POLITECNICO DI TORINO

Master's Degree
in Mechatronics Engineering

Master's Degree Thesis

Image Processing for Self-Driving Cars

Supervisor
prof. Stefano Malan

Candidate
Cristina Chicca

Academic Year 2021-2022

Contents

List of Figures

List of Tables

1 Introduction

Self-driving vehicles are steadily becoming a reality and they could change our world in unexpected ways. According to the mere definition, self-driving cars incorporate vehicular automation, meaning a ground vehicle capable of sensing its environment and moving safely with little or no human input. They incorporate a wide variety of sensors such as thermographic cameras, radar, lidar, sonar, GPS, odometry, Inertial Measurements Units (IMU) and advanced control systems interpret this sensory information to allow a safe journey, respecting the road and traffic signs, identifying appropriate navigation paths and behaving correctly in presence of obstacles. The promise of driverless technology has long been enticing due to its potential to take people out of high-risk working environments, streamline the industries and, most importantly, transform the experience of traveling assuring more safety. The World Health Organization estimates that more than 1.3 million people die every year as a result of road traffic crashes [1]; in fact, human errors cause 90% of car incidents. Currently, Waymo self-driving vehicles from Google have covered the most kilometers without any incidents and with human intervention needed every 21 to 9000 km on average. Traffic jams from the big cities can be avoided: IBI Group's report [2] mentioned that self-driving cars could enable more efficient, user friendly and low cost on demand transportation services even in low demand areas. Another major impact is derived from the reduction of carbon emissions because personal cars will not be necessary anymore, paving the way for more sustainable ways of living. The ultimate vision experts are working towards completely driverless vehicles, both within industry, wider transport networks and, in case, personal-use cars that can be deployed and used anywhere and everywhere around the world.

In order to grasp more practical solutions that currently exist, the Society of Automotive Engineers (SAE) provides six classifications of autonomous vehicles [3], from Level 1 to Level 5 according to the extension to which a human intervention is needed to drive and monitor the vehicle and its surrounding environment. More in details:

- Level 0 (No Driving Automation): the driver is fully responsible for every decision, including reacting to hazards.
- Level 1 (Driver Assistance): cars which have systems for driver assistance such as cruise control. Some examples consists in adaptive cruise control where the vehicle can keep a safe distance behind the next car, and park assist feature where the driver needs to control the speed of the vehicle while the car controls the steering. It is the lowest level of automation.
- Level 2 (Partial Driving Automation): this means Advanced Driver Assistance Systems (ADAS). The car has internal systems that take care of all aspects of driving and the driver must be able to take control of the car at any time if any part of the system fails.
- Level 3 (Conditional Driving Automation): if compared to level 2, this level is more a technological perspective than human perspective and these vehicles can be truly considered autonomous.
- Level 4 (High Driving Automation): vehicles can take decisions even if things go wrong or the system fails and they are so capable that the driver is not

required to intervene at all. The restriction is that they operate in self-driving mode only within limited area (geofencing), otherwise the driver must take the control or the car must be able to get itself to a safety zone.

- Level 5 (Fully Driving Automation): these vehicles will not have steering wheel or acceleration and braking pedals and will be free from geofencing. There is no legislative structure for cars of this level.

Although much work is still essential to reach from Level 3 to Level 5 solutions, most modern cars already have Level 1 to Level 2. Some examples are found in the BMW iX which comes with Level 2 semi-autonomous driving features that rely on 12 ultrasonic sensors, 5 cameras and 5 radar sensors and the Tesla Model 3 which comes with autopilot and full-self driving capability that rely on a powerful onboard computer [3].

The main characteristic of these categories is summarized in what is known as ADAS, aiming at assisting drivers reducing the number of car accidents. As shown in Fig. 1, essential safety-critical ADAS applications include [4]:

- Pedestrian detection/avoidance
- Lane departure warning/correction
- Traffic sign recognition
- Automatic emergency braking
- Blind spot detection

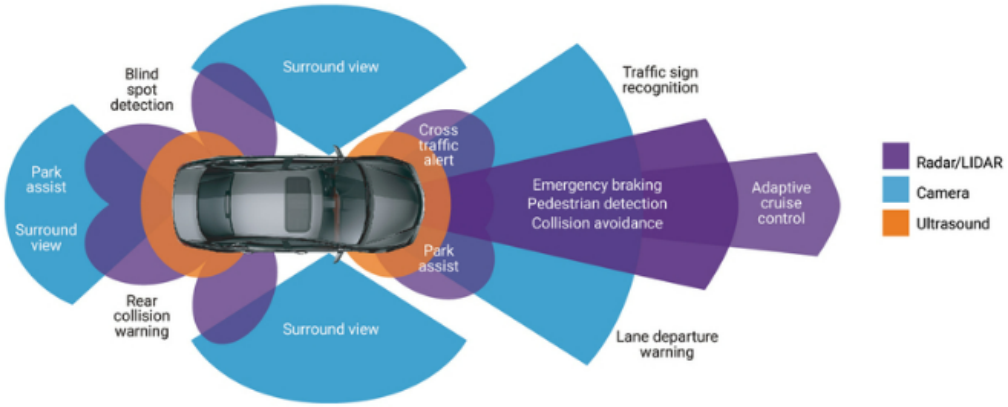

Figure 1: ADAS main features according to sensors placement.

The autonomous system needs to sense the environment, to determine the exact position on the road and needs to decide how it should behave in a given situation: this is the reason why self-driving cars are highly dependent on software to bridge the gap between sensor physics and the mechanical actuation of the vehicle. In this context, the utilization of image processing is crucial: the implementation of cameras in the vehicle involves a new Artificial Intelligence (AI) function that uses sensor fusion to identify and process objects. Sensor fusion combines large amounts of data with the help of image recognition software, ultrasound sensors, lidar and radar;

summarizing, this technology can analyze streaming video in real time, recognize what the video shows and determine how to react to it. More practically, radars enables the detection of vehicles and other moving objects around the car, front facing camera helps to detect and recognize objects like cars, trees, driving lane, humans, traffic signals and other important data.

In this work, the focus will be on the imaging where cars perform object detection, in particular:

- Lane Detection and Follow
- Intersection Detection
- Traffic Sign Detection
- Traffic Light Detection
- Object Detection

More specifically, image processing has been used on a particular car prototype granted by Bosch in a special challenge regarding exactly the autonomous drive. The content of the chapters is therefore briefly explained:

- Chapter 2: Details about Bosch Future Mobility Challenge and the overall project, including software and hardware components.
- Chapter 3: Explanation of the communication between files, including crucial Python utilities and communication with LAN and UDP.
- Chapter 4: Introduction to Image processing, details about the utilized camera and important aspects common to image processing algorithms.
- Chapter 5: Explanation of 2 main Lane Detection algorithms, one using the Hough Transform method, the other using Histogram Statistics method and final comparison of the performance. Moreover, a particular attention is given to Intersection Detection.
- Chapter 6: Description of Traffic Sign Recognition using both a linear SVM classifier and a Tensorflow-based neural network. At the end, they are compared analysing both advantages and disadvantages.
- Chapter 7: An amateur Traffic Light Detection algorithm is provided, analysing methods for color detection and calculation of pixel intensities.
- Chapter 8: The analysis of a first Object Detection algorithm is performed, subdividing the topic into Pedestrian Detection and Semaphore and Car Detection, the former exploiting a built-in function in OpenCV, the latter a shape-based function.
- Chapter 9: Conclusions about the work done and suggestions on further developments are reported.

2 Bosch Future Mobility Challenge

2.1 Introduction

This work has been realized based on the effort and the assignments performed during the participation to the so-called *Bosch Future Mobility Challenge* (BFMC): it is an international autonomous driving and connectivity competition for bachelor and master students organized by *Bosch Engineering Centre Cluj* since 2017. The competition invites student teams from all over the world every year to develop autonomous driving and connectivity algorithms on $1:10$ scaled vehicles, provided by the company, to navigate in a designated environment simulating a miniature smart city. The students work on their projects in collaboration with Bosch experts and academic professors for several months to develop the best-performing algorithms.

The author of this work has joined the challenge under the team *PoliTron* (Fig. 2) composed by 4 other colleagues from the master's degree program in Mechatronic Engineering of the Polytechnic of Turin, with the guidance of the supervisor himself, Professor Stefano Malan.

Figure 2: PoliTron team logo.

The job to carry out during the challenge, which lasts from November to May, consists in developing the algorithms involved in the realization of the autonomous car guide and implementing them into the received car, therefore it commits both the software and the hardware parts. All in all, it is a real and complete accomplishment of self-driving car.

2.2 The Competition

The competition requires that, in addition to the activities carried out by the teams to achieve the final objective, participating teams send a monthly periodic status via the competition website containing the followings to show their progress to the Bosch representatives:

- A **technical report** describing the development in the last sprint.
- A **project plan** alongside with the **project architecture**.
- A **video file** emphasizing with visual aid the contributions from the past month activity (already present in the report and project plan).

In the middle of the competition, on middle March, a first eliminatory phase takes place, the Mid-Season Quality Gate, in which each team is requested to send a 3-minutes (at most) long video in which the car must perform the following tasks in a single autonomous run:

1. Lane keeping.
2. Intersection crossing.

3. Complete manoeuvre after the following signs:

   3.1. *Stop* sign – the car must stop for at least 3 s.

   3.2. *Crosswalk* sign - the car must visibly reduce the speed and if a pedestrian is crossing, the car must stop.

   3.3. *Priority Road* sign - act normal, you are on a priority road and any vehicle coming from a non-priority road should stop.

   3.4. *Parking* sign - it is found before and after a parking lot and, if empty, can be used to perform the parking manoeuvre.

These tasks can be demonstrated by means of one of three possible alternatives:

- A video of the car performing the actions on a real-life like map.
- A video of the car in front of a Desktop, taking a video as a simulated input and acting accordingly.
- A video of the car in front of a Desktop where the simulator is running, taking as visual input the one from the camera inside the simulator.

The author's team has chosen the first option, realizing physically the track shown in Figure 3.

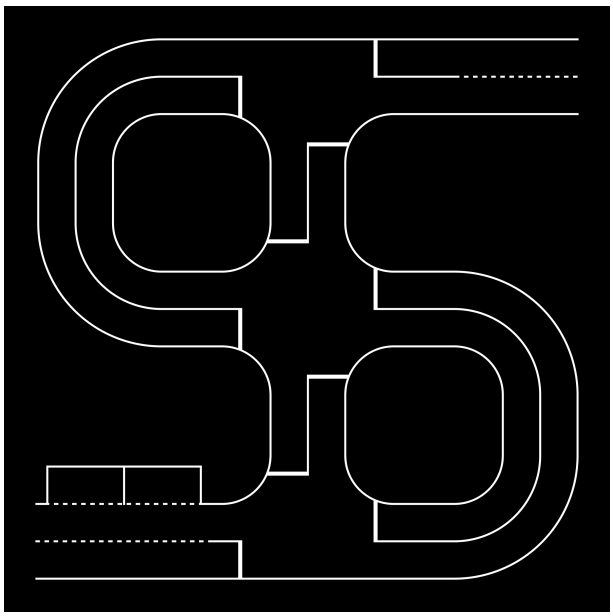

Figure 3: Test Track.

Based on the videos, the jury will decide which teams possess the right skills to continue the competition and to go to the Bosch Engineering Centre site in Cluj-Napoca (Romania) for the qualifications and possibly semifinals and finals in May.

During the race period in Romania, the teams will have to face two challenges: the technical and the speed one. The former requests that the car can correctly respect most of the road signs, such as traffic signs, traffic lights, lanes, intersections, ramps, and roundabouts. Moreover, it must detect pedestrians and overtake other cars present in the same lane. The latter asks the car to complete a determined

path in the shortest time possible, this time respecting only the lanes and the road markings. In addition to this, the teams will make a presentation in front of the jury.

Only a maximum of 8 teams will be selected to participate to the final race, in which the first 3 qualified teams will win both a money prize and the car kit, and another team, not included in the top 3, will be rewarded as the "best newcomer", meaning a team which did not take part to the competition in the previous year. All the phases of the challenge are reported in Figure 4.

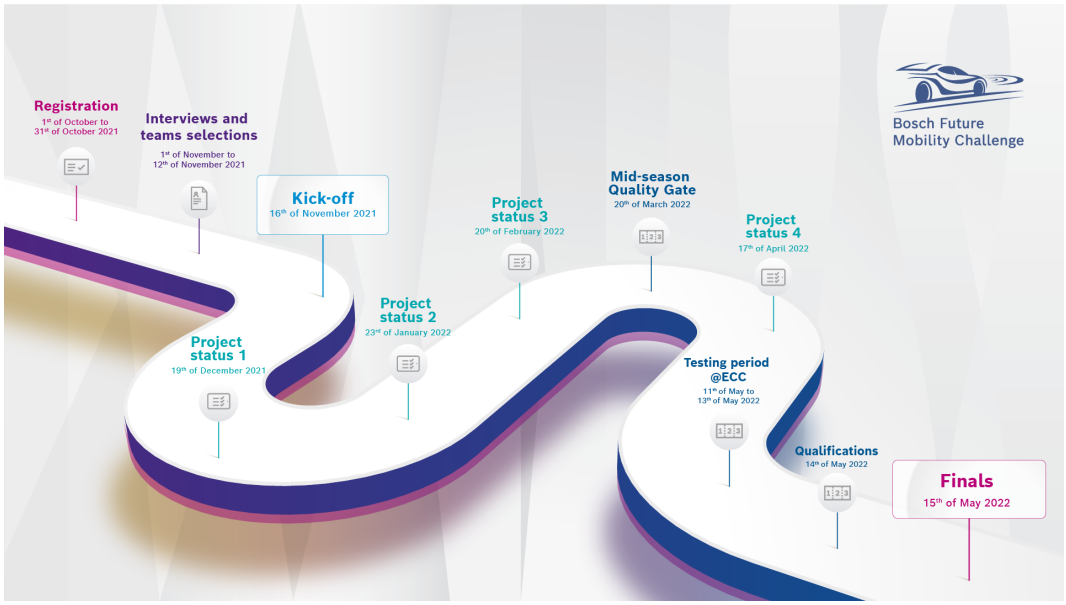

Figure 4: Bosch Future Mobility Challenge 2022 - Timeline.

The author's team managed to reach the finals and competed with other 7 talented teams from Greece, Romania, Portugal and Italy and won the Best new participating team award (Figure 5).

Figure 5: Bosch Future Mobility Challenge 2022 - Best New Participating Team Award.

2.3 The Car-Kit

Going into the details of the car kit provided by Bosch, the following components are to be found:

- Nucleo F401RE.
- Raspberry Pi 4 Model b.
- VNH5012 H-bridge Motor Driver.
- ATM103 Encoder.
- DC/DC converters.
- Servomotor.
- LiPo Battery.
- Chassis.
- Camera.
- IMU Sensor.

The fundamental components are shown in Figure 6.

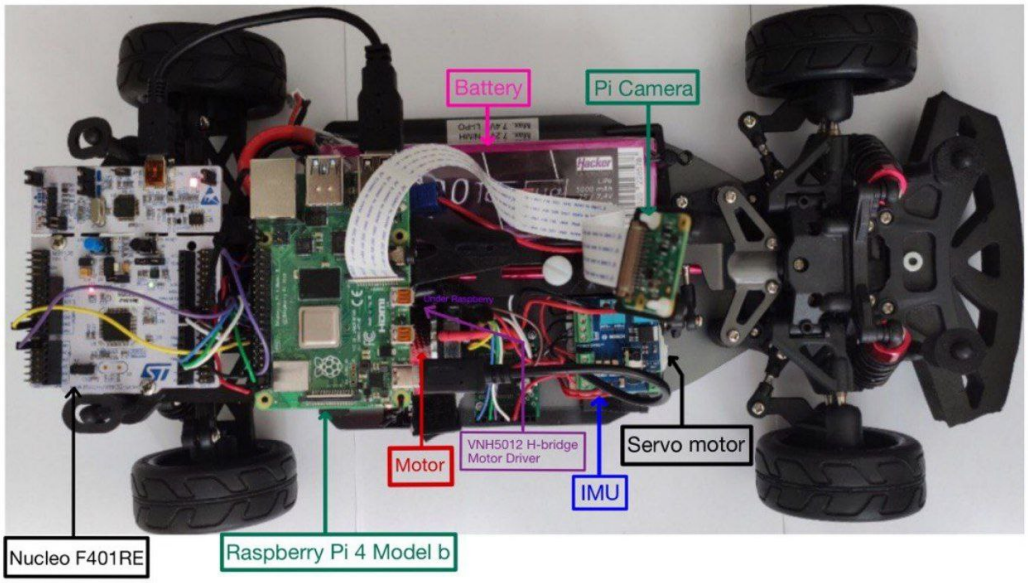

Figure 6: Car with the listed Components.

In addition to these basic elements, the team decided to furnish the car with a LiDAR sensor and an Ultrasonic sensor, placed respectively in the front and the right-hand side of the car.

2.4 The Project

To start working on the project, the teams are provided with a complete documentation necessary to understand better the structure of the project, especially the hardware side and the base Python/C++ codes for the correct communication of all the components of the car. The cited documentation is subdivided as shown in Figure 7.

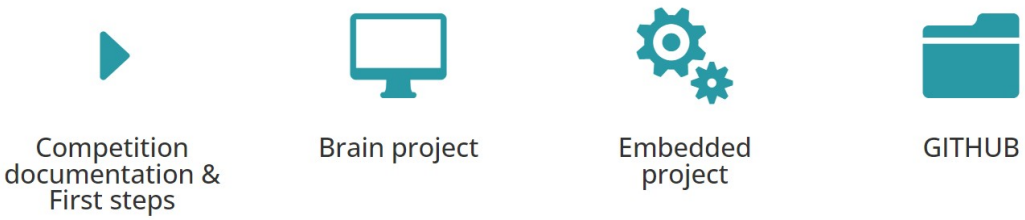

Figure 7: BFMC website - Layout of the Shared Documentation.

A brief explanation of the content of each section is reported in the following subchapters.

2.4.1 Competition Documentation and First Steps

It includes:

- Connection diagram and description with official links to the components of the car.

- Racetrack: the description of the provided racetrack and its elements, the given components, and the diagrams, as well as a starting point and directions of the knowledge required.
- V2X-Vehicle to everything: it includes localization, semaphore, environmental server, and vehicle-to-vehicle communication.
- Printed components and circuit boards.
- Hardware improvements: it includes settings for the hardware components.
- Useful links for Raspberry Pi, ROS, and Python.
- Periodic status: project plan and architecture, reports and media.

