

Self-Driving Car

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology in **Computer Science & Engineering**

by

Kushal Sah

18BCE2490

Rahul Kumar Karn

18BCE2492

Amit Chaudhary

18BCE2498

Under the guidance

of

Prof. Archana T

SCOPE

VIT, Vellore.



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

June, 2022

DECLARATION

We hereby declare that the thesis entitled “Self Driving Car” submitted by us, for the award of the degree of *Bachelor of Technology in Computer Science & Engineering* to VIT is a record of bonafide work carried out by us under the supervision of **Dr.Archana T.**

We further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place : Vellore

Date :

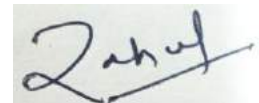
Name: Kushal Sah

Signature:




Name: Rahul Kumae Karn

Signature:



Name: Amit Chaudhary

Signature:



CERTIFICATE

The project “Self-Driving Car “ presented by:

- | | | |
|---|------------------|-----------|
| 1 | Kushal Sah | 18BCE2490 |
| 2 | Rahul Kumar Karn | 18BCE2492 |
| 3 | Amit Chaudhary | 18BCE2498 |

under the supervision of our guide and approved by the project examination committee, has been accepted by the VIT university, Vellore in partial fulfillment of the requirements for the four year degree of **B.Tech CSE**.

Place : Vellore

Date :

Signature of the Guide

Internal Examiner

External Examiner

Head of the Department

BTech.CSE

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to our supervisor Dr. Archana T for her encouragement and support throughout this whole project. We also thank her for her patience, motivation, enthusiasm and immense knowledge. Her guidance has helped us to complete this project and thesis.

Beside our advisor, we would also like to thank our seniors especially Mr. Kshitiz Chaudhary for helping us in the completion of our project.

A special thanks to VIT University for providing us such a nice environment in order to complete our project.

We would like to acknowledge our friends for their endless support throughout this project.

Last but not the least we would like to thank our parents for their moral and material assistance.

Executive Summary

Initially, tracks are deployed on surface to gather data through video streaming using webcam with help of OpenCV which is open-source computer vision library. After the collection of data, deployment of image processing such thresholding, Hough Line transformation to find the Centre point and frame point. With help of these Centre and frame point, we are able to find the curve/turn value of car and traffic sign to distinguish different classes i.e. right, left, forward and stop.

After this, we are able to collect the dataset for our model to make self-automation vehicle. Dataset contain the image and csv file which contain image location and curve value of turn. These datasets are collected manually which leads to innovate the model according to environment we provide to the car. After collection of data, OS operation are performed to extract the data and label it from location then image processing technique are applied to get better feature result from image and such technique may be augmentation, thresholding, Hough line detection and some preprocessing like gaussian blur and resizing the image were required to remove noise and equal size dimension. Thereafter, we were ready make HAAR cascade model using cascade trainer gui which performance is better on image or video.

LIST OF ABBREVIATIONS

NN	Neural Networks
CNN	Convolutional Neural Networks
DNN	Deep Neural Networks
AI	Artificial Intelligence
PWM	Pulse Width Modulation
DAPRA	Defense Advanced Projects Research Agency
ML	Machine Learning
API	Application Programming Interface
ROI	Region of Interest
DC	Direct Current
OCR	Optical Character Recognition
GPU	Graphic Processing Unit
RELU	Rectified Linear unit
PVC	Polyvinyl Chloride
IoT	Internet of Things
LIDAR	Light Detection and Ranging

SAM	Seamless Autonomous Mobility
HMI	Human Machine Interface
LCD	Liquid Crystal Display
PCB	Printed Circuit Board
AVT	Autonomous Vehicle Technology
CV	Computer Vision
RIDER	Real-time Intelligent Driving Environment Recording System
NDS	Naturalistic Driving Study
GPS	Global Positioning System
MIT	Massachusetts Institute of Technology
PC	Personal Computer

List of Figures

- Fig 1.1 – Basic model for self driving cars
- Fig 1.2 – Survey chart
- Fig 1.3 – Self-driving environment by Volkswagen
- Fig 1.4 – Prototype of Self-driving car
- Fig 1.5 – Block diagram of Project
- Fig 2.1 – High level view of data collection system
- Fig 2.2 – Training the neural network
- Fig 2.3 – Simulation of DAVE
- Fig 2.4 – Flowchart of image processing for lane detection
- Fig 2.5 – Processing First Stage
- Fig 2.6 – Processing Second Stage
- Fig 3.1 – Supervised Learning
- Fig 3.2 – Unsupervised Learning
- Fig 3.3– Biological Neural Network
- Fig 3.4 – Multi layered neural network and biological neural network
- Fig 3.5 – Deep Neural Network
- Fig 3.6 – Convolutional Neural Network
- Fig 3.7 – Image matrix multiplies kernel or filter matrix
- Fig 3.8 – 3x3 output matrix
- Fig 3.9 – Max Pooling
- Fig 3.10 – After Pooling Layer
- Fig 3.11 – Complete CNN Architecture
- Fig 4.1 – Flowchart of Hardware Working
- Fig 4.2 – DC motor
- Fig 4.3 – Batteries
- Fig 4.4 – H-Bridge
- Fig 4.5- Arduino
- Fig 4.6 – Webcam
- Fig 4.7 – Track
- Fig 5.1 – Concept of Self Driving Car Block
- Fig 5.2 – Prototype of self-driving car on block
- Fig 5.3 – Canny Edge Detection
- Fig 6.1 – Block Diagram Training Phase
- Fig 6.2 - Code

TABLE OF CONTENTS

1. INTRODUCTION...	13
1.1. Overview.....	14
1.2. Project Overview.....	18
1.3. Project Objective.....	19
1.4. Project Methodology.....	20
1.5. Thesis Structure.....	22
1.6. Conclusion... ..	22
2. LITERATURE SURVEY	23
2.1. End to End Deep Learning for Self-Driving Cars.....	23
2.2. Implementation of lane detection algorithm for self-driving cars.....	26
2.2.1. Design and Implementation of lane detection algorithm for self-driving car.	27
2.3. MIT Autonomous Vehicle Technology Study.....	31
2.4. Learning Affordance for Direct Perception Autonomous Driving.....	32
2.5. Conclusion... ..	33
3. PROJECT PRELIMINARIES	34
3.1. Machine Learning	34
3.2. Neural Networks... ..	35
3.3. Deep Neural Networks.....	36
3.4. Convolutional Neural Networks... ..	37
3.5. Conclusion.	40
4. HARDWARE... ..	41
4.1. Toy Car... ..	42
4.1.1. Motor.....	42
4.2. Rechargeable battery.....	43
4.3. H-bridge Motor Driver.....	44
4.3.1. Schematic Diagram... ..	44
4.3.2. Board Dimensions and Pin Configuration... ..	45
4.4. Arduino Mega-2560.....	45

4.4.1. Steering Angle...	46
4.4.2. Motor Controlling.....	46
4.4.3. Serial Communication... ..	47
4.5. LCD.....	47
4.6. Laptop	48
4.7. Webcam... ..	48
4.8. Tracks.....	49
5. Image Processing and Deep Learning Techniques used in the Project.....	52
5.1. Overview.....	52
5.2. Data Collection.	53
5.2.1. Classification.....	53
5.2.2. Grayscale.....	54
5.2.3. Gaussian Blurring.....	54
5.2.4. Canny Edge Detection.....	55
5.2.5. Region of Interest.....	55
5.2.6. Bitwise AND Operator.....	56
5.2.7. Hough Transform... ..	56
5.2.8. Display Lines... ..	56
5.3. CNN Architecture	58
5.3.1. Data Pre-processing	58
5.3.2. Input Layer.....	59
5.3.3. Hidden Layers.....	59
5.3.4. Output Layer... ..	59
5.3.5. Model Evaluation.....	59
5.4. CNN Hyper Parameters	60
5.4.1. Convolutional and Pooling Layers Parameters.....	60
5.4.2. Add/Subtract and Sequencing of Convolutional and Pooling Layers.....	61
5.4.3. Number of fully Connected Layers and their Parameters	61
5.4.4. Model Compilation Parameters	61
5.4.5. Model Evaluation Parameters	61
5.5. Training of Data	62

5.6.	Testing using Trained Model...	62
5.7.	Serial Communication...	62
5.8.	Conclusion	62
6.	SOFTWARE...	63
6.1.	Overview.....	63
6.2.	Computer Vision...	65
6.2.1.	Video Capturing and Saving using OpenCV.....	65
6.2.2.	Video to Frames Conversion using OpenCV.....	66
6.2.3.	Image reading using OpenCV.....	67
6.2.4.	Images to Video Conversion.....	68
6.2.5.	Importing Libraries	69
6.2.6.	Function Defined for Preprocessing of Data	69
6.2.7.	Function Defined for Region of Interest.....	70
6.2.8.	Function Defined for Displaying Hough Lines Transform.....	71
6.2.9.	Averaging of Slopes and Intercepts of Hough Lines	71
6.2.10.	Main of Computer Vision Section	72
6.3.	Convolutional Neural Networks	73
6.3.1.	Importing of Necessary Packages.....	74
6.3.2.	Initialization and Sorting of Data.....	75
6.3.3.	Preprocessing of Data and Labels.....	76
6.3.4.	Scaling of Raw Image Pixel Intensities.....	77
6.3.5.	Splitting Data into Training and Testing.....	77
6.3.6.	Binarizing Label for Testing and Training Data.....	77
6.3.7.	CNN Model for Training of Data.....	77
6.3.8.	Model Summary.....	79
6.3.9.	Model Evaluation.....	79
6.3.10.	Plot for Training Accuracy and Loss.....	80
6.3.11.	Model Saving and Label Binarization to Disk.....	81
6.4.	Serial Communication between Python and Arduino.....	81
6.4.1.	Importing Libraries	81
6.4.2.	Arduino Port Detection	82

6.4.3. Loading Trained Model for Pickling	82
6.4.4. Prediction from Different Track Images	83
6.4.5. Finding Labels According to Prediction	84
6.4.6. Labels for Multiple Directions	84
6.4.7. Writing Text Data and Labels on Frames.....	85
6.5. Arduino Controller Working According to Prediction	86
6.5.1. Initialization	86
6.5.2. Controlling Setup	87
6.5.3. LCD Display and Motors Controlling	87
6.5.4. Initializing Serial Communication and Angle Calculation.....	87
7. CONCLUSION.....	91
8. REFERENCES	93

Abstract

Self-driving cars developed by different companies such as Google, Tesla, Nissan, Audi etc., uses Artificial Intelligence techniques as a key ingredient to achieve autonomous capabilities under dynamic environmental conditions. It is expected that there will be around 21 million driverless cars in the United States and 27 million in Europe by 2030. The idea of self-driving cars or autonomous cars originates by witnessing accidents due to careless driving of people which could sometimes become extremely harmful. In this project, a toy car is converted into a self-driving car capable to move autonomously under a constraint environment. Self-driving capability is developed using computer vision and AI.

Initially, tracks are deployed on surface in order to gather data through video streaming by using a webcam. From these video, images are extracted for classification of data into four different classes, i.e; right, left, forward and stop. This classified data is converted into required format using computer vision algorithms such that the data consists of only bright Hough lines on a black image. This data is then trained using HAAR cascade such that the trained model is able to predict in real-time whether to move the car left, right, forward or stop, accordingly. This project does not include any sensor and is totally computer vision based. The training and inference is done using a core i5 laptop. The trained model takes the input images from live feed and predicts which direction to choose or stop. The trained model after prediction generates a string and through serial communication the string is sent to Arduino. Finally, the Arduino processes the wrappers embedded in its code according to the string received from the trained model and sends control signals to the motors of the toy car to move or stop according to the prediction.

Chapter One

INTRODUCTION

1.1 Overview

This chapter explains the project overview, project objectives and project methodology alongside with basic idea behind Self Driving Cars, history of self-driving car, industries producing these autonomous vehicles, pros and cons and market trends. A self-driving car is a car that guides its self without human interference. It is designed to sense its own environment using deep learning and computer vision techniques.

Autonomous cars have now gone uphill from science fiction to reality. It seems like this technology emerged overnight, but in reality the path to self-driving cars has been a long and tedious one. The history of self-driving cars went through several milestones. After the invention of man driven motor vehicles, it did not take too long for the inventors to think upon self-driven vehicles. In the year 1925, the inventor Francis Houdina invented a radio-controlled car, which he drove through the streets of Manhattan without any human steering the wheel. The radio-controlled vehicle can start its engine, shift gears, and turn on its horn [2]. In year 1969, John McCarthy who is the one of the founding fathers of artificial intelligence demonstrated something likely to the modern autonomous vehicle in an essay titled “Computer-Controlled Cars.” McCarthy refers to an “automatic chauffeur,” capable of navigating a public road via a “television camera input that uses the same visual input available to the human driver” [2]. In early 90s, Carnegie Mellon researcher Dean Pomerleau wrote a PhD thesis, describing how neural networks could allow a self-driving vehicle to take in raw images from the road and output steering controls in real time [2]. Pomerleau was not the only researcher working on self-driving cars, but his use of neural networks proves way more efficient than alternative attempts to manually divide images into “road” and “non-road” categories. Waymo, which is also known as “Google Self-driving car” collects tons of data and feed the data to deep learning algorithm from labeling and processing. Waymo, nowadays is being

used as an online cab service like Uber in different states of the US [2]. The basic model for a self-driving car is shown in figure 1.1 [3]:

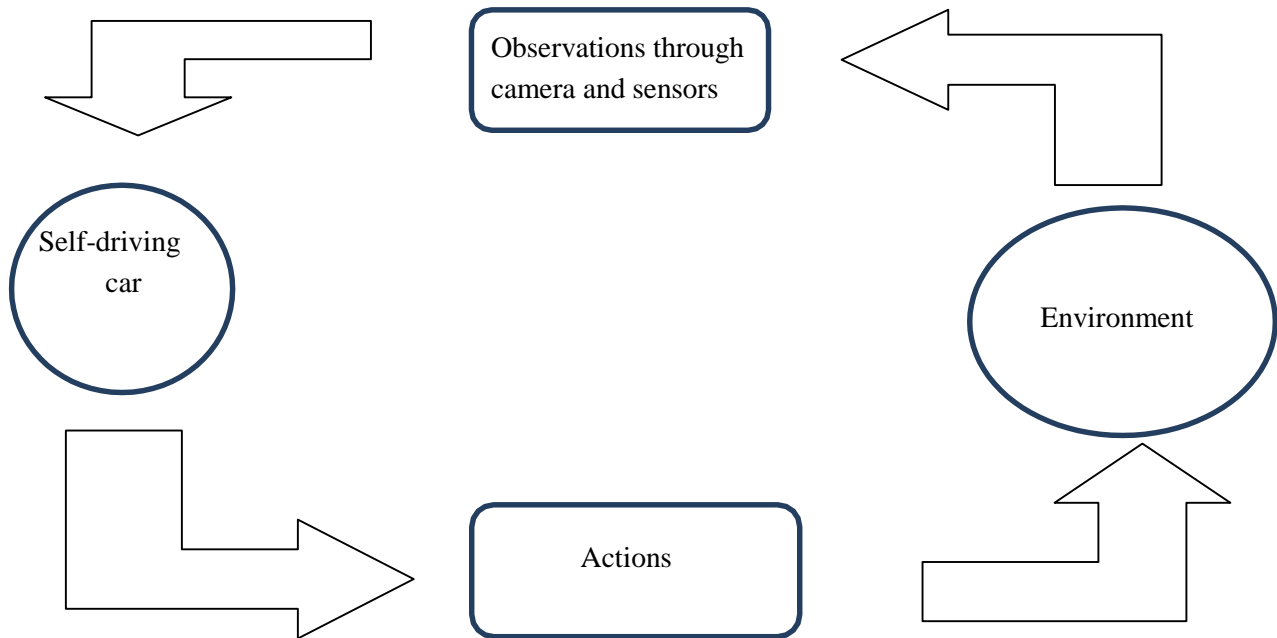


Figure 1.1: Basic model for self-driving cars [3]

The basic concept behind self-driving car is to sense its environment and take actions accordingly as shown in fig 1.1. The car collects data of its environment through a single or multiple cameras along with different sensors. This data is then processed through advanced computer vision and other algorithms to generate action required to maneuver the car according to the environment [3].

For example, in case of Waymo, machine learning plays an important role. With the collaboration of Google AI researchers, Waymo is integrated with AutoML which enables the autonomous car to optimize models for different scenarios at a great velocity. AutoML has a graphical user interface to train, assess, improve, and arrange models based on the data [4].

GM cruise comes second to cover most number of miles autonomously [5]. It is considered to be world's most advanced self-driving vehicles to carefully connect people with the places, things, and experiences they care about. Safety plan and functioning requirements in obedience with federal, state and local regulations are kept in focus. Cruise self-driving car has a balanced array of sensors so that the vehicle can map out the complex city streets intelligently and with a 360-degree view of the world around it. Each car contains 10 cameras that take pictures at 10 frames per second [5]. That is how this car is able to see more of its surrounding environment and therefore, can respond more quickly and safely.

At Nissan, advancement in artificial intelligence is making the autonomous vehicles smarter, more responsive, and better at making their own decisions. Nissan is developing a vehicle that will be competent of self-driving on a single lane road in the near future. The next step will be multi-lane road, then self-driving in the city, and in the end fully autonomous driving in all situations. Nissan self-driving cars are designed to get smarter with time. Seamless Autonomous Mobility is a system developed from NASA technology to help autonomous cars deal with unexpected, the vehicle sends live data to a mobility manager who instantly teaches it what to do then it shares what it learned with other cars in the system so they get smarter too [6]. Once this knowledge is absorbed into the system, vehicles can start using it to solve completely new challenge.

How safe should a self-driving car be? This question is becoming increasingly important as companies like Waymo, Uber and other test their self-driving cars on public road. Around the world, 1.3 million people are killed each year by vehicles [7]. Many of these deaths are due to human error. If a self-driving car can help in this, it would be a great achievement. In contrast, the self-driving vehicles do not drink, they do not text and they do not fall asleep at the wheel, thus reducing deaths and injuries. Statistics illustrated in figure 1.2 [6] show the likeness of people around world towards self-driving vehicles.

