VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

AI-POWERED AUTONOMOUS LANE NAVIGATION

Embedded Project

Submitted

By

G MUKKESH – 17BCE1128

AKSHAY KUMAR - 17BCE1290

ROHIT SUBRAMANIAN- 17BCE1291

Submitted

To

Dr. Rekha D

Associate Professor

School of Computing Science and Engineering

Robotics And its Applications – CSE3011

Winder Semester 2020

I. Abstract:

Driving is one of the most sophisticated tasks that a human can perform. It involves the coordination of many senses, and needs near perfect responsiveness to adapt to the changing environment. The concept of self-driving cars has been around since the 70s and 80s and it is truly being realized now thanks to the advancements in the fields of both Robotics and Artificial Intelligence. The recent surge in this field is mainly due to companies such as Tesla and WayMo leveraging the power of Deep Learning to help the cars make critical and human-like decisions on the road. Various sensors, starting from cameras to radars are used by these cars to (semi) autonomously navigate through traffic.

The main goal of this project is to try to recreate the Autonomous lane navigation technology present in these AI powered cars. We used traditional image processing workflows to manually identify lanes on the roads and calculate the corresponding steering angles required to navigate the same. Using this known data, we were able to build a custom Image based neural network which was able to entirely eliminate this workflow and replace it with around 97% accuracy.

II. Problem Statement:

To use Deep Learning to replace the traditional multi-algorithm Image Processing workflow needed to help the car navigate along a road (Lane Detection and Steering Angle Prediction) and develop a prototype of an Algorithm that can be used in Self-Driving cars.

III. Introduction:

The fields of AI and Robotics were strongly connected in the early days of AI (1960s and 70s) but had since diverged. Researchers initially worked with the goal of creating robots which can mimic human intelligence and perform full-fledged activities and take intelligent decisions on their own. This would need them to work on both AI and Robotics at the same time. However, the computing power of the hardware in that era was far from enough to actually make the AI algorithms work. Hence, these two fields slowly diverged and took different paths. The field of robotics moved away from relying on AI to propel its development and started using procedural and modular programming frameworks for the tasks of automation.

This trend came to be followed in vision-based robotic systems too. Historically, traditional Image Processing algorithms outperformed Machine Learning and Deep Learning algorithms till the late 90s. The AI algorithms were very primitive and due to the unavailability of powerful hardware, they were pretty much restricted to theoretical scenarios. Even though the Image Processing algorithms were nowhere close to achieving human-level accuracy, they were still the only algorithms to push forward

the vision-based robotic systems. They helped the robotic systems identify and segment objects in their field of vision. Hence, these algorithms were used in a wide array of robots (industrial and experimental). They were used in factories to automate mundane tasks such as quality management, and also helped exploration robots to identify targets and take actions accordingly. Even though these robots were not able to take intelligent decisions, there were still able to replace millions of human workforces for decades.

However, the scenario changed with the advent of GPUs and more powerful algorithms such as Backpropogation and Convolutional Neural Networks. Theoretical AI algorithms were now being implemented in real time, and were nearing human level accuracy on a wide variety of tasks, especially in the field of Computer Vision. The seminal AlexNet algorithm surpassed human level accuracy in labelling, segmentation and localization tasks in the ImageNet challenge. This prompted the return of focus on AI algorithms to enhance robot's decision-making skills. Subsequently in the following years, robots have greatly benefitted from Deep Learning based vision systems, which are far more powerful and efficient when contrasted with the prior systems. The emergence of actual self-driving cars stands as a testimony to this fact.

IV. Motivation:

The ImageNet challenge is one of the most revered competitions in the field of computer vision. Year after year, computer vision researchers worked very hard to reduce the error rates of their classification, localization and segmentation algorithms. Many breakthroughs in this field happened in this competition, and one of the most significant one was the AlexNet Neural Network. This Deep Learning algorithm, a type of Convolutional Neural Network, won the competition by a margin of 30% (being 30% more accurate than the 2nd best algorithm, a non-AI based one, in the competition). This triggered a wide spread adoption of Deep Learning into the field of computer vision and companies such as Tesla are completely relying on vision-based systems (not Lidar-based) to power their Autonomous vehicles.

Tesla is one of the Industry leaders in the field of AI-powered self-driving vehicles. They have custom computing hardware within their cars which are touted to perform better than the traditional GPUs found in computers. They also have 2 of these processors to act as a failsafe. Typical to the robotics framework, these cars also have sensors, processors, controllers and actuators. The sensors themselves are both external and internal. The internal sensors include various components such as battery level sensor, hardware checking sensor, GPS, etc. These are used to monitor the current state of the car's components. The external sensors include 8 surround cameras (Main forward Camera, Narrow Forward Camera, Wide Forward Camera, Forward Looking side Cameras and Rearward Looking Side Cameras) for visual input, 12 ultrasonic sensors complement this vision, allowing for detection

of both hard and soft objects and a forward-facing radar with enhanced processing provides additional data about the world on a redundant wavelength that is able to see through heavy rain, fog, dust and even the car ahead.

This motivated us to try and recreate a small part of their complex architecture, the autonomous lane detection and navigation system. We decided to pursue both the traditional and AI based approach to build this system, since we wanted to know the differences and similarities between them first hand.

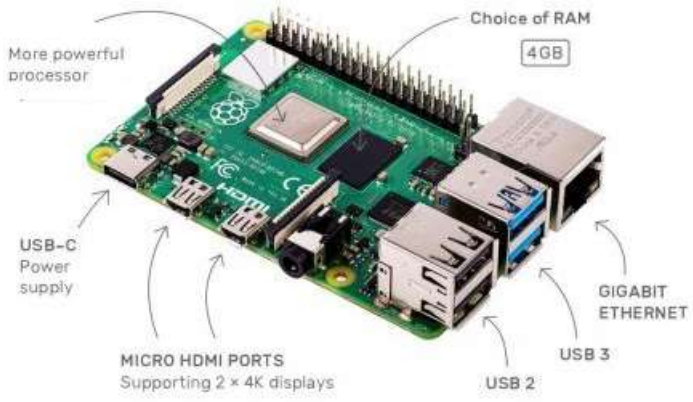
V. Basic Workflow:

We have two pipelines set up for this project. One follows the pre-AI workflow of passing the image sequentially into a number of Image Processing algorithms, and then getting the steering angle for that particular lane detected. The other uses this data to learn and just create a single neural network that can map the input image directly to the steering angle needed.

Both these systems can give their output to the car's actuator, which will make the car move accordingly (hardware interfaced).

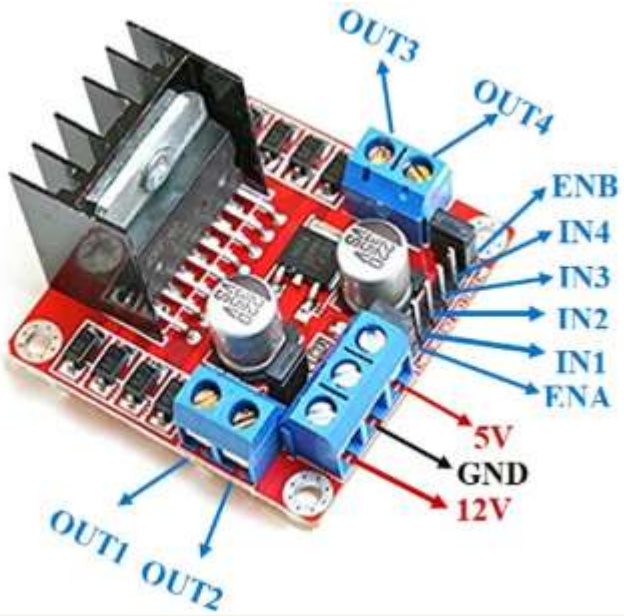
VI. Components:

1. *PI 4 Model B*



This the latest edition of the Raspberry Pi series, the Raspberry Pi 4B.
Raspberry Pi 4 Model B has a 1.5 GHz 64-bit quad core ARM Cortex-A72 processor, on-board 802.11ac Wi-Fi, Bluetooth 5, full gigabit Ethernet (throughput not limited), two USB 2.0 ports, two USB 3.0 ports, and dual-monitor support via a pair of micro HDMI (HDMI Type D) ports for up to 4K resolution . The Pi 4 is also powered via a USB-C port, enabling additional power to be provided to downstream peripherals, when used with an appropriate PSU.
The 4Gb RAM of the raspberry pi facilitates the integration and execution of AI and Deep learning algorithms with good performance.

2. *Motor Driver*



This L298N Motor Driver Module is a high-power motor driver module for driving DC and Stepper Motors. This module consists of an L298 motor driver IC and a 78M05 5V regulator. L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control.

Pin Name	Description
IN1 & IN2	Motor A input pins. Used to control the spinning direction of Motor A
IN3 & IN4	Motor B input pins. Used to control the spinning direction of Motor B
ENA	Enables PWM signal for Motor A
ENB	Enables PWM signal for Motor B
OUT1 & OUT2	Output pins of Motor A
OUT3 & OUT4	Output pins of Motor B
12V	12V input from DC power Source
5V	Supplies power for the switching logic circuitry inside L298N IC
GND	Ground pin

3. *Robot Chassis*

This chassis comes with encoder disc and dual shaft BO motors for developing various autonomous and manual controlled robot. Arduino compatible holes and standoffs are included so arduino can be easily mounted on top. Build Multiple types of Robots from one chassis. Has slots for all your general needs. Chassis size is 210 x 149mm. 4-Wheel Robot Chassis Kit, an easy to assemble and use robot chassis platform. The Chassis kit provides you with everything you need to give your robot a fast four wheel drive platform with plenty of room for expansion to add various sensors and controllers.

