networks are the computing systems comprising of numerous interconnected elements that process the information by changing the response based upon the external inputs. It is based upon basic neural structure of a mammal’s cerebral cortex. The basic structure. It has organization of layers having interconnected nodes representing an activation function. Communication is done through weighted connection between input and hidden layer. Hidden layers are then connected to the output layer that classifies the input. (Neural Networks, n.d.)

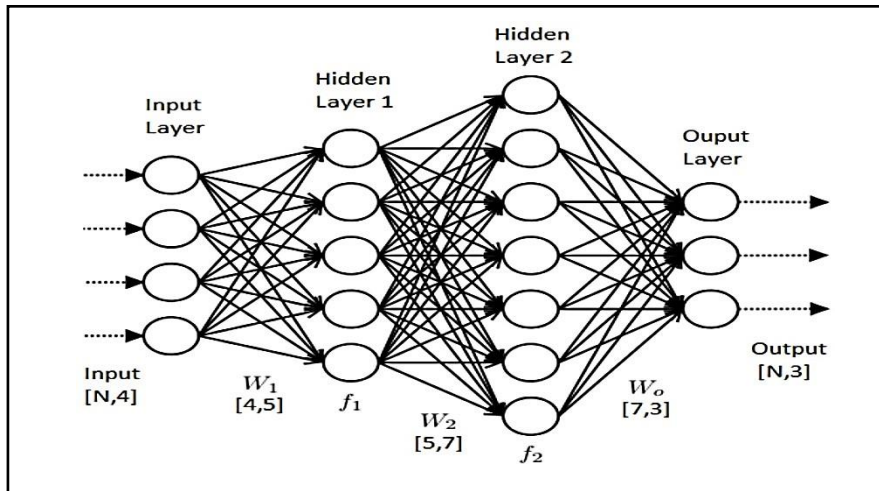


Figure 2 Neural Networks

Convolutional Neural Networks are in accordance with ANN. It is a technique of Deep Learning common nowadays. Convolution involves a simple filter on the input followed by an activation. With repetitive filter application an activation map called the Feature Map is formed indicating the strengths of different features of the input that is an image. CNN is involved in automatic learning and training of large database. It has a set of layers which are explained.

2.2.1.1 Convolutional Layer

This layer takes in 3-D image as an input. Digital images have 3 channels. Images move through the convolutional networks as a matrix of different dimensions and

the dimensions lessen from layer to layer. Each pixel is represented in the form of $30 \times 30 \times 3$ and its intensity of Red, Green and Blue would be a number showing element one of the three channels. Convolutional network has a filter and filter moves on the image as a matrix as well. The filter falls on the patch of the image and pixel wise multiplication take place in that region. The result is placed on another matrix called activation map that equals the times dot product taken. The profundity of the convolutional layer is equivalent to the profundity of an info picture.

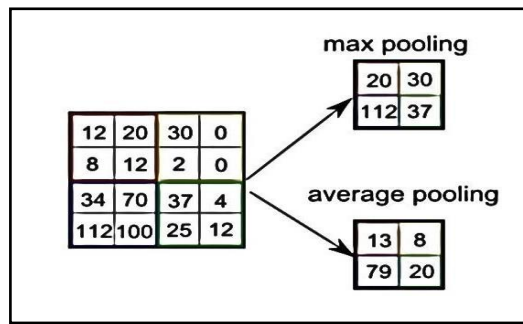
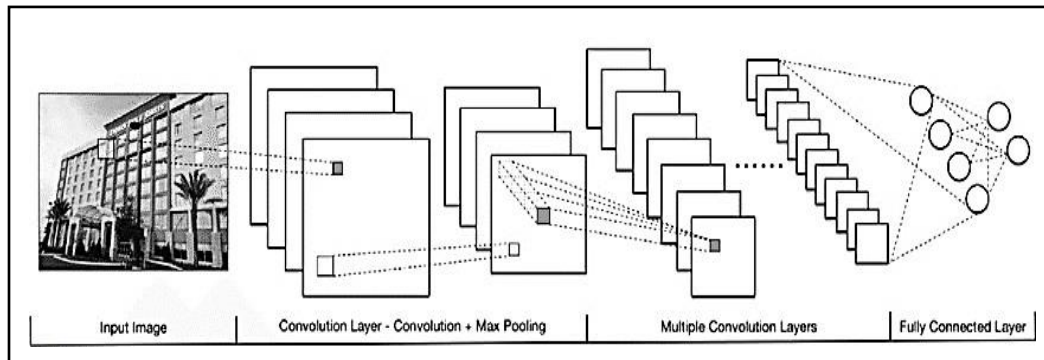


Figure 3 Convolutional Layer



2.2.1.2 Pooling Layer

For the reduction of spatial size of the Feature Map pooling layer is necessary. It helps to reduce processing capacity required for the processing of data by the reduction in dimensions. It is also helpful for the extraction of Key Features from the input that are invariant to rotations and position as well. There are two kinds of pooling:

- Max Pooling: Largest pixel intensity within a mask is returned.
- Average Pooling: This returns average (i.e., summation over total) of all the values present under the mask.

It acts like a Noise Removal/Subdue as it removes the noise present. Max pooling is better in this regard so mostly it is used. The convolutional layer along with max pooling layer form a single layer of CNN. The complexities of the input image define the number of layers required for the computation. These layers help the model to understand the features of the input image correctly and follow

forward to the next step of CNN.

2.2.1.3 *Flattening*

Flattening is the last part of the CNN model which converts high dimensional array to 1-D array. The output is flattened for the formation of a single long feature vector.

2.2.1.4 *Classification- Fully Connected Layer*

It has Softmax or ReLU as their activation function at the output layer. Fully connected means that every neuron of is connected to other layer coming next. The main purpose of the layer is to use the features for the classification of the input image. In addition to the classification, it is also a cheaper way out for the learning of non-linear combinations of the features. The convolutional and pooling layer may help in the classification, but the various combinations of the features would work even better. The probabilities of this layer add up to 1 and to make

sure, we use Softmax function as the activation function which converts the real values in values between 0 and 1.

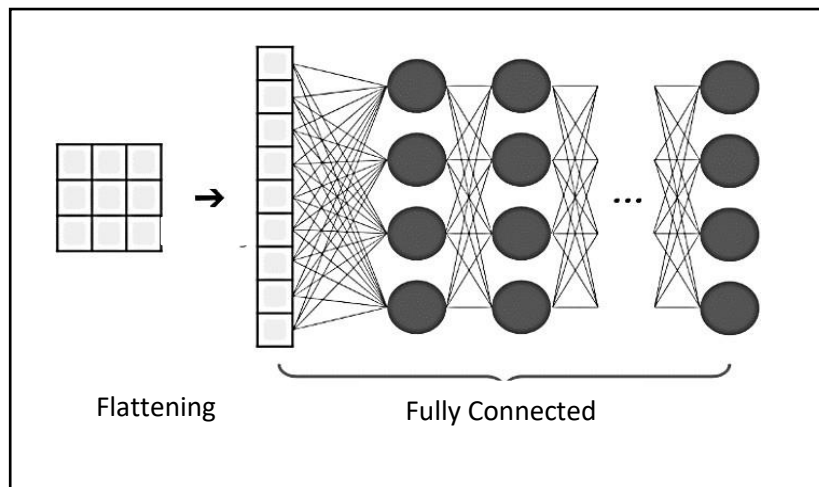


Figure 5 Fully Connected Layer

2.2.1.5 *Real Time Object Detection Using CNN*

The recognition and classification tasks can be easily solved using convolutional neural

networks. A neural network is some mathematical model that consists of interconnected artificial neurons. The network accepts the characteristic vector as input, and then sequentially passes them through the layers of the network. At the output, the probabilities of belonging to the given classes are obtained. Usually, a neural network operates with numeric, and not symbolic values.

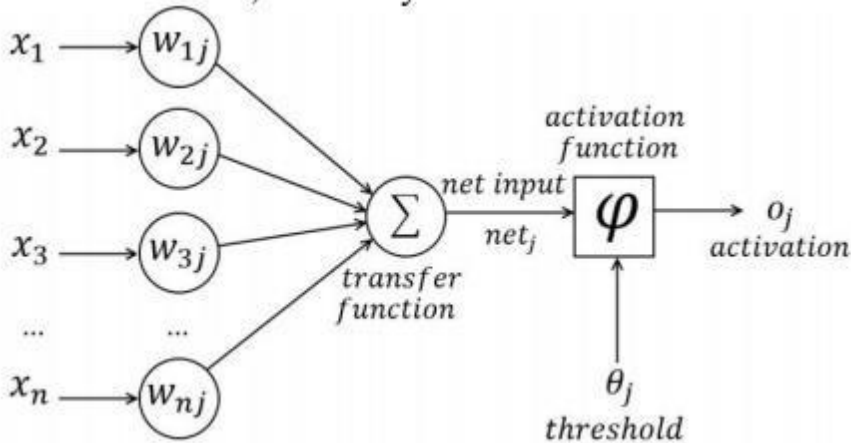


Figure 6 Scheme of an artificial neuron

Figure 6 shows the scheme of an artificial neuron. As an input, it considers the parameters, which are either initial data or output parameters of other neurons. Each parameter has a weight, which is a multiplier of every parameter. Then the weighted parameters are summed using some function. The resulting value is sent to the, which, after calculating the result, decides whether to transmit the signal to the next neuron. The output value will act as one of the parameters in another neuron. At present, convolutional neural networks are most effective for image processing. A two-dimensional image is applied to the input of the convolutional neural network, which is then processed by convolutional layers. The convolutional layers transform the image fragments into a feature map.

2.2.2 Overview of Deep Convolutional Neural Networks

To start off with recognition, we first investigated neural networks that could be used to classify datasets accurately. Moreover, we investigated neural networks that could be used to perform object detection within a natural scene and extract the required

objects. For this purpose, we came across a range of neural networks.

2.2.2.1 LENET

The presented method uses a modified LeNet-5 network to extract a deep representation of traffic signs to perform the recognition. Lecun et al. [27] proposed the well-known LeNet-5 convolutional neural network that is mostly used for handwritten recognition. Besides it was introduced in 1998, it became popular to solve other problems due to its simple and efficient architecture. It is composed of 7 layers, 3 Convolutional layers followed by Sub-sampling layers (except in the last), 1 Fully connected layer and the final output layer composed of Euclidean RBF units. The input size for this network is 32x32 pixels. Jung et al. [10] used LeNet-5 to classify 6 types of Korean traffic signs obtaining an accuracy of 100% correctly recognizing 16 signs while driving on the KAIST campus road. As the results were promising in their study, we also trained the network with our proposed dataset for comparison. It is constituted of a Convolutional Neural Network (CNN) modified by connecting the output of all convolutional layers to the Multilayer Perceptron (MLP). The training is conducted using the German Traffic Sign Dataset and achieves good results on recognizing traffic signs. LeNet is a well-developed architecture developed by Yann Lucan. It is optimized for processing images and can process most type of images which also includes classifying traffic signs. It was also proven to work pretty good on the previous works carried out on traffic signs.

2.2.2.2 SINGLE SHOT DETECTORS

In CNN approach, Image classification takes an image and predicts the object in an image. Let's say we built a cat-dog classifier with CNN and predict images. For instance if there is an image with both cat and dog present, we need to identify the location of the objects in image, with Object detection algorithm (e.g. RCNN). Unlike image classification, detection requires localizing (likely many) objects within an image.

Classification+ Localization=Object Detection

The difference between object detection algorithms (e.g., RCNN) and classification algorithms (e.g., CNN) is that in detection algorithms, we try to draw a bounding box around the object of interest (localization) to locate it within the image.

From the above discussion we got to know that solutions for real-time image recognition are divided into two general types: Region Proposal (one by one the regions of a frame are proposed and classified) and Single Shot (all objects are simultaneously recognized in the whole image). The first type includes such neural networks as R-CNN [4], Fast R-CNN [4], Faster R-CNN

The second one includes YOLO CNN [5], SSD [6]. Neural networks using recognition by region have a rather slow recognition time for qualitative detection of objects. However, for mobile platforms, Single Shot CNNs are more suitable, as they are quite faster.

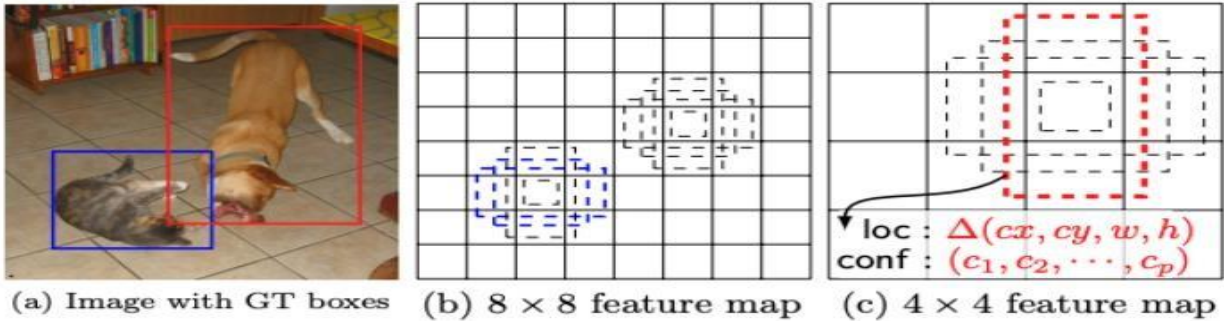
Since we did not want to compromise on speed and wanted to come up with both detection and classification problems, we started to dig in Single Shot solution for both localizing and recognizing traffic signs in images. In this regard we came across YOLO CNN AND SSD MOBILE NET.