2.4.2 Brain Project

The Brain Project describes the given code for the RPi platform. It includes the start-up code and the documentation for the provided API's, which will help the interaction with the V2X systems. The project uses concepts of multi-processing and distributed system, and it implements a basic flexible structure, which can be extended with new features. This folder contains:

- Introduction: concept and architectures, in particular remote car control and camera streaming, installation and configuration, IMU displayer.
- Utils layer: camera streamer, remote control.
- Hardware layer: camera, serial handler process and camera spoofer process.
- Data acquisition layer: traffic lights, localization system, environmental server.

The computer project is already implemented on the provided Raspberry Pi, while the embedded project is already implemented on the Nucleo board. Together, they give a good starting point for the project, providing a remote keyboard control, remote camera stream, constant speed control of the given kit and others.

2.4.3 Embedded Project

This documentation describes the low-level application which runs on the micro-controller Nucleo-F401RE. It aims at controlling the car movement and providing an interface between higher level controllers and lower-level actuators and sensors.

The project has four parts:

- Tools for development containing the instructions to upload the codes related to the correct functioning of the Nucleo.
- Brain layer contains the state machine of the Nucleo (speed and steering).
- Hardware package includes the drivers for actuator and sensors.
- Signal, utils and periodics namespace: 'signal' includes libraries for processing signals, 'utils' package incorporates some util functionalities and 'periodics' layer includes some periodic tasks.

2.4.4 GITHUB

Bosch provided their own link of GitHub in which all the Python/C++ codes related to the topics described above are held. Specifically:

- `Brain` and Brain_ROS: the project includes the software already present on the development board (Raspberry Pi) for controlling the car remotely, use the API's and test the simulated servers, respectively for Raspbian and ROS.
- `Startup\_C`: the project includes some of the scripts transcribed in C++ language from the startup project.
- `Embedded\_Platform`: the project includes the software already present on the Embedded platform (Nucleo board). It describes all the low-level software for controlling the speed and steering of the car.
- `Simulator`: the project includes the software for the Gazebo simulator, which is the official on-line environment of the competition.
- `Documentation`: the project includes all the general documentation of the competition environment, guides, diagrams, car components, etc.

2.5 The Structure behind the Algorithms

The tasks to perform by the end of the competition are the following:

- Lane Keeping and Following.
- Intersection Detection and crossing.
- Correct manoeuvres under the detection of the following traffic signs: stop, priority, crosswalk, parking, roundabout, highway entrance and highway exit, one-way, no entry.
- Parallel and perpendicular parking.
- Object Detection: pedestrian and overtake of a static and/or moving vehicle.
- Navigation by means of nodes and localization system (GPS).

The brain of the car must be inserted in the Raspberry Pi which, basing on the tasks to perform, sends the commands to the Nucleo which, in turn, acts on the motor and on the servo motor to regulate both the speed and steer. More in details, in order to process the image, the Raspberry takes as input the camera frame and the IMU data for the position of the vehicle, runs the specific control algorithms and sends the corresponding output commands to the Nucleo; for example, an increased speed in presence of a ramp which signs the entrance to the highway, a decreased speed and a specific steer when traveling along a tight curve and a zero speed when the traffic light turns red. The correlation between all the project components, the sensors, the algorithms and the vehicle actuation is represented in the project architecture shown in Figure 8.

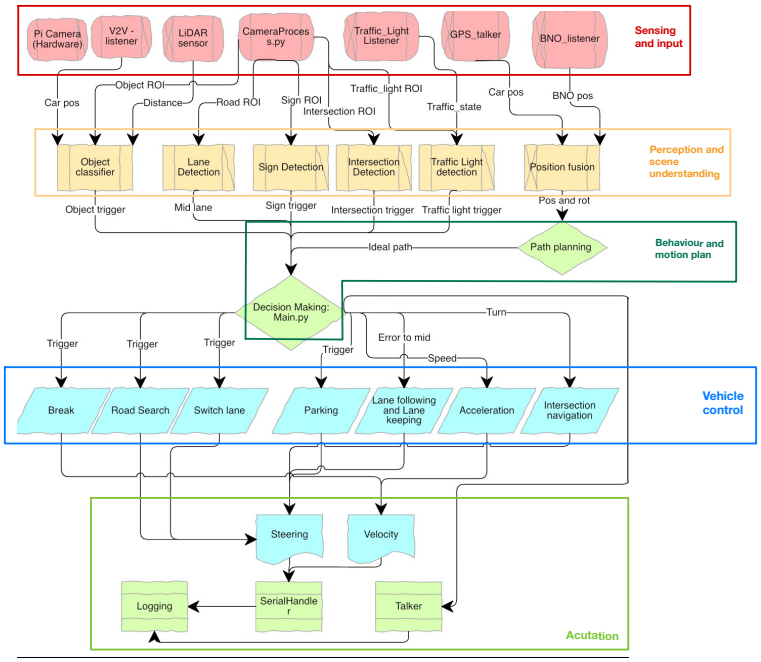

Figure 8: Architecture of the completed project.

The project presented in this chapter sinks the roots for the work developed by three members of team PoliTron: specifically, Cristina Chicca deals with the image processing part, Gianmarco Picariello's work consists in the development of MPC controllers in this context and Claudia Viglietti's thesis concerns optimization algorithms for path planning.

3 Files communication

Before diving into the algorithms dealing with the paper matter, it is of particular interest to define appropriately how the files shown in Figure 8 communicate, both by means of Python3 and using a given server responsible for the car localisation system.

3.1 Parallelism, Thread and Processes

The whole project is composed by processes and threads which ensure that all the algorithms present on the Raspberry Pi run correctly and in parallel since the car, to perform a correct self-drive, needs to execute them concurrently: for example, it has to always follow the lane while respecting traffic and road signs, checking for pedestrians crossing the street etc.

Python multiprocessing library offers two ways to implement process-based parallelism:

- Process: used when functions-based parallelism is required.
- Pool: offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data-based parallelism).

In this project case, the **Process** method has been used: when it is run, it has a specific address on the memory and all the used variables are accessible only by the same process, so they cannot be read by another process unless a pipe is used.

A **Pipe** is a method to pass information from one process to another one: it offers only one-way communication and the passed information is held by the system until it is read by the receiving process. What is returned from the *pipe()* function is a pair of file descriptors (r,w) usable for reading and writing respectively.

In addition to the Process module, the multiprocessing module offers the threading module. The **Thread** class represents an activity that is run in a separate thread of control. This function represents the capability of Python to handle different tasks at the same moment: to sum up, it is a separate flow of execution in the sense that Python script appears to have more threads happening at once. Its syntax is the following:

Thread(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)

In particular, *target* is the callable object to be invoked by the *run()* method, *name* is the thread name and *args* is the argument tuple for the target invocation. Moreover, the *.daemon* property ensures that the daemon thread does not block the main thread from exiting and continues to run in the background.

Multiple threads work in parallel as shown in Figure 9.

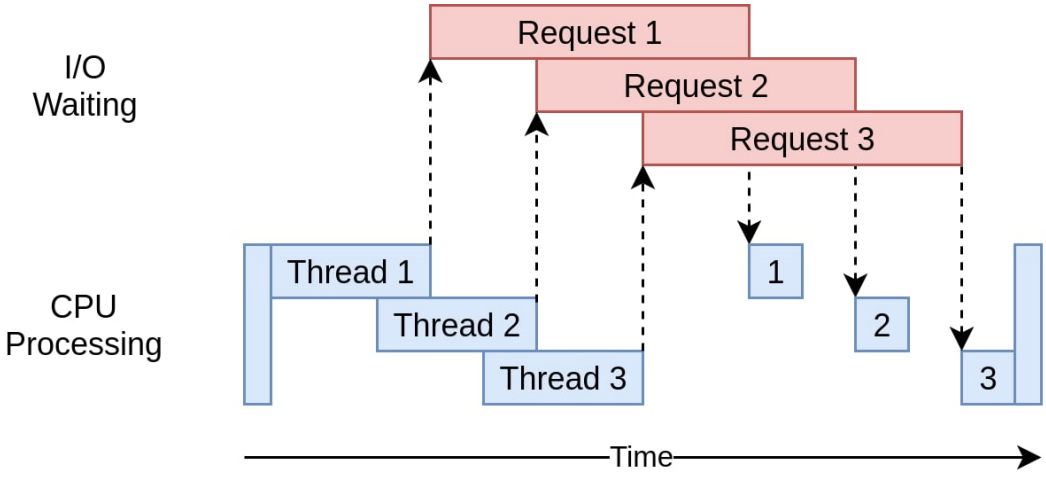

Figure 9: Threads parallelism.

In order to give an idea of how all these functions are related to one another inside a working script, an example in the context of autonomous drive follows: it is necessary to set the commands to send to the STM32 Nucleo microcontroller on a dedicated process and, by means of a pipe, these commands are sent to the *SerialHandlerProcess*, which deals with the communication with the Nucleo. It implements the *WriteThread* function to send commands to Nucleo and the *ReadThread* function to receive data from Nucleo. The commands are generated after having processed the images which come from the *CameraProcess* implementing the Raspberry Pi Cam and they are sent to the *LaneDetectionProcess*, responsible for the lane detection. This means that *LaneDetectionProcess* has to receive two pipes, one for receiving the images and one for sending the commands.

The project contains many processes, each defining a particular algorithm for the self-driving control: the definitions explained until now are useful to outline a structure in common with all these processes, which is shown below.

```
class Name(WorkerProcess):

    def __init__(self, inPs, outPs):
        """ Process used for sending images over the network to a targeted
            IP via UDP protocol (no feedback
        required). The image is compressed before sending it.

        Used for visualizing your raspicam images on remote PC.

        Parameters
        ----------
        inPs : list(Pipe)
            List of input pipes, only the first pipe is used to transfer the
                captured frames.
        outPs : list(Pipe)
            List of output pipes

        In this section you can also define the variables initialization.
        """

        super(MainLaneDetectionProcess, self).__init__(inPs, outPs)
```

```
    def run(self):
        """ Apply the initializing methods and start the threads. """

        super(MainLaneDetectionProcess, self).run()

    def _init_threads(self):
        """ Initialize the sending thread. """

        # Thread for elaborating the received frames:
        receiveFrameT = Thread(name='receiveFrameThread',
            target=self._generate_Output,
                               args=(self.inPs, self.outPs,))
        receiveFrameT.daemon = True
        self.threads.append(receiveFrameT)
```

After creating the class whose name is subjective (in this case it is called *Name*), the utilized functions and methods are the following:

- `__init__`: takes as input `self`, `inPs` and `outPs` which correspond to the list of input pipes and the list of output pipes respectively. This function is called when a class is "instantiated", meaning when the class is declared and any argument withing this function will also be the same argument when instantiating the class object. These initial arguments are the data manipulated throughout the class object. Under this function, some instance attributes are defined and assigned to `self` to be manipulated later on with other functions.
- `super()`: it inherits, uses code from the base (existing) class (i.e., *WorkerProcess*) to define the structure of the new class (i.e., *Name*) – it guarantees the access methods from a parent class within a child class reducing repetitions in the code.
- `run()`: function that initializes the sending thread for the processing of received frames.
- `.append()`: adds a single item to the existing list. It does not return a new list of items, but it will modify the original list by adding the item to the end of the list. After executing the method `append` on the list, the list size increases by one.

All of them send their output to a particular process called *MovCarProcess*: it is responsible for setting the correct values of steer and speed of the car according to the road situation, e.g the value of the lane curve, the detected traffic sign, the intersection etc.. These values are integers representative of the manoeuvre: for example, a value of 999 corresponds to speed equal to 0 in the *SpeedThread*. Summarizing, *MovCarProcess* sets the representative value according to the output received from the control processes, whereas *SpeedThread* and *SteerThread* contain the actual command sent to the Nucleo for, respectively, speed (action 1) and steer (action 2). An example is given by the code shown below, in which the *MovCarProcess* sets the value by means of which the car stops in presence of a STOP or CROSSWALK sign and both *SpeedThread* and *SteerThread* actually build the physical command.

```
""" Extract from MovCarProcess """
    if STOP or CROSSWALK:
```

```
        value = 999

""" Extract from SpeedThread """
    #Stop
    if curveVal == 999:
        command = {'action': '1', 'speed': 0.0}

""" Extract from SteerThread """
    #Stop
    if curveVal == 999:
        command = {'action': '2', 'steerAngle': 0.0}
```

Similarly, these threads will set speed and steer values different from 0 whenever the car has to travel along the path, in absence of road and traffic signs that would impede it.

3.2 The Main.py file

All the processes which have to be run on the Raspberry Pi, including their inputs and outputs, the way in which they communicate, are described inside the *main.py* file. As every main function, it has the job to put together the functions involved in the autonomous-driving solution, searching them from their specific folder.

```
    ArcShRead, ArcShSend = Pipe(duplex=False) # for serial handler

    FrameRead1, FrameSend1 = Pipe(duplex=False) # Frame towards Lane
        Detection
    FrameRead2, FrameSend2 = Pipe(duplex=False) # Frame towards
        Intersection Detection
    FrameRead3, FrameSend3 = Pipe(duplex=False) # Frame towards Sign
        Detection
    FrameRead4, FrameSend4 = Pipe(duplex=False) # Frame towards
        Localization Process

######## IMAGE PROCESSING ALGORITHMS ##########
    curveValRead, curveValSend = Pipe(duplex=False)
    IntersectionRead, IntersectionSend = Pipe(duplex=False)
    SignDetRead, SignDetSend = Pipe(duplex=False)

######## LOCALISATION ALGORITHMS ##########
    LocalizationRead1, LocalizationSend1 = Pipe(duplex=False)

######## PROCESSES ##########
    AshProc = SerialHandlerProcess([ArcShRead], []) #receives the data from
        MovCar and sends it to the Nucleo
    allProcesses.append(AshProc)

    AcamProc = CameraProcess([], [FrameSend1, FrameSend2, FrameSend3,
        FrameSend4])
    allProcesses.append(AcamProc)

    ALaneProc = MainLaneDetectionProcess([FrameRead1], [curveValSend])
    allProcesses.append(ALaneProc)

    AInterProc = IntersectionDetectionProcess([FrameRead2],
```

```
    [IntersectionSend])
allProcesses.append(AInterProc)

ASignProc = SignDetectionProcess([FrameRead3],[SignDetSend])
allProcesses.append(ASignProc)

AtrajProc = RaceTrajectoryProcessSO([FrameRead4], [LocalizationSend1])
allProcesses.append(AtrajProc)

AEnvProc = EnvironmentalProcessSO([LocalizationRead1], [])
allProcesses.append(AEnvProc)

AcurveValProc = MovCarProcess([curveValRead, IntersectionRead,
    SignDetRead, LocalizationRead1],[ArcShSend])
allProcesses.append(AcurveValProc)
```

The example above shows an extract from the *main.py* file: a **pipe** is defined for every process which has to receive the frame from the camera as input (in this case, there are 4 processes which require it) and also, a pipe for localisation and serial handler data is defined. Then, every process is declared, in the first brackets the inputs are listed, whereas in the second brackets the outputs are listed. *CameraProcess* has no input but only the frames to send as output, whereas *SerialHandlerProcess* has the output of *MovCarProcess* as input and no output. It is important to highlight that not all the processes receive as input the camera frames: *EnvironmentalProcess*, responsible for sending the encountered objects to the server, receives as input the coordinates of the car from the *RaceTrajectoryProcess*, whereas *MovCarProcess* receives the inputs from the other processes (car localisation and objects detection) and sends the commands to *SerialHandlerProcess*.

3.3 Server Communication and UDP

The car has an indoor localisation system which detects and sends by **UDP connection** the relative position of itself and other cars present on the race track.

The UDP communications describe the programming for the User Datagram Protocol provided in the TCP/IP to transfer datagrams over a network. Informally, it is called "Send and Pray" because it has no handshake, session or reliability, meaning it does not verify that the protocol has reached the destination before it sends data. UDP has a 8-byte header that includes source port, destination port, packet length (header and data) and a simple (and optional) checksum.

The checksum, when utilized, provides limited integrity to the UDP header and data since it is simply an algorithm-based number created before data is sent to ensure data is intact once received: this procedure is done by running the same algorithm in the received data and comparing the number before and after the reception.

UDP avoids the overhead associated with connections, error checks and retransmission of missing data, it is suitable for real-time or high performance applications that does not require data verification or correction. In fact, the IP network delivers datagrams that can be up to 65507 bytes in length but does not guarantee that they are delivered at the destination and in the same order as they are sent. Moreover, UDP provides pre-process addressing through ports where IP provides addressing of a specific host. The process is described as follows:

1. These ports are 16-bit values used to distinguish different senders and receivers at each end point.

2. Each UDP datagram is addressed to a specific port at the end host and incoming UDP datagrams are demultiplexed between the recipients.

The advantage of using UDP is the absence of retransmission delay, meaning it is fast and suitable for broadcast. The disadvantage regards no guarantee of packets ordering, no verification of the readiness of the receiving computer and no protection against duplicate packets. Anyway, UDP is often used for streaming-type devices such as lidar sensors, cameras and radars since there is no reason to resend data if it is not received. Moreover, due to high data rates, resending past and corrupted data would slow things down tremendously. A comparison between TCP and UDP is given by Figure 10.

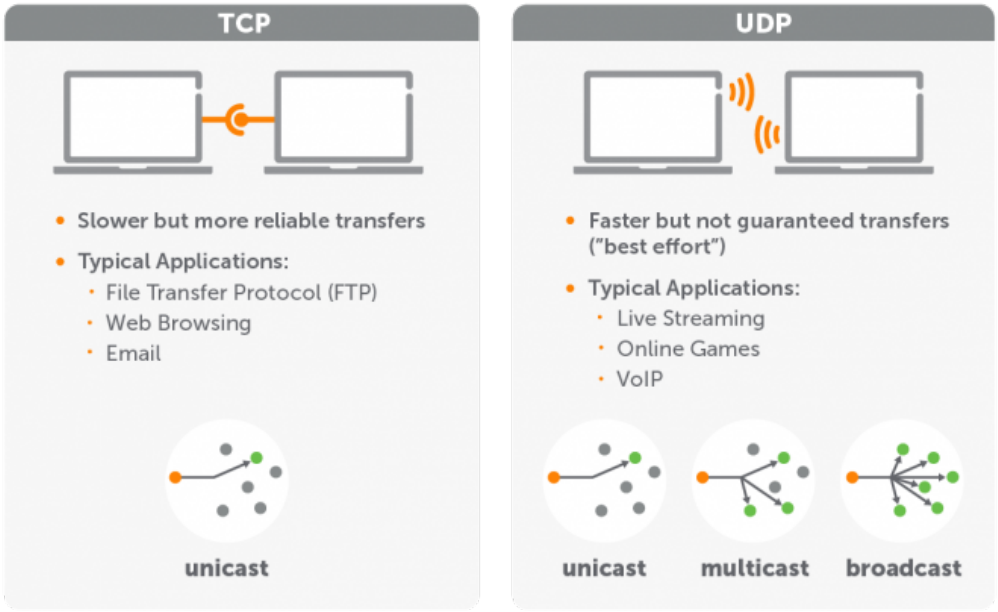

Figure 10: Comparison between TCP and UDP protocols.