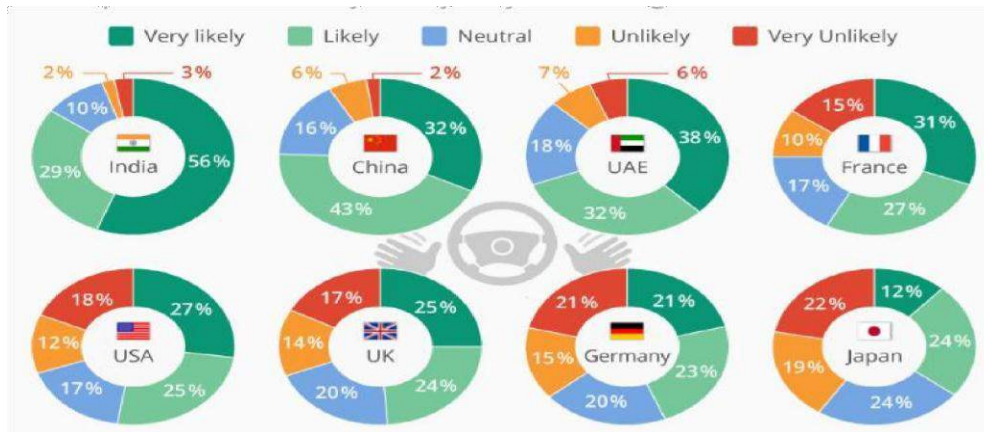


Figure 1.2: Survey chart [6]

Twenty percent of people living in America are suffering with disabilities [13]. Eleven million medical appointments missed each year because of transportation barriers which leads to greater health care expenses later on, but this new technology of self-driving cars can be a promising solution to this problem [13]. Self-driving cars lead to 19 billion dollars in annual health care savings [13]. Two million more people with disabilities could become employed with the help of reliable transport. Self-driving cars could save around 1.3 trillion dollars through fuel and accident avoidance [7]. A lot of traffic is caused by human error. Self-driving car can eliminate human error that causes traffic jams. Self-driving cars can be programmed to space out between cars automatically as illustrated in figure 1.3 [2], thereby eliminating the problem of congested traffic. Cars using adaptive cruise control infer the effect of a braking event smoother than those vehicles without the activated technology [5].



Figure 1.3: Self-driving environment by Volkswagen [2]

There are many ways that self-driving cars can improve our lives. There also exist some downsides. The automobile industry could suffer. Self-driving cars mean that car manufacturers make fewer models and less cars, resulting in fewer jobs and less choice for the consumer. Self-driving cars are totally dependent upon computers, which makes them more vulnerable to hackers and other cyber threats. And the worst of all threat is the unemployment.

1.2 Project overview

In this project, a working prototype of a self-driving car is developed using computer vision and AI. Self-driving car made in this project is able to navigate the track by making prediction using the trained data set with the help of HAAR cascade model. Before feeding the data to cascade trainer GUI model for training, it is pre-processed using computer vision techniques such as Gray Scale, Gaussian blur, canny-edge detection, bitwise AND operator and Hough transform. The pre-processing is done to identify the lane line on track on which car has to move. Initially, tracks are deployed on the ground in order to gather the data in a form of videos using OpenCV with webcam interface. From these videos, images are extracted for classification of the data into four different classes i.e. right, left, forward or stop. Before feeding the data to HAAR cascade model, Hough transform is applied using OpenCV for finding the lane line. This data is trained using HAAR cascade model and a classifier is set which is able to predict in real time whether to move the steering of the car left, right, forward or stop accordingly. This project does not include any sensor and it is totally computer vision based. The training and inference is done using core i5 laptop. In our cascade model, we have used 15 stages with learning rate of 0.001. Classifier takes the input images from the live feed and predicts which direction to choose or stop. Classifier after prediction generates a string and through serial communication the string is sent to Arduino. Finally, the Arduino processes the wrappers embedded in its code according to the string received from the classifier and the car moves according to the prediction. The trained model predicts in which direction to move and can also respond to traffic sign, such as a stop sign. The picture of the toy car used in this project is shown in figure 1.4. This is a motorized car in which additional modifications such as steering angle control, motor driver control and high resolution camera interfacing has been done.



Figure 1.4: Prototype of self-driving car

1.3 Project Objective

- Development of working prototype of a self-driving using a toy car that will mainly navigate using computer vision and AI
- Usage of HAAR cascade model to identify a stop sign

1.4 Project Methodology

The block diagram of project is shown in figure 1.5. Initially, tracks are deployed on surface in order to gather data through video streaming through a webcam using OpenCV which is an open source computer vision library. After the collection of data, the video was segmented into frames and classified into four classes i.e. right, left, forward and stop. This classified data is converted into required format using computer vision algorithms such that the data consists of only bright Hough lines on a black image. To get the required format of image, the image was first converted into gray scale. To reduce noise and smoothing the data images, Gaussian blur is applied. As noise in image create false edges and effect edge detection, so after smoothing of images the canny method is applied to identify edges in the image. To identify the region on the image for Hough

lines, the region of interest is found and to mask out anything else in the image bitwise and operator is used. Initially the Hough lines are drawn on a zero pixel image using bitwise and operator inside the region of interest. Weights of Hough line transform image and the real image of the track are added. By adding weights of these images, the Hough lines are displayed. These displayed lines are then averaged according to their slopes and intercepts in order to display the lines in equal ratio. After pre-processing, the data is feed to a CNN model for training. The training and inference is done using core i5 laptop and 1070 core i7 system. Training data used in our project is about 70 percent of the complete data. Supervised learning is used for training of the data. This data is classified and labeled as Right, left and forward. This data is trained using CNN sequential model. CNN model used for the training of the data contains 15 hidden layers. These layers include dense layer, convolutional-2D layer, maxpooling-2D layer, flatten layer and fully connected layers. CNN is used for extracting the features from the images and learn through these features by updating the bias and weights of the perceptron. Categorical cross entropy with Adam optimizer and a learning rate of 0.001 is used in this model. The trained model then takes the input images from live camera and predicts which direction to choose or stop. The trained model after prediction generates a string and through serial communication the string is sent to Arduino. Finally, the Arduino processes the wrappers embedded in its code according to the string received from the trained model and sends control signals to the H-bridge to drive motors of the car to move or stop according to the prediction. The block diagram of hardware is shown in figure 1.5 [1].

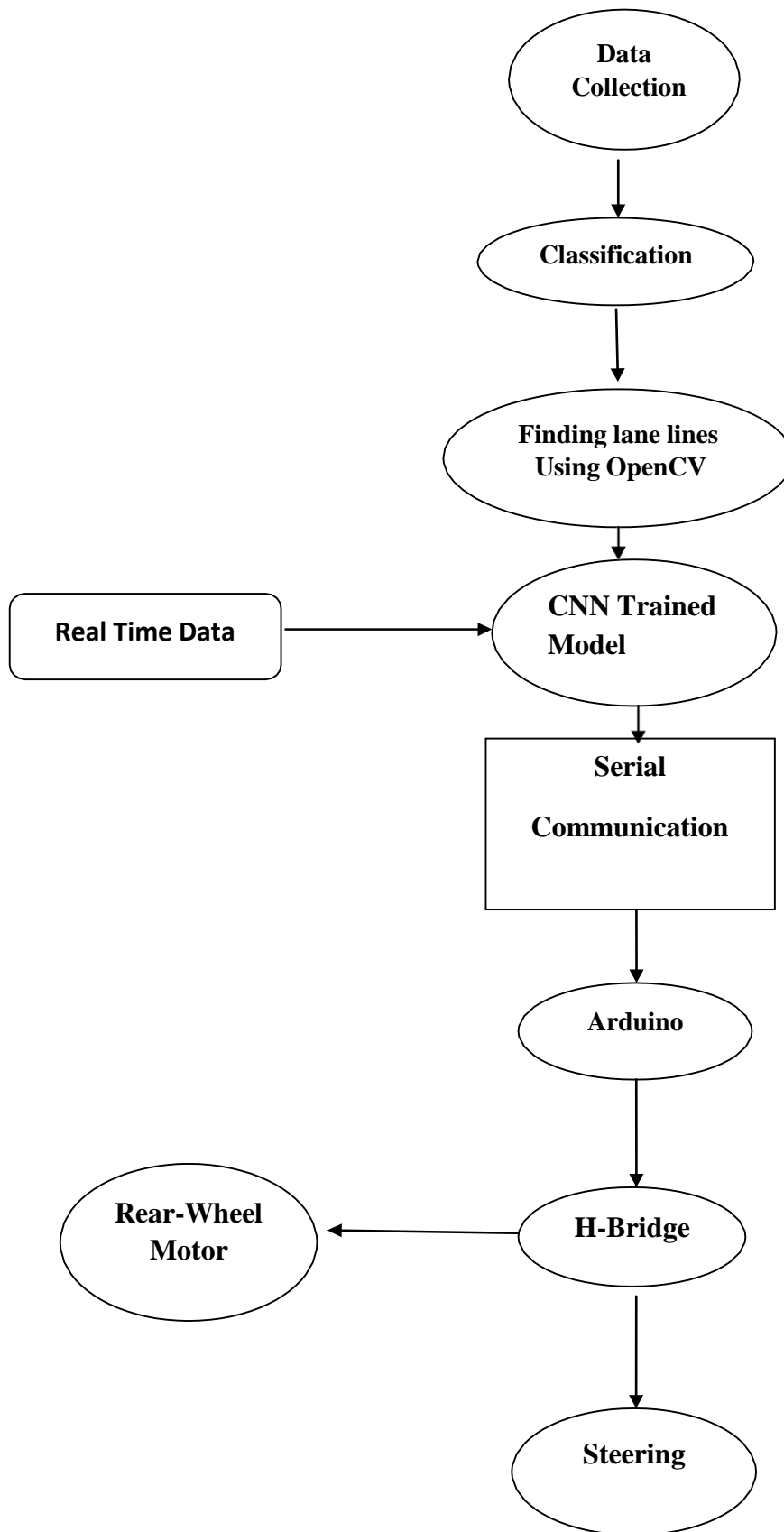


Figure 1.5: Block Diagram of project

1.5 Thesis Structure

- Chapter 1 provides a general introduction to self-driving cars. It includes a project overview, project objectives, importance of the project, methodology and brief history of self-driving cars and industries working on this technology.
- Chapter 2 provides the literature review. Review contains explanation related to the techniques used in the project, i.e. machine learning, computer vision, image processing, neural networks and convolutional neural networks.
- Chapter 3 consists of project preliminaries and discussion. It explains the learning methods and procedure to be followed to complete the project.
- Chapter 4 explains the hardware design. In this chapter, hardware methodology to design and develop a toy car into a self-driving car is explained.
- Chapter 5 Describes image processing and deep learning techniques used in the project.
- Chapter 6 presents the software portion of this project. The techniques for collecting data to apply Hough transform and neural networks models for training.
- Conclusion is presented in Chapter 7.
- Chapter 8 provides the list of references.

1.6 Conclusion

Autonomous cars have now gone uphill from science fiction to reality. Basic technologies behind the self-driving cars are deep learning and computer vision techniques. Self-driving cars can potentially overcome the mistakes made by human drivers thus saving human. In this project, a prototype of self-driving is developed that has the ability to maneuver on predefined paths using convolutional neural networks and computer vision techniques.

Chapter Two

LITERATURE SURVEY

This chapter explains the review of few research papers explaining how different deep learning techniques are being used with computer vision to drive an autonomous cars without any human work. This chapter also includes a brief history and state-of-the-art of self-driving cars.

Since 2018, Waymo has done incredible work in deploying and testing vehicles in various domains and reached the highest running self-driving car i.e. running 10 million miles [8]. From the machine learning perspective all this achievement is because of data collection and all of these miles driven with a primary sensor being a camera, so that is computer vision and there exist a set of neural networks behind it. That is probably the largest deployment of neural networks in the world that has direct impact on human life and is able to decide life's critical decisions.

2.1 End-to-End Deep Learning for Self-Driving Car [9]

NVIDIA'S new application for self-driving cars has used CNN to plot the real time images pixels from a front-facing camera to the steering commands. Here, the end-to-end approach means that with the minimum data to train, provided by humans. The system will learn to steer, with or without lane lines, on both local roads and highways. The system that the NVIDIA used is trained to automatically learn necessary processing steps, such as detecting valuable road features, with only the human steering angle as the training signal. The CNN used by the NVIDIA self-driving car is way beyond basic pattern recognition. They have developed a system that learns the entire processing needed to steer an automobile.

The base for this project was actually started 10 years ago in a Defense Advanced Research Projects Agency (DARPA) project known as DARPA Autonomous Vehicle (DAVE). DAVE was trained on hours of human driving. The training was done by using two cameras and the steering

commands sent by a human operator. But DAVE's performance was not suitable for driving it on high-ways autonomously.

NVIDIA started working on improving DAVE in 2016 to create a robust system for driving on public roads and named it DAVE-2.

For the collection of training data three cameras are mounted (as shown in in block diagram of figure 2.1 [9]) behind the windshield and video is captured all together with the steering angle applied by the human driver. Vehicle's Controller Area Network (CAN) bus is tapped to get steering command.

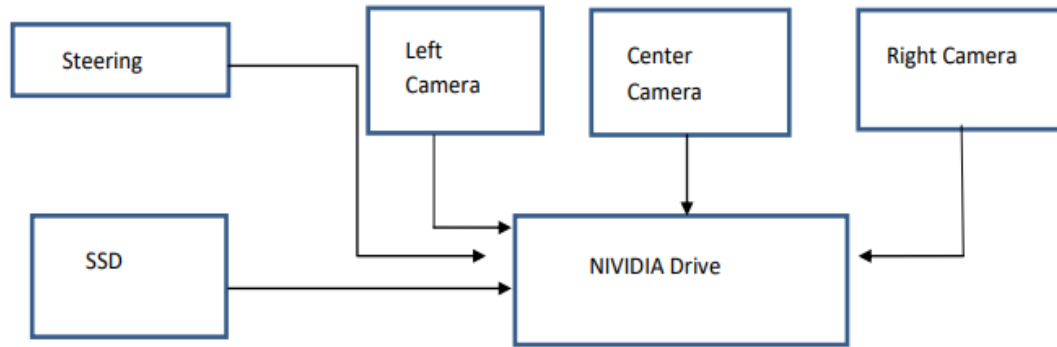


Fig 2.1: High level view of data collection system [9]

Training data consists of single image extracted from the video and paired with the corresponding steering command ($1/r$) [9]. The steering command is represented by $1/r$, where r is the turning radius in meter. To avoid the singularity while driving straight $1/r$ is used. Only the human driver is not sufficient for training; the network must also learn how to recover from any mistake. The training data is, therefore, improved with added images that shows the car in different shifts from the center of the lane line and rotations from the direction of the road. The block diagram of training the neural network is shown in figure 2.2 [9].

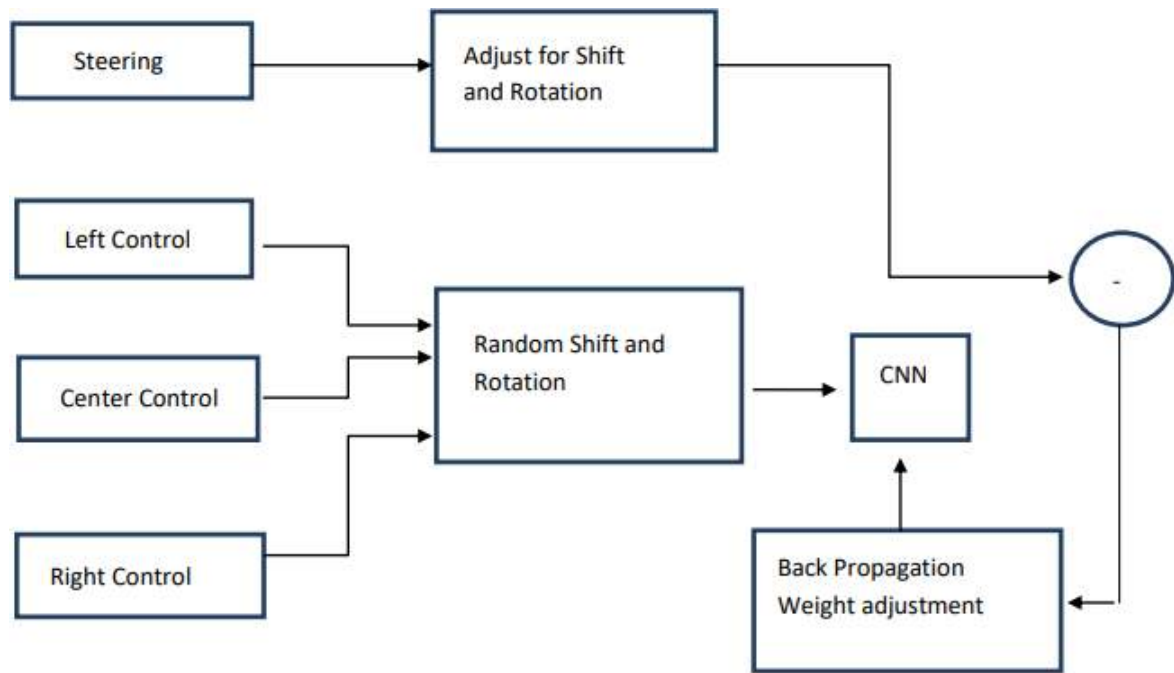


Figure 2.2: Training the neural network. [9]

The first step to train the neural network is to select frames from data. The data collected by NVIDIA was labeled by road type, weather condition and drivers activity. After selecting the final set of frames, adding artificial shifts and rotations were added to teach the network how to recover from a poor position or direction. Before taking it on the road the network was tested on a simulator. Figure 2.3 shows the block diagram of simulation of DAVE-2 [9].

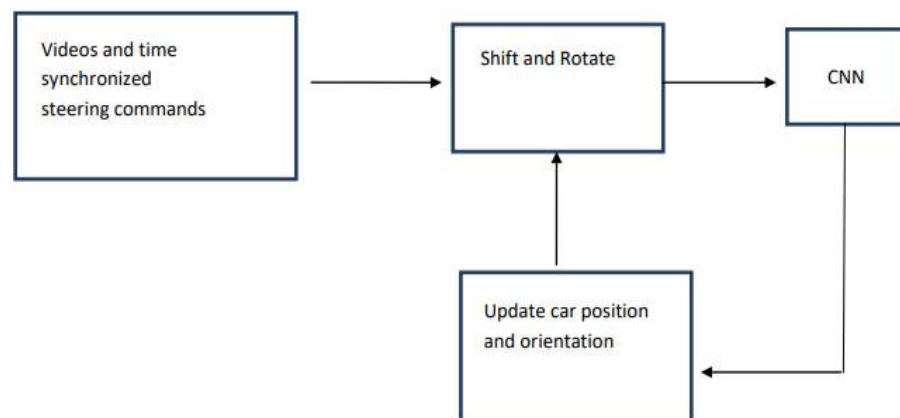


Fig 2.3: Simulation of DAVE-2[9]

NVIDIA proved that CNN is able to learn the entire task of lane line and road following without manual work. A small amount of training data in a few hundred hours of driving was enough to train the car to operate in autonomous condition, on highways, local and residential roads in sunny, cloudy, and rainy conditions. The CNN is able to learn meaningful road features from the steering alone.

2.2 Implementation of Lane Detection Algorithm for Self-Driving Car [10]

Making a self-driving car requires one to have a proper command on machine learning and CNN but also self-driving car requires several other topics that need to be learned more deeply. Such as Deep Learning, Computer Vision, Fusion Sensor, Localization, Control and Path Planning.

