

POLITECNICO DI TORINO

Master in Mechanical Engineering



Master's thesis

VIRTUAL ENVIRONMENT FOR AN AUTONOMOUS VEHICLE'S LiDAR PERCEPTION

Overview of Apollo and LG SVL Simulator and the challenges of LiDAR perception

Candidate:

Stefano Trentini

Supervisors:

Daniela Misul
Gianluca Toscano
Alberto Bertone

Acknowledgments

I would like to acknowledge and give my most sincere thanks to my supervisor Daniela Misul, for believing in me and offering me this opportunity. This work would not have been possible without her.

Many thanks also to my *Teoresi*'s supervisors Gianluca Toscano and Alberto Bertone for welcoming me into the company and giving me without hesitation all the tools I needed to complete this work.

I must also thank *Teoresi*'s Salvatore Dorner and Alessandro Serrapica for the continuous technical support they gave me throughout these intense months, I would not have completed this project without their help.

Thanks to Petoria, Bdb and all my friends: I could not have succeeded without you by my side. You are my most precious asset, without which I would feel lost. A special mention has to be made for Elisabetta, who was there supporting me during the toughest times of this journey, and without whom I would be where I am today.

Last, I want to thank my family always for supporting me, no matter what the situation might be. Thank you for the unconditional love you give me, there is nothing more important in life. I love you.

Contents

List of figures	6
List of tables	10
Abstract	11
1. Introduction and theoretical background	12
1.1 Autonomous driving	12
1.2 Perception and object recognition.....	13
1.3 Evolution of autonomous driving in history	15
1.4 Sensors and LiDARs.....	22
1.5 Data representation from LiDAR scans	25
2. Most popular CNNs for perception	28
2.1 Convolutional Neural Networks for object detection	28
2.2 Evaluation metrics	30
2.2.1 Intersection over Union	30
2.2.2 Mean Average Precision (mAP).....	31
2.3 Two-Stage detectors.....	34
2.3.1 R-CNN.....	34
2.3.2 Fast R-CNN	38
2.3.3 Faster R-CNN	39
2.4 Single-stage detectors	42
2.4.1 YOLO	42
2.4.2 SSD.....	45
2.4.3 Normalized Cuts Segmentation	47
2.5 3D Detection methods.....	48
2.5.1 PointNet	49
2.5.2 PointPillars.....	50

3. LG SVL Simulator and Apollo.....	53
3.1 LG SVL Simulator.....	53
3.2 Apollo	57
3.3 Perception module of Apollo	61
4. Set up of the virtual environment	67
4.1 Car model and sensors	67
4.2 Map	72
5. Simulations.....	77
5.1 First simulation	77
5.2 Second set of simulations.....	82
5.3 Third set of simulations	88
5.4 Fourth set of simulations.....	89
5.5 Fifth set of simulations.....	92
6. Conclusions	97
Bibliography.....	99

List of figures

1. SAE's autonomous driving standard levels	p.13
2. HOG's principle	p.14
3. DPM approach	p.15
4. American Wonder	p.16
5. The Stanford Cart	p.17
6. DARPA's ALV	p.18
7. Stanford University's winner in the 2005 challenge	p.19
8. Lexus RX450h modified from Google for testing	p.20
9. Waymo's driverless taxi	p.21
10. Pros and cons of most used sensors in ADAS	p.22
11. Typical sensor configuration and respective range	p.22
12. Schematic representation of a mechanical LiDAR	p.23
13. Schematic representation of a solid-state LiDAR	p.24
14. Representation of a RGB image	p.25
15. Raw LiDAR point cloud	p.26
16. BEV (first 3 from the left) and front view (right)	p.27
17. Taxonomy of object detectors	p.28
18. Architectures of two-stage (a) and single-stage (b) object detection algorithms	p.29
19. IoU visualization	p.30
20. Real case of IoU visualization	p.31
21. Precision and recall chart	p.32
22. Precision and recall curve	p.33

23. Precision and recall curve, interpolated	p.34
24. Architecture of R-CNN	p.35
25. Over-segmentation	p.35
26. Result of Selective Search	p.36
27. AlexNet feature extractor	p.37
28. Fast R-CNN pipeline	p.38
29. Fast R-CNN and R-CNN time comparison	p.39
30. Architecture of Faster R-CNN	p.40
31. Anchor boxes	p.41
32. Faster R-CNN testing time compared	p.41
33. Grid and example of bounding box	p.43
34. Scheme of the proposed method	p.43
35. Architecture of YOLO	p.44
36. Architecture of SSD	p.45
37. Anchor boxes example	p.46
38. Weighted graph	p.47
39. Bad partitioning of isolated nodes	p.48
40. Architecture of PointNet	p.49
41. Architecture of PointPillars	p.51
42. Pillars representation	p.51
43. Pillar Feature Net detail	p.52
44. Pipeline of SVL Simulator coupled with an AV framework	p.55
45. Borregas Avenue, one of the standard maps of SVL	p.55
46. Different types of sensor vision	p.56
47. HD Map example and annotation tool	p.57
48. Pipeline of modules interaction	p.58
49. Versions of Apollo	p.58

50. Task tab	p.59
51. Module controller tab	p.60
52. Route editing tab	p.60
53. Perception module architecture	p.61
54. ROI and LUT	p.63
55. Scheme of Apollo's FCNN for cell-wise obstacle prediction	p.65
56. Obstacle clustering	p.65
57. <i>Teoresi's</i> YOYO, side view	p.67
58. <i>Teoresi's</i> YOYO, back view	p.68
59. Detail of LiDARs and cameras	p.69
60. Example vehicle in Unity Editor	p.70
61. XEV digital model	p.70
62. Example of sensor customization	p.71
63. Example of sensor configuration	p.72
64. Example of initial map without annotations	p.73
65. Example of map with annotations	p.74
66. Example of map with annotations	p.74
67. Our map, first section	p.75
68. Figure 68. Our map, second section	p.75
69. LiDAR FoV	p.76
70. CNN Segmentation parameters	p.78
71. First simulation	p.79
72. Module delay associated to figure 70	p.79
73a. First simulation, different time instant	p.80
73b. Lateral view of scenario of Fig. 72	p.80
74. Detection of pedestrians at crosswalk	p.81
75. LiDAR FoV of Fig. 73	p.82

76. Detection with reduced height_thresh	p.83
77. Detection with reduced height_thresh	p.83
78. Detection with reduced point_cloud_range	p.84
79. Detection with confidence_range lowered	p.84
80. Detection with confidence_range lowered	p.85
81. Detection with confidence_range lowered	p.85
82. Detection with objectness_tresh lowered	p.86
83. Detection with lower orientation angle	p.88
84. Detection at intersection	p.89
85a. Close range detection of pedestrian	p.90
85b. LiDAR's FoV of Fig. 85a	p.90
86. Detection of two pedestrians	p.91
87. FoV of the two rotative LiDARs	p.93
88. FoV of all 3 LiDARs, superimposed	p.93
89. Perception with the 3 LiDARs working	p.94
90. Perception with the 3 LiDARs working	p.94
91. Module delay	p.95

List of tables

1. Precision/Recall table	p.32
2. Comparison of YOLO and other methods	p.44
3. Comparison of SSD and other methods	p.46
4. Comparison of PointNet and other networks	p.50
5. Comparison of PointPillars and other networks	p.52
6. Input and output of HDMap ROI filter	p.62
7. Sensor configuration of the different simulations	p.86

Abstract

This thesis has been carried out as a collaboration between *Politecnico di Torino*, the candidate and the Italian company *Teoresi*, a leader consultancy company in the fields of aerospace and defense, automotive, railway, artificial intelligence, among many others.

Teoresi recently purchased an electric vehicle with the aim of developing a self-driving car using a commercial framework, that shall be discussed later. For this purpose, the company made some modification on the chassis to mount some of the required sensors (2 mechanical LiDARs, 1 solid-state LiDAR, 3 cameras) needed for a state-of-the-art autonomous vehicle. Hence, the first part of this paper is an introduction to the overall concept of autonomous driving and its history.

Before putting the car onto the actual road, it has been decided to first test the self-driving framework and all the sensors on a driving simulator, as to prevent any real and potentially costly damage to the car and people, and to make sure the software works properly. The latter is the ultimate goal of this work, putting the focus on the object detection module.

The first chapter is an introduction to AV and its history. In the second chapter, an overview of the most popular Deep Learning models for object detection is given, since the understanding of how they work is vital for the outcome of a quality simulation.

The next chapter introduces the simulator (LG SVL) and the autonomous driving framework, Apollo. In the successive section we will show how to upload onto the simulator the digital version of the car, its sensors and the map, used as the standard scenario of drive.

In the chapter 5 chapter we show the simulations and discuss one main challenge presented: during the first simulations, we noticed that the object detection from the solid-state-LiDAR only wasn't performing as expected. In particular, the framework is not able to correctly detect the obstacles and pedestrians when they are at short distance. Since the used framework is still experimental, and the solid-state-LiDAR is also a relatively new sensor, there is no-to-little literature of such specific problem. Therefore, there is no source or study that can be adopted to verify how the parameters and use of the algorithms may be changed to address the above-mentioned problem. For these reasons, the approach of this issue will be mainly experimental, using different configurations and interpreting the results. Finally, the last section covers the possible solutions to be implemented in the future to resolve this issue.

Chapter 1

Introduction and theoretical background

In this chapter, theory and background information about the subjects crucial for this thesis are presented. At first, a general definition of autonomous driving is given. Subsequently, we will give a first overview on object recognition and perception of autonomous vehicles (AVs).

1.1 – Autonomous driving

Nowadays, autonomous driving may be the hottest topic in the automotive field. We can define autonomous driving as the ability of a vehicle to ride safely in all type of situations, and react to road contingencies in a way to preserve safety of passengers and road users. One of the most crucial challenges of autonomous driving research is environment detection. For humans, this task is relatively simple and subconscious, as we can easily detect any object, possible obstacles, and vulnerable road users (VRU) through our vision. Conversely, for a machine, such task is rather non-trivial, as it has to take over all the tasks related to road observation. We must then first equip the machine with sensors that simulate human perception, and then teach them with dedicated software how to use the data these sensors return, in order to recognize what object are present in the environment, and which decisions to take based on their type, location, and state of movement.

Clearly, autonomous driving comes in levels: before total autonomy there are many other tasks that help drivers drive safely, or in general have an easier driving experience. These tasks are included into what it is called Advanced Driver-Assistance Systems (ADAS). In this category we can list some features that have been perfected over the last decade, such as adaptive cruise control, lane keeping, blind spot detection, signal recognition. These are all steps that needed to be completed before aiming for full self-driving vehicles.

In this regard, to standardize the level of autonomy, a classification system with six levels – ranging from fully manual to fully automated systems – was published in 2014 by standardization body SAE International [1]: *'Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems'*, which details are revised periodically. This classification is based on the amount of driver intervention and attentiveness required, rather than the vehicle's capabilities, although these are loosely related. In the United States in 2013, the National Highway Traffic Safety Administration (NHTSA) adopted the SAE standard, and SAE classification became widely accepted. *Figure 1* explains in detail the requirements of each level.

SAE J3016™ LEVELS OF DRIVING AUTOMATION™
Learn more here: sae.org/standards/content/j3016_202104

Copyright © 2021 SAE International. The summary table may be freely copied and distributed AS-IS provided that SAE International is acknowledged as the source of the content.

	SAE LEVEL 0™	SAE LEVEL 1™	SAE LEVEL 2™	SAE LEVEL 3™	SAE LEVEL 4™	SAE LEVEL 5™
What does the human in the driver's seat have to do?	You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			You are not driving when these automated driving features are engaged – even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
<small>Copyright © 2021 SAE International.</small>						
	These are driver support features			These are automated driving features		
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met		This feature can drive the vehicle under all conditions
Example Features	<ul style="list-style-type: none"> • automatic emergency braking • blind spot warning • lane departure warning 	<ul style="list-style-type: none"> • lane centering OR adaptive cruise control 	<ul style="list-style-type: none"> • lane centering AND adaptive cruise control at the same time 	<ul style="list-style-type: none"> • traffic jam chauffeur 	<ul style="list-style-type: none"> • local driverless taxi • pedals/steering wheel may or may not be installed 	<ul style="list-style-type: none"> • same as level 4, but feature can drive everywhere in all conditions

Figure 1. SAE's autonomous driving standard levels

1.2 – Perception and object recognition

The systems described in the previous section can be perceived as an intermediate step towards the full automation, but for now, they only offer assistance to the driver to enhance comfort and safety. For a car to be fully autonomous, it has to be able to take over all the tasks of the human driver. The most crucial one is object detection: a computer vision task which goal is recognizing and localizing objects such as pedestrians, traffic lights/signs, other vehicles, and barriers in the AV vicinity.

Not only a good estimation and correct recognition is needed, but a crucial factor is also the velocity of such tasks. A pseudo-real-time detection module allows the vehicle to react quickly in order to plan its next decision, be it a change in direction or be it braking/accelerating. It is the foundation for high-level tasks during self-driving operation, such as object tracking, event detection, motion control, and path planning.

The evolution of object detectors began in the 2000s with a real-time human face detector developed by P. Viola and M. Jones [2], which was one of the breakthrough algorithms in this field, using Haas features (line, edge and rectangular features) to extract information from a picture or frame.

Only a few years later the Histograms of Oriented Gradients (HOG) gained popularity as detectors for pedestrian [3]. This approach divides the image in cells and then calculates the gradients among the different pixels and used it as features to characterize the shape of a human body, which will be then compared with the HOG of the unknown images to see if they contain pedestrians. The classification was then performed using a Support Vector Machine (SVM). Fig. 2 describes the process.

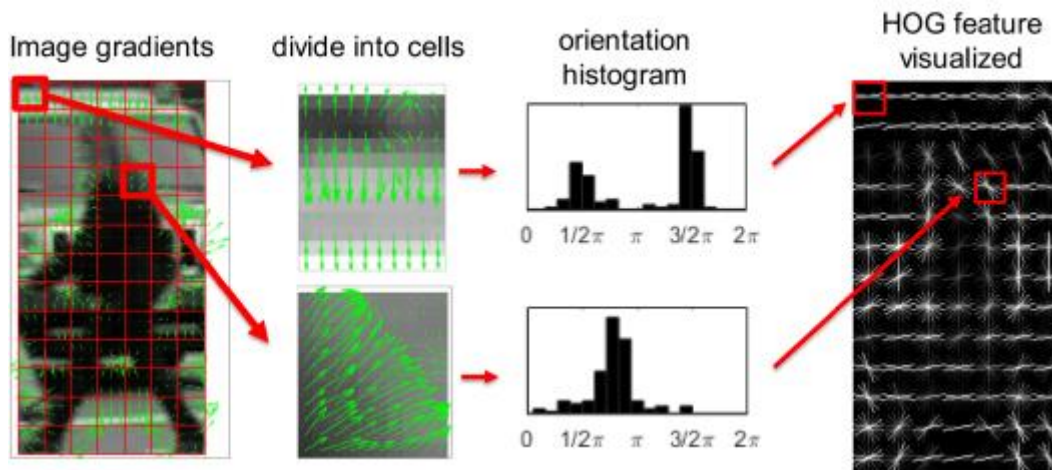


Figure 2. HOG's principle

HOG were successively improved as to become the Deformable Part-based Models (DPM), the first approach which allowed recognition of multiple objects at the same time [4]. The main difference is that DPM accounts for the idea that the same class of objects can present differences in terms relative positions of their key components, or different scales, so they split up every object into its parts (for example head, arms, legs, trunk), and by doing so it was possible to recognize objects of much different scales, seen by another point of view or even partially occluded, because the model learnt the relative locations that all the parts have among them. Fig. 3 explains this approach.

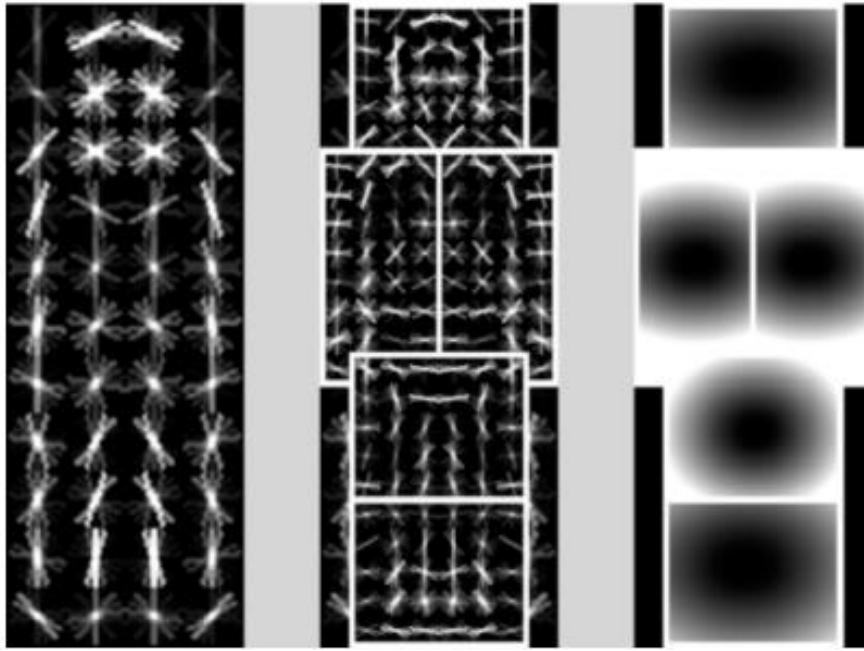


Figure 3. DPM approach

Starting from 2013, due to the increasing interest in deep convolutional neural networks (CNN), another family of object detectors using CNN was proposed, called the Regions CNN, or R-CNN [5]. We shall discuss the main properties of such detectors in a successive section of this paper. These methods led to a breakthrough for multiple object detection, with a 95.84% improvement in mean Average Precision (mAP) over the state-of-the-art. This development helped redefine the efficiency of object detectors and made them attractive for entirely new application domains, such as for AVs. Since 2014, the evolution in deep neural networks and advances in GPU technology have paved the way for faster and more efficient object detection on real-time images and videos. AVs today rely heavily on these improved object detectors for perception, pathfinding, and other decision making.

1.3 – Evolution of autonomous driving in history

It is worth having a quick look at the history of autonomous driving to understand its evolution. The first records of driverless vehicles date back to the 1920s, where, precisely in 1925, inventor Francis Houdina built a car, named *American Wonder*, that was able to drive through the streets of New York thanks to an antenna that received radio signals from an operator that could control the vehicle remotely from another car following the first one. The antenna would then transfer the signals to switched which controlled small actuators that directed every movement of the car. The ride was not a total success, as the vehicle bumped into another at some point, but

despite this early mishap, the industry did not give up hope on remote-controlled cars. The car is shown in Fig. 4.

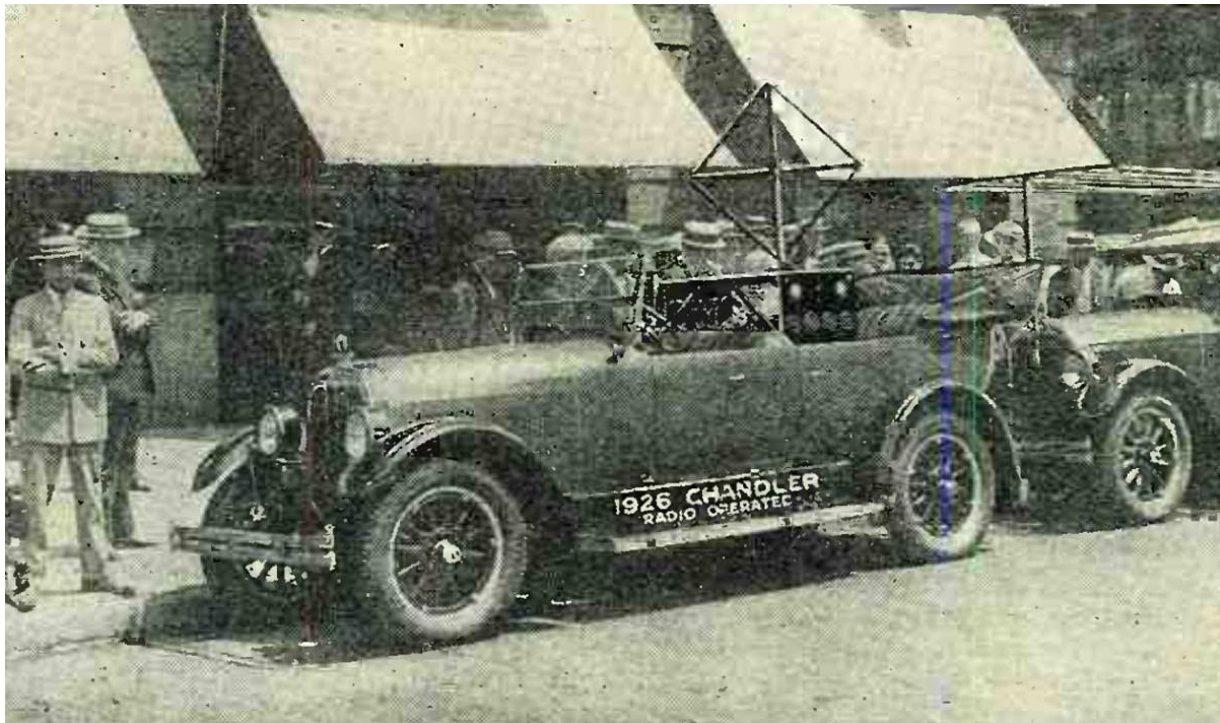


Figure 4. American Wonder

A decade later, precisely in 1939, General Motors sponsored Norman Bel Geddes's Futurama at the World's Fair: it was an electric vehicle guided by radio-controlled electromagnetic fields and operated from magnetized circuits embedded in the roadway. The car contained sensors that could detect the current flowing through the wires embedded in the road, which could be manipulated to move the steering wheel left or right.

The embedded electrical wires into the road were the leading technology that was implemented through the 1960s with projects from RCA Labs, General Motors, Ohio State University, the United Kingdom's Transport and Road Research Laboratory, Stanford University, University of Illinois.

The first automated robot to include a camera was Stanford University's Stanford Cart in 1961, a simple cart constituted by 4 wheels, a fixed position camera and a battery. The project was meant to be a study on how to remotely control a rover to be sent on Mars by a NASA mission. The camera was not the only novelty introduced with this project: a computer program was written which guided the cart through narrow spaces, gaining knowledge of the world totally from the images broadcasted on the TV system (Fig. 5).

The cart used several types of stereo vision to locate objects around it in three dimensions and to learn its own motion. It was able to plan a path to a desired destination on the basis

of a model built with this information. The plan changed as the cart detected new objects on its journey. The process was reliable for short runs, but extremely slow: after driving one meter it stopped, took new pictures, and processed them. Then it planned a new path, executed a little of it, and paused again. It successfully drove the cart through several 20-m courses (each taking about 5 h). The *Stanford Cart* was really a breakthrough in the field and was an inspiration to many projects yet to come.

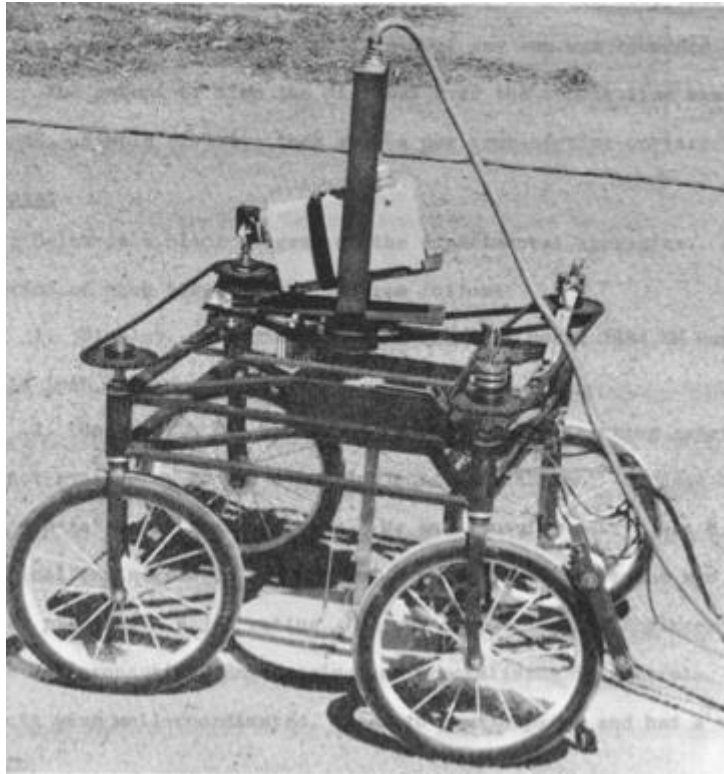


Figure 5. The Stanford Cart

In the 1980s, Prof. Ernst Dickmanns and his research group at Bundeswehr University of Munich built the world's first actual robotic street vehicles, using computer vision and probabilistic algorithms to drive 20 km on an empty highway with velocities of up to 96 km/h [6]. His research continued in the European publicly-funded EUREKA-project Prometheus, which ran from 1987 to 1995. It included many European participants and received a significant amount of public funding.

During the same years, DARPA's Autonomous Land Vehicle (ALV) project was created. It was an immense 12-foot tall, 8 wheeled vehicle that achieved the first road-following demonstration using LiDAR, computer vision and autonomous control to direct the vehicle at speeds of up to 31 km/h. It was able to drive autonomously in the hills outside of Denver in 1985. It is shown in Fig. 6.

The 1990s marked a crucial point in autonomous driving. As Carnegie Mellon University pioneered the use of neural networks for steering, these quickly became the new and

exciting technology applied to vehicles. Companies such as General Motors, Toyota, Honda, Mercedes-Benz all joined the conversation with their own projects



Figure 6. DARPA's ALV

One of the most noticeable projects among them was Ernst Dickmanns' VaMP, which in 1994 was able to drive about 1000 km on a 3-lne highway in Paris in standard traffic conditions at speeds up to 130km/h. The car was able to autonomously manage to drive in free lanes, convoy lines and passing of other vehicles. One year later Dickmann proposed a new project in which an S-Class Mercedes completed a 1600km trip achieving even higher speeds and with a peak of 158 km driven without human intervention.

A similar result was obtained by Carnegie Mellon's Navlab project in 1995 , which completed a 5000km trip across the USA, of which 98.2% was autonomously controlled. Note that all these vehicles equipped a bulk mainframe in the inside to perform all the calculations required.

THE DARPA CHALLENGES

By the early 2000s, the autonomous car industry was in full swing. The U.S. Department of Defense's research arm, DARPA, sponsored a series of challenges, called *Grand Challenges*, to expedite autonomous cars. The prize for the winning team was 1 million dollars.

In 2004 the first edition took place, and it consisted in a 240km race through the Mojave desert, but no team was able to complete the race. Although the outcome was not the desired one, this event attracted much interest and gained popularity among the scientific community. In 2005 the second challenge was held, and this time around five teams completed the course, Stanford University and Carnegie Mellon University got respectively 1st, 2nd and 3rd place. For the third edition, in 2007, DARPA moved the competition to an urban area course. Teams had to complete an urban course, negotiating four-way intersections, blocked roads or parking lots, in mixed traffic with robotic as well as human-driven vehicles. Over 90 teams applied to the Urban Challenge, 53 teams received site visits, 36 were invited to the final event, and just eleven made it into the final race. The event saw the first autonomous vehicle traffic jam as well as the first, though minor, collision of two self-driving vehicles. Carnegie Mellon's Vehicle Boss won the 60-mile race, followed by Stanford's Junior. Figure 7 shows 2005 edition's winner.

The influence of these events on the industry was enormous: the DARPA prize money attracted top-notch researchers, who attracted leading automotive manufacturers (Volkswagen, General Motors among others), large automotive suppliers (Continental, Mobileye, Bosch), chipmakers (Intel was powering the top-2 teams) and also Google as one of the main sponsors.

The challenges also contributed to the evolution of the LiDAR sensors, which rapidly became the most important sensors for the vehicles' perception. By the time the 2007 Urban Challenge took place, these sensors produced by the company Velodyne were mounted on top of five of the six vehicles that finished the course. This unforeseen success sparked a whole new industry, where Velodyne is now still one of the top manufacturers.



Figure 7. Stanford University's winner in the 2005 challenge

As previously mentioned, Google also joined the business, attracted from the very promising technologies implied in the DARPA *Grand Challenges*. For a couple of years, Google kept secrecy around its project, until in 2010 they officially announced their own self-driving vehicles program, hiring some of the brightest engineers that previously worked for Stanford's team. The company from Mountain View started modifying and equipping cars (Fig. 8) for testing in real world scenarios, first in Arizona, and successively in Florida and California. They founded a secret lab called *secret X lab*, which was the center of research and development for the project. In some years Google was able to successfully develop self-driving golf carts and little carts used on their campus in California.



Figure 8. Lexus RX450h modified from Google for testing

Many other major automotive manufacturers, like BMW, General Motors, Ford, Toyota, Nissan, Mercedes-Benz, Volkswagen, Audi, and Volvo, are now in the process of testing driverless car systems, and are doing so successfully. Many systems have been introduced or improved, such as lane adaptive cruise control, lane keeping, parking assist, driver fatigue detection.

In October 2014 Tesla announced its first version of Autopilot. The vehicles equipped with this system were capable of lane control with autonomous steering, braking, and

speed limit adjustment based on signals image recognition. The system also includes autonomous parking. This software was able to give Tesla a level 2 autonomy according to SAE. Differently from other companies in the business, Tesla vehicles don't rely on LiDAR sensor, but rather only on cameras and radars. Up to this point, Tesla's features reached level of 2 and 3, according to SAE, but the aim of the company is that of achieving a Full Self Driving (FSD) level 5 autonomy. After years of research, Tesla released in 2020 their beta software for level 5, initially given only to a small portion of employees and engineers to be tested. Tesla's CEO, Elon Musk, stated "*[the beta version] will be limited to a small number of people who are expert and careful drivers*", and by doing so the company allowed 1000 customers to use their FSD software. As the months went on, Tesla slowly increased the number of customers allowed to use the FSD system. As of January 2022, the vehicles equipped with such update is about 60000.

In 2018, the previously known Google's *secret X lab*, now become one of its subsidiary companies, launched the first driverless autonomous taxi fleet in Phoenix, Arizona. After years of experience in the business, *Waymo* is today one of the top and most advanced autonomous driving companies in the world, and the first to ever launch a fleet of self-driving taxi service, operating in Phoenix, Arizona. One of the fleet's vehicles is shown in Fig 9. Singapore also launched the same service with the company NuTonomy.

As in 2022 the vast majority of commercialized autonomous vehicles (including public transportation) reach level 3 of SAE's standards, but, as we've seen, the evolution is very rapid.



Figure 9. Waymo's driverless taxi

1.4 – Sensor and LiDARs

Autonomous vehicles today are equipped with several different sensors: radars, LiDARs, cameras, sonars (or ultrasound), GPS, IMU (accelerometers, gyroscopes..). The reason for such a variety of devices is that each one presents its pros and cons, depending on weather conditions, range, task, lighting conditions, as illustrated in Fig. 10. The red, yellow and green dots respectively represent poor, acceptable and good performance of the specific task.