The connection between the car and the server is validated by means of the **API communication**, which ensures the reading of the car given ID together with a certain port responsible for the communication of the coordinates of all the moving obstacles. An API communication is a type of Application Programming Interface which adds communication channels to a particular software. It allows two pieces of software hosted on the cloud to connect to each other and transfer information.

An example of the UDP protocol used inside the project is given by a file responsible for reading the position of the car in real time (*position_listener.py*).

```
import sys
sys.path.insert(0,'.')

import socket
import json
from complexDealer import ComplexDecoder
```

```
class PositionListener:
   """PositionListener aims to receive all message from the server.
   """
   def __init__(self, server_data, streamPipe):

      self.__server_data = server_data

      self.__streamP_pipe = streamPipe

      self.socket_pos = None

      self.__running = True

   def stop(self):
      self.__running = False
      try :
         self.__server_data.socket.close()
      except: pass

   def listen(self):
      while self.__running:
         if self.__server_data.socket != None:
            try:
               msg = self.__server_data.socket.recv(4096)

               msg = msg.decode('utf-8')
               if(msg == ''):
                  print('Invalid message. Connection can be interrupted.')
                  break

               coor = json.loads((msg),cls=ComplexDecoder)
               self.__streamP_pipe.send(coor)
            except socket.timeout:
               print("position listener socket_timeout")
               # the socket was created successfully, but it wasn't received
                   any message. Car with id wasn't detected before.
               pass
            except Exception as e:
               self.__server_data.socket.close()
               self.__server_data.socket = None
               print("Receiving position data from server " +
                   str(self.__server_data.serverip) + " failed with error: "
                   + str(e))
               self.__server_data.serverip = None
               break
      self.__server_data.is_new_server = False
      self.__server_data.socket = None
      self.__server_data.serverip = None
```

Similarly to the Process object, the class `PositionListener` is composed by the main functions `__init__`, `stop` and `listen`. In this case, the variables of interest are `server_data`, `streamPipe` and `socket`.

A network socket is a software structure within a node of a computer network that serves as an endpoint for sending and receiving data. The structure and properties of a socket are defined by an API for the networking architecture. Sockets are created

only during the lifetime of a process of an application running in the node.

The function `listen` performs the following steps:

1. After the subscription on the server, it is listening the messages on the previously initialized socket.
2. It decodes the messages and saves in 'coor' member parameter.
3. Each new message will update the 'coor' parameter and the server will send the result (car coordinates) of last detection. If the car has been detected by the localization system, the client receives the same coordinates and timestamp.

The UDP socket programming fundamentals are represented by Figure 11.

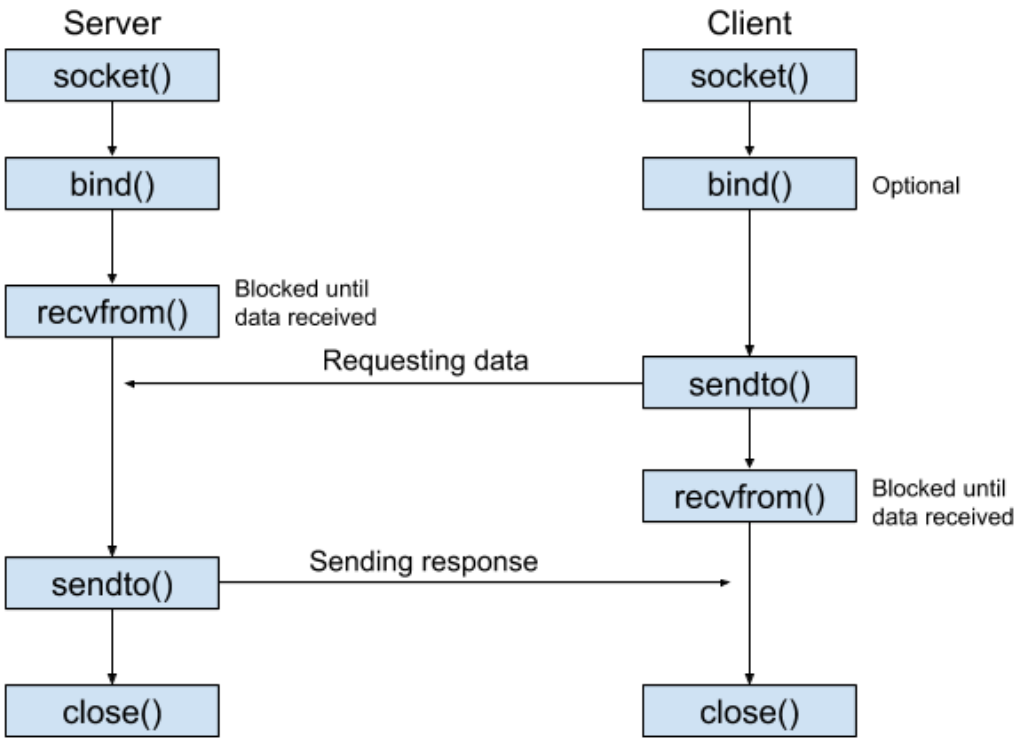

Figure 11: Fundamentals of UDP socket programming.

4 Image Processing

Image Processing is a computer field which deals with processing and extracting meaningful and useful data from images. Since they are unstructured data, they are significantly more difficult than processing structured data such as tables and forms. Image Processing is a large and growing branch currently used in multiple areas such as Medical, Machine/Robot Vision, Driverless cars, etc.

In particular, much study has been done in the field of image detection for Autonomous cars: features such as lane detection, traffic sign detection and methods for implementation were reported by many researchers around the world.

In this paper, some of these methods are proposed, specifically used for the BFMC2022. The first element to analyze is how the images are received by the car.

The car-kit the team has received was only composed by two main hardware components which constituted the whole communication between the car and the surrounding environment: the front-facing camera and the IMU sensors. This is because, even though the most technological cars rely upon a wide variety and quantity of sensors, the mass production of autonomous vehicles is not feasible considering all this expensive equipment. As regarding the camera, not only is it the cheapest sensor available on the market, but it also is the very first element utilized by the car to sense the environment and act consequently. The main idea is due to the fact that if the human can drive using only the eyes, then the autonomous vehicle can do the same.

This preface gives a hint of the importance involving Image Processing, since every algorithm explained in this section takes as input the image coming from the camera.

The camera is shown in Figure 12.

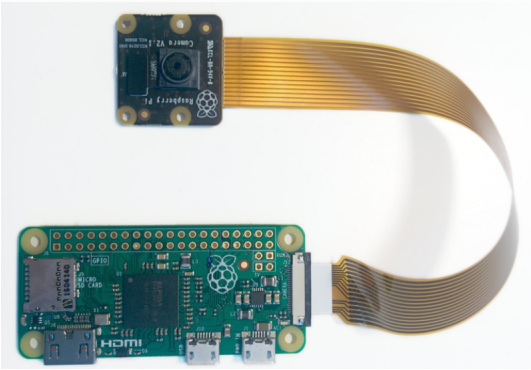

Figure 12: PiCamera v2.1 set-up.

The Raspberry Pi Camera Module v2 is the newest official camera board which connects to any Raspberry Pi or Computer Module [5]. Its main features are the following:

- Fixed focus lens on-board.
- Improved resolution - 8 megapixel native resolution sensor-capable of 3280 x 2464 pixel static images.
- Supports 1080p30, 720p60 and 640x480p90 video.

- Uses the Sony IMX219PQ image sensor - high-speed video imaging and high sensitivity.
- Optical size of 1/4".

As anticipated in chapter 3, in addition to other Python files responsible for the correct communication between the Raspberry and the camera, the main camera file is the *CameraThread*. Similar to the Process structure, the Thread is mainly composed by the `init` and `run` functions: in the former, the camera settings are reported, such as resolution, framerate, brightness, shutter speed, contrast, iso and image size, in the latter the frame sequence is captured and recorded. In particular, the frame rate has been set to 10fps and the image size is the standard one (640x480 pixels). The other settings can be changed according to anyone's preference but in this project they have been kept with the default values, as written in Table 1. Another important feature is the color space of the received image: from the

Camera Resolution	(2592, 1944)
Camera Brightness	50
Camera Shutter Speed	2000000
Camera Contrast	0 (auto)
Camera ISO	0 (auto)

Table 1: Camera settings.

Thread, it is immediately converted into the **RGB format**. RGB represents the three-dimensional colour space in which channel 0 si red, channel 1 is green and channel 2 is blue.

Once the image is correctly received from the camera at the speed of 10 frames per second (trade-off between latency of computation and quality of the image), it is processed by means of Python algorithms. The most utilized Python libraries are, of course, **OpenCV** and **NumPy**: the former is a computer vision and machine learning library and the latter is a library for array manipulation. The two are related since an OpenCV image is represented as NumPy array, which is also very flexible and compatible with many other libraries.

By means of these libraries, it is possible to elaborate the frames, which stands for:

1. Read the frame.
2. Manipulate the image.
3. Select the Region of Interest (ROI) in order to pick only the image section of interest according to the algorithm, as shown in Figure 13.

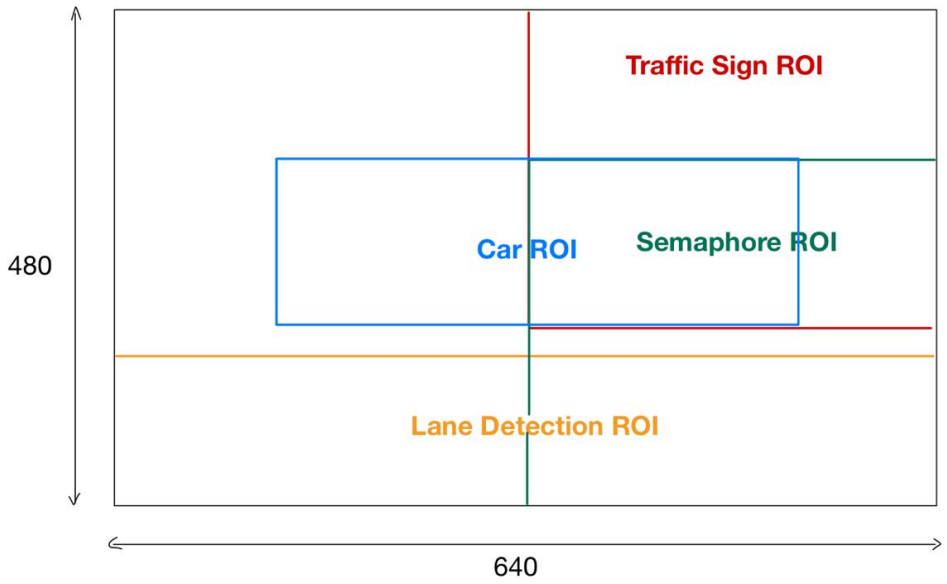

Figure 13: ROIs for some Image Processing algorithm.

4. Apply the controls for detecting the specific object present in the image.

The list of Processes used in the project and responsible for the Image Processing implementation are the following:

- *MainLaneDetectionProcess*
- *IntersectionProcess*
- *TrafficSignDetectionProcess*
- *TrafficLightProcess*
- *ObjectDetectionProcess*

Once every Process has returned the corresponding output (e.g. the value of the curve from the *MainLaneDetectionProcess* or the recognized traffic sign from the *TrafficSignDetectionProcess*), it is sent to the *MovCarProcess*, responsible for acting on both steer and speed of the car according to the road situation. A simplifying scheme is given in Figure 14.

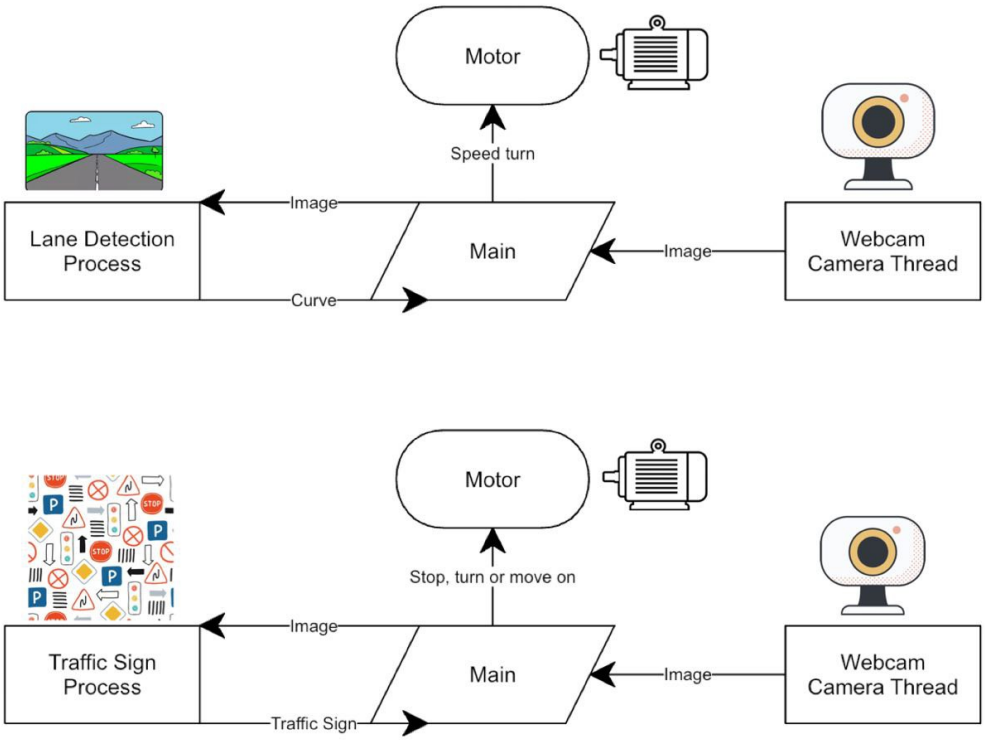

Figure 14: Relationship between Camera, Detection Process and Motor.

This procedure is shared by all the Processes involved in the Image Processing framework and which will be explained in details in the following chapters.

5 Lane Detection and Follow (LD & LF)

The lane detection system involves the process of identifying the lane markings and estimating the position of the vehicle from the centre of the lane. Computer Vision (CV) technologies have played an important role in lane detection domain: the latest CV advances represented by deep learning are robust and efficient, but they are not suitable for embedded platforms where the computing resources are limited. Traditional structured lane detection algorithms, instead, are extremely useful and much lighter than deep learning techniques. They are mainly divided into two paradigms: feature-based methods and model-based methods. The former are much more flexible in the lane diversity and fit well in real-time processing. For this reason, the work proposes two feature-based algorithms for lane detection [6]:

- Lane detection based on **Hough Transform** to detect lines, calculate the slope of the line and generate the value of the curve accordingly.
- Lane detection based on **Perspective correction, Thresholding** and **Histogram statistics** [7].

It is important to highlight the reason why this section deals with *Lane Following* (LF) and *Lane Keeping* (LK): LK intervenes with the steering precisely when lane drifting is about to occur, whereas LF constantly attempts to keep the vehicle at the centre of the lane. The input of the system is, as mentioned previously, the frames captured from the camera mounted on the car, whereas the output is the steering angle in degrees. The two techniques will be compared and implemented on the car.

5.1 Hough Transform method

This method is schematized in Figure 15.

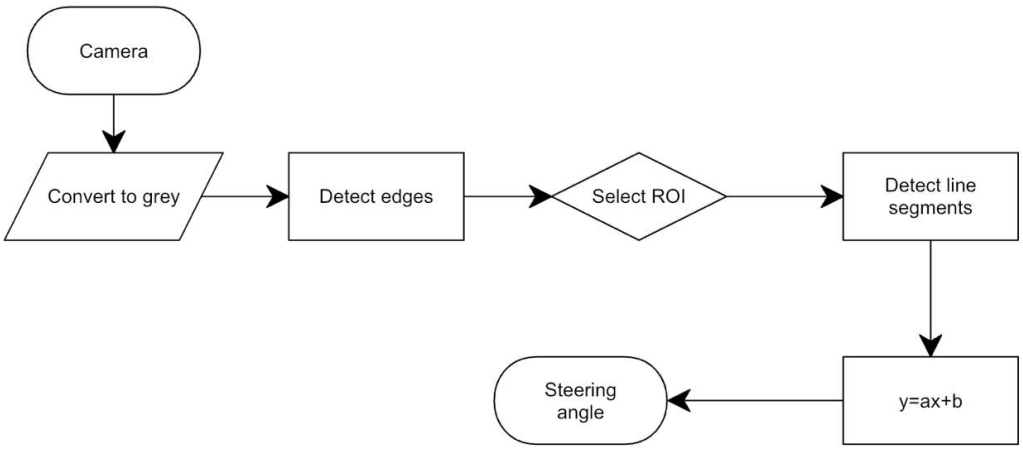

Figure 15: Hough Transform method scheme.

More specifically, this algorithm applies filtering, edge detection, lane detection and offset distance calculation based on the position of the vehicle. After the processing is performed, the control system issues commands to the vehicle to adjust its steering according to the offset so it stays close to the centre of the lane (LF). In the code, the whole procedure is represented by the `_generate_curveVal` function.

```
def _generate_curveVal(self, inPs, outPs):

    try:
        time.sleep(5)

        while True:
            stamps, img = inPs.recv() #take images from Camera Thread

            edges = self.detect_edges(img)
            roi = self.region_of_interest(edges)
            line_segments = self.detect_line_segments(roi)
            lane_lines = self.average_slope_intercept(img, line_segments)
            lane_lines_image = self.display_lines(img, lane_lines)
            steering_angle = self.get_steering_angle(img, lane_lines)

            heading_image = self.display_heading_line(lane_lines_image,
                steering_angle)
            cv2.imshow("masked_edges", roi)
            cv2.imshow("heading line", heading_image)
            steering_angle = steering_angle -90
            outPs[0].send(steering_angle) #send the correct steering angle
                to MovCarProcess

            cv2.waitKey(1)

    except Exception as e:
        print("\nCould not read the image\n")
        print(e)
```

The libraries utilized for this lane detection code are: `math`, `cv2`, `numpy`, `thread` and `time`.