Computer vision plays an important role in driving an autonomous car. Computer vision is applied machine learning technique on image processing. Computer vision uses machine learning to identify patterns for the analysis of images. Just like the process of human vision; it helps to distinguish between objects, classify them and arrange them according to their size. Computer vision, in self-driving cars is used to find lane line on the track. This is achieved using a computer vision open source library called OpenCV in python language.

Self-Driving Car concept including Computer Vision, Sensor Fusion, Deep Learning, Path Planning, Actuator and Localization.

Computer vision is a subnet field of image processing that deals with how computers can be made to understand digital images or videos. From the engineering point of view, it seeks to automation tasks and perform emulation of vision at human scale. One of the important tasks in self-driving cars is being able to correctly sense the surroundings. To make this possible the self-driving cars are equipped with multiple sensors or cameras. Having these sensors giving the same outputs with some plus and minuses, combining all the outputs will end up giving better output. Sensor fusion is combining of data collected from different sensors or data derived from different sources such

that the resulting information has less ambiguity than would be possible when these sources were used to collect data individually. This is same as human being using all its five senses to make meaning of an environment around. Deep learning is part of a broader family of machine learning, it uses special algorithms to describe and analyze data, learn from it, improve and predict useful outcomes. Mostly, it is not possible to directly program a computer to perform specific tasks such as driving car, speech recognition and object detection are way too complex to just program whereas, machine learning algorithm can learn and improve, based on experience. It also interacts with environment to learn, to detect and predict meaningful patterns to achieve desire results. The learning occurs between a learner and an environment, on that note. Path-planning is an important primitive for self-driving cars that lets the car find the optimal path between two points. Otherwise optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. Localization in self-driving car mean the ability of car to determine its own position in its frame of reference and then to plan a path towards some goal location. In order to navigate in its environment, the robot or any other mobility device requires representation, i.e. a map of the environment and the ability to interpret that representation.

2.2.1 Design and Implementation of Detection Algorithm for Self-Driving Car

The first step in finding lane lines are use of edge detection algorithm. An edge corresponds to the region of an image where the sharp change in intensity occurs. Edge detection includes a variety of mathematical methods such as greyscale, Gaussian blur and canny edge detection that aim at identifying sharp changes in intensity in adjacent pixels. Edge detection is a fundamental tool in image processing and computer vision, particularly in the areas of detecting or extracting features out of an image or video.

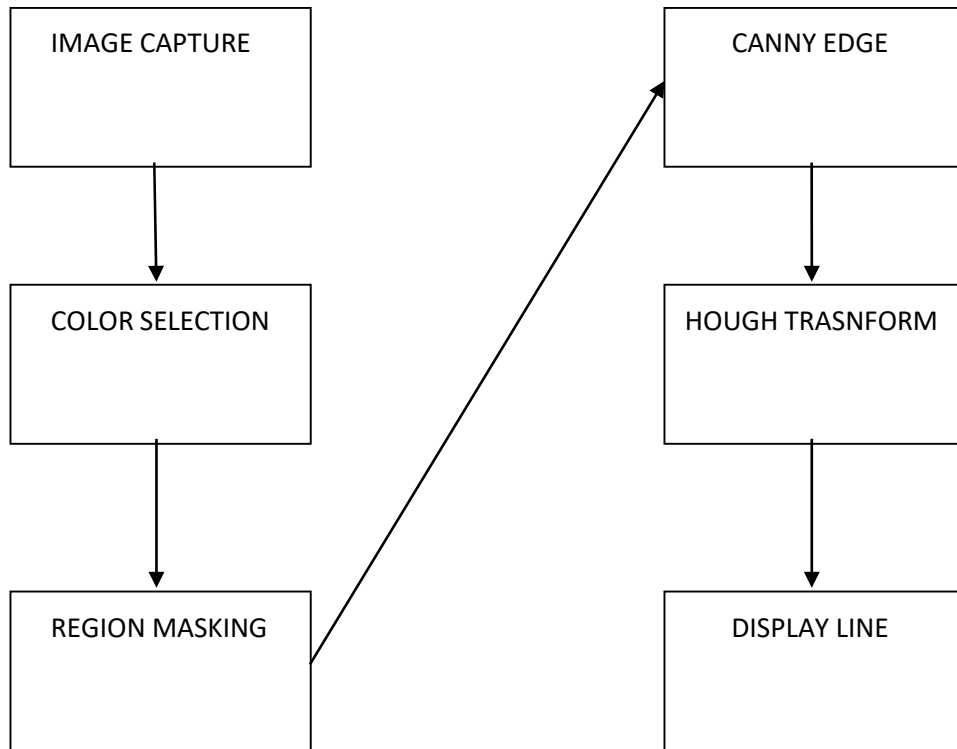


Figure 2.4: Flowchart of image processing for lane detection [10]

Figure 2.4 [10] shows the complete flow chart of image processing for lane detection step wise. In computer vision and image processing, feature detection refers to methods that aim at computing essential information of image and making decisions at every image point whether there is an image feature of a given type at that point or not. In this stage, image is divided into image capture and then color selection until region masking. The processing of first stage is shown in figure 2.5 [10].

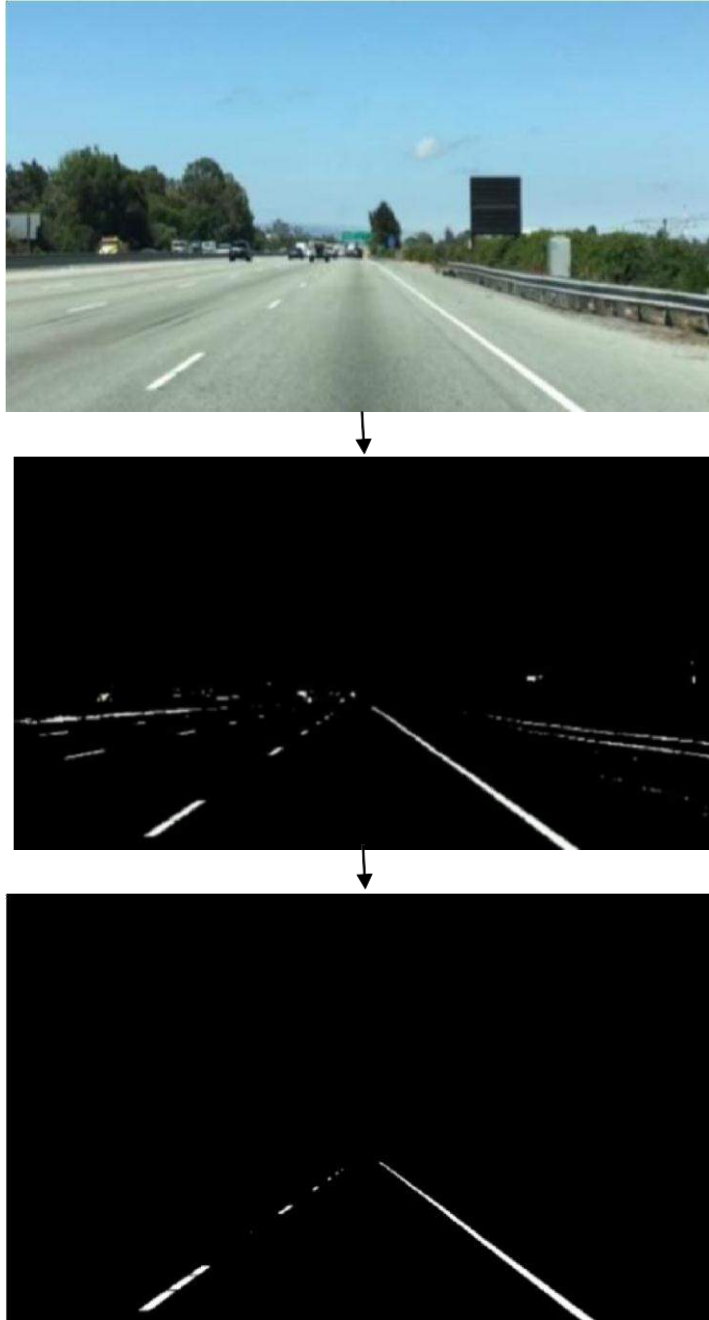


Figure 2.5: Processing First Stage [10]

The Hough transform is a feature extraction technique used in computer vision, and digital image processing [12]. The purpose of this technique is to find the points on the image with same threshold and create lines on them. This technique has ability to join the points having gaps on the image and same features. The classical Hough transform was only used in identification of lines

in the image, but later the Hough transform has been later extended to identify arbitrary shapes most commonly circles or ellipses. The processing of second stage is shown in figure 2.6 [10].

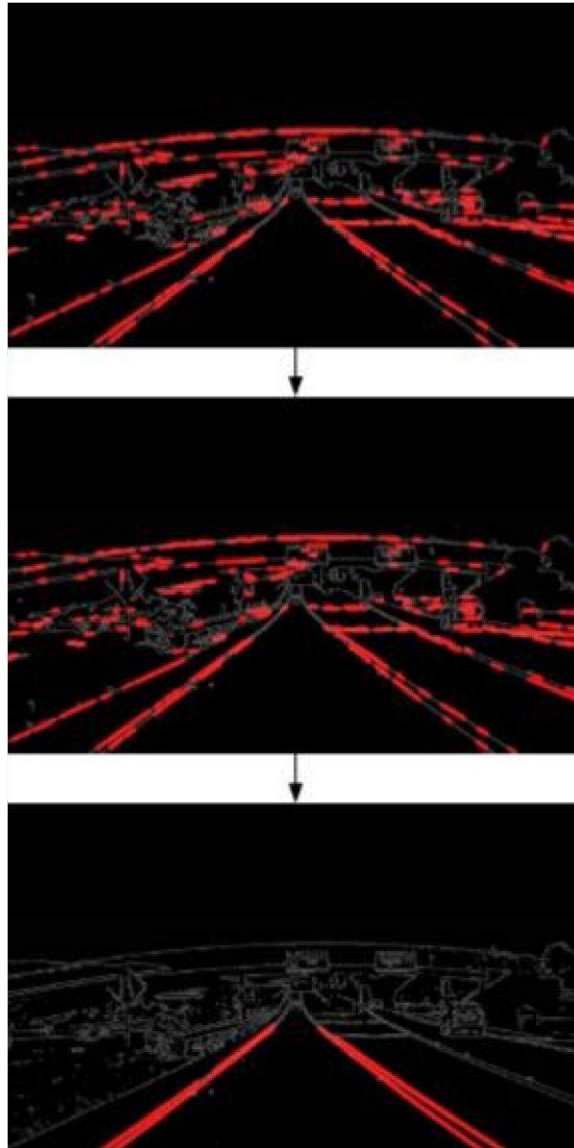


Figure 2.6: Processing Second Stage [10]

The result shows this algorithm needed to add some method that can adaptively change the parameter during day and night. Because of constant parameters can only be used in the same lighting conditions. Overall the implementation method in an OpenCV can successfully detect the road lane.

2.3 MIT Autonomous Vehicle Technology Study [11]

MIT Autonomous Vehicle Technology (MIT-AVT) study is all about taking extensive real world driving data in the form of images and high quality video to help in the development of deep learning based internal and external observation systems, which helps in gaining comprehensive understanding of how human beings interact with vehicle automation technology by integrating video data. Also identifying how AVT and other factors related to automation adoption can be improved in ways that save lives. This review presents the design of the study, the data collection hardware, the processing of the data, and the computer vision algorithms currently being used to extract actionable knowledge from the data.

In order to gain these objectives, the MIT team instrumented 23 Tesla Model S and Model X vehicles, 2 Volvo S90 vehicles, 2 Range Rover Evoque, and 2 Cadillac CT6 vehicles for both over a year per driver and one month per driver data collection.

Inside a car there are three cameras, one looking at the driver face which captures things like where the driver is looking, the emotional state also there is camera looking at the driver's body including hands. And finally there is a forward facing camera attached to windshield looking forward capturing external environment and characteristics of the road. These three cameras help to study the driver's behavior and interaction with automation. The MIT-AVT is all about collecting large amount of naturalistic driving data. All the three cameras are connected to RIDER (Real-time Intelligent Driving Environment Recording system). RIDER consists of single board computer running a custom version of Linux. This single board computer integrates all of cameras, sensor and GPS and it offloads all to the solid state drive. After the collection of data, the data is processed using computer vision and deep learning algorithm.

The application of state-of-the-art embedded system programming, software engineering, data processing, distributed computing, computer vision and deep learning techniques to the collection and analysis of large-scale naturalistic driving data in the MIT-AVT study seeks to break new ground in offering insights into how human and autonomous vehicles interact in the rapidly changing transportation system. This work presents the methodology behind the

MIT-AVT study which aims to inspire the next generation of naturalistic driving studies. The governing design principle of this study is that, in addition to prior successful NDS approaches, leverage the power of computer vision and deep learning to automatically extract patterns of human interaction with various levels of autonomous vehicle technology.

2.4 Learning Affordance for Direct Perception in Autonomous Driving [12]

Chen et al. [12] present an approach which instead of learning the mapping from pixels, first estimates a several human key perception indicators which are directly related to affordance measures such as the distance to surrounding cars.

There are two major paradigms for self-driving. First one is the mediated perception approach which detects interesting objects in the image and computes driving commands based on those detections. The second one is behavior reflex, this approach maps the input image directly to let the car drive. The third paradigm proposed in this paper is the direct perception. Giving an image to a small number of key perception indicators that directly relate to the affordance of a road/traffic state for driving. For demonstration, deep Convolutional Neural Network is used, recording 12 hours of human driving in a video game which shows that the model can work well to drive a car in a very diverse set of virtual environments.

Model is built upon the state-of-the-art deep Convolutional Neural Network framework to automatically learn image features for estimating affordance related to autonomous driving. For training set a human driver where asked to play a car racing video game TORCS for 12 hours while recording the screenshots and the corresponding labels. TORCS is an Open Racing Car Simulator [12] mostly used by AI researchers. The data collected is then stored and used to train a model to estimate affordance in a supervised learning manner. In the testing phase, at each time step, the trained model takes a driving scene image from the game and estimates the affordance indicators for driving. In both, testing and training, traffic is configured by putting a number of pre-programmed AI cars on road.

This representation leverages a deep CNN algorithm to estimate the affordance for driving actions instead of analyze entire scenes which is a part of mediated perception approaches, or blindly mapping an image directly to driving commands which is behavior reflex approaches. Experiments show that this approach can perform well in both virtual and real environments.

2.5 Conclusion:

NVIDIA proves that CNN is able to learn the entire task of lane line and road following without manual work. A small amount of training data in a few hundred hours of driving is enough to train the car to operate in autonomous condition. Lane line detection algorithm using Hough transform needed to add some method that can adaptively change the parameter during day and night. NDS plays an important role towards the safe driving of autonomous vehicles.

Chapter 3

Project Preliminaries

This chapter explains the preliminaries related to self-driving cars. The topics covered in this chapter includes; machine learning, neural networks, deep neural network and convolutional neural network.

3.1 Machine learning

Machine learning uses algorithms to describe and analyze data, learn from it, improve and predict useful outcomes. Mostly, it is not possible to directly program a computer to perform specific tasks such as driving car, speech recognition and object detection are way too complex to just program whereas, machine learning algorithm can learn and improve, based on experience. It also interacts with environment to learn, to detect and predict meaningful patterns to achieve desire results. Generally speaking, there are two types of machine learning algorithms, supervised and unsupervised learning. Supervised learning is the most popular machine learning technique that uses date to teach the algorithm what conclusion it should come up with. It typically begins with a dataset associated with labelled features that define the meaning of the data and find pattern which can be applied to analytical processing [3]. In figure 3.1 [3] supervised machine learning block diagram is shown.

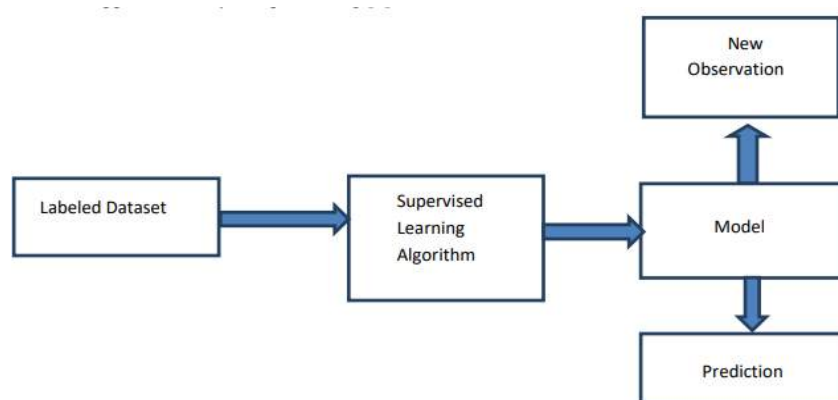


Figure 3.1: Supervised Learning [3]

Unsupervised learning is another machine learning approach that occurs between the learner and its environment. In this process, a learner gets a large data set with no labels and the learner task is to process that data, find similarity and differences in the information provided and act on that information without prior training. Figure 3.2 [3] is showing unsupervised learning.

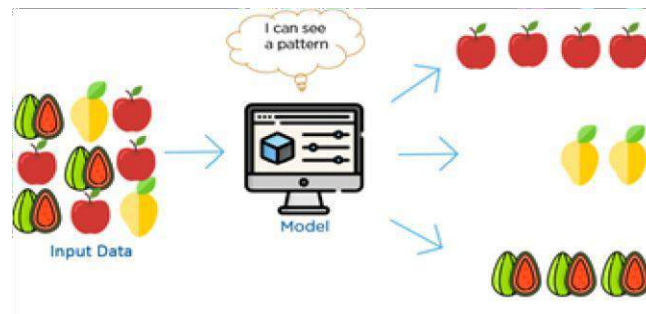


Figure 3.2: Unsupervised learning [3]

3.2 Neural network

Neural networks have been around for a long time. But in a past few decades neural networks have become a prevalent in the field of Artificial Intelligence. As their name suggests, neural networks are inspired by biological neural networks and vaguely emulate the way humans learn [18]. The most basic form of the neural network is the perceptron analogues to a biological neuron of a brain.

