Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN
INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

Predictive Techniques for Scene Understanding by using Deep Learning

Autor: Carlos Gómez Huélamo

Tutor/es: Luis Miguel Bergasa Pascual

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA INDUSTRIAL

Trabajo de Fin de Máster

“Predictive Techniques for Scene Understanding by using Deep Learning”

Carlos Gómez Huélamo

2019

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN
INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

"Predictive Techniques for Scene Understanding by using Deep-Learning"

Autor: Carlos Gómez Huélamo

Tutor/es: Luis Miguel Bergasa Pascual

TRIBUNAL:

Presidente: D. Daniel Pizarro Pérez

Vocal 1º: D. Rafael Barea Navarro

Vocal 2º: Luis Miguel Bergasa Pascual

Calificación: ……

Fecha: 30/09/2019

"En este vasto mundo
navegáis en pos de un sueño,
surcando el ancho mar
que se extiende frente a vosotros.

El puerto de destino es el mañana
cada día más incierto.

Encontrad el camino,
cumplid vuestros sueños,
estáis todos en el mismo barco
y vuestra bandera es la libertad"

Espero que te guste, allá donde estés …

ACKNOWLEDGEMENTS

Este Trabajo de Fin de Máster supone el culmen a dos años realmente duros, cargado de emociones, triunfos y tropiezos a partes iguales. Al igual que cuando realicé mi Trabajo de Fin de Grado, mantengo mi filosofía de aprendizaje continuo, formación continua, que es lo que más me gusta, para así cada día entender el mundo un poquito mejor. Si toda la dedicación y estudio que he depositado en este trabajo sirven para algo en mi futuro, sé que todo el esfuerzo habrá merecido la pena.

En primer lugar, me gustaría agradecer a mi tutor Luis Miguel Bergasa Pascual por ofrecerme un tema desconocido para mí hasta hace apenas unos meses y que, sin embargo, considero ahora mismo realmente interesante y con mucho futuro como es la aplicación de técnicas de aprendizaje profundo (Deep Learning) al seguimiento de objetos. Habrá que seguir esta tendencia muy de cerca en los próximos años. A mis compañeros, mejor dicho, amigos, de laboratorio: Edu, los Javis, Óscar, Álvaro, Miguel y Felipín por todas las risas que tenemos juntos y por toda la ayuda y consejos que me han prestado durante estos meses en los que ha durado el TFM. Este trabajo también es vuestro. A mis profesores del grupo RobeSafe, al que con mucho orgullo pertenezco, Rafa, Elena y Pedro, por tantas lecciones aprendidas de ellos.

A mis amigos de la universidad, especialmente a Antonio, Alejandro, Cristina, Esther, Rocío, Juan Carlos, Sergio Rodríguez, Sergio Pérez, Adri, Rubén, Pablo, Jesús y Ramón. Aún me acuerdo de cuando empezamos con la carrera y como ahora la vida nos va perfilando poco a poco a cada uno. Os deseo lo mejor en vuestro futuro.

A mi buen amigo Samuel, con quien gran parte de mi vida he compartido. Con especial cariño guardo las conversaciones después de entrenar, siempre sabiendo lidiar con los problemas que te comentaba.

A mis amigos del pueblo, Rodri, Jorge, Iván, Alberto, Diego, Dani, Luis y Rubén porque sin saberlo, muchas veces me he relajado con ellos cantando un par de canciones en una verbena de pueblo volviendo a casa más contento para afrontar más contento mi día a día.

A mi familia, uno de los pilares de mi vida. A mi padre Juan Antonio y a mi madre Petra, que en paz descanse, les debo todo lo que soy y es por ello por lo que les estaré siempre agradecido. A mi querida hermanita Silvia, con quien tantas regañinas he tenido, pero el cariño que nos tenemos las supera a todas. A mi perrita Nuka, que sin saberlo me ha despejado muchísimas veces sacándola a dar un paseo, dando rienda suelta a mi cabeza para imaginar nuevas propuestas mientras miraba el cielo azul. Al resto de la familia, gracias por todo.

Y, por último, la persona más importante de mi vida ahora mismo. Mi querida Marta, la persona que en tan poco tiempo me has dado tanto. En los momentos buenos, en los no tan buenos, en

las risas y en los lloros, siempre estás ahí conmigo. Eres maravillosa y deseo pasar mi vida contigo. Mi corazón siempre estará con el tuyo. Te quiero.

CONTENTS

List of Figures

List of Tables

List of Acronyms

1. **CNN**: Convolutional Neural Network
2. **RNN:** Recurrent Neural Network
3. **MOT:** Multi-Object Tracking
4. **SOT:** Single Object Tracking
5. **VOT:** Visual Object Tracking
6. **AI:** Artificial Intelligence
7. **ML:** Machine Learning
8. **DL:** Deep Learning
9. **LiDAR:** Light Detection and Ranging
10. **BEV:** Bird Eye View
11. **IoU:** Intersection over Union
12. **GPS:** Global Positioning System
13. **SORT:** Simple Online and Real Tracking
14. **CARLA:** Car Learning to Act
15. **ROS:** Robot Operating System
16. **PCL:** Point Cloud Library
17. **MOTA:** Multi-Object Tracking Accuracy
18. **MOTP:** Multi-Object Tracking Precision

Code of Interest

Key Concepts

1. **Object tracking:** Process of locking on to a moving object, being able to determine if the object is the same as the one present in the previous frame.

2. **Sensor fusion:** Sensor data combination or data derived from different sources such that the resulting information has less uncertainty than would be possible when the sources are used individually.

3. **CNN (Convolutional Neural Network):** Kind of Artificial Neural Network specialized in image classification or segmentation.

4. **Groundtruth:** In the context of detection and tracking, it refers to the exact position, velocity or any required variable of a determined object.

5. **BEV (Bird Eye View):** Elevated view of an object from above, with a perspective as the observer were a bird, very used in the context of self-driving applications (filtering the Z-axis for the objects).

6. **Kalman filter:** Algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, producing the estimation of unknown variables that are expected to be more accurate than those based on a single measurement.

7. **SmartElderlyCar:** Autonomous electric vehicle able to drive in the campus of the University of Alcalá.

8. **ROS (Robot Operating System):** Software framework for robot software development.

9. **PCL (Point Cloud Library):** Standalone, large scale, open project for 2D/3D image and point cloud processing.

10. **Docker:** Open source project that offers a software development solution known as containers

11. **CARLA (Car Learning to act):** Open source simulator for urban driving based on the simulation engine Unreal Engine 4.

12. **MOTA (Multiple Object Tracking Accuracy):** MOT metric that combines three error types: missed targets (M_t), false positive (f_{p_t}), and the identity switches or mismatches mm_t, respectively at frame t.

13. **MOTP (Multiple Object Tracking Precision):** Total position error for matched object-hypothesis pairs over all frames, averaged by the total number of matches mode.

Resumen

El presente trabajo propone una arquitectura software precisa y en tiempo real para el seguimiento de múltiples objetos basada en aprendizaje profundo (*Deep Learning*) en el contexto de la navegación autónoma. Se ha llevado a cabo una fusión sensorial entre el seguimiento visual 2D basado en los algoritmos CenterNet y Deep SORT [2] [49] usando una cámara y el clusterizado de la nube de puntos 3D procedent del LiDAR [11] sobre la plataforma de desarrollo robótico ROS y contenedores Docker.

Se ha llevado a cabo una comparación entre el enfoque tradicional Precision-Tracking [46], tracking visual basado en *Deep Learning* y fusión sensorial con LiDAR comparando las posiciones estimadas para cada uno de ellos.

Las propuestas han sido validadas en el *benchmark* de KITTI para seguimiento de vehículos [69], en el simulador de CARLA [31] para el seguimiento de peatones y en el campus de la Universidad de Alcalá sobre nuestro vehículo autónomo desarrollado en el proyecto *SmartElderlyCar*.

Palabras clave: Seguimiento de múltiples objetos, *Deep Learning*, ROS, CARLA, *SmartElderlyCar*.

Abstract

The present work proposes an accurate and real-time Deep Learning based Multi-Object Tracking architecture in the context of self-driving applications. A sensor fusion is performed merging 2D visual tracking based on CenterNet and Deep SORT algorithms [2] [49] using a camera, and 3D proposals using LiDAR point cloud [11] over the ROS framework and Docker containers.

A comparison between the traditional Precision-Tracking [46] strategy, Visual Object Tracking based on deep learning and sensor fusion approach with LiDAR is carried out comparing the obtained pose estimations for each of them.

The proposals have been validated on KITTI benchmark dataset for vehicle tracking [69], on CARLA simulator [31] for pedestrian tracking and on the Campus of the University of Alcalá using our autonomous vehicle developed for the SmartElderlyCar project.

Keywords: Multi-Object Tracking, Deep Learning, ROS, CARLA, SmartElderlyCar.

Extended Abstract

One of the most critical aspects when developing an *autonomous vehicle* is to perceive and understand the dynamic scene as accurately as possible, so that the actions taken by the vehicle guarantee the safety of both the vehicle itself and the surrounding obstacles, such as people, other vehicles or road infrastructure.

In that sense, the object detection and tracking strategies play a fundamental role, allowing the vehicle to predict future positions, as well as plan its possible trajectories dynamically, based on previous frames in the scene.

Although there has been great progress in the *object detection* and *tracking* methods, object tracking remains a problem of interest due to the challenges present in real-world applications, such as occlusions, changes in the point-of-view or lighting. In that sense, the scientific community is continually developing robust object detection and tracking approaches.

Moreover, the Deep Learning paradigm has supposed a breakthrough for the research community, giving rise to rapid advances overall in terms of object detection. Recently, more and more tracking algorithms, both single and multiple, have started exploiting the representational power of deep learning. The strength of *Deep Neural Networks* (DNNs) resides in their ability to learn rich representations and extract complex and abstract features from their input, generally an image. *Convolutional Neural Networks* (CNNs) currently constitute the state-of-the-art in spatial pattern extraction, employed in tasks such as image classification or object detection. Then, since *Deep Learning (DL)* methods have been able to reach top performance in many of those tasks, the research community is now progressively introducing DL in most of the top performing tracking algorithms.

In this work, a Deep Learning based Multi-Object Tracking is presented in the context of self-driving applications. This architecture is divided in two branches: The first branch performs the object tracking using camera data, so tracking is performed in 2D, using a state-of-the-art object detector (*CenterNet* [49]) and a efficient tracking algorithm with deep appearance descriptor (*Deep SORT* [2]) in order to avoid above mentioned tracking challenges as occlusions or changes in the point-of-view. Then, these 2D proposals are projected onto the *Bird Eye View* (BEV) space using projection matrices. The second branch focuses on performing the clustering of the 3D point cloud (LiDAR data) in order to obtain the most relevant objects in the environment and project them in the BEV. Both proposals are fused to get the best positions of the detected object bounding boxes over time.

Finally, in order to evaluate the architecture proposal performance, some tests are carried out in KITTI benchmark [69], CARLA simulator [31] and our Campus using our real autonomous vehicle, comparing the BEV pose using the Precision-Tracking [46], Visual Object Tracking (VOT) based on deep learning and sensor fusion (Merged VOT) approaches.