5.1.1 Edge Detection

Removing noise from the image is very important because the presence of noise will affect the detection of the edges: the process of smoothening involves removing noise from the image by using a filter, then an edge operator is applied to generate a binary edge map. This is followed by the application of the Hough Transform to the edge map to detect lines: it is a technique responsible for the isolation of features of a particular shape within an image. Three fundamental steps are performed in edge detection phase:

1. **Smoothening**: application of filters for noise reduction.
2. **Edge point detection**: this operation extracts all the points in the image that are potential members to become edge points.
3. **Edge localization**: is a process to select a member from the edge points only, the points that are truly members of the set of points incorporated in an edge.

In this section, the function `detect_edges` is analyzed:

```
def detect_edges(self, img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # kernel = 5
```

```
    # blur = cv2.GaussianBlur(gray, (kernel, kernel), 0)
    # reduce the noise of the image: if it's not between the 5-by-5 Kernel
        it is deleted
    edges = cv2.Canny(gray, 100, 200) #100 and 200 are min and max gradient
        intensities
    cv2.imshow("Canny",edges)
    cv2.waitKey(1)
    return edges
```

First of all, the image is converted from **BGR** (Blue-Green-Red) into grayscale via the `cv2.cvtColor`, which takes as input arguments the image in question and the color space conversion. Then the filtering technique is applied: among the filters available in OpenCV, the most common one is the `GaussianBlur`, attenuating the intensity of high-frequency signals and for which it is necessary to specify the image, the kernel size and the standard deviation in x and y directions. Anyway, if one uses the `cv2.Canny` function, the Gaussian filter is already applied as default, since it performs the following steps:

1. Denoise the image with a 5x5 Gaussian filter.
2. Compute the intensity gradients: the smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (Gx) and vertical direction (Gy).

$$Edge\_gradient(G) = \sqrt{Gx^2 + Gy^2}$$

3. Apply non-maximum suppression (NMS) on the edges - the algorithm selects the best edges from a set of overlapping edges.
4. Apply a double threshold to all the detected edges to eliminate any false positive.
5. Analyse all the edges and their connection to each other to keep the real edges and discard the weak ones.

With those settings, any edge with intensity gradient greater than 200 is sure to be an edge, whereas those below 100 are sure to be non-edge: the edges lying between these thresholds are classified edges or non-edges based on their connectivity. If they are connected to sure-edge pixels, they are considered part of edges, discarded otherwise.

5.1.2 Region Of Interest

Computing **Regions of Interest** (ROIs) considerably simplifies the interaction with image data, since a rectangular region in NumPy is easily defined with an array slice. The aim of ROIs is to keep the car focus on a particular section of the image (in this case, the lane lines), ignoring anything else in the surrounding environment.

```
def region_of_interest(self, canny):
    height = canny.shape[0] #first dimension value
    width = canny.shape[1] #second dimension value
    mask = np.zeros_like(canny) #mask initialization
    shape = np.array([[(0, height), (width, height), (width, 320), (0,
        320)]], np.int32) #polygon
```

```
    cv2.fillPoly(mask, shape, 255) #mask with polygon size
    masked_image = cv2.bitwise_and(canny, mask) #final result
    cv2.imshow("Canny mainlane", masked_image)
    cv2.waitKey(1)
    return masked_image
```

The function `region_of_interest` takes the edged frame (output of Canny function) and draws a polygon with four pre-set points: to perform this task, first the dimensions of the canny image are extracted, then a mask is initialized using the `np.zeros_like` function based on the canny image size. Afterwards, the polygon (*shape*) is created and the previously created mask is filled with this region using the `cv2.fillPoly` function: the last argument represents the color of the fill, white in this case. It is important to notice that here the mask is selected taking into consideration only the lower half of the camera video: considering the image size as 640x480, the first tuple represents the lower left corner, the second tuple is the upper left corner, the third one is the upper right corner and the final tuple is the lower right corner. `cv2.bitwise_and` performs the concatenation between two arrays, the canny image and the mask, which results in a final output containing the original image and the mask overlapped.

5.1.3 Lanes Detection

Detection of line segments from the edged frame can be performed using the Hough Transform method [8]: with this transform, straight lines passing through edge points can be drawn, using a mapping from the Cartesian space to a parameter space. As known by the reader, a line in the image can be expressed with two variables, depending on the chosen coordinate system, for example in the Polar coordinate system a line equation can be written as:

$$y = (cos\theta/sin\theta)x + (r/sin\theta)$$

In general, for each point ($x0$,$y0$), a family of lines going through that point can be defined as

$$r_\theta = x_0 * cos\theta + y_0 * sin\theta$$

The family of lines that goes through the given point is a sinusoid; in the plane θ - r, only the points such that $r > 0$ and $0 < \theta < 2\pi$ are considered. The same operation can be done for all the points in an image: if the curves of two different points intersect in the plane θ - r, they both belong to the same line. The more curves intersect, the more points are contained in the line represented by that intersection so, in general, a threshold for the minimum number of intersections needed to detect a line is defined.

The advantage of using Hough Transform is that the pixels forming a line need not to be contiguous with that of other pixels; therefore, it is a useful tool for detecting lines with short breaks inside them due to noise or partial occlusion by objects.

OpenCv provides two functions, *HoughLines* and *HoughLinesP*. The former uses the standard Hough Transform and the latter uses the probabilistic version of the same, which analyses a subset of the image points and estimates the probability that these points all belong to the same line. Since the latter is an optimized version, it is implemented in the code in a way that it returns a representation of each line as a single point and an angle without information about endpoints.

```
def detect_line_segments(self, cropped_edges):
    rho = 1
    theta = np.pi / 180
    min_threshold = 20

    line_segments = cv2.HoughLinesP(cropped_edges, rho, theta,
        min_threshold, np.array([]), minLineLength=20, maxLineGap=200)

    return line_segments
```

The arguments of the `HoughLinesP` are:

- `cropped_edges`: output of the edge detector (binary image).
- `rho`: resolution of the parameter r in pixels (1 as default).
- `theta`: resolution of the parameter θ in radians (1 degree).
- `min_threshold`: minimum number of intersections to detect line.
- `lines`: empty array.
- `minLineLength`: minimum number of points that can form a line.
- `maxLineGap`: maximum gap between two points to be considered in the same line.

The final two values have been found after many experimental trials.

5.1.4 Calculate and Display Heading Lines

The heading line is the central line which is responsible for giving the steering servomotor the direction in which it should rotate and giving the throttling DC motor the speed at which it should operate. The computation of such a line is a pure trigonometric operation and the extreme cases in which the car finds only one or no lane line are studied in the `if-elif-else` conditions.

```
def get_steering_angle(self, frame, lane_lines):
        height, width, _ = frame.shape

        if len(lane_lines) == 2:
            left_x1, left_y1, left_x2, left_y2 = lane_lines[0][0]
            right_x1, right_y1, right_x2, right_y2 = lane_lines[1][0]

            slope_l=math.atan((left_x1-left_x2) / (left_y1-left_y2))
            slope_r=math.atan((right_x1-right_x2) / (right_y1-right_y2))

            slope_ldeg = int(slope_l * 180.0 / math.pi)
            steering_angle_left = slope_ldeg

            slope_rdeg = int(slope_r * 180.0 / math.pi)
            steering_angle_right = slope_rdeg

            if left_x2>right_x2: #horizontal line
                if abs(steering_angle_left) <= abs(steering_angle_right):
                    x_offset = left_x2 - left_x1
```

```
                y_offset = int(height / 2)
            elif abs(steering_angle_left) > abs(steering_angle_right):
                x_offset = right_x2 - right_x1
                y_offset = int(height / 2)
        else: #normal left line
            mid = int(width / 2)
            x_offset = (left_x2 + right_x2) / 2 - mid
            y_offset = int(height / 2)

    elif len(lane_lines) == 1:
        x1, _, x2, _ = lane_lines[0][0]
        x_offset = x2 - x1
        y_offset = int(height / 2)

    elif len(lane_lines) == 0:
        x_offset = 0
        y_offset = int(height / 2)

    #angle_to_mid_radian = math.atan(x_offset / y_offset)
    alfa = 0.6

    angle_to_mid_radian = alfa*self.angle +(1-alfa)*math.atan(x_offset
        / y_offset)
    angle_to_mid_deg = int(angle_to_mid_radian * 180.0 / math.pi)
    steering_angle = angle_to_mid_deg +90
    self.angle = angle_to_mid_radian

    return steering_angle
```

The `get_steering_angle()` function is responsible for steering the car correctly according to the curvature value. In the most general case in which the steering angle is not 90°, if it is greater than this quantity the car will steer on the right, on the left otherwise, being 90° the reference for going straight on, as shown in Figure 16.

Figure 16: Steering angle variation according to the value of curvature.

The function is divided into 3 main parts:

1. **lane lines = 2**: two cases can happen. In the case the line on the left is, in reality, an intersection line, the code ignores the steepest one: this happens when the two lines, left and right, intersect, which means when the right-most point of the left line is greater than the left-most point of the right line. The same reasoning is done when the right line is an intersection. Otherwise, the points of the line are calculated as usual:
    - `x_offset` checks how much the average differs from the middle of the screen.

- `y_offset` is always `height/2`. To better clarify these points, the Figure 17 shows them on the plane.

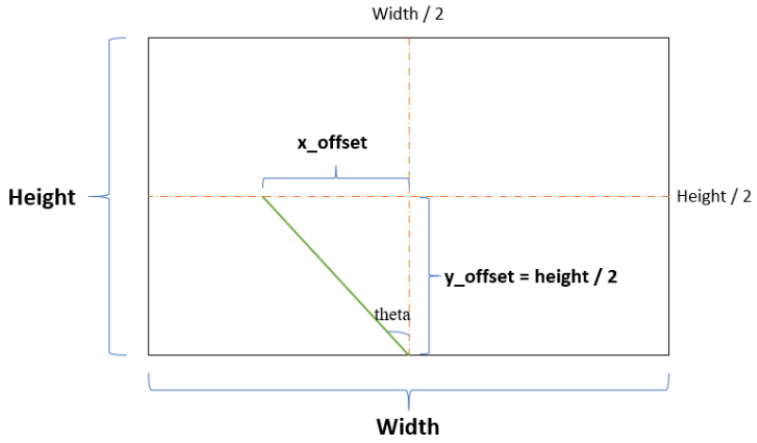

Figure 17: Representation of x_offset and y_offset for calculation of steering angle.

2. **lane lines = 1**: in the case the detected lane is only one, the car will follow it anyway.

3. **lane lines = 0**: the car does not perform any lane keeping, consequently the steering angle is 90°.

Then, the central line is displayed using the `display_heading_line()` function. This function is very similar to the one used to display also the lateral lines.

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

        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 / 1.75)

        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
```

To be noticed that the important parameters of this function are the image in which the line has to be displayed, the color, the line width and the 4 points defining the line (the heading line is the green one shown in Figure 19).

5.2 Perspective correction and Histogram statistics method

In this chapter, a multi-line detection algorithm based on histogram statistics is proposed for the track-following application. This method is summarized in Figure 18.

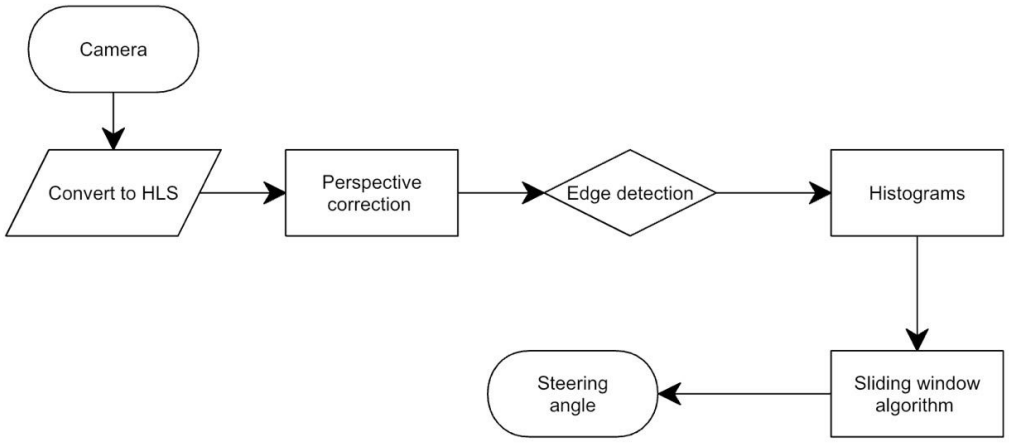

Figure 18: Perspective correction and histogram statistics method scheme.

After the preprocessing of the original image and projecting, the pixel histogram in the bird eye view space can be obtained and the starting points of the lane detection are obtained by filtering and clustering the histogram.

Subsequently, the sliding windows are moved to capture the pixels on the lines and, finally, the quadratic curves are fitted as the model of the lines and are projected back to the original image space. Theoretically, and based on the work done in [9], this type of algorithm can deal with multi-curve or cross horizontal lines with better robustness.

As it can be seen from Figure 18, the last step is coincident with that of the previously explained method, so the functions `display_heading_line()` and `get_steering_angle()` are the same.

```
def _generate_curveVal(self, inPs, outPs):

    try:
        self.compute_perspective(1024, 600, [160, 500], [500, 310], [546,
            310], [877, 500])

        while True:
            stamps, img = inPs.recv() #take images from Camera Thread
            img = cv2.resize(img,(640,480))
            img_warped = self.warp(img)
            img_hls = cv2.cvtColor(img_warped,
                cv2.COLOR_BGR2HLS).astype(np.float64)
            img_edge = self.edge_detection(img_warped[:, :, 1])
            (img_binary_combined, img_binary_solo) =
                self.threshold(img_hls[:, :, 1], img_edge)
            hist = self.histogram(img_binary_combined)
            hist_solo = self.histogram(img_binary_solo)
            left_lanes = []
            right_lanes = []
```

```
            ### LANES ON HISTOGRAM ###
            if (len(left_lanes) > self.MIN_DETECTIONS):
                lanes = self.lanes_partial_histogram(hist, left_lanes,
                    right_lanes, 30)
            else:
                lanes = self.lanes_full_histogram(hist)

            ### SLIDING WINDOW ###
            ret, sw = self.slide_window(img_warped, img_binary_combined,
                lanes, 15)
            if ret:
                left_lanes.append(deepcopy(sw.left))
                right_lanes.append(deepcopy(sw.right))
            else:
                # In case of problems, use the previous detection
                sw.left = left_lanes[len(left_lanes) - 1]
                sw.right = right_lanes[len(right_lanes) - 1]

                left_lanes.append(sw.left)
                right_lanes.append(sw.right)

            img_lane = self.show_lanes(sw, img_warped, img)

            #### LANE FOLLOW PART ####
            steering_angle = self.get_steering_angle(img, img_lane)
            #cv2.imshow('result',img_lane)

            heading_image = self.display_heading_line(img_lane,
                steering_angle)
            cv2.imshow('result',heading_image)
            steering_angle = steering_angle -90
            outPs[0].send(steering_angle) #send the correct steering angle
                to MovCarProcess

            cv2.waitKey(1)

        except Exception as e:
            print("\nCould not read the image\n")
            print(e)
```

5.2.1 Perspective Correction

The perspective depends on the focal length of the lens and the position of the camera. Once the camera is mounted on the car, the perspective is fixed so it can be taken into consideration to correct the image. To this purpose, there are a couple of basic OpenCv functions: `getPerspectiveTransform` and `warpPerspective`. The former computes the perspective transformation which takes as inputs 2 arrays composed by 4 points, one is the source with the original perspective and the other one is the destination with the desired perspective. The latter is used to obtain the bird's eye view from the perspective correction previously calculated. Thanks to this view, now the road lines are seen as parallel (vertical).

```
# pt1, pt2, ptr3, and pt4 are four points defining a trapezoid used for
    the perspective correction
def compute_perspective(width, height, pt1, pt2, pt3, pt4):

    perspective_trapezoid = [(pt1[0], pt1[1]), (pt2[0], pt2[1]), (pt3[0],
        pt3[1]), (pt4[0], pt4[1])]
    src = np.float32([pt1, pt2, pt3, pt4])
    # widest side on the trapezoid
    x1 = pt1[0]
    x2 = pt4[0]
    # height of the trapezoid
    y1 = pt1[1]
    y2 = pt2[1]
    h = y1 - y2
    # The destination is a rectangle with the height of the trapezoid and
        the width of the widest side
    dst = np.float32([[x1, h], [x1, 0], [x2, 0], [x2, h]])
    perspective_dest = [(x1, y1), (x1, y2), (x2, y2), (x2, y1)]

    perspective_correction = cv2.getPerspectiveTransform(src, dst)
    perspective_correction_inv = cv2.getPerspectiveTransform(dst, src)
    warp_size = (width, h)
    orig_size = (width, height)

def warp(img, filename):
    img_persp = img.copy()

    cv2.line(img_persp, perspective_dest[0], perspective_dest[1], (255,
        255, 255), 3)
    cv2.line(img_persp, perspective_dest[1], perspective_dest[2], (255,
        255, 255), 3)
    cv2.line(img_persp, perspective_dest[2], perspective_dest[3], (255,
        255, 255), 3)
    cv2.line(img_persp, perspective_dest[3], perspective_dest[0], (255,
        255, 255), 3)

    cv2.line(img_persp, perspective_trapezoid[0], perspective_trapezoid[1],
        (0, 192, 0), 3)
    cv2.line(img_persp, perspective_trapezoid[1], perspective_trapezoid[2],
        (0, 192, 0), 3)
    cv2.line(img_persp, perspective_trapezoid[2], perspective_trapezoid[3],
        (0, 192, 0), 3)
    cv2.line(img_persp, perspective_trapezoid[3], perspective_trapezoid[0],
        (0, 192, 0), 3)

    save_dir(img_persp, "persp_", filename)

    return save_dir(cv2.warpPerspective(img, perspective_correction,
        warp_size, flags=cv2.INTER_LANCZOS4), "warp_",
                    filename)
```

To be noticed that, in addition to the original image, the perspective correction and the size of the warp, `warpPerspective` takes as input also the type of interpolation which in this case is `INTER_LANCZOS4`.

5.2.2 Thresholding

This step is almost the same as the Edge Detection one utilized in the Hough Transform Lane Keeping method, the only difference consists in how the final result is achieved: instead of being converted into grayscale, the original image is converted into **HSL color space** (Hue, Saturation, Lightness), since the green channel can be used for edge detection and the L channel can be used as additional thresholding.

Moreover, instead of using the Gaussian filter, the Scharr method is used, which computes a derivative detecting the difference in colors in the image.