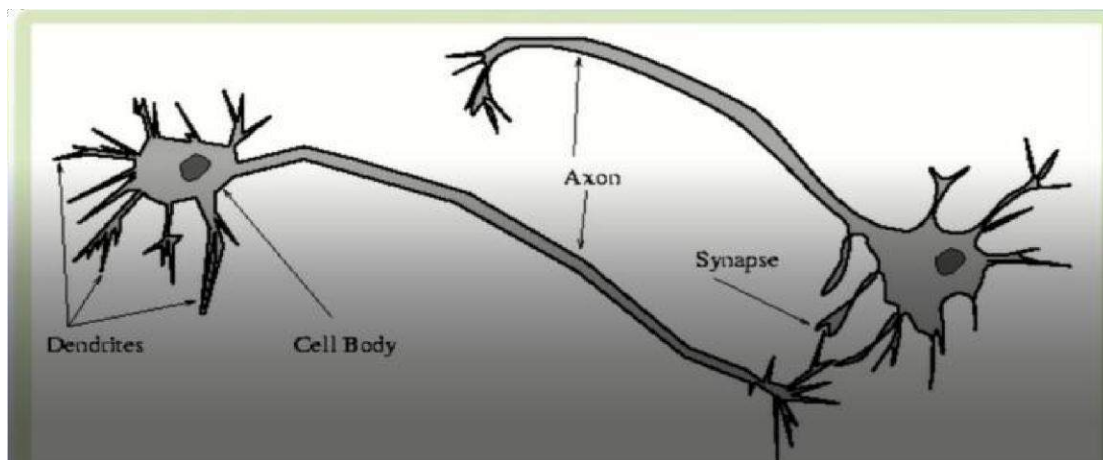


Figure 3.3: Biological neural network [3]

The perceptron is trained to receive input in the form of input nodes and transfer the appropriate output similar to have with neurons, the dendrites receive electrical signals while the axon branches to multiple axon terminals to send the appropriate signal. A basic biological neural network is shown in figure 3.3 [3] while an artificial neural network is shown in figure 3.4 [3]. As shown, the basic artificial neural network consists of input layers, hidden layers and output layers.

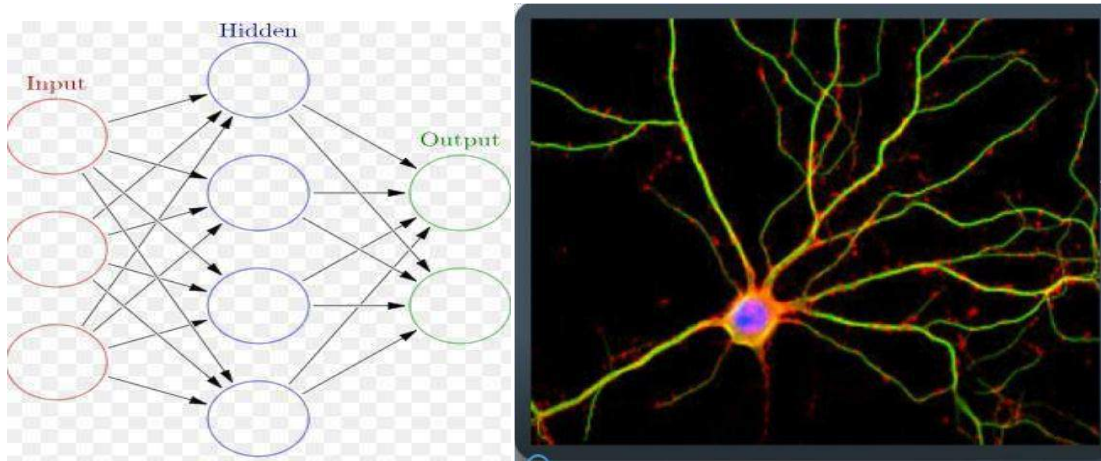


Figure 3.4: Multi layered neural network and biological neural network [3]

3.3 Deep Neural Networks

In dealing with complex data, we require a deep neural network as shown in figure 3.5 [3] with much higher capacity to learn. A deep neural network consists of several hidden layers interconnected to each other.

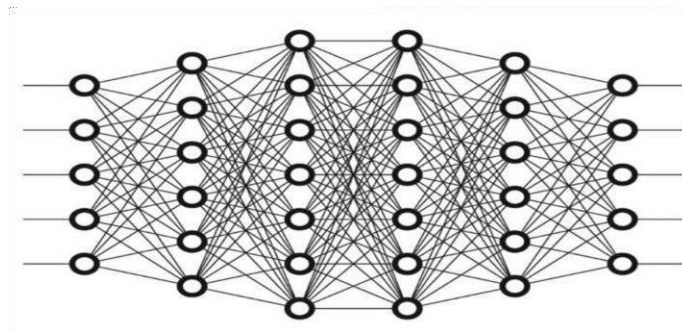


Figure 3.5: Deep neural network

3.4 Convolutional Neural Network

This is the most popular and most successful neural network architecture for deep learning. CNN has the exceptional ability to extract important and distinctive feature from images for each class. In figure 3.6 [3] a basic convolutional neural network block diagram is shown.

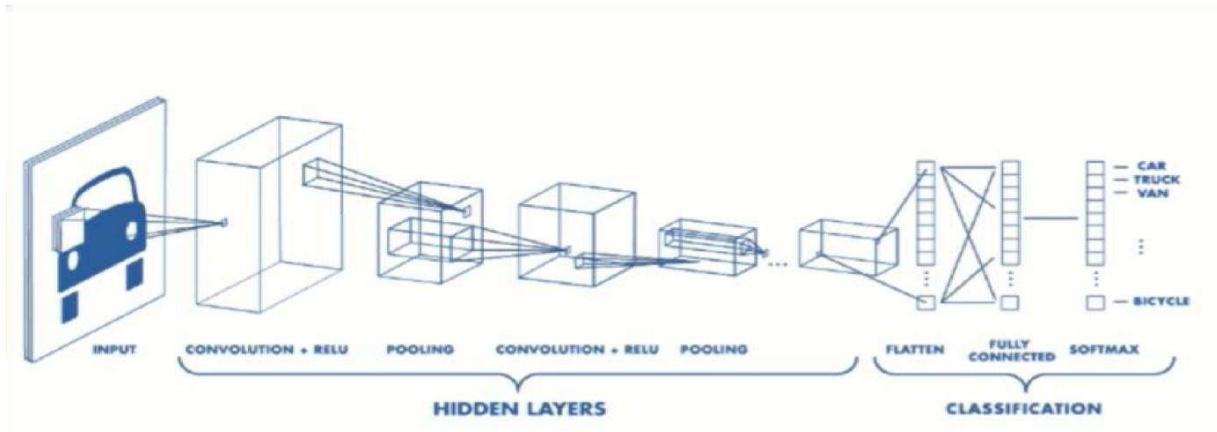


Figure 3.6: Convolutional Neural Network [3]

Convolutional layer is the first layer to take out features from an input image. Convolution conserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a kernel. Consider a 5 x 5 image matrix whose pixel values are 0 & 1 and kernel of 3 x 3 (shown in figure 3.7) [3].

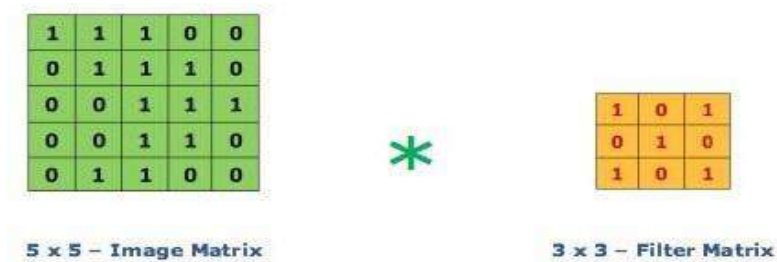


Figure 3.7: Image matrix multiplies kernel or filter matrix [3]

Then the convolution of 5 x 5 image matrix multiplies with 3 x 3 kernel (shown in figure 3.8 [3]) which is called “Feature Map”.

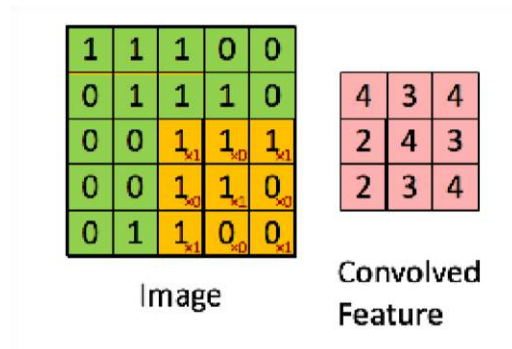


Figure 3.8: 3 x 3 Output matrix [3]

Convolution of an image with different kernels can carry out operations such as edge detection, sharpen and blur by applying kernel. Rectified Linear Unit (RELU) is used to introduce non-linearity in a convolutional neural network. There are other nonlinear functions such as tan or sigmoid but performance wise RELU is better than others.

The function of pooling layers is to reduce the number of parameters when the images are too large. Pooling layer is also called sub-sampling or down-sampling which changes the dimensionality of each map but keep the important information. The different types of pooling are max pooling, average pooling and sum pooling. Max pooling take out the largest element from rectified feature map, as shown in the figure 3.9 [3].

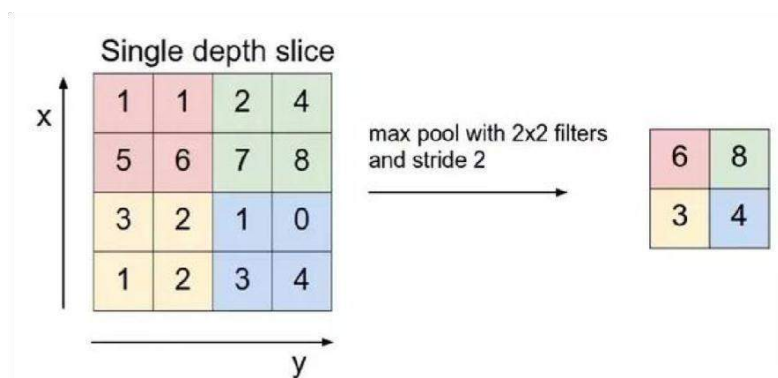


Figure 3.9: Max pooling [3]

The layer is called as fully connected layer, flattened the matrix into vector and feed it into fully connected layer like neural network.

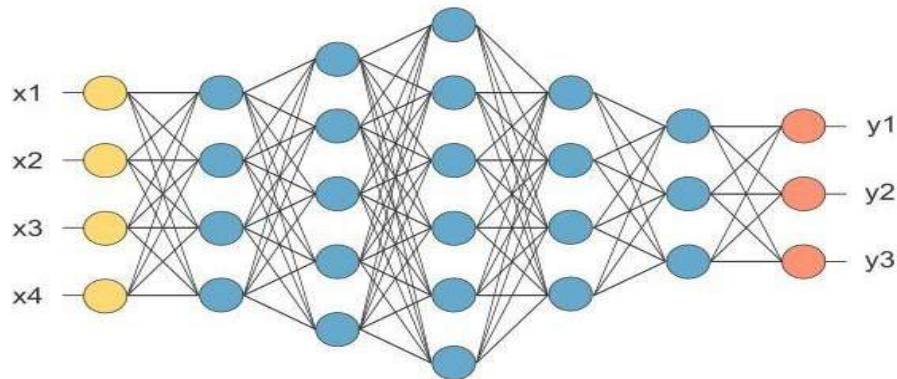


Figure 3.10: After pooling layer, flattened as FC layer [3]

As shown in the above figure 3.10 [3], feature map matrix will be transformed as vector ($x_1, x_2, x_3, \dots, x_N$). With the help of fully connected layers and features, a model is made. Finally, we have an activation function such as sigmoid to categorize the outputs as animal, pedestrian, car and truck etc. A complete CNN architecture is shown in figure 3.11 [3] from data collection to output layer.

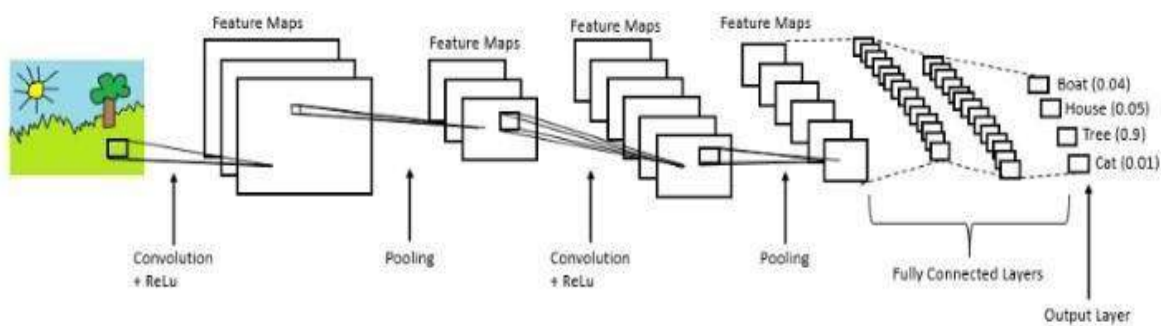


Figure 3.11: Complete CNN architecture [3]

3.5 Conclusion

In this chapter, an overview of machine learning, neural networks and convolutional neural network was presented. Due to a number of hidden layers, a convolutional neural network has the ability to learn and predict large classes of images, which otherwise is extremely difficult to do using shallow neural network.

Chapter 4

HARDWARE

This chapter explains the hardware design of prototype of a self-driving car. For making a prototype, an assembly of toy car has been used. The assembly has two DC motors, one for moving the rear wheel and other for steering control. All the other instruments are removed from the assembly and a motor driver is fitted to drive the both DC motors along with Arduino for command functioning. A variable resistor is also added on the steering which is further connected with Arduino and LCD display for measuring and display of steering angles. Hardware components used in this project are as follows:

- Toy Car
- Rechargeable Battery
- Laptop as Main Processor
- H-bridge Motor driver
- Arduino Mega 2560
- LCD
- Webcam
- Training tracks

The trained CNN model after prediction generates a string and through serial communication the string is sent to Arduino. Finally, the Arduino processes the wrappers embedded in its code according to the string received from the trained model and sends control signals to the motors drivers, which further controls the motors of the toy car to move or stop according to prediction of trained model. Figure 4.1[4] shows the block diagram of hardware which explained in this chapter.

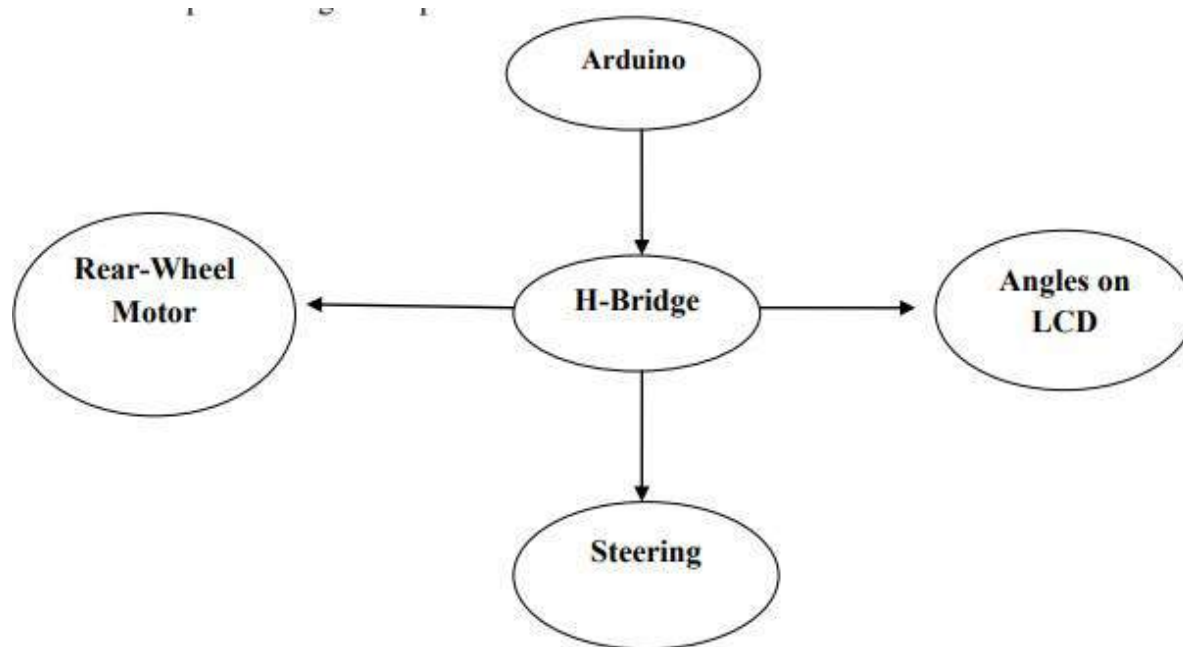


Figure 4.1: Flow chart of Hardware working [4]

4.1 Toy Car

We have used a toy car in our project since as a prototype self-driving car. It has two main motors, one is for steering control and the second motor is rear wheel motor. It has a rechargeable battery for its power source. All the other instruments that the toy car came up with, are removed and assembled with Arduino, motor driver and variable resistor on the steering which is further connected with Arduino and LCD display for measuring and display of steering angles.

4.1.1. Motor

Toy Car DC motor is used for controlling steering and rear wheel with assembly. It takes 12V DC voltage from battery and current of 1.6A and it has a resistance of 3 Ohms. Since, the values of current and voltage are known so we can calculate its power that is approximately 20 Watts. A motor with assembly is shown in figure 4.2 [14].

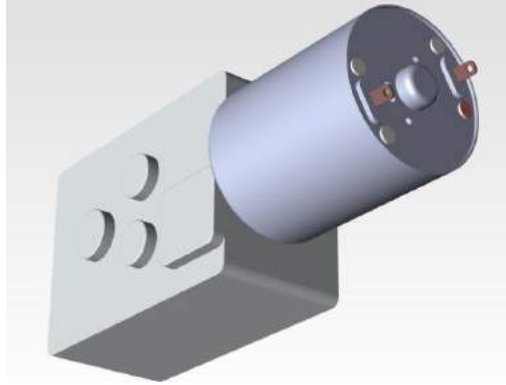


Figure 4.2: DC motor with assembly [14]

Second motor which is used to move steering is also a DC motor. It takes 12V and a current of 1.2 Ampere and has a resistance of 5 Ohms. It gives power of approximately 14 watts. For driving these motors, 2 H-bridge motor driver has been used which we will discuss later.

4.2 Rechargeable Battery

The car consists of a rechargeable battery of 12 volts which has a rating of 2.3 AH. It has 12 volts output and can supply a maximum of 4A current with initial current of 0.69A. To recharge the battery we are using a smart charger. Use of these batteries are efficient, saves time and good for environment as Rechargeable batteries produce less waste because they can be recharged with a simple battery charger and reused hundreds of times. Battery used is shown in figure 4.3 [14].



Figure 4.3: Diagram of Rechargeable battery [14]

4.3 H-Bridge Motor driver

An H-bridge is an electronic circuit that reverse the polarity of an applied voltage to a load [15]. H-bridges are usually available in the form of integrated circuits. An H-bridge motor driver can be built with four switches (solid state or mechanical) [15]. Only two switches are closed at a time and two switches are in open state. The positive voltage as well as the negative voltage can be applied to allow motors to operate in forward or backward direction. A bipolar stepper motor is invariably driven by a motor controller containing two H-bridges. The perspective view of L298N is shown below in the figure 4.4 [15].