Chapter 1. Introduction

1.1. Motivation

Autonomous Vehicles (AVs) have held the attention of technology enthusiasts and futurists for some time as evidenced by the continuous development and research in *Autonomous Vehicle Technologies (AVT)* over the past two decades, being one of the emerging technologies of the Fourth Industrial Revolution, and particularly of the Industry 4.0.

The phrase Fourth Industrial Revolution was first introduced by Klaus Schwab, CEO (Chief Executive Officer) of the World Economic Forum, in a 2015 article in Foreign Affairs (American magazine of international relations and United States foreign policy). A technological revolution is defined as a period in which one or more technologies are replaced by other kinds of technologies in a short amount of time. Hence, it is an era of accelerate technological progress featured by Researching, Development and Innovation whose rapid application and diffusion cause an abrupt change in society. In particular, the Fourth Industrial Revolution is expected to be marked by breakthroughs in emerging technologies in fields such as Artificial Intelligence (AI), Computer Vision, Internet of Things (IoT), fifth-generation wireless technologies (5G), Robotics, 3D printing and the scope of this master thesis, fully autonomous vehicles. The sum of all these advances are resulting in machines that can potentially see, hear and what is more important, think, moving more deftly than humans.

Moreover, *Industry 4.0* is the subset of the Fourth Industrial Revolution that concerns industry, that is, this concept focuses the existence of factories in which machines are enhanced with sensors and wireless connectivity, connected to a system that can visualize the whole production line and make decision on its own. In fact, if it is substituted "the whole production" by the environment, the concept refers to a self-driving car.

A self-driving car (also known as driverless car or autonomous car) is a vehicle that can sense its environment and moving safely with little or even no human input. They combine a variety of sensors to recognize their environments, such as *GPS*, *camera*, *Inertial Measurements Units (IMUs), radar, sonar or LiDAR (Light Detection and Ranging).* Then, advanced control systems process this sensory information in order to calculate in a proper way navigation paths, traffic signs or detect and track the road obstacles (which is the main purpose of this thesis) to ensure a safe driving.

Furthermore, statistics show that 69 % of the population in the European Union (EU), including associated states, lives in urban areas. According to the World Health Organization, nearly one third of the will live in cities by 2030. Aware of this problem, the Transport White Paper published by the European Commission in 2011 indicated that new forms of mobility ought to be proposed so as to provide sustainable solutions for people and

goods safely. For example, regarding safety, it sets the ambitious goal of halving the overall number of read deaths in the EU between 2010 and 2020. Nevertheless, this goal does not seem to be easy since only in 2014 more than 25,700 people died on the roads in the EU, many of them caused by an improper behaviour of the driver on the road.

Autonomous driving is considered as one of the solutions to the before mentioned problems and one of the greatest challenges of the automotive industry today. The existence of reliable and economically affordable autonomous vehicles will create a huge impact on society affecting social, demographic, environmental and economic aspects. Besides this, it is estimated to cause a reduction in road deaths, reduce fuel consumption and harmful emission associated and improve traffic flow, as well as an improvement in the overall driver comfort and mobility in groups with impaired faculties, such as disable or elderly people. Other industrial applications of autonomous vehicles are agriculture, retail, manufacturing, commercial and freight transport or mining.

1.2. Historical context

Autonomous Vehicles in the form of self-driving cars have become a challenge for auto competitions and technology companies, which has derived in an intense competition. Nevertheless, the AVs are not new.

The study of Automated Driving Systems (ADS) was started in the 1920s, though trials began in the 1950s. The first semi-automated car was developed in 1977 by Japan's Tsukuba Mechanical Engineering Laboratory. The vehicle reached speeds up to 30 km/h with the support of an elevated rail.

Nevertheless, the first truly autonomous cars appeared in the 1980s with Carnegie Mellon University's Navlab and ALV projects funded by the American company DARPA (Defense Advanced Research Projects Agency) in 1984 and EUREKA Prometheus project (1987) developed by Mercedes-Benz and Bundeswehr University Munich's. By 1985, the ALV project had shown self-driving speeds on two-lane roads with obstacle avoidance added in 1986 and off-road driving in day and night conditions by 1987. Furthermore, from the 1960s through the second DARPA Gran Challenge in 2005 (212 km off-road course near the California-Nevada state line, surpassed by all but one of the 23 finalists), automated vehicle research in the United States was primarily funded by DARPA, the US Army and US Navy, yielding rapid advances in terms of speed, car control, sensor systems and driving competence in more complex conditions. This caused a boost in the development of autonomous prototypes by companies and research organizations, most of them Americans.

Figure 1.2-1 Self-driving SUV from Carnegie Mellon University's Tartan Racing team

Figure 1.2-1 shows the Self-driving SUB from Carnegie Mellon University's Tartan Racing team during the 10th anniversary celebrations of Tartan Racing's victory in the 2007 DARPA Urban Challenge.

Even though self-driving cars have not yet displaced conventional cars, there can be found several examples of how it has become a hot topic for powerful companies such as Delphi Automotive Systems, Audi, BMW, Tesla, Mercedes-Benz or Waymo.

In 2005 Delphi broke the Navlab's record achievement (driving 4,584 km while remaining 98 % of the time autonomously) by piloting an Audi, improved with Delphi technology, over 5,472 km through 15 states while remaining in self-driving mode 99 % of the time. Moreover, this year the US states of Michigan, Virginia, California, Florida, Nevada and the capital of the United States, Washington D.C., allowed the testing of automated cars on public roads.

In 2017, Audi stated that its A8 car prototype would be automated at speeds up to 60 km/h by using its perception system named "Audi AI". Also, in 2017 Waymo (self-driving technology development company subsidiary of Alphabet Inc) started a limited trial of a self-driving taxi service in Phoenix, Arizona.

Figure 1.2-2 shows the distance between disengagements and total distance travelled in California (2018) by the most important self-driving technology development companies in the world. A disengagement may be defined as the deactivation of the autonomous mode when a failure of the autonomous technology is detected or when a safe operation requires than the autonomous vehicle test driver disengages the autonomous mode and takes immediate manual control of the vehicle.

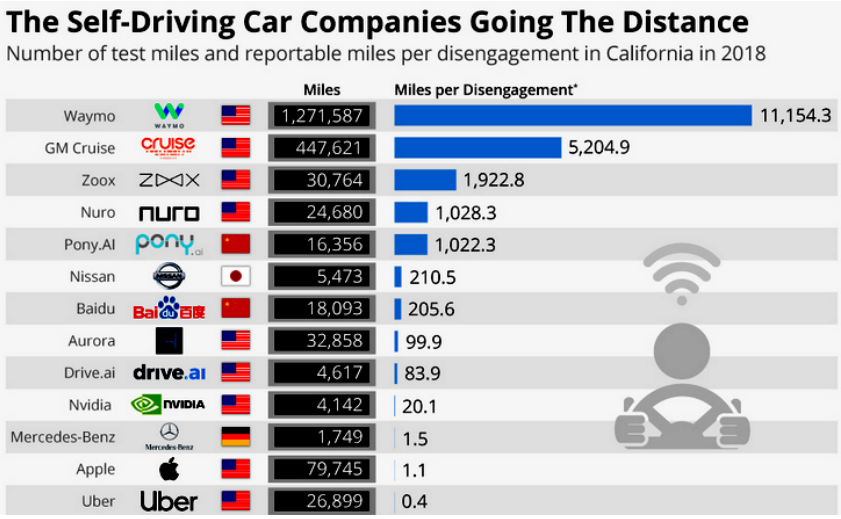


Figure 1.2-2 Number of test miles and reportable miles per disengagement in California (2018)

1.3. Problem statement

To sum up what commented above, increasing the level of autonomous navigation in mobile robots (from agriculture to public and private transport) creates tangible business benefits to those users and companies employing them. However, designing an autonomous navigation system does not seem to be an easy task. In that sense, it can be found mainly five challenges when designing an autonomous navigation system:

1. Gathering enough Real-World data: Functional autonomous navigation systems cannot be developed exclusively in a lab (unless they are designed for that purpose). The reason is simple: A prototype will not work in a proper way until it has been in a certain number of different scenarios since the problems that it is going to experience in the wild cannot be replicated in a lab, such as a pedestrian crossing the road through a non-allowed space or maintenance works on the road. Essentially, this challenge deals with huge amounts of different data; The more edge cases the prototype can solve, the better the navigation solution.

2. Installation must be simple: If a company aims a genuinely scalable product, the installation process must not be technical but simple. Despite the fact that nowadays many robots require an engineer for installation and validation into a new environment, if the autonomous machines aim to replace conventional cars, they should show a user interface intuitive and lean, dealing with non-technical employees who want to use these machines, i.e. designing a system that is easy to use.

3. Reducing False Positives: This challenge is mostly related with detection. In the same way a human being can identify if a certain object is a pedestrian, a cyclist or a car on the road, if the robot cannot tell a person from a chair on the floor, the decision-making layer of the

floor will be affected by these false positives. In that sense, it is critical that the prototype is trained to detect and track using sensor data from both virtual environments and real-world scenarios to reduce these false positives. Figure 1.3-1 shows an example of false positives detection in a real-world situation.

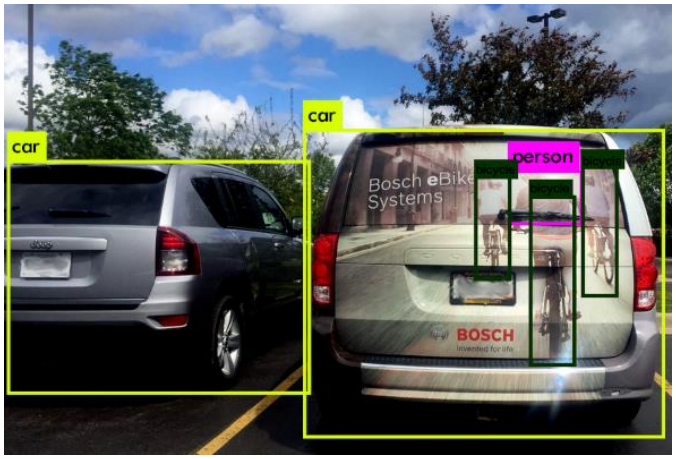


Figure 1.3-1 False positives detection in a real-world situation

4. Getting the right Software (SW) and Hardware (HW): Getting the right HW and SW is specially challenging for robots that will be autonomously navigating in dynamic environments like high-traffic workspaces, malls or airports. Since these areas present tight spaces and continuously changing obstacles, the use of suitable HW (odometry with a great accuracy for positioning and navigating the complex routes, cameras to identify and classify the objects or LiDAR to create a 3D scene of the environment) and SW (writing software that handles these challenging situations with and end-user mind, also known as decision-making layer in the prototypes) seems to be a mandatory task.

5. Creating precision Motion-Control: The last challenge illustrates the design an autonomous navigation for the mobile robot creating a system that has accurate and precise motion control. In that sense, this task must deal with situations as the need of driving as close as possible to an edge, wall or an obstacle. In that sense, designing a system that is as accurate and tight as possible gives rise to much better capabilities to navigate in complex spaces.

As shown, the design of a reliable architecture for an autonomous robot seems to be a really hard task. In that sense, this master thesis is mainly related with the challenge *1. Gathering enough Real-World data, 3. Reducing False Positives and 4. Getting the right Software (SW) and Hardware (HW)* in order to perform an accurate perception of the environment and consider the odometry, velocity and predict future actions associated the obstacles identified in the scene, also known as object tracking.