	Camera	Radar	LiDAR	Ultrasonic	LiDAR+Radar+Camera
Object detection	●	●	●	●	●
Object classification	●	●	●	●	●
Distance estimation	●	●	●	●	●
Object edge precision	●	●	●	●	●
Lane tracking	●	●	●	●	●
Range of visibility	●	●	●	●	●
Functionality in bad weather	●	●	●	●	●
Functionality in poor lighting	●	●	●	●	●

Figure 10. Pros and cons of most used sensors in ADAS

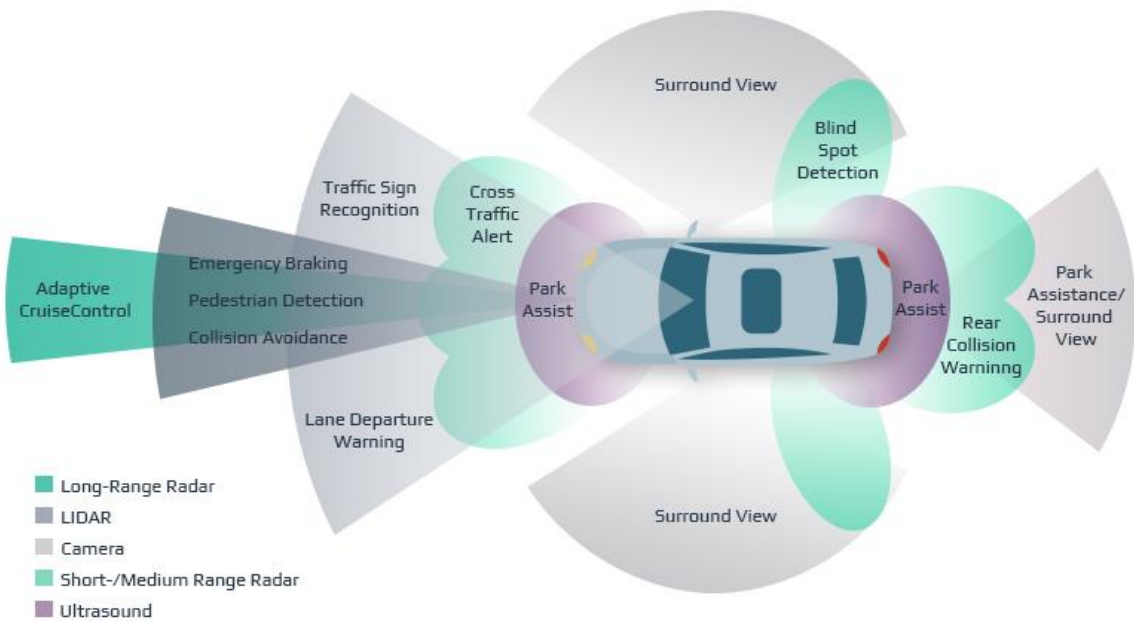


Figure 11. Typical sensor configuration and respective range

Depending on the task they serve, their position on the vehicle is affected. A typical configuration is the one depicted in Fig. 11, although some companies may prefer to not use some of the types included in this example.

Among these sensors, perhaps the most important one for the vehicle's perception is the LiDAR, so let us summarize its capabilities and why it has become the first and most important sensor for this task is autonomous vehicle.

A LiDAR ('light detection and ranging') is an active sensor used to draw a high-definition map of the surrounding environment based on the reflection of laser beams. The first LiDARs were developed in the early 1960s by an aerospace company with the goal of satellites tracking. They rapidly became popular also in military applications, but the first time the public became aware of its usefulness and accuracy was when it was used in the Apollo 15 mission to scan the surface of the moon. Today different kinds of this technology are used in countless fields such as military, geology, physics, automotive, climatology, oceanography, archeology, astronomy, robotics and more.

In recent years these sensors have become the predominant choice for autonomous driving, since a precise and real-time 3D map of all the surrounding environment is needed to feed to the perception algorithms to identify pedestrians, vehicles, motorbikes, cyclists, eventual obstacles on the road and other unexpected objects for the safety of the passengers.

A LiDAR sensor emits laser beams at a certain frequency up to millions of times per second; these beams then strike the surrounding objects and reflect on them, travelling their way back to the sensor, which, based on the time that the pulse needs to travel back, can calculate the distance of the struck object from the emitter.

For the purpose of this paper, we will focus on two of the main types of LiDARs:

1. Mechanical LiDAR

This is the most commonly used LiDAR family. It is constituted by a laser emitter and a motor that rotates the emitter at 360° at a given angular velocity. The emitter sends pulses of laser at a given frequency, and then the receiver catches the reflected beams. Fig. 12 illustrates, in simplicity, its architecture.

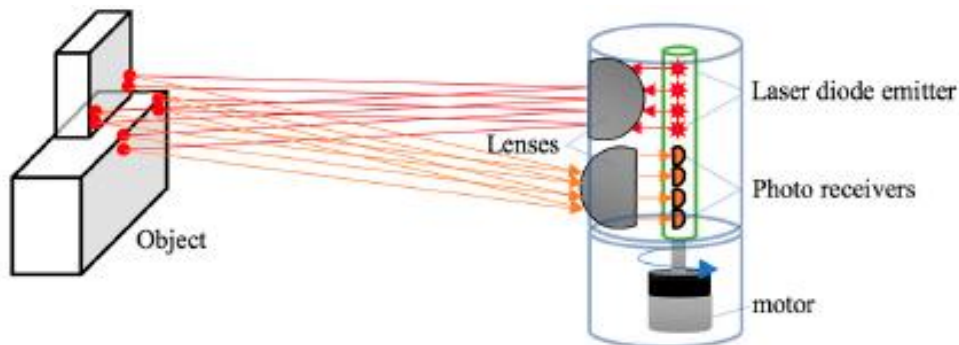


Figure 12. Schematic representation of a mechanical LiDAR

Given its rotative mechanical parts, that must be very precise in order to keep a steady rotation velocity, and their fragility, this type of LiDAR results in being very expensive and more prone to failure.

2. Solid-state LiDAR

Given the high cost of the traditional mechanical LiDARs, in recent years a new version of the same sensor has been developed, which is called solid-state LiDAR. This type has no rotating parts, hence the field-of-view (FoV) is just a portion of the environment that lies ahead. The cost is much lower since the lack of a rotating motor, but drawback is that obviously this sensor does not provide a 360° scan, while only a portion of the surrounding space. The technology used to calculate the distance of the points is the same of the classic LiDAR. Fig. 13 shows a typical solid-state configuration.

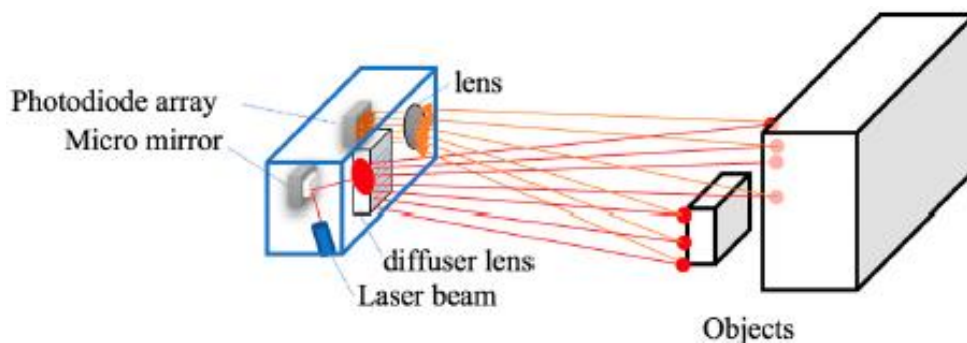


Figure 13. Schematic representation of a solid-state LiDAR

Given their accuracy, with the surge of the autonomous driving business LiDARs have become the most important sensor when dealing with the perception of the vehicle. The main technical characteristics that differ from sensor to sensor are:

- Horizontal field-of-view (if solid-state then $< 360^\circ$)
- Vertical field-of-view (typically around 40°)
- Number of planes (also called channels, which is the number of lasers emitted vertically)
- Number of points scanned per second
- Horizontal angular resolution (depends on rotation velocity and frequency of emission)

All companies, except Tesla Motors, base their perception algorithms on the input that LiDARs provide describing the surrounding environment.

1.5 – Data representation from LiDAR scans

The image (or video) representation deriving from a camera is well known, capturing the environment as a number of tiny portions of space, called pixels, and assigning each one of them values of intensity (for black and white images) or color domain (most typically RGB, but also CMYK, HSV and other models are sometimes used). The result is often defined as matrix-like visualization, where such matrix is more than 2-dimensional when the color system is based on 3+ entries, and where each dimension represents the intensities of a specific color. This makes the image easy to visualize and provide discriminative information, but nevertheless due to the lack of a depth information it is a challenging task to detect objects of varying shapes, sizes, and orientations by using only one camera. Fig. 14 provides a simple explanation of representation of a RGB image as a 3-dimensional matrix, where each dimension represents respectively the intensity of Red, Green and Blue.

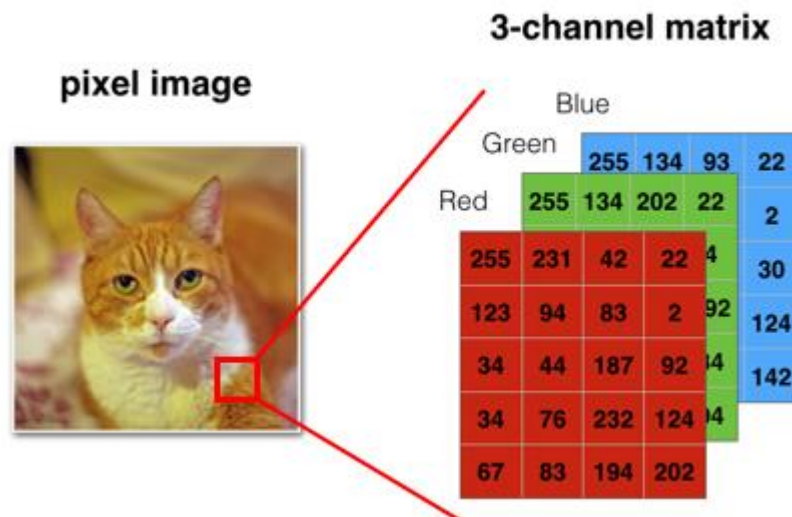


Figure 14. Representation of a RGB image

On the other hand, a lidar provides a 3-dimensional point cloud of the environment, which yields information about the distance in the scene, i.e. depth. This data is very well applicable for obstacle detection. However, point clouds obtained by lidar sensors lack color and texture information and only sparsely capture the environment.

From a raw LiDAR scan, we can obtain the following properties of the points, XYZ coordinates, reflectivity (sometimes also called intensity), and distance, calculated using the ToF (*time of flight*, i.e. the time needed for the light beam to strike the object and return to the detector). We must take into consideration that the raw points obtained from a raw LiDAR scan are unstructured and often sparse, meaning that there are many areas where the signal doesn't return to the detector, implying a null value of intensity because no object was in the range of the sensor hence it did not reflect back the beam. Apart from

being sparse, a LiDAR point cloud suffers from highly variable density, especially when increasing the distance from the sensor. The main reasons behind this are non-uniform sampling of the space, effective range, occlusion, relative position and orientation of the sensor. Fig. 15 is a typical point cloud derived from a mechanical LiDAR.

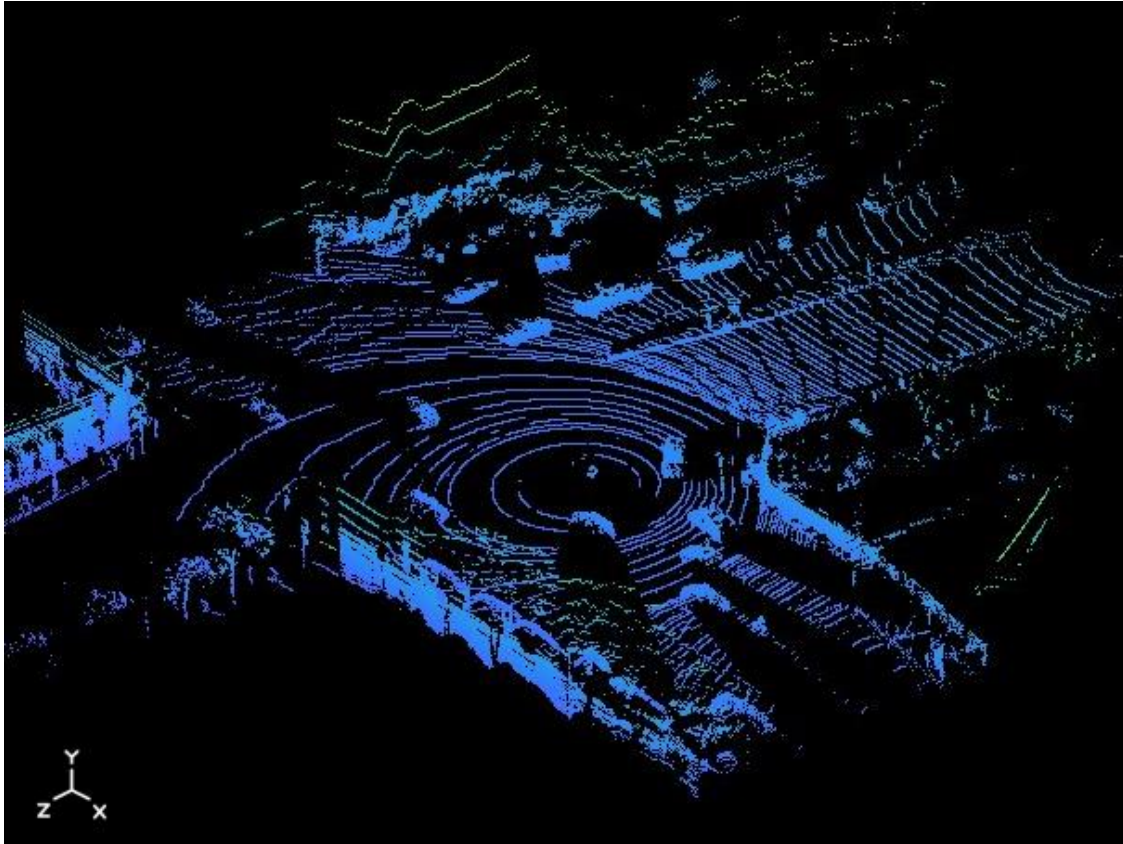


Figure 15. Raw LiDAR point cloud

There are several approaches used in most recent models to process a raw point cloud for object detection. One of the most popular ones is the voxelization of the point cloud as used in [7], which consists in sub-dividing the space into 3D cubes called voxels (the 3D correspondent of pixels), so that a grid of such voxels can be created and is easier to manage and structure raw points.

Another very used method is projecting the 3D space into a 2D map, like used in [8], [9]. Using this approach, one may think that one dimension is lost, but instead of representing it as a 3 dimensional cartesian space, this third dimension is often represented by a color map. Color maps often represent altitude, intensity distance. The most popular 2D projections are the Bird's Eye View (BEV) and the frontal view, which uses cylindrical coordinates. Examples in Fig. 16.

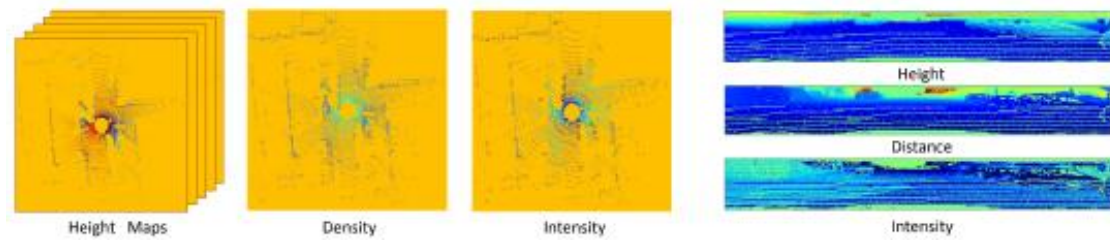


Figure 16. BEV (first 3 from the left) and front view (right)

Chapter 2

Most popular CNNs for perception

2.1 – Convolutional Neural Networks for object detection

At present, deep learning models have been widely adopted in the whole field of computer vision, including general object detection and domain-specific object detection. Most of the state-of-the-art object detectors utilize deep learning networks as their backbone and detection network to extract features from images or videos.

Object detection consists of two sub-tasks: localization, which involves determining the location of an object in an image (or video), and classification, which involves assigning a class (e.g., ‘pedestrian’, ‘vehicle’, ‘traffic light’) to that object. Existing domain-specific image object detectors usually can be classified according to the type of data used as input or to the type of network used. Fig. 17 explains the taxonomy of object detectors.

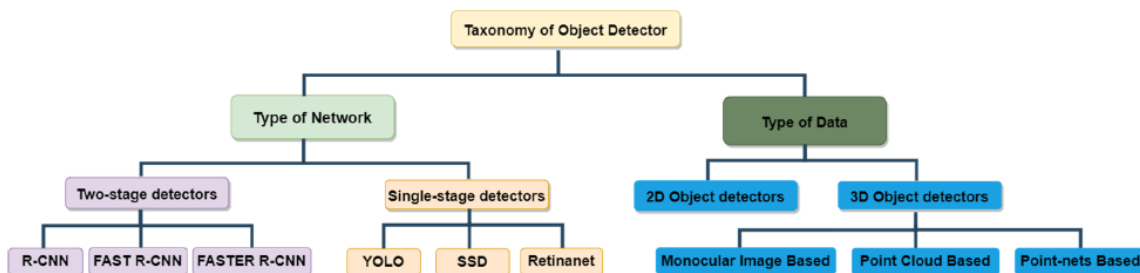


Figure 17. Taxonomy of object detectors

We will now focus on the taxonomy based on the type of network. The state-of-the-art networks used for object detection can be divided into two families: the two-stage detectors and the single-stage detectors.

Two-stage deep learning-based object detectors are subdivided into 2 tasks: region proposals and object classification. In the first phase, as the name suggests, the algorithm proposes several Regions Of Interest (ROIs) of an input image that present the highest

likelihood of containing an object, often using the Selective Search algorithm [10]. In the second stage, only the most ‘promising’ ROIs are passed on and only in these the classification will take place, discarding, for sake of computation time, the non-interesting regions. This step is done with deep CNNs to extract features from the image, topped off by a SVM or fully connected layer to give each category a score of likelihood. See Fig. 18 (a) for a schematic representation of the pipeline. The two-stage networks have normally higher localization and object recognition accuracy than the single-stage methods. The most popular networks of this family are R-CNN and all its derivatives (Fast R-CNN, Faster R-CNN, Mask R-CNN). We will see more in detail the characteristics of some of them in the next section.

On the other hand, single-stage object detectors use a single feed-forward neural network that creates bounding boxes and classifies objects in the same stage. These detectors are faster than two-stage detectors, hence their use for real-time applications is more advised, but they are also typically less accurate. Among the most popular single-stage detector are YOLO (*You Only Live Once*) and SSD (Single-Shot Detector). Fig. 18 (b) shows the typical architecture.

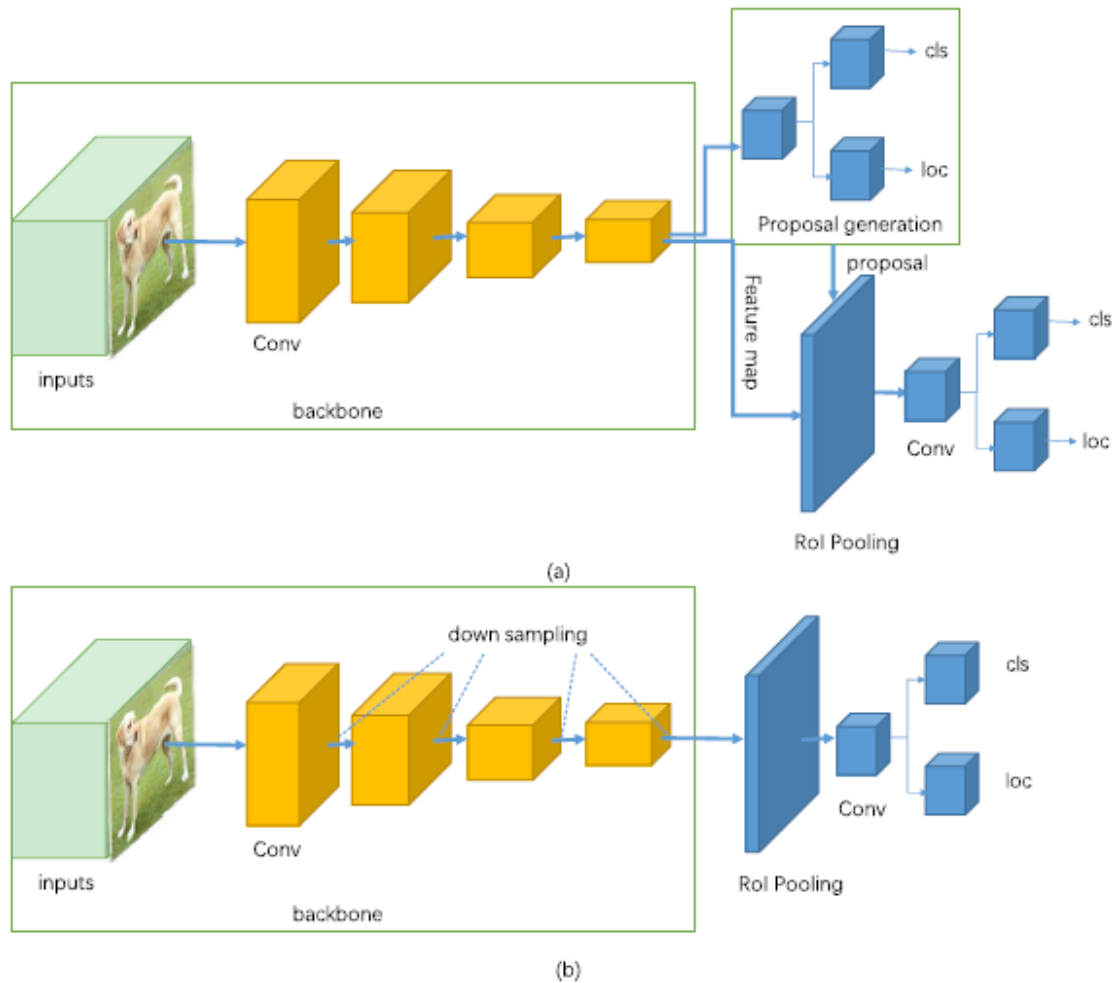


Figure 18. Architectures of two-stage (a) and single-stage (b) object detection algorithms

2.2 – Evaluation metrics

Before analyzing different CNN architectures, it is worth mentioning the main evaluation metrics which are used to measure the precision of statistical and machine learning model. Evaluating machine learning models or algorithms is essential for any project, and there are many different types of evaluation metrics available to test a model. We will name just the most important related to our project. It is nonetheless very important to use multiple evaluation metrics to evaluate one model, and the reason is because an algorithm may perform well using one measurement from one evaluation metric, but may perform poorly using another measurement from another evaluation metric. Using evaluation metrics is critical in ensuring that your model is operating correctly and optimally.

2.2.1 – Intersection over Union

A very important metric to quantify the performance of an object detector is the Intersection over Union (IoU). As already explained, object detector not only have to classify objects, but also correctly localize them by drawing a bounding box that encapsules the target object. This evaluation metric focuses solely on the latter task. In fact, IoU measures the ratio between the overlapping area, i.e. the intersection of the predicted bounding box and the ground-truth bounding box, and the intersection of the same two areas. It is definitely easier to understand this metric graphically, so let us use Fig. 19 to clarify it.

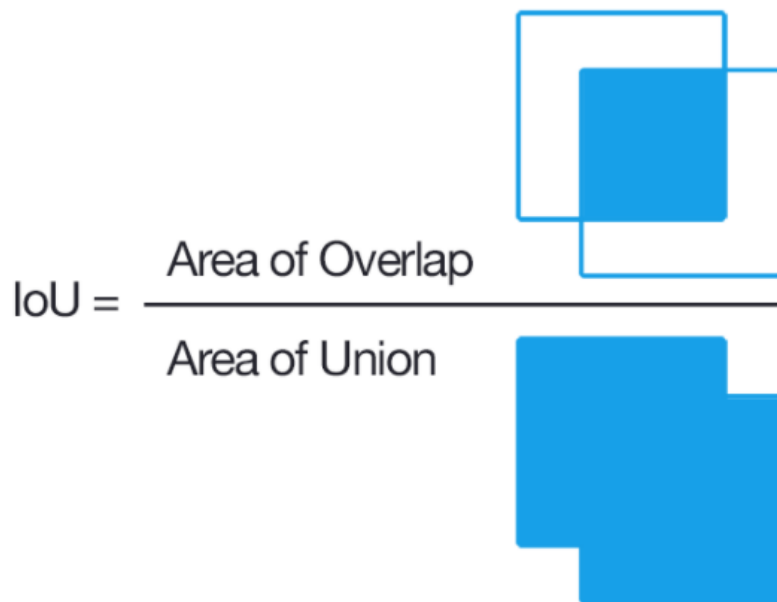


Figure 19. IoU visualization

Clearly, when dealing with 3D detectors, this approach can be taken into 3 dimensions simply substituting areas with volumes. This approach is used by KITTI¹ object detection benchmark and many other datasets.

¹ http://www.cvlibs.net/datasets/kitti/eval_object.php

Usually, a prediction is considered a True Positive (TP) when the value of $\text{IoU} > 0.7$, while it will be considered a False Positive when $\text{IoU} < 0.7$. This threshold can obviously be changed if needed. A False Negative (FN) will be considered when there's an object that hasn't been bounded by any box.

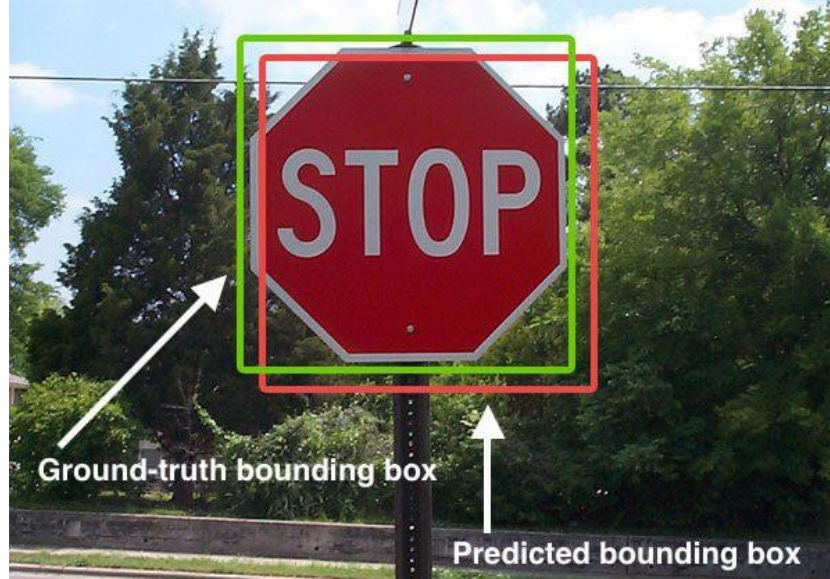


Figure 20. Real case of IoU visualization

2.2.2 – Mean Average Precision (mAP)

The second important, yet a bit more complicated, metric of evaluation of performance of an object detector is the mean Average Precision (mAP).

Before defining that, we need to clearly have in mind two keystone metrics of statistics: precision and recall. Precision is given as the ratio of true positives (TP) and the total number of predicted positives, which also include false positives (FP). It measures how many of the predictions that your model made were actually correct. In formula:

$$precision = \frac{TP}{TP+FP} \quad (1)$$

Similarly, the recall of a given class in classification, is defined as the ratio of TP and total of ground truth positives, which also includes the false negatives (FN). In formula:

$$recall = \frac{TP}{TP+FN} \quad (2)$$

Fig. 21 gives a visual representation to better understand these concepts.

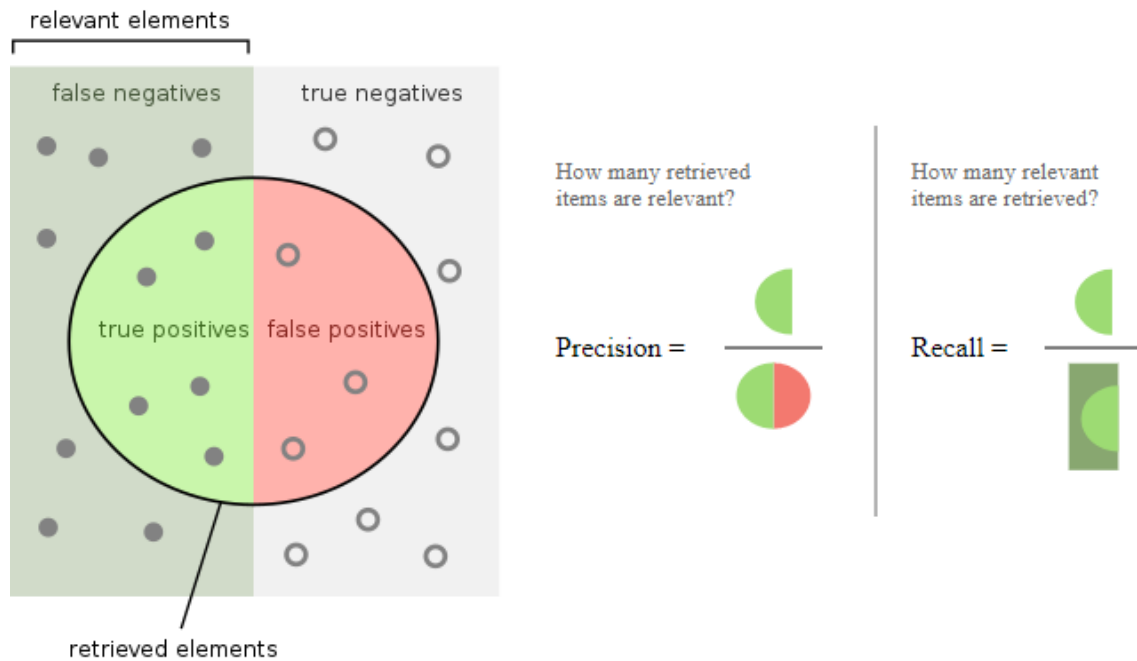


Figure 21. Precision and recall chart

Given these definitions, we can create a table where, knowing how many true positives we have in our dataset, we can insert the value of precision and recall for each prediction of the model. An example of a dataset containing 3 TP and 4 FP is given in Table 1.

Table 1. Precision/Recall table

Rank	Correct?	Precision	Recall
1	True	1.0	0.2
2	True	1.0	0.4
3	False	0.67	0.4
4	False	0.5	0.4
5	False	0.4	0.4
6	True	0.5	0.6
7	True	0.57	0.8
8	False	0.5	0.8
9	False	0.44	0.8
10	True	0.5	1.0

Let's take the row with rank #3 and demonstrate how precision and recall are calculated in this case.

Precision is the proportion of TP, which turns out as $2/3 = 0.67$

Recall is the proportion of TP out of the possible positives: $2/5 = 0.4$.

We can now draw the well-known precision-recall curve in Fig.22, simply plotting the points highlighted in Table 1.

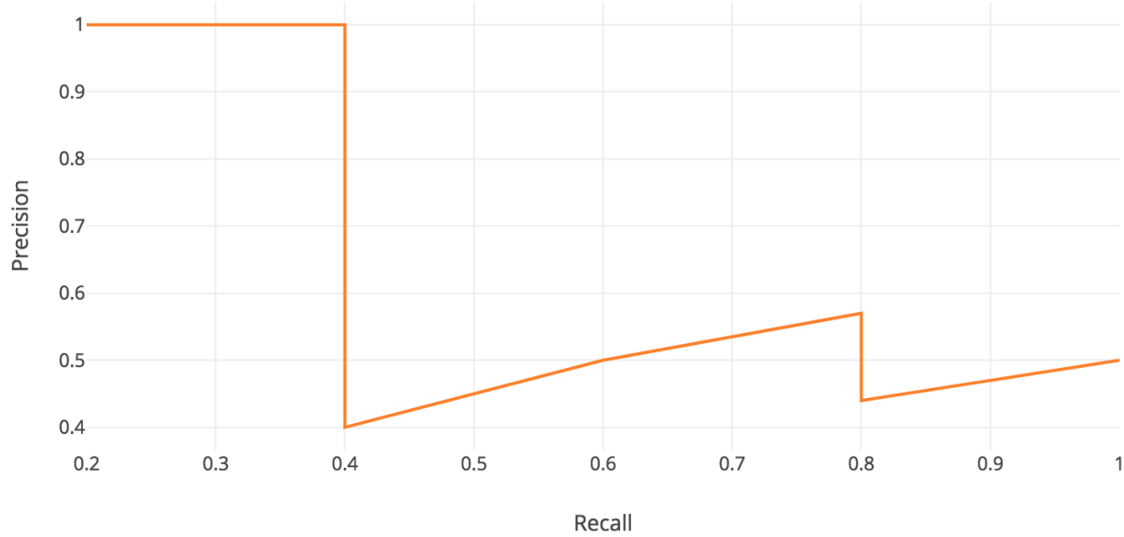


Figure 22. Precision and recall curve

We can now give the general definition of Average Precision (AP), that is the area under the graph. In formula:

$$AP = \int_0^1 precision(r) \cdot dr \quad (3)$$

where r indicates the recall.