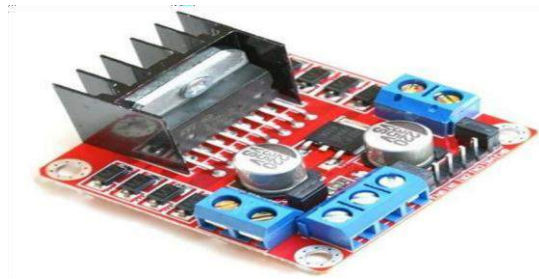


Figure 4.4: Perspective view of L298N [15]

4.3.1. H-Bridge L298N Schematic

The schematic diagram of L298N IC is shown in the Figure 4.5 [4]. L298N is connected to a micro-controller to trigger the different terminals of stepper motor through the H-bridge.

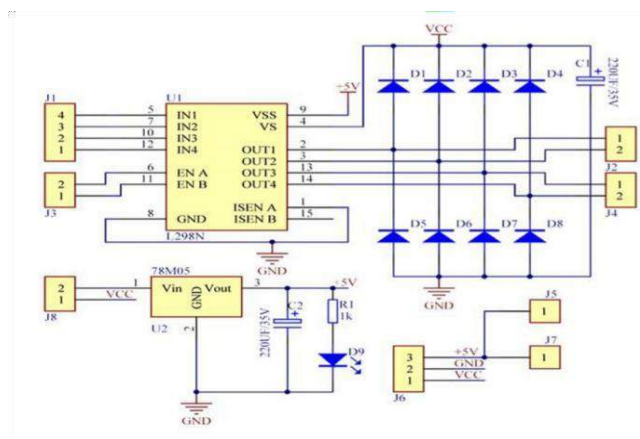


Figure 4.5: Schematic Diagram of L298N [4]

4.3.2. H-Bridge Pin Configuration

The pin configuration of H-bridge is shown in Figure 4.6 [4] illustrating different components used in the H-bridge IC.

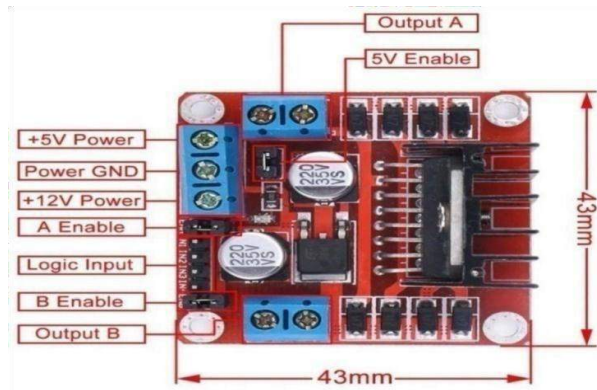


Figure 4.6: Pin Diagram of L298 [4]

4.4 Arduino Mega 2560

The Arduino Mega 2560 is a micro-controller board (shown in figure 4.7 [4]). It has 54 digital input and output pins. The 15 pins can be used as PWM output, 16 analog input pins, 4 UARTs (hardware serial ports pins), a 16 MHz crystal oscillator, a USB connection, a power jack, an ISCP header and a reset button [19]. We give a supply of 5V to “Arduino” by using the Arduino power cable.



Figure 4.7: Arduino mega 2560 [4]

We have used Arduino to calculate the steering angle, to control the DC motors with H-bridge motor driver and for serial communication with python code.

4.4.1 Steering Angle

Steering angle is calculated using an Arduino board with the help of variable resistor. Direct connection is established between potentiometer and steering motor such that when the steering motor operates, it causes a change in potential in the variable resistance. This change is calculated in terms of steering angle by making a program for its conversion from voltage to angle. After calculating angle it can be easily displayed on an LCD shown in figure 4.8 [4].

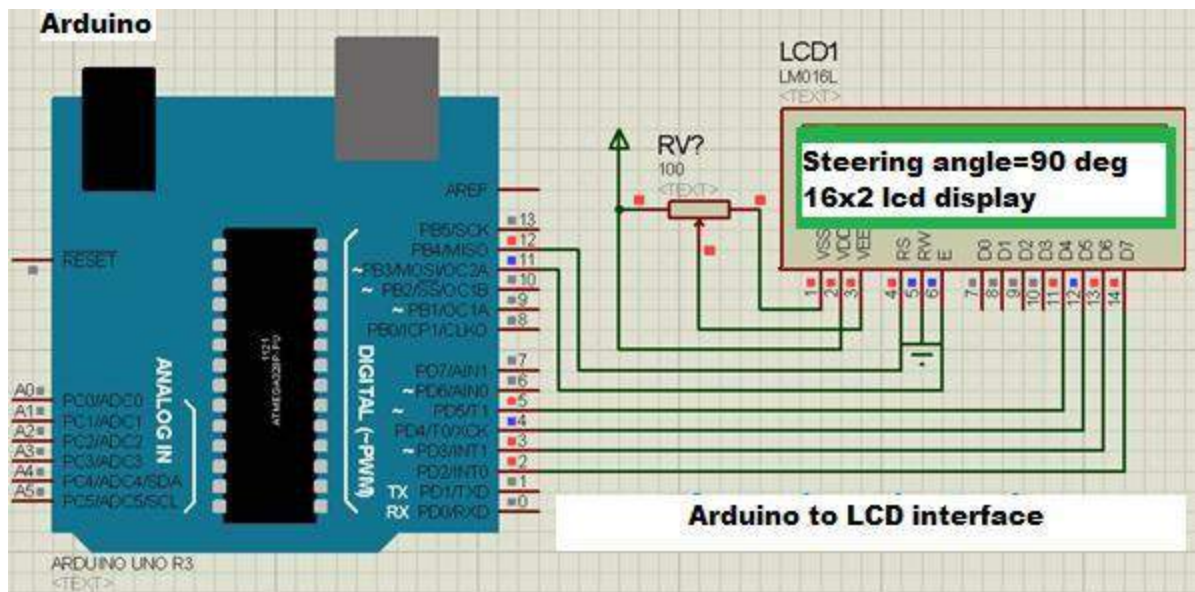


Figure 4.8: LCD displaying steering angle [4]

4.4.2 Motors Controlling

Speed of motor can be controlled by using Arduino Mega 2560. For controlling the speed, the input voltage is varied using a PWM signal which is given to the H-bridge motor driver. When switches 1 and 4 are closed and 2 and 3 are open then the voltage flows from the supply to the motor through motor driver module and the motor will rotate in positive direction and vice versa. Working of motor controlling is shown in figure 4.9 [4] below.

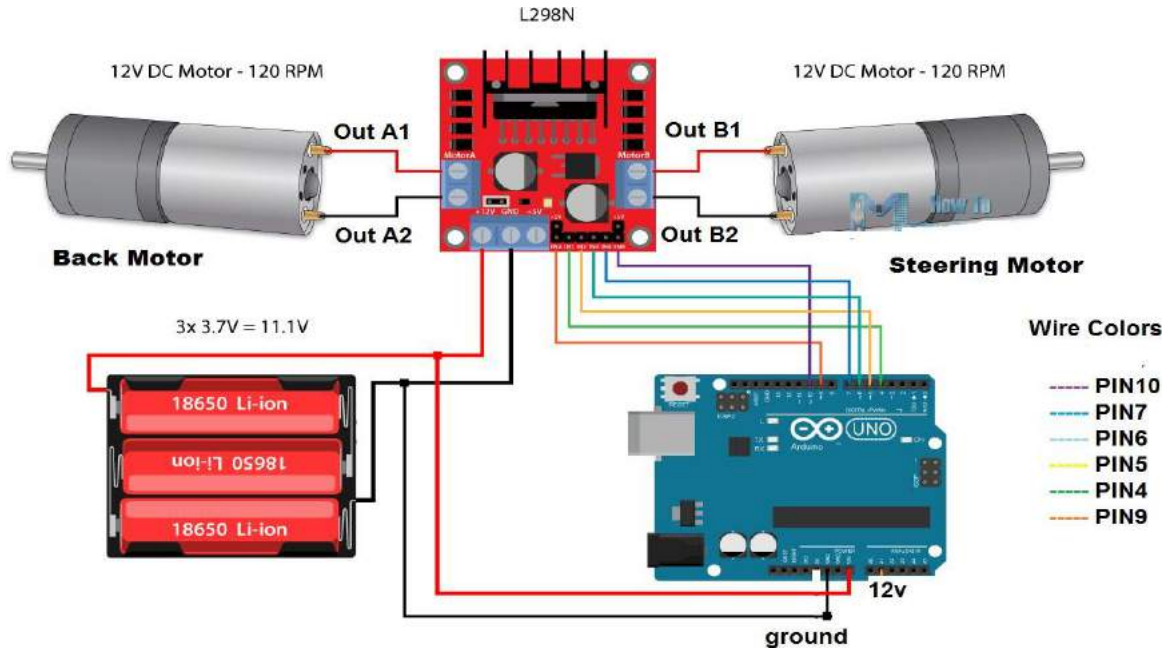


Figure 4.9: Motor Controlling [4]

4.4.3 Serial Communication

For interfacing with the image processing python code running on a laptop, we have used “Serial Communication”. By this we can interface between python code and Arduino code. In our project, we are using a signal from python code to Arduino; after receiving signal, the arduino decides to perform an action accordingly. For example, in our case, the python code has to decide the line path using a webcam, it will predict depending upon the track and then that signal will pass to Arduino through serial communication [17].

4.5 LCD

Liquid crystal display is a screen with electronic display module and a wide range of applications which can display 16 characters per row and there are 2 columns i.e. 16 x 2 [16]. In LCD each character is displayed in size of 5x7 pixel matrix. It consists of mainly 2 registers command and data LCD is shown in figure 4.10 [4].

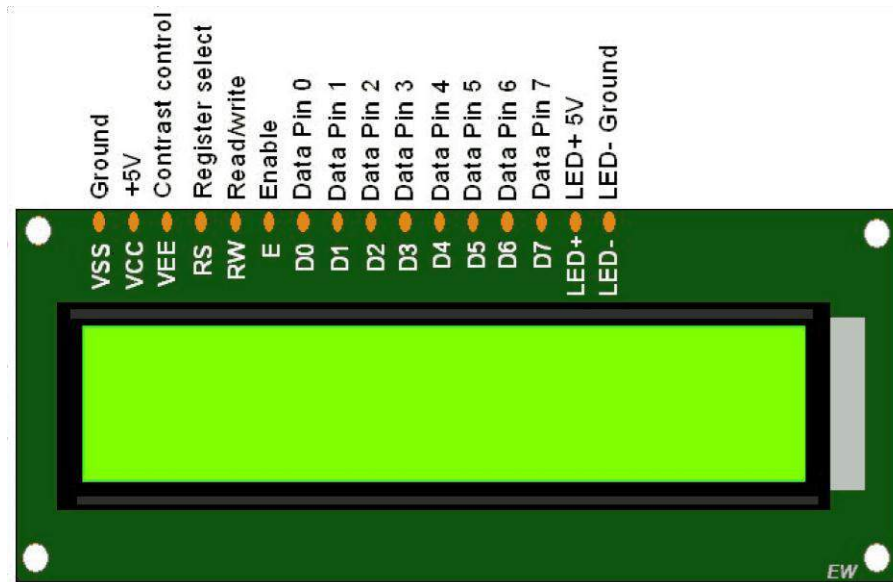


Figure 4.10: Liquid Crystal Display [4]

LCD has different pins as we can see in its diagram there are 8 data pins from D0 to D7 and other pins are to control its contrast and display visibility.

4.6 Laptop

We are using a laptop as main processor instead of Raspberry Pi due to its limited processing speed.

4.7 Webcam

We have used a webcam of Logitech C930e as shown in figure 4.11. It supports H 264 with scalable video coding and UVC 1.5 encoding to minimize its dependence on computer and network resources. It captures video at 1080p HD quality. Webcam is used for making data set for its training. It is used to capture pictures of track and also used for capturing video in real time. The webcam is shown in figure 4.11 [4].



Figure 4.11: Webcam [4]

4.8 Tracks

We have used different tracks for data sets and training models some of them are shown in figure 4.12 (a) and (c). With the help of webcam, images of track from prototype of self-driving car are captured to be trained using convolutional neural networks.

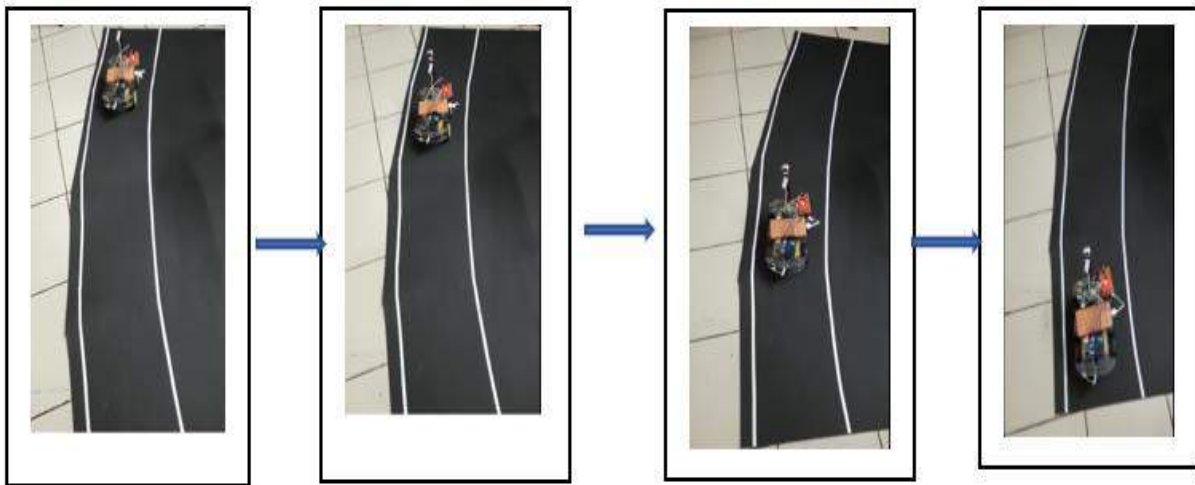


Figure 4.12(a): Track 1

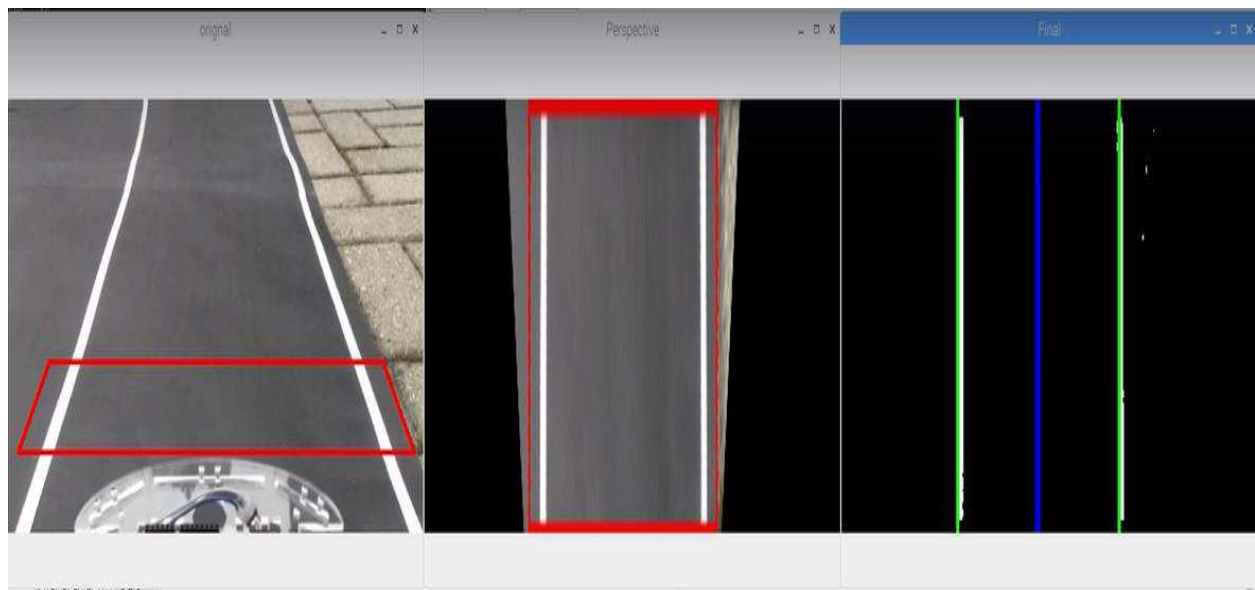


Fig 4.12 (c): Track 3

Conclusion

In this chapter we have discussed about hardware components and their working. How hardware is integrated with software part.

Chapter Five

Image Processing and Deep Learning Techniques used in the Project

Overview

This chapter is about working of prototype of the self-driving car used in this project. The car selected is a small toy car in which a 5 years kid can sit and play. The main ingredients of this project are computer vision techniques and convolutional neural networks, which are explained in this chapter. In addition, data collection, data pre-processing, classification of data, training CNN model for gathered data and testing on pre-defined paths are discussed. Figure 5.1 [5] shows the basic block diagram of self-driving car control system. In this block diagram, self-driving car control system is the main laptop processor. This processor enables processing of all actions of self-driving car. All steps involved in maneuvering of self-driving car are directly linked with control system. And all of these sections are linked with each other through control system. Control system is the main core of complete system which controls all actions performed using computer-vision, deep-learning, path-planning and localization as shown in fig 5.1

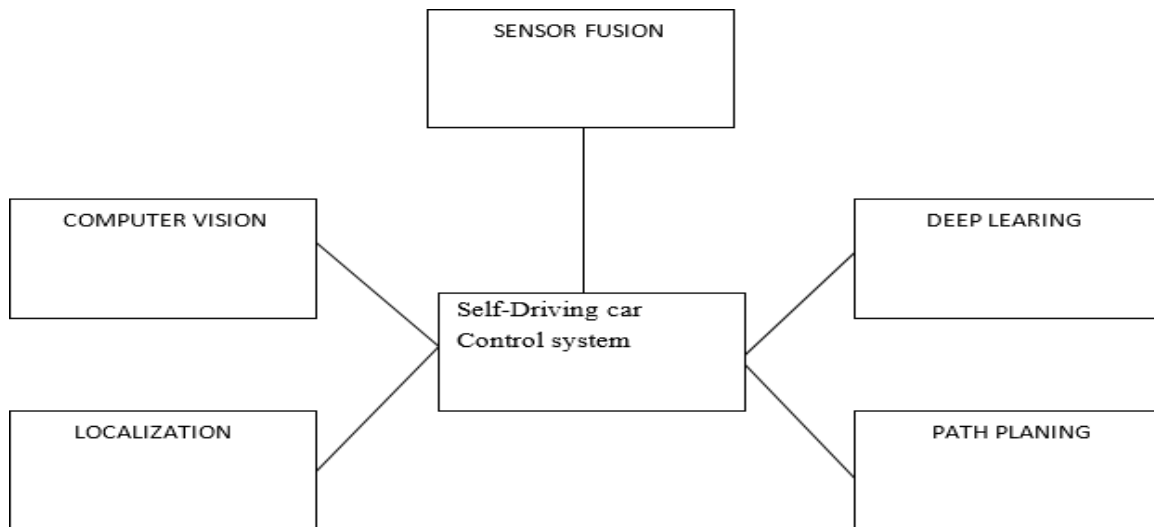


Figure 5.1: Concept of Self Driving Car Block [5]

5.1 Data Collection:

Data for our project are the images of the track that is deployed on the surface on which the self-driving car has to run. Tracks are established using small pieces of PVC pipes as a boundary and of white chart paper in middle of the track to segment it into two lanes. After that we placed our toy car on the start of the track with laptop placed on it and webcam mounted on the front bonnet of the toy car. For data collection we move the toy car on the track using Bluetooth control HC-05, which is a Bluetooth module controlled with the help of Arduino and commands of right, left and forward was given using an android app. To operate the webcam, we have used OpenCV running on a laptop to capture the images of the track as shown in figure 5.2.