While detecting objects in a scene has been getting a lot of attention from the computer vision and robotics community, a lesser known and yet an area with widespread

applications is object tracking in a real (or virtual) scene, both in an off-line video or in real-time situations, as can occur with autonomous vehicles.

Object tracking is the process of locating a single or multiple moving object over time using a given device (generally a camera or LiDAR) that detects the environment. It is an important part of a human-computer collaboration in a moving environment in terms of allowing the computer to obtain a better model of the simulation or real-world. Object tracking requires to merge the detection of the objects in the scene (frame by frame, i.e., in static images) analysing temporal information in order to get the best predicted trajectories. It is important to consider the difference between Object Detection and Object Tracking. In object detection, the purpose is to detect an object in a single frame and tie this one with a mask around or a bounding box, in addition to classify the object (i.e., obtain the position of the object and its semantic information in the scene). However, detection process ends at this point since it processes each frame independently and identifies several objects in that frame. On the other hand, an object tracker must track a particular object across the entire scene. For example, if the detector detects two cyclists and three cars. In that sense, the object tracking must identify the five separate detections and needs to track them across the subsequent frames associating a unique ID to each object.

Object tracking presents a wide range of applications [5] in computer vision and associated disciplines such as human computer interaction, traffic flow monitoring, surveillance or human activity recognition.

1.4. Tracking foundations

Today, most tracking approaches are based on traditional techniques. However, deep learning is gaining weight and new approaches are emerging using this discipline. Although each object tracking method (both traditional or Deep-Learning based) is characterized by its particular features, most of them can be classified according to four criteria:

1.4.1. Detection-based vs Detection-free trackers

Detection-free tracking: It requires a manual initialization of fixed number of objects in the first frame of the scene. Then, it localizes these objects (but not others) in the subsequent frames, so it cannot deal with the case where new objects appear in the middle frames.

Detection-based tracking: The consecutive video frames are given to a pretrained object detector that gives rise to a detection hypothesis which in turn is used to elaborate tracking trajectories. It is more popular than detection free tracking because new objects are detected and tracked whilst disappearing objects are terminated automatically. In this approach the tracker is used in those cases when object detection algorithm fails. Other alternatives of detection-based tracking are to run the object detector for every *n* frame and the remaining predictions are done using the tracker. For that reason, this approach is very suitable for tracking for a long time.

1.4.2. Single vs Multiple Object trackers

Single Object Tracking (SOT): In this approach a single object is tracked even if the scene presents multiple objects in it. The target object to be tracked is determined by the initialization in the first frame.

Multiple Object Tracking (MOT): All the relevant objects present in the scene are tracked across the frames. If a detection-based tracker is used for MOT it can even track new objects that appear in the middle frames of the process. Note how Figure 1.4-1 shows two subsequent frames with multiple objects. As explained, their identification must be kept until they come from the scene.

Figure 1.4-1 Multiple Object Tracking example using Deep Sort

1.4.3. Online vs Offline trackers

Offline trackers: This kind of trackers is used when the process require to track an object in a recorded stream. For that reason, it can be used both the past and future frames (because the sequence is already recorded) to elaborate more accurate tracking predictions (no causal systems). For example, in the case of basketball teams that have recorded videos of a match of an opponent team which needs to be analysed for strategic analysis or if it is required to analyse *rosbags* (file format in ROS for storing ROS message data) recorded for a robot to check HW/SW implementation.

Online trackers: This kind of trackers cannot use future frames to improve the tracking predictions results (causal systems). Hence, they are used where predictions must be available immediately.

1.4.4. Online vs Offline strategy

Only learning trackers: These trackers usually learn about the object to be tracked by using the initialization frame and few subsequent frames. These trackers are more general since the algorithm just draws a bounding box around the object and track it. The tracker would learn about that target object using these consecutive frames and would continue to track it.

Offline learning trackers: Whilst online learning trackers learnt about the objects across the frames, offline learning trackers do not learn anything during run time. In other words, they must be trained and improve offline, but they will not improve themselves while they are tracking the objects in the scene.

1.5. Objectives of this work

Most of the current tracking systems are based on traditional techniques. However, the main scope of this work is to study the state-of-the-art of Deep Learning based Multi-Object Tracking and implement and validate an optimal architecture both in simulation and real world. It is a hard issue due to the lack of literature in this specific branch of deep-learning based object tracking applications, as shown in [2] [3] [4]. Moreover, in order to achieve the main scope, the following objectives will be met:

1. Researching of current Deep-Learning based Multi-Object Tracking approaches.

2. Study of state-of-the-art software technologies and sensors to perform the MOT problem both in simulation and real-world.

3. Explanation of a real-world project named SmartElderlyCar, including its devices and hardware and software architecture.

4. Propose an architecture for Deep-Learning based Multi-Object Tracking.

5. Validate the proposed architecture for MOT both in CARLA simulator and real world.

1.6. Structure of this work

The organization of this document has been done as follows:

- *Chapter 2* presents a technical background about current object tracking approaches, including Visual Object Tracking, LiDAR based and sensor fusion. Then, as this master thesis focuses of 2D tracking, challenges in Visual Object Tracking are shown. Then, Deep Learning in MOT is studied in addition to some state-of-the-art approaches.

- *Chapter 3* focuses on the software technologies used in this master thesis, that is, ROS for sensor communication, PCL as point cloud processing, Docker as a tool to increase the portability and testability of the project and CARLA as simulator environment.

- *Chapter 4* presents the SmartElderlyCar project, an autonomous electric car able to drive in the University of Alcalá campus, as the reference to develop the architecture proposal of this work.

- *Chapter 5* shows the Deep Learning based Multi-Object Tracking architecture proposal. This is the main chapter of this master thesis.

- *Chapter 6* shows the validation of the architecture proposal in the KITTI benchmark as well as quantitative and qualitative results both in CARLA simulator, KITTI benchmark and in the real prototype of the SmartElderlyCar.

- *Chapter 7* illustrates the conclusions and future works of this project.

- *Appendix A* details how Kalman filter works in the object tracking context.

- *Appendix B* shows the Artificial Intelligence paradigm, including Machine Learning and Deep Learning concepts. In addition, a brief explanation of how CNNs and RNNs work due to its tight relation with object detection and tracking.

- *Appendix C* shows some interesting parts of the code created and developed in order to perform most of exposed tasks throughout this master thesis.

- *Appendix D* illustrates the user's manual so as to install the system requirements and reproduce the obtained results.

- *Appendix E* represents the main hardware and software specifications used in this project.

- *Appendix F* illustrated an estimation of the required budget to develop this thesis.

Chapter 2. Technical background and State-of-the-Art

2.1. Introduction

As commented in Chapter 1, even though object tracking is a well-studied problem within the area of traditional image processing, it is still considered a complex problem to solve even more considering current applications such as robot navigation, intelligent video surveillance or self-driving. The broad area of application shows how importance of reliable, exact and effective object tracking must be nowadays.

An object tracking system presents three stages, like object *modelling* and *segmentation*, *object location* in each frame and *object prediction*. In order to solve the initial detection and so the following stages three different approaches can be used.

2.1.1. Visual Object Tracking (VOT)

Visual Object Tracking is the process of determining the target object location in a sequence (i.e., in a video sequence or a real-world situation). It is one of the many remarkable topics in computer vision that has gained considerable weight over the past decade. In VOT approach the information of the scene is only captured by cameras, taking advantage of video technologies, such as low cost, great portability and considerable speed. There are mainly three traditional types [7] to perform VOT:

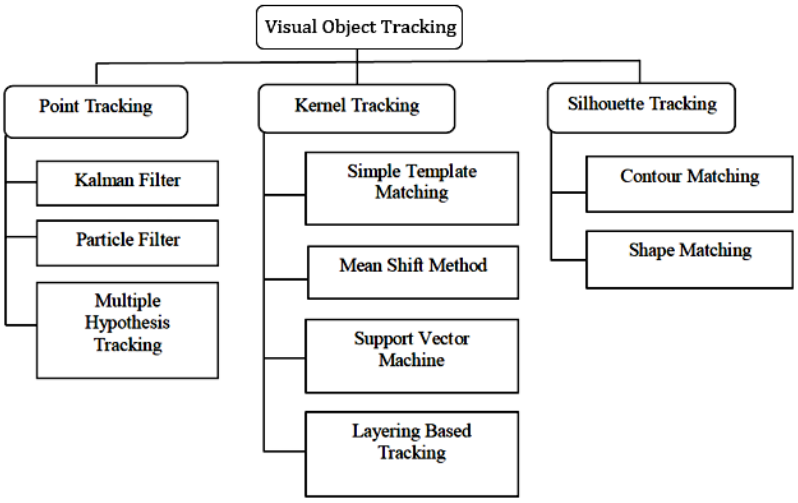


Figure 2.1-1 Traditional techniques to perform Visual Object Tracking

1. *Point tracking approach:* In an image structure, moving objects are represented by their feature points during tracking. In that sense, point tracking is a complex problem specially in those cases that present occlusions, false detection or background cluttering. It this technique the recognition is carried out relatively simple, by thresholding and then identifying these points of interest.

2. *Kernel Based tracking approach [8]:* This approach of VOT is usually performed by computing the moving object that is represented by an envelope object region, from one frame to the next. Then, the object motion is usually in the form of parametric motion such as rotation, affine or translation. Different algorithms of kernel-based tracking diverge in terms of the presence representation used, the number of objects to be track and the method used for approximation to object motion.

3. *Silhouette based tracking approach:* This approach focuses on problems of VOT related with the geometry of the target objects. Some object will have complex shape such as fingers, shoulders or hands which cannot be well defined by simple geometric shapes. Silhouette based methods [8] address this problem with an accurate shape description for the objects. In that sense, the aim of a silhouette-based object tracking is to find the object region in every frame by means of an object model generated by the previous frame. Then, these algorithms are able of handling with object occlusion, variety of object shapes or even object split.

Moreover, some VTO based works focus on obtaining the shape and appearance of the target object not considering the background [9] or even a performing a visual decomposition model [10]. Figure 2.1-2 shows an example of visual tracking decomposition, successfully tracking a target even though there are severe pose variations, occlusion, abrupt motions and illumination changes at the same time.

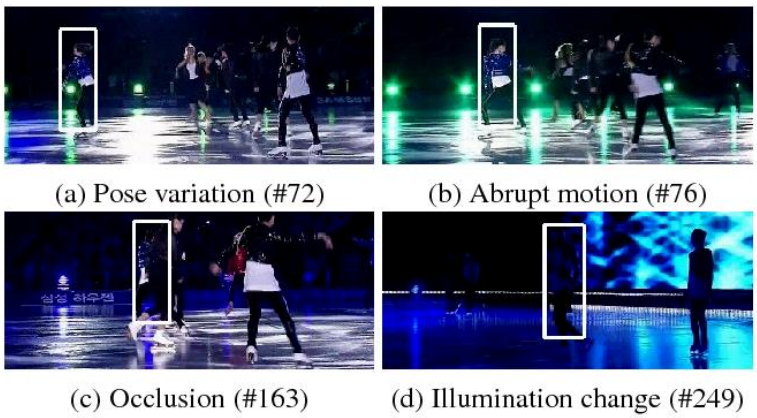


Figure 2.1-2 Visual Tracking Decomposition

2.1.2. Only LiDAR based object tracking

Another approach for object tracking is LiDAR based [11]. Real-world driving scenarios are very dynamic and complex on their own. In this sense, a sensor that directly provide 3D

information is more interesting than those that obtain depth in a recovery process based on a priori calibration. In addition, the appearance of those scenarios can be greatly affected by some issues such as context factors (urban, highway, road, etc.), meteorological conditions (rain, fog, snow, etc.) or day-time (sunrise, night, sunset, etc.). In that sense, optical cameras are likely to fail perceiving correctly the environment in certain conditions, other robust sensors are required.