4. *Standard DC Motors*

Rate voltage: DC 3V-6V
Reduction ratio: 1:48
3V-170mA; 115RPM, 0.7KGf.cm torque

5. *Wheels*

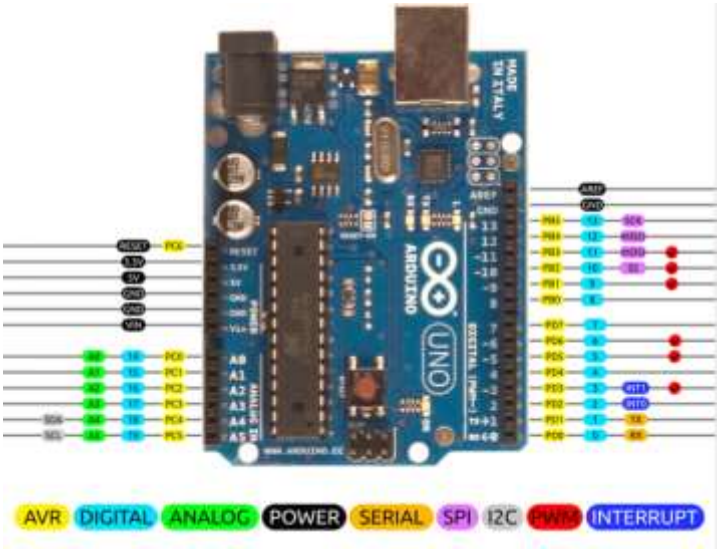
Material: Plastic
Size of the motor: 64.2x22.5x22.5mm(without the shaft)
Diameter of the tire: 67mm/2.6"
Width of the tire: 27mm/1.06"

6. *Pi Cam*

Second Generation Raspberry Pi Camera Module with Fixed Focus Lens
Sony Exmor IMX219 Sensor Capable of 4K30 1080P60 720P180 8MP Still
3280 (H) x 2464 (V) Active Pixel Count
Maximum of 1080P30 and 8MP Stills in Raspberry Pi Board
2A Power Supply Highly Recommended

7. *IP Cam (alternative to PiCam)*

8. *Arduino Uno*

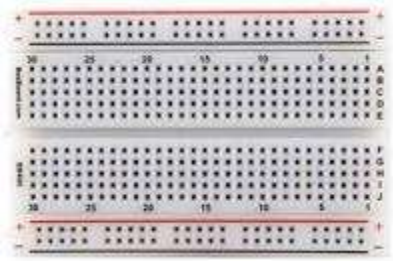


The Arduino board is use a microcontroller used to interface the Motor driver to control the motion of the autonomous car after receiving inputs from the Raspberry Pi through serial communication.

Pin Category	Pin Name	Details
Power	Vin, 3.3V, 5V, GND	Vin: Input voltage to Arduino when using an external power source.5V: Regulated power supply used to power microcontroller and other components on the board.3.3V: 3.3V supply generated by on-board voltage regulator. Maximum current draw is 50mA.GND: ground pins.
Reset	Reset	Resets the microcontroller.

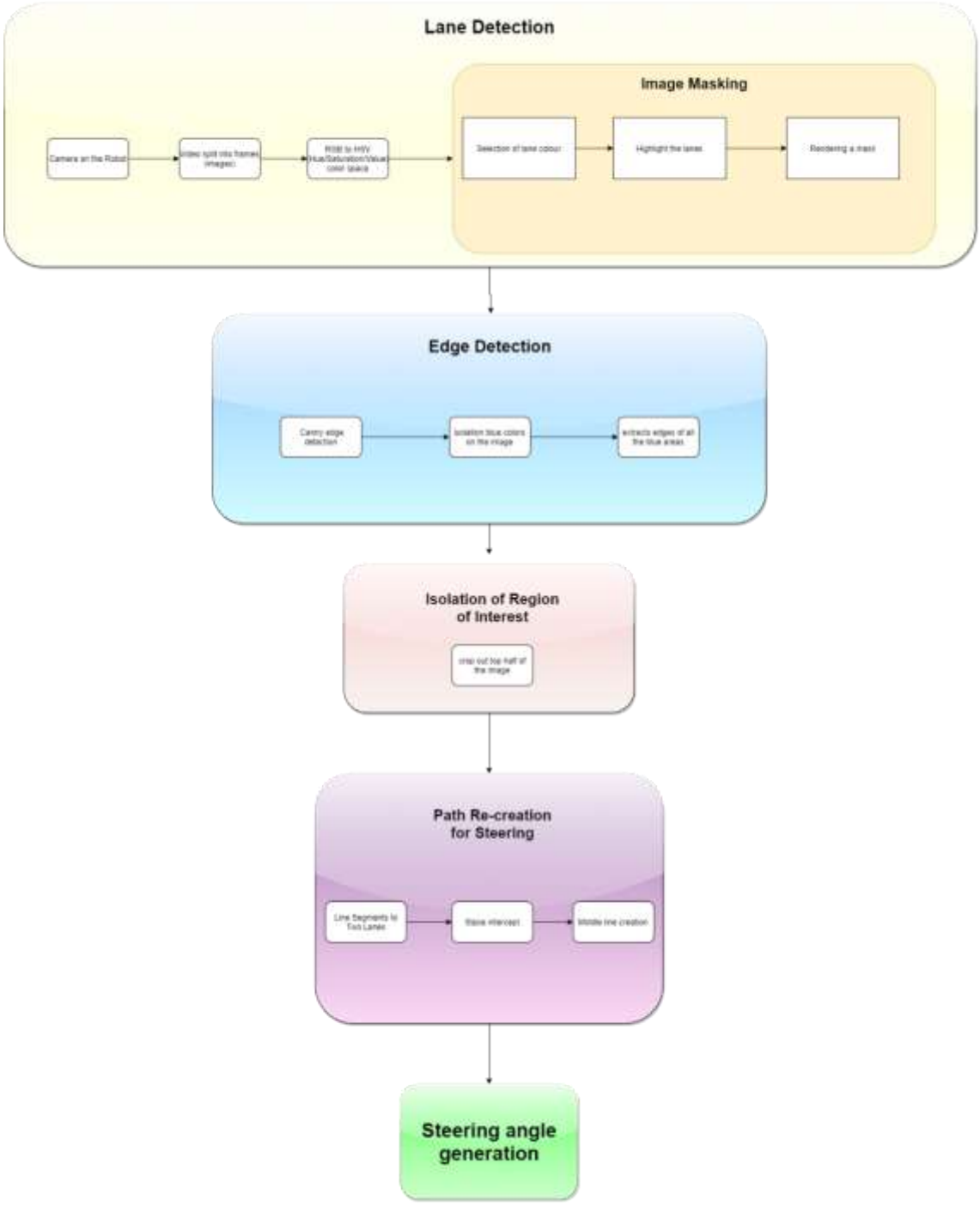
Analog Pins	A0 – A5	Used to provide analog input in the range of 0-5V
Input/Output Pins	Digital Pins 0 - 13	Can be used as input or output pins.
Serial	0(Rx), 1(Tx)	Used to receive and transmit TTL serial data.
External Interrupts	2, 3	To trigger an interrupt.
PWM	3, 5, 6, 9, 11	Provides 8-bit PWM output.
SPI	10 (SS), 11 (MOSI), 12 (MISO) and 13 (SCK)	Used for SPI communication.
Inbuilt LED	13	To turn on the inbuilt LED.
TWI	A4 (SDA), A5 (SCA)	Used for TWI communication.
AREF	AREF	To provide reference voltage for input voltage.

9. *BreadBoard*



Breadboards are used to help you connect components to complete your basic circuit. These holes let you easily insert electronic components to prototype (meaning to build and test an early version of) an electronic circuit, like this one with a battery, switch, resistor, and an LED.

VII. Traditional Workflow:



1. *Hardware-Software Interfacing:*

The Motors are controlled by an Arduino, which has its own motor driver. The motors connect to the Arduino, while the Arduino itself is interfaced with the Raspberry Pi. This gives us the freedom to code at a higher level of abstraction. The OpenCV environment is compiled for ARM processor on the Pi, to be used to run Computer Vision algorithms.

2. *Computer Vision and Image processing for Lane Detection:*

A lane-keep assist system has two components, namely, perception (lane detection) and Path/Motion Planning (steering). Lane detection's job is to turn a video of the road into the coordinates of the detected lane lines. One way to achieve this is via the computer vision package, which we installed on our Raspberry Pi, OpenCV.

The image from the PiCam is first captured. It is then processed in the following way to achieve the objective.

The colour of our lanes was chosen to be blue (controlled environment), but later we were able to change it to any arbitrary colour.

The image is then converted from the RGB colour space to HSV (Hue,Saturation,Value) colour space (different shades of blue to be regarded the same).

We then chose to lift only the pixels that fall under the blue range spectrum. This will give us a rendered mask of just the blue pixels detected.

Canny edge detection is used on top of this mask to get the edge mask. This mask is passed through another function to focus only on the region of interest.