MOBILE-NET SSD

Mobilenet-SSD is an object detection model that computes the bounding box and category of an object from an input image. This Single Shot Detector (SSD) object detection model uses Mobilenet as backbone and can achieve fast object detection optimized for mobile devices.

ARCHITECTURE:

Mobilenet-SSD takes a (3,300,300) image as input and outputs (1,3000,4) boxes and (1,3000,21) scores. Boxes contains offset values (cx, cy,w,h) from the default box. Scores contains confidence values for the presence of each of the 20 object categories, the value 0 being reserved for the background.



At training time, we first match these default boxes to the ground truth boxes. For example, we have matched two default boxes with the cat and one with the dog, which are treated as positives and the rest as negatives. The model loss is a weighted sum between localization loss (e.g. Smooth L1 [6]) and confidence loss (e.g. Softmax).

Figure 7 SSD Framework

2.2.2.3 YOLO (YOU ONLY LOOK ONCE)

YOLO CNN is a convolutional neural network that allows to detect and classify objects in the form of bounding boxes. Such bounding box is the minimum sized rectangle, which will contain the whole found object. YOLO works on the principle of Single Shot. This means that the network architecture is arranged in such a way that in one pass of the frame, all objects are detected simultaneously.

ARCHITECTURE:

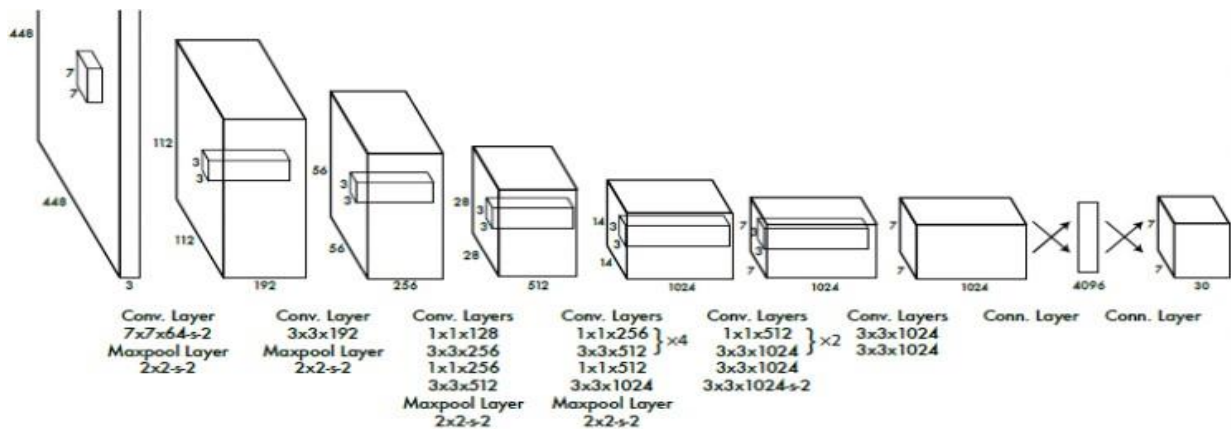


Figure 8 Architecture of YOLO CNN

Figure 8 shows the architecture of YOLO CNN. The YOLO input is provided with a three- channel image, which is resized to 448x448. The first conversion is to run the image through a portion of the modified GoogLeNet architecture. After this conversion, we get the feature maps with the size 14x14x1024. Then, two convolutions are applied. After the second convolution, the dimension decreases to 7x7x1024. Then, another convolution is performed. The result is twice used in a fully connected layer, changing to a dimension of 1470x1 and is transformed into a tensor of 7x7x30. The obtained tensor is subjected to a detection procedure, at the output of which a resultant detection is obtained. The tensor is a 7x7 mesh display in the image. 30 values carry information about the cell: 10 values describe two possible frames; 20 values show the relation to each of the 20 available classes. All this information is filtered, the filtered data is displayed.

YOLO may be a refreshingly straightforward and effective model for visual object detection. Firstly, YOLO as an easy convolutional neural network simultaneously predicts multiple bounding boxes and sophistication probabilities. It initially is trained supported full images and therefore the performance is optimized. Secondly YOLO is extremely fast which may achieve quite twice of the mean average precision (mAP) of other real-time systems.

2.2.2.3.1 YOLO Loss Function

YOLO uses one's single loss function for both bounding box and the classification of the object. The loss function is:

$$\begin{aligned}
 LOSS &= \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 & (1) \\
 &+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 & (2) \\
 &+ \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 & (3) \\
 &+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 & (4) \\
 &+ \sum_{i=0}^{S^2} \mathbf{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 & (5)
 \end{aligned}$$

Figure 9 Loss function

The loss function can be parsed into 5 parts, where parts (1) and (2) are focusing on the loss of the bounding box coordinates, parts (3) and (4) are penalizing the differences in confidence of having an object in the grid and part (5) is penalizing for the difference in class probability. It is interesting to note that the loss function for the bounding box size is based on the square root of the dimensions. This is used to address that the small deviations in larger bounding boxes should incur less of a penalty than in smaller bounding boxes.

2.2.2.3.2 Yolo Versions

In 2020, three YOLO versions had been released, including YOLOv4, YOLOv5, and PP-YOLO. While YOLOv4 was released, it had been considered the fastest and most accurate real-time detection model. It inherits the Darknet and has obtained a definite AP value (43.5%) on COCO dataset while achieved a quick detection speed on Tesla V100. Compared with YOLOv3, the AP and FPS are effectively improved.

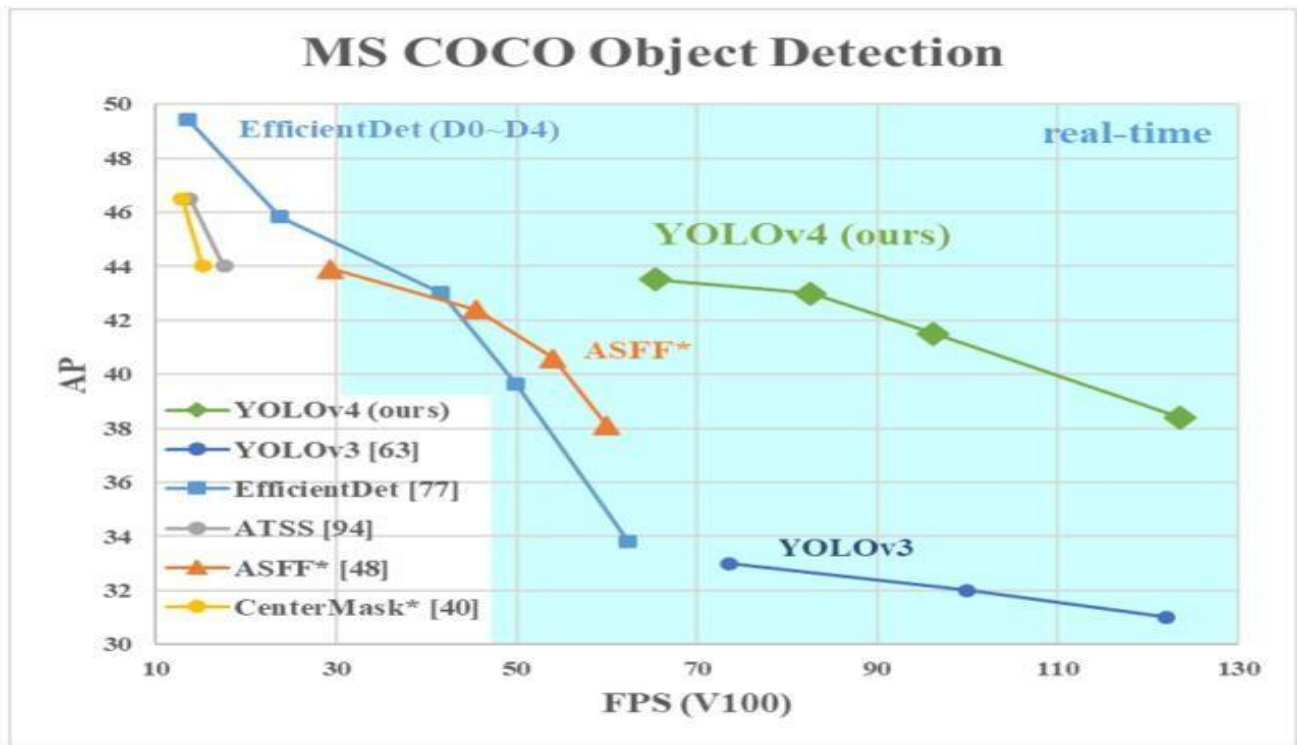


Figure 10 MS COCO Object Detection

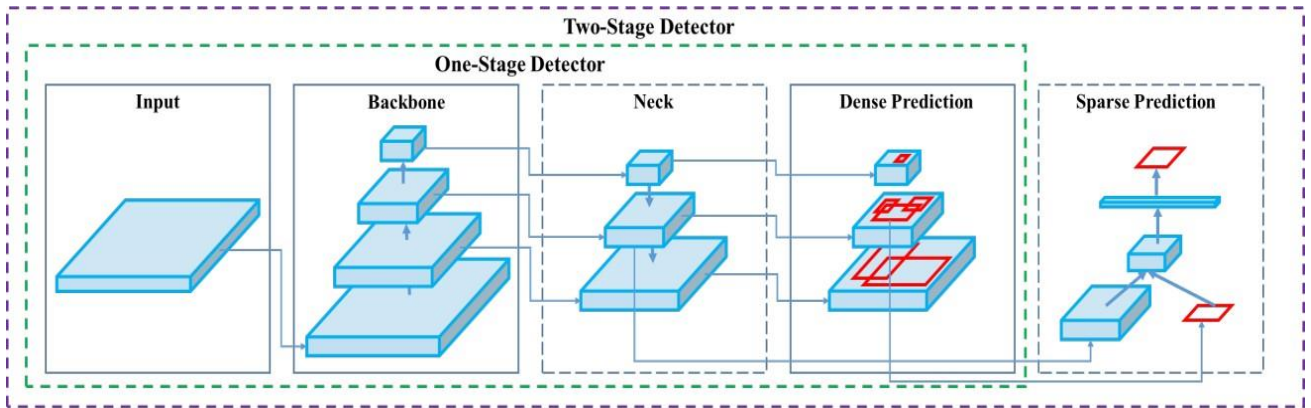


Figure 11 Two Stage detector

2.3 Hardware Selection

As mentioned above, the aim of this project was to develop a perception stack of a autonomous vehicle using deep learning and computer vision. We have developed a large number of deep learning, image processing and computer vision algorithms which need to be implemented real time for the intelligent movement of vehicle. There is no use of any algorithms no matter how much accuracy they gave or how much efficient they are if they cannot be implemented real-time on a video input coming from camera.

For real time implementation we need powerful hardware device with high computation power.

Several hardware devices were considered on the basis of the following factors:

- CPU and GPU specifications and functionalities
- Support for AI and deep learning frameworks
- Power
- Cost

These factors narrowed down the scope of hardware to choose our product from. as per the GPU and CPU functionalities as well as compatibility with IoT, AI and deep learning frameworks, NVIDIA's available range of

products are:

1. Jetson Nano
2. Jetson Xavier NX
3. Jetson AGX Xavier
4. Jetson TX2

Raspberry Pi

1. Raspberry Pi 4- 4GB
2. Raspberry Pi 4 – 8GB

2.3.1 Device Chosen

After comparing the specifications of all the available devices, it was found out that the NVIDIA JETSON AGX XAVIER best suits our requirements since our project is more towards Deep learning having a good GPU was the prime requirement and a hardware that can perform complex computation in real time. Attributed to the analysis and study, the NVIDIA JETSON AGX XAVIER was chosen.

2.3.2 NVIDIA JETSON XAVIER

AGX Xavier is ideal for deploying advanced AI and computer vision to the edge, enabling robotic platforms in the field with workstation-level performance and the ability to operate fully autonomously without relying on human intervention and cloud connectivity. Intelligent machines powered by Jetson AGX Xavier have the freedom to interact and navigate safely in their environments, unencumbered by complex terrain and dynamic obstacles, accomplishing real-world tasks with complete autonomy. Jetson AGX Xavier's high-performance can handle visual odometry, sensor fusion, localization and mapping, obstacle detection, and path planning algorithms critical to next-generation robots.

2.3.2.1 *Introduction*

It's an AI computer for autonomous machines, delivering the performance of a GPU workstation in an embedded module under 30W. Jetson AGX Xavier is designed for robots, drones, and other autonomous machines. With the NVIDIA Jetson AGX Xavier developer kit, you can easily create and deploy end-to-end AI robotics applications for manufacturing, delivery, retail, agriculture, and more.