To make (3) anatically easy to solve, PASCAL VOC [11] 2008 challenge proposed to interpolate the data (green line in Fig. 23) and evenly split the recall in 11 points (0, 0.1, 0.2, ..., 1.0). Then, take the maximum precision value for each of these points, which will be called $precision_{interp}$. Mathematically, we replace the precision value for recall \hat{r} with the maximum precision for any recall $\geq \hat{r}$.

$$precision_{interp}(r) = \max_{\tilde{r} \geq r} precision(\tilde{r}) \quad (4)$$

The result is shown graphically in Fig. 23.

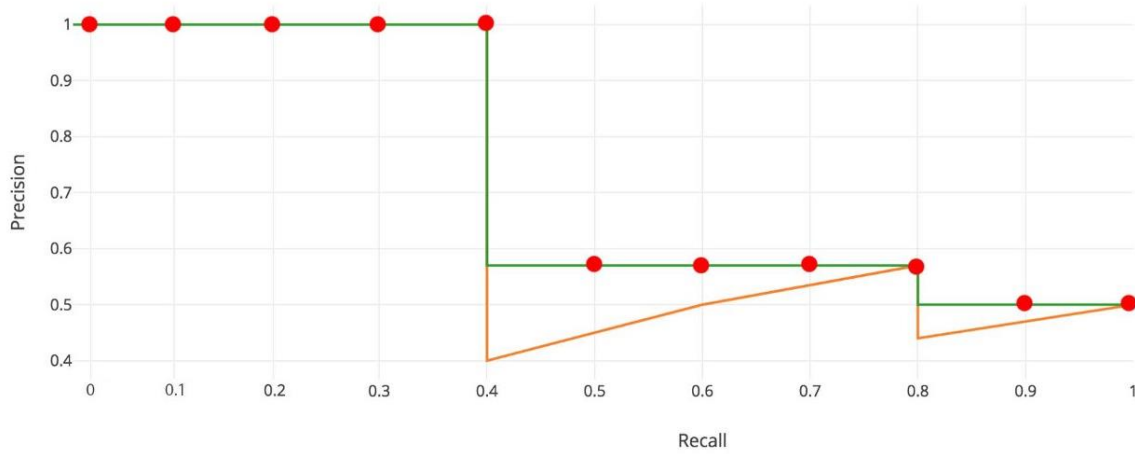


Figure 23. Precision and recall curve, interpolated

At this point, the Average Precision AP for a class of object is simply the average of the 11 values of interpolated precisions:

$$AP = \frac{1}{11} \sum_{r \in (0.0, \dots, 1.0)} precision_{interp}(r) \quad (5)$$

Finally, we define the Mean Average Precision as the average of all Average Precisions over all categories of objects:

$$mAP = \frac{1}{N} \sum_N AP_N \quad (6)$$

2.3 – Two-Stage detectors

In this section we will give an overview of the most popular families of two-stage object detectors and their functioning.

2.3.1 – R-CNN

In 2014, *Girshick et al.* proposed a way to combine classification task with localization task with their paper “*Rich feature hierarchies for accurate object detection and semantic segmentation*”, also called R-CNN [5]. The implemented method changed the approach to object detection forever. The main problem they were trying to address was how to computationally reduce the cost of spatially localizing the objects of interest, since you cannot know in advance how many there might be in one single picture, and in which scale or unexpected shape. Previous methods implemented a sliding window approach, where basically different sized rectangles are just superposed over the whole image in order to analyze at those smaller portions in a brute-force-method. The

problem is that this generates a giant number of smaller images to look at, hence the computational cost is way too high. Methods using the sliding-window approach include [12] [13]. The revolutionary idea that came with R-CNN was to pair the network for classification with an algorithm that could indicate Region of Interest (RoI), so as to look for objects only in those areas which the network selects as the most likely to contain objects of interest, instead of analyzing the whole image. Figure 24 outlines the architecture of the proposed pipeline.

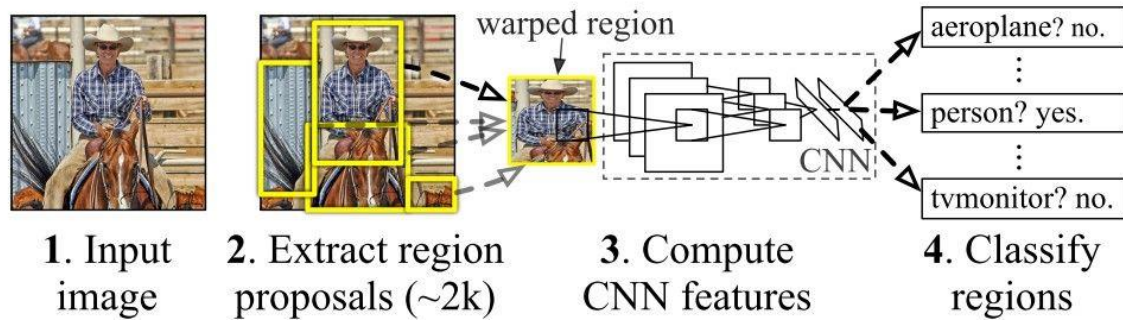


Figure 24. Architecture of R-CNN

Among the different algorithms of region proposal, the authors decided to opt for the Selective Search algorithm [14], a method based on similarity among adjacent areas through different parameters. As a first step, the Selective Search starts with the over-segmentation based on the segmentation of images proposed by Felzenszwalb *et. al.* [15], which adopts pixel intensity as its main feature. By over-segmenting, the objects in the image will present an exceedingly amount of segmented regions, like shown in Fig. 25.



Figure 25. Over-segmentation

As a second step, Selective Search algorithm takes these over-segments as initial input and performs the following steps:

1. Add all bounding boxes corresponding to segmented parts to the list of regional proposals
2. Group adjacent segments based on similarity
3. Reiterates from point 1 until a certain threshold is reached

At each iteration, larger segments are formed and added to the list of region proposals. Hence, we create region proposals from smaller segments to larger segments in a bottom-up approach. The similarity used by the method is made of 4 factors:

1. Color similarity: a color histogram of 25 bins is calculated for each channel of the image and histograms for all channels are concatenated to obtain a color descriptor.
2. Texture similarity: texture features are calculated by extracting Gaussian derivatives at 8 orientations for each channel.
3. Size similarity: size similarity encourages smaller regions to merge early. It ensures that region proposals at all scales are formed at all parts of the image. If this similarity measure is not taken into consideration a single region will keep gobbling up all the smaller adjacent regions one by one and hence region proposals at multiple scales will be generated at this location only.
4. Shape compatibility measures how well two regions fit into each other. If one fits into another region we would like to merge them in order to fill gaps, and if they are not even touching each other they should not be merged.

The total similarity is then calculated as a linear combination of the 4 similarity features, and the result is an indication of approximatively 2000 region proposals. Fig. 26 shows an example of the best 200 regions applied to a non-automotive scenario.

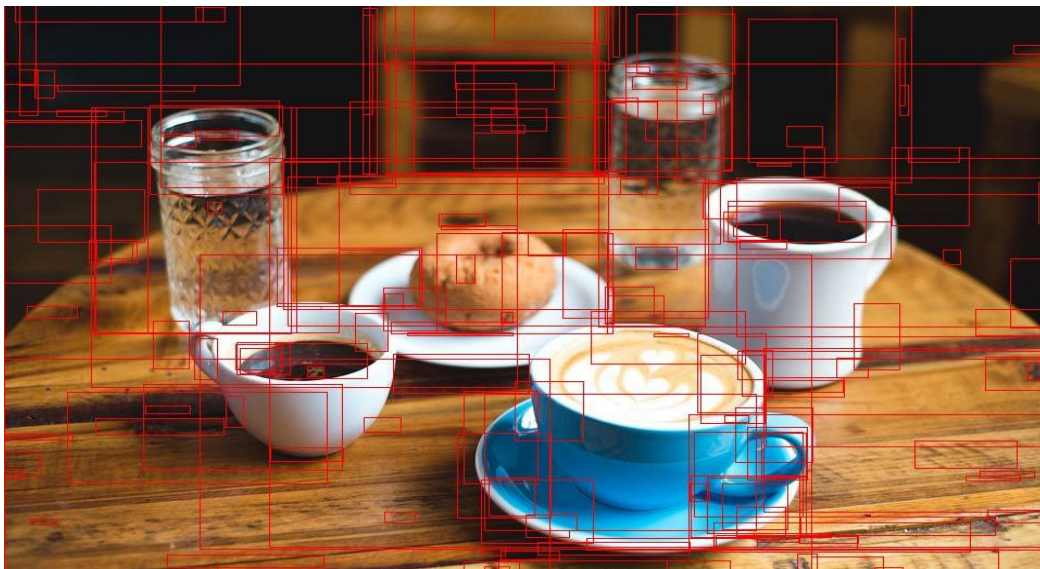


Figure 26. Result of Selective Search

At this point, having the RoI, the model has to reshape the regions as to align them all to a unique size, since the input of the successive feature extractor must be fixed. The authors decided to use a 227×227 pixels size window.

The next step is the feature extractor from the Regions of Interest. With this deep network, a multi-dimensional vector identifying the region's features is created. There are countless possibilities of choice of such network today, where the most popular are VGG [16], Inception-Resnet-v2 [17], AlexNet [18], GoogLeNet [19]. The authors decided to opt for AlexNet, which at the time was arguably the most advanced state-of-the-art feature extractor. This feature extractor consists in 5 convolutions, and 3 fully connected layers at the end, the last of which is a softmax layer. The whole process generates a 4096-dimensional vector that encapsules the region's features. The overall architecture is shown in Fig. 27.

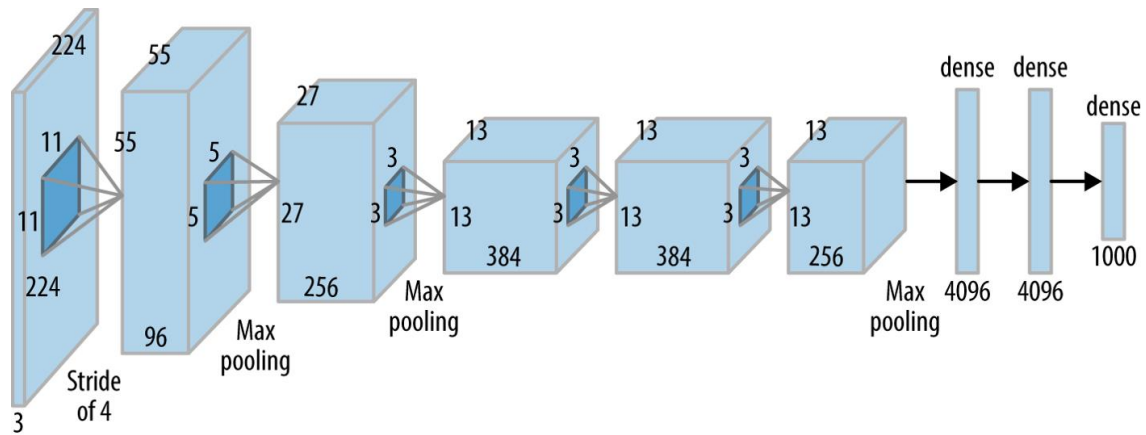


Figure 27. AlexNet feature extractor

For the final classification output, R-CNN replaces the softmax layer with a pre-trained Support Vector Machine (SVM) algorithm to calculate confidence scores for each region of interest that describe the likelihood that a certain category of object has been detected. A Support Vector Machine is a family of machine learning supervised algorithm based on the principle that, given labelled data, one can classify unknown data by selecting an hyperplane that best separates the data and therefore the different classes.

The last module of R-CNN is dedicated to bounding-box regression. Basically it is fed as input the center coordinates of the region proposed, its width and height and also takes the same parameters but of the ground-truth bounding-box. Then these parameters are compared and the model is trained to try to learn a transformation that maps the proposed bounding-box exactly over the ground-truth bounding-box. The authors achieved with this method an increase in mAP of 3%, so it is worth mentioning it.

As much as this model was a revolution in the field of object detectors, it sure presented its drawbacks:

1. It is a multi-stage model, where each stage is an independent component. Thus, it cannot be trained end-to-end, and it means the Region Proposal, SVM and bounding box regression networks cannot be trained all in parallel.
2. R-CNN depends on the Selective Search algorithm for generating region proposals, which takes a lot of time. Moreover, this algorithm cannot be customized to the detection problem.
3. Each region proposal is fed independently to the CNN for feature extraction. This makes it impossible to run R-CNN in real-time. In fact, each image takes about 47 seconds to go through the whole pipeline.

2.3.2 – Fast R-CNN

Fast R-CNN [20] was published a year later solely by Ross Girshick, and aims at solving some of the problems that affected its predecessors, R-CNN. As the name suggests, the main improvement was making the new pipeline faster than the previous one. Girshick (2015) improved the training procedure by unifying three independent modules of R-CNN into one single, jointly trained stage and increasing shared computation results.

Instead of extracting CNN feature vectors independently for each region proposal (remember R-CNN generated about 2000 RoI), this model aggregates them into one CNN forward pass over the entire image and the region proposals share this feature matrix. Then the same feature matrix is branched out to be used for learning the object classifier and the bounding-box regressor. In conclusion, computation sharing speeds up R-CNN. The pipeline of this method is shown in Fig. 28.

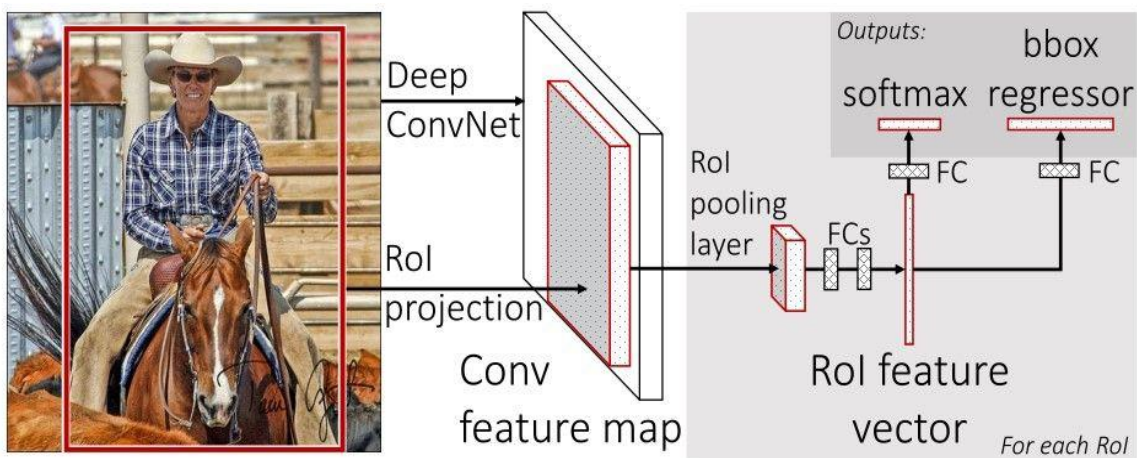


Figure 28. Fast R-CNN pipeline

To go more into detail, the real difference-maker was the new layer called RoI Pooling, which is able to extract equal-length feature vectors from all region proposals in the same image. In this way, Faster R-CNN shares computations (most importantly convolutional layer calculations) across all proposals rather than doing the calculations for each proposal independently.

The RoI Pooling layer is basically a type of max pooling to convert features in the projected region of the image of any size, $H \times W$, into a small, fixed window, $h \times w$. The input region is divided into $h \times w$ grids, then apply max-pooling in each grid.

The extracted feature vector using the ROI Pooling is then passed to some FC layers. The output of the last FC layer is split into 2 branches:

1. Softmax layer to predict the class scores, that substitutes the SVM used in R-CNN
2. Fully connected layer to predict the bounding boxes of the detected objects

The result in terms of training time and test time that Fast R-CNN brings were astonishing, comparing with its predecessor. These results are summarized in Fig 29.

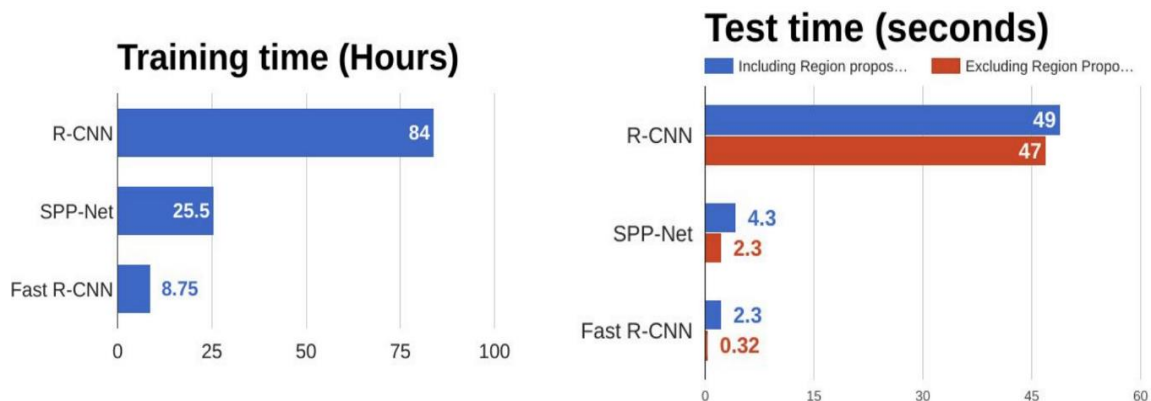


Figure 29. Fast R-CNN and R-CNN time comparison

2.3.3 – Faster R-CNN

In 2016, Girshick *et. al.* worked on a solution to make Fast R-CNN even faster. The problem was that the RoI proposal network was still much time consuming, so the authors' idea was to integrate the region proposal model into the CNN. Faster R-CNN [21] is doing exactly this: construct a single, unified model composed of a Region Proposal Network (RPN) and fast R-CNN with shared convolutional feature layers. In this way, the feature extraction uses the same convolutions as the region proposal network. The architecture of this model is shown in Fig. 30.

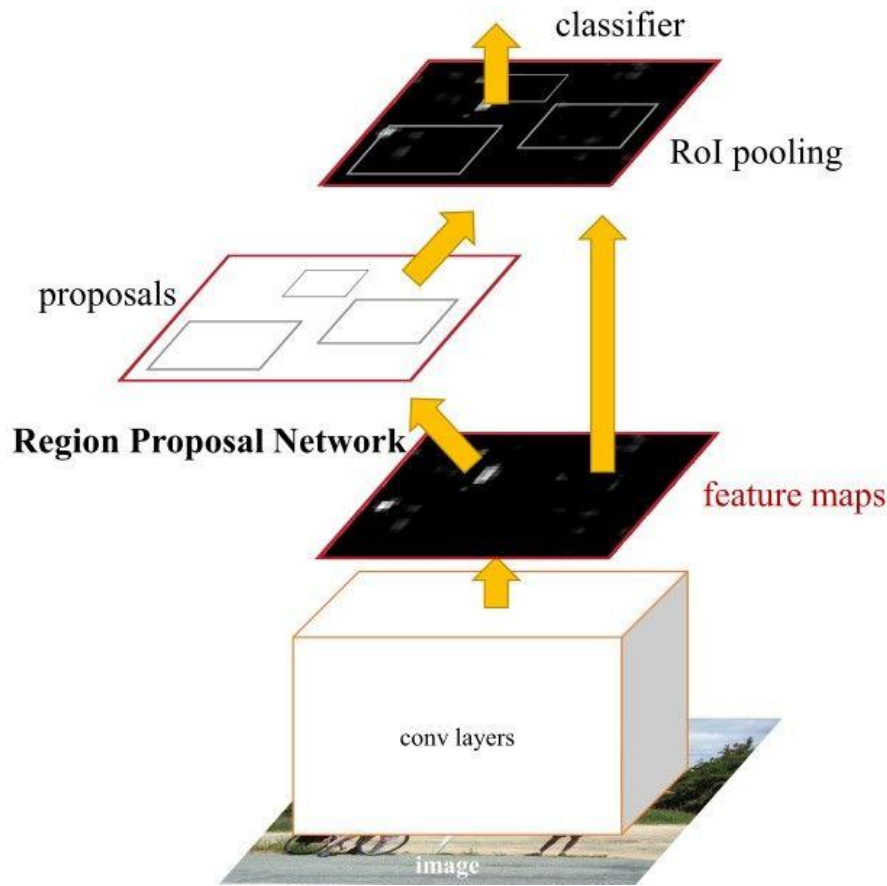


Figure 30. Architecture of Faster R-CNN

The main new contributions of this paper are:

1. Region proposal network (RPN), which is a fully convolutional network that generates proposals with various scales and aspect ratios. The RPN implements the terminology of neural network with attention to tell the object detector (Fast R-CNN) where to look. Because the proposals are generated using a network, this can be trained end-to-end to be customized on the detection task. Thus, it produces better region proposals compared to generic methods like Selective Search, that cannot be customized on a specific task. The RPN processes the image using the same convolutional layers used in the Fast R-CNN detection network. Thus, the RPN does not need extra time to produce the proposals.
2. Rather than using pyramids of images (i.e. multiple instances of the image but at different scales) or pyramids of filters (i.e. multiple filters with different sizes), this paper introduced the concept of anchor boxes. An anchor box is a reference box of a specific scale and aspect ratio. With multiple reference anchor boxes, then multiple scales and aspect ratios exist for the single region. This can be thought of as a pyramid of reference anchor boxes. Each region is then mapped to each reference anchor box through a sliding window approach, and thus

detecting objects at different scales and aspect ratios. Fig. 31 displays some of the aspect ratios of the boxes.

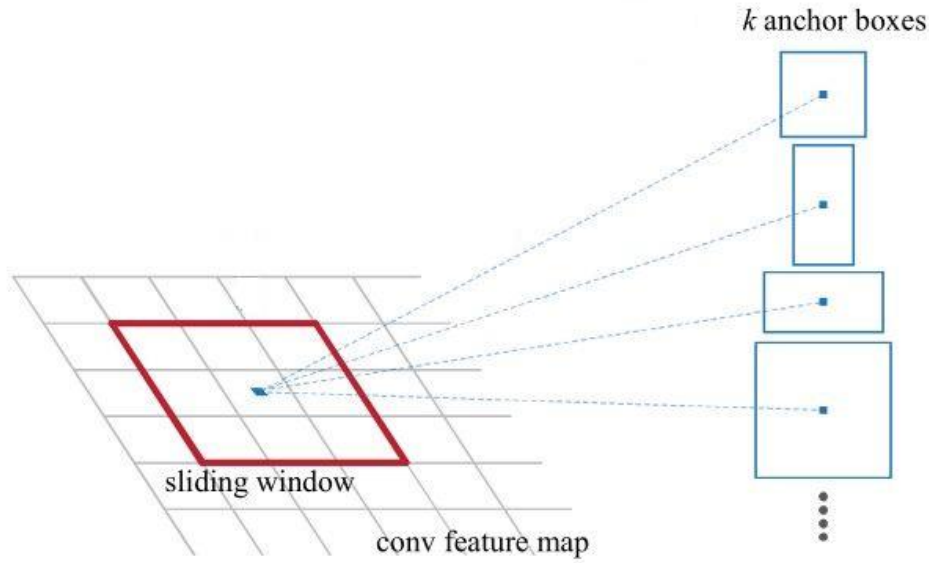


Figure 31. Anchor boxes

Basically, the RPN works on the output feature map returned from the last convolutional layer shared with the Fast R-CNN. This is shown in the next figure. Based on a rectangular window of size $n \times n$, a sliding window passes through the feature map. For each window, several candidate region proposals are generated. These proposals are not the final proposals as they will be filtered based on their "objectness score". Using these reference anchors, a single image at a single scale is used while being able to offer scale-invariant object detectors, as the anchors exist at different scales. This avoids using multiple images or filters.

Once the proposals are found, the rest of the model goes on like the previous Fast R-CNN algorithm.

In Fig. 32 the testing time of this model is compared to the ones of its predecessors.

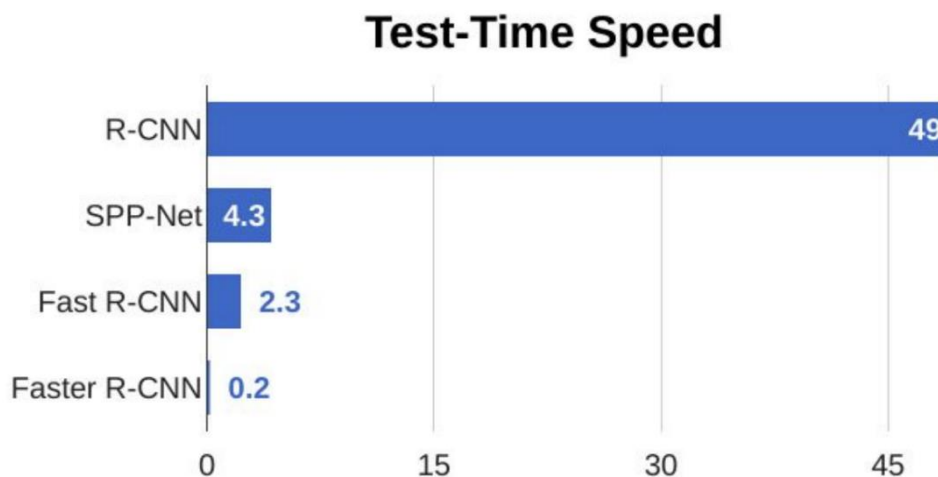


Figure 32. Faster R-CNN testing time compared

2.4 – Single-stage detectors

2.4.1 – YOLO

In 2016 J. Redmon *et. al.* proposed a new approach for object detection with their model *You Only Look Once* (YOLO) [22], which combines what was once a multi-step process, now using a single neural network to perform both classification and prediction of bounding boxes for detected objects. As such, it is heavily optimized for detection performance and can run much faster than running two separate networks. It does this by repurposing traditional image classifiers to be used for the regression task of identifying bounding boxes for objects. In this section we will only explain the first version of the paper, *YOLOv1*, the first of several updates of the paper, that include also Fast YOLO [23] YOLO9000 [24] and YOLOv2 [25]. Although the subsequent iterations feature numerous improvements, the basic idea behind the architecture stays the same. YOLO can perform at 45 frames per second, making it a great choice for applications that require real-time detection. It looks at the entire image at once, and only once - hence the name *You Only Look Once* - which allows it to capture the context of detected objects. This halves the number of false-positive detections it makes over R-CNNs which look at different parts of the image separately. Additionally, YOLO can generalize the representations of various objects, making it more applicable to a variety of new environments.

YOLO is based on the idea of segmenting an image into smaller images. The image is split into a square grid of dimensions $S \times S$, like in Fig. 33. The cell in which the center of the object lies is then responsible for its detection. Each cell predicts B bounding boxes and also a confidence score associated to each box. The confidence score varies between 0 and 1, and reflects the certainty with which we can state if there is an object that lies inside such box, or in other terms can be seen also as the value of Intersection over Union of the predicted and ground-truth boxes. To each bounding box other 4 parameters are associated: the x and y coordinates of its center, the height and the width; hence a total of 5 variables are coupled with each bounding box. The output of the network will be a N -dimensional vector, where N is the total number of classes, where each entry is the confidence that indicates what class the detected object belongs to. Fig. 33 also shows one bounding box, its center, height, width and its confidence.

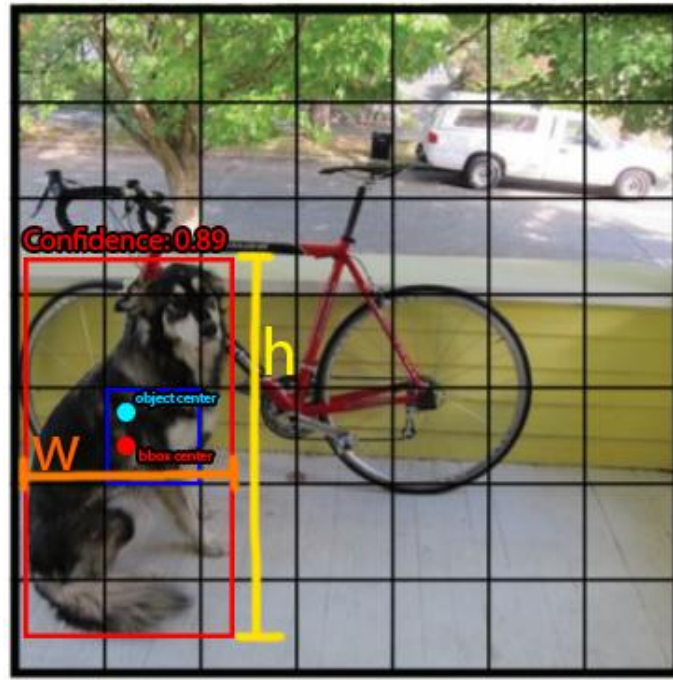


Figure 33. Grid and example of bounding box

The whole image undergoes the same procedure, with each cell predicting B bounding boxes, its 5 parameters, and the vector that predicts the scores for each class. In total, this will create a $S \times S \times (B * 5 + C)$ tensor, where C represents the number of classes. By selecting the class with the highest confidence score, a class probability map will be created over the image, like shown in Fig. 34.

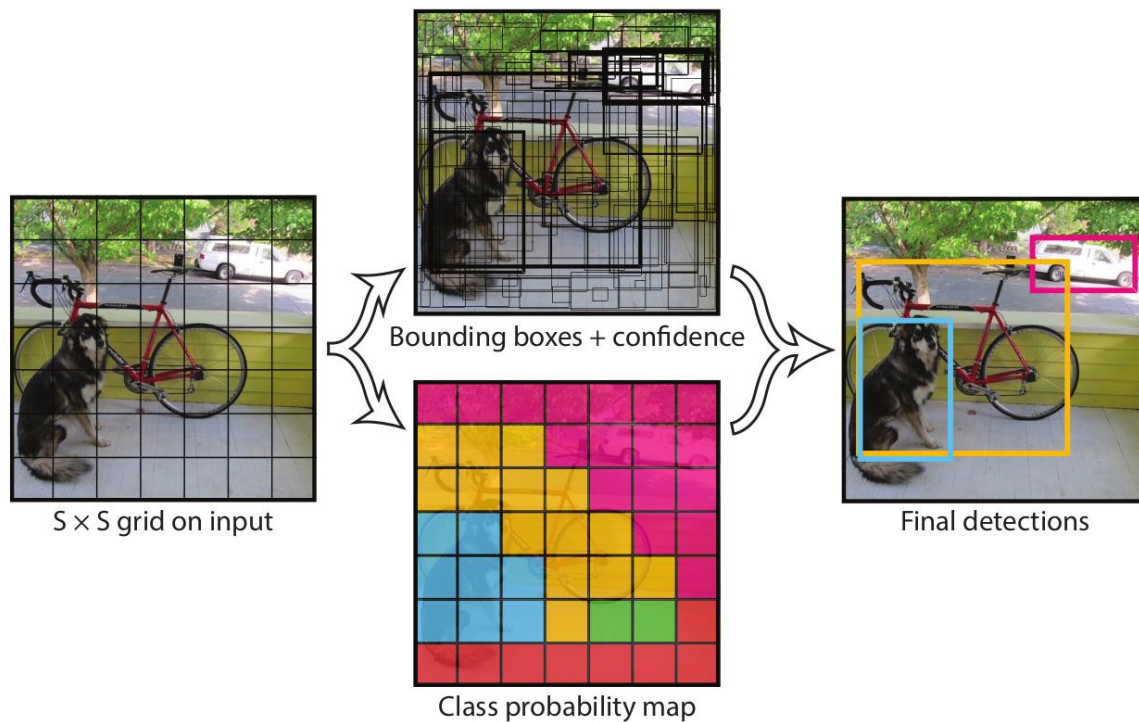


Figure 34. Scheme of the proposed method

YOLO is made up of three components: the head, neck, and backbone. The backbone is the part of the network made up of convolutional layers to detect key features of an image and process them. The backbone is first trained on a classification dataset, such as ImageNet, and typically trained at a lower resolution than the final detection model, as detection requires finer details than classification. In their model, the authors use 24 convolutional layers. The neck uses the features from the convolution layers in the backbone with 2 fully connected layers to make predictions on probabilities and bounding box coordinates. The first 2 parts are shown in Fig. 35. The head is the final output layer of the network which can be interchanged with other layers with the same input shape for transfer learning. As discussed earlier, the head is an $S \times S \times (B * 5 + C)$ tensor and is $7 \times 7 \times 30$ in the original YOLO research paper with a split size S of 7, 20 classes C , and 2 predicted bounding boxes B . These three portions of the model work together to first extract key visual features from the image then classify and bound them.