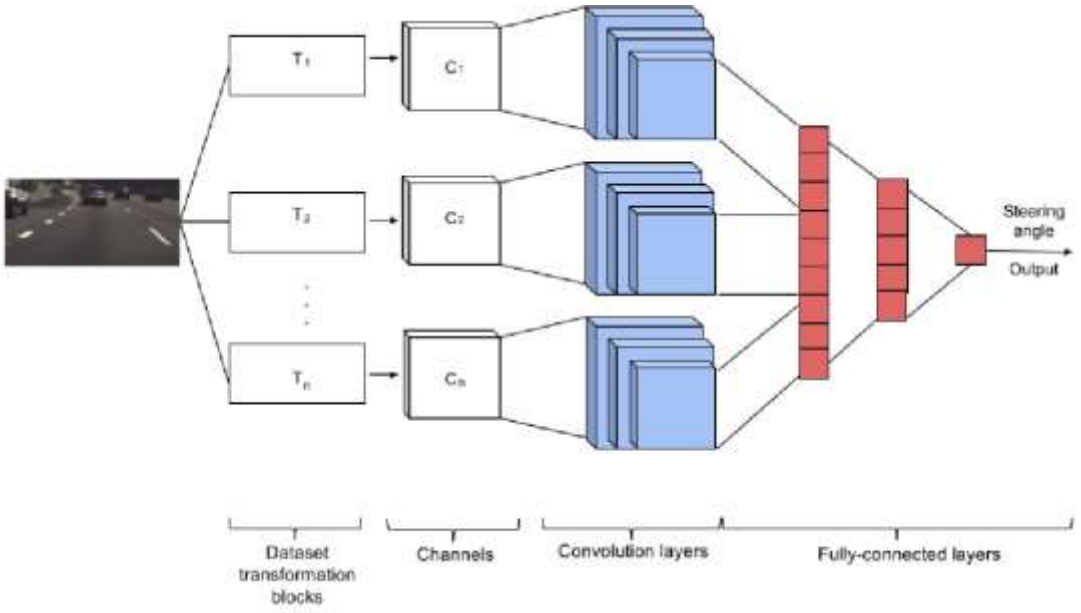
This image is once again passed through Hough Line Transform which finds out the line segments present in our lanes. These line segments will be too many in number. Hence, we had to average these lines out based on their slope, x and y coordinates, to come up with the equation of 2 lines (lanes) or one line (depending on the image).

Using these 2 /1 line segment equations, we are then able to calculate the steering angle needed to navigate through the lane, keeping the car in the middle.

3. *Actuator:*

This detected steering angle is passed to our high-level car control system, which will accordingly change the direction of the car's motion.

IX. AI Workflow:



1. *Hardware-Software Interfacing:*

The hardware is identical to the traditional workflow. However, in the software side, an additional Tensorflow environment, which was specifically compiled for ARM processor was set up to run our deep learning models locally on the Pi.

2. *Computer Vision and Lane Navigation:*

Deep Learning and Machine Learning algorithms are essentially universal function mappers. Given a set of inputs and outputs, if a pattern exists, the Neural Networks must be able to capture it and mimic it. This is called the Universal Theory of Approximation. Based on this, we decided that the entire sequential IP workflow can be replaced by a single Neural Network.

3. *Dataset creation for Deep Learning:*

Deep Learning algorithms need huge amounts of data to train. Only with sufficiently large pairs of inputs and outputs, our model will converge.

We need our deep learning model to replace the existing system, and hence it needs to mimic this. So, we took a few videos of cars travelling through roads and applied our Image Processing workflow on it. This gave us the steering angles needed for every frame. With this dataset created, we proceeded to train our model.

4. *Model creation:*

We chose a custom CNN architecture for this task. After many iterations, we chose to not use any pretrained model and create one from scratch.

5. *Model training:*

It was trained on the dataset for 30 Epochs and was able to achieve an accuracy of 97%. Adam optimizer was used and the mean squared loss error metric was used.

6. *Inference Model:*

The trained model is transferred to the Pi for inference. The image taken by the Picam is passed to this Neural Network and it'll output the steering angle needed for the actuator. Every image is passed through the neural network and the output is received by the controller module.

7. *Actuators:*

The actuators are functionally the same as the traditional workflow. The Python program on the Raspberry Pi interfaces with the Arduino to control the motors needed for vehicular movement.

The different parts of the workflow will be explained in much more detail by the individual contributors.

X. Code:

Arduino Code for motor control:

```
char receivedChar;
boolean newData = false;
//Motor A
const int motorPin1  = 9;  // Pin 14 of L293
const int motorPin2  = 10;  // Pin 10 of L293
//Motor B
const int motorPin3  = 6; // Pin  7 of L293
const int motorPin4  = 5;  // Pin  2 of L293


void setup() {
 Serial.begin(9600);
   //Set pins as outputs
   pinMode(12, OUTPUT);
   pinMode(motorPin1, OUTPUT);
   pinMode(motorPin2, OUTPUT);
   pinMode(motorPin3, OUTPUT);
   pinMode(motorPin4, OUTPUT);
}
void loop() {
 recvInfo();
 lightLED();
}
void recvInfo()
{
 if (Serial.available() > 0)
 {
   receivedChar = Serial.read();
   newData = true;
 }
}
```

```
void lightLED() {
  int in = (receivedChar - '0');
  while(newData == true)
  {
   // digitalWrite(led, LOW);
    digitalWrite(12, HIGH);
    delay(2000);
    digitalWrite(12, LOW);
    if(in==1)
    {
      forward();
    }
    else if(in==2)
    {
      backward();
    }
    else if(in==3)
    {
      left();
    }
    else if(in==4)
    {
      right();
    }
    newData = false;
  }
}
void forward()
{
    digitalWrite(12, HIGH);
    delay(500);
    digitalWrite(12, LOW);
    delay(500);
```

```
  digitalWrite(12, HIGH);
  delay(1000);
  digitalWrite(12, LOW);
  delay(1000);
   //This code will turn Motor A counter-clockwise for 2 sec.
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 180);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 180);
  delay(5000);
  /*
  //This code will turn Motor B clockwise for 2 sec.
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 180);
  analogWrite(motorPin3, 180);
  analogWrite(motorPin4, 0);
  delay(1000);
  //This code will turn Motor B counter-clockwise for 2 sec.
  analogWrite(motorPin1, 180);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 180);
  delay(1000);    */
  //And this code will stop motors
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 0);
}
void backward()
{
  analogWrite(motorPin1, 180);
  analogWrite(motorPin2, 0);
```

```
  analogWrite(motorPin3, 180);
  analogWrite(motorPin4, 0);
  delay(5000);
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 0);
}
void left()
{
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 180);
  analogWrite(motorPin3, 180);
  analogWrite(motorPin4, 0);
  delay(5000);
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 0);
}
void right()
{
  analogWrite(motorPin1, 180);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 180);
  delay(5000);
  analogWrite(motorPin1, 0);
  analogWrite(motorPin2, 0);
  analogWrite(motorPin3, 0);
  analogWrite(motorPin4, 0);

}
```

Pi interfacing with Arduino check:

```
import serial
ser = serial.Serial('/dev/ttyACM0', 9600)
ser.write('1')
```

Higher Level Python driver control:

```
import logging
import picar
import cv2
import datetime
from hand_coded_lane_follower import HandCodedLaneFollower
from objects_on_road_processor import ObjectsOnRoadProcessor

_SHOW_IMAGE = True



class DeepPiCar(object):

    __INITIAL_SPEED = 0
    __SCREEN_WIDTH = 320
    __SCREEN_HEIGHT = 240

    def __init__(self):
        """ Init camera and wheels"""
        logging.info('Creating a DeepPiCar...')

        picar.setup()

        logging.debug('Set up camera')
        self.camera = cv2.VideoCapture(-1)
        self.camera.set(3, self.__SCREEN_WIDTH)
        self.camera.set(4, self.__SCREEN_HEIGHT)
```

```
        self.pan_servo = picar.Servo.Servo(1)
        self.pan_servo.offset = -30  # calibrate servo to center
        self.pan_servo.write(90)

        self.tilt_servo = picar.Servo.Servo(2)
        self.tilt_servo.offset = 20  # calibrate servo to center
        self.tilt_servo.write(90)

        logging.debug('Set up back wheels')
        self.back_wheels = picar.back_wheels.Back_Wheels()
        self.back_wheels.speed = 0  # Speed Range is 0 (stop) - 100 (fastest)

        logging.debug('Set up front wheels')
        self.front_wheels = picar.front_wheels.Front_Wheels()
        self.front_wheels.turning_offset = -25  # calibrate servo to center
        self.front_wheels.turn(90)  # Steering Range is 45 (left) - 90 (center) - 135 (right)

        self.lane_follower = HandCodedLaneFollower(self)
        self.traffic_sign_processor = ObjectsOnRoadProcessor(self)
        # lane_follower = DeepLearningLaneFollower()

        self.fourcc = cv2.VideoWriter_fourcc(*'XVID')
        datestr = datetime.datetime.now().strftime("%y%m%d_%H%M%S")
        self.video_orig = self.create_video_recorder('../data/tmp/car_video%s.avi' % datestr)
        self.video_lane = self.create_video_recorder('../data/tmp/car_video_lane%s.avi' % datestr)
        self.video_objs = self.create_video_recorder('../data/tmp/car_video_objs%s.avi' % datestr)

        logging.info('Created a DeepPiCar')

    def create_video_recorder(self, path):
        return cv2.VideoWriter(path, self.fourcc, 20.0, (self.__SCREEN_WIDTH, self.__SCREEN_HEIGHT))

    def __enter__(self):
```

```
        """ Entering a with statement """
        return self

    def __exit__(self, _type, value, traceback):
        """ Exit a with statement"""
        if traceback is not None:
            # Exception occurred:
            logging.error('Exiting with statement with exception %s' % traceback)

        self.cleanup()

    def cleanup(self):
        """ Reset the hardware"""
        logging.info('Stopping the car, resetting hardware.')
        self.back_wheels.speed = 0
        self.front_wheels.turn(90)
        self.camera.release()
        self.video_orig.release()
        self.video_lane.release()
        self.video_objs.release()
        cv2.destroyAllWindows()

    def drive(self, speed=__INITIAL_SPEED):
        """ Main entry point of the car, and put it in drive mode
        Keyword arguments:
        speed -- speed of back wheel, range is 0 (stop) - 100 (fastest)
        """

        logging.info('Starting to drive at speed %s...' % speed)
        self.back_wheels.speed = speed
        i = 0
        while self.camera.isOpened():
            _, image_lane = self.camera.read()
```

```
            image_objs = image_lane.copy()
            i += 1
            self.video_orig.write(image_lane)

            image_objs = self.process_objects_on_road(image_objs)
            self.video_objs.write(image_objs)
            show_image('Detected Objects', image_objs)

            image_lane = self.follow_lane(image_lane)
            self.video_lane.write(image_lane)
            show_image('Lane Lines', image_lane)

            if cv2.waitKey(1) & 0xFF == ord('q'):
                self.cleanup()
                break

    def process_objects_on_road(self, image):
        image = self.traffic_sign_processor.process_objects_on_road(image)
        return image

    def follow_lane(self, image):
        image = self.lane_follower.follow_lane(image)
        return image
############################
# Utility Functions
############################
def show_image(title, frame, show=_SHOW_IMAGE):
    if show:
        cv2.imshow(title, frame)

def main():
    with DeepPiCar() as car:
        car.drive(40)
```

```
if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG, format='%(levelname)-5s:%(asctime)s: %(message)s')
    main()
```