Supported by NVIDIA JetPack and DeepStream SDKs, as well as CUDA®, cuDNN, and TensorRT software libraries, the kit provides all the tools you need to get started right away. And because it's powered by the new NVIDIA

Xavier processor, you now have more than 20X the performance and 10X the energy efficiency of its predecessor, the NVIDIA Jetson TX2.

2.3.2.2 Specifications

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 HEVC/(2x) 4Kp60 12-Bit Support
Size	105 mm x 105 mm x 65 mm
Deployment	Module (Jetson AGX Xavier)



Figure 13 NVIDIA JETSON AGX XAVIER

2.3.2.3 PORTS AND CONNECTIONS

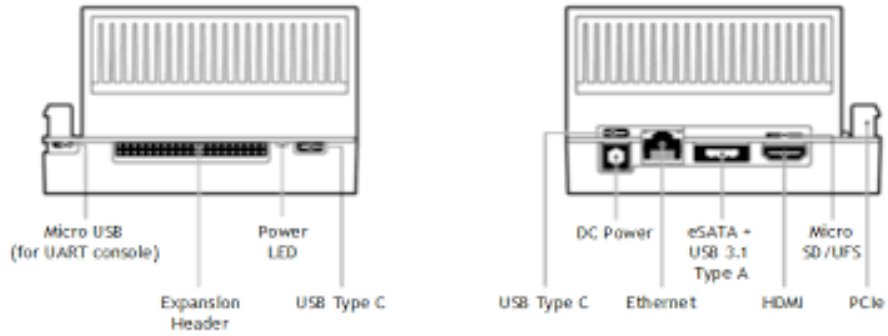
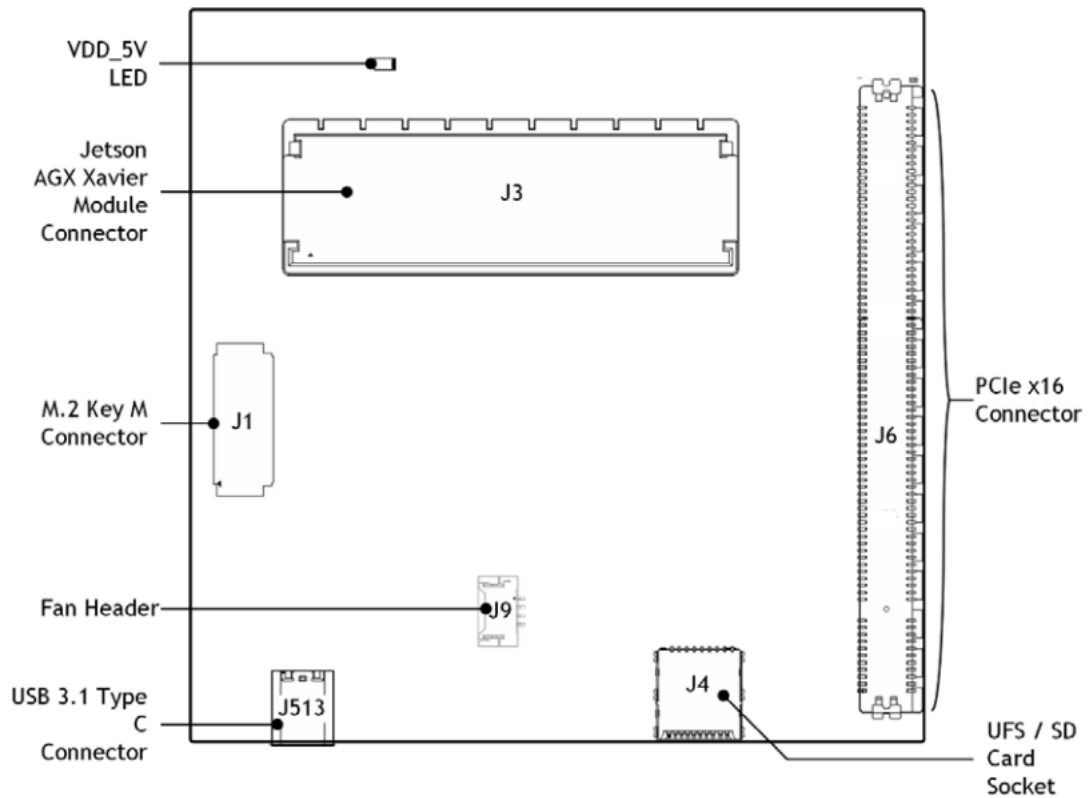
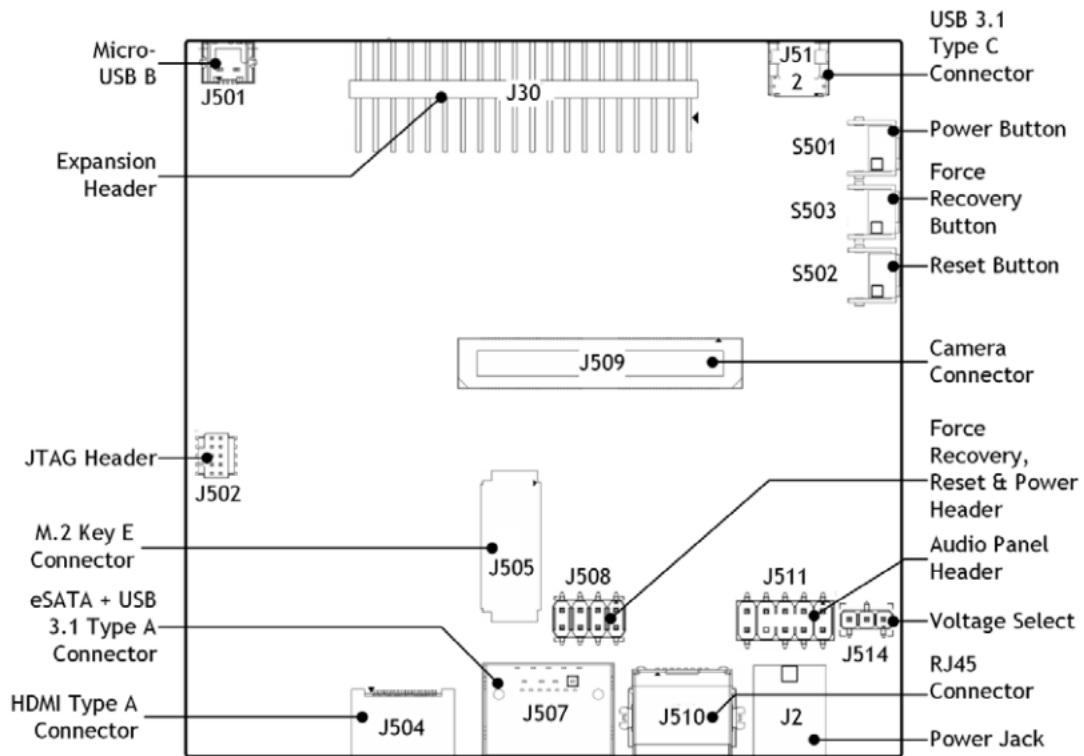


Figure 14 FRONT AND REAR VIEW

Top view of developer kit carrier board



Bottom view of developer kit carrier board



Interface Details

This list highlights some of the Jetson AGX Xavier Developer Kit carrier board interfaces. See the Jetson AGX Xavier Developer Kit Carrier Board Specification for comprehensive information:

- [J1] M.2 Key M connector for high speed NVMe storage.
 - To reach it, you must detach the combined module and thermal solution.
- [J2] Power available for peripherals is limited to power supply capability (65W from included power supply) minus developer kit system power usage (maximum of 30W in default configuration).
- [J4] Slot accepts either an SD Card or a UFS card.
- [J6] PCIe x16 connector routes to a x8 PCIe 4.0 controller.
- [J6] This connector is also where the lanes are connected for SLVS cameras.
- [J501] Micro-USB connector provides access to the UART console.



For example, you can access the serial console of the developer kit from a terminal emulator on a computer connected to this micro-USB port.
- [J504] HDMI 2.0.
- [J505] M.2 Key E connector can be used for wireless networking cards, and includes interfaces for PCIe (x1), USB 2.0, UART, I2S & I2C.

- [J507] Hybrid connector can be used for either eSATA or USB3 Type-A. The eSATA connector can supply 5V.
- [J509] Camera connector supports up to six directly connected cameras via CSI-2, or up to 16 cameras via the virtual channel feature of CSI-2.
- [J512, J513] USB Type-C connectors.
 - J512 can be used to flash the developer kit.
 - Either connector can be used to power the developer kit from USB Type-C power supplies listed in Jetson AGX Xavier Supported Component List.
 - Both connectors support DisplayPort, so you can run three displays at once byusing these plus the HDMI adapter.

2.3.3 Cameras

We have identified, evaluated, and narrowed down all components required in perception, to be utilized for development of autonomous perception stack. We have selected the cameras appropriate for safe and efficient operation with consideration to eliminate all blind spots for the vehicle to be aware of its surroundings. An arrangement of six cameras will be used to model the prototype vehicle, which include Stereo cameras for depth perception, fish-eye cameras for wide field of view, and high-speed cameras for object detection, and semantic segmentation. The details for selected cameras are provided below:

Table 30. Description of Cameras

Camera Name	Camera Placement	View	Camera Model
MYNT Eye S	Front and Back Stereo Camera	122 deg	
Global Shutter High Speed 120fps CS Mount Varifocal 5-50mm UVC Plug Play Driverless USB Camera with Mini Case	Front 5-50mm Lens: CS Mount Varifocal High-FPS Camera	80-100 deg	

The Stereo Camera provides accurate depth sensing with a flexible range between 0.5 to 18 meters. It has optimized performance in normal light conditions or low light conditions and precision with a wide field of view. The Fish-Eye lens camera covers the entire side view and

supports the other cameras, leaving very little blind spot. The full-scale arrangement of cameras and resulting field of view is shown:

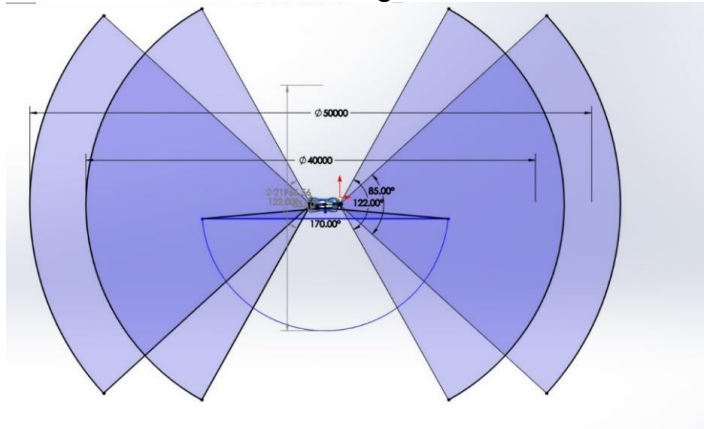


Figure 131. Arrangement of Cameras (Full-scale)

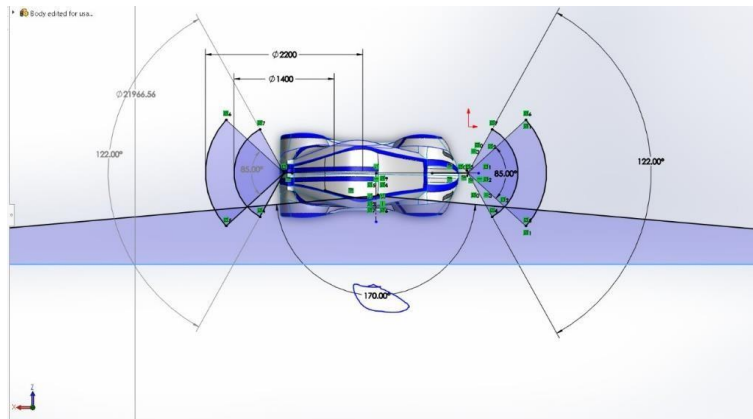


Figure 132. Arrangement of Cameras (zoomed in)

The stereo camera is coupled with a six axis IMU combined with frame synchronization which provide accuracy at less than one millisecond. Complete package with SDK is simple to integrate providing easy development and quick integration with the depth data created through the EYE S sensor.

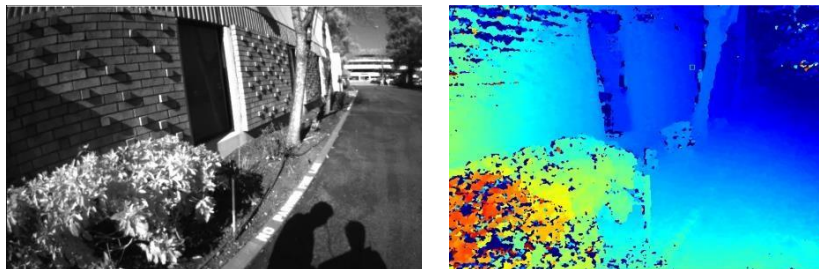


Figure 133. Benchmark Performance of MYNT Eye S camera (depth sensing)

CHAPTER 3: Image Processing

3.1 Pre-Processing

The images cannot be use directly by the CNN. With intelligent use of pre-processing techniques over images collected, it can benefit us and easily solve the problem. Image preprocessing is just like normalizing in the mathematical data set which is one of main steps required for any type of feature description methods. Raw images need to be directly enhanced for the use of training purposes using the computer vision techniques.