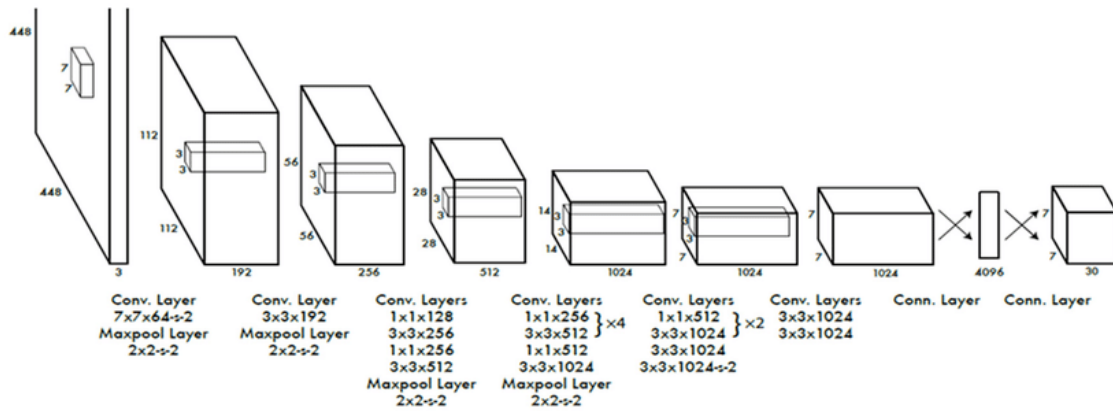


Figure 35. Architecture of YOLO

Table 2 summarizes the performance of YOLO compared to other popular methods.

Table 2. Comparison of YOLO and other methods

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [30]	2007	16.0	100
30Hz DPM [30]	2007	26.1	30
Fast YOLO	2007+2012	52.7	155
YOLO	2007+2012	63.4	45
Less Than Real-Time			
Fastest DPM [37]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[27]	2007+2012	73.2	7
Faster R-CNN ZF [27]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Nonetheless, YOLO also presents some limitations with respect to other object detectors:

1. As already mentioned, each cell will only predict 2 boxes, in the original paper. And though that number can be increased, only one class prediction can be made per cell, limiting the detections when multiple objects appear in a single grid cell. Thus, it struggles with bounding groups of small objects, such as flocks of birds, or multiple small objects of different classes.
2. As the original paper [22] states: “we train the model on a loss function that approximates detection performance, our loss function treats errors the same in small bounding boxes versus large bounding boxes. A small error in a large box is generally benign but a small error in a small box has a much greater effect on IOU. Our main source of error is incorrect localizations.”

2.4.2 – SSD

Again in 2016, another very popular object detector was introduced by W. Liu *et. al*, named Single Shot Detector [26], normally referred to as SSD. Like YOLO, the aim of the model was to speed up the not sufficient inference time of the state-of-the-art methods of the time, i.e. Faster R-CNN. The authors eliminated the Region Proposal Network and to recover the drop in accuracy, SSD applies a few improvements including multi-scale features and default boxes. These improvements allow SSD to match the Faster R-CNN’s accuracy using lower resolution images, which further pushes the speed higher. The pipeline consists mainly in 2 blocks: extraction of feature maps at different scales and then application of convolutions for the detection of objects.

SSD uses VGG16 [16] as base feature extractor backbone and extracts feature maps at different scales reducing the resolution at every step, this is called multi-scale feature maps. SSD uses lower resolution layers to detect larger scale objects. For example, the 4×4 feature maps are used for larger scale object. After VGG16, the authors add other 6 convolutional layers of decreasing kernel size to detect object at different scales independently. The whole architecture is shown in Fig. 36.

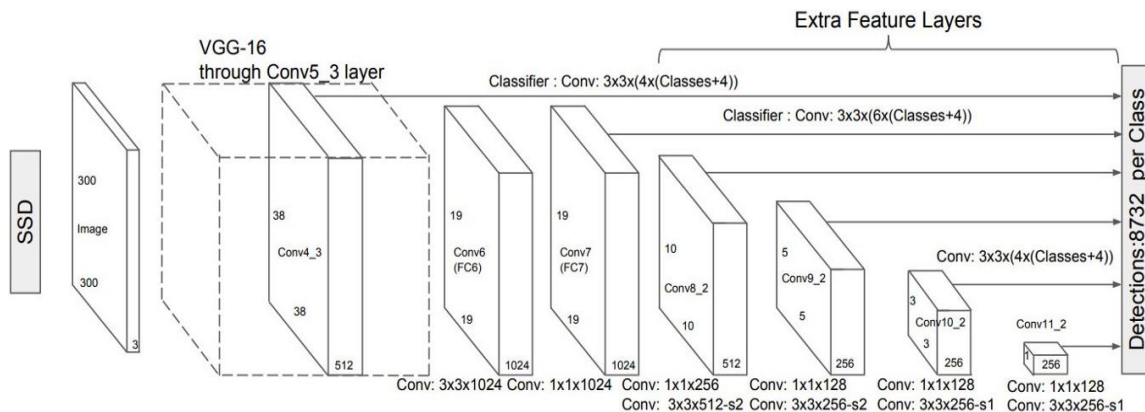


Figure 36. Architecture of SSD

Much like in Faster R-CNN, for each cell, at every different scale, SSD utilizes 4 default boxes to try and fit the eventual objects present. The ratio and scales of these boxes are fine-tuned during the training process, and they learn scale-variant shapes that indeed vary and are optimized to fit objects that are more commonly present at one specific scale. It computes both the location and class scores using small convolution filters. After extracting the feature maps, SSD applies 3×3 convolution filters for each cell to make predictions. Each filter outputs 25 channels: 21 scores for each class plus one boundary box parameters.

Fig. 37 is an example of how SSD combines multi-scale feature maps and default boundary boxes to detect objects at different scales and aspect ratios. The dog below matches one default box (in red) in the 4×4 feature map layer, but not any default boxes in the higher resolution 8×8 feature map. The cat which is smaller is detected only by the 8×8 feature map layer in 2 default boxes (in blue).

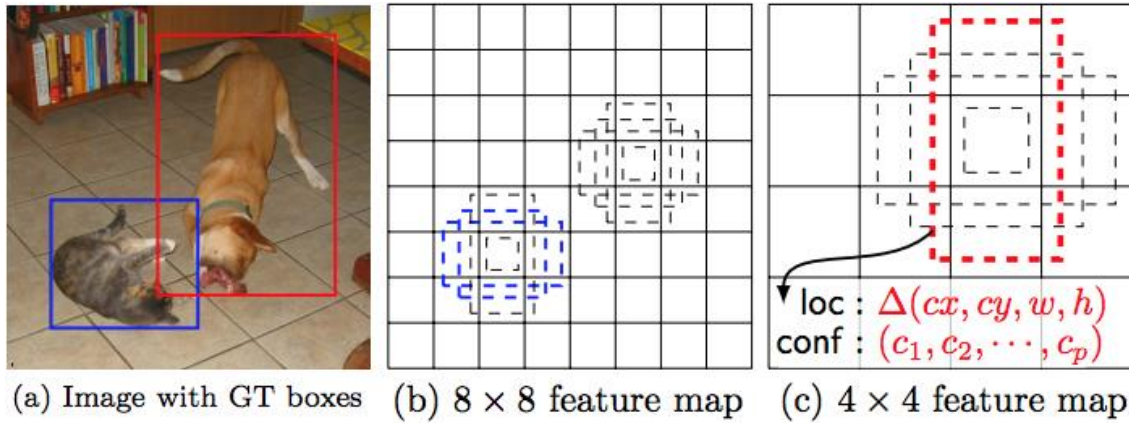


Figure 37. Anchor boxes example

Table 3 shows how SSD compares with other detection methods.

Table 3. Comparison of SSD and other methods

System	VOC2007 test <i>mAP</i>	FPS (Titan X)	Number of Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	~6000	~1000 x 600
YOLO (customized)	63.4	45	98	448 x 448
SSD300* (VGG16)	77.2	46	8732	300 x 300
SSD512* (VGG16)	79.8	19	24564	512 x 512

In conclusion, SSD is faster and achieves even higher accuracy than Faster R-CNN because multi-scale feature maps improve the detection of objects at different scales, and also makes more predictions. The main disadvantage is that it lacks precision when dealing with small objects.

2.4.3 – Normalized cuts graph segmentation

We will now briefly discuss this method, since it is one of the possible methods used by the framework that we will use as our autonomous driving platform, even if it is a much older method and now less used. It is based on the work of Jianbo Shi and Jitendra Malik '*Normalized Cuts and Image Segmentation*' [27]. The basic principle is to build a weighted graph, which is the set of our points connected with some edges. In this kind of graph, each edge is coupled with a weight that represents the degree of similarity between the 2 point it connects. The similarity is simply calculated in terms of intensity, texture, color (not in our case since LiDARs do not provide this information). Note that not all the points are connected with all the others. Fig. 38 provides an example of a weighted graph.

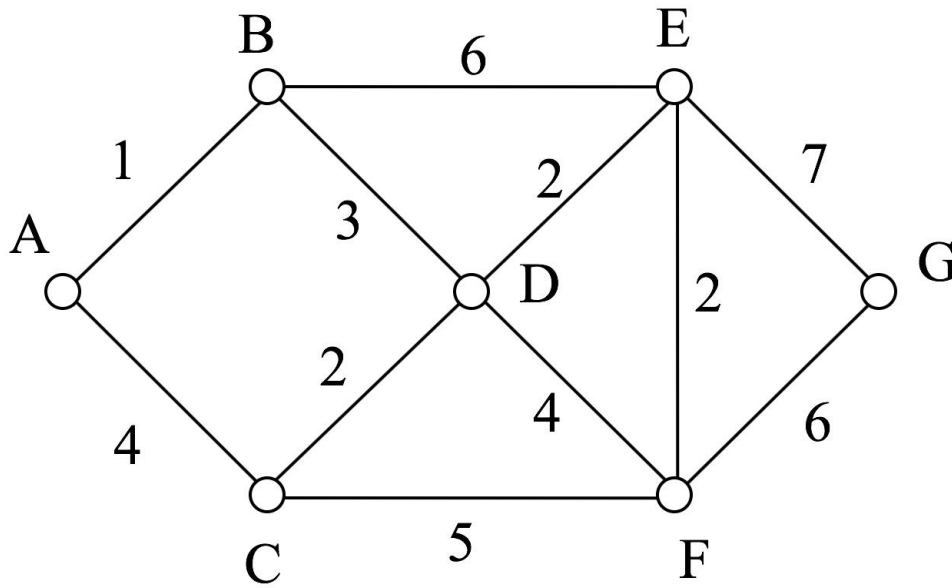


Figure 38. Weighted graph

To segment the graph in 2 regions A and B, the idea is to apply a graph cut by minimizing the sum of the weights that have to be cut to achieve such a partitioning. This method is called minimum cut. In formula:

$$\min cut(A, B) \quad (7)$$

$$where: \quad cut(A, B) = \sum_{p \in A, q \in B} w(p, q)$$

Where $w(p, q)$ denotes the weight associated between nodes p and q.

Unfortunately, this method often leads to cutting isolated nodes in the graph due to the small values achieved by partitioning such nodes, since they normally have less connected edges than center nodes. Fig. 39 explains this idea.

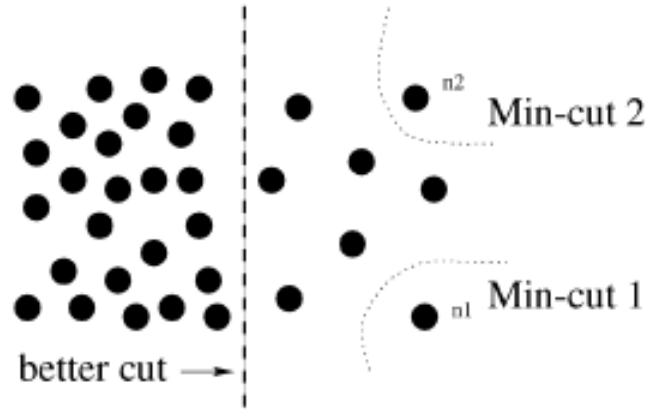


Figure 39. Bad partitioning of isolated nodes

To prevent this, Jianbo Shi and Jitendra Malik proposed to, instead of looking at the value of total edge weight connecting the two partitions, compute the cut cost as a fraction of the total edge connections to all the nodes in the graph. They call this disassociation measure the normalized cut (Ncut):

$$Ncut = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)} \quad (8)$$

Where $assoc(A,V) = \sum_{p \in A, t \in V} w(u,t)$, and V denotes the entire set of nodes. $assoc(A,V)$ is defined in the same way.

This problem can be written in matrix form and is then solvable with an eigenvalue problem. Clearly, in the case of partitioning of a road-scenario, there is the need to segment many different objects. For this purpose, this algorithm is simply re-iterated until a certain threshold is reached.

The algorithm is then coupled with a detection head, that carry out the tasks of classification and bounding box regression. Unfortunately, the framework Apollo does not provide an explanation of the architecture used as the detection head as it is already a default implemented method, but we might suppose it could be a VGG16 network or similar.

2.5 – 3D detection methods

In this shorter section we will give an overview of 2 common methods for the detection starting from raw point clouds returned from LiDAR sensors. Let's remember that a raw point cloud is an unordered set of points that contain the 3 cartesian coordinates and the intensity value. In general, these methods adopt the same techniques of the algorithms

explained in sections 2.2 and 2.3, where the main difference is that we now need to give the point cloud some kind of order, in order to relate relative special positions. Then, all convolution layers will now be carried out with 3D kernels instead of 2D.

2.5.1 – PointNet

The main problem with point cloud deep learning is that typical convolutional architecture requires highly regular input data format, like image or temporal features. As point clouds are not in regular format, the common approaches are to transform the data to regular 3D voxel grid or projections, as seen in section 1.5

Pointnet [28] by C. Ruizhongtai *et. al.* was the initial approach for novel type of neural network that directly consumes unordered point clouds without the need of discretization, which also takes care of the permutation invariance of points in the point cloud. Pointnet can do object classification and also part segmentation. The main feature of Pointnet is the network is robust with respect to input transformations like translation and rotation, and input permutation.

The overall architecture, shown in Fig. 40, is composed by a shared multi-layer perceptron (MLP) to map each of the n points from three dimensions to 64 dimensions. It's important to note that a single multi-layer perceptron is shared for each of the n points (i.e., mapping is identical and independent on the n points). This procedure is repeated to map the n points from 64 dimensions to 1024 dimensions. With the points in a higher-dimensional embedding space, max pooling is used to create a global feature vector in \mathbb{R}^{1024} . Finally, a three-layer fully-connected network is used to map the global feature vector to k output classification scores.

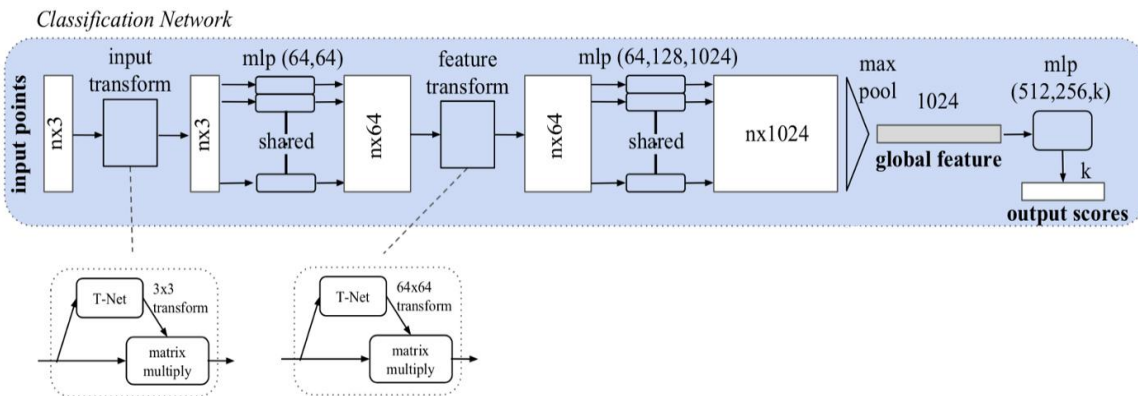


Figure 40. Architecture of PointNet

To ensure transformation invariance, 2 sub-nets (called T-Net in Fig. 33) are developed to ensure a pose normalization, which means that the network must then be able to recognize an object regardless of eventual rotations/translations applied to it. Based on the input, a regression network obtains the transformation matrix applied to the specific

case, then a grid is constructed over the desired object and a simple matrix multiplication is carried out to obtain pose normalization. The process is repeated a second time after the MLP to ensure also the features extracted from the point cloud follow the same requirements.

To ensure permutability of the N input points, the solution is to apply a symmetric function to the point cloud. The authors opted for the max pooling function, as it gave the best results among others like average and sum. The result is a global feature vector that aims to capture an aggregate signature of the n input points, and is then directly used for classification.

Table 4 compares the results of detection of PointNet with other 3D object detectors. It is worth mentioning that the network performs very well also on segmentation task.

Table 4. Comparison of PointNet and other networks

	input	#views	accuracy avg. class	accuracy overall
SPH [11]	mesh	-	68.2	-
3DShapeNets [28]	volume	1	77.3	84.7
VoxNet [17]	volume	12	83.0	85.9
Subvolume [18]	volume	20	86.0	89.2
LFD [28]	image	10	75.5	-
MVCNN [23]	image	80	90.1	-
Ours baseline	point	-	72.6	77.4
Ours PointNet	point	1	86.2	89.2

2.5.2 – PointPillars

In 2018, Gregory P. Meyer *et. al.* developed a network called PointPillars [29] that is still one of the fastest detection algorithms with great accuracy on autonomous driving datasets. PointPillars runs at 62 fps, which is orders of magnitude faster than the previous works in this area.

The architecture, presented in Fig. 41, is constituted in 3 parts: the Pillar Feature Net, the Backbone for feature extraction, and the Detection Head, which is basically the SSD algorithm seen in section 2.3.2.

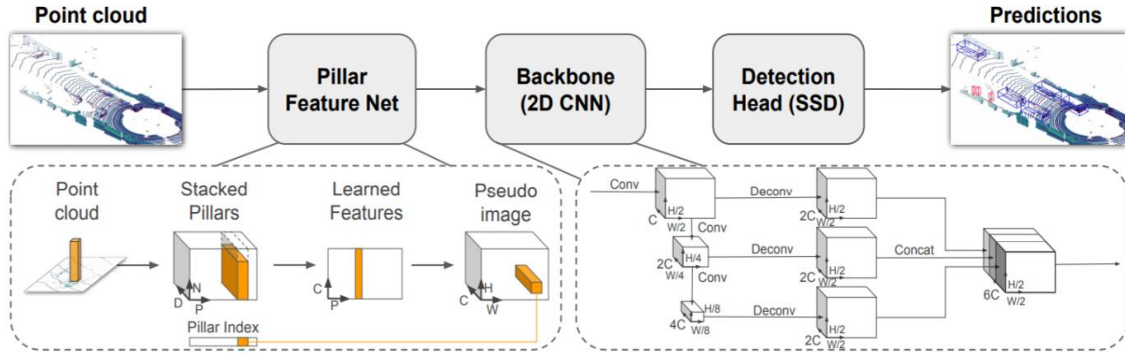


Figure 41. Architecture of PointPillars

The first part, the Pillar Feature Net, converts the point cloud into a sparse pseudo image. First, the point cloud is divided into grids in the x-y coordinates, creating a set of pillars along the z dimension, like shown in Fig. 42. Each point in the cloud, which is a 4-dimensional vector (x,y,z, intensity), is converted to a 9-dimensional vector containing the additional information explained as follows:

- X_c, Y_c, Z_c : distance from the arithmetic mean of the pillar c the point belongs to in each dimension.
- X_p, Y_p : distance of the point from the center of the pillar in the x-y coordinate system.

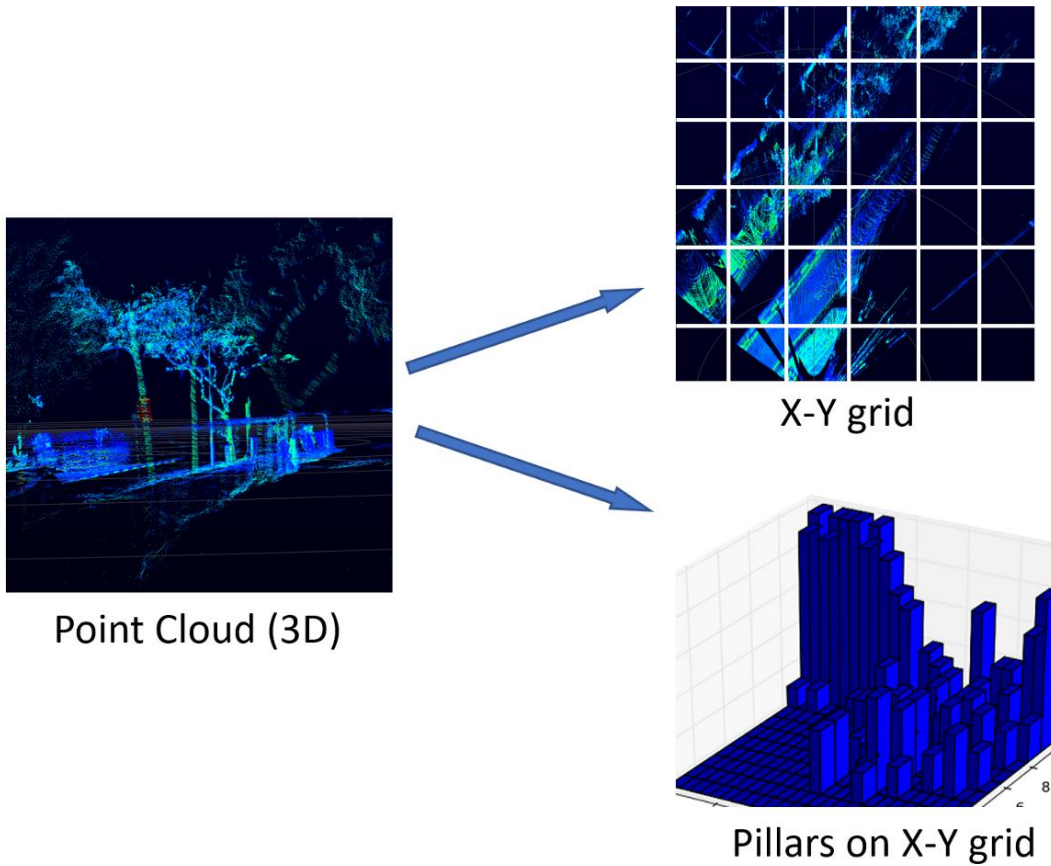


Figure 42. Pillars representation

The set of pillars will be mostly empty due to sparsity of the point cloud, and the non-empty pillars will in general have few points in them. This sparsity is exploited by imposing a limit both on the number of non-empty pillars per sample (P) and on the number of points per pillar (N) to create a dense tensor of ‘stacked pillars’ of size (D, P, N). Note that $D = [x, y, z, r, X_c, Y_c, Z_c, X_p, Y_p]$.

Now, from the above-mentioned tensor features have to be extracted. As it is made of a 3D point cloud, the authors decide to use PointNet to extract such features, that will result in a 2D matrix. From this 2D matrix, a pseudo-image is re-created using the pillar index for each point. So originally, where the point was converted to a D dimensional vector, now it contains a C dimensional vector, which are the features obtained from a PointNet. Fig 36 is an enlargement of Fig. 43 and shows these passages.

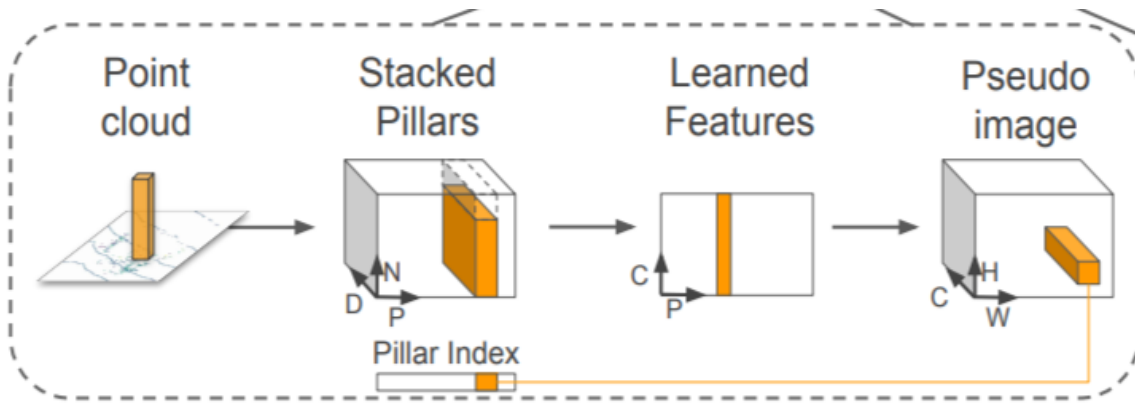


Figure 43. Pillar Feature Net detail

The next part is a network that is used to isolate possible object candidates much like a Region Proposal Network. It is composed of 3 fully convolutional 3D layers, each followed by a batch normalization and ReLU function. Then upsample the output of every block to a fixed size and concatenated to construct the high-resolution feature map.

The last part is the detection head, which uses the network of SSD with the aim of detecting and drawing bounding boxes around the proposed detections.

To understand how well PointPillars performs, we shall have a look at Table 5, which compares this method with other detectors in a 3D detection task from the KITTI dataset.

Table 5. Comparison of PointPillars and other networks

Method	Modality	Speed (Hz)	mAP	Car			Pedestrian			Cyclist		
			Mod.	Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard
MV3D [2]	Lidar & Img.	2.8	N/A	71.09	62.35	55.12	N/A	N/A	N/A	N/A	N/A	N/A
Cont-Fuse [15]	Lidar & Img.	16.7	N/A	82.54	66.22	64.04	N/A	N/A	N/A	N/A	N/A	N/A
Roarnet [25]	Lidar & Img.	10	N/A	83.71	73.04	59.16	N/A	N/A	N/A	N/A	N/A	N/A
AVOD-FPN [11]	Lidar & Img.	10	55.62	81.94	71.88	66.38	50.80	42.81	40.88	64.00	52.18	46.61
F-PointNet [21]	Lidar & Img.	5.9	57.35	81.20	70.39	62.19	51.21	44.89	40.23	71.96	56.77	50.39
VoxelNet [31]	Lidar	4.4	49.05	77.47	65.11	57.73	39.48	33.69	31.5	61.22	48.36	44.37
SECOND [28]	Lidar	20	56.69	83.13	73.66	66.20	51.07	42.56	37.29	70.51	53.85	46.90
PointPillars	Lidar	62	59.20	79.05	74.99	68.30	52.08	43.53	41.49	75.78	59.07	52.92

Chapter 3

LG SVL Simulator and Apollo

In this chapter we will briefly describe the 2 softwares which have been the basis for this work. The first one, LG SVL Simulator, is a driving simulator, while the second one, Apollo (developed by Baidu), is the framework that connects to the driving simulator to make the digital vehicle fully autonomous.

3.1 – LG SVL Simulator

SVL Simulator is a simulation platform used autonomous driving and robotics applications. It has been developed by LG Electronics America R&D Lab, a research center in Santa Clara, California. The first version was an open project uploaded in 2018 on GitHub² with the goal of enabling developers to build autonomous vehicles and robots through end-to-end, high performance 3D simulation. Since its first release, SVL Simulator has now turned into a commercial product helping automotive manufacturers, robotics companies, and universities around the world.

An outstanding work describing the simulator is [30] by Guodong Rong *et. al.*.

*“SVL Simulator consists of the simulation software, software tools, the ecosystem of content and plugins that enable tailored use cases, and the cloud environment which enables simulation and scenario testing at scale. By simulating a virtual environment, one or more ego vehicles or autonomous systems and their sensors, and traffic and other dynamic objects, the simulation software provides a seamless and customizable interface with a user's System Under Test. This allows the developer to debug, perform modular testing, and perform integration testing.”*³

⁽²⁾ <https://github.com/>

⁽³⁾ <https://www.svlsimulator.com/docs/getting-started/introduction/>

As already mentioned, SVL is used for several specific applications, among which:

- L4/L5 autonomous vehicle systems
- L2/L3 ADAS/AD systems
- Warehouse robotics
- Outdoor mobile robotics
- Future Mobility services
- Autonomous racing
- Sensor/sensor systems development and marketing
- Automotive and autonomous system security
- Synthetic data generation
- Real-time embedded systems for automotive

Specific uses of the simulator include:

- L4/L5 autonomous vehicle systems
- L2/L3 ADAS/AD systems
- Warehouse robotics
- Outdoor mobile robotics
- Future Mobility services
- Autonomous racing
- Sensor/sensor systems development and marketing
- Automotive and autonomous system security
- Synthetic data generation
- Real-time embedded systems for automotive

For the purpose of this paper, the environment of SVL Simulator offers real-time, high-performance and realistic simulations, with variety of scenarios, vehicles and road users to be used. It also enables to import and modify HD maps as well as customize all sensors required for autonomous driving.

The simulator is to be coupled with an autonomous driving framework, in our case Apollo, to which it communicates all the data and informations coming from the sensors and the surrounding environment, as to make it possible to carry out all the tasks of an autonomous vehicle. That is, running the localization, perception, prediction, planning and control modules. The connection between the two is made possible by a so-called bridge; in our case, CyberRT will be used.

Fig. 44 shows the architecture and tasks of the simulator coupled with a generic autonomous driving platform. Figures 45 shows an example of simulation environment, while Fig. 46 displays some sensors views and features of the simulator.