OpenCV Test:

```
import cv2
def main():
    camera = cv2.VideoCapture(-1)
    camera.set(3, 640)
    camera.set(4, 480)
    while( camera.isOpened()):
        _, image = camera.read()
        cv2.imshow('Original', image)
        b_w_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        cv2.imshow('B/W', b_w_image)
        if cv2.waitKey(1) & 0xFF == ord('q') :
            break
    cv2.destroyAllWindows()
if __name__ == '__main__':
    main()
```

Image Processing workflow:

```
import cv2
import numpy as np
import logging
import math
import datetime
import sys
_SHOW_IMAGE = False
class HandCodedLaneFollower(object):
    def __init__(self, car=None):
        logging.info('Creating a HandCodedLaneFollower...')
```

```
        self.car = car
        self.curr_steering_angle = 90

    def follow_lane(self, frame):
        # Main entry point of the lane follower
        show_image("orig", frame)
        lane_lines, frame = detect_lane(frame)
        final_frame = self.steer(frame, lane_lines)
        return final_frame
    def steer(self, frame, lane_lines):
        logging.debug('steering...')
        if len(lane_lines) == 0:
            logging.error('No lane lines detected, nothing to do.')
            return frame
        new_steering_angle = compute_steering_angle(frame, lane_lines)
        self.curr_steering_angle = stabilize_steering_angle(self.curr_steering_angle, new_steering_angle,
len(lane_lines))

        if self.car is not None:
            self.car.front_wheels.turn(self.curr_steering_angle)
        curr_heading_image = display_heading_line(frame, self.curr_steering_angle)
        show_image("heading", curr_heading_image)

        return curr_heading_image

############################
# Frame processing steps
############################
def detect_lane(frame):
    logging.debug('detecting lane lines...')

    edges = detect_edges(frame)
    show_image('edges', edges)
```

```
    cropped_edges = region_of_interest(edges)
    show_image('edges cropped', cropped_edges)

    line_segments = detect_line_segments(cropped_edges)
    line_segment_image = display_lines(frame, line_segments)
    show_image("line segments", line_segment_image)

    lane_lines = average_slope_intercept(frame, line_segments)
    lane_lines_image = display_lines(frame, lane_lines)
    show_image("lane lines", lane_lines_image)

    return lane_lines, lane_lines_image


def detect_edges(frame):
    # filter for blue lane lines
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    show_image("hsv", hsv)
    lower_blue = np.array([30, 40, 0])
    upper_blue = np.array([150, 255, 255])
    mask = cv2.inRange(hsv, lower_blue, upper_blue)
    show_image("blue mask", mask)
    # detect edges
    edges = cv2.Canny(mask, 200, 400)
    return edges

def detect_edges_old(frame):
    # filter for blue lane lines
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    show_image("hsv", hsv)
    for i in range(16):
        lower_blue = np.array([30, 16 * i, 0])
```

```
        upper_blue = np.array([150, 255, 255])
        mask = cv2.inRange(hsv, lower_blue, upper_blue)
        show_image("blue mask Sat=%s" % (16* i), mask)
    #for i in range(16):
        #lower_blue = np.array([16 * i, 40, 50])
        #upper_blue = np.array([150, 255, 255])
        #mask = cv2.inRange(hsv, lower_blue, upper_blue)
       # show_image("blue mask hue=%s" % (16* i), mask)

        # detect edges
    edges = cv2.Canny(mask, 200, 400)
    return edges

def region_of_interest(canny):
    height, width = canny.shape
    mask = np.zeros_like(canny)
    # only focus bottom half of the screen
    polygon = np.array([[
        (0, height * 1 / 2),
        (width, height * 1 / 2),
        (width, height),
        (0, height),
    ]], np.int32)
    cv2.fillPoly(mask, polygon, 255)
    show_image("mask", mask)
    masked_image = cv2.bitwise_and(canny, mask)
    return masked_image

def detect_line_segments(cropped_edges):
    # tuning min_threshold, minLineLength, maxLineGap is a trial and error process by hand
    rho = 1  # precision in pixel, i.e. 1 pixel
    angle = np.pi / 180  # degree in radian, i.e. 1 degree
    min_threshold = 10  # minimal of votes
```

```
    line_segments = cv2.HoughLinesP(cropped_edges, rho, angle, min_threshold, np.array([]), minLineLength=8,
                                    maxLineGap=4)
    if line_segments is not None:
        for line_segment in line_segments:
            logging.debug('detected line_segment:')
            logging.debug("%s of length %s" % (line_segment, length_of_line_segment(line_segment[0])))
    return line_segments


def average_slope_intercept(frame, line_segments):
    lane_lines = []
    if line_segments is None:
        logging.info('No line_segment segments detected')
        return lane_lines

    height, width, _ = frame.shape
    left_fit = []
    right_fit = []

    boundary = 1/3
    left_region_boundary = width * (1 - boundary)  # left lane line segment should be on left 2/3 of the screen
    right_region_boundary = width * boundary # right lane line segment should be on left 2/3 of the screen

    for line_segment in line_segments:
        for x1, y1, x2, y2 in line_segment:
            if x1 == x2:
                logging.info('skipping vertical line segment (slope=inf): %s' % line_segment)
                continue
            fit = np.polyfit((x1, x2), (y1, y2), 1)
            slope = fit[0]
            intercept = fit[1]
            if slope < 0:
                if x1 < left_region_boundary and x2 < left_region_boundary:
                    left_fit.append((slope, intercept))
```

```
            else:
                if x1 > right_region_boundary and x2 > right_region_boundary:
                    right_fit.append((slope, intercept))

    left_fit_average = np.average(left_fit, axis=0)
    if len(left_fit) > 0:
        lane_lines.append(make_points(frame, left_fit_average))

    right_fit_average = np.average(right_fit, axis=0)
    if len(right_fit) > 0:
        lane_lines.append(make_points(frame, right_fit_average))

    logging.debug('lane lines: %s' % lane_lines)  # [[[316, 720, 484, 432]], [[1009, 720, 718, 432]]]

    return lane_lines


def compute_steering_angle(frame, lane_lines):
    """ Find the steering angle based on lane line coordinate
        We assume that camera is calibrated to point to dead center
    """
    if len(lane_lines) == 0:
        logging.info('No lane lines detected, do nothing')
        return -90

    height, width, _ = frame.shape
    if len(lane_lines) == 1:
        logging.debug('Only detected one lane line, just follow it. %s' % lane_lines[0])
        x1, _, x2, _ = lane_lines[0][0]
        x_offset = x2 - x1
    else:
        _, _, left_x2, _ = lane_lines[0][0]
        _, _, right_x2, _ = lane_lines[1][0]
```

```
    camera_mid_offset_percent = 0.02 # 0.0 means car pointing to center, -0.03: car is centered to left, +0.03 means car pointing to right
    mid = int(width / 2 * (1 + camera_mid_offset_percent))
    x_offset = (left_x2 + right_x2) / 2 - mid

  # find the steering angle, which is angle between navigation direction to end of center line
  y_offset = int(height / 2)

  angle_to_mid_radian = math.atan(x_offset / y_offset)  # angle (in radian) to center vertical line
  angle_to_mid_deg = int(angle_to_mid_radian * 180.0 / math.pi)  # angle (in degrees) to center vertical line
  steering_angle = angle_to_mid_deg + 90  # this is the steering angle needed by picar front wheel

  logging.debug('new steering angle: %s' % steering_angle)
  return steering_angle


def stabilize_steering_angle(curr_steering_angle, new_steering_angle, num_of_lane_lines, max_angle_deviation_two_lines=5, max_angle_deviation_one_lane=1):
  """
  Using last steering angle to stabilize the steering angle
  This can be improved to use last N angles, etc
  if new angle is too different from current angle, only turn by max_angle_deviation degrees
  """
  if num_of_lane_lines == 2 :
    # if both lane lines detected, then we can deviate more
    max_angle_deviation = max_angle_deviation_two_lines
  else :
    # if only one lane detected, don't deviate too much
    max_angle_deviation = max_angle_deviation_one_lane

  angle_deviation = new_steering_angle - curr_steering_angle
  if abs(angle_deviation) > max_angle_deviation:
    stabilized_steering_angle = int(curr_steering_angle
                                    + max_angle_deviation * angle_deviation / abs(angle_deviation))
```

```
    else:
        stabilized_steering_angle = new_steering_angle
    logging.info('Proposed angle: %s, stabilized angle: %s' % (new_steering_angle, stabilized_steering_angle))
    return stabilized_steering_angle


############################
# Utility Functions
############################
def display_lines(frame, lines, line_color=(0, 255, 0), line_width=10):
    line_image = np.zeros_like(frame)
    if lines is not None:
        for line in lines:
            for x1, y1, x2, y2 in line:
                cv2.line(line_image, (x1, y1), (x2, y2), line_color, line_width)
    line_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
    return line_image


def display_heading_line(frame, steering_angle, line_color=(0, 0, 255), line_width=5, ):
    heading_image = np.zeros_like(frame)
    height, width, _ = frame.shape

    # figure out the heading line from steering angle
    # heading line (x1,y1) is always center bottom of the screen
    # (x2, y2) requires a bit of trigonometry
    # Note: the steering angle of:
    # 0-89 degree: turn left
    # 90 degree: going straight
    # 91-180 degree: turn right
    steering_angle_radian = steering_angle / 180.0 * math.pi
    x1 = int(width / 2)
    y1 = height
    x2 = int(x1 - height / 2 / math.tan(steering_angle_radian))
```

```
    y2 = int(height / 2)
    cv2.line(heading_image, (x1, y1), (x2, y2), line_color, line_width)
    heading_image = cv2.addWeighted(frame, 0.8, heading_image, 1, 1)
    return heading_image

def length_of_line_segment(line):
    x1, y1, x2, y2 = line
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

def show_image(title, frame, show=_SHOW_IMAGE):
    if show:
        cv2.imshow(title, frame)
def make_points(frame, line):
    height, width, _ = frame.shape
    slope, intercept = line
    y1 = height  # bottom of the frame
    y2 = int(y1 * 1 / 2)  # make points from middle of the frame down
    # bound the coordinates within the frame
    x1 = max(-width, min(2 * width, int((y1 - intercept) / slope)))
    x2 = max(-width, min(2 * width, int((y2 - intercept) / slope)))
    return [[x1, y1, x2, y2]]
############################
# Test Functions
############################
def test_photo(file):
    land_follower = HandCodedLaneFollower()
    frame = cv2.imread(file)
    combo_image = land_follower.follow_lane(frame)
    show_image('final', combo_image, True)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def test_video(video_file):
```

```
    lane_follower = HandCodedLaneFollower()
    cap = cv2.VideoCapture(video_file + '.avi')

    # skip first second of video.
    for i in range(3):
        _, frame = cap.read()
    video_type = cv2.VideoWriter_fourcc(*'XVID')
    video_overlay = cv2.VideoWriter("%s_overlay.avi" % (video_file), video_type, 20.0, (320, 240))
    try:
        i = 0
        while cap.isOpened():
            _, frame = cap.read()
            print('frame %s' % i )
            combo_image= lane_follower.follow_lane(frame)

            cv2.imwrite("%s_%03d_%03d.png" % (video_file, i, lane_follower.curr_steering_angle), frame)
            cv2.imwrite("%s_overlay_%03d.png" % (video_file, i), combo_image)
            video_overlay.write(combo_image)
            cv2.imshow("Road with Lane line", combo_image)
            i += 1
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
    finally:
        cap.release()
        video_overlay.release()
        cv2.destroyAllWindows()

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    test_video('/home/pi/DeepPiCar/driver/data/tmp/video01')
    #test_photo('/home/pi/DeepPiCar/driver/data/video/car_video_190427_110320_073.png')
    #test_photo(sys.argv[1])
    #test_video(sys.argv[1])
```