3.1.1 *Color conversion of image*

An image consists of three levels of a single image. The three levels are based upon the three primary colors Red, Blue and Green. In processing an image, one must cater for all the three levels individually. This makes the processing more difficult. All the three levels have range of 256 shades having additive same level color. Grayscale image contains the intensity information of an image. It has different shades of gray in it that range from 0-255. The feeblest intensity is of black while that of white is the strongest. It has only one level, so the image is easy for the processing as only 2D image matrix is to be processed instead of 3D.

For every value of a pixel there are 3 channels in an image having color and intensity information. The corresponding Grayscale pixel is found out by the pixel value for all the 3 channels and the formula given for the conversion to grayscale intensity (I) value is,

$$I = 0.299R + 0.587G + 0.144B$$

The OpenCV imports an image in BGR format but as we want to convert it into RGB so we have to use `cv2.Color_BGR2RGB ()`, than it can convert the image from RGB to gray directly from its' inbuilt function `cv2.Color_RGB2GRAY ()` that will give the input image as a gray image. This will help the training model for

efficient predictions. OpenCV has many features that are oriented basically for image processing purposes. Instead of writing a code for every pixel they have made it easier by their in-built function. In the same way one may re-convert it to RGB image.

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

3.1.2 *Equalizing an Image*

Histogram Equalization, as the name suggests, stretches the histogram to fill the dynamic range and at the same time tries to keep the histogram uniform.

By doing this, the resultant image will have an appearance of high contrast and exhibits a large variety of grey tones. Hence to standardize the lighting and improve contrasts we equalize our grayscale images.

```
img = cv2.equalizeHist(img)
```

3.1.2 *Reshaping Images*

After images are equalized, we reshape our images using built in function of Numpy.

```
img = cv2.equalizeHist(img)
```

3.1.2 *Data Augmentation*

Next, we augment our images to make them more generic. Data Augmentation is a technique of creating new data from existing data by applying some transformations such as flips, rotate at a various angle, shifts, zooms and many more. Training the neural network on more data leads to achieving higher accuracy. In real-world problem, we may have limited data. Therefore, data augmentation is often used to increase train dataset. We have used ImageDataGenerator() class function to use a range of transformations.

CHAPTER 4: Camera Calibration

4.1 Camera Geometry and Image Formation

Stereo Vision is one of the passive ways to 3D reconstruct a scene, which uses two simultaneous images of a same scene, with cameras in a displaced position with respect to each other. The basic intuition of the technique is such that similar objects in both images are displaced due to camera displacement, the displacement of the objects is inversely proportional to the distance of the object from the scene. Disparity of both objects in the scene is used for 3D re-projection.

$$d \propto \frac{1}{D}$$

Where d is the distance of the object, and D is the disparity of the object.

To understand the stereo reconstruction, we will look into camera geometry, projective transformation, and stereo geometry.

4.1.1 Pinhole Camera Model

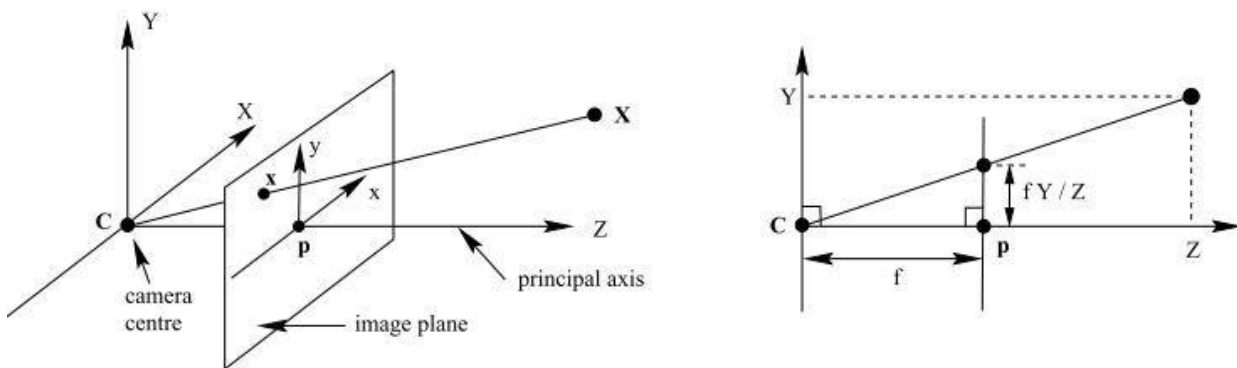


Figure 4: Pinhole Camera Model

The above image shows basic pinhole camera model, all the rays from the point in space converge towards the camera center, and the imaging plane where the image will be formed is between the point and the camera center. In this pinhole model, any point $X = (X, Y, Z)$ in space is mapped to a point in imaging plane where the ray from X to C intersects the image plane namely $x' = (x, y)$. By similarity of triangles, we can find the relation:

$$(X, Y, Z)^T \mapsto \left(\frac{fX}{Z}, \frac{fY}{Z} \right)^T$$

The center at which all rays intersect is also known as optical center. The line from the optical center perpendicular to the image plane is called principal axis and the point where the principal axis intersects the image plane is called principal point.

4.1.2 Central Projection as Linear Mapping using Homogeneous Co-ordinates

In projective geometry, we often work with homogenous coordinates, which is a way to represent vectors with one additional dimension. If we have a point $X = (x, y, z)$ in Cartesian co-ordinate system then the same point using homogenous coordinates will be

$X = (x, y, z, 1)$ or the point $X = (X, Y, Z, W)$ in homogenous coordinates will be the point $X = (X/W, Y/W, Z/W)$ in Cartesian coordinates. Homogenous coordinates allow us to represent quantities at infinity using finite numbers, as this becomes very useful in projective geometry where we have to deal with points and lines at infinity.

Now if we use homogenous coordinates, the transformation can be linearly described as:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX \\ fY \\ Z \\ 1 \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \text{ or simply } x = PX$$

Where P is the projection matrix, x is the three-dimensional image point in homogenous coordinates and X is the four-dimensional world point in homogenous coordinates.

4.1.3 Principal Point Offset

In the first initial derivation, we assumed camera center to be aligned with image center, but there is case where there is a translation of image center with respect to camera center.

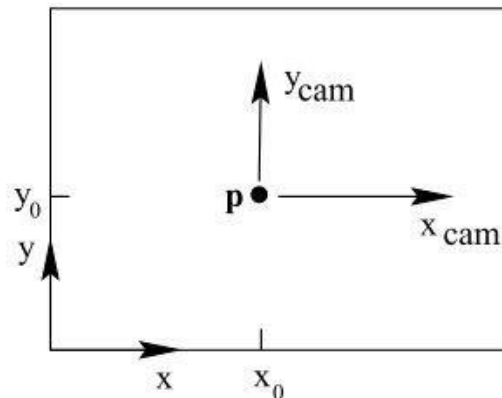


Figure 5

4.1.4 Rotation and Translation of camera

It may happen that the world co-ordinate system may not align with the camera co-ordinate system, and generally, points in 3D are represented in world co-ordinate system. Now if we have a point $X = (X_w, Y_w, Z_w)$ in world coordinate system, the same point can be represented as $X = (X_c, Y_c, Z_c)$ in camera coordinate system, the two systems are related to each other by a rotation and a translation matrix, as the camera may be displaced or its orientation may be different (yaw, pitch, roll), the transformation is given as:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = R \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

The transformation can be re-written as:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = [R | t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

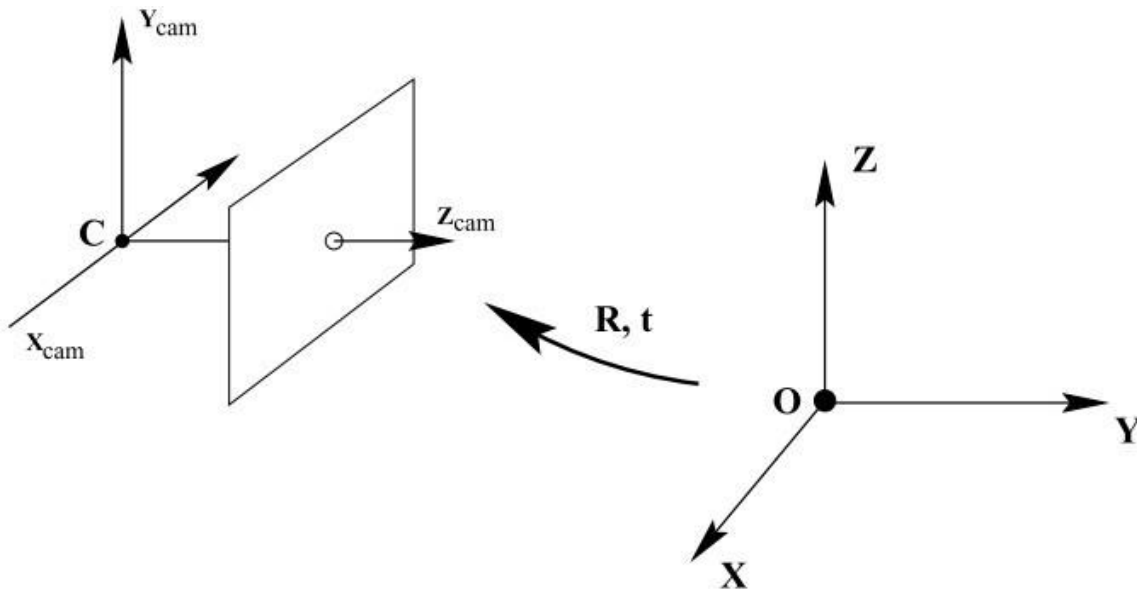


Figure 6: Transformation between world coordinate system and camera coordinate system

The complete camera transformation of a 3D point to an image point can be written as:

$$x = K[R | t]X$$

4.2 Camera Distortions

The pinhole camera model is an ideal camera model, but in real life, the lens captures all the light rays, and due to the aperture of lens, there arise several distortions, two significant distortions that are relevant to our project are discussed below:

4.2.1 Radial Distortions:

This type of distortion arises due to bending of light rays at varied angles around the center and edges of the lens, thus giving the effect that straight lines in an image are no more parallel, but rather curved. Such distortions should be addressed before processing images for stereo reconstruction.

4.2.2 Tangential Distortions:

Such distortion arises when the camera lens is not properly aligned with respect to image sensor, thus giving a tilted or stretched effect, objects at the same distance seems to appear at varied distance.

4.3 Camera calibration methodology

As in 3D reconstruction, the first step is to know the perspective projection (camera projection). The purpose of camera calibration is to estimate the camera internal matrix, camera external matrix is different for each camera setting, thus the goal of single camera calibration is to estimate camera internal matrix, it also allows us to estimate coefficients for distortions that were discussed in chapter 3, the distortion coefficients will be later used in the pipeline to undistort the images. The method OpenCV employ and we used to calibrate camera is to capture pictures of a known pattern or object whose dimensions are known and such as CHESSBOARD pattern that will be used. We can arbitrarily assign 3D coordinates to the corners of CHESSBOARD pattern and also easily locate the pixel location of those corners in the images. Hence, we have 3D points and corresponding image points available, thus we estimate our camera parameters.

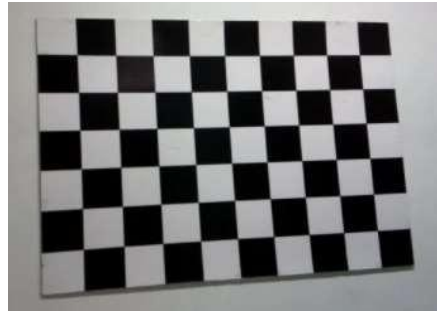


Figure 14: Chessboard Pattern defining 3D world coordinates

We define our 3D world coordinate system by arbitrarily assigning origin to the top left of the CHESSBOARD, and since the CHESSBOARD is planar hence for every point on CHESSBOARD, we arbitrarily assign $Z=0$, and we can easily assign X , Y coordinates to the corners as they are all equally spaced. The CHESSBOARD pattern is widely used because the corners have high gradient in both directions and can easily be localized with high accuracy.



Figure 15: Detected Corners in CHESSBOARD pattern

The next step in camera calibration is to take several pictures of the CHESSBOARD pattern at varied angles, as the algorithm follows iterative approach to estimate camera parameters, at least around 11 images are required for optimum calibration. For every input image, corner points are detected which are the image points and the corresponding object points are known, we get an estimate of internal matrix K , and a relative rotation and translation matrix for every images. As discussed in the previous chapter that there are some distortions due to the aperture of the lens, the calibration also estimates parameters for distortions.

CHAPTER 5:

Lane Detection for Autonomous steering

There are several approaches utilized for the detection on lane lines which are either based on computer vision approach or deep learning-based approach.