LiDAR technology is described in Chapter 4. This technology provides the desired robustness and precision under harsh conditions [12], that make LiDAR technology suit for self-driving applications. LiDAR point clouds have been traditionally processed following geometrical approaches like in [13], where clustering algorithms are used to segment the data and then assigning the resulting groups to different classes. Other strategies benefit from prior knowledge of the environment structure to ease the object segmentation and clustering [14]. 3D voxels (volumetric pixels) can be also created in order to reduce computational costs by grouping sets of neighbour points. Furthermore, more recent methods are able to process the point cloud space (raw or reduced in voxels) so as to extract hand-crafted features such as shape models or geometric statistics. For example, [15] uses this second approach (extracting hand-crafted features) and then encodes the sparse LiDAR point cloud with different features. The resulting representation is scanned by using a 3D sliding window of different sizes and an SVM (Support Vector Machine) followed by a voting scheme is used to classify the final candidate windows, as shown in Figure 2.1-3.

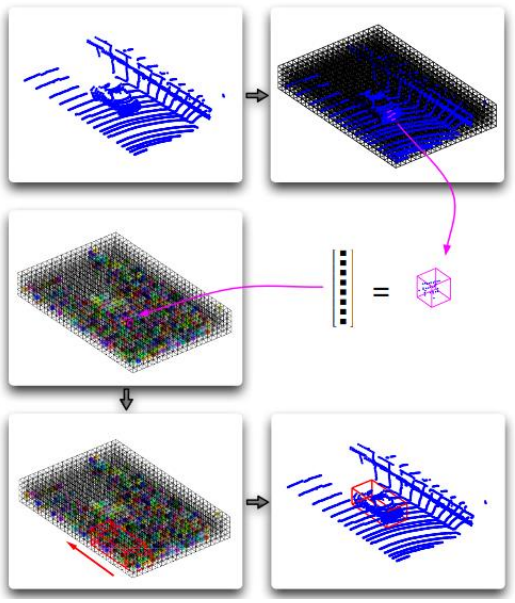


Figure 2.1-3 Example of voting in online point cloud object detection

Other solutions are not based on models but they detect and search for dynamic objects using LiDAR which are then segmented using a Bayesian approach (sequence of alternations between prediction and update or correction) [16], as illustrated in Figure 2.1-4:

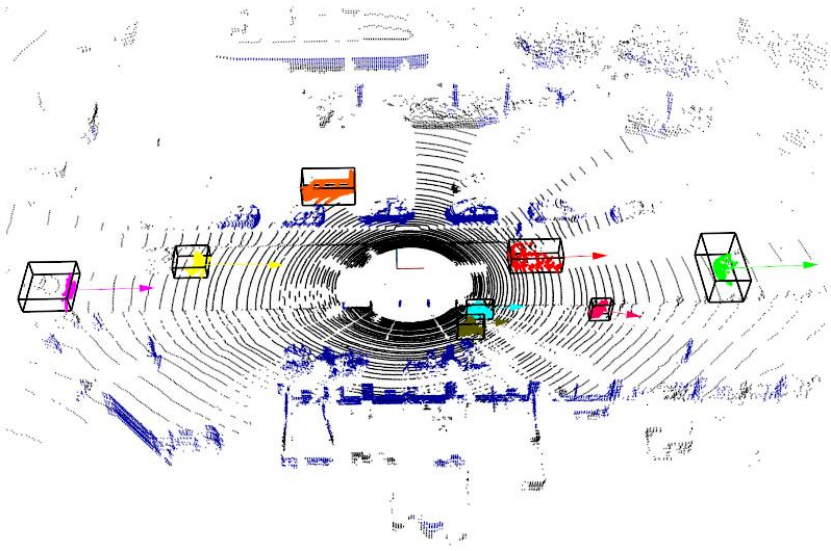


Figure 2.1-4 Object detection and tracking based on dynamic search and Bayesian segmentation

2.1.3. Sensor fusion based object tracking

The third main approach in order to track objects, in particular for self-driving applications, is the fusion of different sensors such as camera, LiDAR, radar, IMU or GPS, as shown in Figure 2.1-5.

Figure 2.1-5 Different sensors in a self-driving car

Chapter 4 offers a more detailed explanation for the sensors used in this master thesis, both in real world and simulation. As shown in Figure 2.1-6, each of these sensors has advantages and disadvantages. The aim of sensor fusion is to use the advantages of each sensor so as to

understand its environment in an accurate way. For example, one of the challenges when using LiDAR is the relatively small vertical field of view and angular resolution (30 to 41.3 º and 1.33 to 2 º for Velodyne LiDAR), giving rise to a small number of points of the object to be tracked. While camera is a very good sensor for detecting roads, recognizing the semantic information of a target object (car, pedestrian or cyclist, for example) or reading signs, the LiDAR technology is better at accurately estimating the position of the object and radar is better at accurately estimating its speed.

Figure 2.1-6 Sensor benchmark

[17] presents an approach for fusing distance data gathered by a LiDAR (in the form of a 3D point cloud) with the luminance data from a wide-angle imaging sensor. [18] presents its tracking system (Figure 2.1-7) as a combination between a fusion layer and sensor layer, showing results as illustrated in Figure 2.1-8. As demonstrated in this work, the newly introduced vision targets are really useful to improve the performance of data association and movement classification for measurements from active sensors.

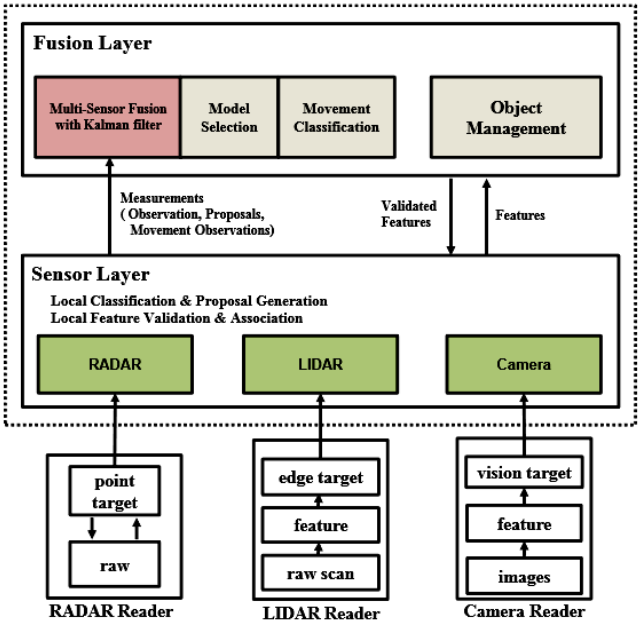


Figure 2.1-7 Sensor fusion based tracking system architecture proposed in [18]

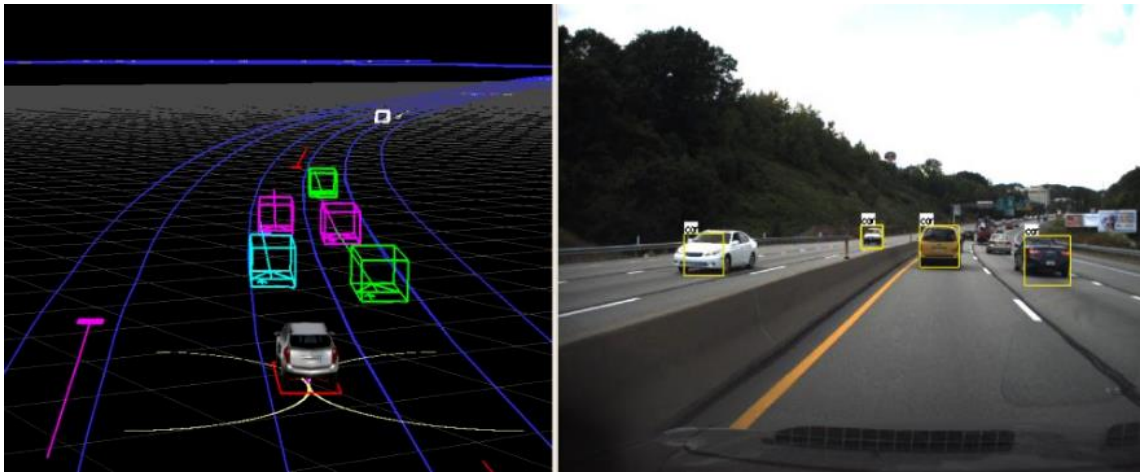


Figure 2.1-8 Tracking results of [18] for the qualitative evaluation

To finish the introduction of this chapter, it can be concluded that the object tracking is a complex problem even more when speaking about self-driving applications. As commented in previous sections, this master thesis focuses on deep learning based multi-object tracking by using image data, due to data and code for the learning process are mainly available for images, our approach carries out 2D tracking over images and then recovering 3D poses of the surrounding objects for our real autonomous driving application. For that reason, the next section covers the main challenges associated with Visual Object Tracking to take into account how the incorporation of deep learning approaches has leveraged this technique.

2.2. Challenges in Visual Object Tracking:

Some of the fundamental problems in VOT are abrupt object motion, noise in the image sequences, changes in scene illumination, object-to-object and object-to-scene occlusions, changing appearance patterns of the object and the scene, non-rigid object structures and real time processing requirements, which is getting more importance with the presence of self-driving applications. [19] In that sense, there are several important challenges that must be considered in the design of a robust multi-object tracking system:

1. *Object modelling:* One of the major tasks in VOT is to find an appropriate visual description which makes the object distinguished from other objects and background. The development of Deep-Learning and CNNs (Convolutional Neural Networks) has meant a revolution in the task of object modelling and detection.

2. *Changes in shape and appearance:* Both changes are really important to be considered during VOT. The appearance of an object may vary as camera angle changes. Deformable objects such as pedestrians can change their shape and appearance (different when walking, running or being static) along different video frame sequences or in a real situation. On the other hand, the shape and appearance can also change due to a different point-of-view (objects close to the camera appear bigger than those farther from the camera).

3. *Illumination changes:* Handling with illumination changes is one of the hardest challenges for visual object tracking. The appearance of an object and in general the background of the scene can largely be affected by illumination changes, not to mention that an object may look different in outdoor environment (sun light) than indoor environment (artificial light). Even the weather conditions (cloudy, sunny, etc.) and specially time of day (morning, afternoon, evening) can be the causes of illumination changes. Even though current deep networks are both reliable and efficient (at least in standard conditions), there is a large accuracy downgrade when these methods are taken to adverse conditions such as night-time.

4. *Shadows and reflections:* Some of the features such as shape, motion and background are more sensitive for a shadow on the ground which appears and behaves like the object that casts it. In the same way, reflections of moving objects on smooth surfaces can cause problems when dealing with MOT.

5. *Occlusion:* This phenomenon occurs either due to one object is occluded by some component of the background or it is occluded by another object. A robust tracking system must be capable to check the individuality of the objects involved in the occlusion, before and after occlusion takes place.