Neural Network Model:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import scipy

def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def conv2d(x, W, stride):
  return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='VALID')

x = tf.placeholder(tf.float32, shape=[None, 66, 200, 3])
y_ = tf.placeholder(tf.float32, shape=[None, 1])

x_image = x

#first convolutional layer
W_conv1 = weight_variable([5, 5, 3, 24])
b_conv1 = bias_variable([24])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1, 2) + b_conv1)

#second convolutional layer
W_conv2 = weight_variable([5, 5, 24, 36])
b_conv2 = bias_variable([36])

h_conv2 = tf.nn.relu(conv2d(h_conv1, W_conv2, 2) + b_conv2)
```

```
#third convolutional layer
W_conv3 = weight_variable([5, 5, 36, 48])
b_conv3 = bias_variable([48])

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 2) + b_conv3)

#fourth convolutional layer
W_conv4 = weight_variable([3, 3, 48, 64])
b_conv4 = bias_variable([64])

h_conv4 = tf.nn.relu(conv2d(h_conv3, W_conv4, 1) + b_conv4)

#fifth convolutional layer
W_conv5 = weight_variable([3, 3, 64, 64])
b_conv5 = bias_variable([64])

h_conv5 = tf.nn.relu(conv2d(h_conv4, W_conv5, 1) + b_conv5)

#FCL 1
W_fc1 = weight_variable([1152, 1164])
b_fc1 = bias_variable([1164])

h_conv5_flat = tf.reshape(h_conv5, [-1, 1152])
h_fc1 = tf.nn.relu(tf.matmul(h_conv5_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

#FCL 2
W_fc2 = weight_variable([1164, 100])
b_fc2 = bias_variable([100])
```

```
h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

#FCL 3
W_fc3 = weight_variable([100, 50])
b_fc3 = bias_variable([50])

h_fc3 = tf.nn.relu(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)

h_fc3_drop = tf.nn.dropout(h_fc3, keep_prob)

#FCL 3
W_fc4 = weight_variable([50, 10])
b_fc4 = bias_variable([10])

h_fc4 = tf.nn.relu(tf.matmul(h_fc3_drop, W_fc4) + b_fc4)

h_fc4_drop = tf.nn.dropout(h_fc4, keep_prob)

#Output
W_fc5 = weight_variable([10, 1])
b_fc5 = bias_variable([1])

#linear
#y = tf.multiply((tf.matmul(h_fc4_drop, W_fc5) + b_fc5), 2)
## atan
y =  tf.multiply(tf.atan(tf.matmul(h_fc4_drop, W_fc5) + b_fc5), 2)#scale the atan output
```

Model Training:

```
import os
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
from tensorflow.core.protobuf import saver_pb2

import driving_data

import model


LOGDIR = './save'

sess = tf.InteractiveSession()

L2NormConst = 0.001

train_vars = tf.trainable_variables()

loss = tf.reduce_mean(tf.square(tf.subtract(model.y_, model.y))) + tf.add_n([tf.nn.l2_loss(v) for v in
train_vars]) * L2NormConst

train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)

sess.run(tf.initialize_all_variables())


# create a summary to monitor cost tensor

tf.summary.scalar("loss", loss)

# merge all summaries into a single op

merged_summary_op =  tf.summary.merge_all()

saver = tf.train.Saver(write_version = saver_pb2.SaverDef.V1)

# op to write logs to Tensorboard

logs_path = './logs'

summary_writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())


epochs = 30

batch_size = 50


# train over the dataset about 30 times

for epoch in range(epochs):

  for i in range(int(driving_data.num_images/batch_size)):

    xs, ys = driving_data.LoadTrainBatch(batch_size)

    train_step.run(feed_dict={model.x: xs, model.y_: ys, model.keep_prob: 0.8})

    if i % 10 == 0:

      xs, ys = driving_data.LoadValBatch(batch_size)

      loss_value = loss.eval(feed_dict={model.x:xs, model.y_: ys, model.keep_prob: 1.0})
```

```
    print("Epoch: %d, Step: %d, Loss: %g" % (epoch, epoch * batch_size + i, loss_value))

  # write logs at every iteration
  summary = merged_summary_op.eval(feed_dict={model.x:xs, model.y_: ys, model.keep_prob: 1.0})
  summary_writer.add_summary(summary, epoch * driving_data.num_images/batch_size + i)

  if i % batch_size == 0:
    if not os.path.exists(LOGDIR):
      os.makedirs(LOGDIR)
    checkpoint_path = os.path.join(LOGDIR, "model.ckpt")
    filename = saver.save(sess, checkpoint_path)
 print("Model saved in file: %s" % filename)
```

Running Simulation:

```
#pip3 install opencv-python
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import scipy.misc
import model
import cv2
from subprocess import call
import math
from skimage.transform import resize
from skimage import img_as_ubyte
import numpy as np

sess = tf.InteractiveSession()
saver = tf.train.Saver()
saver.restore(sess, "save/model.ckpt")

# 0 indiactes color image
img = cv2.imread('steering_wheel_image.jpg',0)
```

```
rows,cols = img.shape

smoothed_angle = 0



#read data.txt
xs = []
ys = []
with open("driving_dataset/driving_dataset/data.txt") as f:
    for line in f:
        xs.append("driving_dataset/driving_dataset/" + line.split()[0])
        #the paper by Nvidia uses the inverse of the turning radius,
        #but steering wheel angle is proportional to the inverse of turning radius
        #so the steering wheel angle in radians is used as the output
        ys.append(float(line.split()[1]) * scipy.pi / 180)