```
def edge_detection(channel, filename):
    edge_x = cv2.Scharr(channel, cv2.CV_64F, 1, 0) # Edge detection using
        the Scharr operator
    edge_x = np.absolute(edge_x) #no matter the sign of the derivative

    return save_dir(np.uint8(255 * edge_x / np.max(edge_x)), "edge_",
        filename)
```

In order to select the intensity of the pixels to choose, the interpolated threshold technique is utilized: a higher threshold is applied to the bottom of the image where there is a better resolution, a sharper image and more noise, whereas a lower threshold is applied on the top where the pixels are stretched by the perspective correction.

```
def threshold(channel_threshold, channel_edge, filename):
    # Gradient threshold
    binary = np.zeros_like(channel_edge)
    height = binary.shape[0]

    # Interpolated threshold
    threshold_up = 15
    threshold_down = 60
    threshold_delta = threshold_down - threshold_up

    for y in range(height):
        binary_line = binary[y, :]
        edge_line = channel_edge[y, :]
        threshold_line = threshold_up + threshold_delta * y / height
        binary_line[edge_line >= threshold_line] = 255

    save_dir(binary, "threshold_edge_only_", filename)
    save_dir(channel_threshold, "channel_only_", filename)

    binary[(channel_threshold >= 140) & (channel_threshold <= 255)] = 255

    binary_threshold = np.zeros_like(channel_threshold)
    binary_threshold[(channel_threshold >= 140) & (channel_threshold <=
        255)] = 255

    return (save_dir(binary, "threshold_", filename),
        save_dir(binary_threshold, "threshold_other", filename))
```

5.2.3 Histograms

Since the lines have been found in the image, the histograms are used to know exactly where the lines are placed. Using vertical lines, one way could be to count the white pixels on a certain column, but since, in a turn, the lines are not vertical, the bottom part of the image is selected so that the lines can be considered almost vertical. The histograms show that there are basically two peaks in correspondence of the pixels in which the lines are present, so the `argmax` function is used to calculate such peaks.

```
def histogram(img):
    partial_img = img[img.shape[0] * 2 // 3:, :] # Select the bottom part
        (one third of the image)
    hist = np.sum(partial_img, axis=0)
    size = len(hist)

    # Detect the peaks by splitting the array into two halves
    max_index_left = np.argmax(hist[0:size // 2])
    max_index_right = np.argmax(hist[size // 2:]) + size // 2

    return HistLanes(max_index_left, max_index_right, hist[max_index_left],
        hist[max_index_right])
```

At the end, the value `hist[index]` itself can be considered the confidence of having identified the lane, since more pixels mean more confidence.

5.2.4 Sliding Window Method

At this point, the reader is only aware of where the line starts, but has no clue on where it ends: the solution is to focus on the area around the line and proceed to "follow" it. This is done using a **Sliding Window algorithm**: a rectangle is created representing the window of interest. The functioning principle is explained as follows:

- The first window on the bottom of each lane is centred on the respective peak of the histogram.
- The width of each window depends on the selected margin and the height depends on the chosen number of windows: these two numbers can be changed to reach a balance between a better detection and the possibility to detect more difficult turns with a smaller radius.

```
def slide_window(img, binary_warped, hist, num_windows):
    img_height = binary_warped.shape[0]
    window_height = np.int(img_height / num_windows)
    # Indices (e.g. coordinates) of the pixels that are not zero
    non_zero = binary_warped.nonzero()
    non_zero_y = np.array(non_zero[0])
    non_zero_x = np.array(non_zero[1])
    # Current positions, to be updated while sliding the window; we start
        with the ones identified by the histogram
    left_x = hist.x_left
    right_x = hist.x_right
    # Movement we are allowing to the lane
```

```
    margin = 80
    # Set minimum number of pixels found to recenter window
    min_pixels = 25
    left_lane_indexes = []
    right_lane_indexes = []

    for idx_window in range(num_windows):
        # X range where we expect the left lane to land
        win_x_left_min = left_x - margin
        win_x_left_max = left_x + margin
        # X range where we expect the right lane to land
        win_x_right_min = right_x - margin
        win_x_right_max = right_x + margin
        # Y range that we are analyzing
        win_y_top = img_height - idx_window * window_height
        win_y_bottom = win_y_top - window_height

        # Non zero pixels in x and y
        non_zero_left = ((non_zero_y >= win_y_bottom) & (non_zero_y <
            win_y_top) & (non_zero_x >= win_x_left_min) & (
                non_zero_x < win_x_left_max)).nonzero()[0]
        non_zero_right = ((non_zero_y >= win_y_bottom) & (non_zero_y <
            win_y_top) & (non_zero_x >= win_x_right_min) & (
                non_zero_x < win_x_right_max)).nonzero()[0]

        left_lane_indexes.append(non_zero_left)
        right_lane_indexes.append(non_zero_right)
        # If you found > min_pixels pixels, recentre next window on the
            mean position
        if len(non_zero_left) > min_pixels:
            left_x = np.int(np.mean(non_zero_x[non_zero_left]))

        if len(non_zero_right) > min_pixels:
            right_x = np.int(np.mean(non_zero_x[non_zero_right]))

    # Polynomial fitting
    valid, sw = fit_slide_window(binary_warped, hist, left_lane_indexes,
        right_lane_indexes, non_zero_x, non_zero_y)

    return valid, sw
```

The reasoning behind this code can be summarized as follows:

1. Choose only the coordinates of the pixels selected by the thresholding.
2. Initialize the current positions with the ones identified by the histogram.
3. Choose a margin (for example, half of the window width of the sliding window) and the minimum number of pixels to detect to accept a new position for the sliding window.
4. Update the coordinates of the sliding window by computing the new lateral and vertical coordinates (x and y).
5. Select only those pixels which are white and constrained in the window.

6. Update left and right position with the average of the positions only if there are enough points.

7. Obtain a line from the selected points using polynomial fitting.

5.3 Comparison

As previously mentioned, LD algorithm represents the basis of every autonomously driving car and, for this reason, it has been the first task the team had to accomplish during the competition.

The **Hough Transform method with Canny Edge Detector** is a very simple and straightforward solution and, with the implementation of a smooth function responsible for calculating the amount of steepness of a curve, it guarantees a very good performance [9]. In fact, apart from a couple of adjustments, especially regarding the confusion with the intersection line, it showed robustness also with an overall increase of the speed. As reported in [10], this method is not preferable in case of more realistic traffic situations such as confusing road textures and uneven illumination.

Another effective solution consists in the **Perspective correction and Histogram statistics method**. Despite the fact that it has a more complex coding structure, it is more utilised in real life contexts, since in the track-following application, multi-curve scenes are common and cross horizontal lines should be filtered out. Although the Hough Transform method can filter the horizontal line by post-processing (slope intercept correction), it remains sensitive to ambient noise and may lead to a large number of abnormal lines.

It is clear that, for the competition purpose, both methods show similar efficiency and, for both, the output is depicted in Figure 19, where the green line is the heading line responsible for correctly centering the car on the road and the blue lines are the recognized lane lines, from which the curvature value (and, consequently, the steering angle) is calculated.

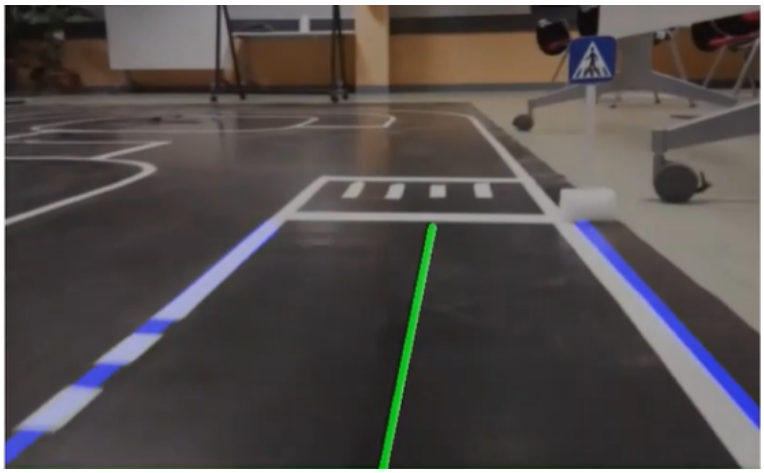

Figure 19: Output of the LD algorithms.

5.4 Intersection Detection

Due to the complex and dynamic character of intersection scenarios, the autonomous driving strategy at intersections has been a difficult problem and a hot point in the research of intelligent transportation systems in recent years [11]. In the BFMC context, there were mainly 3 different types of intersections: **crossroad**, **T-intersection** and **roundabout**. The navigation throughout these intersections has been defined in the `MovCarProcess.py` and it consists in turning left, right or going straight on manoeuvres according to the current intersection. More specifically, for the Mid-Term Quality Gate, the team implemented the logic of the state machine model, which divides the states of the vehicle into a limited number of categories. It is obvious that this classical method is only suitable for simple scenarios rather than complex dynamic ones where the artificially defined rules cannot adapt to all situations. In fact, during the semi-finals and finals at the Bosch Engineering Centre, the team adopted a state machine-based algorithm which was running also according to the GPS data.

As for LD, the objective of Intersection Detection in the field of Image Processing regards finding the horizontal line which precedes the intersection on the track. This step was crucial for the correct functioning of the autonomous drive for this project, since only the accurate detection of the intersection could assure also the exact behaviour and decision-making of the car on the track.

Since the logic of individuating the horizontal line is the same as individuating the lane lines (with the only change in slope), the team implemented almost the same Hough Transform code for the LD with some crucial differences: in particular, during the testing phases, the addition of the detected horizontal line often misconfused the LF part, since the car was detecting the horizontal line as if it were one of the 2 lane lines and wrongly tried to follow it. The correction of this error is taken into account in the function `average_slope_intercept` inside the *MainLaneDetectionProcess*, similarly to what is done in the `get_steering_angle` function: the former performs the correction for calculating the correct slope of the lines in order to display the detected lines, the latter does the same but for calculating the true steering angle. All in all, it is a sort of double check that the detected line is effectively a lane line or an intersection and act accordingly. The former function is highlighted in the code below:

```
def average_slope_intercept(self, frame, line_segments):
    ....
    previous = 0 #previously found line
    subsequent = 0 #next line

    for line_segment in line_segments:
        for x1, y1, x2, y2 in line_segment:
            if x1 == x2:
                continue

            fit = np.polyfit((x1, x2), (y1, y2), 1)
            slope = (y2 - y1) / (x2 - x1)
            intercept = y1 - (slope * x1)

            subsequent = previous
            previous = slope
```

```
            #When the two lines are perpendicular
            if subsequent*previous<-90:
                if abs(previous)<abs(subsequent):
                    slope=previous
                else:
                    slope=subsequent
    ....
        return lane_lines
```

To this purpose, the control implemented regards the definition of 2 new variables: *previous* and *subsequent*. They are used to "remember" the slope of the line which was detected before the horizontal one, which should be in fact one of the 2 lane lines. In fact, the *subsequent* variable is set as the *previous* one, which is then set to the calculated slope.

The control on the slope is based on the perpendicularity between the lane line (which is usually almost vertical) and the horizontal line: if the product between the 2 lines is greater (in absolute value) than 90°, two cases may happen.

1. The slope of the previous line is less than the slope of the next line, meaning the next line is in fact a horizontal line and so the slope is kept as the one of the previously detected line.

2. The slope of the previous line is greater or equal to the slope of the next line, and so the slope corresponds to the next line, meaning the detected one is effectively a lane line.

Another important aspect to consider is the ROI, which is moved slightly upward if compared to the LD one, since the intersection must be detected in advance. It is crucial to remind that the definition of the ROIs strongly depends on both the inclination and the height of the camera (Figure 20).

Figure 20: Inclination of the Raspberry Pi Camera.

Finally, the output of the *IntersectionDetectionProcess* is a variable called `self.Intersection` which is equal to 1 whenever the intersection is detected, 0 otherwise.

6 Traffic Sign Recognition

Traffic Sign Detection (TSD) and Classification (TSC) are among the most challenging tasks of autonomous vehicles: as for LD, it is based on the usage of vehicle cameras to capture real-time road images and then on detecting and identifying the traffic signs encountered on the road, thus providing accurate information to the driving system.

However, the road conditions in the actual environment are very complicated. Nowadays, there exists a wide variety of computer vision methods to detect traffic signage but further research and improvement are still needed.

In the context of traffic sign recognition, there are two tasks to accomplish: finding the locations and sizes of traffic signs in natural images (TSD) and classifying the detected traffic signs into their specific sub-classes (TSC). Traffic signs are designed in such a way to attract human drivers' attention but, for computer algorithms, there are many difficulties due to illumination changes, color deterioration, motion blur, partial occlusion, etc. These problems are more easily dealt by deep learning techniques [12] because, unlike traditional machine learning methods, they provide neural networks which can be trained automatically and extract image features increasing significantly the accuracy of detection. The main drawback of such methods is the heavy computational effort, in most cases not suitable for low-cost hardware packages such as Raspberry Pi.

In this work, for the sake of completeness, two main techniques have been implemented and, finally, compared:

- Pre-trained **linear Support Vector Machine** (SVM) model with Histogram of Oriented Gradients (HOG) features descriptor.
- Deep learning model using **Tensorflow**.

Both methods rely upon common characteristics of machine learning algorithms, which are shown in Figure 21.

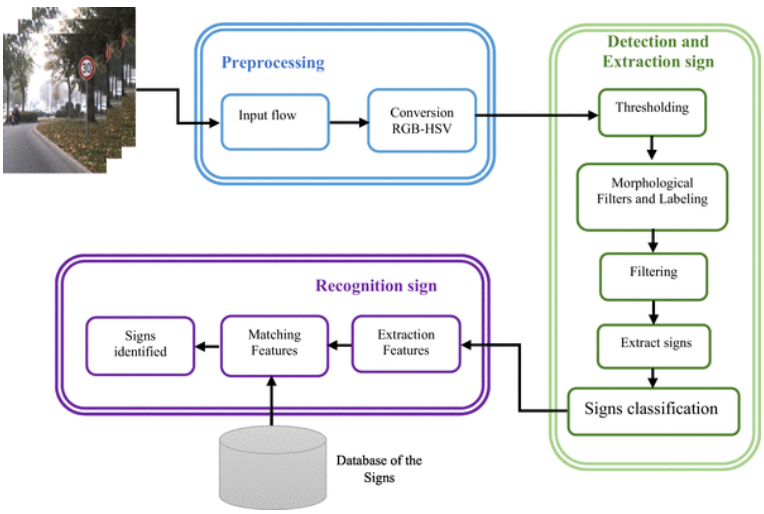

Figure 21: Flow Chart for Traffic Sign Detection.

To summarize the functioning of such algorithms, they both count on a **dataset** containing many complex images of the traffic signs to be recognized, such as sign

tilt, uneven lightning, traffic sign distortion, occlusion and similar background colors so that, ideally, the classification model is capable of recognizing the objects in every situation. For both models, the dataset has been created using images of the traffic signs in question, shown in Figure 22, taken both from the Raspberry Pi Camera and other devices.

Figure 22: Traffic Signs to recognize from the BFMC22.

Despite their differences, which will be explained better in the next subchapters, both the linear SVM and the neural network are trained by means of the dataset and, according to this training, should be capable of detecting the traffic signs coming from the car camera. It is quite evident that, the more images are contained in the dataset, the better the performance of the models will be: in fact, the dataset created in this work is composed by a total of approximately 2000 images.

6.1 Linear SVM classifier

Before diving into the code structure, it is important to understand what a linear SVM is and how it works: the algorithm creates a line or a hyperplane which separates the data into classes. Practically, SVM takes the data as an input and outputs a line that separates those classes if possible but, since there exist infinite lines that can separate the classes, SVM finds the one which is the farthest from the points of each class in order to decrease the misclassified points, as it is displayed in Figure 23. The optimization problem which this algorithm solves is the maximization of the so-called **margin**, the distance between the line and the support vectors.

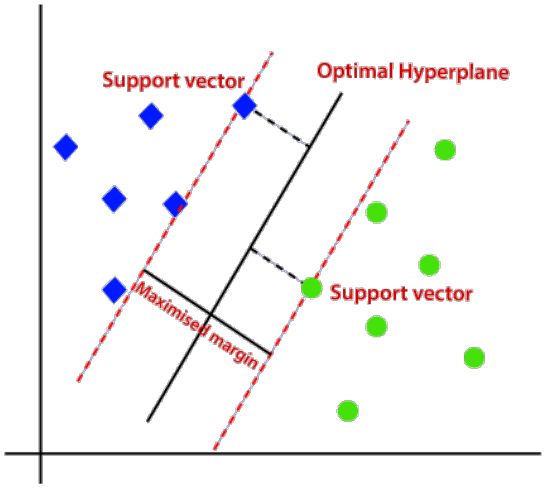

Figure 23: Linear SVM optimal hyperplane.

The algorithm is divided into two main phases: **detection** and **classification**. The detection phase uses image processing techniques that create contours on each video frame and find ellipses or circles among those contours: they are marked as candidates for traffic signs. The detection strategy involves the following steps:

1. Increase the contrast and dynamic range of the video frame.
2. Remove unnecessary colors with HSV Color Range.
3. Use Laplacian of Gaussian to display border of the objects.
4. Make contours using Binarization.
5. Detect ellipse-like and circle-like contours.

In the classification phase, instead, a list of images is created by cropping the original frame based on candidates' coordinate. The pre-trained SVM model will classify these images to find out which type of traffic sign they correspond to. If a traffic sign is detected, it will be tracked until it disappears or there is another bigger sign in the frame.

The **dataset** contains 13 folders, each corresponding to a specific object to detect: in addition to the traffic signs shown in Figure 22, there is a "OTHER" folder for anything that is not classified as one of the classes and other 2 folders for "PEDESTRIAN" and "SEMAPHORE", even though these last 2 objects are better classified using other object detection methods (Chapter 8).

The algorithm is mainly composed by 3 files: `SignDetectionProcess.py`, `classification.py` and `common.py`. Apart from the classical Process present in every algorithm, the `common.py` file contains functions used in the `classification.py` file, whereas the `classification.py` file is recalled by the Process. More specifically, the `classification.py` file is responsible for training and saving the SVM model, so it performs the following tasks:

- Loads the traffic sign dataset from the correct directory.
- Finds and calculate the HoG parameters using the `cv2.HOGDescriptor` function

- Trains and saves the model after splitting the dataset into training (90%) and validation (10%).

Inside the Process, the main steps are reported in the code below.