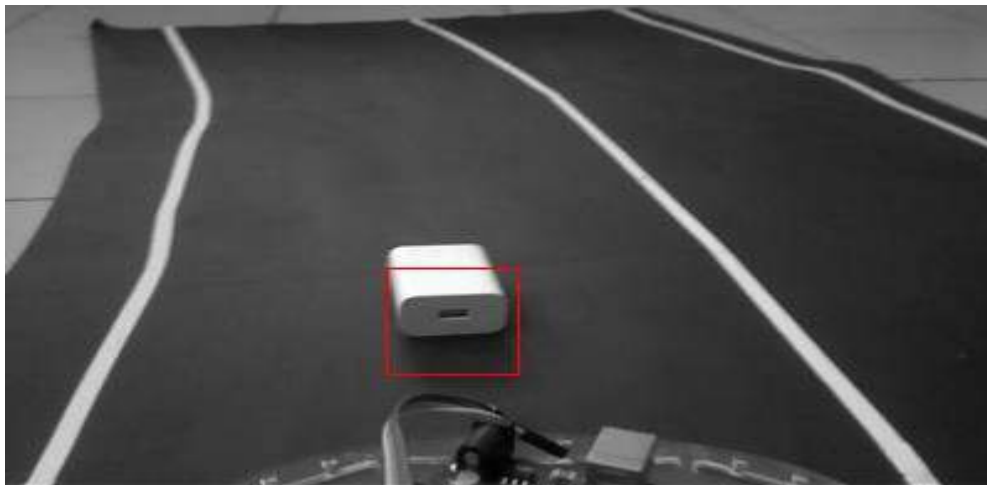


Figure 5.2: Prototype of self-driving car on track

5.1.1 Classification

After the collection of video, the videos are converted into the frames using OpenCV. After conversion of videos to frames, the collected data in form of frames are classified using supervised learning before processing of data. Collected data is classified into three classes of forward, left and right. Data collection is vital in order to train the model as accurate as possible. All classes forward, right and left must have equal data in number and normalized in magnitude.

5.2 Operations for Finding Lane Lines

Before feeding the data to neural network model it is pre-processed using OpenCV to find lane lines for self-driving car. There are number of image processing operations required to tune for finding lane lines as follows:

- Gray scale conversion
- Gaussian Blurring
- Canny edge detection
- Region of interest
- Bitwise AND operation
- Hough Line Transform
- Display lane lines

5.2.1 Grayscale

As data collected is in the RGB format, so processing of RGB data requires more computational power increasing complexity of the algorithm. Processing gray scale image is easier than RGB image because RGB image has three planes one for each red, green and blue along with different pixel intensities for each plane at same position. OpenCV's `CV_RGB2GRAY` conversion is given below:

$$\text{Gray_scale_value} = 0.30 \cdot \text{R_value} + 0.59 \cdot \text{G_value} + 0.11 \cdot \text{B_value}$$

Figure 5.3 shows image after converting it to gray scale image from the RGB.



Figure 5.3: Grayscale

5.2.2 Gaussian Blurring

This technique is used to reduce noise and for smoothing of image. Reducing noise and smoothing is done by Gaussian filter. Blurring is necessary before canny edge detection otherwise edge would not be detected causing noise points detection as well. Gaussian blur is applied on the gray scale image. The formula of a Gaussian function in one dimension is

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

In two dimensions, it is the product of two such Gaussian functions, one in each dimension:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The OpenCV command is : Blur=
cv2.Gaussianblur(gray.(5,5),0)

Using 5*5 kernel on our gray image, Figure 5.4 shows the Gaussian blur image which has less noise than figure 5.3.

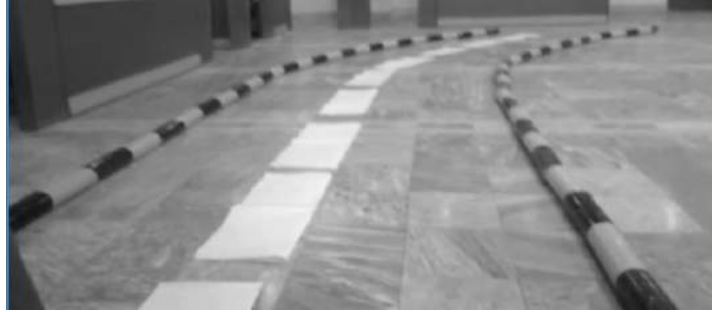


Figure 5.4: Gaussian blur

5.2.3 Canny Edge Detection

This technique is used for detecting edges in our image. Canny edge detection method is applied on blurred image. Blurred image is fed to canny edge detection so as to ensure accurate edges detection with no noise factor as shown in figure 5.5. This edge detection technique is precisely applied to easily find region of interest in images. Minimum and maximum thresholding applied ranges between 50 and 150, but it can be varied between 0-255 as per requirement. The result of Canny Edge Detection is shown in Fig. 5.5

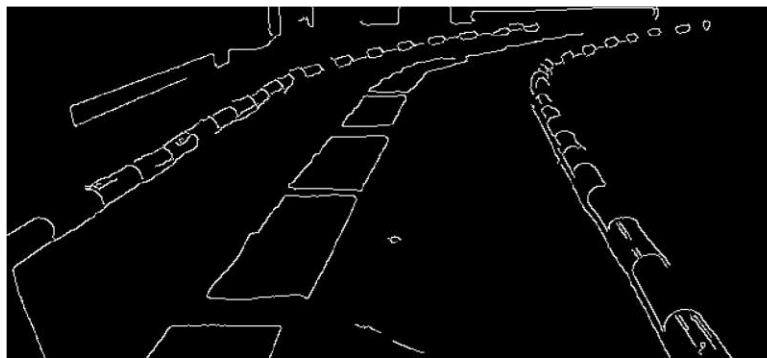


Figure 5.5 Canny Edge Detection

5.2.4 Region of Interest

Canny edge detected images are given as an input for locating region of interest. From classified data, different images of different classes are plotted using matplotlib library as shown in figure 5.6. Coordinates for the polygon are extracted through these plotted images for right, left and

forward classes. These coordinates are then embedded into the code in order to find the region of interest.



Figure 5.6: Region of Interest

5.2.5 Bitwise AND Operator

This technique is used to calculate the per-element bit-wise conjunction of two arrays or an array and a scalar. It is specifically used to only show the specific portion of the image and everything else masked. The OpenCV function used is shown below:

5.2.6 Hough Transform

Hough transform is a feature extraction technique used in image processing. The purpose of this technique is to find the points on the image with same threshold and create lines on them as in figure 5.7. This technique has ability to join the points having gaps on the image and same features.

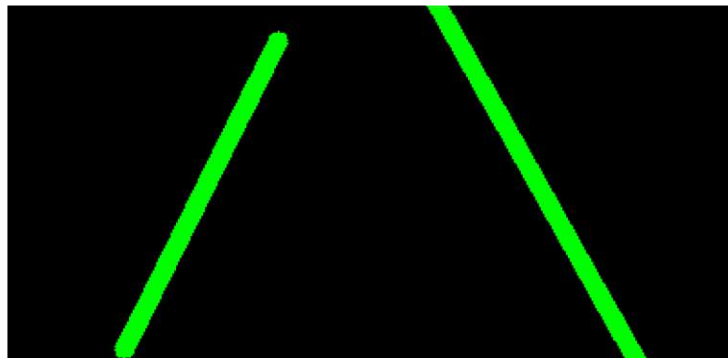


Figure 5.7 Hough lines

5.2.7 Display lines

Lines are displayed on the images using OpenCV. Initially the Hough lines are drawn on a zero pixel image using bitwise AND operator inside the region of interest shown in figure 5.8 (a) [10]. Weights of Hough line transform image and the real image of the track are added. By adding weights of these images, the Hough lines are displayed as shown in figure 5.8(b). These displayed lines are then averaged according to their slopes and intercepts in order to display the lines in equal ratio.

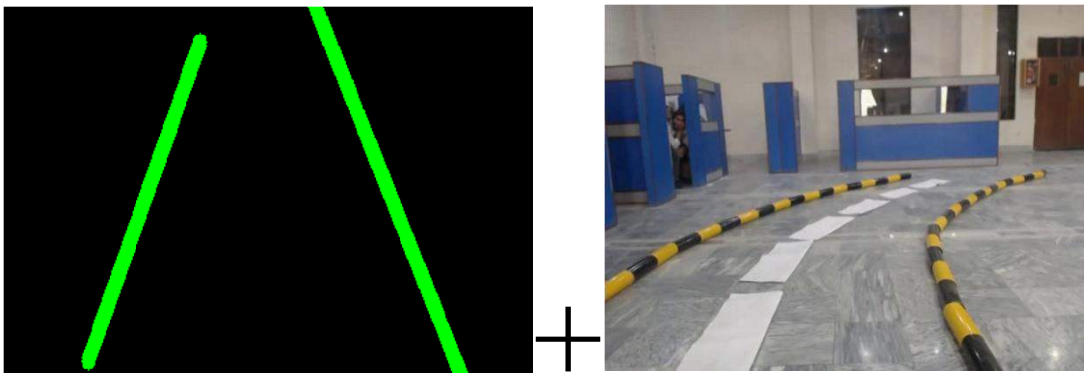


Figure 5.8(a): Display lines

Result of adding weight is shown below:

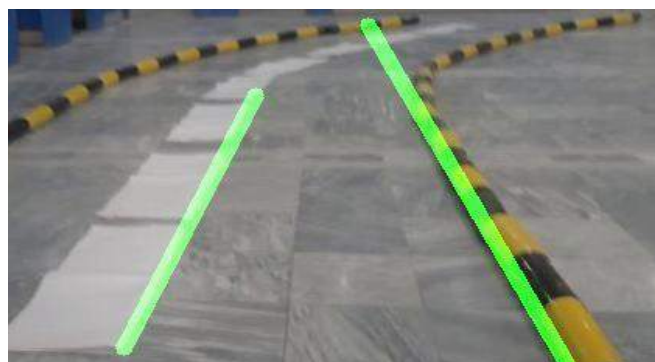


Figure 5.8(b): Display lines

5.3 CNN Architecture

Convolutional neural networks is a class of deep neural networks. Convolutional neural networks contains multilayer perceptron and are capable to understand complex patterns. A complete CNN architecture is shown in figure 5.9 [3]. It contains number of building blocks for creating neural networks model which are as follows:

- Data Pre-processing
- Input Layer
- Hidden Layers
- Output Layer
- Model Evaluation

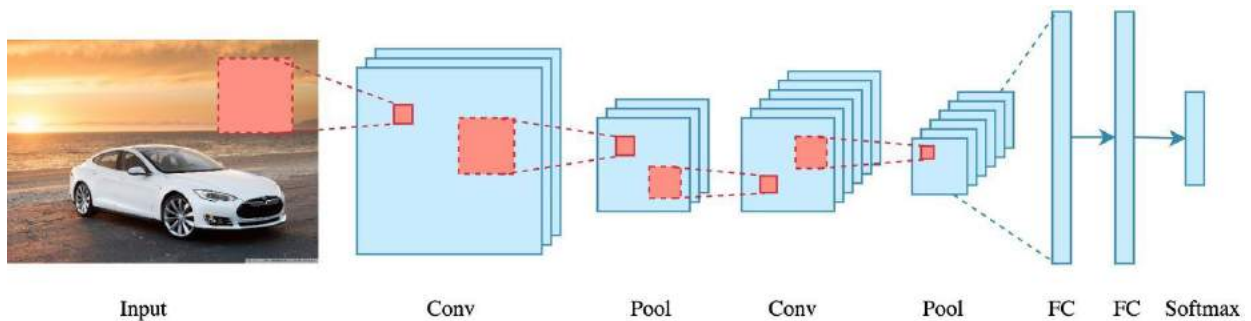


Figure 5.9: CNN Architecture [3]

5.3.1 Data Pre-processing

In convolutional neural networks architecture data pre-processing is one of the key components which is of a lot of significance in this project. Data pre-processing provides processed data to input layer of CNN architecture. Data pre-processing includes classified processed data for testing/training which is further classified into forward, left and right classes.

5.3.2 Input Layer

Input layer is the beginning layer of neural networks model. From this layer, processed data is fed into neural networks model. As input for input layer are image having dimensions width *height*depth. It is a matrix of pixel values.

5.3.3 Hidden Layers

Hidden layers are the ones with which the neural network gets deeper and deeper. These hidden layers includes convolutional layers, max pooling layers, dropout layers, flatten layers and dense or fully connected layers. Hidden layers are structured in such a way that for example, first two layers are convolutional layers, 3rd pooling layer, 4th and 5th are dropout and flatten layers, 6th and 7th are convolutional and pooling layers and finally from 8th to 13th layers are fully connected and dropout layers.

5.3.4 Output Layers

Output layers contains number of outputs which can be predicted with maximum probability. Output layer is connected to fully connected layer or Dense layer. Output layer has a lot of significance with respect to model prediction and model evaluation is based on the output of output layer.

5.3.5 Model Evaluation

Model evaluation is done on basis of model prediction from output layer. In model evaluation accuracy and loss of trained model can be measured. Even plots for accuracy/loss of trained model can be plotted against model validation.

5.4 CNN Hyper Parameters

Convolutional neural networks techniques are embedded in prototype of self-driving car.

Convolutional neural networks model used contains fifteen layers in it. The Hyper parameters which can be tuned in this model are as follows:

- Convolutional and Pooling Layer Parameters
- Add/Subtract and sequencing of convolutional and pooling layers
- Number of fully connected layers and their parameters
- Model compilation parameters
- Model evaluation parameters

5.4.1 Convolutional and Pooling Layers Parameters

Convolutional and max pooling layers have number of parameters which are tuned. Convolutional layer requires number of perceptron, input image size, filter kernel size and activation function. Now for max pooling layer, these layers requires stride size and padding, etc. All of these parameters can be tuned in order achieve higher accuracy of model avoiding overfitting.

5.4.2 Add/ Subtract and Sequencing of Convolutional and Pooling Layers

Layer can be added or subtracted as per requirement of model evaluation in order to achieve higher accuracy and avoiding overfitting. If the model is showing 100% accuracy then it means it is over-fitted then number of subsequent layer can be subtracted else. If a model is under-fitted so subsequent layer can be added to the model. Layers are sequenced as per requirement of model first convolutional and max pooling layers and then dully connected layer as this subsequent model contains.

5.4.3 Number of Fully Connected Layers and their Parameters

Number of fully connected layers are as per requirements of model which are managed after model evaluation. Fully connected layers which connect input and output layers to subsequent model hidden layers. Fully connected layers have number of parameters which can be tuned are number of perceptron and activation function. These parameters can be varied for achieving better results with higher accuracy avoiding over-fitting.

5.4.4 Model Compilation Parameters

Model compilation is one of the most significant step in model training. Model compilation requires number of parameters, is selected as Adam with low learning of 0.001, and Mean Square Error as the Loss function. Learning rates can be adjusted, optimizer and loss functions can also be changed as per requirement of model after evaluation of model if required.

5.4.5 Model Evaluation Parameters

Model evaluation requires model fitting with which training session initiates. Model evaluation requires training data, steps per epochs and epochs. Steps per epochs and number of epochs can be varied by observing accuracy rate of model after training session is completed.

5.5 Training of data

Training data used in our project is about 70 percent of the complete data. Supervised learning is used for training of the data. This data is classified and labeled as Right, left and forward. This data is trained using CNN sequential model. CNN model used for the training of the data contains 15 hidden layers. These layers include dense layer, convolutional-2D layer, max pooling-2D layer, flatten layer and fully connected layers. CNN is used for extracting the features from the images and learn through these features by updating the bias and weights of the perceptron. Categorical cross entropy with Adam optimizer and a learning rate of 0.001 is used in this model.

5.6 Testing using trained model

A training data set consists of inputs each input corresponds to some label to train the model. The CNN learns from this data, update the bias and weights accordingly and come up with a trained model. Test images are the new data that the neural network has never seen before. The difference between test image and training data is that the training data is used to train the neural network which keeps on updating until the error is minimize. So when a neural network model is introduced to test image which it has never seen before, the model should be able to classify correctly.

5.7 Serial Communication

Interfacing of Arduino with python code running on laptop is done with the help of “Serial Communication”. The trained CNN model after prediction generates a string and through serial communication the string is sent to Arduino. So, according to the prediction made by CNN model the Arduino coded function is called which moves the car according to the prediction of model. Arduino is further connected to motor drivers which moves accordingly. With the movement of steering motor the attached variable resistor also moves which is further connected to Arduino and LCD to display the steering angle.

5.8 Conclusion

This chapter is about working of prototype of a self-driving car with detailed explanation. In this chapter, data collection, its pre-processing, hyper parameters, CNN architecture, model training/testing and serial communication, etc., is discussed.

Chapter Six

Software

Software executes a program performing some specific tasks as a set of instructions, data or program to operate computers. All new technologies require software because it reduces human efforts and has capability to perform millions of tasks with just few lines of code. It can perform multitasking and nowadays with the advancement in technology, software has made man capable of controlling anything from anywhere in world by just sitting at one place with just a click of a button. In this chapter, the main software modules used in the project are illustrated along with description.