#get number of images
num_images = len(xs)



i = math.ceil(num_images*0.8) #using after 80% of images i.e. Test Images
print("Starting frameofvideo:" +str(i))

#Press q to stop
while(cv2.waitKey(10) != ord('q')):
    # Driving Test Image displayed as Video
    full_image = scipy.misc.imread("driving_dataset/driving_dataset/" + str(i) + ".jpg", mode="RGB")
    image = img_as_ubyte(resize(full_image[-150:], (66, 200))) / 255.0
    degrees = model.y.eval(feed_dict={model.x: [image], model.keep_prob: 1.0})[0][0] * 180.0 / scipy.pi
    #call("clear")
    print("Steering angle: " + str(degrees) + " (pred)\t" + str(ys[i]*180/scipy.pi) + " (actual)")
    cv2.imshow("frame", cv2.cvtColor(full_image, cv2.COLOR_RGB2BGR))
    #make smooth angle transitions by turning the steering wheel based on the difference of the current angle
```

```
    #and the predicted angle
    #Angle at which the steering wheel image should be rotated
    smoothed_angle += 0.2 * pow(abs((degrees - smoothed_angle)), 2.0 / 3.0) * (degrees - smoothed_angle) /
abs(degrees - smoothed_angle)
    M = cv2.getRotationMatrix2D((cols/2,rows/2),-smoothed_angle,1)
    dst = cv2.warpAffine(img,M,(cols,rows))
    cv2.imshow("steering wheel", dst)
    i += 1

cv2.destroyAllWindows()
```

(The Code which we weren't able to execute):

Deep Learning hardware Interfaced Lane traversal:

```
import cv2
import numpy as np
import logging
import math
from keras.models import load_model
from hand_coded_lane_follower import HandCodedLaneFollower
_SHOW_IMAGE = False
class EndToEndLaneFollower(object):
    def __init__(self, car=None,
                 model_path='/home/pi/DeepPiCar/models/lane_navigation/data/model_result/model.h5'):
        logging.info('Creating a EndToEndLaneFollower...')
        self.car = car
        self.curr_steering_angle = 90
        self.model = load_model(model_path)

    def follow_lane(self, frame):
        # Main entry point of the lane follower
        show_image("orig", frame)
        self.curr_steering_angle = self.compute_steering_angle(frame)
        logging.debug("curr_steering_angle = %d" % self.curr_steering_angle)
```

```
        if self.car is not None:
            self.car.front_wheels.turn(self.curr_steering_angle)
        final_frame = display_heading_line(frame, self.curr_steering_angle)
        return final_frame
    def compute_steering_angle(self, frame):
        preprocessed = img_preprocess(frame)
        X = np.asarray([preprocessed])
        steering_angle = self.model.predict(X)[0]
        logging.debug('new steering angle: %s' % steering_angle)
        return int(steering_angle + 0.5) # round the nearest integer


def img_preprocess(image):
    height, _, _ = image.shape
    image = image[int(height/2):,:,:]  # remove top half of the image, as it is not relevant for lane following
    image = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)  # Nvidia model said it is best to use YUV color space
    image = cv2.GaussianBlur(image, (3,3), 0)
    image = cv2.resize(image, (200,66)) # input image size (200,66) Nvidia model
    image = image / 255 # normalizing, the processed image becomes black for some reason.  do we need this?
    return image


def display_heading_line(frame, steering_angle, line_color=(0, 0, 255), line_width=5, ):
    heading_image = np.zeros_like(frame)
    height, width, _ = frame.shape

    # Note: the steering angle of:
    # 0-89 degree: turn left
    # 90 degree: going straight
    # 91-180 degree: turn right
    steering_angle_radian = steering_angle / 180.0 * math.pi
    x1 = int(width / 2)
    y1 = height
    x2 = int(x1 - height / 2 / math.tan(steering_angle_radian))
    y2 = int(height / 2)
```

```
    cv2.line(heading_image, (x1, y1), (x2, y2), line_color, line_width)
    heading_image = cv2.addWeighted(frame, 0.8, heading_image, 1, 1)

    return heading_image

def show_image(title, frame, show=_SHOW_IMAGE):
    if show:
        cv2.imshow(title, frame)

############################
# Test Functions
############################
def test_photo(file):
    lane_follower = EndToEndLaneFollower()
    frame = cv2.imread(file)
    combo_image = lane_follower.follow_lane(frame)
    show_image('final', combo_image, True)
    logging.info("filename=%s, model=%3d" % (file, lane_follower.curr_steering_angle))
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def test_video(video_file):
    end_to_end_lane_follower = EndToEndLaneFollower()
    hand_coded_lane_follower = HandCodedLaneFollower()
    cap = cv2.VideoCapture(video_file + '.avi')
    for i in range(3):
        _, frame = cap.read()

    video_type = cv2.VideoWriter_fourcc(*'XVID')
    video_overlay = cv2.VideoWriter("%s_end_to_end.avi" % video_file, video_type, 20.0, (320, 240))
    try:
        i = 0
```

```
        while cap.isOpened():
            _, frame = cap.read()
            frame_copy = frame.copy()
            logging.info('Frame %s' % i)
            combo_image1 = hand_coded_lane_follower.follow_lane(frame)
            combo_image2 = end_to_end_lane_follower.follow_lane(frame_copy)

            diff = end_to_end_lane_follower.curr_steering_angle - hand_coded_lane_follower.curr_steering_angle;
            logging.info("desired=%3d, model=%3d, diff=%3d" %
                         (hand_coded_lane_follower.curr_steering_angle,
                         end_to_end_lane_follower.curr_steering_angle,
                         diff))
            video_overlay.write(combo_image2)
            cv2.imshow("Hand Coded", combo_image1)
            cv2.imshow("Deep Learning", combo_image2)
            i += 1
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
    finally:
        cap.release()
        video_overlay.release()
        cv2.destroyAllWindows()

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    test_video('/home/pi/DeepPiCar/models/lane_navigation/data/images/video01')
    #test_photo('/home/pi/DeepPiCar/models/lane_navigation/data/images/video01_100_084.png')
    # test_photo(sys.argv[1])
    # test_video(sys.argv[1])
```

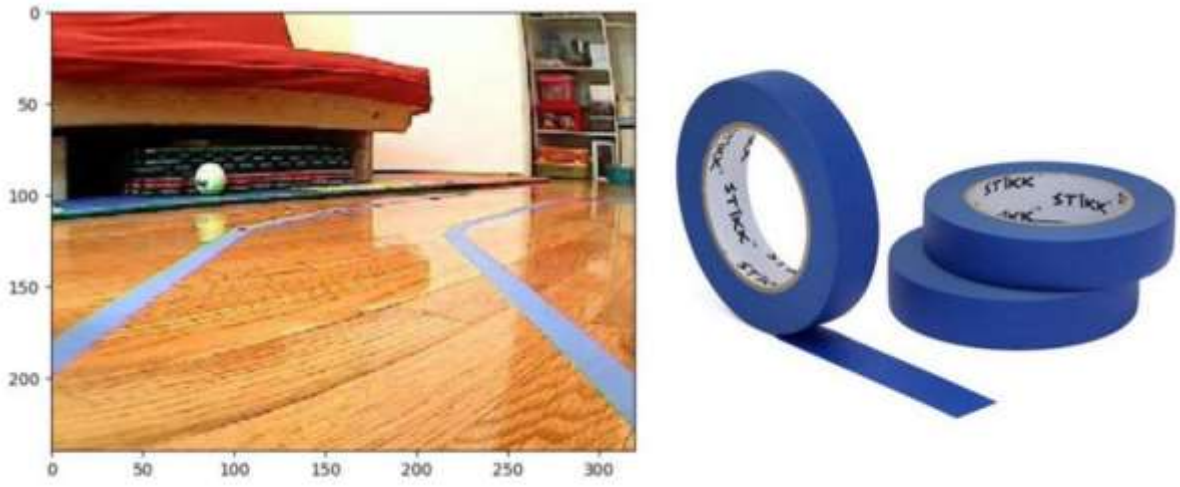
XI. Individual Contribution:

Use OpenCV to detect color, edges and lines segments and other Image Processing techniques applied to detect the lanes.

A lane keep assist system has two components, namely, perception (lane detection) and Path/Motion Planning (steering). Lane detection's job is to turn a video of the road into the coordinates of the detected lane lines. One way to achieve this is via the computer vision package, which we installed in Part 3, OpenCV. But before we can detect lane lines in a video, we must be able to detect lane lines in a single image. Once we can do that, detecting lane lines in a video is simply repeating the same steps for all frames in a video.

Isolate the Color of the Lane

When we set up lane lines for our car , we used the blue painter's tape to mark the lanes, because blue is a unique color easy to identify and the tape won't leave permanent sticky residues on the hardwood floor.



Original DashCam Image (left) and Blue Masking Tape for Lane Line (right)

The first thing to do is to isolate all the blue areas on the image. To do this, we first need to turn the color space used by the image, which is RGB (Red/Green/Blue) into the HSV (Hue/Saturation/Value) color space. The main idea behind this is that in an RGB image, different parts of the blue tape may be lit with different light, resulting them appears as darker blue or lighter blue. However, in HSV color space, the Hue component will render the entire blue tape as one color regardless of its shading. It is best to illustrate with the following image. Notice both lane lines are now roughly the same magenta color.

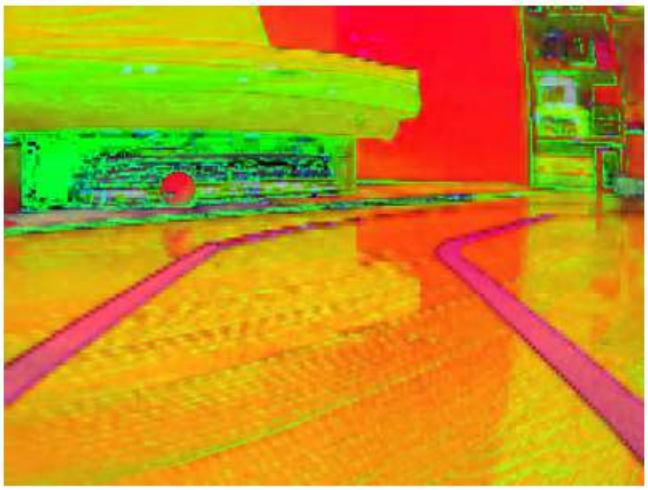


Image in HSV Color Space

Below is the OpenCV command to do this.