It is important to mention that these previous challenges are significant to both multi-object tracking and single-object tracking. Nevertheless, multi-object tracking also requires solving the issue of modelling the multiple objects so the tracking method must be able to distinguish different objects so as to keep them consistently labelled.

2.3. Deep Learning in Multi-Object Tracking

Recently, more and more tracking algorithms (both SOT and MOT) have started exploiting the representational power of deep learning [3]. The strength of Deep Neural Networks (DNN), resides in their ability to learn rich representations and extract complex and abstract features from their input (generally an image). The reader is referred to the Appendix B in order to review Artificial Intelligence concepts behind this section. CNNs currently constitute the state-of-the-art in spatial pattern extraction, employed in tasks such as image classification, object detection or for object tracking in those cases in which the CNN is responsible of the first stage of the tracking algorithm (object detection). On the other hand, RNNs (like the LSTMs) are used to process sequential data, such as temporal series, text, audio signals. In this context, they would process a self-driving scene. Since DL methods have been able to reach top performance in many of those tasks, the community research is now progressively seeing them used in most of the top performing tracking algorithms (overall MOT). In that sense, there are different methodologies [71] when performing Deep Learning based MOT:

- *Classification-based trackers:* Trackers for generic object tracking often follows a tracking-by-classification approach. A tracker will sample *foreground* patches near the target object while *background* patches farther away from the target. These patches are used to train a *foreground-background* classifier, and this classifier is used to score potential patches in subsequent frames in order to estimate the new target location. This classifier is usually first trained off-line and fine-tuned during online tracking. Many neural-network trackers follow this approach (such as below studied MDNet [65]) have surpassed traditional trackers [72] and achieved state-of-the-art-performance [73]. However, these trackers are inefficient in terms of real time applications, such as self-driving, since neural networks are very slow to train in an online fashion. Another drawback of such a design is that it does not fully utilize all scene information, particularly explicit temporal correlation.

- *Regression based trackers*: Some recent works [61] [74] have attempted to perform object tracking as a regression instead of classification problem. [61] trains a CNN so as to regress directly from two images to the location in the second image of the object shown in the first image. Moreover, [74] proposes a fully-convolutional siamese network. These DL methods can run at frame-rates beyond real time (> 100 fps) while maintaining state-of-the-art performance. Unfortunately, they only extract features independently from each video frame and only perform comparison between two consecutive frames, not allowing the fully feature of their CNNs to use longer-term contextual and temporal information.

- *Recurrent-neural-network trackers*: [75] [76] propose the use of recurrent neural networks for the problem of visual tracking. [76] presents an RNN to predict the absolute position of the target frame and [75] similarly trains an RNN for tracking using the attention mechanism. Although these works brought good intuitions from RNN,

these methods have not yet demonstrated competitive results on modern benchmark. On the other hand, [64] proposes a spatially supervised recurrent convolutional neural network in which a YOLO [77] network is applied on each frame to produce object detections and an RNN is used to directly regress YOLO detections.

Hereafter, following points review some of the most interesting state-of-the-art approaches. Table 2.3-1 shows a simple qualitative comparison of the below mentioned approaches.

Table 2.3-1 Comparison of some interesting state-of-the-art Deep Learning based MOT approaches

	GOTURN	**MV-YOLO**	**MDNet**	**ROLO**	**Re3**
Language	C++	Python	M language	Python	Python
Framework	Caffe	Tensorflow	MATLAB	Tensorflow	Tensorflow
Typical FPS	100 (on GPU)	28 (using Yolo V3)	1	35	150
RNN	No	No	No	Yes	Yes
Capable of MOT	No	Yes	No	Yes	Yes
Motion information	No	Yes	Yes	Yes	Yes
Long-Term variations	No	Yes	No	Yes	Yes

2.3.1. GOTURN

GOTURN [61] is a Deep Learning based object tracking algorithm, originally implemented in Caffe [62]. GOTURN changed the way to apply DL to the tracking problem by learning the motion of an object in an *offline* way. While previously most tracking algorithms are trained in an *online* way (the tracking algorithm learns the appearance of the object in the runtime), GOTURN is trained on thousands of video sequences and does not need to perform any learning runtime.

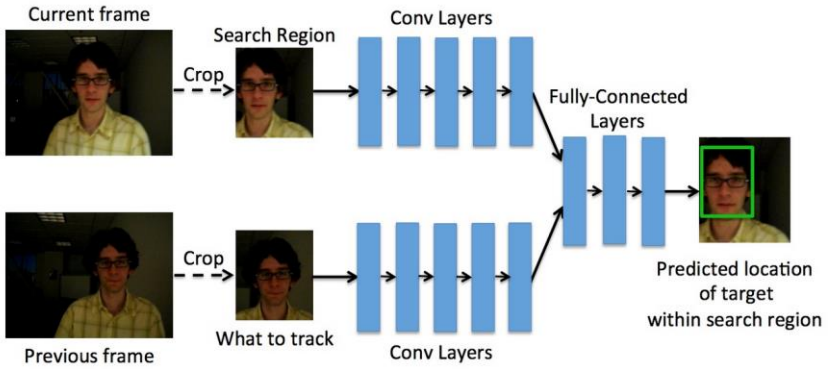


Figure 2.3-1 GOTURN architecture

As shown in Figure 2.3-1, it is trained using a pair of *cropped* frames from thousands of videos. In the first frame (the previous frame), the location of the object is known, and the frame is cropped to two times the size of the bounding box (BB) around the object. The object in the first frame is always centered.

On the other hand, the location of the object in the second frame (current frame) must be predicted. The BB used to crop the first frame is also used to crop the second frame. Since the object might have moved, the object is not centered in the second frame. To do that, a CNN is trained to predict the location of the BB in the second frame. The layers of this CNN are simply the first five conv-layers of the CaffeNet [62] architecture. The outputs of these conv-layers are concatenated into a single vector of length 4096. Then, this vector is the input to 3 FC-layers. Finally, the last FC layer is connected to the output layer containing four nodes, representing the top and bottom points of the BB.

2.3.2. MV-YOLO

MV-YOLO (Motion Vector-YOLO) [63] is a Deep Learning based tracking algorithm that combines decoded Motion Vectors (MVs) with semantic object detection (performed by YOLO [51]) operating on fully decoded frames. Basically, MVs (which already exist in the compressed video bitstream) are good enough to indicate the approximate location of the target object. Then, semantic object detection refines the target object location by providing pixel-precision BB on the decoded frame. The idea of two-stage tracking has been covered in other works, like ROLO [64]. ROLO approach is pixel-domain tracker, while MV-YOLO is the first hybrid one. Figure 2.3-2 shows the MV-YOLO architecture.

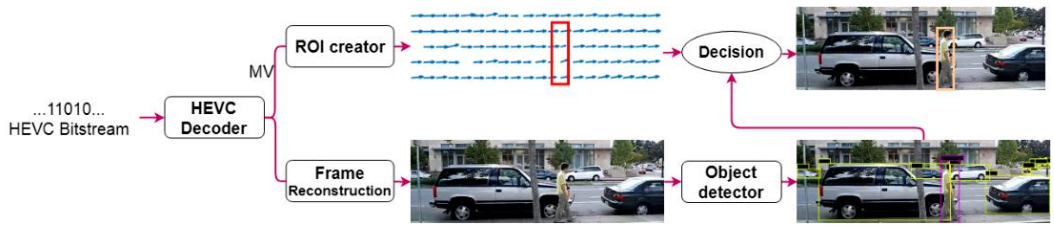


Figure 2.3-2 MV-YOLO architecture

This proposal incorporates data reuse from the compressed video bit stream and semantic object detection. Based on the MVs extracting during the video decoding process (note that this video may also be a real-time situation), a Region of Interest (ROI) for the target object is created in the current frame. Then, the output of the semantic object detector (YOLO) is used to more precisely localize the target object with the help of the ROI.

[63] shows that MV-YOLO is fast and robust, though depending on the particular object detector used (YOLOv2, v3, etc.). Even the slowest version is reasonably fast (28 fps), with an accuracy comparable to the recent trackers based on deep models.

2.3.3. MDNet

MDNet (Multi-Domain Network) [65] is a CNN architecture to learn the shared representation of targets from multiple annotated video sequences for tracking, where each video is regarded as a separate domain. Figure 2.3-3 shows the architecture of the network. It has separate branches of domain-specific layers for binary classification at the end of the network, sharing the common information captured from all sequences in previous layers for generic representation learning.

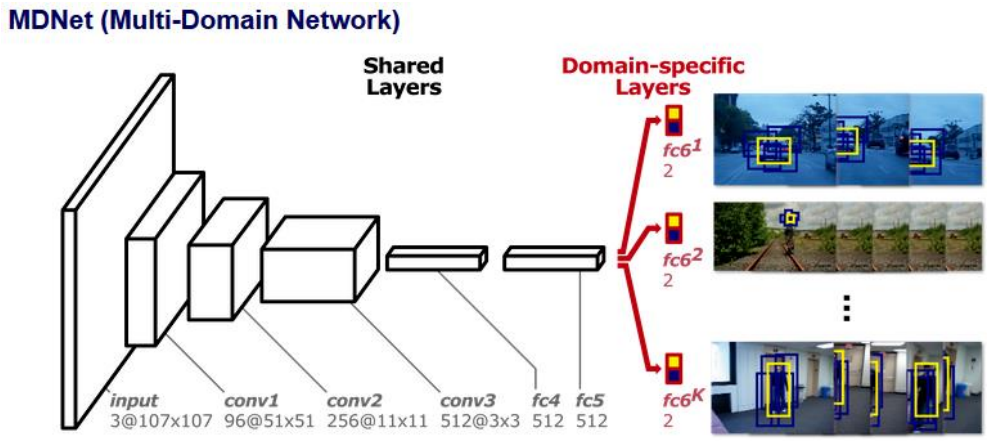


Figure 2.3-3 MDNet architecture

Each domain in MDNet is trained iteratively and separately while the shared layers are updated in every iteration. Using this strategy, domain-independent and domain-specific information is separated in order to learn generic feature representation. Besides this, [65] proposes an effective online tracking framework based on MDNet strategy. When a test sequence is given, all the existing branches of binary classification layers (used in the training phase) are removed and a new single branch is built to compute target scores in the test sequence. The new classification layer the FC layer within the shared layer are online fine-tuned during tracking so as to adapt to new domain. Finally, the online update is used to model both long-term and short-term appearance variations of a target for robustness and adaptiveness, respectively.

2.3.4. ROLO

ROLO (Recurrent YOLO) [64] is a visual tracking approach based on RNNs, which extends the neural network learning and analysis into the spatial and temporal domain. The key motivation of ROLO is that tracking failures can often be effectively recovered by learning from historical visual semantics and tracking proposals. Traditional Kalman or related temporal predictions methods usually only consider the location history. In contrast, ROLO

examines both the location history as well as the robust visual features of past frames for each target object. Figure 2.3-5 shows the ROLO architecture. It uses the YOLO object detector to collect robust and rich visual features. Then, LSTM are used in the next stage as it is spatially deep and appropriate for sequence processing. Then, as ROLO proposal, this architecture uses the regression capability of LSTM and proposes to concatenate high-level visual features produced by conv-nets with region information.