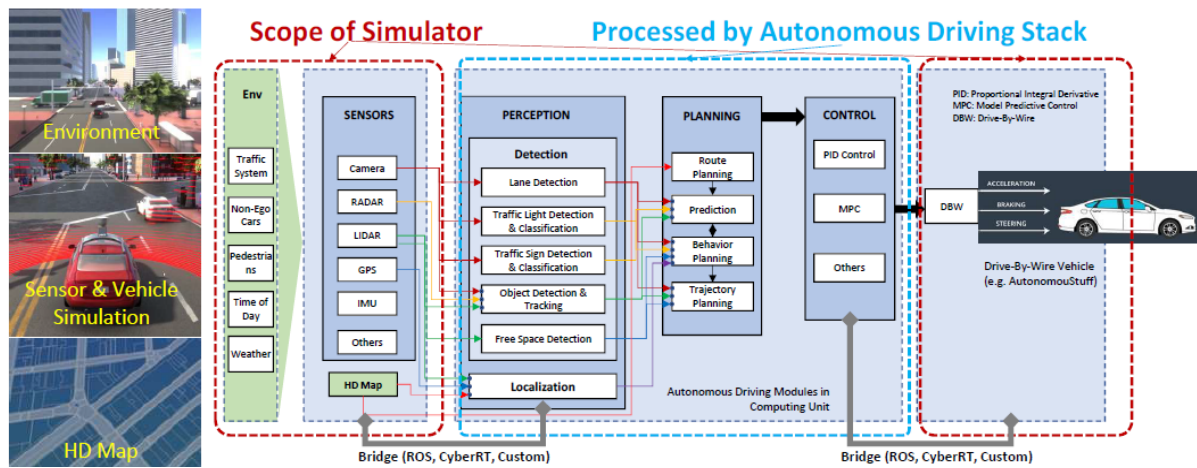


Figure 44. Pipeline of SVL Simulator coupled with an AV framework



Figure 45. Borregas Avenue, one of the standard maps of SVL

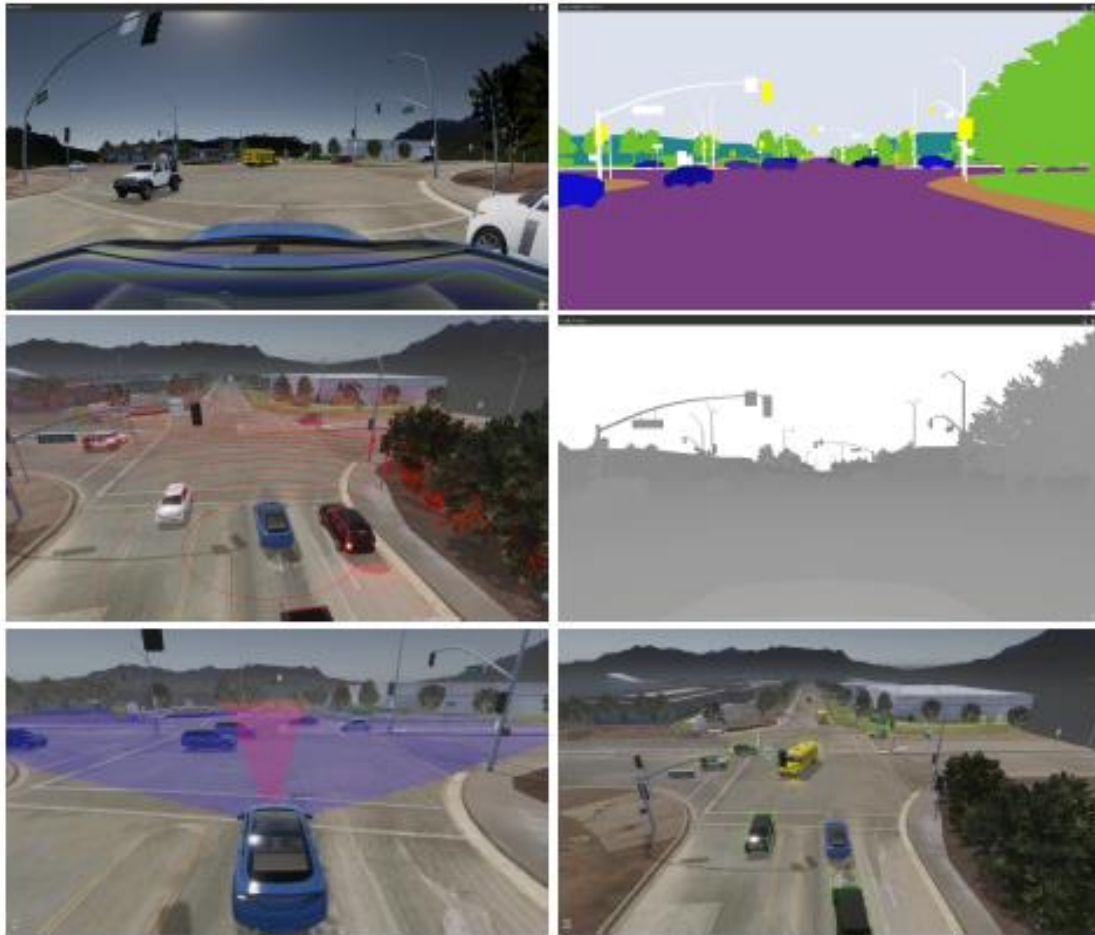


Figure 46. Different types of sensors. Left (top to bottom): Fish-eye camera, LiDAR, Radar; Right (top to bottom): Segmentation, Depth, 3D Bounding Box.

It is worth mentioning what an HD map is, given their importance in AD. HD maps are roadmaps with centimeter-perfect accuracy and high environmental fidelity – they contain information about the exact positions of pedestrian crossings, traffic lights and signs, barriers and more. This is necessary for autonomous vehicles because they cannot compensate for map inaccuracies the way humans do when following a GPS: if a map is a meter or two off, a human driver is not going to crash because of it – we simply understand what the map refers to in the scene we see through the windshield. They as well offer semantic data about the road network. Nowadays High-Definition maps are widely used in the autonomous driving industry and are now considered essential in ensuring safety on the road.

HD maps creation requires great effort and resources but due to their importance, all the companies involved in this field are investing their time to optimize the process of map generation seeking to lower the cost.

In SVL, there exist a tool that allows to make annotation on HD maps as to label an object or modify pre-existent labels. Fig. 47 shows an example of HD on SVL.

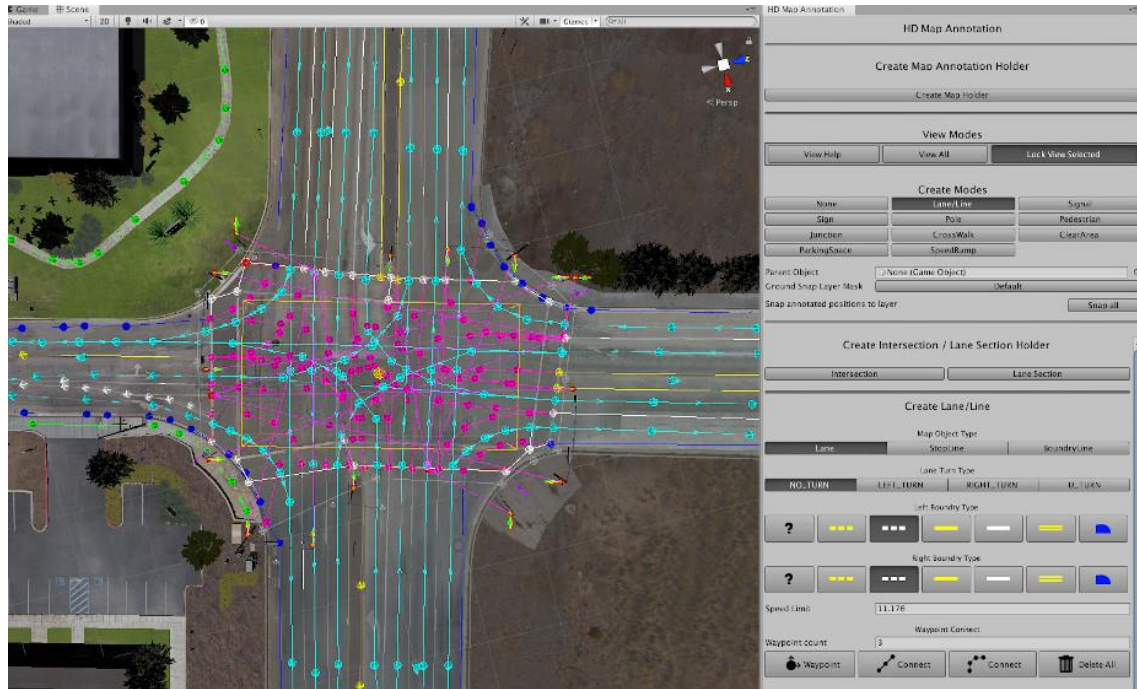


Figure 47. HD Map example and annotation tool in LGSVL Simulator.

3.2 – Apollo

Baidu's Apollo is an open source self-driving vehicle tech platform. It provides a complete hardware and software service solution that includes cloud data services, software, vehicle and hardware platform. Apollo offers open source code and capabilities in obstacle perception, trajectory planning, vehicle control, vehicle operating systems, as well as a complete set of testing tools and other functions. All Apollo's codes, uploads and data are uploaded to its GitHub page [31].

Apollo has several modules and features for autonomous driving but needs to be configured and calibrated perfectly before trying them for a test. For what regards this specific paper, the calibration steps had already been executed before start of the collaboration, so we will consider them as given.

The Apollo software is based on the following modules, that all together manage the framework:

- Localization: localize the vehicle inside the HD map
- Perception: identify and localize obstacles and road users
- Prediction: predict the trajectory of road user for possible crashing trajectories
- Planning: plan the best route for reaching the selected destination avoiding collisions
- Control: act on steering, brakes and throttle to achieve planning route

Fig. 48 shows the pipeline and logics of interactions between modules. Additionally, it is worth mentioning the traffic light module, that can locate traffic lights and understand their color in real time.

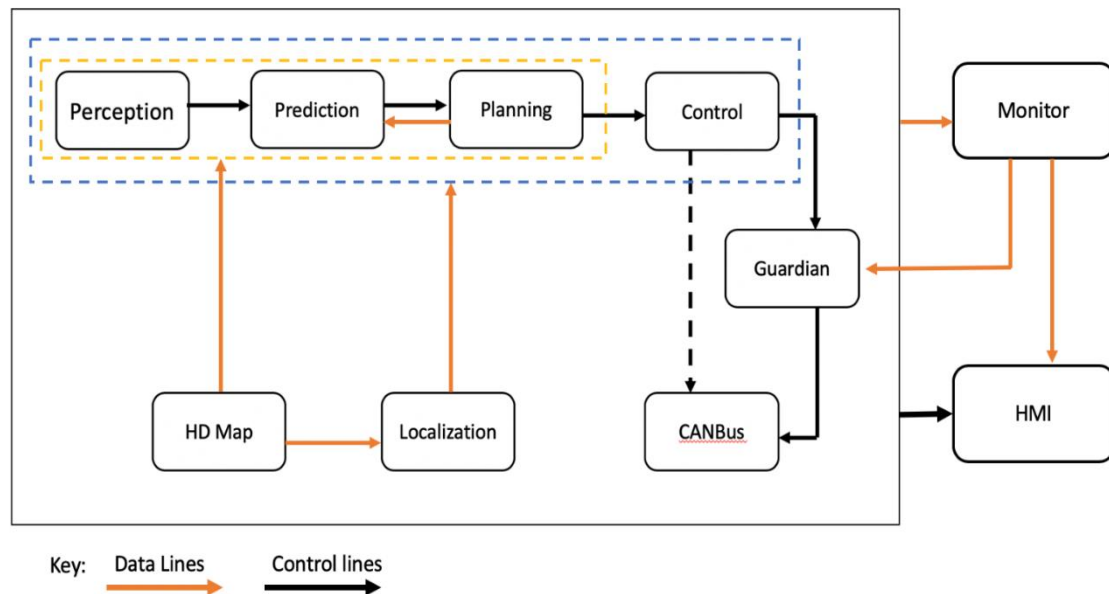


Figure 48. Pipeline of modules interaction

Through the years, Apollo has developed several versions of its software, with increasing capabilities for autonomous driving to remain up to date with the state-of-the-art new technologies that advanced every year. Fig. 49 shows the various versions just mentioned.



Figure 49. Versions of Apollo

The team has opted to use the latest version, 6.0, for these reasons:

- With previous versions, the module off traffic light detection did not work, resulting obviously in severe mistakes
- Same happened with the perception module, resulting in not recognizing obstacles and road users in the scenario

- Some GPU cards were not supported, like Titan V, GTX 16xx, RTX

Let's now see a bit more details about the latest version, Apollo 6.0, because it is the one that we will use for our simulations.

Apollo's GitHub page cites: *"Apollo 6.0 incorporates new deep learning models to enhance the capabilities for certain Apollo modules. This version works seamlessly with new additions of data pipeline services to better serve Apollo developers. Apollo 6.0 is also the first version to integrate certain features as a demonstration of our continuous exploration and experimentation efforts towards driverless technology."*

The new version also integrates emergency vehicle detection through audio devices like microphone. This feature is though not object of interest for this paper.

Once installed, calibrated and built, the Apollo user interface, called Dreamview, is an interactive environment with different panels, windows and buttons to control the entire process. The next figures show examples of the most important features.

Fig. 50 shows the task tab. Highlighted in red with number 1 is the header, that contains the selection of the vehicle, map, and mode in which we would like the simulation to be carried out. The red box denoted as 2 enables to switch between the different tasks for controlling the autonomous drive. The red box denoted as 3 is the panel of the task tab. In here we can perform some of the tasks supported by the selected mode. For example, in the 'Quick Start' menu we can turn on all modules with the 'Setup' button, or inversely turn them all off with 'Reset'. In the 'Others' menu we can find some buttons to activate frequently used features. In 'Module Delay' we can find the delay, in seconds between two messages of the same module. Last, the 'Console' menu shows any message or error from the simulation.

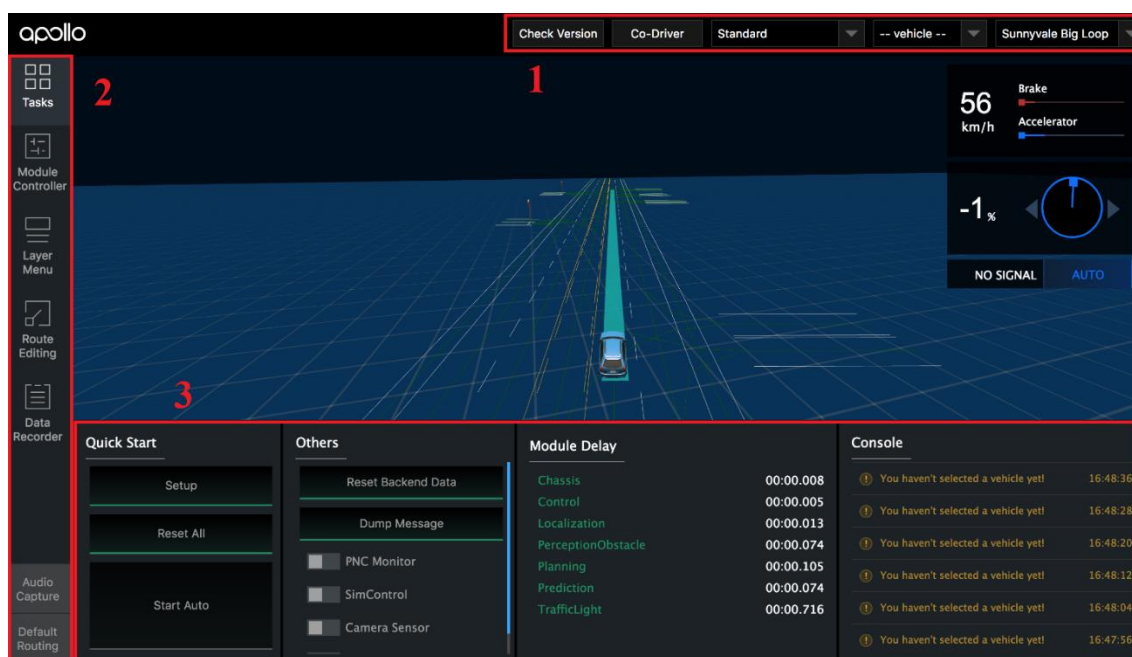


Figure 50. Task tab

Fig. 51 shows the module controller tab. This tab is very important since it's here that we can individually activate all modules. Note that, as best practice, it is recommended to activate modules in a precise sequence, which is: transform, localization, perception, prediction, (traffic light), planning, control. 'Transform' is a module that makes a transformation between the coordinate system used in the driving simulator (SVL) and the one adopted Apollo, and it's hence indispensable to activate.

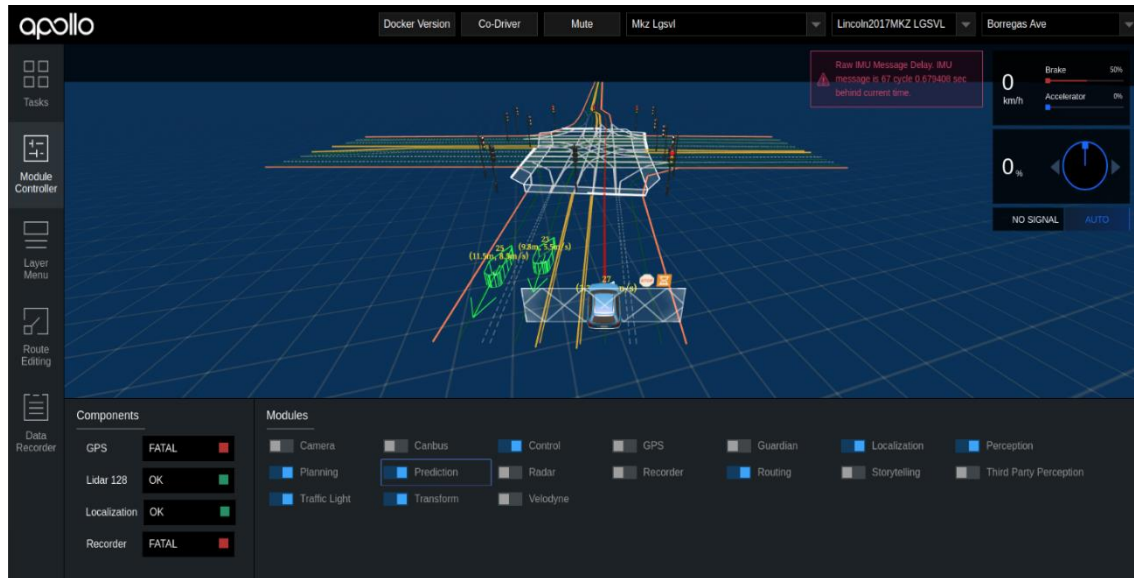


Figure 51. Module controller tab

Fig. 52 displays the route editing tab, in which we can see the map and position of the vehicle, and set the point(s) that we want the vehicle to reach. The buttons to add points of interest, cancel them and finally send the routing request are highlighted in the red box. The two red points seen in the figure are indeed points of interest set by the user, while the violet polygons are bounding boxes that encapsule other vehicles of the road.



Figure 52. Route editing tab

3.3 - Perception module of Apollo

We shall now focus on the module which is the main topic of this paper: the detection module. As already mentioned, this is the part of the software in charge of locating and categorizing in real time the different obstacles and road users that fall inside the vehicle's field of view. This module needs to be effective in all weather and road conditions, as it is arguably the most important, yet more challenging, part of an autonomous driving software.

The input of this module are:

- LiDARs point clouds
- Radar data
- Cameras data
- Extrinsic parameters of radars calibration
- Intrinsic and extrinsic parameters of cameras calibration
- Velocity and angular velocity of the vehicle, given by IMU sensor

The outputs of the module are:

- The 3D obstacle tracks with the heading, velocity and classification information
- The output of traffic light detection and recognition (not part of this work)

Fig. 53 shows the architecture of the module.

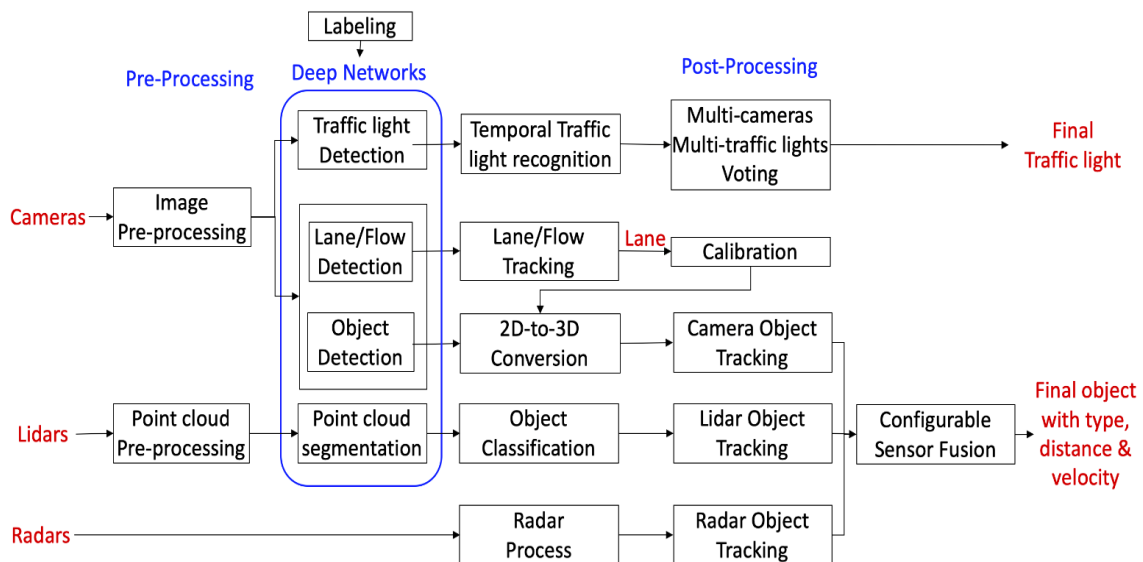


Figure 53. Perception module architecture

For what concerns the goal of this paper, we will not analyze the branches of the architecture that concern radars and cameras, while we will focus solely on the LiDAR part of the detection module.

The novelty of Apollo 6.0 release is the inclusion of PointPillars [31] in the possible default method of object recognitions that the user can choose from. Apollo also allows the use of 2 other previously developed detection algorithms: NCutSegmentation and MaskPillars. However, by default, Apollo has developed its own CNN architecture for object recognition. Let us briefly see its main key points.

HDMaP Filter

The first step of Apollo's method is constituted by the HDMaP Region of Interest filter: the Region of Interest (ROI) specifies the drivable area that includes road surfaces and junctions that are retrieved from the HD map. The HDMaP ROI filter processes LiDAR points that are outside the ROI, removing background objects, e.g., buildings and trees around the road. What remains is the point cloud in the ROI for subsequent processing.

Given an HDMaP, the affiliation of each LiDAR point indicates whether it is inside or outside the ROI. Each LiDAR point can be queried with a lookup table (LUT) of 2D quantization of the region around the car. The input and output of the HDMaP ROI filter module are summarized in the Table 6.

Table 6. Input and output of HDMaP ROI filter

Input	Output
The point cloud: A set of 3D points captured from LiDAR Sensor.	The indices of input points that are inside the ROI defined by HDMaP.
HDMaP: A set of polygons, each of which is an ordered set of points.	

The Apollo HDMaP ROI filter generally consists of three successive steps:

- Coordinate transformation
- ROI LUT construction
- Point inquiry with ROI LUT

In the coordinate transformation step, Apollo has to change the coordinates to a common and local one, centered in the LiDAR location. This is because the HD map and point cloud initially do not share the same coordinate systems.

In the lookup table construction step, Apollo builds a grid-like LUT that maps the ROI into a birds-eye view, as shown in Fig. 54. As shown in the figure, this LUT covers a rectangle region, bounded by a predefined spatial range around the general view from above of the HDMap. Then it represents the affiliation with the ROI for each cell (i.e., 1/0 represents it is inside/outside the ROI). The blue lines indicate the boundary of HDMap ROI, including road surfaces and junctions. The red solid dot represents the origin of the local coordinate system corresponding to the LiDAR sensor location.

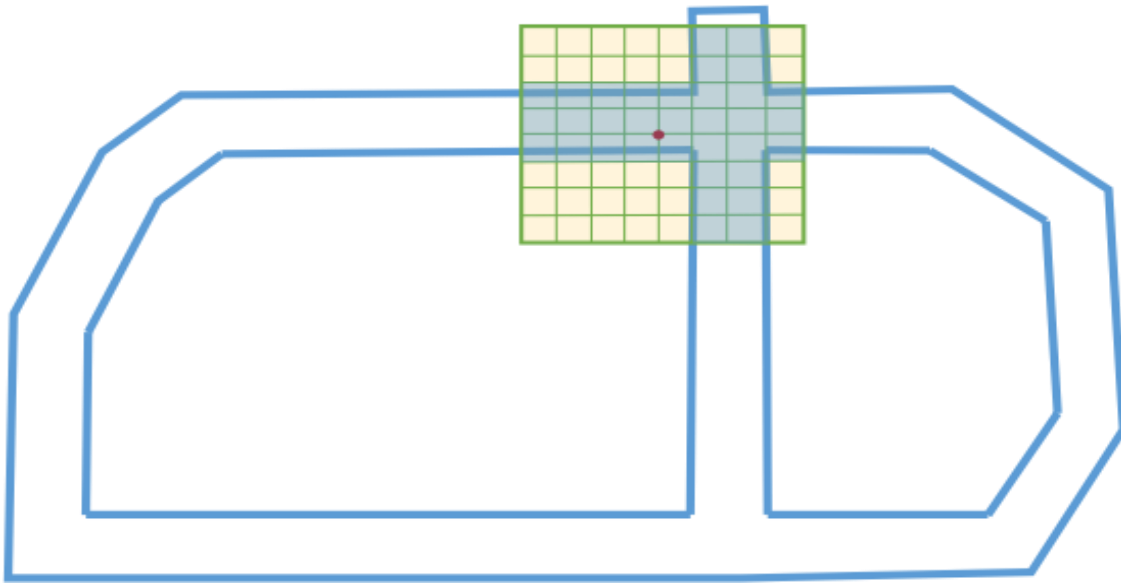


Figure 54. ROI and LUT

In the last step, Apollo performs 3 simple tasks to identify if the selected point belongs to the ROI or not:

- Identifies whether the point is inside or outside the rectangle region of ROI LUT
- Queries the corresponding cell of the point in the LUT for its affiliation with respect to the ROI
- Collects all the points that belong to the ROI and outputs their indices with respect to the input point cloud

CNN Segmentation

At this point we have obtained a filtered point cloud containing only the points belonging to the Region of Interest. Most of the background obstacles, such as buildings and trees around the road region, have been removed, and the point cloud inside the ROI is fed into

the detection module (that Apollo calls segmentation, even though it has all the requirements to be categorized as object detection). Apollo uses a deep CNN for accurate obstacle detection and segmentation. It consists of four successive steps:

- Channel Feature Extraction
- CNN-Based Obstacle Prediction
- Obstacle Clustering
- Post-processing

Given a point cloud frame, Apollo builds a birds-eye view (i.e., projected to the X-Y plane) that is a 2D grid in the local coordinate system. Each point within a predefined range with respect to the origin (i.e., the LiDAR sensor) is quantized into one cell of the 2D grid based on its X and Y coordinates. After quantization, Apollo computes 8 statistical measurements of the points for each cell of the grid, which will be the input channel features fed into the CNN in the subsequent step. The statistical measurements computed are:

- Maximum height of points in the cell.
- Intensity of the highest point in the cell.
- Mean height of points in the cell.
- Mean intensity of points in the cell.
- Number of points in the cell.
- Angle of the cell's center with respect to the origin.
- Distance between the cell's center and the origin.
- Binary value indicating whether the cell is empty or occupied.

Based on the channel features described above, Apollo uses a deep fully-convolutional neural network (FCNN) to predict the cell-wise obstacle attributes including the offset displacement with respect to the potential object center — called center offset, objectness, positiveness, and object height. The input of the network is a $W \times H \times C$ channel image, where W represents the column number of the grid, H that of the row, C represents the number of channel features. The FCNN is composed by 3 parts (as shown in Fig. 55):

- Downstream encoding layers (feature encoder)
- Upstream decoding layers (feature decoder)
- Obstacle attribute prediction layers (predictor)

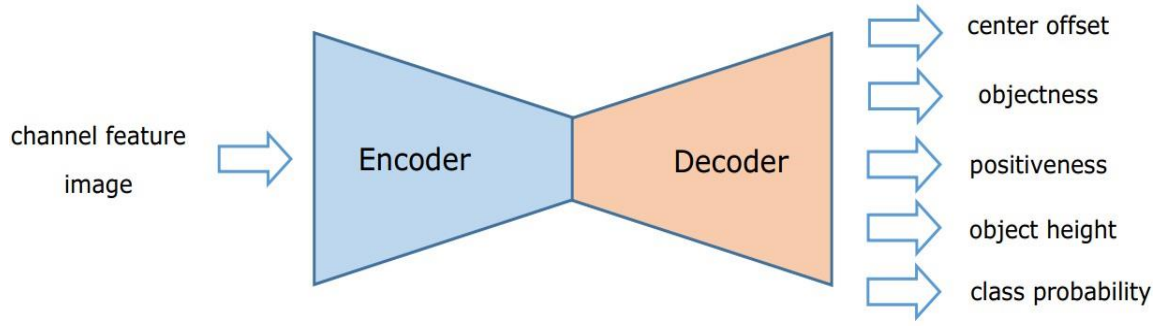


Figure 55. Scheme of Apollo's FCNN for cell-wise obstacle prediction

Unfortunately, Apollo does not provide details on how many layers it adopts, or which kind of activation functions it uses: indeed, being a pre-trained network, it is not meant to be modified by the users. After the CNN step, Apollo uses the 5 outputs of the previous step to propose obstacle clustering. The CNN extracts informations about singular cells, while in this step the goal is to use the extracted data of all neighboring cells in search of the entire object.

To generate obstacle objects, Apollo constructs a directed graph, based on the cell center offset prediction, and searches the connected components as candidate object clusters. As shown in Fig. 56, each cell is a node of the graph and the directed edge is built based on the center offset prediction of the cell, which points to its parent node corresponding to another cell.

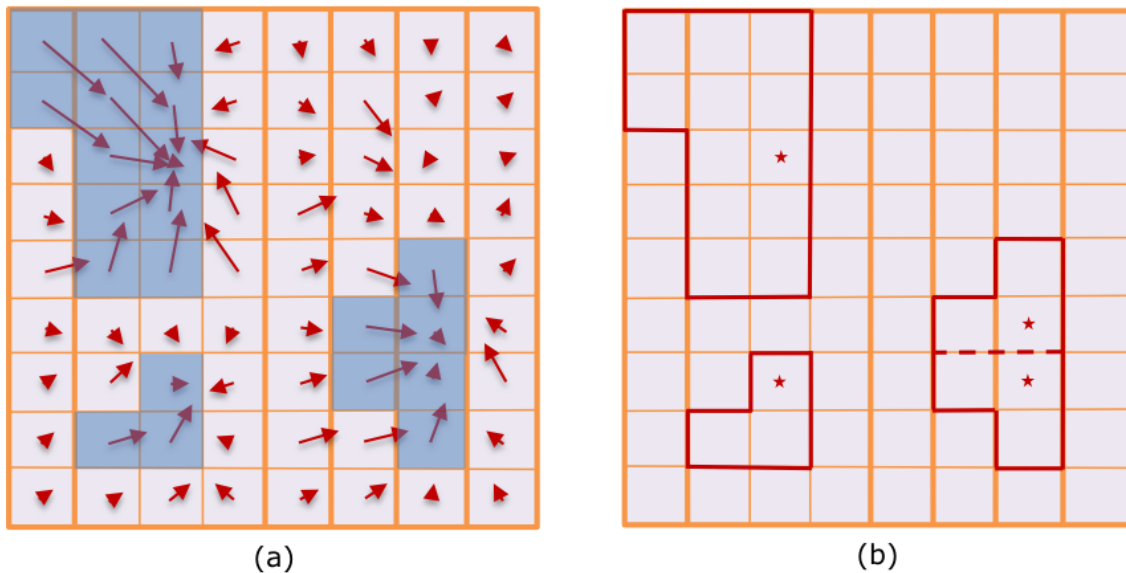


Figure 56. Obstacle clustering

In Fig. 56:

- The red arrow represents the object center offset prediction for each cell
- The blue mask corresponds to the object cells for which the objectness probability is no less than 0.5
- The cells within the solid red polygon compose a candidate object cluster
- The red filled five-pointed stars indicate the root nodes (cells) of sub-graphs that correspond to the connected components

Given this graph, Apollo adopts a compressed Union Find algorithm to efficiently find the connected components, each of which is a candidate obstacle object cluster. The objectness is the probability of being a valid object for one individual cell. Consequently, Apollo defines the non-object cells as the ones with the objectness of less than 0.5.

The class probabilities are summed up over the cells within the object cluster for each candidate obstacle class, including an 'unknown' class. The obstacle class corresponding to the maximum-averaged probability is the final classification result of the object cluster.

After this process, Apollo introduces a post-processing step in which it filters out object clusters with low positiveness score and removes points that are too far from the calculated average height of a specific object cluster.

MinBox Builder and Object Tracker

After these steps, the algorithm proposes a method to draw the box that best encapsules the object minimizing its area. It is based on reconstructing a polygon (with 6 sides) given an initial edge. Then, an object tracking algorithm follows, and predicts, the next movements. In general, it forms and updates track lists by associating current detections with existing track lists, deletes the old track lists if they no longer persist, and spawns new track lists if new detections are identified. The motion state of the updated track lists are estimated after association. In the HM object tracker, the Hungarian algorithm is used for detection-to-track association, and a Robust Kalman Filter is adopted for motion estimation.

Chapter 4

Set up of the virtual environment

In this chapter we will finally describe how we set up the virtual environment for our own simulation. That is, uploading onto SVL Simulator a digitalized version of the vehicle, its sensors, and our own map.

4.1 – Car model and sensors

To set up the virtual environment, there is the need of a procedure to create a digital modeled version of all the components needed for the simulation: i.e. the vehicle, sensors, and the map we wish to make it drive on.

We shall start with the vehicle. *Teoresi* bought a XEV YOYO, shown in Fig. 57 and 58, which is the car has been equipped with suitable sensors to support AV.



Figure 57. *Teoresi's* YOYO, side view



Figure 58. Teoresi's YOYO, back view

The configuration of the car differs from the standard configuration advised from Apollo. For Apollo 6, the website indicates as required sensors the following:

- 2 radars (front and back)
- Top 128 channels mechanical LiDAR
- Front and back 16 channels solid-state LiDAR
- Side cameras
- Front cameras

Instead, for simplicity and economic reasons, the car we will be working on is equipped with two 16 channels mechanical LiDAR *Velodyne16*, one solid-state LiDARs *Robosense MI*, and three cameras that point forward.