5.1 Computer Vision Approach

5.1.1 Introduction

Lane detection and tracking is one of integral aspects for the operation of a vehicle within a constrained space. The lanes are useful for the guidance of both humans and the autonomous vehicle to drive on the road. The control system effectiveness is directly influenced by the efficiency of the lane tracking algorithm, along with the execution time. The computer vision-based approach is effective in constrained environment, with good lane visibility and predetermination of the illumination conditions to design appropriate thresholding parameters. This approach is fast in terms of execution time and relied primarily on vanishing point detection using canny edge detectors and Hough transforms, or perspective detectors which project the road in a birds-eye-view form and quadratic equations, or splines are fit to these lines on the image pixels to determine curvature of the road and a measure of offset from the center of the road. The initial step is the camera calibration, which is achieved using a checkboard grid of size 6x9, which is printed on an A4 sheet, and the camera matrices are determined. A calib file is created from these observations which is used to undistort any incoming camera images. The calibration process is performed using MATLAB camera calibration toolbox with multiple images of the checkboard takes as dataset.

5.1.1 Used methods

- **Perspective Transform**

The next step in the process is perspective transform. In this project, we have the camera at the car that has the front-view perspective. This perspective has provides a lot of problem during the lane detection, First lane detection with front-view perspective bring the lots of error to our vision and we see the all lanes in the images the converge to one point at the end of the roads; this is cause of our vision system that try to transform a 3D space into a 2D space so this error of view decrease our performance in the detection of the lane and cause of error in our detection. Therefore, with implementing the perspective transform, lanes of the image will be seen parallel, and we do not have any convergence of the road in any parts of the lane. Furthermore, perspective transform helps us to concentrate on the region of interest instead of whole the image. In transform perspective we immigrate our vision from front view to the eye-bird view, so this perspective helps us to delete

unnecessary part of the image and focus on the lane region. Hence, perspective transform reduces our error and reprocess step for preparing the image for analyzing. For implementing this technique, we need to choose the 4 points in the output of the transform region. by doing this our input point will be warped to the new coordination at the output, for instance, in straight line we choose two points on each lane as input of image and the output will be imagine where image would be parallel; At the end with inverse perspective transform we draw the lane on the original image or video frames. After perspective transform, we should binarize our image and masking the interested region to extract both yellow and white lane, for reaching this purpose we convert our color space from RGB to the HSL to have better performance in distinguishing the lanes from other regions. Afterward, we are masking the images for yellow color and combine it with the masking of white lane and combine both of this with these with our extractors of edges to have robust lane at the end we will find our lanes based on the peaks of the histogram.

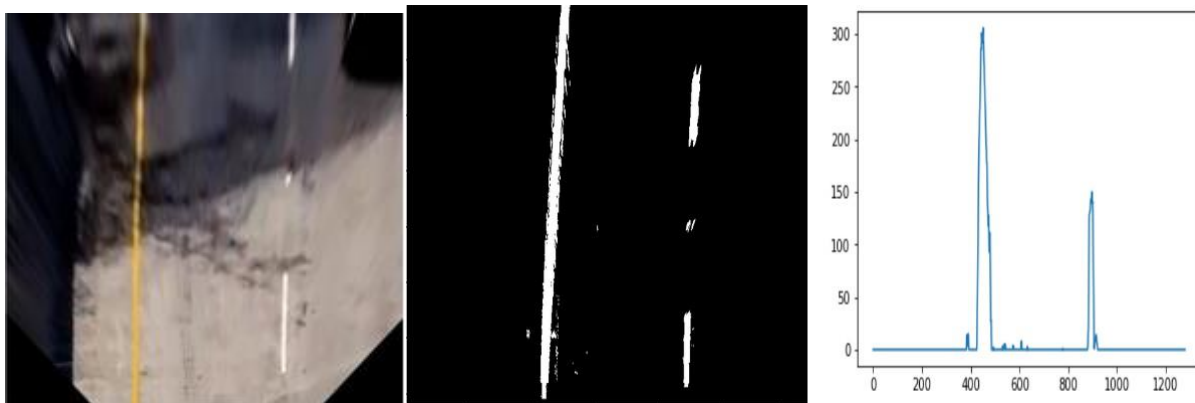


Fig.. The results of image after implementing the perspective transform, edge detect results and histogram

- **Polynomial fitting**

The next step in lane detection after perspective transform is the polynomial fitting, we should fit second order polynomial for both sides of the roads and for reaching this purpose we should follow these steps. The first and important thing is calculating the bottom half of the image and partitioning the image into several horizontal slices, this slice has the performance like the searching method around their given scopes. For finding the interested object in the images we start from the bottom slice and find the pixels that is has similar feature to the lane of the road, specially we are finding the region has the whitest pixel in horizontal coordination. Afterwards we iterate this step vertically to segments of the image in vertical sides, after this traverse, in both x and y direction for whole of the slice windows, we are fitting these points with the polynomial functions. For implementing this scenario to the video, we have the temporal correlation between video, so in this condition if our proposed algorithm cannot find the lane features in specific frame in the video, it easily skips that frame and postpone it to the next frames; Therefore, these techniques can improve the computation time of the process. hence, to have algorithmic perspective from the method we explain it as follow, Calculate a histogram of the bottom half of the image,

Partition the image into 9 horizontal slices Starting from the bottom slice in the image, this slice size is 200 pixel wide window around the left peak and right peak of the histogram we repeat this way up to the horizontal window slices to find pixels that are similar to be part of the left and right lanes.

The results for lane detection in our environment are shown below:

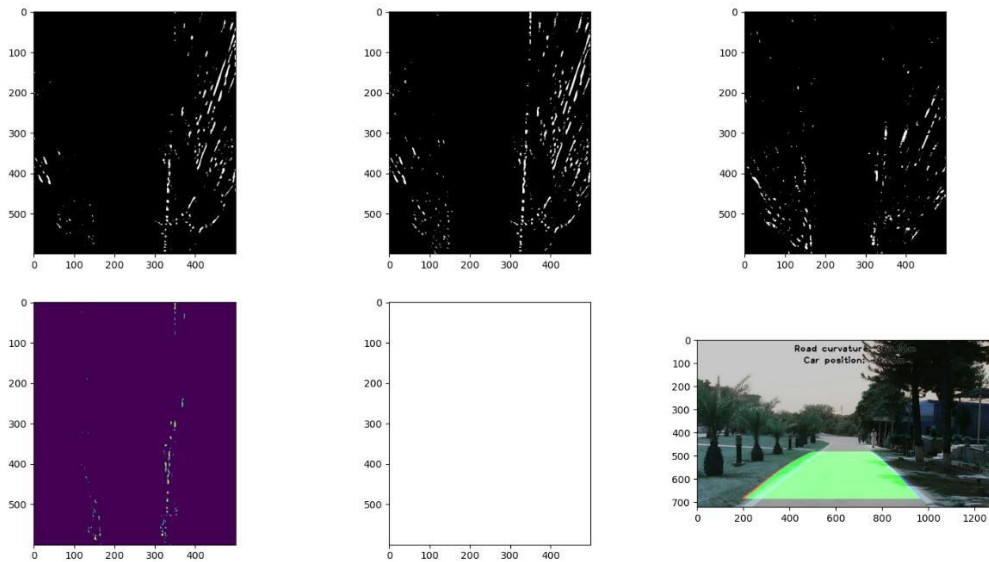


Figure 151. Lane Detection Process and Perspective transform

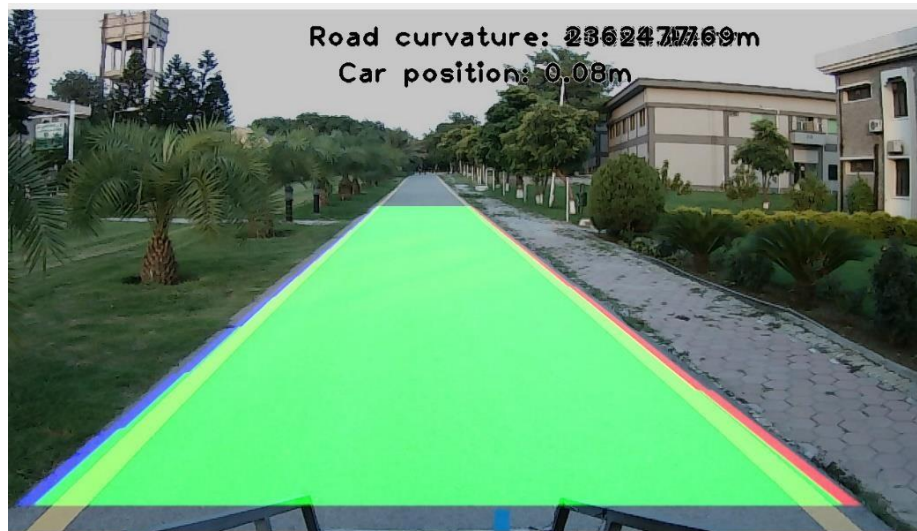


Figure: Results of lane detection



Figure: Results on Lahore Canal Road

5.2 Deep Learning Approach

The deep learning-based approach is an efficient tool for detection of multi-lane lines as opposed to only single lane lines detected by the computer vision algorithms. Deep learning models are trained on several readily available datasets such as CUlane and the benchmark is tested for efficient performance for detection of lane lines.

This approach has the ability to generalize on the data – adapt to newly encounter lane markings, and invariance to the scaling and rotation of lane, which could be present anywhere on the image, and predicts the lane boundary equations for multi-lane tracking. However, this approach is computationally expensive and could lead to higher utilization of memory due to large size of models, rendering the effectiveness to be degraded due to large computational loads, and compromised run-time performance. An approach proposed by Z. Qin et. in the paper “Ultra-Fast Structure-aware Deep Lane Detection” proposes a highly efficient real-time implementation on deep learning approach which has been implemented using Tensor-RT allowing effective utilization of resources and overall good results on test datasets.



Figure: Results of lane detection

CHAPTER 6 :

Traffic Signs & Light Detections

6.1 Introduction

The detection and recognition of traffic lights is one of the most important components of any autonomous vehicle. Outdoor perception is a major difficulty for driver-assisted and autonomous vehicles, as the machine cannot recognize traffic signals, road signs, obstructions, and other objects in the direction of motion unless it is taught to do so. Autonomous vehicles must be able to detect traffic signals and recognize their present condition, whereas humans can quickly recognize the relevant traffic signals. The detection and recognition of traffic lights must be integrated with the autonomous car's CPU (which controls the vehicle), resulting in the resolution of the traffic signal alignment issue. Obstacle detection also necessitates the use of a solution. Machine learning's Open CV2 module is effective in resolving traffic-related issues.

6.2 Traffic Sign Recognition Using Yolo V4

1. YOLO V4

For traffic sign detection, a Yolo based localization algorithm was used and a dark net based classification algorithm was implemented. The weights of the traffic sign classification were trained on Google Colab, with test data to validate the performance of the model. Turkish traffic signs dataset was used for train purposes.

2. Why we went for YOLO V4.

- [YOLO IS A REGRESSION BASED ALGORITHM](#)

It will predict the classes and bounding boxes for the entire image at once. This makes detection faster than classification algorithms. One of the BEST regression-based algorithms is YOLO (“You Only Look Once“)

- [VERY FAST](#)

It is an efficient and powerful object detection model that enables anyone with a GPU to train a super fast and accurate object detector. Light and faster version: YOLO is having a smaller architecture version called Tiny-YOLO which can work at higher framerate (155

frames per sec) with less accuracy compared to the actual model.

- **THE YOLO OBJECT DETECTION IS FREE AND OPENSOURCE**

YOLO ALGORITHM IS POPULAR DUE TO ITS REAL-TIME OBJECT DETECTION CAPABILITY

The network understands a generalized object representation making the real- world image prediction fairly accurate.

- **COMPARED WITH THE PREVIOUS YOLOV3, YOLOV4 HAS THE FOLLOWING ADVANTAGES:**

It is an efficient and powerful object detection model that enables anyone with a 1080 Ti or 2080 Ti GPU to train a super fast and accurate object detector. The influence of state-of-the-art “Bag-of-Freebies” and “Bag-of-Specials” object detection methods during detector training has been verified. The modified state-of-the-art methods, including CBN (Cross-iteration batch normalization), PAN (Path aggregation network), etc., are now more efficient and suitable for single GPU training.

3. Dataset

We used TTSDB (Turkish Traffic Sign Detection Benchmark) preprocessed it bring it into YOLO format. We uploaded the prepared dataset in our directory and then downloaded it for training.

4. Training with Google Collab

For training YOLO V4 for traffic sign recognition we again went for GOOGLE COLAB which is a Jupiter notebook environment that runs completely on a cloud.