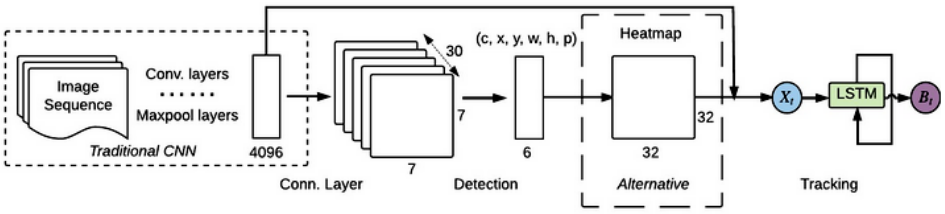


Figure 2.3-4 ROLO architecture

On the other hand, Figure 2.3-5 shows a simplified but very illustrative overview of the tracking process using ROLO. The YOLO input is raw input frames, and its output is a feature vector of input frames and bounding box coordinates. Then, the LSTM input is a concatenation of image features and bounding box coordinates whilst the LSTM output is the bounding box coordinates of the object to be tracked.

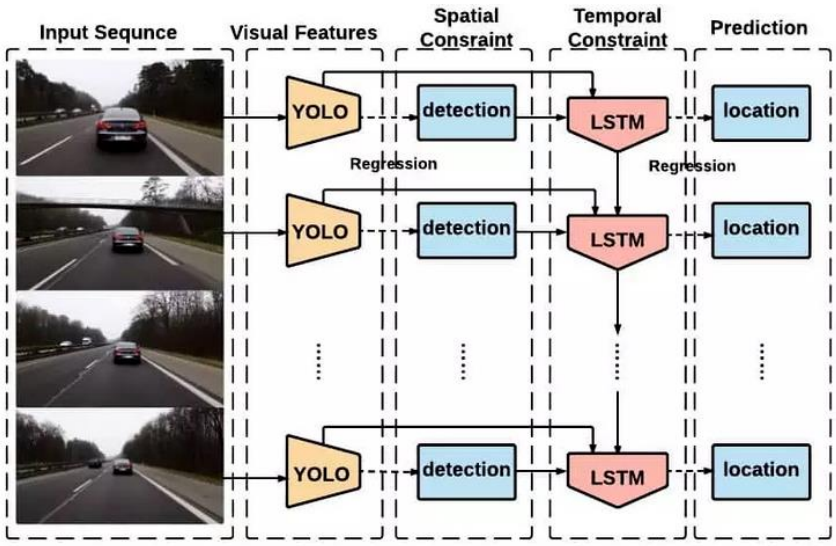


Figure 2.3-5 (Top) Simplified ROLO overview and tracking procedure (Bottom) ROLO architecture

2.3.5. Re3

Re3 (Real-time, Recurrent, Regression) [66] is fast but accurate RNN for generic object tracking. Re3 learns to store and modify relevant object information in the recurrent information.

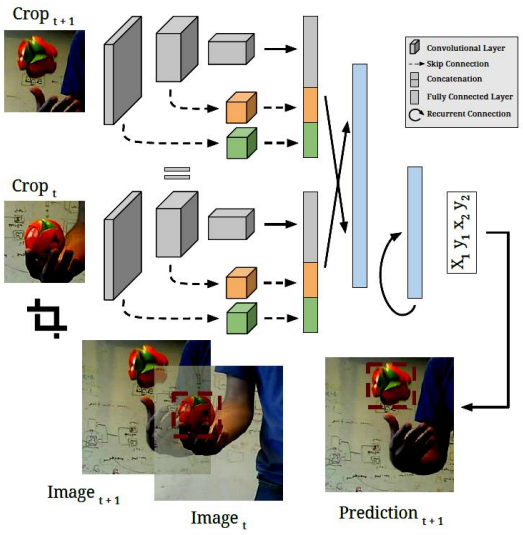


Figure 2.3-6 Re3 architecture

By incorporating information from large collections of videos and images, this RNN learns to produce representations that capture the main features of the tracked object. The goal of this process is to teach the network how any given object is likely to change over time. This strategy makes Re3 computationally cheap and extremely fast during inference, an important quality for algorithms operating on mobile robots, such as autonomous vehicles. Figure 2.3-6 shows the Re3 architecture. It consists of conv-layers to extract the object appearance, recurrent layers to remember this appearance and motion information of previous frames and a regression layer to output the predicted location of the object.

Chapter 3. Software technologies used in the thesis

3.1. Introduction

This chapter aims to explain the software tools used in this thesis in terms of perception systems. Hardware technologies, sensors and the autonomous car involved in the development of this work are detailed in Chapter 4. As mentioned in previous chapters, self-driving applications require robust tracking system in order to perform a safe navigation. Then, there are four main requirements in order to meet this robust tracking system:

- *Sensor redundancy* (if a sensor fails, the vehicle should be able to continue the trajectory)
- *Perception* of the environment as accurate as possible
- *Great accuracy* in the on-road obstacle position and velocity estimation
- *Real-time operation* in data communication among sensors and decision-making process

To meet these conditions, the software tools used in this master thesis have been: ROS (for multi-sensor communication), PCL (for 3D reconstruction of the environment), Docker (for the automation and portability of the proposed architecture and code between among different computers or microcontrollers) and CARLA (for self-driving simulation, as a preliminary stage to implement the architecture in the real-world). The following sections explain the main features of these tools.

3.2. ROS

The *Robot Operating System (ROS)* [32] is a flexible framework (meta-operating system) for writing robot software. It is considered an open-source collection of libraries, tools and conventions whose aim is to simplify the task of obtaining, building, writing and running code across multiple computer to perform robust and complex behaviours over a wide variety of robotic platforms.

Figure 3.2-1 ROS logotype

It provides the expected services from an operating system, including low-level device control, implementation of commonly-used functionality, hardware abstraction, message-passing between processes and package management. The ROS runtime "graph" is a P2P (Peer-to-Peer) network of processes (potentially distributed across machines) which are loosely coupled using the ROS communication infrastructure. ROS presents several different styles of communication, including asynchronous streaming of data over topics, synchronous RPS-style communication over services and storage of data on a parameter server. Moreover, in spite of the fact that ROS is not a real-time framework, it is possible to integrate ROS with real-time code.

ROS currently only runs on Unix-based platforms and in particular it has primarily tested on Ubuntu and Mac OS X systems. The core ROS system, along with useful tools and libraries are regularly released as a ROS distribution. A ROS distribution is a versioned set of ROS packages. This master thesis is based on two LTS (Long Term Support) distributions: ROS Melodic Morenia, for the docker image related with Deep-Learning based Multi-Object Tracking, and ROS Indigo Igloo in which the docker image of the SmartElderlyCar project is based on. Chapter 5 offers a deeper explanation of the architecture proposal and how ROS has been employed in the project.

3.2.1. Goals

The primary goal of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes (also known as nodes) that enables executables to be individually designed and loosely coupled at runtime. These nodes can be grouped into Stacks and Packages, which can be easily shared and distributed. It also supports a federated system of code Repositories that enable collaboration to be distributed as well. This design, from the filesystem level to the community level, gives rise to independent decisions about development and implementation. In order to meet this primary goal of collaboration and sharing, there are other goals:

1. *Thin:* ROS developers intend for drivers and other algorithms to be contained in standalone executables in order to allow that code written for ROS can be used with other robot software frameworks. This ensures maximum reusability, making ROS easy to use (being the complexity in the libraries). This arrangement also facilitates unit testing and the developed systems can be completely independent of another system.

2. *Peer-to-Peer communication:* Complex robotic systems with multiple links usually have multiple on-board computers in order to perform parallel tasks connected via a network. Peer-to-Peer (P2P) communication avoids the problem of running a central master which would result in severe congestion in one particular link. Figure 3.2-2 shows the difference between centralized and P2P network architecture. While P2P does not require a central server but node to node connections, is resilient and support large messages (essential for sensor communication), centralized architecture is feature by clients that send requests to/via a central server, supports only small messages and the main failure point is the master.

Figure 3.2-2 Peer-to-Peer vs Client/Server architecture

In ROS, a P2P architecture coupled to a buffering system and a lookup system (a name service called master) enables each component to communicate directly with any other, asynchronously or synchronously as required.

3. *Multi-Language:* The ROS framework is featured by a language independence what makes easier to implement in any modern programming language. The ROS specification works at the messaging layer. P2P connections are negotiated in XML-RPS, which exists in a great number of languages. To support a new language, either C++ classes are re-wrapped (as happened for the Octave client) or classes are written enabling messages to be generated. These messages are described in IDL (Interface Definition Language). Currently ROS can be implemented is Python (whose main ROS library is *rospy*), C++ (*roscpp*) and Lisp (*roslisp*).

4. *Easy testing:* ROS has a builtin unit-integration test framework (*rostest*) that makes it easy to bring up and tear down test fixtures.

5. *Scaling:* ROS framework is suitable for large runtime systems and large development processes thanks to the help of the P2P architecture and buffering system.

3.2.2. Main concepts

ROS has three levels of concepts: The *Filesystem level*, the *Computation Graph level* and the *Community level*, that comprise all the relevant information in order to carry out the communication between the sensors, actuators and other mechanisms of the robot.

3.2.2.1. ROS Fylesystem level

The filesystem level concepts cover ROS resource found on disk, such as:

1. *Packages:* Main unit for organizing software in ROS. They are considered the most atomic build item and release item in ROS, so the most granular thing it can be built is a package. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, configuration files, datasets, or anything else which is usefully organized together.

2. *Metapackages:* Specialized packages which only serve to represent a group of related packages.

3. *Package manifests:* Manifests (*package.xml*) provide metadata about a package (such as name, version or description and other metadata information).

4. *Repositories:* Packages which share a VCS (Version Control System) share the same version and can be released together using the catkin release automation tool *bloom*.

5. *Message (msg) types:* Message descriptions define the data structures for messages sent in ROS.

6. *Service (srv) types:* Service descriptions define the request and response data structures for services in ROS.

3.2.2.2. ROS Computation Graph level

The *Computation Graph* is the P2P network is ROS processes that are processing data together. The basic *Computation Graph* concepts are:

1. *Nodes:* Processes that perform computation. ROS is designed to be modular at a fine-grained scale, so a robot control system usually comprises many nodes (one node controls the wheel motors, another performs localization, etc.) A ROS node is written with the use of a ROS client library, such as *roscpp* (C++) or *rospy* (Python).

2. *Master:* The ROS Master provides name registration and lookup to the rest of the *Computation Graph*. Nodes would not be able to find each other, exchange messages or invoke services without the presence of the Master.

3. *Parameter Server:* Server that allows data to be stored by key in a central location (currently part of the Master).

4. *Messages:* Data structure comprising typed fields. Standard primitive types (like integer or Boolean) are supported, as are arrays of primitive types. Nodes communicate with each other by passing messages.

5. *Topics:* Messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given topic, that is, bus over which nodes exchange messages. A node that is interested in a certain message will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may subscribe and/or subscribe to multiple topics. Each bus (topic) has a name, and anyone can connect to the bus to send or receive messages as long as they are right type.

6. *Services:* The publish/subscribe ROS model y a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC (Remote Procedure

Call) request, which are often required in a distributed system like a multi-sensor robot. Then, Request/Reply is performed via a service (srv file) under a string name, and a client calls the service by sending the request message and awaiting the reply.

7. Bags: ROS format for saving and playing back ROS message data. They are an important mechanism for storing sensor data, essential to be collected for later development and testing algorithms.