Fig. 59 shows the detail of such configuration. The red boxes denoted as 1 and 3 indicate the mechanical LiDARs, the one denoted as number 2 highlights the solid-state LiDAR, while boxes 4, 5 and 6 indicate the 3 forward-facing cameras.



Figure 59. Detail of LiDARs and cameras

To upload a digitalized version of the car we asked XEV, the manufacturer of the vehicle, to provide a *.stl* file (one of the standard files of 3D design) of their YOYO model. Once received, the file is to be open in a 3D modelling software, such as Unity. In Unity, the first steps to carry out are editing the orientation of the axis and creation of meshes for body, wheels and lights separately. Once done, the whole mesh file is exported as a *.fbx* file.

Now, the SVL Simulator needs to be opened from the Unity Editor, and a new scene created. From this point of, the creation of the vehicle follows the exacts steps described in SVL's guide that is reported in the bibliography as [32].

An example of the final result in the editor (not using our car, but an example vehicle) is shown in Fig. 60.

Instead, the final result of digitalization of *Teoresi's* XEV is shown in Fig. 61. Note the fidelity with which the representation has been realized, for example in the position of the 3 LiDARs on top.

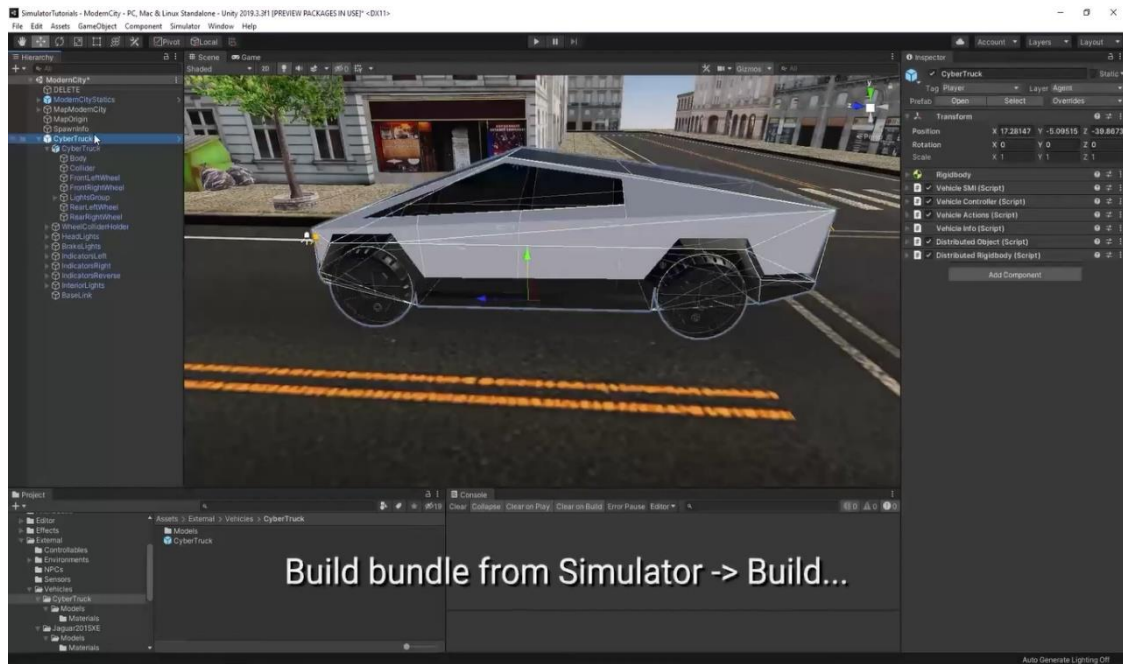


Figure 60. Example vehicle in Unity Editor



Figure 61. XEV digital model

For what regards the creation of the sensors used in our car, the process is easier. The simulator presents already a number of default sensors, including both types of LiDARs, radars, cameras, IMUs among others, and it also allows to customize their technical characteristics (FoV, resolution, orientation, frequency, etc). Since the geometry and weight of our LiDARs do not present any strict constraint and do not interfere with the

functioning of the whole systems, the team has decided to use any of the default sensors, customizing the technical specifications to the exact ones reported on the technical documents of *Velodyne16* and *Robosense M1*.

To do this, we simply go to the vehicle tab, click on our XEV, enter the section of our vehicle, then enter 'Sensor configuration' and then we can edit any of the default types of sensors. Inside this tab we can customize all their technical specifications, as shown in Fig. 62. What, instead, needs to be precise, is their relative location on the chassis. Measurements have been taken on the real car, and replicated with precision on the simulator, as to have the sensor position as similar as possible to the actual, real configuration. Fig. 63 shows an example of sensor configuration. Keep in mind we will use several different sensor configurations throughout the whole study, so Fig. 63 is only for sake of showing one case.

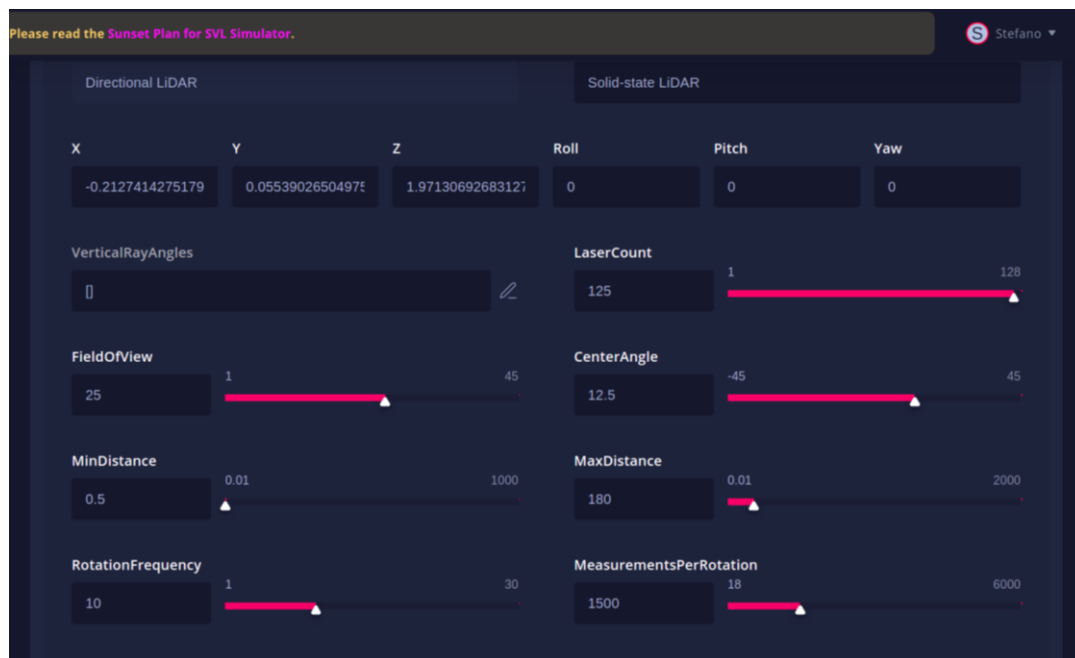


Figure 62. Example of sensor customization

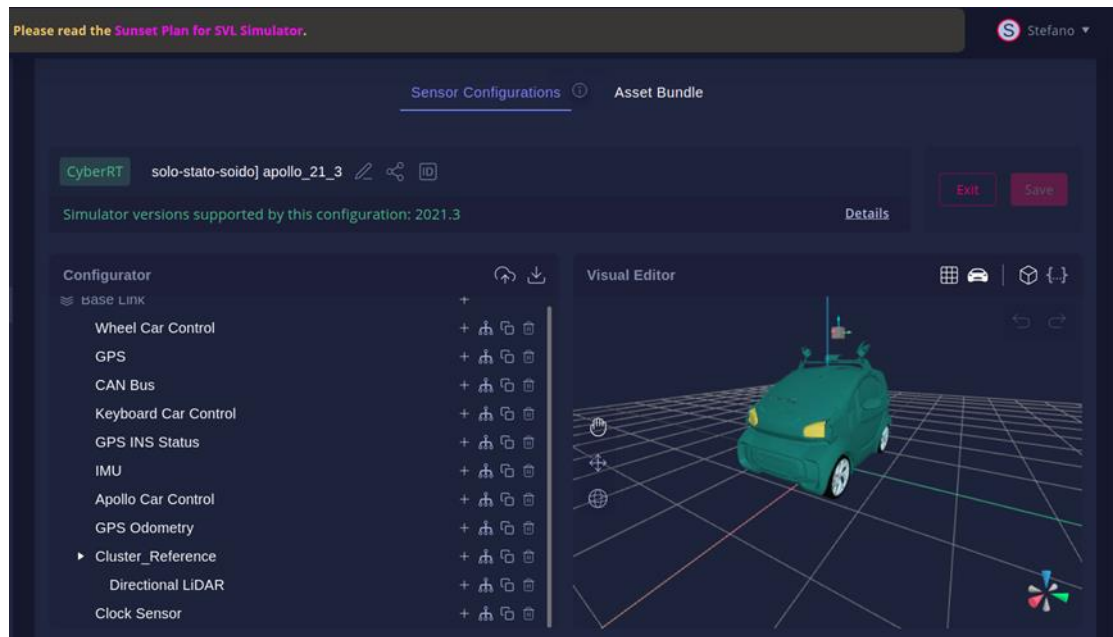


Figure 63. Example of sensor configuration

4.2 – Map

To better simulate a well-known environment of simulation, the team has decided to use as a map of simulation the parking lot of *Teoresi's* facility of Torino. The creation of such map is done with the use of *Blender* (or other 3D modelling softwares) and *Unity Editor* once again. The initial step is the creation of the scenario in *Blender*, utilizing simple geometric *game objects* such as planes, cubes to build the base geometry of the map. Then we followed guide [33], which describes the full process with instructions to transform the simple geometric shapes into buildings, roads, signals, sidewalks, trees, etc. Subsequently, guide [34] contains the steps on how to insert *map annotations*, and upload everything on Apollo in the correct format, so that it is then usable in the future simulations. The map annotations are all the extra semantic informations contained in HD maps that categorize elements. These annotations help the autonomous driving framework take the right decisions knowing high-level information from the map, not only the geometry. These features include:

- dedicated areas where pedestrians must walk
- crossroads
- vertical/horizontal signals
- parking spaces
- double direction roads
- junction

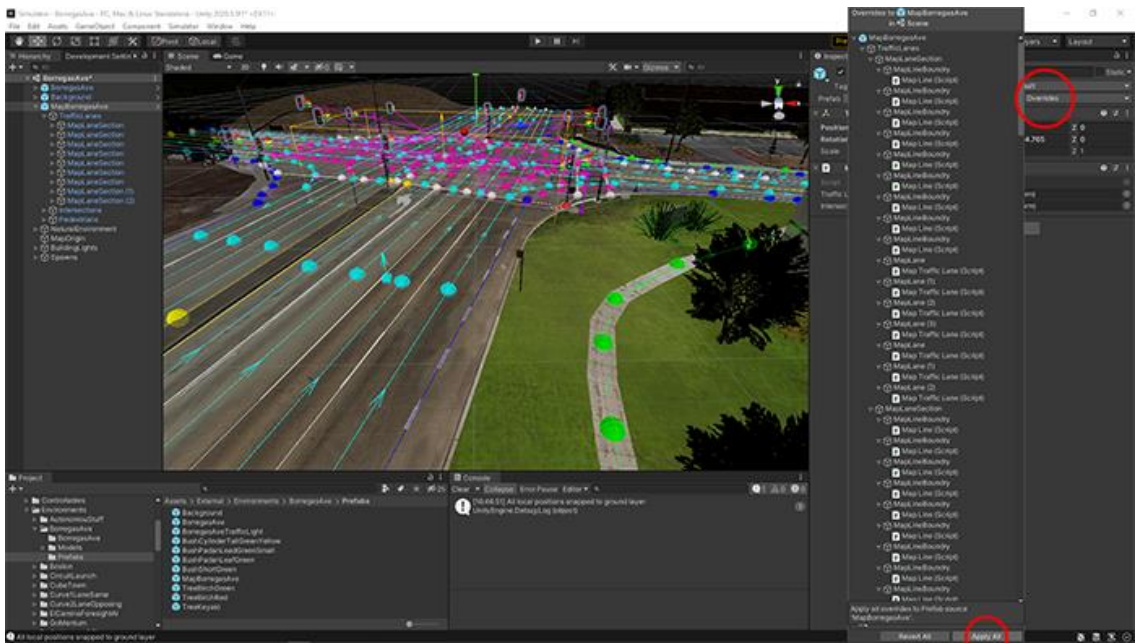


Figure 65. Example of map with annotations

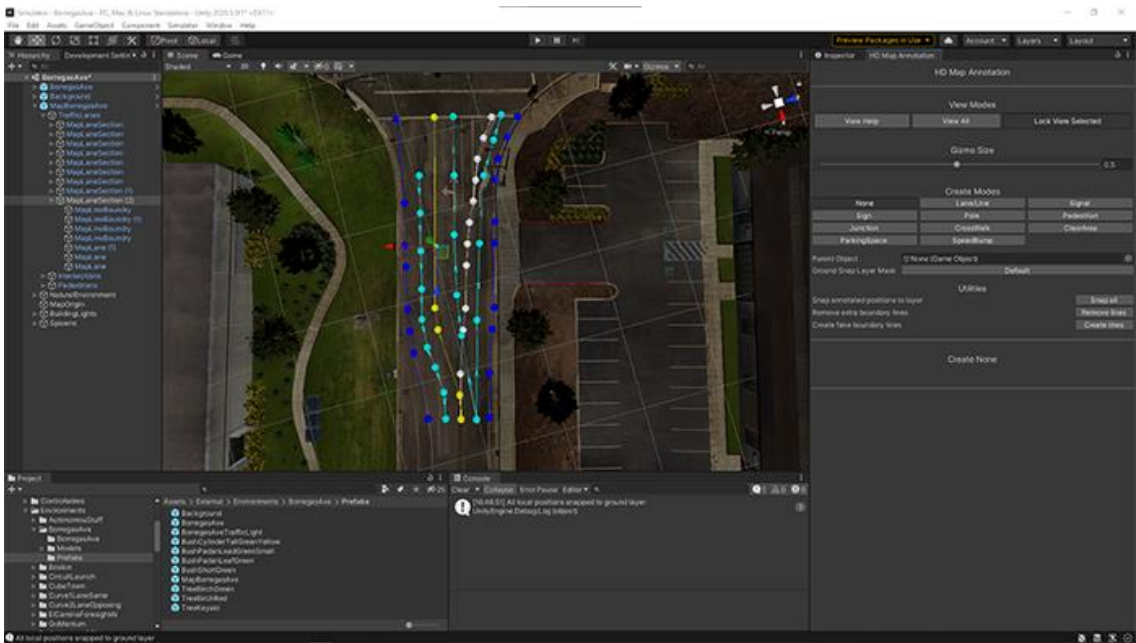


Figure 66. Example of map with annotations

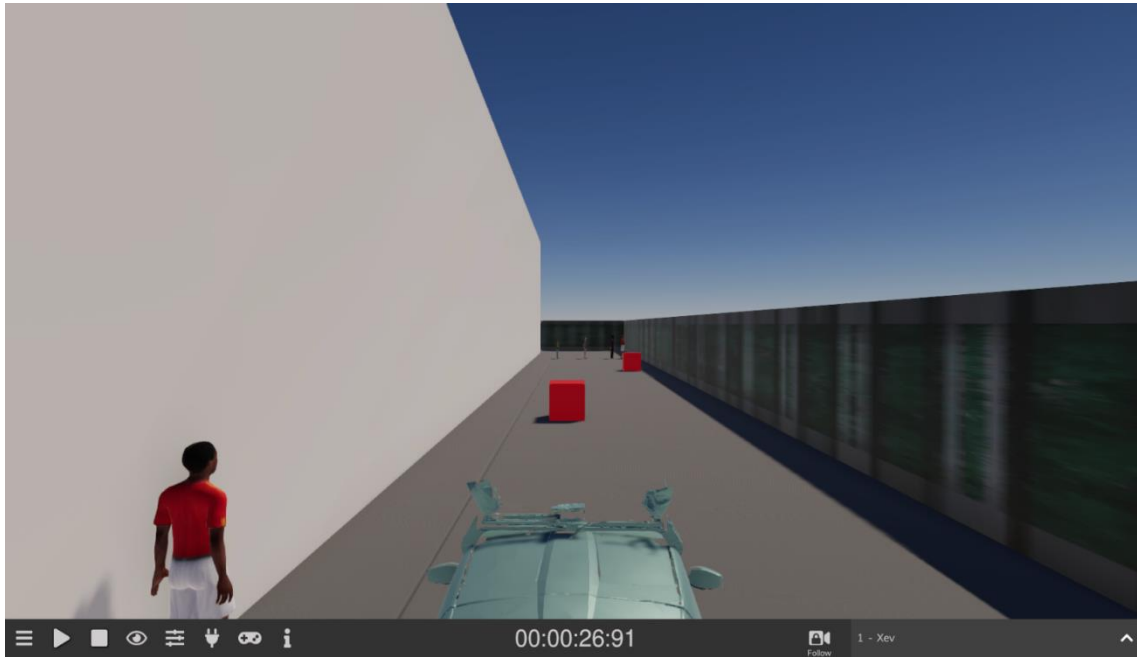


Figure 67. Our map, first section

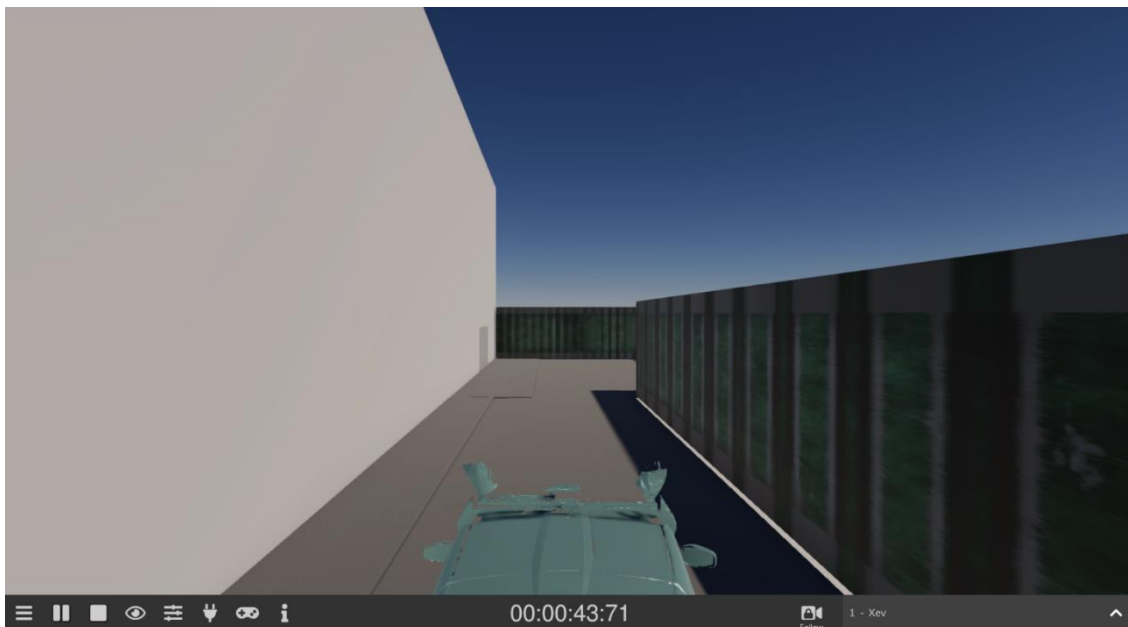


Figure 68. Our map, second section

The simulator also allows us to display on screen the beams emitted by the LiDARs. Fig. 69 shows the example of Field of View of the solid-state LiDAR equipped in our car; it may look confusing but it is due to the density of emitted pulses.

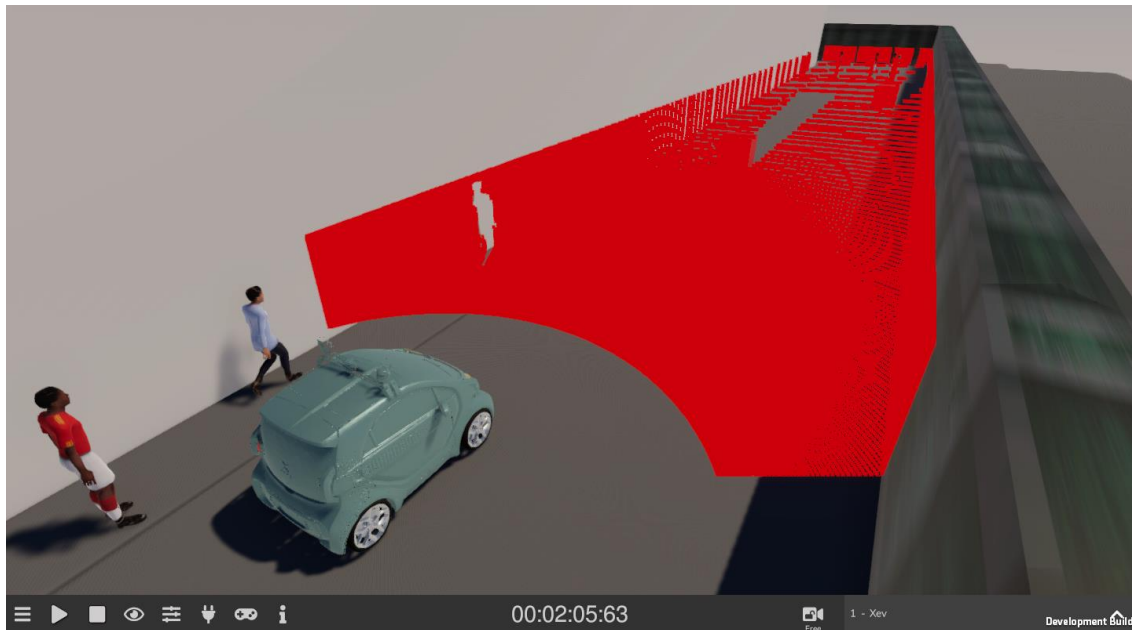


Figure 69. LiDAR FoV

Chapter 5

Simulations

In this chapter we will show the whole scenario we imported onto SVL Simulator and the graphic results of our simulations. Also, an unexpected challenge is presented.

Once uploaded the vehicle, its sensors and the map onto LG SVL Simulator (which also need to be added inside the Apollo master directory), and once completed the built of the Apollo software, we started launching the first simulations to check if any unexpected relevant errors regarding the car, map or the connection between SVL and Apollo were to be found. Luckily, it was not the case. We did have several issues configuring the machine in line with what the hardware and software requirements needed to launch the 2 softwares are, but that is not the objective of this work, hence these issues will not be analyzed. The requirements for SVL and Apollo can be found in [38] and [34], respectively.

Once resolved these problems, the approach taken was experimental: launch many simulations with different settings, changing the map, sensor configuration, detection algorithm, to create diverse scenarios and analyze the eventual issues and differences in detection performance among these several conditions.

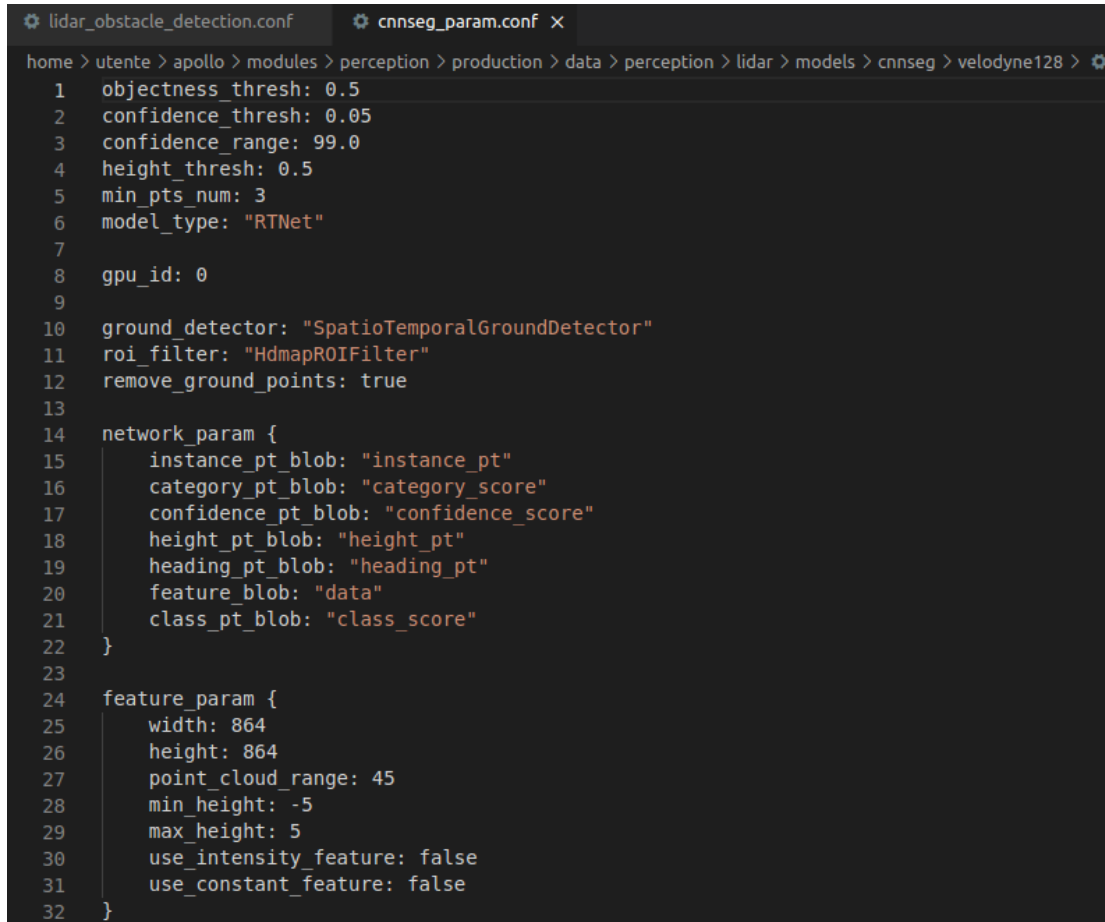
5.1 – First simulation

The goal of these simulations was analyzing how the solid-state LiDAR alone, without the 2 lateral mechanical LiDARs, would perform on object detection.

These simulations were launched used all the default settings, to have a first impact on how the simulation would perform. As default setting, the detection algorithm is the CNN Segmentation developed by Apollo, explained in section 3.3, and all its controlling parameters are unchanged.

As just mentioned, Apollo allows changing the values of some parameters that control certain aspects of its detection algorithm. This does not include editing the core components like layers, activation function or kernel sizes of the convolutions, but more

handy variables that can help improve the detection performance if some issues are to be detected. These parameters are all included in a file, and shown in Fig. 70.



```

1  objectness_thresh: 0.5
2  confidence_thresh: 0.05
3  confidence_range: 99.0
4  height_thresh: 0.5
5  min_pts_num: 3
6  model_type: "RTNet"
7
8  gpu_id: 0
9
10 ground_detector: "SpatioTemporalGroundDetector"
11 roi_filter: "HdmapROIFilter"
12 remove_ground_points: true
13
14 network_param {
15   instance_pt_blob: "instance_pt"
16   category_pt_blob: "category_score"
17   confidence_pt_blob: "confidence_score"
18   height_pt_blob: "height_pt"
19   heading_pt_blob: "heading_pt"
20   feature_blob: "data"
21   class_pt_blob: "class_score"
22 }
23
24 feature_param {
25   width: 864
26   height: 864
27   point_cloud_range: 45
28   min_height: -5
29   max_height: 5
30   use_intensity_feature: false
31   use_constant_feature: false
32 }

```

Figure 70. CNN Segmentation parameters

The values we will change will be the following, where:

- *objectness_thresh* indicates the objectness value over which we consider an object to be detected positively
- *confidence_range*: this refers to the confidence range associated to the prediction
- *height_tresh* indicates the same concept, but taking into consideration the height
- *min_pts_num* indicates the minimum number of points required to detect an object
- *point_cloud_range* indicates the range until which the points of point cloud are considered from the algorithm
- *min_height/max_height* indicate the min/max height to be considered for objects in our road scenario

The results of the first simulation are shown in the following figures.

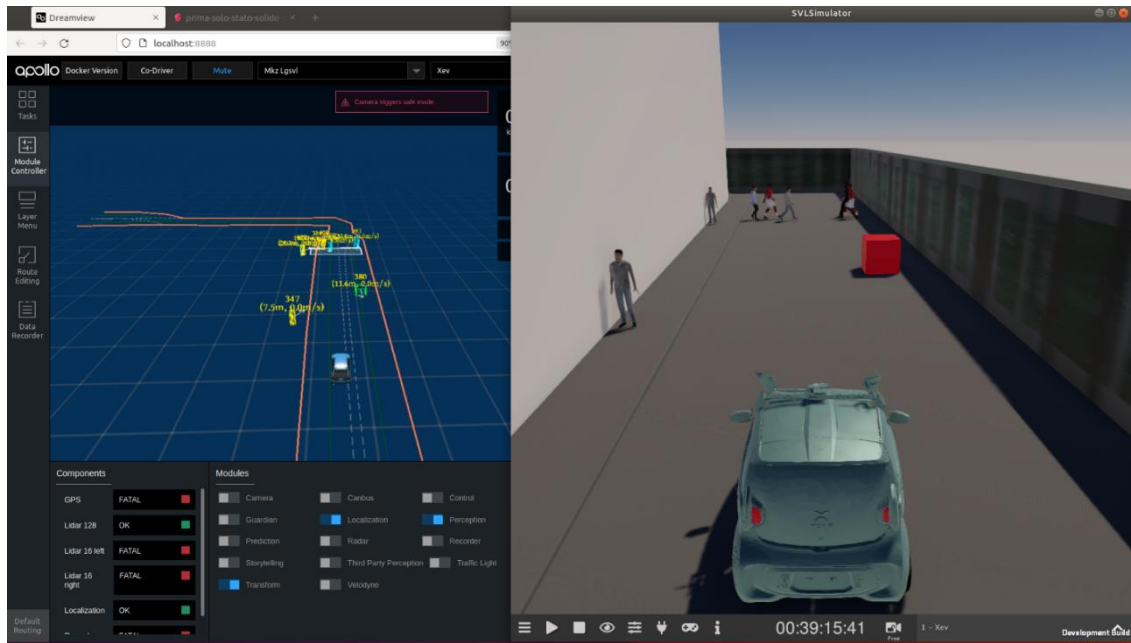


Figure 71. First simulation. Apollo (left) and SVL Simulator (right)

Module Delay	
Chassis	00:00.097
Control	-
Localization	00:00.078
PerceptionObstacle	00:00.082
Planning	-
Prediction	-
TrafficLight	-

Figure 72. Module delay associated to figure 70

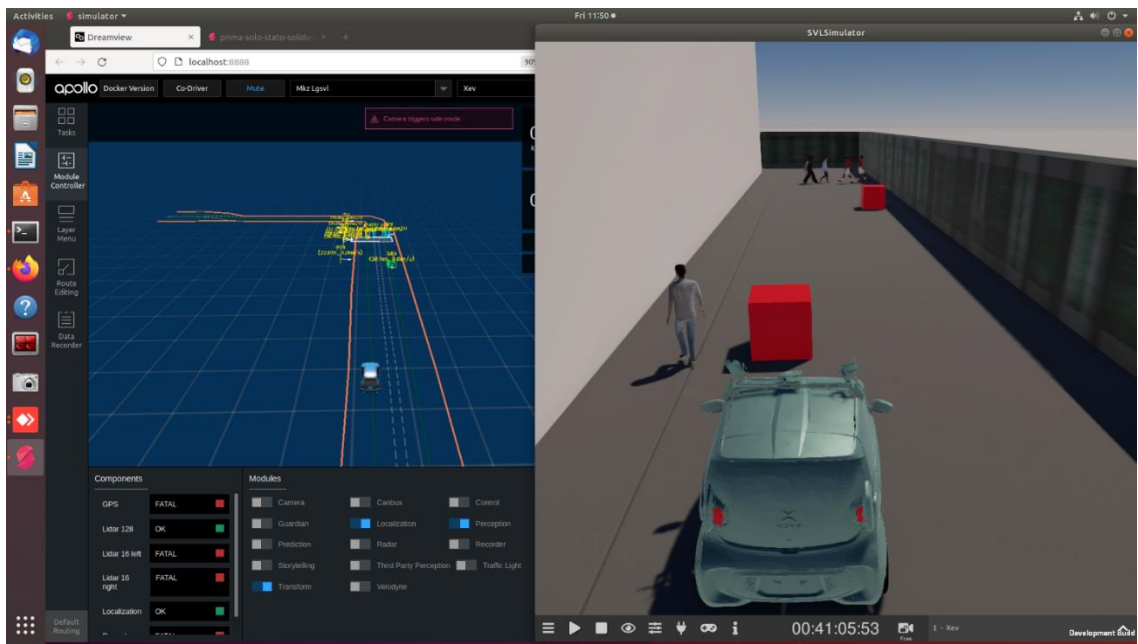


Figure 73a. First simulation, different time instant.