```
import cv2
frame = cv2.imread('/home/pi/DeepPiCar/driver/data/road1_240x320.png')
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Note that we used a **BGR** to HSV transformation, not **RBG** to HSV. This is because OpenCV, for some legacy reasons, reads images into BGR (Blue/Green/Red) color space by default, instead of the more commonly used RGB (Red/Green/Blue) color space. They are essentially equivalent color spaces, just order of the colors swapped.

Once the image is in HSV, we can “lift” all the blueish colors from the image. This is by specifying a range of the color Blue.

In Hue color space, the blue color is in about 120–300 degrees range, on a 0–360 degrees scale. You can specify a tighter range for blue, say 180–300 degrees, but it doesn’t matter too much.

Hue in 0–360 degrees scale

Here is the code to lift Blue out via OpenCV, and rendered mask image.

```
lower_blue = np.array([60, 40, 40])
upper_blue = np.array([150, 255, 255])
mask = cv2.inRange(hsv, lower_blue, upper_blue)
```

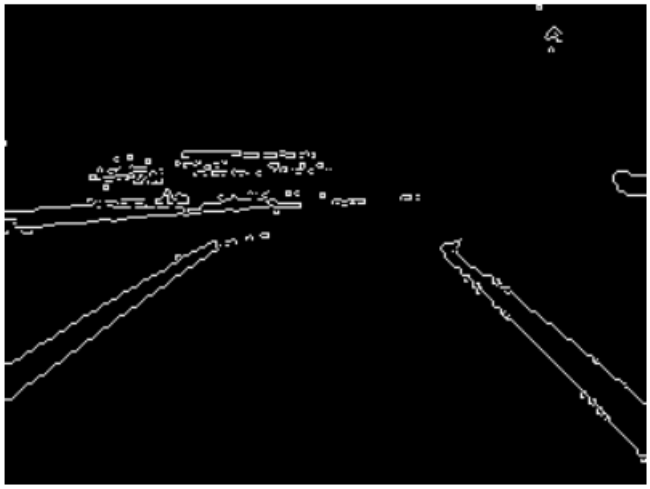
Blue Area Mask

Note OpenCV uses a range of 0–180, instead of 0–360, so the blue range we need to specify in OpenCV is 60–150 (instead of 120–300). These are the first parameters of the lower and upper bound arrays. The second (Saturation) and third parameters (Value) are not so important, I have found that the 40–255 ranges work reasonably well for both Saturation and Value.

Detecting Edges of Lane Lines

Next, we need to detect edges in the blue mask so that we can have a few distinct lines that represent the blue lane lines.

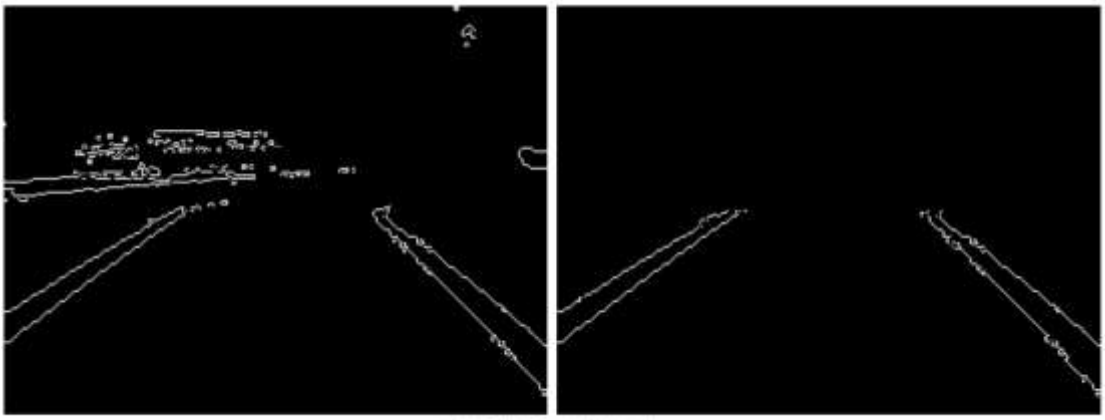
The Canny edge detection function is a powerful command that detects edges in an image. In the code below, the first parameter is the blue mask from the previous step. The second and third parameters are lower and upper ranges for edge detection, which OpenCV recommends to be (100, 200) or (200, 400), so we are using (200, 400).



Edges of all Blue Areas

Isolate Region of Interest

From the image above, we see that we detected quite a few blue areas that are NOT our lane lines. A closer look reveals that they are all at the top half of the screen. Indeed, when doing lane navigation, we only care about detecting lane lines that are closer to the car, where the bottom of the screen. So we will simply crop out the top half.

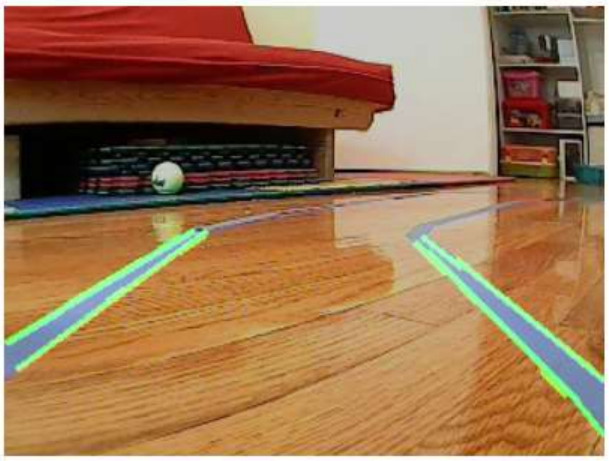


Edges (left) and Cropped Edges (right)

We first create a mask for the bottom half of the screen. Then when we merge the mask with the edges image to get the cropped_edges image on the right.

Detect Line Segments

we need to extract the coordinates of these lane lines from these white pixels. Luckily, OpenCV contains a magical function, called Hough Transform, which does exactly this. Hough Transform is a technique used in image processing to extract features like lines, circles, and ellipses. We will use it to find straight lines from a bunch of pixels that seem to form a line. The function HoughLinesP essentially tries to fit many lines through all the white pixels and return the most likely set of lines, subject to certain minimum threshold constraints.



Line segments detected by Hough Transform

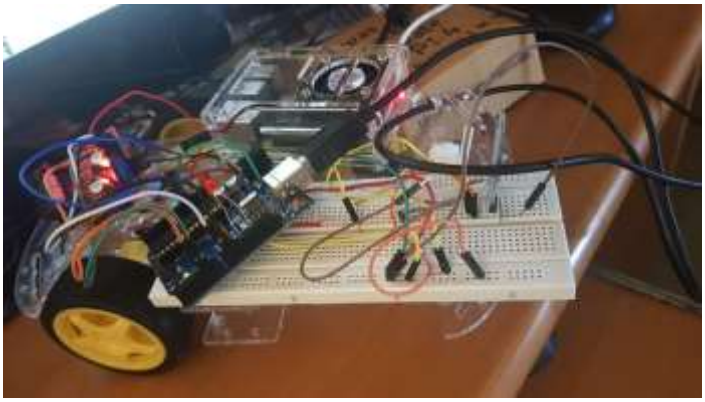
XII. Results and Discussion

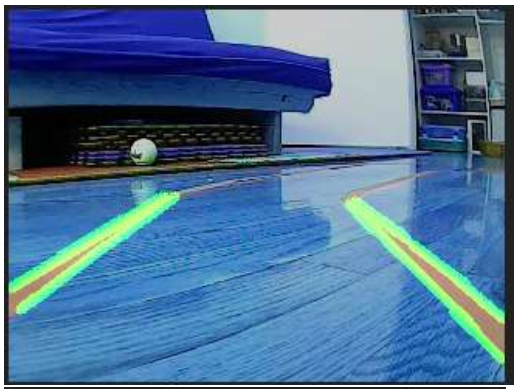
The individual components of the project were tested to ensure their efficient working. The hardware interface was set up between the Motors, Motor Drivers, Arduino and the Raspberry Pi. All levels of controls were tested before interfacing these components together.

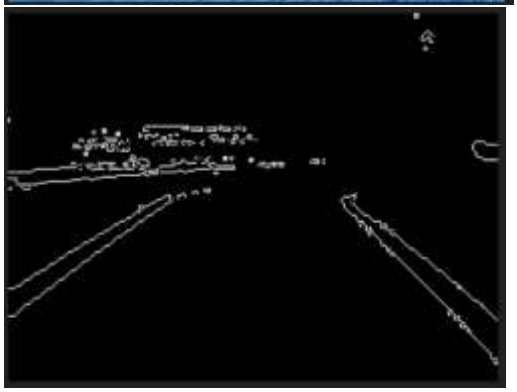
The image processing pipeline was able to generate a steering angle for every frame of a video. It was a highly sequential process which would not greatly benefit from the advances in hardware acceleration devices. Hence, to mimic this on a Neural Network, a model was created. The original workflow was applied on videos to get the dataset needed for the training of this model. It converged at an accuracy of 97% and performed efficiently in the simulation.