```
#ROI definition
frame = frame[0:300, 320:640]

#Finds the sign in the frame, returning the coordinates, the original
    image, the type of sign (with the corresponding text in the image)
coordinate, image, sign_type, text = self.localization(frame,
    args.min_size_components, args.similitary_contour_with_circle, model,
    count,current_sign)

#Draw a rectangle based on the coordinates
if coordinate is not None:
    cv2.rectangle(image, coordinate[0], coordinate[1], (255, 255, 255), 1)

if NEW SIGN FOUND:
    current_sign = sign_type #update of the current sign

    top = int(coordinate[0][1] * 1.05)
    left = int(coordinate[0][0] * 1.05)
    bottom = int(coordinate[1][1] * 0.95)
    right = int(coordinate[1][0] * 0.95)

    #New size
    tl = [left, top]
    br = [right, bottom]

    #Grab the ROI for the bounding box and convert it to the HSV color space
    roi = frame[tl[1]:br[1], tl[0]:br[0]]
    roi2 = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

    #Compute a HSV histogram for the ROI and store the bounding box
    roiHist = cv2.calcHist([roi2], [0], None, [16], [0, 180])
    roiHist = cv2.normalize(roiHist, roiHist, 0, 255, cv2.NORM_MINMAX)
    roiBox = (tl[0], tl[1], br[0], br[1])

elif SAME SIGN:
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    backProj = cv2.calcBackProject([hsv], [0], roiHist, [0, 180], 1)

    #Apply cam shift to the back projection, convert the points to a
        bounding box, and then draw them
    (r, roiBox) = cv2.CamShift(backProj, roiBox, termination)
    pts = np.int0(cv2.boxPoints(r))
    s = pts.sum(axis=1)
    tl = pts[np.argmin(s)]
    br = pts[np.argmax(s)]
```

In this case, the ROI is defined in the half right-hand side of the frame. Then, the main function `localization.py` is called: it is responsible for preprocessing and binarization of the image, finding the contours of the binary image and evaluating the correspondence between the contours and one of the traffic signs. Finally, the rest of the code writes on the original image both the rectangle and the text identifying

the sign according to which sign is detected: all the identifications are updated, if the detected sign is different from the current sign, then the *roi* and the *roiBOX* are calculated on the basis of the new coordinates and the HSV histograms, whereas if the detected sign is equal to the current sign, they are updated using only a CamShift method to track the shift of the sign in the frame. The output of this code is shown in Figure 24.

Figure 24: Linear SVM TSD output.

During the experiments on the real track, the team found out that for some signs such as Parking, Crosswalk and Stop the detection was really good, whereas for others signs like Highway and Priority, the detection was worse. In order to improve the performance of the algorithm, the team tried to insert as many images as possible in the dataset. Still, this type of algorithm has some disadvantages:

- Static image processing since the parameters must be updated for each video with different lightning conditions.
- Low accuracy of detection due to miss signs or wrong areas detection.
- Retraining of the model when running the program.

A better accuracy is obtained for the Neural Network explained in the next subchapter.

6.2 Tensorflow-based Neural Network

Neural networks and **deep learning** currently provide the best solutions to many problems in image recognition. Unlike common classifiers, deep learning techniques manipulate the training data. In fact, what really matters is the data fed to the algorithm rather than the algorithm itself: this is totally different from normal programming, where different tasks usually require different code, even though difficult tasks require more advanced neural networks to perform well. Deep learning can be considered a subset of machine learning where the computation is performed by several computation layers and, from a practical point of view, deep learning is achieved using neural networks, represented in Figure 25.

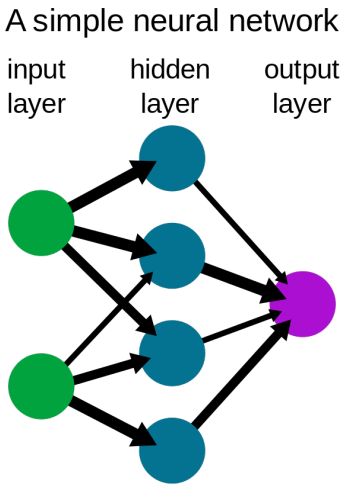

Figure 25: Simple Neural Network.

A neuron is a computation node that produces an output given some input: this computation can be divided into 2 parts.

- The transfer function computes the sum of every input multiplied by its weight (a number): this is a linear operation defining the dependence of the current neuron with its input neurons.
- An **activation function** is applied to the result of the previous operation and it should be a non-linear operation. A common one is the Rectified Linear Unit (ReLU). Introducing a non-linear operation in the activation allows a network to compute non-linear functions that become more and more complex as the number of layers grows.

During the training phase, the parameters of the neuron (bias and weights) are tuned: the whole purpose of training is to find the best possible value for these parameters for the task in question. This implies that the same neural network with different parameters can solve different problems. Compared to the past, deep learning techniques have been improved thanks to some critical advances:

- Many datasets are available on the internet to train the neural network.
- Architectures have become better and more efficient.
- There are several good open source libraries for neural networks.

Regarding the last point, the implemented neural network is in fact based on **Tensorflow**. TensorFlow is an open source library for fast numerical computing created by Google and it was designed for both research and development in production systems. It can run on single CPU systems, GPUs as well as mobile devices. The main code is shown below.

```
images, label = self.load_traffic_dataset()

# Create a graph to hold the model:
graph = tf.Graph()

# Create model in the graph:
with graph.as_default():
    # Placeholders for inputs and labels.
    images_ph = tf.placeholder(tf.float32, [None, 32, 32])
    labels_ph = tf.placeholder(tf.int32, [None])

    flatten_layer = tf.keras.layers.Flatten() # instantiate the layer
    images_flat = flatten_layer(images_ph) # call it on the given tensor

    # Fully connected layer.
    logits = tf.compat.v1.layers.dense(images_flat, 62, self.lrelu)
        #tf.nn.relu for model of type 1 (only 70% accuracy)

    # Convert logits to label indexes (int).
    predicted_labels = tf.argmax(logits, 1)

    # Define the loss function.
    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits
    (logits=logits, labels=labels_ph))

    # Create training op.
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
    train = optimizer.minimize(loss)

    # And, finally, an initialization op to execute before training.
    init = tf.global_variables_initializer()

# Create a session to run the graph we created:
session = tf.Session(graph=graph)

# First step is always to initialize all variables.
_ = session.run([init])

# Run the "predicted_labels" op.
sign_type = session.run([predicted_labels], feed_dict={images_ph:
    frame})[0]
```

First of all, the traffic sign dataset is loaded, in particular each read image is resized to 32x32 pixels. Then, a graph from the Tensorflow is created to hold the neural network so that the learning will be done on the graph and not on the code: the graph will be run on a low-level code (GPU). Inside the model, 2 placeholders are created, one for images and one for labels: a **placeholder** is simply a variable to which data will be assigned later in the code. A flatten layer is instantiated and associated to the images placeholder and then the functional interface of the layer

is defined: in addition to the layer itself, `tf.compat.v1.layers.dense` takes as input the dimensionality of the output space (62) and the activation function. The latter, in particular, has been chosen to be both the classical **ReLU** function and the **LeakyReLU** (`lrelu`).

A ReLU function has output 0 if the input is less than 0 and raw output otherwise; it has the advantage of not having any backpropagation errors and the speed of building models based on ReLU is very fast opposed to other solutions such as sigmoids. The main drawback of this function is that is non differentiable at 0, and this is why the accuracy of the model will be only around 70%. To avoid this problem, the Leaky RelU is used instead:

```
# TensorFlow doesn't have a native implementation of Leaky-ReLu
def lrelu(x):
    return tf.maximum(0.01 * x, x) # giving a bit of gradient also for
        negative values
```

The graph for this function is shown in Figure 26.

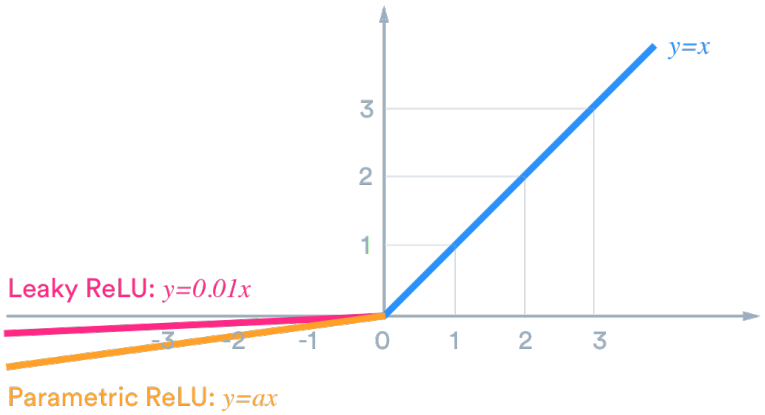

Figure 26: Leaky ReLU and Parametric ReLU.

As the name and the figure suggest, the slope is changed left of x=0 causing a leak and extending the range of the ReLU: instead of not firing at all for large gradients, the neurons do output some values and that makes the layer much more optimized too. In fact, the accuracy is increased up to 90%.

The `tf.compat.v1.layers.dense` outputs the logits of the fully connected layer, which stands for the predicted labels (then converted into 1D vector). In order to train the neural network, an appropriate optimization process must be used that requires a loss function to calculate the model error. Cross-entropy and mean squared error are the two main types of loss functions to use when training neural network models and this operation is done in the `tf.reduce_mean` function. The optimizer consists in the **Adam Algorithm**, an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. The only parameter to be defined is the learning rate which is set as default value.

Finally, the model is run taking as input the logits of the neural network and the camera frame as the images placeholder. According to the output of the `session.run`, the detected traffic sign is derived.

6.3 Results comparison

Both the algorithms were run together with all the other Processes involved in the autonomous driving project in order to test both the accuracy of the detection and the speed of computation. From a first implementation of the Tensorflow-based Neural Network, the detected traffic sign printed in the terminal of the Raspberry Pi corresponded to the actual traffic sign present in the track with an accuracy of 90%, the only problem regarded the significantly reduced speed of computation, meaning that the detected traffic sign was printed with a consistent delay.

On the other hand, the linear SVM model did not lower the speed of computation but demonstrated a much lower accuracy, specifically it performed several miss signs in presence of determinate traffic signs such as priority and highway. The increase of the dataset did not show particular improvements.

It is clear that, both from state-of-the-art research and from the results obtained during the testing phase on track, the optimal solution for traffic sign detection is represented by the Neural Network. Nevertheless, in order to be effectively utilized, the hardware must be upgraded and this can be done inserting a Neural Stick in the Raspberry Pi or replacing the Raspberry with a more powerful on-board computer.

For the purpose of the BFMC project, the team opted for keeping the linear SVM model to detect traffic signs, otherwise the usage of such a heavy neural network-based algorithm would have lowered the performance of the other algorithms.

7 Traffic Light Detection

Traffic signal light is one of the most important information guidance of traffic signals: accurate identification of traffic signal light in advance is conducive to the pre-planning of the path of intelligent networked vehicles. Therefore, the study of traffic signal light identification technology has important significance and good application prospects.

The existing traffic signal lamp detection methods mainly include color feature-based and shape feature-based detection methods. For the identification of traffic lights, the functions have to be:

- Image acquisition and preprocessing of the collected images.
- Canny operator for Edge Detection.
- Method for extracting pixel points to identify the color of traffic lights.

Other more optimized algorithms consist again in training and classification phases typical of a neural network.

For the BFMC purpose, one of the basic requirements for the correct functioning of the autonomous driving regarded the detection of the traffic light. This mainly meant detecting the object itself rather than the color of the semaphore, since the latter was given directly by the Bosch server and communicated to the car by means of a dedicated process, which is part of the V2X communication. To this aim, during the competition, all the cars' Raspberry Pi's are connected to the LAN and API's were given in the brain project for the interaction with all the systems (Chapter 2 and 3). In the traffic light case, each semaphore on the track broadcasts messages with a determined frequency including ID and STATE: ID corresponds to the identification number of the semaphore (from 1 to 4) and STATE is the turned-on color.

However, the team decided to try implementing an algorithm which was capable of detecting also the semaphore colors in order to submit the work done in the last Project Status before the on-site competition.

The semaphore used is shown in Figure 27.

Figure 27: Semaphore used in the BFMC22 (left) versus Real Traffic Lights (right).

As it is noticeable from Figure 27, this semaphore does not show a high contrast for all the colors: in presence of high light conditions, the lighten-up color is not so

distinguishable and this represents the main difficulty for the detection algorithm. All the examples found on the web are applicable to real traffic lights among which the majority of them does have high contrast between the 3 colors, since a driver must be immediately captured by the ON color and act consequently: a confusion may cause great damage to the driver and all the other vehicles involved in the crossroad.

The algorithm proposed in this chapter has the main purpose of acquiring knowledge and getting used to manipulating the image in a context in which the color itself, not the contours of a binary/grey image, matter, more than showing an immediately applicable solution to real autonomous driving problems. Nevertheless, it is a solid basis for developing a working algorithm to detect colors of real traffic lights in which the turned on color is highly visible [13].

7.1 Color Detection

It is clear that the objective of the algorithm is finding a way to detect 3 colors: red, yellow and green [14]. This can be done by using **Color Thresholding**, whose idea is to limit the image to where determinate colors are present. Anywhere in the image that does not correspond to one of these colors is set to zero and shown as black. To effectively isolate the colors of interest, there are a few considerations to cover: the color space, the threshold cutoff and variations to illumination. Typically images are represented in the RGB color space which mixes color and intensity information throughout its channels and this makes the RGB format sensitive to changes in light. This is the reason why it is preferable to convert the image into **HSV** or **HSL** color space. As previously mentioned, HSV stands for:

- HUE: modeled as an angular dimension that encodes color information.
- SATURATION: encodes the intensity of the color.
- VALUE (brightness): represents the amount at which that respective color is mixed with black.

HSV color space uses the Angle to measure the color: 0° represents red, counter-clockwise calculation. Saturation refers to the purity of the color, and the value of this channel is defined by percentages in the range [0,1]: a better representation is given in Figure 28.

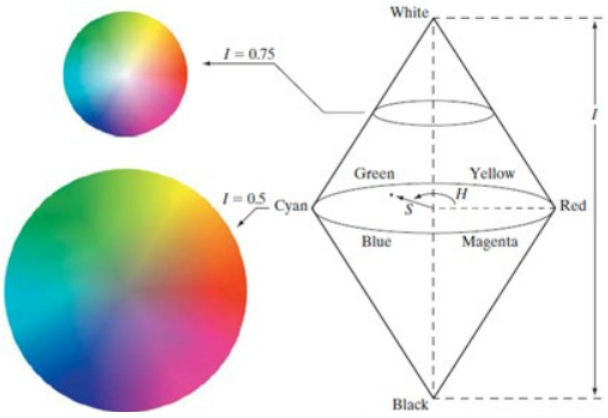

Figure 28: HSV color space model.

Compared to the RGB color space, HSV is not suitable for the display system but more in line with the visual characteristics of human eyes. An example of this conversion is given in Figure 29.

Figure 29: Comparison between RGB and HSV color space.

In the specific case of the traffic light detection algorithm, the utilized function is `find_hsv`, which takes as input the RGB image, computes the HSV correspondent image and finds an average both for brightness and saturation, since these are the main parameters to account for when distinguishing a color from the others.

```
def find_hsv(self,rgb_image):
    img_float = np.float32(rgb_image)
    hsv = cv2.cvtColor(img_float, cv2.COLOR_RGB2HSV)
    h = hsv[:, :, 0]
    s = hsv[:, :, 1]
    v = hsv[:, :, 2]
    sum_brightness = np.sum(hsv[:, :, 2])
    area = 32 * 32
    avg_brightness = sum_brightness / area # Find the average
    sum_saturation = np.sum(hsv[:, :, 1])
    avg_saturation = sum_saturation / area # Find the average
    return avg_brightness, avg_saturation
```

This function is applied to the whole traffic light dataset, which contains both ON and OFF conditions for every color. According to the averaged values obtained from the function output, the range of approximated values for brightness and saturation is shown in Table 2. Also from these results, it is possible to confirm the very slight

CHARACTERISTIC	CONDITION	GREEN	RED	YELLOW
SATURATION	ON	0.62	0.60	0.62
	OFF	0.58	0.52	0.56
BRIGHTNESS	ON	0.45	0.48	0.48
	OFF	0.29	0.33	0.40

Table 2: Brightness and saturation values for ON/OFF conditions of the traffic light.

difference between the intensity of the colors in the utilized semaphore, especially when the colors are turned ON and for which the faint light is weak against the light of the environment. These values are then used in a specific function to define the masks in which the 3 colors are contained.

7.2 Mask Definition

The mask definition has the main role of segmenting the colored region from the image. In the code below, only the part relative to the green color is reported.

```
def color_mask(self,rgb_image, brightness, saturation, hue):
    img_float = np.float32(rgb_image)
    hsv = cv2.cvtColor(img_float, cv2.COLOR_RGB2HSV)
    ...
    # lower bound and upper bound for Green color
    lower_bound = np.array([50, saturation_off, brightness_off])
    upper_bound = np.array([100, 255, 255])
    # find the colors within the boundaries
    mask = cv2.inRange(hsv, lower_bound, upper_bound)

    # Segment only the detected region
    segmented_img = cv2.bitwise_and(img, img, mask=mask)

    # Find contours from the mask
    contours, hierarchy = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
    output = cv2.drawContours(segmented_img, contours, -1, (0, 0, 255), 3)
    ...
    return output_red, output_yellow, output_green, red_mask, yellow_mask,
        green_mask
```

The lower and upper bounds are the boundaries of the color: the first element in the array is the HUE value and it is set according to the HSV palette color space, whereas SATURATION and BRIGHTNESS are set according to the values obtained from the `find_hsv` function. Afterwards, the `inRange()` function returns a binary mask of the frame where the green color is present: wherever the green is detected, the mask shows that as white and the rest of the region is black. By means of the `cv2.bitwise_and()` function, the mask is applied on the frame in only the region where the mask is true. Finally, the boundaries over the detected regions are drawn.

The output of this function is shown in Figure 30, where the images for green, red and yellow masks have been inserted side-by-side.

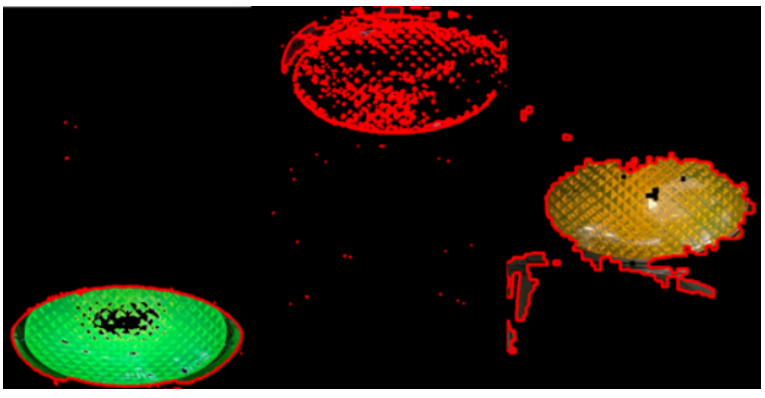

Figure 30: Color Mask Detection.

It is clear that in this function what plays a crucial role is the lower and upper bound arrays: according to the specific situation (the colors intensity), these values

have to be tuned to increase the accuracy of the color detection and this operation requires many trials. In Figure 30, where the image to process is a semaphore with green ON, the green mask is almost perfectly defined, whereas the red and yellow one are more confused, especially for the red color since it is the darkest one among the others.

7.3 Pixel intensities using Histograms

As already studied in Chapter 5, histogram is considered as a graph or plot which is related to frequency pixels in an image. If applied to grayscale images, the shades of gray determine a variation from black at the weakest intensity (pixel value equal to 0) to white (pixel value equal to 255) at the strongest. The histogram plot reports the number of pixels present for a certain value of pixels: this means that left region of histogram shows the amount of darker pixels in the image, whereas right region shows the amount of brighter pixels.

A similar logic can also be applied to a RGB image.