6.1 Overview

Development of prototype of a self-driving car is a big project. This project requires a lot computation power and techniques such as deep learning, computer vision, etc. Different OpenCV algorithms for preprocessing of data i.e. data collection (videos), video to frames extraction, plotting of frames for region of interest, classifying data, frames to video conversion are used. After the pre-processing of data, model is trained up to certain epochs for achieving higher accuracy at very low learning rate. The basic block diagram of training is shown in figure 6.1. Gradient Descent algorithm further lowers cross-entropy value in order to achieve higher accuracy. Weights and biases of perceptron are adjusted using back propagation techniques as the model trains up to some epochs. In order to overcome over-fitting which was present in our subsequent model, certain techniques were implemented which would be discussed in model training section. After training model with higher accuracy, complete algorithm is tested on the prototype of self-driving car. Testing ensures us whether training of data is correct or not, is model facing the issue of over-fitting, either model is to be adjusted or data is to be collected properly or it may require any other changes. At the end of training phase, the trained model is implemented on prototype of self-driving car for monitoring whether the car is operating according to model prediction or it has some errors. String is generated as a result of model prediction. One more algorithm is used which

works as a bridge between preprocessing of data, model training/ testing, model prediction which is written in python and the Arduino microcontroller algorithm which contains wrapper of code to operate according to prediction of model using Pyserial communication. Arduino Mega microcontroller is used to operate the car in forward, backward or left direction according to prediction of model. The block diagram of training phase is shown in figure 6.1.

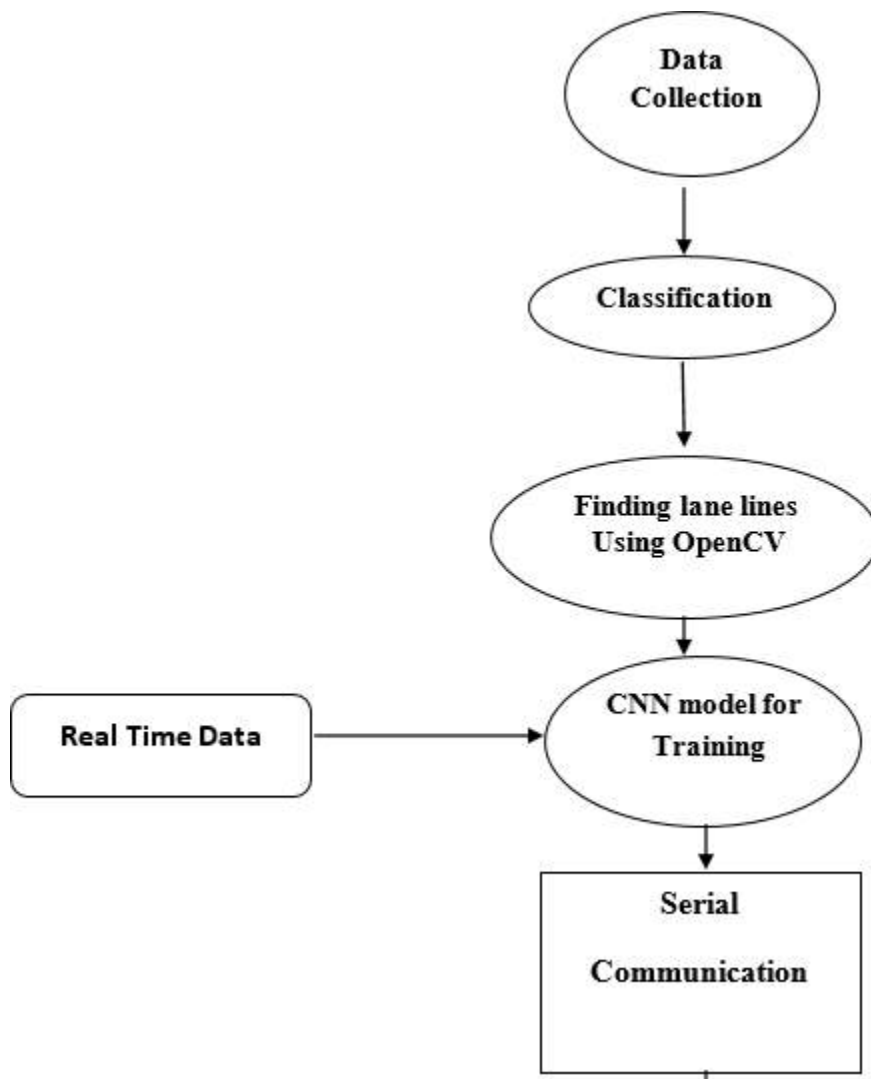


Figure 6.1: Block Diagram Training phase

6.2 Computer Vision

Computer vision is a discipline, which is concerned in building artificial intelligence systems in order to obtain information from images or multidimensional data. With this new emerging technology machines are enabled to see things as humans can in real world and it makes machine capable of making decision on its own. Computer vision strong algorithms and its modules are being used by millions of people worldwide for the sake of building new application which could make life of people easier and safer. Computer vision has many domains and people can use it in any field of interest, like self-driving cars, virtual reality, cyber security and business, etc. As this is about prototype of self-driving car, computer vision has a lot of significance in this project. Computer vision robust algorithms are used in this project as follows:

- Gray scale conversion
- Gaussian blurring
- Canny edge detection
- Hough line transform
- Optimization

6.2.1 Video Capturing and Saving using OpenCV

In this section, openCV is used in order to capture and save video using webcam. Video is captured using the OpenCV library. The capturing and saving of video is done at the same time. Video capturing can be processed using local PC webcam by placing an index zero in video capturing command or it can also be processed with external webcam using an index one in video capturing. Now for saving video at the same time, video writer is used which requires parameters such as video name and its type of format in which it is to be saved, frames per second, and video dimensions as per requirement. Both of these processes initiate when captured video is read, divided into frames and finally processed i.e. flipping of frames before writing video into desired format. Video capturing is controlled using waitkey, with which any key on the keyboard can be specified for ending session. The complete OpenCV code for video capturing is shown in exhibit 1.

```

import cv2

cap = cv2.VideoCapture(0)

# Define the codec and create VideoWriter object
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output134.avi',fourcc, 20.0, (640,480))

while(cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
        frame = cv2.flip(frame,180)

        # write the flipped frame
        out.write(frame)

        cv2.imshow('frame',frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break

# Release everything if job is finished
cap.release()
out.release()
cv2.destroyAllWindows()

```

Exhibit 1. OpenCv code for Video Capturing.

6.2.2 Video to Frames Conversion using OpenCV

Using this OpenCV algorithm defined below, frames from video can be extracted. Video in any format i.e. MP4, AVI, FLV etc. can be broken down into frames in any format i.e. JPEG, PNG, GIF and BMP as per user requirement. So, in order to perform this operation, video is captured using OpenCV first. Afterwards, captured video is read to be converted into frames. Finally, these frames are saved in format as specified by the user in program using image writing in OpenCV. The OpenCV code for converting video to frames is shown in exhibit 2.

```

import cv2
vidcap = cv2.VideoCapture('output132.avi')
success,image = vidcap.read()
count = 0
success = True
while success:
    success,image = vidcap.read()
    cv2.imwrite("frame%d.jpg" % count, image)    # save frame as JPEG file
    if cv2.waitKey(10) == 27:                    # exit if Escape is hit
        break
    count += 1

```

Exhibit 2.code for converting video to frames and saving as jpeg file.

6.2.3 Image Reading using OpenCV

Image and video reading is very similar, it only needs path of folder correctly specified. The only difference in reading images and videos is very little but significant. Reading images requires path of its folder in different ways i.e. when both images and code are not in same folder so copy and paste complete path of its folder from search window but with double back slash or easier way is to arrange both images and code in same folder. Now in case of reading videos, path of its folder is specified in different ways i.e. when both video and code are not in same folder so path of video folder is specified with single forward slash or if both are in same folder so code only requires name of video and its format to be read easily. The OpenCV code for reading frames is shown in exhibit 3.

```

import cv2
import numpy as np
p="C:\\Users\\Ahmad\\Desktop\\desktop data\\keras-tutorial\\general programs\\frame1.jpg"
image=cv2.imread(p,1)
lane_image=np.copy(image)
cv2.imshow('path',image)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Exhibit 3.code for reading frames.

6.2.4 Images to Video Conversion

Images to video conversion is as easy as video to images conversion. This process is also done using OpenCV. Initially, it requires the exact location of folder in which images are placed. Glob module is used in this algorithm in order to find all matching path names which specifies pattern according to rules used by the UNIX shell. Afterwards, image are read in loop and appended in single array. Finally, video is written and saved using video writer OpenCV module. It requires video name, frames per second as per requirement, size of frames as measured earlier using width and height of frames in order to write a new video from frames in any format. The complete OpenCV code for images to video conversion is shown in exhibit 4.

```
import cv2
import numpy as np
import glob

img_array = []
for filename in glob.glob('C:/Users/Ahmad/Desktop/desktop data
                           /keras-tutorial/pmo job/blob detection/task5 field/pblue/*.jpg'):
    img = cv2.imread(filename)
    height, width, layers = img.shape
    size = (width,height)
    img_array.append(img)

out = cv2.VideoWriter('project2.avi',cv2.VideoWriter_fourcc(*'DIVX'), 15, size)

for i in range(len(img_array)):
    out.write(img_array[i])
out.release()
```

Exhibit 4.code for images to video conversion.

6.2.5 Importing Libraries

These are the necessary libraries imported as required. These libraries are most commonly used for running computer vision algorithms, multi-dimensional data/information and matplotlib for plotting raw images in order to find region of interest. The OpenCV code for importing libraries is shown in exhibit 5.

```
In [1]: import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Exhibit 5.code for importing libraries.

6.2.6 Function Defined for Preprocessing of Data

In this canny function which takes an image as an argument is performing three contrasting operations. Initially, raw image received as an argument is converted into gray scale. As gray scale image contains noise which is filtered out using Gaussian blurring. This Gaussian blurring operation smoothens image at the edges and reduces noise using a Kernel size 5x5 in this algorithm as per requirement. Gaussian blurring is the prerequisite of canny edge detection, if smoothing filtered not applied before detection then noise can alter solution because canny edge detection detects edges where pixel value are changed more abruptly. In this function minimum level of 50 and maximum level of 150 is defined for edge detection as per requirement. The pixel values ranges between 50 and 150 are detected. The code for preprocessing of data is shown in exhibit 6.

```
In [2]: def canny(image):
gray=cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blur=cv2.GaussianBlur(gray, (5,5),0)
canny_edge=cv2.Canny(blur,50,150)
return canny_edge
```

Exhibit 6.code for preprocessing of data.

6.2.7 Function Defined for Region of Interest

This function is defined in order to find region of interest. As it also take an image as an input argument. Triangular shaped polygon is created as ROI and its x, y coordinates are defined. Before creating polygon, raw image is plotted using matplotlib so as to properly indicate x, y coordinates of points used in polygon. Mask of a zero pixel image is created and polygon is filled on mask using Polly filled computer vision technique. Finally, bitwise AND operator which is another strong computer vision technique is deployed in order to add Polly filled mask and input image properly indicating ROI. The complete OpenCV code for Region of Interest is shown in exhibit 7.

```
In [3]: def region_of_interest(image):
        #height=image.shape[0]
        polygons=np.array([
            [(20,400),(630,360),(450,50)]
            ],np.int32)
        # polygon1=np.array([
        #     [(260,90),(400,90),(400,35),(260,35)]
        #     ],np.int32)
        mask=np.zeros_like(image)
        # mask1=np.ones_like(image)
        cv2.fillPoly(mask,polygons,255)
        # cv2.fillPoly(mask1,polygon1,0)
        masked_image=cv2.bitwise_and(image,mask)
        # mk=np.copy(mask_image)
        # masked_image=cv2.bitwise_and(mk,mask1)
        return masked_image
```

Exhibit 7.code for Region of Interest.

6.2.8 Function Defined for Displaying Hough Lines Transform

Hough line transform is one of the most powerful computer vision techniques. In this function Hough line are displayed on images indicating path. This function takes an image and Hough lines as input arguments. A mask of zeros is created which is equal in size to input image. For drawing Hough lines on zero pixel image, initially these lines are reshaped in to x, y points. These x, y points are then drawn on the zero pixel image. Hough lines transform requires x, y points of both the lines, RGB color and width of line etc. The code for displaying hough lines transform is shown in exhibit 8.

```
In [4]: def display_lines(image,H_lines):
        l_image=np.zeros_like(image)
        if H_lines is not None:
            for line in H_lines:
                print(line)
                x1,y1,x2,y2=line.reshape(4)
                cv2.line(l_image, (x1,y1), (x2,y2), (0,255,0),10)
        return l_image
```

Exhibit 8.code for displaying hough lines transform.

6.2.9 Averaging of Slopes and Intercepts of Hough Lines

Averaging of slopes and intercepts of Hough Lines is a necessary operation to be done to find slopes and intercepts of lines satisfying data points. As infinite number of lines can be drawn from a point in x, y plane and all of these lines have their particular slopes and intercepts. Having slopes and intercepts of all these lines, a line is drawn in Hough space which is the line of best fit. By averaging all slopes and intercepts a single slope and intercept are obtained, with these parameters a line is drawn satisfying complete data. Now to draw left lane line and right lane line. All points corresponding to negative slope refers to left lane line and all points corresponding to positive slope refers to right lane line. Finally, left and right lane line coordinates are plotted on each

processed frame and two lane lines are displayed on images of track. The code for averaging of slopes and intercepts of hough lines is shown in exhibit 9.

```
In [6]: def average_slope_intercept(image,H_lines):
        left_fit=[]
        right_fit=[]
        for line in H_lines:
            x1,y1,x2,y2=line.reshape(4)
            parameters=np.polyfit((x1,x2),(y1,y2),1)
            # print(parameters[1])
            # print(line)
            slope=parameters[0]
            intercept=parameters[1]
            if slope<0:
                left_fit.append((slope,intercept))
            else:
                right_fit.append((slope,intercept))
        left_fit_average=np.average(left_fit,axis=0)
        right_fit_average=np.average(right_fit,axis=0)
        print(left_fit_average,'left')
        print(right_fit_average,'right')
        left_line=define_coordinates(image,left_fit_average)
        right_line=define_coordinates(image,right_fit_average)
        return np.array([left_line,right_line])
```

Exhibit 9.code for averaging of slopes and intercepts of hough lines.

6.2.10 Main of Computer Vision Section

In this section, main of computer vision is discussed in which all functions are collectively processed. These functions includes video capturing, video to frames conversion, gray scaling, blurring, canny edge detection and Hough lines transform etc. All of these computer vision

techniques are necessary for converting data into required format. The complete OpenCV code for main of computer vision section is shown in exhibit 10.

```
In [7]: cap=cv2.VideoCapture('project.avi')
size = (int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
        int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)))
fourcc = cv2.VideoWriter_fourcc(*'DIVX')
video = cv2.VideoWriter(r'p.avi', fourcc, 30, size)
while(cap.isOpened()):
    ret,frames=cap.read()
    canny_det=canny(frames)
    reg_of_int=region_of_interest(canny_det)
    H_lines=cv2.HoughLinesP(reg_of_int,2,np.pi/180,70,np.array([])
                           ,minLineLength=70,maxLineGap=50)
    average_lines=average_slope_intercept(frames,H_lines)
    line_image=display_lines(frames,average_lines)
    color_image=cv2.addWeighted(frames,0.8,line_image,1,1)
    if ret == True:
        video.write(line_image)
        cv2.imshow('frame',line_image)
    else:
        cap.release()
        break
    if cv2.waitKey(50) & 0xFF==ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Exhibit 10.code for main of computer vision section.

6.3 Convolutional Neural Networks

Convolutional neural networks is a class of deep neural networks and is used to analyze imagery. CNN contains multilayer perceptron referring to fully connected layers. These layers contains neurons and every neuron is connected to all neuron in next layer. Convolutional neural networks works in hierarchical pattern and it has capability to understand more complex patterns and solve these complex patterns using simpler patterns. It is so robust and peerless technique, with which any kind of complex data can be classified. CNN models trains itself up to certain epochs, by

adjusting weights and biases using back propagation technique for predicting with higher accuracy. If model over-fits, there are certain simpler techniques with which overfitting can be avoided i.e. reducing number of perceptron in respective layers, reducing learning rates and reducing number of epochs for training in order to adjust weights and biases etc.

6.3.1 Importing Necessary Packages

Training neural networks model for this prototype, these are necessary packages needed to be imported. Following packages are required to split testing and training data, running model in sequential pattern, labelling data, fully connected layers and plotting results of accuracy or loss vs validation etc. The code for importing necessary packages is shown in exhibit 11.

```
import matplotlib
matplotlib.use("Agg")

# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.models import Sequential
from keras.layers.core import Dense
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dropout, Activation, Flatten
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import random
import pickle
import cv2
import os
```

Exhibit 11. code for importing necessary packages.

6.3.2 Initialization and Sorting of Data

In this section, data and labels are initialized with an empty array for placing the data accordingly and labelling the data. Data and labels are sorted in such a way that images path is grabbed and shuffled randomly. These paths are listed and sorted for reading images in a sequential manner. The code for initialization and sorting data is shown in exhibit 12.

```
# initialize the data and labels
print("[INFO] loading images...")
data = []
labels = []

# grab the image paths and randomly shuffle them
imagePaths = sorted(list(paths.list_images('pics')))
random.seed(42)
random.shuffle(imagePaths)
```

Exhibit 12. code for initialization and sorting data.

6.3.3 Preprocessing of Data and Labels

In pre-processing of data, frames are obtained from their respective paths, resized and appended to data. These images are resized to reduce computational power of the system. If resizing is not done, then processing of larger sized images could take longer time effecting performance of system during prediction. The code for preprocessing of data and labels is shown in exhibit 13.

```

# loop over the input images
for imagePath in imagePaths:
    # load the image, resize the image to be 32x32 pixels (ignoring
    # aspect ratio), flatten the image into 32x32x3=3072 pixel image
    # into a list, and store the image in the data list
    image = cv2.imread(imagePath)
    image = cv2.resize(image, (64, 64))
    data.append(image)

    # extract the class label from the image path and update the
    # labels list
    label = imagePath.split(os.path.sep)[-2]
    labels.append(label)

```

Exhibit 13.code for initialization and sorting data

6.3.4 Scaling of Raw Image Pixel Intensities

Scaling of raw images pixel intensities is the procedure of normalizing pixel intensity values. Scaling is done in order to create an array of images with pixel intensities between 0 and 255. Afterwards, labels are managed in a single Numpy array vis a vis data. As data and labels are in one dimensional array so data points are recognized by labels in array at same level in correspondence to each other. The code for scaling of raw image pixel intensities is shown in exhibit 14.

```

data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {} images {:.2f}MB".format(
    len(imagePaths), data.nbytes / (1024 * 1000.0)))

```

Exhibit 14.code for initialization and sorting data.

6.3.5 Splitting Training and Testing Data

Testing/Training splitting of data is done using sklearn library. Testing data is usually 20% of total data and training data is about 80% of total data. Training model is validated using testing data in order to measure how accurately model is trained. But in this project 25% of total data is used as testing data and training data is about 75% of total data. The code for splitting training and testing data is shown in exhibit 15.

```
(trainX, testX, trainY, testY) = train_test_split(data,  
labels, test_size=0.25, random_state=42)
```

Exhibit 15. code for splitting training and testing data.

6.3.6 Binarizing Label for Training and Testing Data

Labels are binarized for testing and training. Training labels are binarized to fit model. And transforming testing labels is to normalize label between 0 and 1. Normalization transforms non-numerical labels to numerical labels. The code for binarizing label for training and testing data is shown in exhibit 16.