5. Setting up DARKNET Environment

There are very few implementations of the YOLO algorithm that exists on the web. The Darknet is one such open-source neural network framework written in C and CUDA and

serves as the basis of YOLO. It is fast, easy to install, and supports CPU and GPU computation. Darknet is used as the framework for training YOLO, meaning it sets the architecture of the network. The first author of Darknet is the author of YOLO itself (J Redmon). Darknet_for_colab is a darknet folder which was modified specifically to adapt with Colab environment (no MAKEFILE change necessary). Repository for DARKNET was cloned, downloaded, and compiled.

```
# download and compile darknet_for_colab
!git clone https://github.com/quangnhat185/darknet_for_colab.git
%cd darknet_for_colab
!make
!chmod +x ./darknet

Cloning into 'darknet_for_colab'...
remote: Enumerating objects: 1057, done.
remote: Counting objects: 100% (1057/1057), done.
remote: Compressing objects: 100% (845/845), done.
remote: Total 1057 (delta 219), reused 1036 (delta 202), pack-reused 0
Receiving objects: 100% (1057/1057), 3.65 MiB | 5.73 MiB/s, done.
Resolving deltas: 100% (219/219), done.
/content/darknet_for_colab
mkdir -p ./obj/
mkdir -p backup
chmod +x *.sh
```

6. Modifying YOLO V4 architecture

Taking the advantage of the direct python editing feature on Colab, we defined training parameters just by double click on yolov4_config.py and editing. For example, we set classes=4 (our traffic sign dataset has 4 classes), max_batches=8000 (number of training iterations), batch=64 (number of samples in one batch), subdivisions=16 (number of mini_batches in one batch), etc.

```

assert os.getcwd()=='/content/darknet_for_colab', 'Directory should be "/content/darknet_for_colab" instead of "{}".format(os.getcwd())

# Run python script to create our customize yolov4_custom_train.cfg
# and yolov4_custom_tes.cfg in folder /cfg
!python yolov4_setup.py

[INFO] Generating yolov4_custom_train.cfg successfully...
[INFO] Generating yolov4_custom_test.cfg successfully...

```

7. Creating YOLO V4 backup weights in Drive

```

assert os.getcwd()=='/content/darknet_for_colab', 'Directory should be "/content/darknet_for_colab" instead of "{}".format(os.getcwd())

# delete backup folder from our
!rm /content/darknet_for_colab/backup -r

# create Symlinks so we can save trained weight in our Google Drive
# create folder YOLOv4_weight/back in your Drive to store trained weights
!ln -s /content/drive/My Drive/YOLOv4_weight/backup /content/darknet_for_colab

```

6.3 Training with YOLO V4

We used yolov4_setup.py, a python script which automatically generates YOLOv4 architecture config files (yolov4_custom_train.cfg and yolov4_custom_test.cfg) based on user-input parameters in yolov4_config.py.

```

assert os.getcwd()=='/content/darknet_for_colab', 'Directory should be "/content/darknet_for_colab" instead of "{}".format(os.getcwd())

!./darknet detector train data/yolov4.data cfg/yolov4_custom_train.cfg yolov4.conv.137 -dont_show -map
#If you get CUDA out of memory adjust subdivisions above!
#adjust max batches down for shorter training above

CUDA-version: 10010 (10010), cuDNN: 7.6.5, GPU count: 1
OpenCV version: 3.2.0
Prepare additional network for mAP calculation...
 0 : compute_capability = 600, cudnn_half = 0, GPU: Tesla P100-PCIE-16GB
net.optimized_memory = 0
mini_batch = 1, batch = 16, time_steps = 1, train = 0
layer  filters  size/strd(dil)  input          output
0 conv   32      3 x 3/ 1      416 x 416 x   3 -> 416 x 416 x 32 0.299 BF
1 conv   64      3 x 3/ 2      416 x 416 x 32 -> 208 x 208 x 64 1.595 BF
2 conv   64      1 x 1/ 1      208 x 208 x 64 -> 208 x 208 x 64 0.354 BF

```

- Visualizing training results

We initially defined max_batches=8000, but both accuracy and loss from training result did not improve much after 2000 iterations as can be seen from the graph.

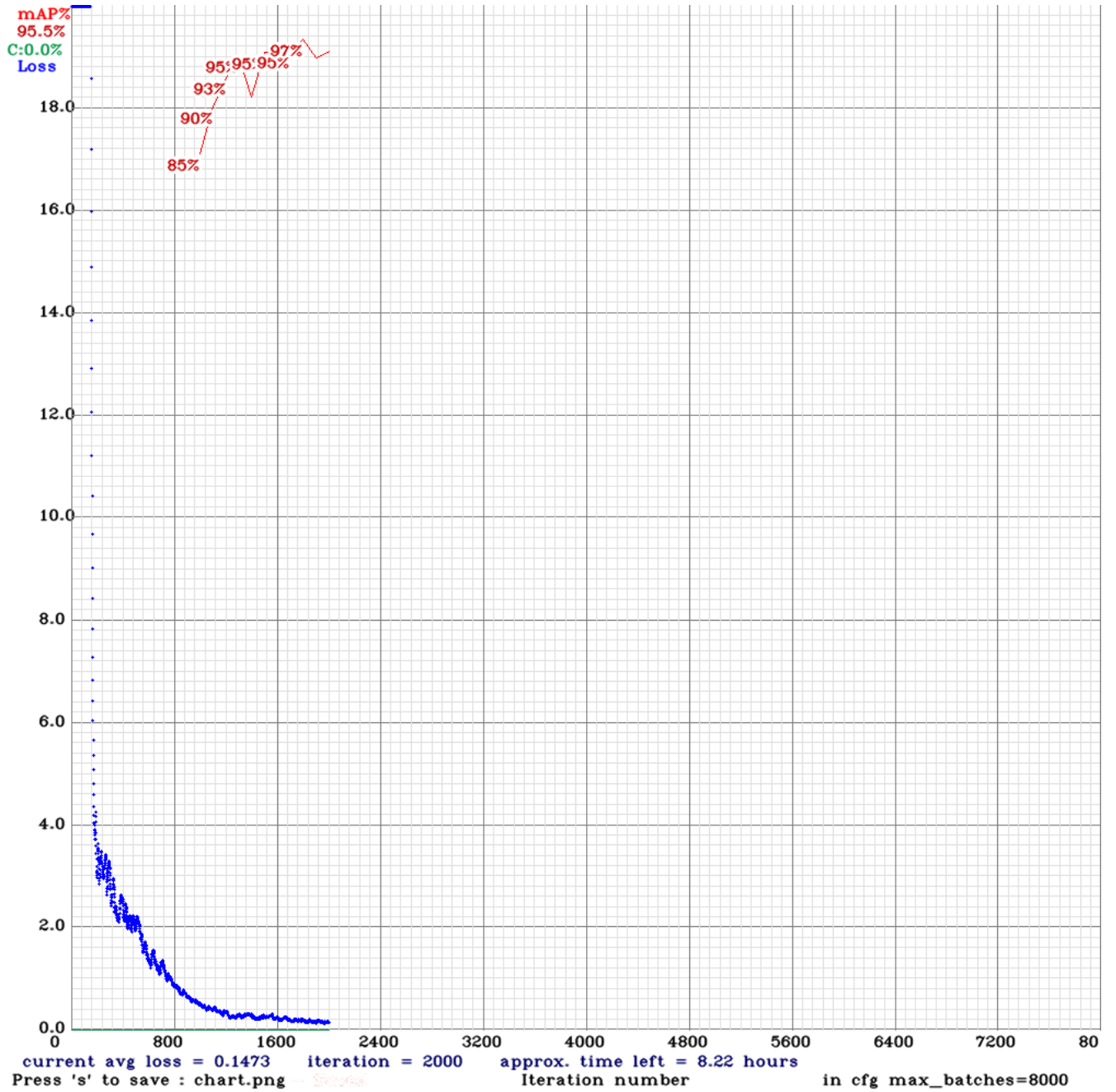


Figure 37 Accuracy map of training result



Figure: Results of traffic sign



Figure: Results of traffic light

CHAPTER 7: Object Detection

7.1 introduction

Object detection is a computer vision technique that allows us to identify and locate objects in an image or video. With this kind of identification and localization, object detection can be used to count objects in a scene and determine and track their precise locations, all while accurately labeling them.

7.2 MODES AND TYPES OF OBJECT DETECTION

Broadly speaking, object detection can be broken down into machine learning-based approaches and deep learning-based approaches.

In more traditional ML-based approaches, computer vision techniques are used to look at various features of an image, such as the color histogram or edges, to identify groups of pixels that may belong to an object. These features are then fed into a regression model that predicts the location of the object along with its label.

On the other hand, deep learning-based approaches employ convolutional neural networks (CNNs) to perform end-to-end, unsupervised object detection, in which features don't need to be defined and extracted separately. For a gentle introduction to CNNs, check out this overview.

Because deep learning methods have become the state-of-the-art approaches to object detection, these are the techniques we'll be focusing on for the purposes of this guide.

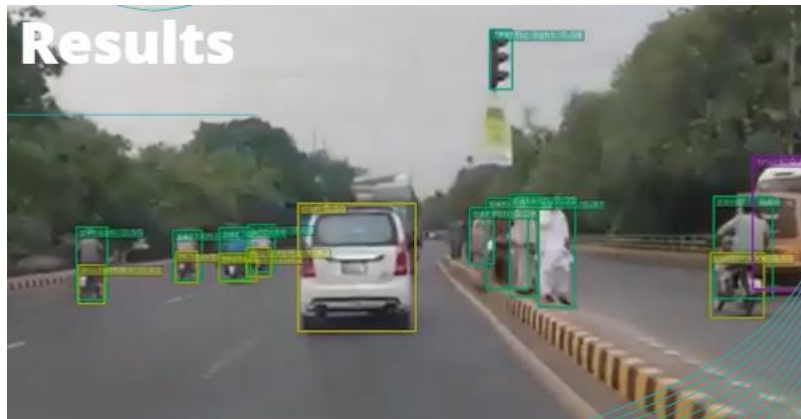
7.3 Basic working structure

Deep learning-based object detection models typically have two parts. An encoder takes an image as input and runs it through a series of blocks and layers that learn to extract statistical features used to locate and label objects. Outputs from the encoder are then passed to a decoder, which predicts bounding boxes and labels for each object.

The simplest decoder is a pure regressor. The regressor is connected to the output of the encoder and predicts the location and size of each bounding box directly. The output of the model is the X, Y coordinate pair for the object and its extent in the image. Though simple, this type of model is limited. You need to specify the number of boxes ahead of time. If your image has two dogs, but your model was only designed to detect a single object, one will go unlabeled. However, if you know the number of objects you need to predict in each image ahead of time, pure regressor-based models may be a good option.

An extension of the regressor approach is a region proposal network. In this decoder, the model proposes regions of an image where it believes an object might reside. The pixels belonging to these regions are then fed into a classification subnetwork to determine a label (or reject the proposal). It then runs the pixels containing those regions through a classification network. The benefit of this method is a more accurate, flexible model that can propose arbitrary numbers of regions that may contain a bounding box. The added accuracy, though, comes at the cost of computational efficiency.

7.4 Results



Results of object detection on canal road

CHAPTER 8: 3D Object Detection

8.1. Introduction

3D object detection is a fundamental requirement of localization of static and dynamic obstacles in an environment, and a crucial engineering problem for autonomous vehicles and mobile robots. The algorithm processes the image obtained through stereo cameras as estimates the depth of the scene by triangulation using to the pin-hole camera model as a reference. The use of binocular setup is generally much cheaper, and it is preferred for low-cost operation for autonomous vehicles. The framework that is used for the purpose of object detection is YOLOStereo3D . It is a lightweight one-stage stereo 3D detection network. To efficiently produce powerful stereo features, the pixel-wise correlation is reintroduced to construct the cost- volume, rather than concatenation of features. The inference pipeline from one-stage monocular 3D detection into stereo 3D detection is used during inference. YOLOStereo3D produces competitive results on the KITTI 3D benchmark during inference on stereo images and with an inference time of less than 0.1 seconds per frame. The neural network architecture of YOLOStereo3D is shown below:

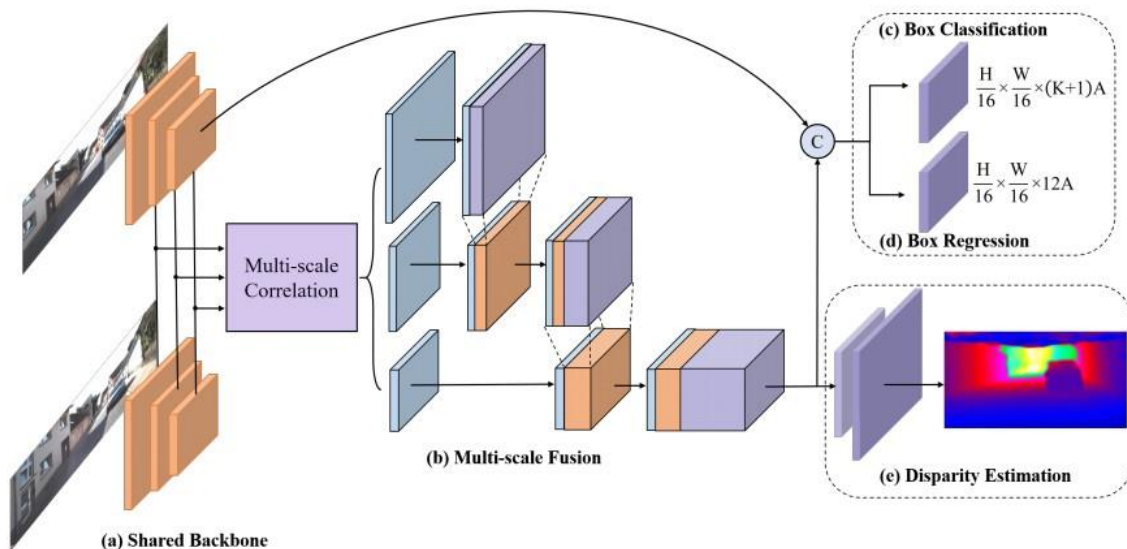


Figure 77. 3D Object Detection Convolution Neural Network Architecture

Data augmentation is useful to improve the generalization ability in deep learning applications. However, the nature of stereo 3D detection

limits the number of possible augmentation choices. Photometric distortion is concurrently applied on binocular images. This improves feature matching, and the inference pipeline is optimized with multi-scale correlation.