```
def histogram_values(self,image,mask,color):
    histg = cv2.calcHist([image],[color],mask,[256],[0,256])
    plt.plot(histg)
    plt.show()
    avg=sum(histg)/len(histg)
    return avg, histg
```

To this purpose, the function `cv2.calcHist` is used and the inputs are:

1. The **source image**.
2. The **index of the channel** for which the histogram must be calculated: for a color image, the index is [0], [1] or [2] to calculate histogram of blue, green or red channel respectively; in this project, same channel of [1] has been selected for both green and yellow due to the similarity of tonality.
3. The **mask image**, None if no mask is given.
4. **Histogram size**: for full scale, it is [256].
5. **Range of colors**: [0,256].

In order to evaluate the pixel intensities in each case of the ON semaphore conditions, it is of interest to calculate the average of the histogram values: this value will then be used to detect the ON color.

Since the histogram has to be calculated for every color, the function is applied 3 times and different inputs for `color` and `mask` are given. An example of plotted histograms is given in Figure 31, where the histogram function has been applied to an image in which the semaphore is green.

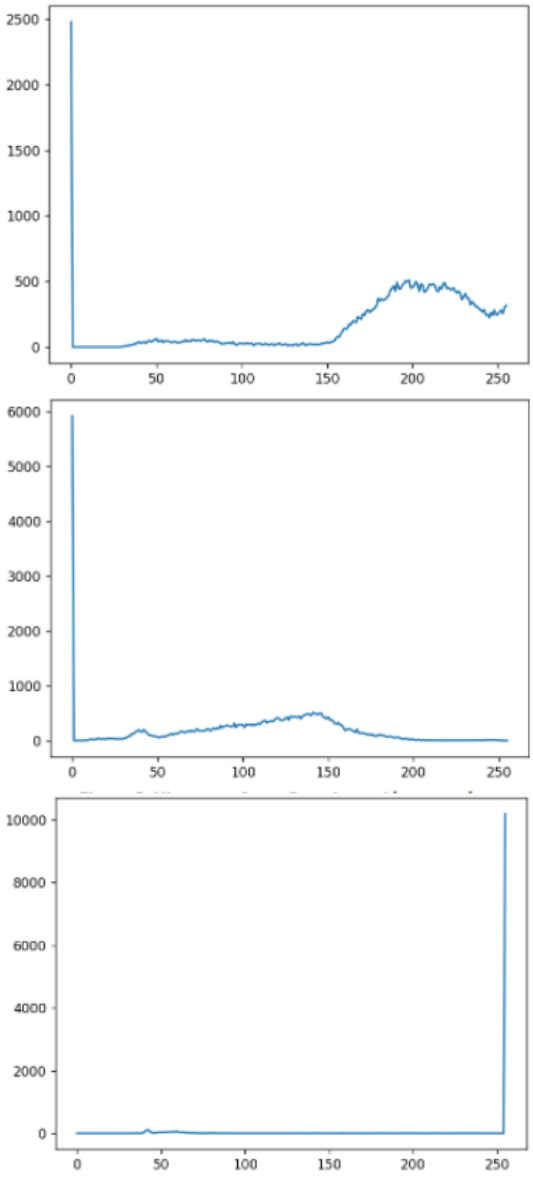

Figure 31: Histogram plots of green, yellow and red channels respectively when green is in ON condition.

Both from a first glance to the graphs and from the numerical calculation of the average value of the pixel intensities in the region of interest (in this case around 150-250 in the x-axis), it is possible to observe that the highest value is reached in the green channel, as expected. The peaks around 0 and 256 are due to black and white regions in the image, not to be considered for this analysis.

7.4 Algorithm implementation

The summary of the procedure involved in the Traffic Light detection developed for the project is:

1. Load and resize the image.

2. Calculate the HSV of the dataset to find the correct mask for each color.

3. Find the mask and select the area of the specific color.

4. Plot the histogram of the image relative to each mask and calculate the average, to be compared with the other colors to detect the correct one.

The first step is done only in the pre-elaborating part, in which the focus is on calculating the lower and upper bound arrays for each color: once these values are obtained, they are inserted manually in the code and given as input to the function. An extract of the `_TrafficLightsMain` function inside the *TrafficLightProcess* is given in the code below.

```
#Vectors of a-priori values (using the find_hsv function) of brightness,
    saturation and hue (red,yellow,green)
brightness_on = [0.48,0.30,0.45]
saturation_on = [0.62,0.40,0.58]
hue_on = [0,0.6,12]

#ROI selection
frame = frame[100:480, 320:680]

#Color mask and histogram calculation
red_out, yellow_out, green_out, mask_red, mask_yellow, mask_green =
    self.color_mask(frame, brightness_on, saturation_on,hue_on)
avg_red,histg_red = self.histogram_values(red_out,mask_red,2)
avg_green,histg_green = self.histogram_values(green_out,mask_green,1)
avg_yellow,histg_yellow = self.histogram_values(yellow_out,mask_yellow,1)

#Color detection based on pixels average
if not(avg_red == 0 and avg_green == 0 and avg_yellow == 0):

    if avg_green >= 30:
        estimated_color = avg_green
        print("GREEN DETECTED" +str(estimated_color))
    elif avg_yellow >= 17:
        estimated_color = avg_yellow
        print("YELLOW DETECTED" +str(estimated_color))
    elif avg_yellow < 17 and avg_green <30 and avg_red<=12 and avg_red>=11:
         estimated_color = avg_red
         print("RED DETECTED:" +str(estimated_color))

for out in outPs:
    out.send(estimated_color)
```

The reasoning behind the color detection is choosing some threshold values of pixel intensities for each color when it is in ON condition: these values are found after many trials when running the code on the Raspberry Pi, however they are not unique, but they strongly depend on the light condition, e.g. the time of the day in which the experiment is carried out.

Obviously, it is possible to implement an auto-exclusion logic since only one color is in the ON condition each time; in the code reported above, it was not needed since the histogram values for each color covered very different ranges from one another. Anyway, the values in the if conditions only have an exemplary function and were true exclusively for a specific light and environment condition.

This is in fact the main drawback of this algorithm: it is not possible to find a fixed histogram value to detect the color in every situation. It is clear that a

better implementation can be found again in the neural network, but it has the disadvantage of the too heavy computational effort.

Anyway, if tested on a particular light condition together with the corresponding histogram values, the detection works well for green and yellow, whereas the code shows difficulties in detecting the red mask, which means strong variation in the resulting thresholds; this is probably due to the poor intensity of the color in the utilized semaphore.

8 Object Detection

Object detection mainly deals with identification of real-world objects such as people, animals, and objects of suspense or threatening [15]. Object detection algorithms use a wide range of image processing applications for extracting the object's desired portion. A basic block diagram of object detection process is shown in Figure 32.

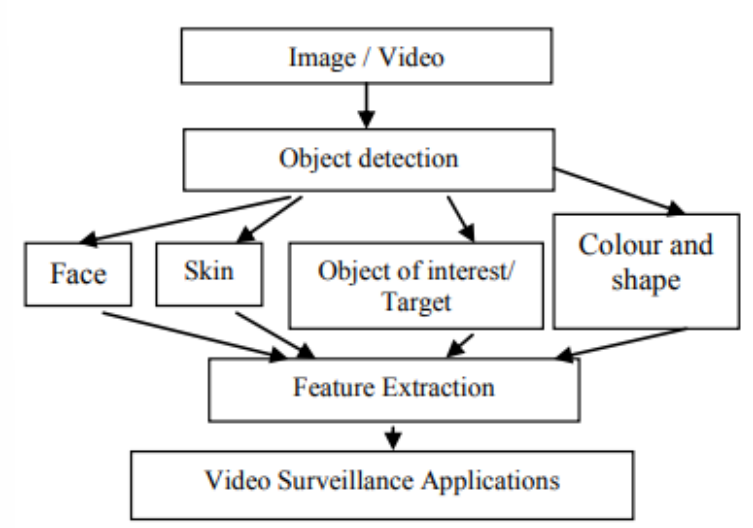

Figure 32: Basic block diagram of object detection process.

From Figure 32, frames are extracted from image or video, then objects are detected based on user's desired choice such as face, skin, colour and shape. Further various features of object detection are extracted for video surveillance applications, but for this work purpose it will be applied to the autonomous driving context, in which it is fundamental to detect other objects present on the road such as pedestrians, semaphores and cars. Specifically, the requirements for object detection (V2X) requested by the BFMC regard detection of a pedestrian crossing the street both in presence of the crosswalk sign and in a random point of the track, detection of the traffic light and detection of both static and moving cars.

In the case of intelligent video surveillance, both in a private context and in the highway for detecting the speed of the vehicles, the camera which extracts the object features is fixed and so these applications rely on differencing on every pair of consecutive frames to detect the objects which are moving: in particular, a specific area of the frame is selected as the one to compare to the previous frame and, in the case of the highway surveillance, only the cars are present and detected. It is clear that such a method cannot be exploited for an application in which the camera itself is always moving, and consequently also the stationary objects are seen in motion.

Due to the limitation imposed by the implementation of the neural network, again the team had to look for other applicable and similarly efficient solutions. The pedestrian detection exploits an algorithm based on the Histogram of Oriented Gradients (HOG) introduced by OpenCV, whereas semaphore and car detection utilizes a function which detects the shape of the object based on the number of short segments that can describe the shape itself. All these functions are implemented inside the same *ObjectDetectionProcess*, whose main is reported in the code below.

```
def object_classification(self,inPs,outPs):
```

```
fps = 10
frame_sem = []
frame_car = []
frame_ped = []
approxs = 0
# Define the codec and create VideoWriter object
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi', fourcc, fps, (640, 480))
try:
    while True:
        stamps, img = inPs.recv() #take images from Camera Thread
        frame = cv2.resize(img, (640, 480))
        #Define 3 different ROIs
        frame_sem = frame[0:200, 350:500] #semaphore ROI
        frame_car = frame[180:300, 200:640] #car ROI
        frame_ped = frame[0:340, 0:420] #pedestrian ROI

        self.ped, imagep = self.pedestrian_detector(frame_ped)
        self.sem, _, images, approxs = self.shape_detector(frame_sem)
            #detection of semaphore and car
        _, self.car, imagec, approxc = self.shape_detector(frame_car)

        #Whole control (they cannot happen at the same time)
        if self.ped >= 1 and not(2<=self.sem<=3) and not(1<=self.car<=2):
            self.count_ped = self.count_ped + 1
        if self.count_ped >= 2:
            self.object = 1
            cv2.putText(frame, "Pedestrian", (50, 50),
                cv2.FONT_HERSHEY_COMPLEX, 1, (255,0,0), 2)
            self.count_ped = 0
        else:
            self.object = 0
        ...
        outPs[0].send(self.object) #send which object: 1 for pedestrian,
            2 for semaphore and 3 for car
```

Similarly to the other image processing algorithms, 3 different ROIs are specified according to the object to detect and in which position it is supposed to be found on the track in order to decrease the false detections due to noise and other objects present in the environment. Another implemented control to achieve good detection results consists in utilizing counters: if the object to detect is the correct one, it is present in the frames continuously for a fixed period of time which is usually longer than the time in which other unwanted objects are subject to the detection algorithm. The values of the counters (one for each object) are set thanks to different experiments on the track.

Moreover, *MovCarProcess* performs a double check between the inputs coming from the detection algorithm and what is sensed by the Ultrasonic and Lidar sensors. More specifically, the Lidar is used to detect frontal objects (such as cars and pedestrians), whereas the Ultrasonic is used to detect side objects such as other cars during parking and overtake manoeuvres.

8.1 Pedestrian Detection

Pedestrian detection using OpenCV can be accomplished by leveraging the **built-in HOG and Linear SVM detector** that OpenCV ships with, allowing to detect people in images [16]. The feature descriptor is a simplified representation of the image that contains only the most important information about the image to recognize its content. The main characteristics of this method are the following:

- The HOG descriptor focuses on the structure or the shape of an object: the difference from a simple edge feature relies on providing both gradient and orientation (magnitude and direction) of the edge (not only its gradient).
- These orientations are calculated in 'localized' portions: the complete image is broken down into smaller regions and, for each region, the gradients and orientation are calculated.
- Finally, the HOG generates a histogram for each of these regions separately.

All in all, the HOG feature descriptor counts the occurrences of gradient orientation in localized portions of an image. An example of the output of this method is given in Figure 33.

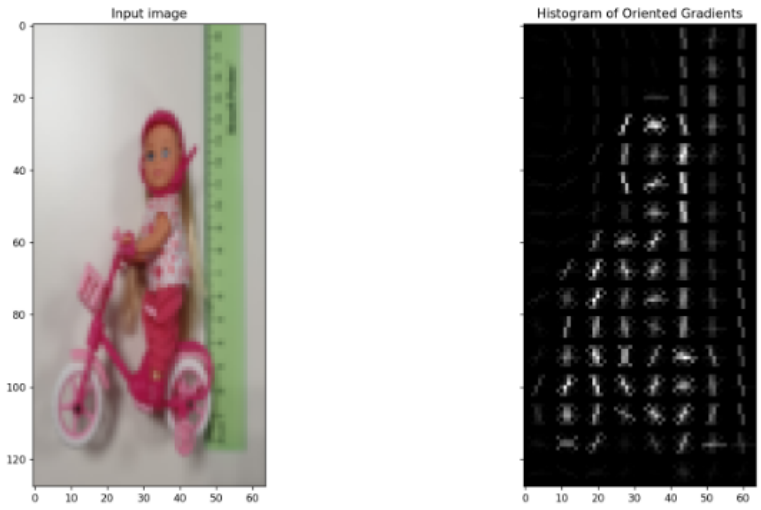

Figure 33: HOG descriptor applied to the pedestrian utilized in the BFMC22.

The HOG pedestrian detector in OpenCV is trained with a model that is 48x96 pixels and therefore it detects objects with a box of the same size.

```
def pedestrian_detector(self, image):
    #function to detect pedestrians only
    # construct the argument parse and parse the arguments
    ap = argparse.ArgumentParser()
    ap.add_argument("-w", "--win-stride", type=str, default="(8,
        8)",help="window stride")
    ap.add_argument("-p", "--padding", type=str, default="(16,
        16)",help="object padding")
    ap.add_argument("-s", "--scale", type=float, default=1.05,help="image
        pyramid scale")
```

```
ap.add_argument("-m", "--mean-shift", type=int,
    default=-1,help="whether or not mean shift grouping should be used")
args = vars(ap.parse_args())

# evaluate the command line arguments (using the eval function like
# this is not good form, but let's tolerate it for the example)
winStride = eval(args["win_stride"])
padding = eval(args["padding"])
meanShift = True if args["mean_shift"] > 0 else False
# initialize the HOG descriptor/person detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
# load the image and resize it
#image = cv2.imread(image)
image = imutils.resize(image, width=min(400, image.shape[1]))