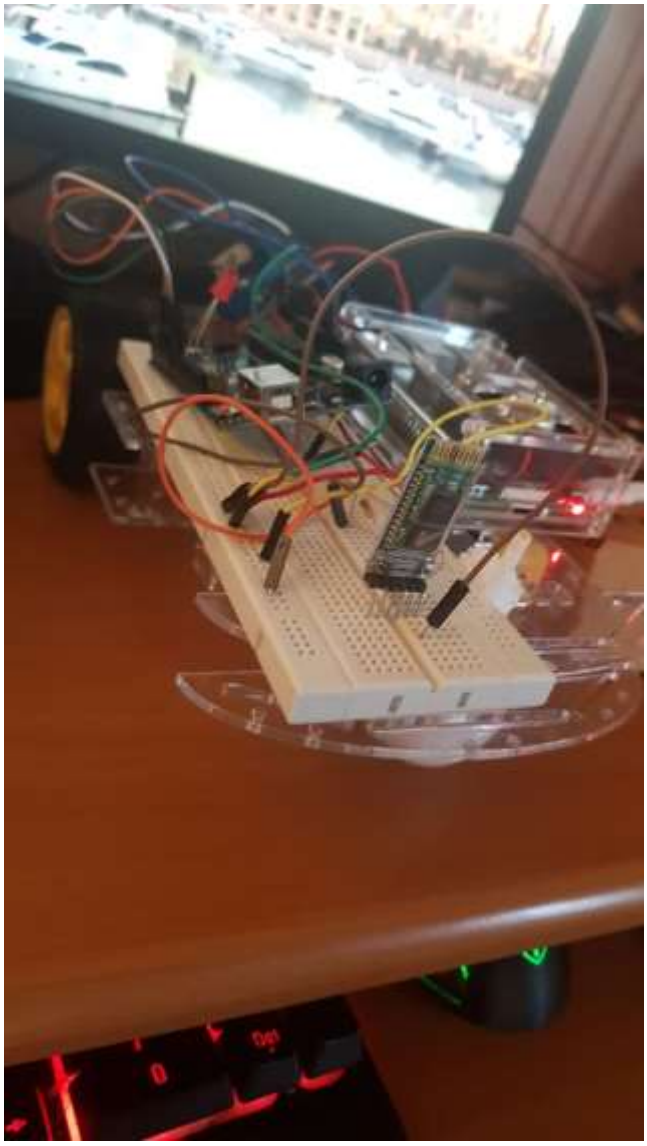
This model can be used to not just output the steering angle, but also braking, traffic sign detection, etc., if trained on more data. Hence, we were able to perform autonomous lane detection and navigation with the help of both IP and AI workflows.

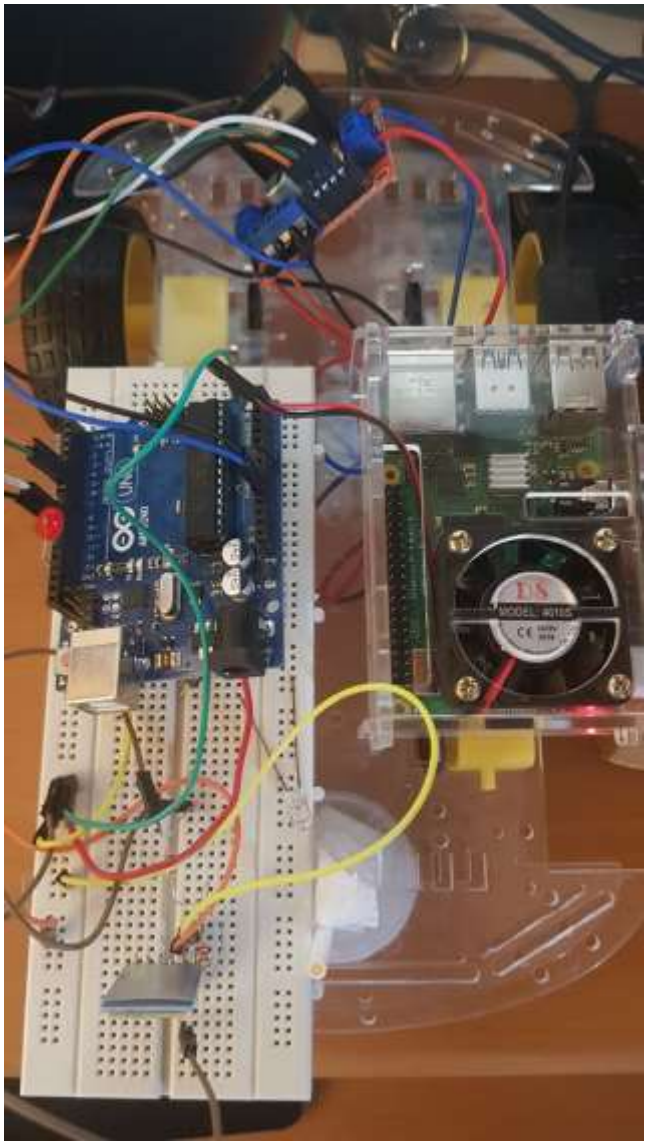
Here are a few images and screenshots of our project:

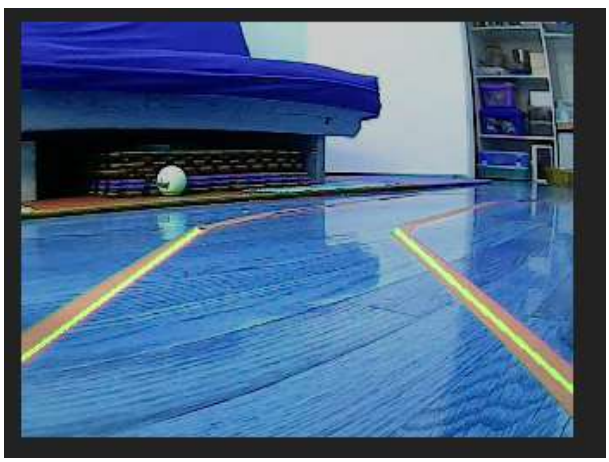


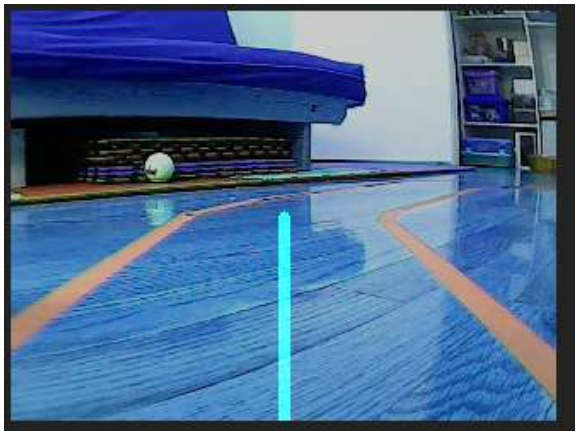












```
In [15]: lane_lines=detect_lane(frame)
    ...: lane_lines_image = display_lines(frame, lane_lines)
    ...:
    ...: img = Image.fromarray(lane_lines_image)
    ...: img.show()
    ...:
    ...: cur=90
    ...: newcur = compute_steering_angle(frame, lane_lines)
    ...: curr_steering_angle = stabilize_steering_angle(cur, newcur, len(lane_lines))
    ...:
    ...: curr_heading_image = display_heading_line(frame, cur)
    ...: img = Image.fromarray(curr_heading_image)
    ...: img.show()

In [16]: newcur
Out[16]: 89
```

XIII. Challenges and Limitations

One of the first challenges that we faced was finding the correct versions of software that could run on a Raspberry Pi. The documentations were scarce so figuring out the version compatibility between different modules and packages was very hard.

Only Akshay had worked with driver hardware before, Mukkesh with Deep Learning and Rohit with Image Processing. To set up a workflow which could make use of all our three areas of interest was a challenging task.

The car that we had did not have steering control. So, for calibration, we were supposed to write our own code. Due to the Covid lockdown, we were unable to implement this part of the software on the hardware (Pi and the car were in different places). Hence, we had to resort to a simulation to check if the car control (steering) was working as expected.

A day of meeting would have made us translate our simulation to a real demo. But unfortunately, it was not possible.

We were also able to build a bigger neural network to perform tasks such as traffic sign detection, but were unable to run it on Raspberry Pi due to budget constraints. A more powerful board such as Nvidia Jetson Nano(https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-nano/) would allow us to run even bigger Neural Networks locally.

XIV. Conclusion and Future Enhancements

We were able to build a prototype Deep Learning-powered autonomous lane navigation system. It worked exceedingly well in the simulation and can be translated to a well calibrated hardware. The workflows which were set up based on Image processing and AI worked really well and were both able to control the navigation of the car.

Bigger budget would have given us the opportunity to build a completely autonomous system on a much more powerful board, which would allow the Deep Learning model to recognise traffic signs and take actions based on the same. A better robotic car would also eliminate the interfacing and calibration between Arduino, motor drivers and Pi to run this software, since it would already have a steering control system.