8.2. Training Scheme and Loss Function

During the training process the stereo feature map is fed into a decoder to predict a disparity map trained with an auxiliary loss. The auxiliary loss can regularize the training process. The network may not be guided to produce local features useful in stereo matching to fully utilize the geometric potential of binocular images, and the network could be trapped in a local minimum like that of a monocular detection network. The focal loss is applied on classification, and smoothed-L1 loss on bounding box regression. The expected distribution of disparity is computed with a hard-coded variance $\sigma = 0.5$:

8.3. Inference of Yolo3D object Detection

The inference of Yolo3DStereo Object Detection is performed on Nvidia Jetson Xavier, with an inference time of 0.3 seconds. The code is modified to create an inference node from the available test code such that the python code takes real-time feed from the camera instead of loading the images from database. The output of the inference node is shown below indicating the position of the vehicle bounded by 3D boxes. The output of the 3D object detection is passed to the behavioral planner for planning of feasible trajectories.



Figure 78. 3D Object Detection Inference

CHAPTER 9: Depth Estimation

9.1. Monocular Camera using deep learning Approach

9.1.1. INTRODUCTION

Depth sensing is essential to many robotic tasks, including mapping, localization, and obstacle avoidance. Existing depth sensors (e.g., LiDARs, structured-light sensors, etc.) are typically bulky, heavy, and have high power consumption. These limitations make them unsuitable for small robotic platforms (e.g., micro aerial and mini ground vehicles), which motivates depth estimation using a monocular camera, due to its low cost, compact size, and high energy efficiency.

9.1.2. EMPLOYED METHODOLOGY

The current state-of-the-art depth estimation algorithms rely on deep learning based methods, and while these achieve significant improvement in accuracy, they do so at the cost of increased computational complexity.

Our approach employs MobileNet as an encoder and nearest neighbor interpolation with depth wise separable convolution in the decoder. We apply state-of-the-art network pruning, NetAdapt and use the TVM compiler stack to further reduce inference runtime on a target platform. We show that our low latency network design, Fast Depth, can perform real-time depth estimation on the NVIDIA Jetson AGX Xavier operating at over 120 frames per second (fps). The Architecture diagram is given as :

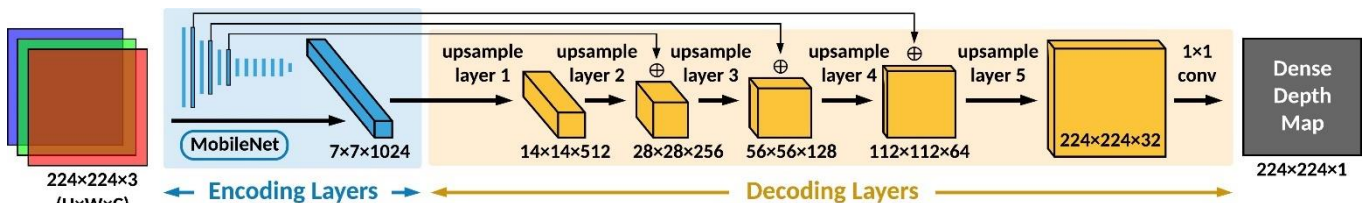


Fig. 2: Proposed network architecture. Dimensions of intermediate feature maps are given as height \times width \times # channels. Arrows from encoding layers to decoding layers denote additive (rather than concatenative) skip connections.

9.1.3. INFERENCE

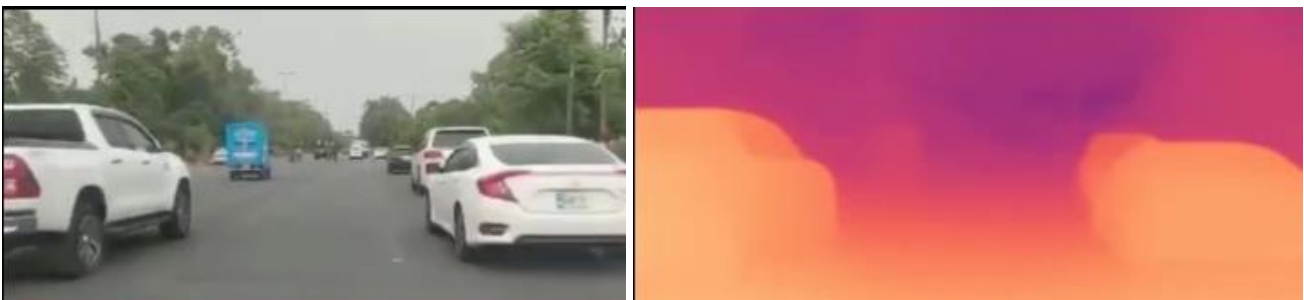
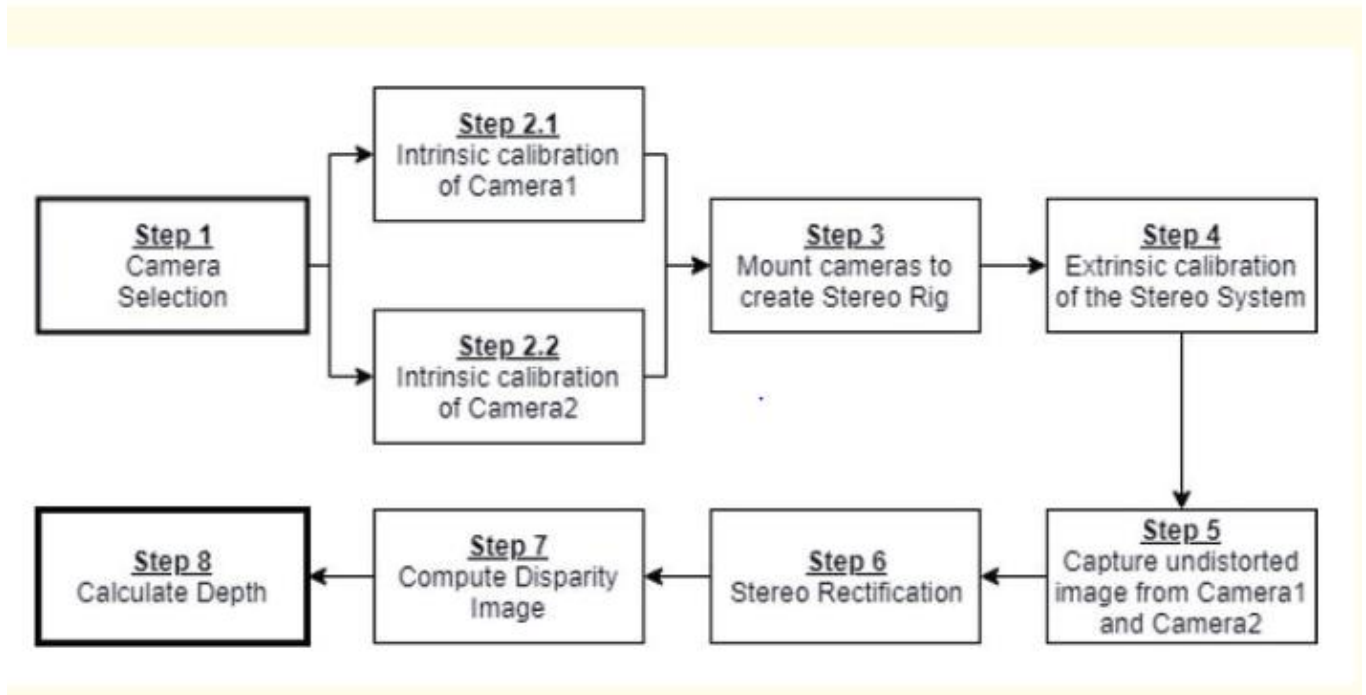


Figure: Result of Deep learning Approach

9.2. Stereo Camera using computer vision Approach

The flow diagram is given as:



After doing camera calibration as described earlier, we will do Stereo Rectification.

- *Stereo Rectification:*

there are several alternatives in OpenCV for rectification task, for example we can use stereo calibrate function to calibrate cameras and recover relative pose between cameras and then estimate fundamental matrix, through which we can get rectification maps to rectify images. But stereo calibrate method was not working well with our setups as it was not a dedicated stereo setup, hence we followed another approach, we detected key points in left and right images and based on those key points we recover the pose of two cameras relative to each other also on the basis of matched key points we can estimate essential matrix as well, when we have our R and t pose matrices and essential matrix, fundamental matrix can be recovered. SIFT descriptors were used to detect correspondence points in both images. Having all the matrices we can make rectification maps in order to rectify images, the images are passed through rectification maps, and we get our rectified images. The functions used in the rectification pipelines were `cv2.recoverPose`, `cv2.findEssentialMatrix`, `cv2.stereoRectify`, `cv2.initUndistortRectifyMaps`, `cv2.remap`.

The function parameters and their input and output can be known from

OpenCV documentation.

The rectification can destroy images if the setup is not accurately aligned, hence many trials were done for rectification as the hardware is not dedicated, and some of the rectification results will be displayed in the results heading at the end of the chapter.

- *Calculating Disparity Map*

After we have obtained our rectified images, now we are at the stage to compute disparities of the object in the two images, OpenCV provides several functions to estimate disparity out of which two are the most popular `cv2.stereoBM` and `cv2.stereoSGBM`. Now both functions apply some version of block matching algorithm, we used `cv2.stereoSGBM` for block matching which is the implementation of semi global matching algorithm. There are lot of tuning parameters for this function; it is worth to mention the description of some.

Cv2.StereoSGBM function parameters:

NumDisparities: The parameter defines disparity value's range. The range is calculated from minimum disparity to maximum disparity, the value should be the multiple of 16, increasing disparity range increases accuracy of depth map.

blockSize: Window size for block matching for stereo correspondence.

PreFilterType: Parameter to define any filter that is applied before image is processed to calculate disparity.

A point to know that is that block-matching algorithm returns a 16-bit signed single channel image with disparity values scaled by 16. Thus, to calculate the actual disparity value division by 16 is necessary. It is also necessary to understand that function does not returns depth map rather the disparity, which is the relative displacement of the object in the two scenes. The map obtained will be very noisy, because a matching is done and the map to be passed through filter in order to smooth the transition and fill the untextured regions. In order to smooth the disparity map, we used the implementation of WLS (weighted least square) filter, for disparity smoothing in OpenCV. `Cv2.createDisparityWLSFilter` takes input the actual object of `StereoSGBM`. Note the disparity we calculated is respect to the left image but for filtering using WLS filter, we need a disparity map with respect to right image as well. Hence, we first computed right disparities using `cv2.createRightMatcher`, and pass all the arguments to WLS filter object to filter the map.

- *Reprojection from depth map*

Now we can use the disparity map and the transformation of the form given below to move from disparity to depth map.

$x - x' = Bf/Z$, where x and x' are the correspondence points in the left and right image respectively, B is the base line and f is the focal length, the parameter to estimates is Z . Note we are too optimistic about all the parameters in the above equation, the baseline can be fixed but the focal length of two cameras can differ from each other in that case the equation does not hold and we have to apply the approximation using SSD in order to estimate the optimum focal length. However, we tried initial formula that gave us satisfactory results.