# detect people in the image
start = datetime.datetime.now()
(rects, weights) = hog.detectMultiScale(image,
    winStride=winStride,padding=padding, scale=args["scale"],
    useMeanshiftGrouping=meanShift)
print("[INFO] detection took: {}s".format((datetime.datetime.now() -
    start).total_seconds()))
# draw the original bounding boxes
for (x, y, w, h) in rects:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
    cv2.putText(image, "Pedestrian", (x, y), cv2.FONT_HERSHEY_COMPLEX,
        1, (255,0,0), 2)
cv2.imshow("Pedestrian detection",image)
return len(rects) #return the number of rectangles (number of
    pedestrians detected)
```

The `argparse` module of Python makes it easy to write user-friendly command-line interfaces, and by means of the same, the parameters for the `detectMultiScale` function are defined. In fact, since the HOG descriptor for pedestrian is already built by default, the only parameters to modify are exactly those which are parsed, because they can increase the number of false-positive detections and dramatically affect the speed of the detection process. It is very important to find the trade-off between speed and accuracy, especially when this detector has to be run in real-time on resource constrained devices such as the Raspberry Pi. These parameters are:

- **Img** (required): can be either color or grayscale.
- **hitThresold** (optional): changes the max Euclidean distance between the input HOG features and the classifying plane of the SVM and a higher threshold means the classifier is confident with the result.
- **winStride** (optional): 2-tuple that dictates the "step-size" in both x and y location of the sliding window. In the context of object detection, a sliding window is a rectangular region of fixed width and height that "slides" across an image: at each stop of the sliding window, the HOG features are extracted and passed on to the Linear SVM. Since this procedure is expensive, it is preferable to evaluate as little windows as possible if the intention is to run out Python scripts in real-time, but also the smaller it gets, the more windows need to be

evaluated. Good custom is to start with (4,4) and increase the value until a reasonable trade-off between speed and detection accuracy is obtained.

- **Padding** (optional): tuple which indicates the number of pixels in both the x and y direction in which the sliding window ROI is "padded" prior to HOG feature extraction. Adding a bit of padding surrounding the image ROI prior to HOG feature extraction and classification can increase the accuracy of the detector. Typical values include multiples of (8,8).

- **Scale** (optional): it controls the factor in which the image is resized at each layer of the image pyramid, ultimately influencing the number of levels (the smaller it is, the more is the time taken to process the image). Typical values for scale are normally in the range [1.01, 1.5] and if it is run in real-time, this value should be as large as possible without significantly sacrificing detection accuracy.

- **finalThreshold** (optional): apply a "final threshold" to the potential hits weeding out potential false positives.

- **useMeanShiftGrouping** (optional): Boolean variable indicating whether or not it should handle potential overlapping bounding boxes, but to obtain better results it is more advisable to use non-maxima suppression, therefore this parameter has to be set to False.

Such parameters have to be tuned according to the application and the best suggestions include:

1. Resizing the image or frame to be as small as possible in order to reduce the width and height of the image in order to have less data to process meaning a faster detection.

2. Tune especially the **Scale** and **winStride** parameters because they have a tremendous impact on the object detector speed and performance.

The values shown in the code above have been tuned once in the competition track since the type of pedestrian to detect is unique, anyway this algorithm works for every kind of pedestrian, as shown in Figure 34.

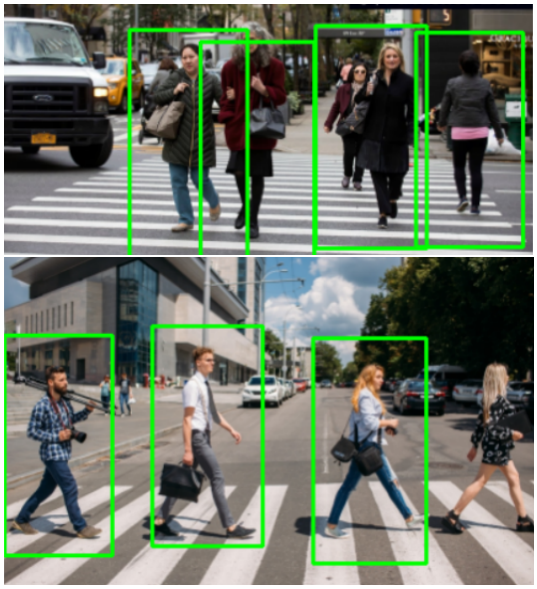

Figure 34: Pedestrian detector applied on random images from the web.

Instead, Figure 35 shows an image from the real competition track with the pedestrians to detect. The shown results are due to different values of the parameters to tune: `winStride, padding` and `Scale`. In particular, the values are reported in Table 3. The correct detection corresponds to the pink-dressed pedestrian on the bicycle, whereas the wrong one corresponds to a person who is standing in the background of the image. The values shown in the "Wrong Detection" column are the ones suggested in [17] and are useful to detect real pedestrians, but since in the BFMC the pedestrian to detect is fictitious, these parameters have been changed and written in the "Correct Detection" column. In particular, the following considerations can be made:

- winStride of (4,4) already represents a reasonable trade-off between speed and detection accuracy since the time elapsed is 0.08s (not considering the delays of running contemporary all the processes on the Raspberry).
- The scale factor of 1.02 has been chosen to speed up the detection time (1.01 is also correct but it is slower).

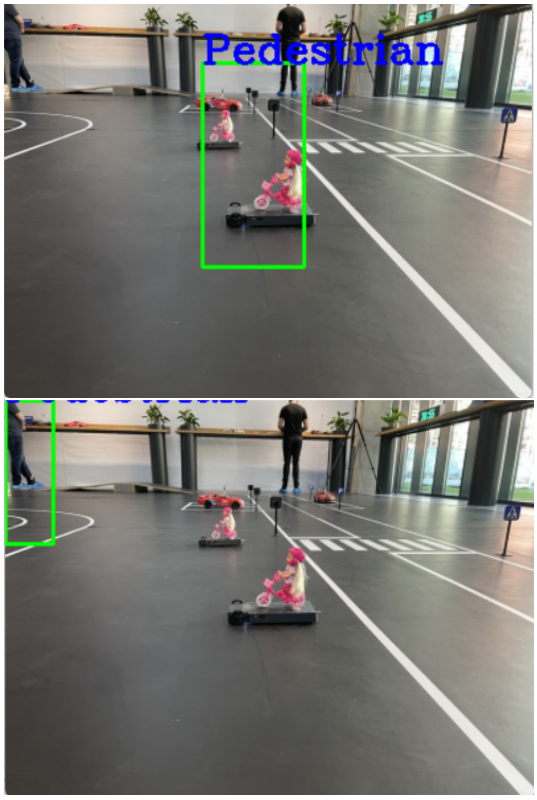

Figure 35: Pedestrian detector applied on an image of the competition track of the BFMC22.

Parameter	Correct Detection	Wrong Detection
winStride	(4,4)	(16,16)
Padding	(4,4)	(32,32)
Scale	1.02	1.05

Table 3: detectMultiScale parameters in case of correct and wrong pedestrian detection.

To sum up, this algorithm has good accuracy and it is quite simple to understand and develop, but the main drawback is the very low possibility of manipulation since it is an already built-in code. Still, for the project purpose, it is a good starting point.

8.2 Semaphore and Car Detection

A simple and low-computational effort method for detecting these objects is based on contour-based approximations of shapes: this process consists in lessening the number of vertices in such a way that the distance between the contours of shapes is equal to or less than the specified precision. A built-in function in OpenCV is used to approximate the shape of polygon curves to the specified precision: `approxPolyDP()`. It returns the approximated contour whose shape is the same as the input curve.

The idea of detection based on this method is that the semaphore is composed by 3 almost perfect circles, so in the code there is a variable which counts the number of

circles present in the ROI of the semaphore and if this value is between 2 and 3, then the semaphore is detected. The car is a little bit more complex to be approximated to basic shapes, in fact the detected circles are more elliptical than those present in the semaphore: for this reason, the shape which is representative of the car is an ellipse. The variables are then set to 0 once the object has been detected and to start again the count for the next object.

```
def shape_detector(self,image):
    #function to detect semaphores and static cars
    imagegray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(imagegray, (5, 5), 0)
    #using threshold() function to convert the grayscale image to binary
        image
    imagethreshold = cv2.threshold(blurred, 60, 255, cv2.THRESH_BINARY)[1]
    #finding the contours in the given image using findContours() function
    imagecontours, _ = cv2.findContours(imagethreshold, cv2.RETR_TREE,
        cv2.CHAIN_APPROX_SIMPLE)
    #for each of the contours detected, the shape of the contours is
        approximated using approxPolyDP() function and the contours are
        drawn in the image using drawContours() function
    for count in imagecontours:
        epsilon = 0.01 * cv2.arcLength(count, True)
        approximations = cv2.approxPolyDP(count, epsilon, True)
        cv2.drawContours(image, [approximations], 0, (0,255,0), 3)
    i, j = approximations[0][0]
    if 9 <= len(approximations) < 16:
        self.circle = self.circle + 1
        if self.circle == 3:
            self.circle = 0
    elif len(approximations) >= 16:
        self.ellipse = self.ellipse + 1
        if self.ellipse == 2:
            self.ellipse = 0
    else:
            pass
    return self.circle, self.ellipse, imagethreshold, len(approximations)
```

The image is re-elaborated using the usual grayscale and Gaussian blur technique to lower the weight of the data to process, it is then thresholded to individuate the contrast between the object and the environment, the contours of every object in the image are drawn and the approximate function is applied. This function takes as input the curve array (in this case, the contours), epsilon which is the parameter specifying the approximation accuracy (maximum distance between the original curve and its approximation) and a boolean variable that, if true, means that the approximated curve is closed. In order to guarantee a high approximation accuracy, an epsilon equal to 1% of the arc length has been chosen. The difference between an epsilon equal to 10% and another equal to 1% is shown in Figure 36.

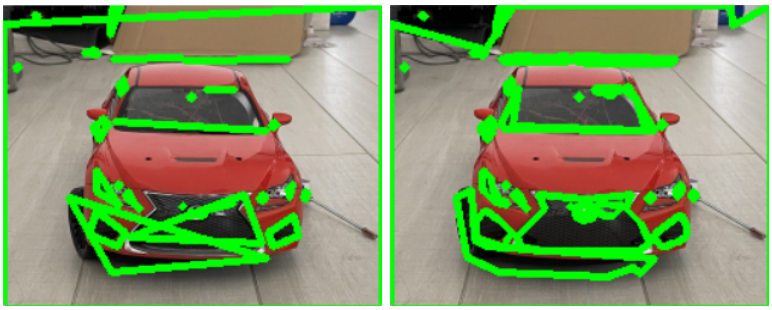

Figure 36: Approximation function with epsilon equal to 10% (left) and 1% (right) of the arc length of the curves.

The length of the approximations is useful to understand which shape it corresponds to and, given general criteria [17], it has to be set according to the particular shape of the object to detect. In fact, the type of circles detected in the semaphore are of length 11-12, whereas the ones detected on the car are between 15 and 19. For this reason, the length of the curves in the range 9-16 are chosen as circles, whereas those greater or equal to 16 are associated to ellipses.

The output of the process when an image of the semaphore is provided as input is shown in Figure 37.

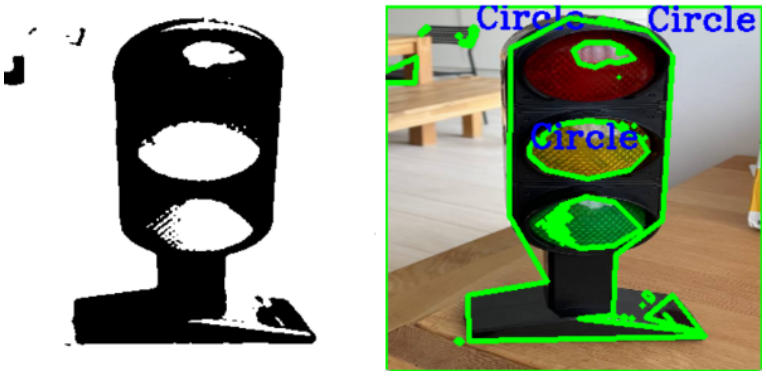

Figure 37: Approximation function applied to a semaphore: 3 circles detected.

The output of the process for an image of the car as input is illustrated in Figure 38.

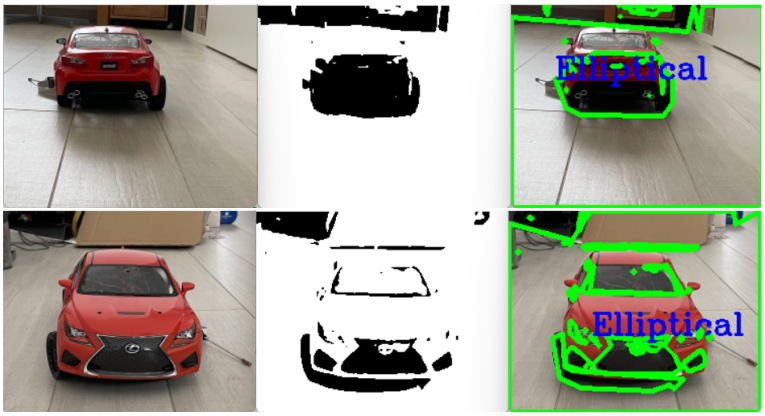

Figure 38: Approximation function applied to a car (front and rear views): 1 ellipse detected.

The simplicity of this algorithm is, on one hand, positive since it allows wide range of manipulation, on the other hand it is not so robust since it is often subject to false detections due to other objects present in the environment: the technique of using counters decreases this phenomenon but it does not completely cancel it. This is why the usage of neural networks would be more reliable for this task.

In any case, the fusion between these algorithms and the sensors represents a good choice: both cars and pedestrian are sensed by means of Lidar and Ultrasonic and can be distinguished thanks to the algorithm differentiation, whereas the semaphore can be detected using both the object detection and the traffic light detection (Chapter 7), so the circles of the semaphore can be associated to green, yellow and red colors. All the "data fusion" has to be performed inside the *MovCarProcess*.

9 Conclusions

The whole work was born from the idea of getting to know a nowadays challenge and reality that many engineers are facing all over the world: autonomous driving. The introduced benefits are countless: from a mere comfort of the driver to the security for everyone present in the road. To this aim, Bosch Romania has decided to challenge students internationally to join and participate in the quest for optimally functioning autonomous driving algorithms. The author, together with her colleagues, has been really excited to be part of this event and, although many difficulties arised, gained a once in a lifetime experience.

This is the reason why this thesis has been developed, to report such hard work, especially for those willing to continue the challenge in the next years. Although many of the explained algorithms are not immediately applicable to real autonomous driving cars, which have to perform traffic sign and object detection in the most efficient way, they can solve the problem of hardware insufficiency whenever the components are low-cost and not so computationally powerful.

LD & LF algorithms are proved to be effective, both using Hough Transform method and Perspective Correction and Histogram Statistics one, even if the latter is more useful in presence of road defects and more pronounced curves. They both belong to the feature based models category, in which local visual characteristics of interest are used to detect the road shapes. Other approaches consists in model-based and learning-based methods.

The Traffic Sign Recognition algorithm implemented for the BFMC22 is composed by a linear SVM which demonstrated very good capability in recognizing certain traffic signs and poor accuracy when recognizing other traffic signs. This gap can be compensated by increasing the dataset, capturing images of the traffic signs in every light and environment condition. In any case, the implementation of a simple neural network based on the Tensorflow Python library proved that, as a deep learning method, it has a higher accuracy (90% when using the Leaky ReLU activation function) but requires a more powerful computational board (for example, a graphic card) or a Neural Stick to connect to the Raspberry Pi, which offers the access to all the neural network functionalities since it incorporates computer vision and artificial intelligence services.

Traffic Light Detection using color masks is based on the idea of detecting different colors inside an image: the implemented algorithm has the aim of studying a way of using image processing to detect colors more than being a directly applicable solution to detect semaphores.

Finally, the Object Detection algorithms represent a more realistic and consistent solution to detect objects present in the road. Despite the fact of not relying on neural networks, the pedestrian detector based on the Python built-in HOG features descriptor has good accuracy, which can be improved just by changing few parameters inside the utilized function, whereas the semaphore and car detector relies on a shape approximation function and makes use of counters to decrease the number of false detections.

All in all, these Image Processing algorithms, especially Lane and Intersection detection, have demonstrated good performance during the competition and it has been also thanks to them and to the *MovCarProcess* structure that the team PoliTron successfully gained a position as finalist and winner in the BFMC22. Nevertheless, the author recognizes that further improvements can be made, principally regarding

Object Detection (both of traffic signs and objects in general), since for autonomous driving it is almost mandatory to use deep-learning techniques.

Moreover, a more robust control usually requires the fusion between the algorithm output and the sensors: this is why, for object detection, Ultrasonic and LiDAR sensors are used. The team opted for using an Ultrasonic sensor to detect side objects (parked cars) and LiDAR sensor to detect frontal ones (pedestrians). In particular, not only does the LiDAR sense the distance between two objects, but it is also sensible to the angle between the LiDAR unit and the angle on which the pulse to detect the object was fired: since in reality the roads are not completely smooth, the laser pulse may not be fired horizontally with respect to the road.

An alternative to increase the detection accuracy may be represented by a moving camera, which is capable of detecting objects in every direction, for example by attaching a DC motor to the Raspberry Pi camera or using directly a camera with a higher optical range.

References

[1] J. Cusack, "How driverless cars will change our world," Available at https://www.bbc.com/future/article/20211126-how-driverless-cars-will-change-our-world (11/2021).

[2] S. Coicheci and I. Filip, "Self-driving vehicles: current status of development and technical challenges to overcome," in *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2020, pp. 000 255–000 260.

[3] S. Kim, "These are the 10 best family cars with self-driving features in 2022," Available at https://www.hotcars.com/best-family-cars-with-self-driving-features/ (01/2022).

[4] Synopsys, "What is adas?" Available at https://www.synopsys.com/automotive/what-is-adas.html.

[5] A. K. Jain, "Working model of self-driving car using convolutional neural network, raspberry pi and arduino," in *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018, pp. 1630–1635.

[6] L. Venturi and K. Korda, *Hands-on: Vision and Behaviour for Self-Driving Cars*, 1st ed. Packt Publishing, 2020.

[7] J. He, S. Sun, D. Zhang, G. Wang, and C. Zhang, "Lane detection for track-following based on histogram statistics," in *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, 2019, pp. 1–2.

[8] OpenCV, "Hough line transform," Available at https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html.

[9] P. Maya and C. Tharini, "Performance analysis of lane detection algorithm using partial hough transform," in *2020 21st International Arab Conference on Information Technology (ACIT)*, 2020, pp. 1–4.

[10] A. Kumar and P. Simon, "Review of lane detection and tracking algorithms in advanced driver assistance system," *International Journal of Computer Science and Information Technology*, vol. 7, pp. 65–78, 08 2015.

[11] L. Wei, Z. Li, J. Gong, C. Gong, and J. Li, "Autonomous driving strategies at intersections: Scenarios, state-of-the-art, and future outlooks," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 44–51.

[12] C. Wang, "Research and application of traffic sign detection and recognition based on deep learning," in *2018 International Conference on Robots Intelligent System (ICRIS)*, 2018, pp. 150–152.

[13] TechVidvan, "Detect objects of similar color using opencv in python," Available at https://techvidvan.com/tutorials/detect-objects-of-similar-color-using-opencv-in-python/?msclkid=9ef10edbb00911ec9c807f7092cdb0af.

[14] G. Shuqing and L. Yuming, "Traffic signal light detection and recognition based on canny operator," *Journal of Measurements in Engineering*, vol. 9, no. 3, pp. 167–180, aug 2021. [Online]. Available: https://doi.org/10.21595/jme.2021.22024

[15] A. Raghunandan, Mohana, P. Raghav, and H. V. R. Aradhya, "Object detection algorithms for video surveillance applications," in *2018 International Conference on Communication and Signal Processing (ICCSP)*, 2018, pp. 0563–0568.

[16] Z. Yi and H. Xiaoyong, "Research on pedestrian detection system based on tripartite fusion of "hog+svm+median filter"," in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 2020, pp. 484–488.

[17] A. Rosebrock, "Hog detectmultiscale parameters explained," Available at https://pyimagesearch.com/2015/11/16/hog-detectmultiscale-parameters-explained/ (11/2015).

Ringraziamenti

In primis vorrei ringraziare la mia famiglia per avermi sostenuta sia psicologicamente che economicamente in ogni decisione presa durante la mia carriera universitaria.

In secundis vorrei ringraziare l'Università, il mio relatore e i miei colleghi del Team PoliTron, senza i quali questa tesi e l'esperienza fatta durante la BFMC22 non sarebbero stati possibili.

Infine vorrei ringraziare il mio ragazzo Massimiliano per avermi accompagnata durante questo ultimo anno di magistrale, per avermi sopportata durante le sessioni più dure della mia vita e per essermi sempre vicino.