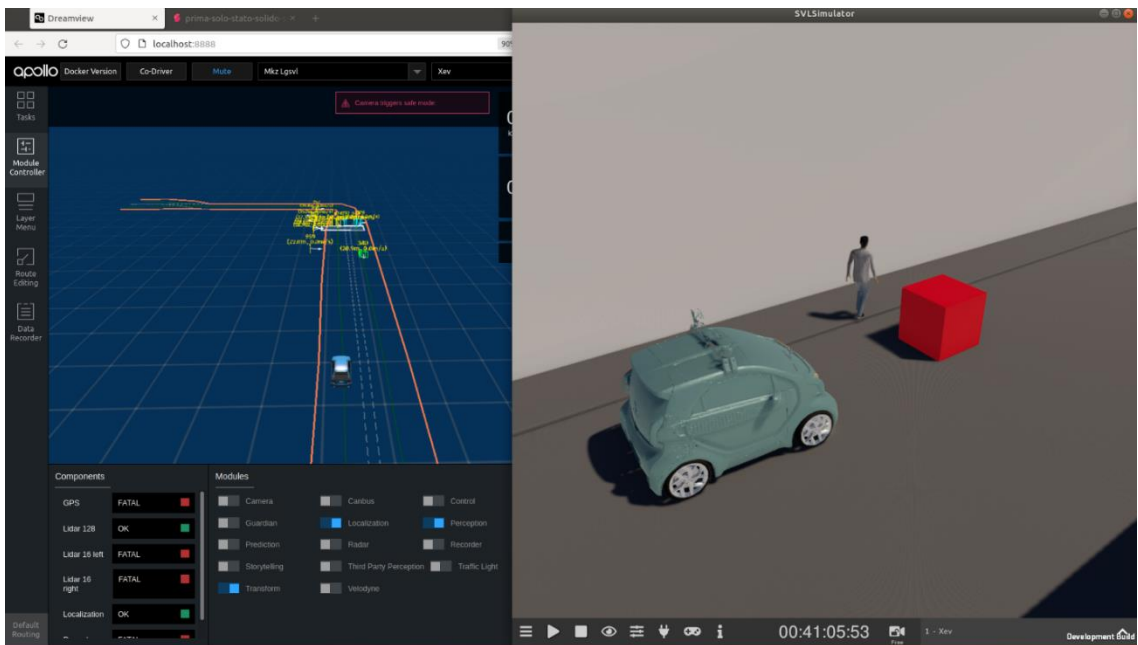


Figure 73b. Lateral view of scenario of Fig. 72

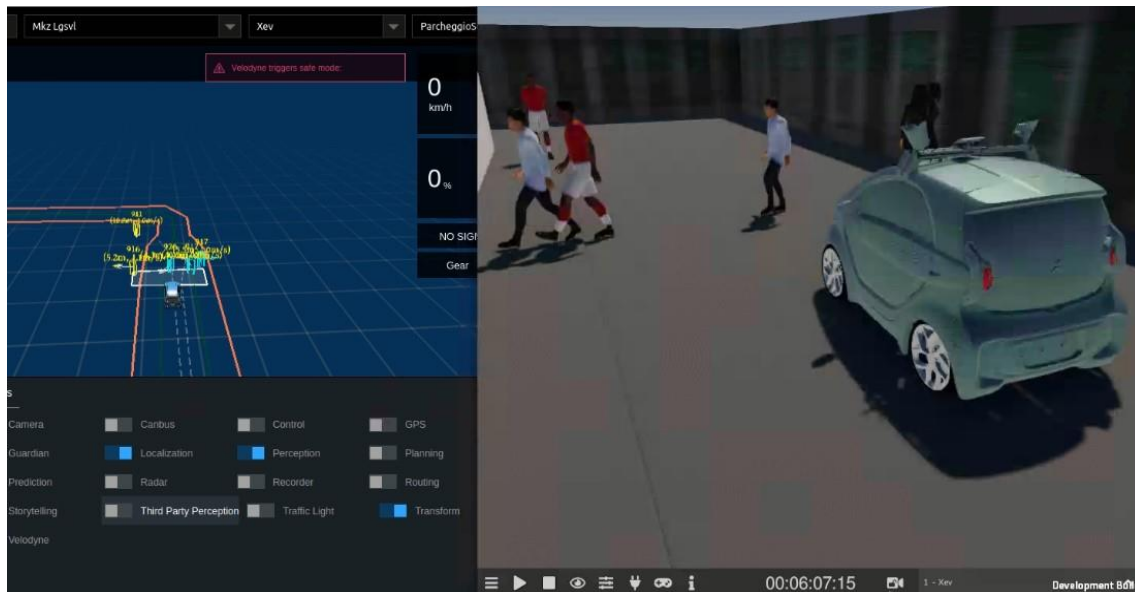


Figure 74. Detection of pedestrians at crosswalk

COMMENTS

As we can see in Fig. 71, the detection module seems to work relatively well: all the pedestrians are detected and encapsuled in yellow polygons with good accuracy, while the red cube obstacle is also detected correctly by a green box.

The second good result is that the prediction is made with little delay, as shown in Fig. 72. The delay of the detection module is 0.082 seconds in that specific instant of time. Clearly, this is an indispensable feature for an autonomous driving application. Note that this is not the average but a real-time value that oscillates continuously.

On the other hand, during these first simulations we have noticed an issue: under a certain short distance, approximatively a couple meters, the program is not able to detect objects and/or pedestrians in front of the vehicle. This problem is shown in Fig. 73a and 73b, where we can clearly see, from two different perspectives, the above-mentioned detection fault: neither the red box nor the pedestrian in front of the vehicle are being detected in Apollo's view.

Figure 74 shows the result when approaching a crossroad with pedestrians. Here the detection of the pedestrians is pretty accurate, while the vehicle is still close to the crossroad. This result tells us that the algorithm generally struggles more in the detection of short objects, like the ones shown in Fig. 73, while it has less problems when several taller shapes are grouped together.

As SVL Simulator allows to visualize the field of view of each sensor, we activated that of the solid-state LiDAR (the only in use in this simulation) to see whether the problem

was related to the field of view (for example due to only a partial visibility of the object) or not. The result is shown in Fig. 75.

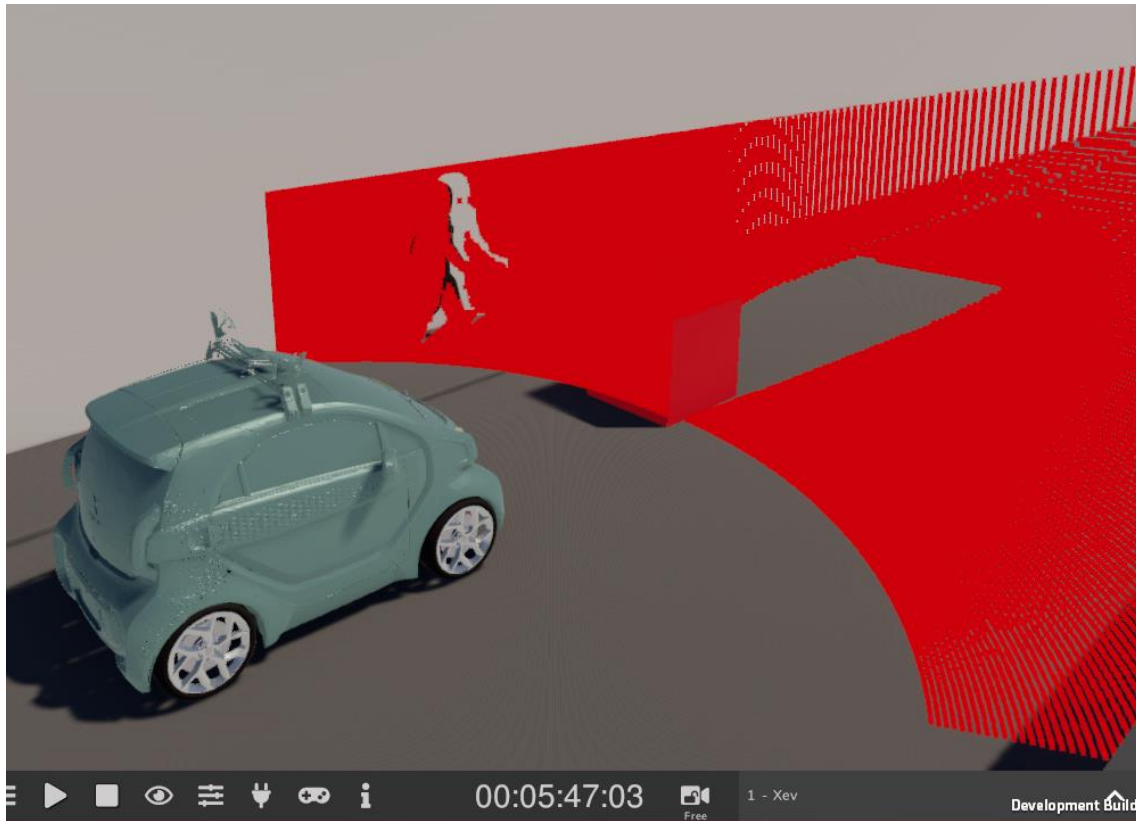


Figure 75. LiDAR FoV of Fig. 73

As we can clearly see from the image, the object and pedestrian are fully struck by the beams emitted by the LiDAR while still not being detected. This proves that the detection fault is most likely not due to an incomplete visibility of the obstacle in the FoV of the LiDAR, but rather to a more low-level issue in the algorithm that we yet do not comprehend.

5.2 – Second set of simulations

Given that the problematic highlighted in the first simulations does not depend on the field of view of the sensor, the first idea was that the fault would be caused by an error within the detection algorithm. As explained in section 5.1, we now experimentally try to modify the parameters that control the default CNN Segmentation used by Apollo, and see if we manage to find a combination that limits the problem.

In the following pages are reported images of simulations with several different configuration of parameters. The adopted configurations are reported in Table 7.

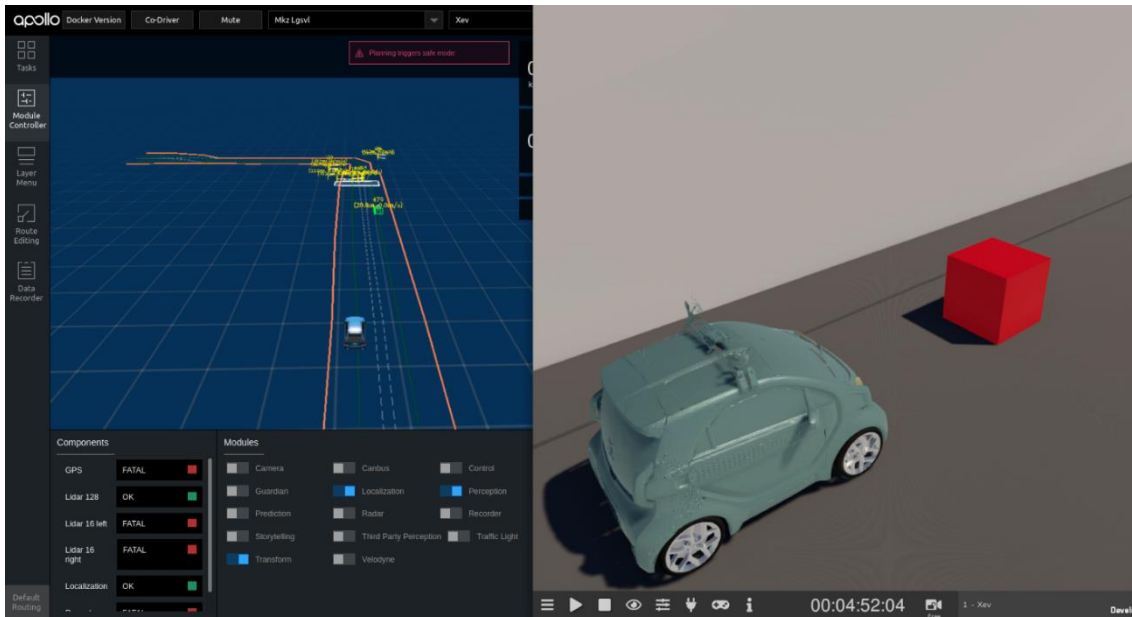


Figure 76. Detection with reduced height_thresh

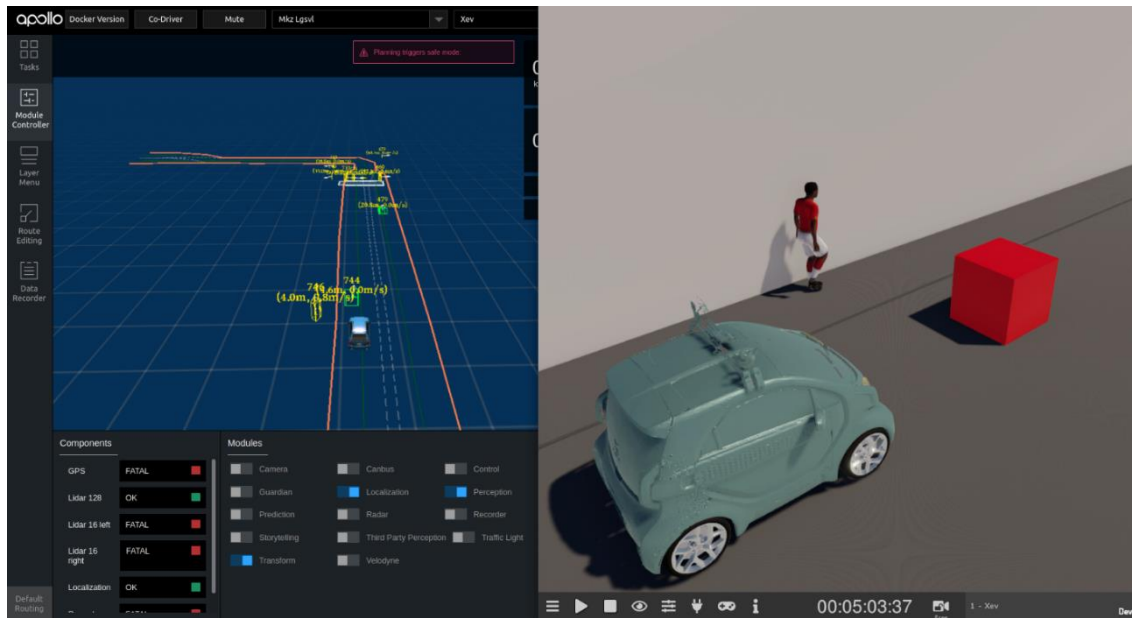


Figure 77. Detection with reduced height_thresh

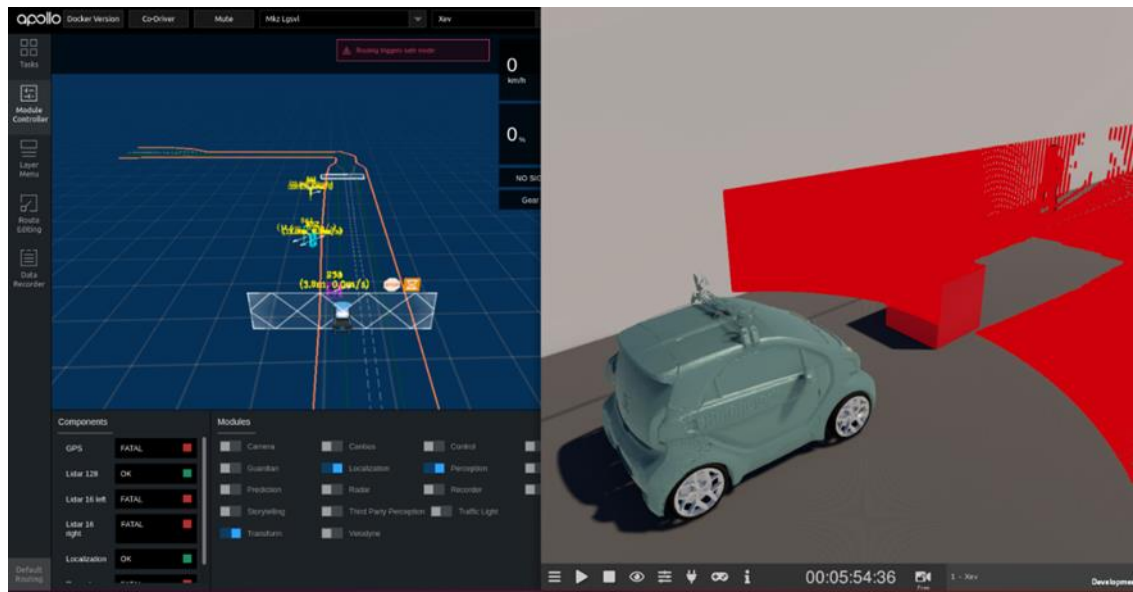


Figure 78. Detection with reduced point_cloud_range

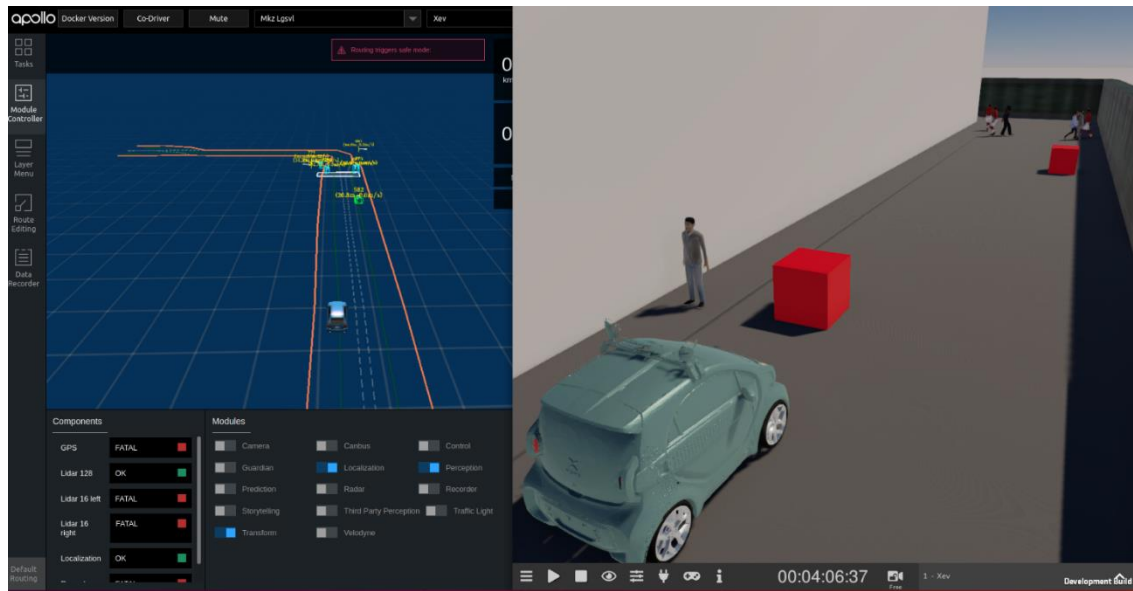


Figure 79. Detection with confidence_range lowered

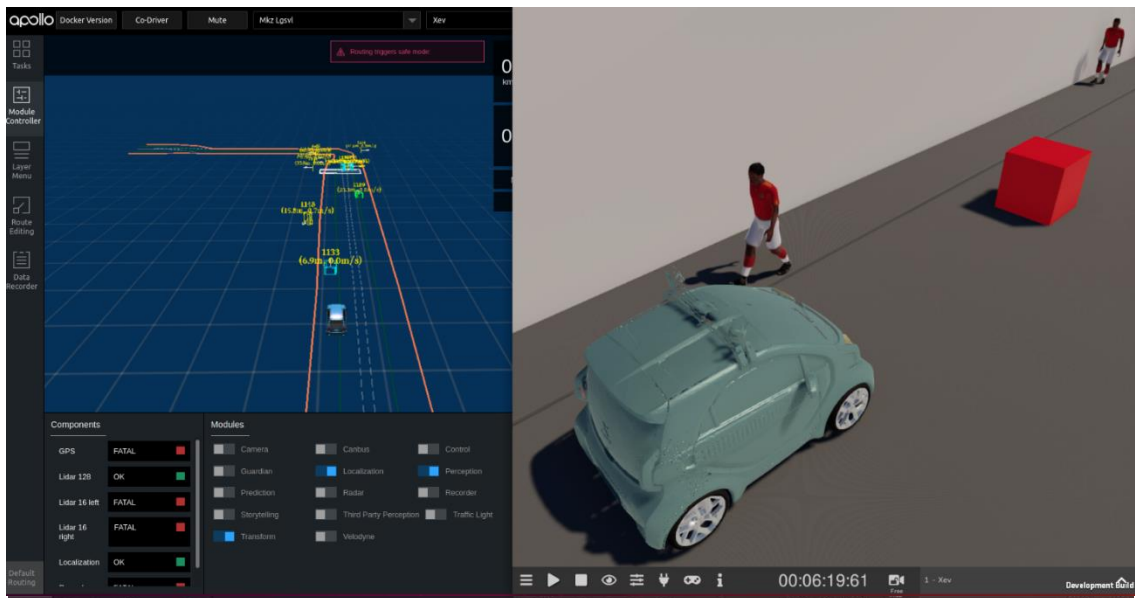


Figure 80. Detection with confidence_range lowered

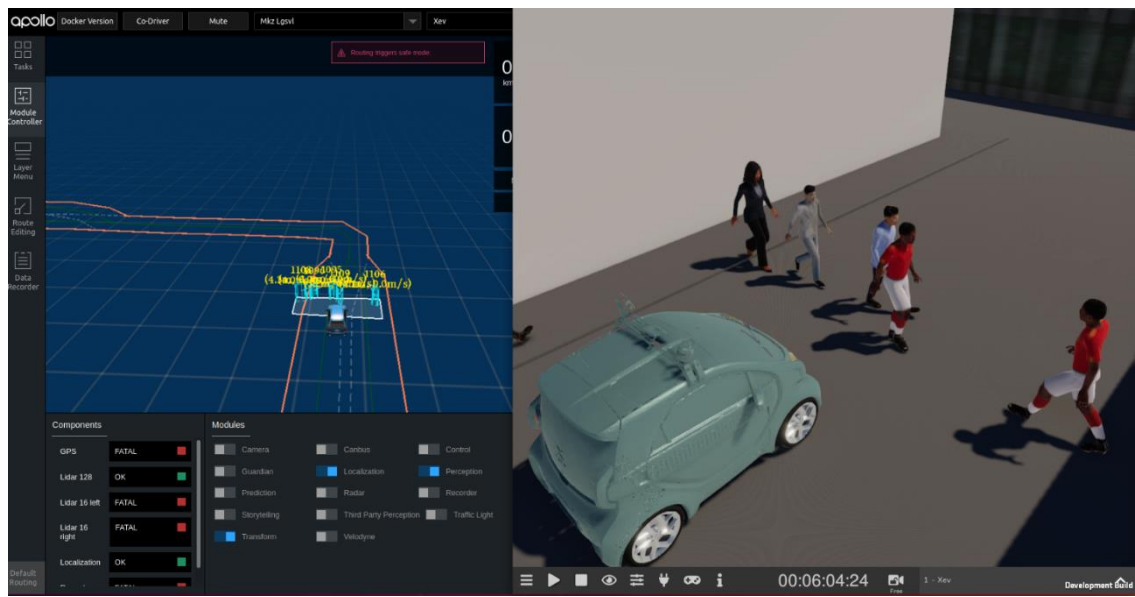


Figure 81. Detection with confidence_range lowered

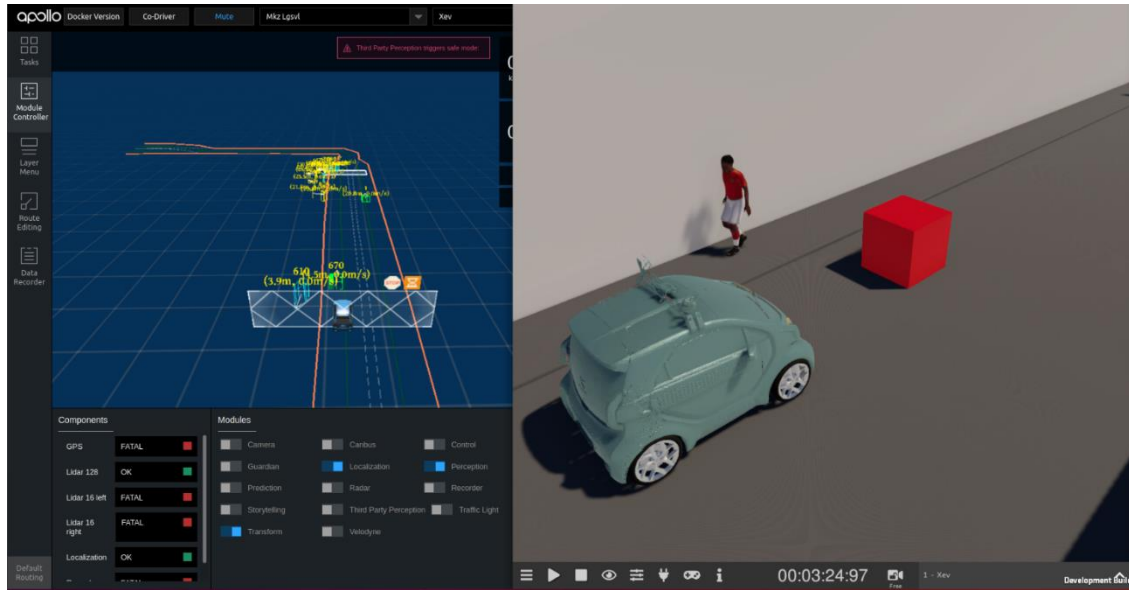


Figure 82. Detection with Objectness_tresh lowered

Table 7. Sensor configuration of the different simulations

	Edited parameter
Figure 75	height_treshold reduced to 0.3
Figure 76	height_treshold reduced to 0.3
Figure 77	point_cloud_range reduced to 20
Figure 78	confidence_range reduced to 80
Figure 79	confidence_range reduced to 80
Figure 80	confidence_range reduced to 80
Figure 81	objectness_tresh reduced to 0.3

COMMENTS

We shall now comment the result of these different configurations that we tried, with respect to the original standard sensor configuration presented in section 5.1.

In Fig. 76 and 77, the parameter *height_thresh* has been reduced, as we noticed that obstacles are more prone to be wrongly not detected rather than pedestrians. Therefore we thought that the height threshold could be something affecting the detection fault. As we see from the figures, no substantial change has emerged, although comparing the two images we notice a strange behavior: maintaining the same distance from the obstacle, the algorithm is able to detect the red cube when a pedestrian enters the FoV and walks close to it, as it had some kind of influence on the way the software processes the point cloud. We are not sure what precisely causes this issue, but it is important to take notice of it.

In the simulation displayed in Fig. 78 we tried lowering the parameter that controls the range of points considered for the detection, called *point_cloud_range*. As we expect from the name, the first difference that is easy to notice is that the software does not detect the obstacle farther away, nor the pedestrians crossing. In return, the vehicle is able to detect the closer obstacle up to a shorter distance, as shown in the image.

Fig. 79, 80 and 81 represent instants of the simulation in which we lowered the *confidence_range* parameter, in hope of improving the situation. The results are somewhat in contrast. In Fig. 79 and 80 we see how both the obstacle and the pedestrian are not detected when in close range, like in the original issue. On the other hand, Fig. 81 shows good improvement when detecting multiple pedestrians in front of the vehicle in short range, which did not happen with this efficiency with the original sensor configuration.

In the last simulation we reduced twice the threshold of minimum score of objectness to see if this would lead to more detections, even if that might induce the detection of more false positives. The result is shown in Fig. 82. As we see, no improvement is made, and the scenario is similar to that of Fig. 77: the obstacle is detected when a pedestrian walks close to it. This last one is an important result because it proves that the problem is not that the detection is made but then discarded because it has a low objectness score associated, but rather on the detection step itself, that in some way is not able to process the information from the point cloud to correctly detect the shape as an object.

We also tried a configuration lowering the value *min_height*, but no improvement was noticed. Hence, no image regarding this scenario has been reported in the paper.

Table 8 summarizes the results described in a more eye-catching fashion.

Table 8. Sensor configuration changes in the different images

Configuration	Outcome
height_treshold reduced to 0.3	Obstacle detected only when a pedestrian walk close to it.
point_cloud_range reduced to 20	Obstacle detected until shorter distance, but farther obstacles are not detected anymore.
confidence_range reduced to 80	No improvements in normal situation, only improved detection at a pedestrian crossroad.
objectness_tresh reduced to 0.3	Same results of first case.
min_height reduced to 2	No improvements.

5.3 – Third set of simulations

In the previous section we tried to change all the software parameters that most control the algorithm used for detection, CNN Segmentation. In this short third type of approach we instead try to change one critical physical parameter: the angular orientation of the solid-state LiDAR.

Since the very first simulations we notice the presence of a *shadow cone* right in front of the vehicle, as shown for example in Fig. 75. This is due to:

- the orientation of the sensor
- the vertical FoV of the sensor
- the presence of the hood of the vehicle that blocks the pulses of the sensor

Clearly, we do not have control over the second and third constraints. Therefore, we shall now try to change the orientation of the sensor by tilting it down, trying to narrow down the *shadow cone* right in front of the vehicle. Obviously, given that the vertical FoV of our sensor is fixed and cannot be changed, we expect the farthest areas of the previous settings to now fall out of the field of view. The result is given in Fig. 83.

The initial setting for LiDAR orientation, based on a parameter called *CenterAngle*, is of 12.5° .

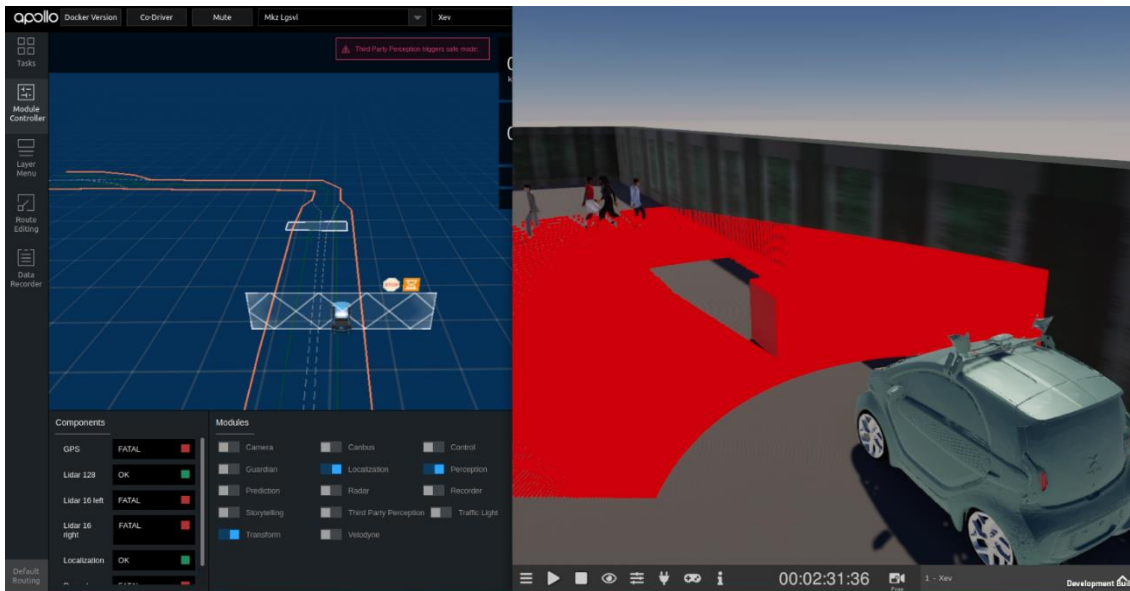


Figure 83. Detection with lower orientation angle

COMMENTS

As we can see from Fig. 83, the FoV is not more concentrated in the areas closer to the front of the vehicle, while the parts farther ahead are now not struck by the pulses of light. Therefore, the object can now be fully inside the FoV up until a shorter distance, but

despite this, the software still does not detect the obstacle. Since the loss of information in the farthest parts of the and no improvement in detection of the red cube, we can discard this option as a solution as it does not add any positive aspect. This configuration further proves the fact that this issue cannot be related simply to the limited vertical field of view of the sensor (25°), but rather on a lower-level problem related to the detection step.