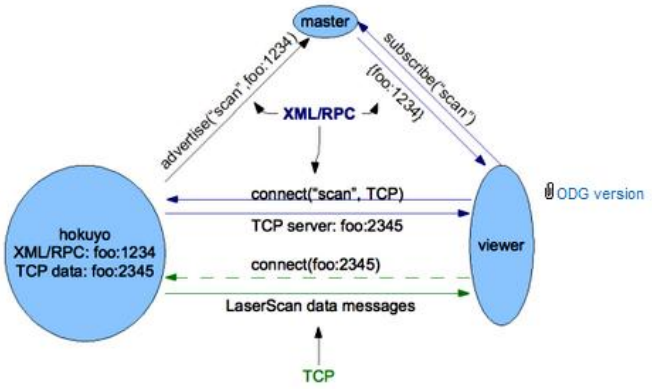


Figure 3.2-3 Example of publisher/subscriber and its relationship with the Master

The ROS master acts as a nameservice in the ROS Computation Graph, storing topics and services registration information for ROS nodes. Nodes connect to other nodes directly, and the Master only provides lookup information (like a DNS (Domain Name Server)). It is important to note that names have a very important role in ROS. Nodes, topics, services and parameters all have names.

3.2.2.3. ROS Community Level

The ROS Community Level concepts are ROS resources that enable separate communities to exchange knowledge and software. These resources include:

1. *Repositories:* ROS relies on a federated network of code repositories, where different institutions can develop and release their own SW components.

2. *Distributions:* Like Linux distributions, ROS Distributions are collections of packages and code that make easier to install this collect of SW in the computer.

3. *ROS Wiki:* Main forum for documenting information about ROS.

4. *Mailing lists:* Primary communication channel about new updates to ROS, as well as a forum to ask questions about ROS software.

5. *ROS anwwers:* A Q&A (Questions and Answers) site for dealing with ROS-related questions.

3.2.3. Main ROS tools used in this thesis

All the code (both in C++ and Python) developed for this master thesis is directly or indirectly related with ROS. Apart from the required ROS packages to develop this code, there have been used four main tools:

1. *roslaunch:* Tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML files (configuration files with the .launch extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

2. *rqt:* Software framework of ROS that implements the various GUI tools in the form of plugins (such us the TF tree). The tools can still run in a traditional standalone method, but rqt makes it easier to manage all the various windows on the screen at one moment.

3. *Rviz (ROS visualization):* 3D visualizer for displaying sensor for displaying sensor data and state information from ROS, such as camera data, infrared or LiDAR measurements, sonar data and more.

4. *RoboGraph:* ROS tool that allows to define algorithms as computational graphs. Once the graph is defined, it can be executed as if it was a software program and get the expected outputs. Each graph, made up by nodes and transitions, can accept any number of inputs and give at maximum one output. Hierarchical Petri Nets [41], in which the SmartElderlyCar project (Chapter 4) is currently based, have been built on RoboGraph.

3.3. Point Cloud Library (PCL)

The *Point Cloud Library (PCL)* (Figure 3.3-1) [47] is a standalone, large-scale, open project for 2D/3D image and point cloud processing. A point cloud is a data structure that represents a set of points in several dimensions (XYZ geometric coordinates) of a sampled surface. In addition, it can add a fourth dimension if colour is available. Point clouds can be acquired from HW sensors such as 3D scanners, stereo cameras or flight-time cameras, or even they can be generated from a computer program synthetically.

Figure 3.3-1 Point Cloud Library logotype

PCL is a modern C++ library modelled for 3D point cloud processing. It is based on the Eigen library for mathematical operations and on FLANN (Fast Library for Nearby Neighbours) for the search of neighbouring points. PCL also uses shared Boost pointers. It incorporates several 3D processing algorithms, such as filtering, features extraction, surface reconstruction or segmentation) which integrate all its functionalities in a compact way. Like ROS, it is an open-source software, being free for commercial and research use. In addition, it is a multiplatform SW compatible with Windows, MacPS, Android/iOS and Linux.

To simplify the point cloud development, PCL is divided into a series of smaller libraries, which can be compiled separately, allowing distribution on platforms with computational limitations. The main modules to be used in this work are as following:

1. *Filters (pcl_filters):* Module that contains noise elimination mechanisms and outliers for 3D point cloud data filtering applications.

2. *Kd-tree (pcl_kdtree):* Module that provides the kd-tree data structure, which allows quick searches of neighbouring points closest to the analysis point. Kd-tree (K-dimensional tree) is a data structure that stores a set of k-dimensional points in a tree structure to perform range and nearest-neighbour searches.

3. *Segmentation (pcl_segmentation):* Module that contains algorithms to segment a point cloud into different clusters (subsets of relevant spatially isolated points within the point cloud). It is used to process a point cloud made up by spatially isolated regions, which is divided to be processed independently.

4. *Input/output (pcl_io):* Module that contains classes and functions for reading and writing point cloud data files (PCD).

5. *Visualization (pcl_visualization):* Module that allows prototyping and visualizing the results of the algorithms that operate in 3D point cloud data. It allows to represent and set visual properties, draw 3D shapes and their visualization.

3.3.1. PCL-ROS

The PCL design philosophy relies on the fact that most of the applications that deal with point cloud processing are generated as a set of blocks that are parameterized to achieve results.

Based on the design of other 3D processing libraries and ROS, each *PCL-ROS* (Figure 3.3-2) algorithm is available as an independent block that can be easily connected to other blocks in the same way that nodes connect to each other in ROS. In addition, since point clouds are large data structures, in order to ensure that point clouds are not copied in critical ROS applications *nodelets* are created, which are dynamically loadable add-ons that look and work as ROS nodes but in a single process (such as simple/multiple threads).

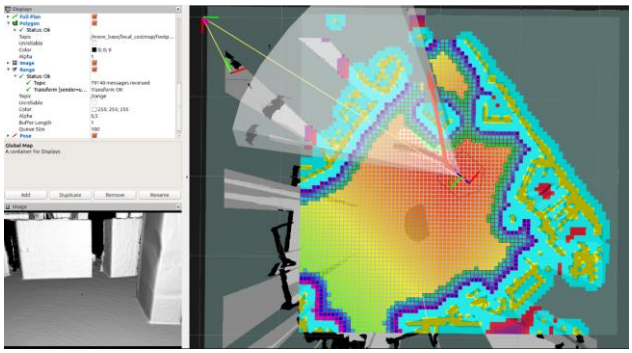


Figure 3.3-2 Bird Eye View of PCL-ROS

3.4. Docker

Docker [60] (Figure 3.4-1) is an open source project that offers a software development solution known as containers. It is a tool designed to benefit both developers and system administrators, making it a part of many DevOps (Develops + Operations) toolchains. For developers (for example, when developing the code of this master thesis), it means that they can focus on writing code without worrying about the system that it will ultimately be running on. Since containers are platform- independent, Docker can run across both Windows and Linux-based platforms. In that sense, the main purpose of Docker is that it lets a developer (as the development of this work) run microservice applications in a distributed (like the SmartElderlyCar architecture). Currently Docker can be run in desktop (Mac OS, Windows 10), Server (various Linux distributions and Windows Server 2016) and Cloud (Amazon Web Services, Microsoft Azure or Google Compute platform).

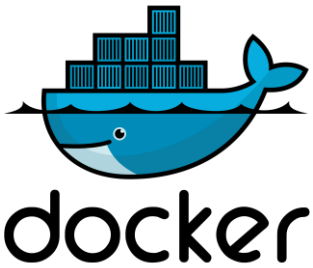


Figure 3.4-1 Docker logotype

3.4.1. Docker engine

Docker engine (Figure 3.4-2) is the base of Docker. It allows the user to develop, assemble, ship and run applications using the following components:

1. *Docker Daemon:* Persistent background process that manages Docker containers, networks, storage volumes and images. It constantly listens for Docker API requests and processes them.

2. *Docker Engine REST API:* API used by applications to interact with the Docker daemon. It can be accessed by an HTTP client.

3. *Docker CLI:* Command line interface client for interacting with the Docker daemon. It greatly simplifies the way to manage container instances.

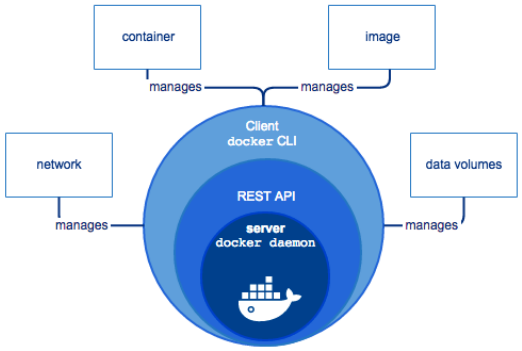


Figure 3.4-2 Docker engine

3.4.2. Docker architecture

The *Docker architecture* uses a client-server model that comprises of the Docker Client, Docker Host, Network and Storage Components. As shown in Figure 3.4-3, there are three types of Docker communication, that is, Build, Pull and Run. For example, pull must be done when downloading an image from a determined registry (e.g., DockerHub), build when creating an image from a DockerFile and run in order to create a container from a given image.

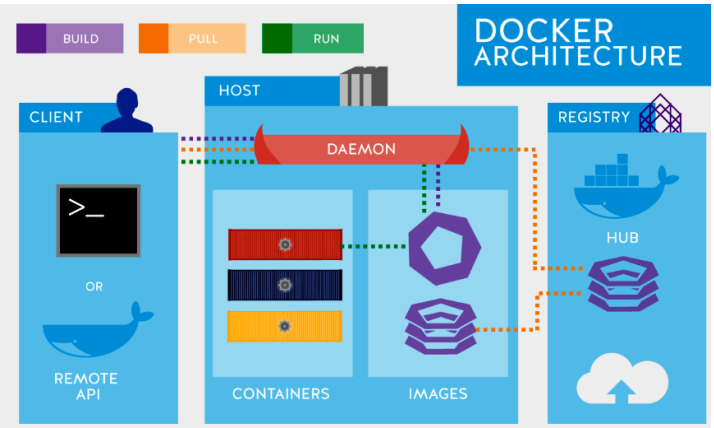


Figure 3.4-3 Docker architecture

3.4.2.1. Docker Client

The *Docker client* enables users to interact with Docker. It can reside on the same host as the daemon or connect to a daemon on a remote host. In addition, a docker client can communicate with more than one daemon. It provides a CLI (*Command Line Interface*) in order to build, run and stop application commands to a Docker daemon. Its main purpose is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Common commands are *docker build, docker pull* or *docker run* (build an image from a Dockerfile, pull an image from a registry or run a container, respectively).

3.4.2.2. Docker Host

The *Docker Host* provides a complete environment to run and execute applications. It comprises of Docker daemon, Containers, Images, Networks and Storage. As commented, the Docker Daemon is s responsible for all container-related actions, receiving commands from the REST API and CLI. The Daemon pulls and builds container images as request by the client. Once the image is pulled, the daemon builds a working model for the container using a build file (set of instructions).

3.4.2.3. Docker Objects

There are some objects used in the assembling of an applications. The main objects related with the development of this master thesis have been:

1. *Images:* Read-only binary templates used to build containers. They contain metadata that describe the container's capabilities and needs. Images are used to store and ship applications. Container images can be shared across teams using a private container registry or shared with the world using a public registry (Docker Hub). In the case of this master thesis, Docker Images are stored in Solid-State Disk (SSD), so the image is portable in an USB or via-Ethernet if it is required to be pulled in another computer.