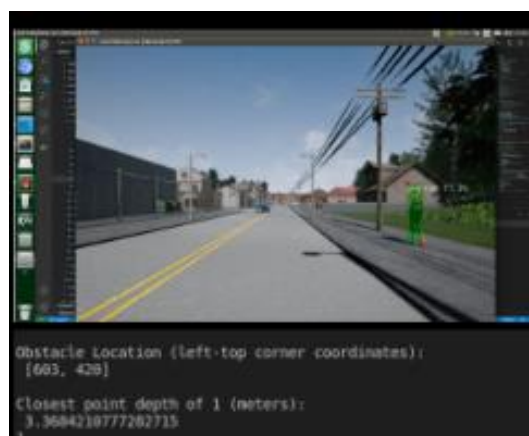
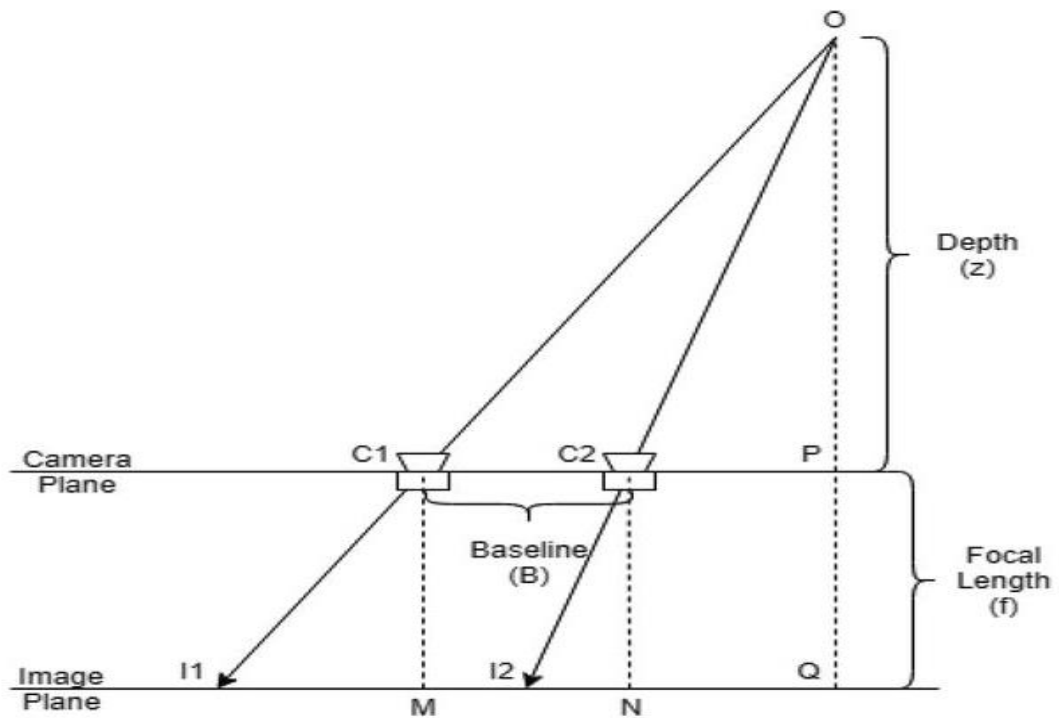


Figure: Result using stereo camera

CHAPTER 10: Hardware Optimization

10.1. Tensor Rt

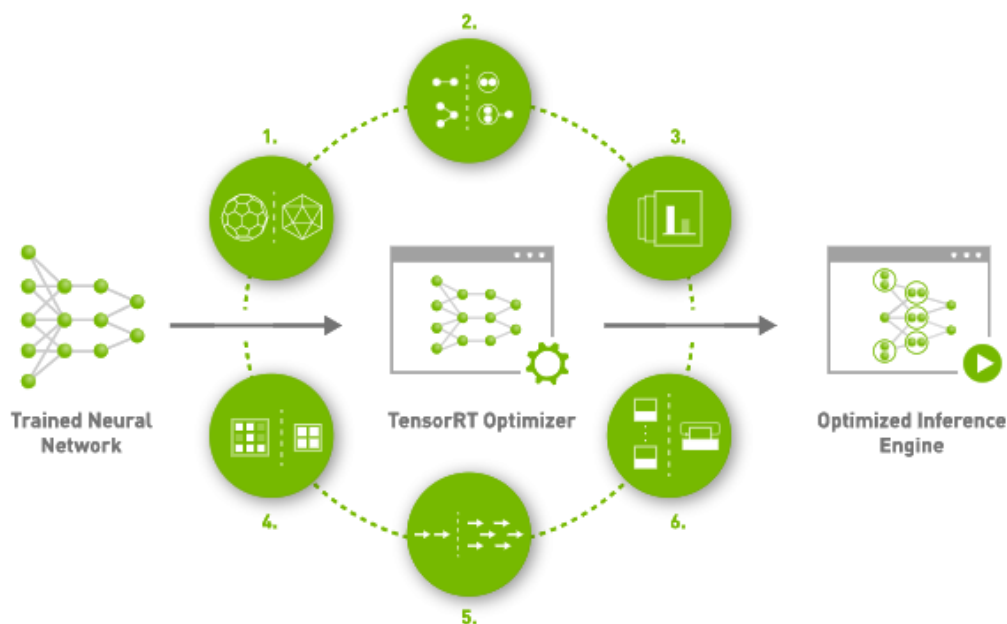
10.1.1. Overview:

One of the perpetual problems of deep neural networks is figuring out the speed of learning and optimizing it which is often hit and trial. NVIDIA TensorRT is an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for deep learning inference applications.

TensorRT-based applications perform up to 40X faster than CPU-only platforms during inference. With TensorRT, you can optimize neural network models trained in all major frameworks, calibrate for lower precision with high accuracy, and deploy to hyper-scale data centers, embedded, or automotive product platforms.

TensorRT is built on CUDA, NVIDIA's parallel programming model, and enables you to optimize inference leveraging libraries, development tools, and technologies in CUDA-X for artificial intelligence, autonomous machines, high-performance computing, and graphics. With new NVIDIA Ampere Architecture GPUs, TensorRT also leverages sparse tensor cores providing an additional performance boost.

TensorRT provides INT8 and FP16 optimizations for production deployments of deep learning inference applications such as video streaming, speech recognition, recommendation, fraud detection, and natural language processing. Reduced precision inference significantly reduces application latency, which is a requirement for many real-time services, as well as autonomous and embedded applications.



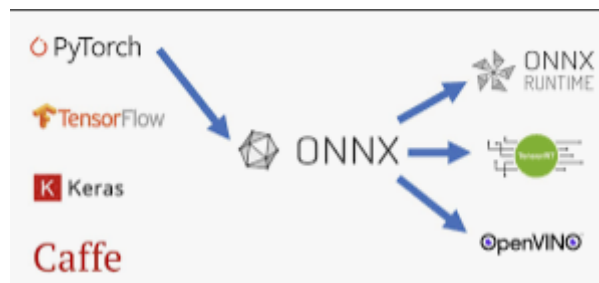
10.2. ONNX

The end result of a trained deep learning algorithm is a model file that efficiently represents the relationship between input data and output predictions. A neural network is one of the most powerful ways to generate these predictive models but can be difficult to build into production systems. Most often, these models exist in a data format such as a .pth file or an HD5 file. Oftentimes you want these models to be portable so that you can deploy them in environments that might be different than where you initially trained the model.

10.1.2. Overview

At a high level, ONNX is designed to allow framework interoperability. There are many excellent machine learning libraries in various languages—PyTorch, TensorFlow, MXNet, and Caffe are just a few that have become very popular in recent years, but there are many others as well.

The idea is that you can train a model with one tool stack and then deploy it using another for inference and prediction. To ensure this interoperability you must export your model in the model.onnx format which is serialized representation of the model in a protobuf file. Currently there is native support in ONNX for PyTorch, CNTK, MXNet, and Caffe2 but there are also converters for TensorFlow and CoreML.



10.1.3. ONNX in Practice

Let's imagine that you want to train a model to predict if a food item in your refrigerator is still good to eat. You decide to run a bunch of photos of food that is at various stages past its expiration date and pass it into a convolutional neural network (CNN) that looks at images of food and trains it to predict if the food is still edible.

Once you have trained your model, you then want to deploy it to a new iOS app so that anyone can use your pre-trained model to check their own food for safety. You initially trained your model using PyTorch but iOS expects to use CoreML to be used inside the

app. ONNX is an intermediary representation of your model that lets you easily go from one environment to the next.

Using PyTorch you would normally export your model using

```
torch.save(the_model.state_dict(), PATH)
```

Exporting to the ONNX interchange format is just one more line:

```
torch.onnx.export(model, dummy_input, 'SplitModel.proto', verbose=True)
```

Using a tool like ONNX-CoreML, you can now easily turn your pre-trained model in to a file that you can import in to XCode and integrate seamlessly with your app.

10.1.4. Conclusion

As more and more deep learning frameworks emerge and workflows become more advanced, the need for portability is more important than ever. ONNX is a powerful and open standard for preventing framework lock-in and ensuring that you the models you develop will be usable in the long run.

CHAPTER 11: Robotic Operating System (ROS)

11.1. Overview:

It is an open-source robotics middleware suite. Although ROS is not an operating system (OS) but a set of software frameworks for robot software development, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post, and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS is not a real-time operating system (RTOS). However, it is possible to integrate ROS with real-time code.

Software in the ROS Ecosystem can be separated into three groups:

- language-and platform-independent tools used for building and distributing ROS-based software.
- ROS client library implementations such as roscpp, rospy, and roslisp.
- packages containing application-related code which uses one or more ROS client libraries.

Both the language-independent tools and the main client libraries (C++, Python, and Lisp) are released under the terms of the BSD license, and as such are open-source software and free for both commercial and research use. The majority of other packages are licensed under a variety of open-source licenses. These other packages implement commonly used functionality and applications such as hardware drivers, robot models, datatypes, planning, perception, simultaneous localization and mapping, simulation tools, and other algorithms.

The main ROS client libraries are geared toward a Unix-like system, primarily because of their dependence on large collections of open-source software dependencies. For these client libraries, Ubuntu Linux is listed as "Supported" while other variants such as Fedora Linux, macOS, and Microsoft Windows are designated "experimental" and are supported by the community. The native Java ROS client library, rosjava, however, does not share these limitations and has enabled ROS-based software to be written for the Android OS. rosjava has also enabled ROS to be integrated into an officially supported MATLAB toolbox which can be used on Linux, macOS, and Microsoft Windows. A JavaScript client library, roslibjs has also been developed which enables integration of software into a ROS system via any standards-compliant web browser.

11.2. Tools :

ROS's core functionality is augmented by a variety of tools which allow developers to visualize and record data, easily navigate the ROS package structures, and create scripts automating complex configuration and setup processes. The addition of these tools greatly increases the abilities of systems using ROS by simplifying and providing solutions to a number of common robotics development problems. These tools are provided in packages like any other algorithm, but rather than providing implementations of hardware drivers or algorithms for various robotic

tasks, these packages provide task and robot-agnostic tools which come with the core of most modern ROS installations.

11.2.1. rviz

It is a three-dimensional visualizer used to visualize robots, the environments they work in, and sensor data. It is a highly configurable tool, with many different types of visualizations and plugins.

11.2.2. rosbag

It is a command line tool used to record and playback ROS message data. rosbag uses a file format called bags,[71] which log ROS messages by listening to topics and recording messages as they come in. Playing messages back from a bag is largely the same as having the original nodes which produced the data in the ROS computation graph, making bags a useful tool for recording data to be used in later development. While rosbag is a command line only tool, rqt_bag provides a GUI interface to rosbag.

11.2.3. catkin

It is the ROS build system, having replaced rosbuilt as of ROS Groovy. catkin is based on CMake, and is similarly cross-platform, open-source, and language-independent.

11.2.4. rosbash

The rosbash package provides a suite of tools which augment the functionality of the bash shell. These tools include rosls, roscd, and roscp, which replicate the functionalities of ls, cd, and cp respectively. The ROS versions of these tools allow users to use ros package names in place of the file path where the package is located. The package also adds tab-completion to most ROS utilities, and includes rosed, which edits a given file with the chosen default text editor, as well rosrn, which runs executables in ROS packages. rosbash supports the same functionalities for zsh and tcsh, to a lesser extent.

11.2.5. roslaunch

It is a tool used to launch multiple ROS nodes both locally and remotely, as well as setting parameters on the ROS parameter server. roslaunch configuration files, which are written using XML can easily automate a complex startup and configuration process into a single command. roslaunch scripts can include other roslaunch scripts, launch nodes on specific machines, and even restart processes which die during execution.



For monocular camera

```
brain@brain-desktop: ~  
brain@brain-desktop:~$ roslaunch usb_cam usb_cam-test.launch
```

YOLO traffic sign

```
brain@brain-desktop: ~  
brain@brain-desktop:~$ rosrunc yolo-trt sample
```

YOLO traffic light

```
brain@brain-desktop: ~  
brain@brain-desktop:~$ rosrunc yolo-trt signal
```

For Stereo Camera

```
brain@brain-desktop: ~  
brain@brain-desktop:~$ roslaunch mynt_eye_ros_wrapper mynteye.launch
```

Conclusion

The main task was implementation of perception stack in real time. We faced a number of issues in doing that because a large number of available algorithms work for images and does not give real time results i.e. very low fps(frames per second are obtained).So optimization of available algorithms was required for real time implementation. We have tested different algorithms with a number of parameters and choose the one best suitable to our requirements. The state of the art algorithm of YOLO was optimized using ONNX and TensorRT giving us 25 to 30 fps in object detection which was just giving 1 to 2 fps before optimization. Finally all our algorithms for object detection, traffic signs, light detection, lane detection and depth estimation working realtime.

References

www.coursera.org/learn/visual-perception-self-driving-cars
www.udacity.com/course/self-driving-car-engineer-nanodegree--nd0013
<https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit>
<https://pjreddie.com/darknet/yolo/>
www.towardsdatascience.com/the-evolution-of-deeplab-for-semantic-segmentation-95082b025571
<https://developer.nvidia.com/tensorrt>
<https://arxiv.org/abs/1903.03273>
<https://arxiv.org/abs/2103.09422>
<https://www.coursera.org/lecture/visual-perception-self-driving-cars/lesson-3-part-2-visual-depth-perception-computing-the-disparity-Q00hg>
https://en.wikipedia.org/wiki/Robot_Operating_System
<https://www.ros.org/>
<https://ieeexplore.ieee.org/document/9327478>
<https://github.com/Owen-Liuyuxuan/visualDet3D>
<https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006>
<https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit>
<https://opencv.org/>
<https://www.mynteye.com/>