```
lb = LabelBinarizer()  
trainY = lb.fit_transform(trainY)  
testY = lb.transform(testY)
```

Exhibit 16. code for binarizing label for training and testing data.

6.3.7 CNN Model for Training of Data

Convolutional neural networks is used to train model. This model contains 15 layers and a sequential model training pattern is followed. In this model five convolutional layers are imported in which a number of parameters are required. Convolutional layers require number of perceptron, kernel size, subsample, input image dimensions and activation function etc. This model contains

dense and dropout layers. Dense layers are the fully connected layers used in model for processing and connecting model to output layer of model. Dense layers requires number of perceptron and activation function etc. For compilation of model, Adam optimizer is applied at low learning rate of 0.001. Mean squared error loss function and matrices for accuracy are also used for compilation of model. The code for CNN model for training of data is shown in exhibit 17.

```
model = Sequential()
model.add(Convolution2D(24, 5, 5, subsample=(2, 2),
input_shape=(66, 200, 3), activation='relu'))
model.add(Convolution2D(36, 5, 5, subsample=(2, 2),
activation='relu'))
model.add(Convolution2D(48, 5, 5, subsample=(2, 2),
activation='relu'))
model.add(Convolution2D(64, 3, 3, activation='relu'))

model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(Dropout(0.5))
model.add(Flatten())

model.add(Dense(100, activation = 'relu'))
    model.add(Dropout(0.5))

model.add(Dense(50, activation = 'relu'))
    model.add(Dropout(0.5))

model.add(Dense(10, activation = 'relu'))
    model.add(Dropout(0.5))

model.add(Dense(1))

optimizer = Adam(lr=1e-3)
model.compile(loss='mse', optimizer=optimizer)
return model
```

Exhibit 17.code for CNN model for training of data.

6.3.8 Model Summary

Model is summarized by validation against loss and accuracy level of training and testing data. Model summarization requires fitting of model specifying number of parameters. Model fitting requires training and testing data, steps per epoch, number of epochs, verbose and shuffling of data accordingly. Steps per epochs means number of data points normalized by number of epochs specified. Number of epochs means how many times data is passed through model and trained by adjusting weights and biases of perceptron using back propagation. The code for model summary is shown in exhibit 18.

```
model = nvidia_model()
print(model.summary())
history = model.fit_generator(batch_generator(X_train,
                                             y_train, 100, 1),
                             steps_per_epoch=300,
                             epochs=10,
                             validation_data=
                                batch_generator(X_valid, y_valid, 100, 0),
                             validation_steps=200,
                             verbose=1,
                             shuffle = 1)
```

Exhibit 18.code for model summary.

6.3.9 Model Evaluation

Model evaluation is also necessary procedure and very easily be done. An integral part of model deployment is model evaluation. Model evaluation helps us to find best model which could represent our data and way it performs in future. There are several ways of model evaluation likewise Hold-Out and Cross-Validation. This model uses cross validation method and evaluates the predicted data for different types of predictions i.e. forward, left or right. The code for model evaluation is shown in exhibit 19.

```

print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=32)
print(classification_report(testY.argmax(axis=1),
                             predictions.argmax(axis=1), target_names=lb.classes_))

```

Exhibit 19.code for model evaluation.

6.3.10 Plot for Training Accuracy and Loss

Plots are created and validated for loss and accuracy against validation data. Validation plots are displayed for training and testing data accuracy/loss functions in order to ensure any changes required in model. Model changes with respect model overfitting/under-fitting and to increase trained model accuracy by varying a number of parameters. The code for plot for training accuracy and loss is shown in exhibit 20.

```

N = np.arange(0, EPOCHS)
plt.style.use("ggplot")
plt.figure()
plt.plot(N, H.history["loss"], label="train_loss")
plt.plot(N, H.history["val_loss"], label="val_loss")
plt.plot(N, H.history["acc"], label="train_acc")
plt.plot(N, H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy (Simple NN)")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.savefig('output/plot.png')

```

Exhibit 20.code for plot for training accuracy and loss.

6.3.11 Model Saving and Label Binarization to Disk

Trained model is saved to be imported and to be used in program at run time. Trained model is saved along with its binaries labels. Such that when they are called in another python program so they can perform accordingly. The code for model saving and label binarization to disk is shown in exhibit 21.

```
print("[INFO] serializing network and label binarizer...")
model.save('output/simple_nn.h5')
f = open('output', "wb")
f.write(pickle.dumps(lb))
f.close()
```

Exhibit 21. code for model saving and label binarization to disk.

6.4 Serial Communication between Python and Arduino

Serial communication is one of the most important key feature of this project. Serial communication is necessarily be very important ensuring sending of data correctly according to prediction of trained model. Serial communication is accomplished for transferring the generated string by the prediction of trained model such that car has to move either forward, left or straight. This is actually communication between python and C++ Arduino languages in string format.

6.4.1 Importing Libraries

Necessary libraries are imported in serial communication code. In this program a number of tasks are being performed by processor. Along with all the necessary libraries, trained model is also imported with help of which this algorithm would be capable of predicting according to trained model. The code for importing libraries is shown in exhibit 22.

```

from keras.models import load_model
import pickle
import cv2
import time
import serial

```

Exhibit 22.code for importing libraries.

6.4.2 Arduino Port Detection

Part of this algorithm is capable of detecting Arduino port with which serial communication is to be initialized. For serial communication, baud rate of 9600 is defined. Establishment of serial communication requires wait or time delay which is defined to be 2 seconds in code. The code for Arduino port detection is shown in exhibit 23.

```

#board=ArduinoMega('COM3')
ArduinoSerial = serial.Serial('COM7',9600)
#Create Serial port object called arduinoSerialData
time.sleep(2)
#wait for 2 seconds for the communication to get established

```

Exhibit 23.code for Arduino port detection.

6.4.3 Loading Trained Model for Pickling

Trained model imported is loaded in this section of algorithm. Afterwards pickles are imported and loaded for the sake of serializing data such that they can be saved and utilized when required. In this algorithm, trained model is pickled as it is loaded. The code for loading trained model for pickling is shown in exhibit 24.

```

cap = cv2.VideoCapture(1)
print("[INFO] loading network and label binarizer...")
model = load_model('output/simple_nn.h5')
lb = pickle.loads(open('output/simple_nn_lb.pickle', "rb").read())

```

Exhibit 24.code for loading trained model for pickling.

6.4.4 Prediction from Different Track Images

For prediction from different track images, frames captured are pre-processed before prediction. Pre-processing includes resizing, normalization of frames pixel intensities, classifying images of different types, finally loading trained model and prediction according to trained model either the car wants to move in forward, left or right direction. The code for prediction from different track images is shown in exhibit 25.

```

while(True):
    # var = input() #get input from user
    # print ("you entered", var) #print the input for confirmation

    _, image = cap.read()
    #image = cv2.imread('pics/straight/opencv_frame_6.png')
    output = image.copy()
    image = cv2.resize(image, (64,64))

    # scale the pixel values to [0, 1]
    image = image.astype("float") / 255.0

    # dimension
    # check to see if we should flatten the image and add a batch

    # otherwise, we must be working with a CNN -- don't flatten the
    # image, simply add the batch dimension
    image = image.reshape((1, image.shape[0], image.shape[1],
                           image.shape[2]))

    # load the model and label binarizer

    # make a prediction on the image
    preds = model.predict(image)
    k = model.predict_proba(image)

```

Exhibit 25.code for prediction from different track images.

6.4.5 Finding Labels According to Prediction

After label binarization, class labels of respective frames with largest probability are found. The labels of these classes are printed on command prompt in order to be verified by the prediction of trained model. These predictions are printed on frames to see as well as to observe car's movement according to prediction of trained model. The code for finding labels according to prediction is shown in exhibit 26.

```
# find the class label index with the largest corresponding
# probability
i = preds.argmax(axis=1)[0]
label = lb.classes_[i]
print(label)
```

Exhibit 26. code for finding labels according to prediction.

6.4.6 Labels for Multiple Directions

Multiple paths have been with assigned multiple labels. So to be differentiated from each other. Labels assigned are in string format, which are encoded and saved for particular class of data. Multiple labels i.e. forward, left, and right are assigned different code words in string format as described. These encoded string elements are sent using serial communication to Arduino and same code word in string format are assigned to different Arduino functions for execution of certain tasks. The code for labels for multiple directions is shown in exhibit 27.

```
if (label == 'straight'):
    ArduinoSerial.write('70a'.encode())

if (label == 'left'):
    ArduinoSerial.write('76a'.encode())
```

```

if (label == 'right'):
    ArduinoSerial.write('82a'.encode())

if (label == 'unknown_path'):
    ArduinoSerial.write('89a'.encode())

```

Exhibit 27.code for labels for multiple direction.

6.4.7 Writing Text Data and Labels on Frames

Writing text data and labels on frames is necessary for verifying prediction of trained model and movement of car such that how accurately it propagates. This procedure is accomplished to draw class labels and output probability of prediction in terms of accuracy. As laptop is on roof of car so labels printing on it is seen very clearly. Writing text on images is very easy using OpenCV specifying parameters such as output images, text data, RGB color, writing format and depth of color etc. The code for writing text data and labels on frames is shown in exhibit 28.

```

if (label == 'stop'):
    ArduinoSerial.write('89a'.encode()) #send 0
    #print (ArduinoSerial.readline())
    #print ("LED turned OFF")
    #time.sleep(1)
    # draw the class label + probability on the output image
    text = "{:}. {:.2f}%".format(label, preds[0][i] * 100)
    cv2.putText(output, text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
                (0, 0, 255), 2)

    # show the output image
    cv2.imshow("Image", output)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()

```

Exhibit 28.code for writing text data and labels on frames.

6.5 Arduino Controller Working According to Prediction

Arduino controlling and working section is so efficient and serene. Arduino operations are controlled by trained model prediction. As trained model predicts, so Arduino has to perform accordingly. Same string codes are set in both region side by side, firstly on Arduino functions and secondly string type labels encoded for different classes of images. Arduino functionality includes calculating steering angle, displaying steering angles on the LCD display and functions defined for right, left and straight to whom has to work depending on prediction of trained model. Trained model generates a string according to prediction of data fed into trained model as input. That particular string as a command is sent to Arduino using Pyserial communication.

6.5.1 Initialization

In this section, Arduino controlling, Arduino pins along with LCD display are initialized globally. LCD display requires 6 pins to be initialized from which 4 data pins and 2 reading and writing pins are initialized. Other pins and variables are initialized for angle calculation, H-Bridge L298N pins enabling and I/O's etc. Another important variable initialized is speed controlling of car which is controlled using PWM signal. PWM signal is generated by scaling up and down between 0-255 which controls speed of car. The arduino code for initialization is shown in exhibit 29.

```
#include<LiquidCrystal.h>
LiquidCrystal lcd(22, 23, 24, 25, 26, 27);

const int IN1 = 9;
const int IN2 = 8;
const int ENA1 = 7;
const int IN3 = 4;
const int IN4 = 5;
const int ENA2 = 6;
int command = 0;
float adc_val = 0.0;
float vout = 0.0;
int angle = 0;
char speed = 180;
int a=0;
```

Exhibit 29.code for initialization.

6.5.2 Controlling Setup

For prototype of a self-driving car controlling and one time execution of functions with LCD display in terms of Baud rate and LCD type e.g. 16, 2 is done using setup function. Setup function of Arduino is one of its Built-in function, which is used for one time execution and initialization of code during the whole processing of code. In this section, motor pins are enabled and initialized. Afterwards Serial communication with Baud rate of 9600 is initiated along with LCD display. The arduino code for controlling setup is shown in exhibit 30.

```
void setup() {  
    m1_init();  
    m2_init();  
    Serial.begin(9600);  
    lcd.begin(16, 2);  
}
```

Exhibit 30.code for controlling setup.

6.5.3 LCD Display and Motors Controlling

LCD display and motor controlling are two distinguished functions. LCD display is only for displaying steering angles of car which is an independent function while motor controlling is a dependent function. Motor controlling is dependent on prediction of trained model. As Arduino receives a string type data/code, then it operates according to prediction and respective functions are called according to prediction from trained model.

6.5.4 Initializing Serial Communication and Angle Calculation

Angles calculation and serial communication is initiated in this section of code. For angles calculation analog values from potentiometer are received. These analog values are digitized using analog to digital conversion

The complete arduino code for LCD display and motors controlling is shown in exhibit 31.

```
void loop() {
  adc_val = analogRead(A0);
  vout = adc_val * (5.0 / 1023.0);
  angle = vout * (180.0 / 5.0);
  lcd.setCursor(0, 0);
  lcd.print("angle");
  lcd.setCursor(0, 1);
  lcd.print(angle);
  Serial.println(angle);
  if (Serial.available()) {
    String command = Serial.readStringUntil('a');

else if (command=="76")
{
  digitalWrite(IN3, HIGH);
  delay(500);
  digitalWrite(IN3, LOW);
  delay(100);
  digitalWrite(IN4, HIGH);
  delay(250);
  digitalWrite(IN4, LOW);
  delay(100);
  forward();
}

else if (command=="82")
{
  digitalWrite(IN4, HIGH);
  delay(500);
  digitalWrite(IN4, LOW);
  delay(100);
  digitalWrite(IN3, HIGH);
  delay(250);
  digitalWrite(IN3, LOW);
  delay(100);
  forward();
}
```



```

else if(command=="89")
{
digitalWrite(IN4, HIGH);
delay(100);
digitalWrite(IN4, LOW);
delay(10);
digitalWrite(IN3, HIGH);
delay(100);
digitalWrite(IN3, LOW);
delay(10);
digitalWrite(IN1, HIGH);
delay(100);
digitalWrite(IN1, LOW);
delay(10);
}

void m1_init()
{
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);
    pinMode(ENA1, OUTPUT);
}
void m2_init()
{
    pinMode(IN3, OUTPUT);
    pinMode(IN4, OUTPUT);
    pinMode(ENA2, OUTPUT);
}
void stop1()
{
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, LOW);
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, LOW);
}

```

```

void forward()
{
  analogWrite(ENA2, speed);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);
  analogWrite(ENA1, speed);
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  //Serial.println("straight");
}

void right()

{
  analogWrite(ENA2, speed);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  analogWrite(ENA1, speed);
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  //Serial.println("right");
}

void left()
{
  analogWrite(ENA2, speed);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENA1, speed);
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  //Serial.println("left");
}

```

Chapter 7

Conclusion

In this project, a toy car has been used for making a prototype self-driving car. The basic concept behind a self-driving car is to sense its environment and take actions accordingly. The steering angle of the car was measured using a potentiometer mounted on the steering wheel. Movement in the steering wheel caused change in the potentiometer voltage which was then digitized for subsequent processing. An LCD has been used to show the value of the steering angle. The motors used to drive the car were controlled using an H-bridge motor driver. The signal to the motor driver card was given through the Arduino board using PWM. The Arduino board gets the signal from a laptop which runs the computer vision algorithm to navigate the car. Convolutional Neural Networks (CNN) has been used to train data to manoeuvre the car on pre-defined paths. CNN model is deployed along with the powerful computer vision techniques to enable the car to navigate autonomously.

During various phases of the project, a number of tracks were prepared for the prototype self-driving car. Initially, a white track was used for training the neural network model. After training the model, its output was tested and its accuracy was not found to be good. Therefore, the initial track which was selected was changed. The track was changed by using colored PVC pipe (black and yellow color) for training the neural network model. This greatly helped to change the results. Convolutional neural networks along with Hough transform turned out to be best in terms of prediction accuracy. The hyperparameters of the CNN model were adjusted along with the architecture of the CNN to get the maximum accuracy. Numerous tracks were improved time to time according to their prediction in terms of validation accuracy. Finally black and yellow colored PVC proved to be the best.

A number of difficulties were faced, for example, processing speed limitation, variation in image brightness, etc. The processing speed was the main issue due to which the frame rate was not that high. To resolve this issue, the speed of the car was kept low. As brightness variations were not fully catered in this project, therefore results that were obtained indoor were better as compared to outdoor testing results. Further improvements can be made in this project by using GPU based high speed processing system and a high definition camera. In addition, computer vision techniques such as camera calibration, structure from motion, etc can have very profound impact on the accuracy of navigation.

8. REFERENCES

- [1] “Definition of self driving car” <http://www.techopedia.com>
- [2] “History of self driving car” <https://www.digitaltrends.com/cars/history-of-selfdriving-cars-milestones/>
- [3] “Self-driving course” <https://www.udemy.com/applied-deep-learningtm-the-complete-self-driving-car-course/>
- [4] “AutoML cloude” <https://cloud.google.com/automl/>
- [5] “GMcruse automation technology” <https://getcruise.com/>
- [6] “Seamless Autonomous Mobility (SAM)" system developed with NASA to realize a fully autonomous mobility.” <https://www.nissan-global.com/EN/TECHNOLOGY/OVERVIEW/sam.html>
- [7] “1.3 million people are killed each year by vehicles,” https://en.wikipedia.org/wiki/List_of_self-driving_car_fatalities, (Accessed on 201805-14).
- [8] “Since 2018 Waymo has done incredible work in deploying and testing vehicles in various domains and reached the highest running self-driving car i.e. running 10 million miles,” <https://deeplearning.mit.edu/>, (Accessed on 2018-05-14).
- [9] [Mariusz, Bojarski](#), [Ben Firner](#), [Beat Flepp](#), [Larry Jackel](#), [Urs Muller](#), [Karol Zieba](#) and [Davide Del Testa](#) “End-to-End Deep Learning for Self-Driving Cars” August 17,2016
- [10] Mochamad Vicky Ghani Aziz, Ary Setijadi Prihatmanto, Hilwadi Hindersah “Implementation of lane detection algorithm for self-driving car on toll road cipularang

using Python language” Published in: 2017 4th International Conference on Electric Vehicular Technology (ICEVT)

- [11] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Aleksandr Patsekin, Julia Kindelsberger, Li Ding, Sean Seaman, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, Bryan Reimer “MIT Autonomous Vehicle Technology Study: Large-Scale Deep Learning Based Analysis of Driver Behavior and Interaction with Automation” April 15, 2019
- [12] C. Chen, A. Seff, A. Kornhauser, J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving,” Proc. IEEE Int. Conf. Comput. Vis., pp. 2722-2730, 2015.
- [13] The Ruderman White Paper: “Self-Driving Cars: The Impact on People with Disabilities”
- [14] <https://components101.com/motors/toy-dc-motor>
- [15] “H-Bridge” www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-thebasics
- [16] “The Full Arduino Pin out Guide” <https://www.circuito.io/blog/arduino-uno-pinout>
- [17] “Serial communication using Pyserial” <https://www.pyserial.com>