5.4 – Fourth set of simulations

In the last 3 sections, we have tried to change every parameter – mechanical or digital – we could think of given our understanding of the whole detection pipeline adopted by Apollo. We noticed some differences between some of these configurations, but unfortunately we could not achieve one that solves the issue. Another change we thought of doing, for sake of experimentally see if any major difference would arise, is to change the map of the simulation. We will use a more complex map, with more annotations (lanes, crosswalks, traffic lights, horizontal/vertical signs). We are not looking for any specific and targeted goal here, just noting down if the software behaves differently, and if so reflect on possible reasons. Let us remember that, also in this case, we are trying the configuration using only the solid-state LiDAR. Fig. 84 to 85b show the results of these simulations.

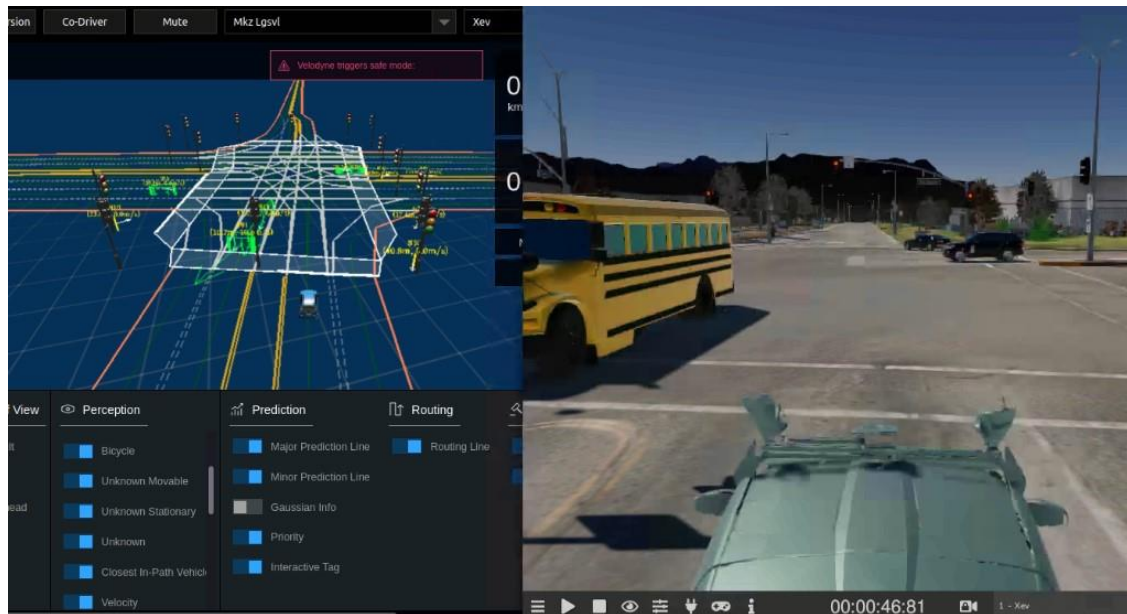


Figure 84. Detection at intersection

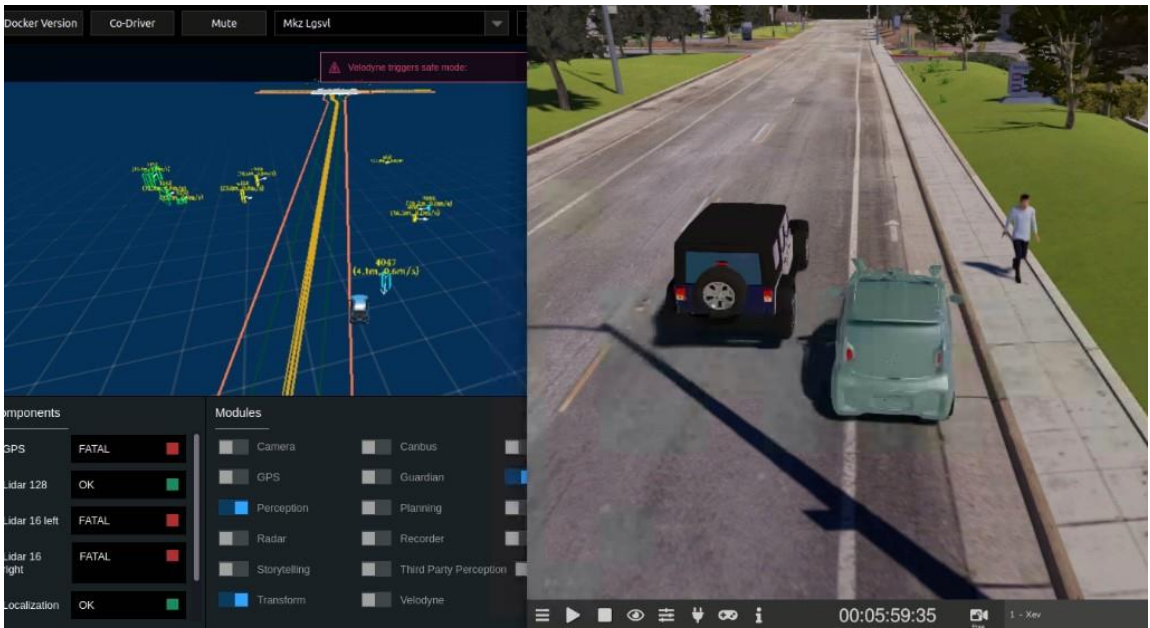


Figure 85a. Close range detection of pedestrian

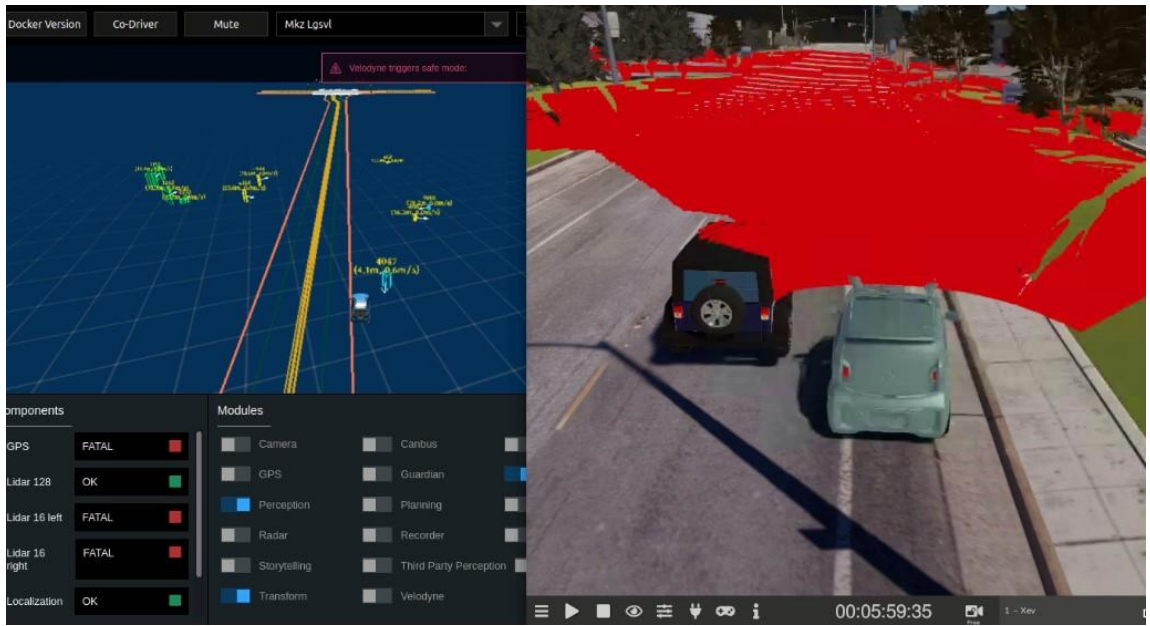


Figure 85b. LiDAR’s FoV of Fig. 85a



Figure 86. Detection of two pedestrians

COMMENTS

As we see from the pictures, this default map of Apollo is much enriched with map annotations like all HD Maps should be. This makes the decision-making easier for the detection and planning modules. Please note that the traffic light module is not being used.

In Fig. 84 the vehicle is about to approach a large crossroads, and it seems to accurately identify the yellow bus coming to his left, and the vehicle right in the middle of the junction.

In Fig. 85a and 85b we put our vehicle in one of the situations in which it struggled in the previous sections: the short-range detection of one single pedestrian. As the images show, this time the detection is precise, although the car on its left is not perceived. But, as shown in Fig. 85b, this is due to the fact that the vehicle is only partially visible inside the LiDAR's FoV. The algorithm was able to correctly detect it once the whole car was visible inside the highlighted area.

Fig. 86 represents another case scenario of detection of pedestrians at different ranges. Also in this case, the software seems to work better as it recognizes both of them. In this situation we also notice that the network identifies the non-road objects on the right (pipes, vegetation) as unknown shapes.

5.5 – Fifth set of simulations

Up to now, in all the simulations carried out, we decided to analyze how the detection would perform using solely the solid-state LiDAR equipped on the vehicle, which is the most economic and obviously also the least performant LiDAR, when compared to the mechanical (rotative) ones that are also mounted on the car. The reason behind it was to maximize the efficiency using only this sensor to see if object detection could be run using only a solid-state LiDAR, which would imply a drastic reduction of the cost of the project.

Since we realized that we are not able to fully do so, at least within a high-level type of control, in these new simulations we will also use the 2 lateral mechanical LiDARs to see if and how much the presented issue is resolved. This technique is called *sensor fusion*, and the idea behind it is simple: join the informations provided by different sensors to augment the effectiveness of the detection module. This is because, as we have seen in Fig. 10, each sensor has its strengths and weaknesses, and performs better under specific conditions of weather/distance/purpose. By using the data provided by different sensors we are able to compensate for these weaknesses and often most of the detection issues vanish.

There are different levels of sensor fusion, based on the level of abstraction in which it is applied. The most common are:

- **Data level:** this type of fusion occurs in the early stages of the pipeline, indeed it merges raw data from the multiple sensors. This results in more accurate readings. In our case this consists in merging the point clouds of different LiDARs into a unique one with a global reference system centered in the center of the vehicle. To do so, a transformation matrix is calculated to map each point belonging to a point cloud into the new reference system. The only drawback is that this method creates quite big input spaces, hence it might influence the velocity of the whole pipeline. Data level fusion is the most common approach to sensor fusion, and it is at the most abstract level.
- **Feature level:** with this approach, the sensor fusion is applied not directly on raw data, but once the net has extracted features from the input. The independent-extracted features are then sent to a fusion node that merges and compares them. Normally not all the features are selected to be passed on but only the most significant ones, which require some training to be selected. This method is more efficient in terms of calculations with respect to the first one.
- **Decision level:** at this level of abstraction the fusion is applied directly on predictions carried out from the individual sensors. Among these hypothesis, the most reliable is then selected, according to some decision-making algorithms.

Decision fusion is the most ‘lightweight’ and it allows for fusion of heterogeneous sensors, since it does not need to merge the input data.

The most common type of sensor fusion for AD applications merge the informations from LiDARs, cameras and sometimes radars too, to gain the whole spectrum of information it possibly can. Notable works that propose sensor fusion are [36] [37] [38] [39].

As already mentioned, we will now run the detection module with the use of all 3 LiDARs equipped on our vehicle. We will not include the cameras as a first stage on this project, since they focus more on the detection of traffic lights and road signals, which is not the primary scope of this work. The results are shown in Fig. 87 to 91.

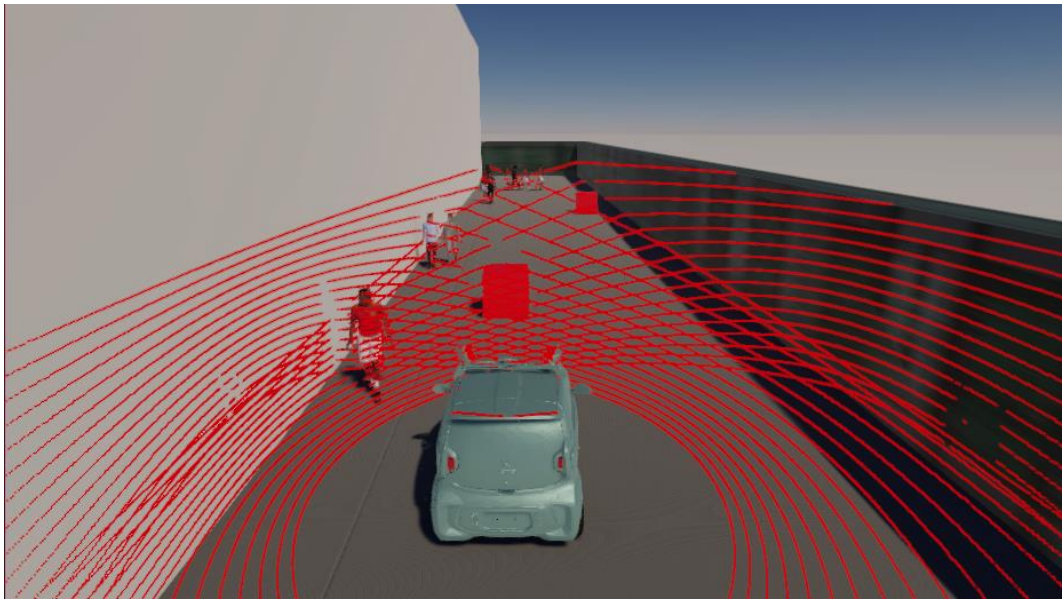


Figure 87. FoV of the two rotative LiDARs

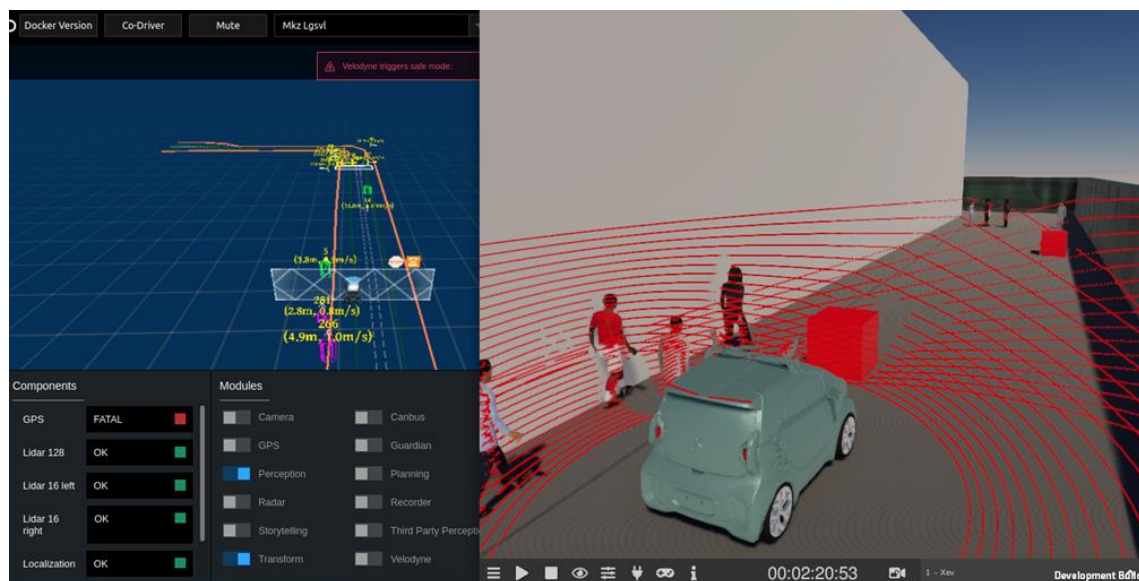


Figure 88. FoV of all 3 LiDARs, superimposed

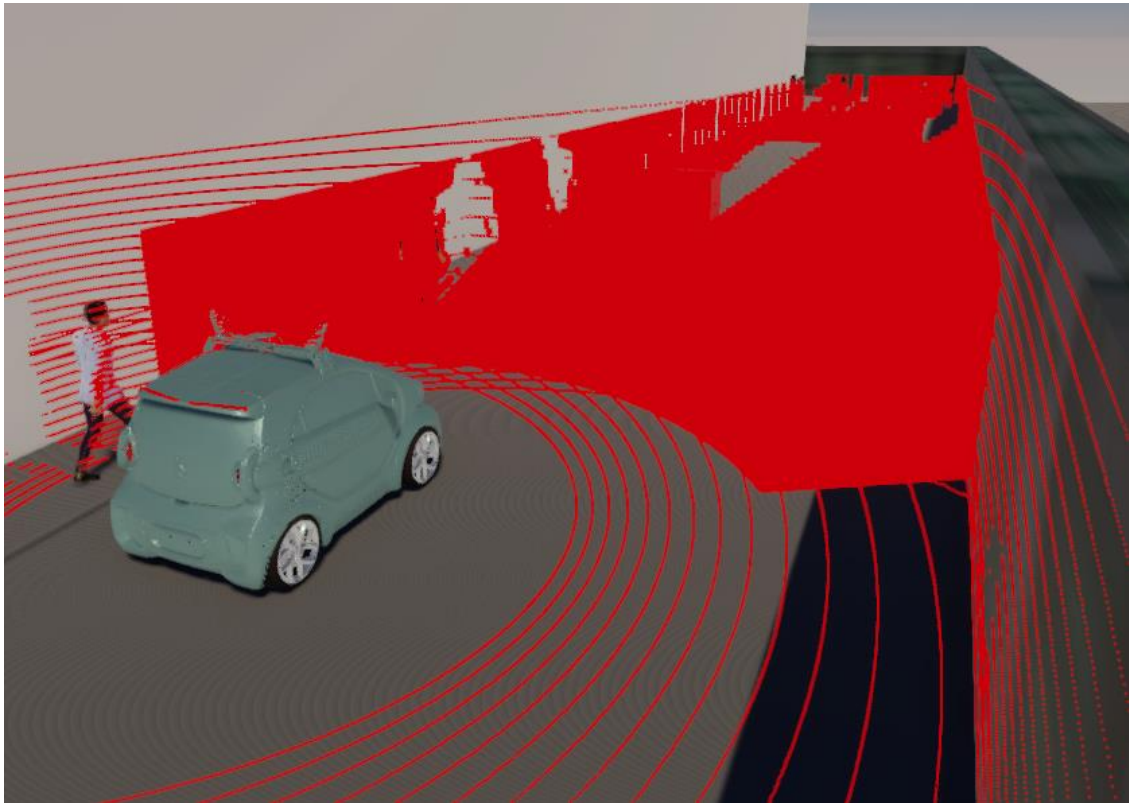


Figure 89. Perception with the 3 LiDARs working

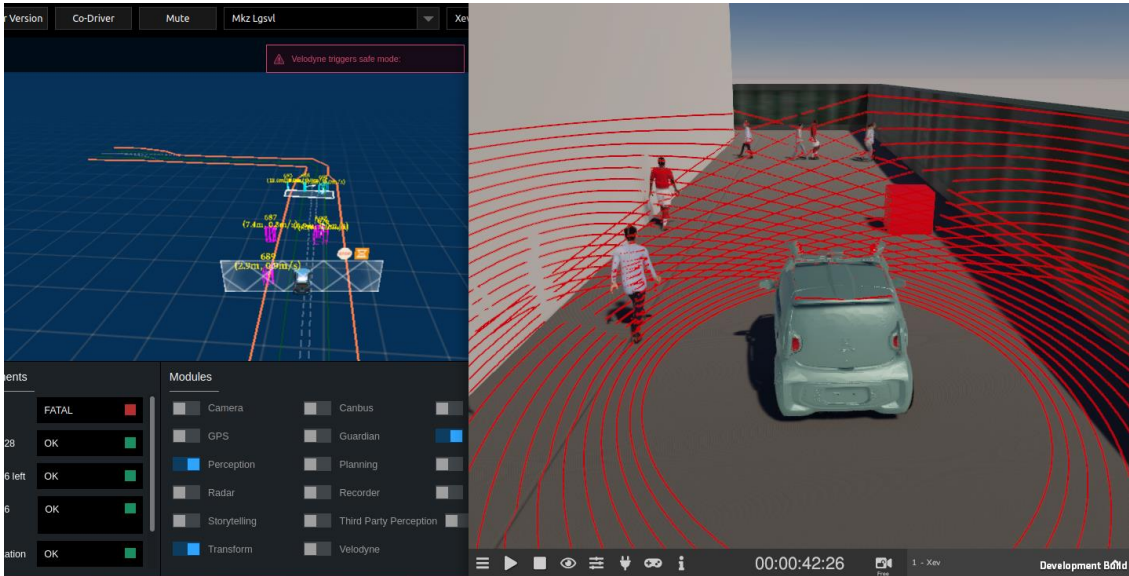
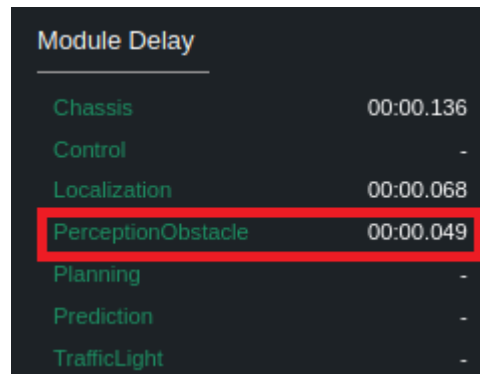


Figure 90. Perception with the 3 LiDARs working



Module Delay	
Chassis	00:00.136
Control	-
Localization	00:00.068
PerceptionObstacle	00:00.049
Planning	-
Prediction	-
TrafficLight	-

Figure 91. Module delay

COMMENTS

Fig. 87 shows the field of view of the 2 lateral rotative LiDARs: these sensors form what essentially is an annular region around the car in which they are able to sense objects. In Fig. 88 we show both the rotative and solid-state LiDARs fields of view working together: it is nothing more than a simple superposition of their individual FoV. The two most important improvement this implies are: a denser forward-facing FoV, and a 360° FoV around the vehicle thanks to the rotative sensors introduced.

In Fig. 89 and 90 we show how the detection network perform with this new configuration. We notice that now the obstacle is finally detected at short distance, and so are the pedestrians. While sensing the close objects the software is also able to recognize the shapes further ahead with good accuracy. With the use of the rotative LiDARs, Apollo is now able to detect pedestrians also on the sides and behind the vehicle.

Fig. 91 proves that the delay time of the module is still at more-than-acceptable level at 0.049 seconds.

This result is exactly what we expected when introducing sensor fusion. The multiple sources of input allows the creation of a much denser point cloud from 3 slightly different perspectives, since the 3 sensors are not physically mounted in the same point but purposely at the maximum distance allowed by the dimensions of the car. Such decision provides a kind of *stereo-vision*, an approach that is well known for improving detection performance from images/point clouds since it merges data from different perspectives.

The outcome has compensated the solid-state LiDAR weaknesses as expected, and it results is a much improved version of the detection, which shows good accuracy, 360° awareness and good inference time.

For future research and further development of the project it would be interesting to see if Apollo fixes the problems it still has running the other detection algorithms it includes, so to see if running these network would eliminate the problems we encountered using

solely the solid-state LiDAR. Also, mounting radars on the vehicle and including them in the sensor fusion, as well as the cameras, should be a topic of study for future development.

Chapter 6

Conclusions

The evolution of autonomous driving systems in the last decades has been just incredible. In this work, we started by diving into the history of autonomous vehicles, from the first rudimental projects that relied on magnetic circuits embedded in the road, to the DARPA challenges, and finally to the total autonomy reached on specific road conditions achieved by Tesla cars and Waymo's taxi fleet.

Perhaps the most important part of an AV is its detection software, which is responsible for detecting, categorizing and tracking obstacles and other road users in the environment. The safety of driver, passengers and other people in the vicinity of the vehicle rely heavily on the performance of its detection module. Therefore, before putting an autonomous car onto the road, it is vital to virtually simulate its performance in a digital environment, as to prevent any possible danger to real people and buildings, but also to tune the parameters for the. This has been the scope of this project.

To do so, we also needed the required theoretical background on the sensors used in AD and the most important networks for object detection, introduced in chapters 1 and 2. In Chapter 3 and 4 we presented the two softwares used for the simulations and their main features. We also described how we uploaded onto the simulator our own customized map and digitalized version of the vehicle with its sensors.

In chapter 5 we started running the simulations with our car and map. The default settings produced fairly good results of the detection module in the middle and long-ranges: both obstacles and pedestrians were correctly identified and tracked in these situations. Nevertheless, a problem arises when dealing with short ranges, in which the software is not able to detect neither the obstacles nor the pedestrians. Therefore, we started to experimentally change some of the parameters that mostly influence the detection network. We did notice some slight differences, but always trading off on some other aspect. We so reached a plateau, and realized that the detection module couldn't be run relying only on the single solid-state LiDAR.

The solution adopted in this case is sensor fusion, which in our scenario means including also the other 2 rotative LiDARs the car is equipped with. By adopting this technique a

new point cloud is now generated merging the input from all 3 sensors, which results in a much denser and richer cloud that improves dramatically the performance of the detection system. In fact, the previously explained issue is now resolved. Moreover, with this configuration the software can now detect objects behind the vehicle and on its sides as well, since the new LiDARs introduced are rotative.

Sensor fusion is the way to go when dealing with autonomous driving detection, and it is adopted by all car manufacturers that produce autonomous vehicles. It is still topic of study by the scientific community, that seek to reduce its computational load while improving even more the accuracy in all conditions.

We were able to create a reliable simulation environment which reproduces the real car, sensors and map with fidelity. The power of this system is that with just a few clicks we can change physical and environmental parameters and rapidly see how they affect the behavior of the car and detection module. Without this structure, it would be very time consuming to physically change these parameters on the actual car, not to mention the eventual danger it would create when tested on an actual road.

For future development on this project, it will be surely important to also include cameras and radars inside the sensor fusion, as to further improve the accuracy of detection in the short range (aspect in which the radars excel) and activate the traffic light and road signaling modules. A second improvement will be possible when Apollo will release a more stable version for the use of the other detection algorithms included in its framework. Unfortunately, we were not able to do so, but as a first approach to the problem the use of the default detection algorithm developed by Apollo is more than sufficient.

Bibliography

- [1] SAE International, *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, revised 2021
- [2] P. Viola *et al*, *Rapid object detection using a boosted cascade of simple features*, 2001.
- [3] N. Dalal, *et al.*, *Histograms of oriented gradients for human detection*, IEEE CVPR, 2005.
- [4] P. F. Felzenszwalb, *et al.* *Object Detection with Discriminatively Trained Part-Based Models*, in IEEE TPMAI, 2010.
- [5] Girshick, Ross, *et al.* *Rich feature hierarchies for accurate object detection and semantic segmentation*, Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.
- [6] E. D. Dickmanns, *Dynamic Vision for perception and control of Motion*. 2010.
- [7] Y. Zhou and O. Tuzel, *VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection*, arXiv, 2017. vi, 7, 21, 27, 29, 30, 31, 45, 52
- [8] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, *Multi-view 3d object detection network for autonomous driving*, in CVPR, 2017. v, 7, 14, 19, 22, 23, 24, 39, 45, 52
- [9] K. Minemura, H. Liao, A. Monrroy, and S. Kato, *LMNet: Real-time Multiclass Object Detection on CPU using 3D LiDAR*, May 2018. 7, 21, 52
- [10] Uijlings, J.R.R., Van de Sande, K.E.A., Gevers, T. *et al.* *Selective Search for Object Recognition*. Int J Comput Vis 104, 154–171 (2013)
- [11] Pascal VOC 2008: <http://host.robots.ox.ac.uk/pascal/VOC/voc2008/index.html>
- [12] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*. In ICLR, 2014.
- [13] N.I. Glumov, E.I. Kolomiyetz, V.V. Sergeyev, *Detection of objects on the image using a sliding window mode*, Optics & Laser Technology, 1995.
- [14] Koen E. A. van de Sande, Jasper R. R. Uijlings, Theo Gevers, Arnold W. M. Smeulders, *Segmentation As Selective Search for Object Recognition*, IEEE International Conference on Computer Vision, 2011

-
- [15] Felzenszwalb, P.F., Huttenlocher, D.P., *Efficient Graph-Based Image Segmentation*. International Journal of Computer Vision 59, 167–181 (2004)
- [16] Simonyan, K., Zisserman, A. (2015, April 10). *Very deep convolutional networks for large-scale image recognition*. arXiv.org
- [17] Szegedy, C., Ioffe, S., Vanhoucke, V., Alemi, A. (2016, August 23). *Inception-V4, inception-resnet and the impact of residual connections on learning*. arXiv.org
- [18] Krizhevsky, A., Sutskever, I., Geoffrey E. Hinton (2012, December 1). *ImageNet classification with deep convolutional Neural Networks*: Proceedings of the 25th International Conference on Neural Information Processing Systems - volume 1.
- [19] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A. (2014, September 17). *Going deeper with convolutions*. arXiv.org.
- [20] Girshick, R., (2015, December 1). *Fast R-CNN*, Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV).
- [21] Ren, S., He, K., Girshick, R., Sun, J. (2016, January 6). *Faster R-CNN: Towards real-time object detection with region proposal networks*. arXiv.org.
- [22] Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2016, May 9). *You only look once: Unified, real-time object detection*. arXiv.org.
- [23] Shafiee, M. J., Chywl, B., Li, F., Wong, A. (2017, September 18). *Fast yolo: A fast you only look once system for real-time embedded object detection in video*. arXiv.org.
- [24] Redmon, J., Farhadi, A. (2016, December 25). *Yolo9000: Better, faster, stronger*. arXiv.org.
- [25] Redmon, J., Farhadi, A. (2018, April 8). *Yolov3: An incremental improvement*. arXiv.org.
- [26] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., Berg, A. C. (2016, December 29). *SSD: Single shot multibox detector*. arXiv.org.
- [27] Jianbo Shi and J. Malik, *Normalized cuts and image segmentation*, in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 8, pp. 888-905, Aug. 2000.
- [28] C. Ruizhongtai Qi, H. Su, K. Mo, and L. J. Guibas, *Pointnet: Deep learning on point sets for 3d classification and segmentation*, in Proc. IEEE Int. Conf. Comput. Vis. Pattern Recogn., 2017.
- [29] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, *Pointpillars: Fast encoders for object detection from point clouds*, CoRR, vol. abs/1812.05784, 2018.
-

-
- [30] Rong, G. et al. (2020, June 22). *LGSVL simulator: A high fidelity simulator for autonomous driving*. arXiv.org.
- [31] <https://github.com/ApolloAuto/apollo>
- [32] <https://www.svlsimulator.com/docs/simulation-content/add-new-ego-vehicle/>
- [33] <https://www.svlsimulator.com/docs/simulation-content/add-new-map/>
- [34] <https://www.svlsimulator.com/docs/simulation-content/map-annotation/>
- [35] <https://www.svlsimulator.com/docs/>
- [36] S. Schneider, M. Himmelsbach, T. Luettel and H. Wuensche, *Fusing vision and LIDAR-synchronization, correction and occlusion reasoning*, 2010 IEEE Intelligent Vehicles Symposium, pp. 388-393, 2010.
- [37] K. Peterson, J. Ziglar and P. E. Rybski, *Fast feature detection and stochastic parameter estimation of road shape using multiple LIDAR* 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 612-619, 2008.
- [38] M. Sualeh and G.-W. Kim, *Dynamic Multi-LiDAR Based Multiple Object Detection and Tracking Sensors*, vol. 19, p. 1474, 2019.
- [39] B. Fortin, R. Lherbier and J. C. Noyer, *A Track-Before-Detect Approach for Extended Target Tracking in Multi-Lidar Systems using a Low-Level Centralized Fusion Framework*, 2014.