2. *Container:* Encapsulated environments in which the user runs applications. The container is defined by the image and any additional configuration options provided on starting the container (such as storage options and network connections). A container only has access to resources that are defined in the image (unless additional access is defined when building the image into a container). When committing the current state of the container, the original image is updated or created a new one if it was committed with a different tag.

3. *Storage:* It can be stored data within the writable layer of a container of a container. In terms of persistent storage, Docker offers several options. The option used in this project has been data volumes. Data volumes provides the ability of creating persistent storage, with the ability to rename volumes, list volumes and also list the container that is associated with the volume. They sit of the host file system, outside the container copy on write mechanism. In other words, if a directory of 59 GB is shared between a docker

container and the host machine, when committing that container its size would not increase in 59 GB, since these data sit on the host file system.

3.4.2.4. Docker Registries

Docker registries are services that provide locations from where it can be stored and downloaded images. That is, a Docker registry contains Docker repositories that contain one or more Docker images. Public registries include a Docker Cloud and Docker Hub, in addition to private registries. Common commands when working with registries are *docker push*, *docker pull* and *docker run* (push an image to a registry, pull an image from a registry and run a service).

3.4.3. Docker advantages

Before commented the docker advantages, it must be compared with a Virtual Machine and why this master thesis has not been developed using a Virtual Machine.

A *Virtual Machine (VM)* is a virtual server that emulates a hardware server. It relies on the system physical hardware to emulate the exact same environment on which the user installs applications. Depending on the case, it can be used a system virtual machine (runs an entire OS as a process, so the real machine can be substituted for a virtual machine) or process virtual machines that allow to execute computer applications alone in the virtual environment. An example of a real-world use case for VMs is the Starling Bank (digital-only bank built in 2018 on VMs provided by Amazon Web Service. Then, since the VMs efficiency deliver over traditional HW servers, as this bank bought thousands of traditional servers, the use of VMs is possible).

However, for this master thesis, the use of a VM is not suitable, due to the needs of continuous Application Development and Running Microservices Applications in standard computers, as used in research. While a container runs natively (for example, on Linux) and shares the kernel of the host machine with other containers (running discrete processes, taking no more memory that other executable, as a system thread), a VM runs a full-operating system with virtual access to host resources through a hypervisor (in general, VMs provide an environment with much more resources than most applications required). Figure 3.4-4 shows a comparison between the architecture of Docker containers and Virtual Machines.

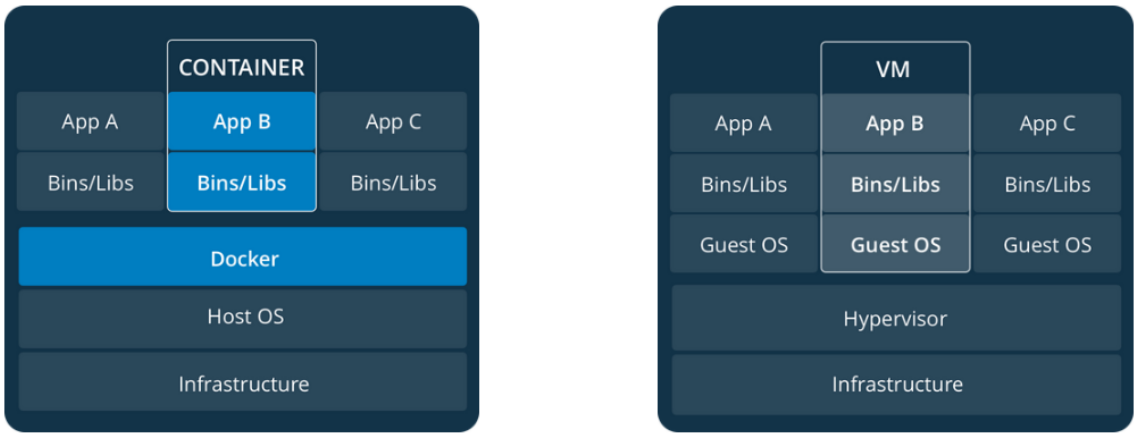


Figure 3.4-4 Docker containers vs Virtual Machines

Finally, the main advantages of using Docker and why it has been chosen to develop this master thesis have been:

- *Isolation and lightweight:* Docker containers are process-isolated and do not require a HW hypervisor, what means that containers are much smaller and require far fewer resources that a VM.

- *Fast:* While a VM can take at least a few minutes to boot and be (development-ready), container starts from a few milliseconds to (as most) a few seconds to start a container from an image.

- *Portability and interchangeable:* Containers can be shared across multiple team members, bringing much-needed portability across the development pipeline. This is essential to reduce the errors when transferring the code from a machine to another one, and in this case, to install the developed code and dependencies in future microcontrollers, as the Jetson Xavier.

- Flexible: Even the most complex applications can be containerized.

3.5. Carla simulator

CARLA (Car Learning to Act) [31] (Figure 3.5-1) is an open source simulator for urban driving. It has been developed from scratch to support training, prototyping and validation of autonomous driving models (including both control and perception). Like CARLA, the content of provided urban environment is also free, which includes a multitude of vehicle models, pedestrians, street signs, buildings, etc. CARLA simulator supports flexible setup of sensor requirements and signals used to train driving strategies, such as speed, acceleration or GPS coordinates. A wide range of environmental conditions can also be specified, such as rainy, cloudy or sunset.

Figure 3.5-1 CARLA logotype

3.5.1. Simulation Engine

CARLA is implemented as an open-source layer over the *Unreal Engine 4* (UE4) [67]. This simulation engine provides CARLA flexibility and realism in the rendering and physics simulation. It provides state-of-the-art rendering quality, realistic physics and an ecosystem of interoperable plugins. In that sense, CARLA simulates a dynamic world (Figure 3.5-2) and provides a simple interface between an agent that interacts with the world and the world. In order to support this functionality, CARLA is designed as a server-client system, where the server runs the simulation and renders the scene. The client API is implemented in Python, responsible for the interaction between the autonomous agent and the server. The client sends meta-commands and commands to the server and receives as response sensor readings. Meta-commands control the behaviour of the server, used for changing the properties of the environment (such as the weather conditions or illumination. On the other hand, commands control the vehicle, including steering, braking and accelerating.

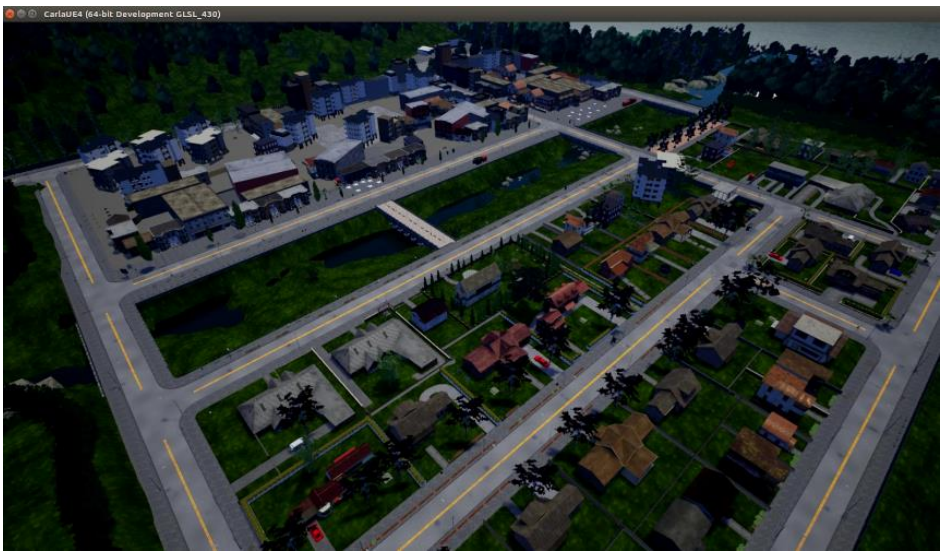


Figure 3.5-2 CARLA world

3.5.2. Environment and sensors

CARLA environment is composed of 3D models of static objects like vegetation, infrastructure, buildings or traffic signs, as well as dynamic objects such as pedestrians or vehicles. They are designed by using low-weight geometric models and textures but maintaining visual realism by carefully crafting the materials and making use of variable level of detail.

To build urban environments, CARLA follows the following steps:

1. Laying out roads and sidewalks.
2. Manually placing vegetation, terrain and traffic infrastructure.
3. Specifying locations where dynamic objects can appear (spawn).

Additionally, CARLA implements a variety of atmospheric and illumination conditions, differing in the position and colour of the sun, intensity and colour of the diffuse sky radiation as well as atmospheric fog, ambient occlusion and even precipitation. This gives rise to a total of 18 illumination-weather combinations.

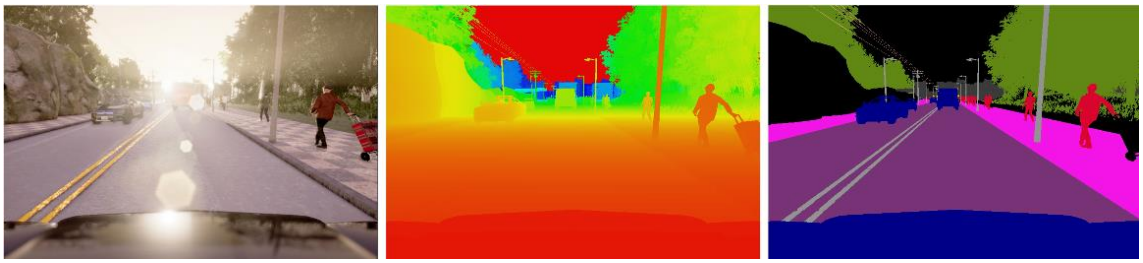


Figure 3.5-3 Different sensing modalities provided by CARLA: Normal vision, ground-truth depth and ground-truth semantic segmentation

From the sensors perspective, CARLA allows for flexible configuration of the agent sensor suite. Most common sensors in CARLA are RGB cameras (and pseudo-sensors that provide semantic segmentation and groundtruth depth, Figure 3.5-3), LiDAR and GPS (main sensors required for self-driving applications). Moreover, camera parameters include 3D orientation and position with respect to the car coordinate system, field-of-view and depth of field.

In addition to pseudo-sensor and sensor readings, CARLA provides a range of measurements associated with the state of the agent and compliance with traffic rules, including vehicle location and orientation with respect to the world coordinate system, acceleration vector, speed and accumulated impact from collusions. A very important feature of CARLA used in this master thesis to validate the proposal in Chapter 6 is that CARLA provides access to exact locations and bounding boxes of all dynamic objects in the environment, what means a groundtruth to validate a real-time 3D prediction in terms of Multi-Object Tracking, like (and even better) that using a static benchmark.

3.5.3. Autonomous Driving

CARLA is used to study the performance of three approaches to autonomous-driving:

1. *Classic modular pipeline* that comprises a vision-based perception module, a maneuver controller and a rule-based planner.
2. *Deep network* that maps sensory input to driving commands, trained end-to-end using *imitation learning*.
3. *Deep network* maps sensory input to driving commands, trained end-to-end using *reinforcement learning*.

All these approaches make use of a common path planning strategy provided by a high-level topological planner. This planner takes the current position of the agent and the location of the goals as input, then uses the A* [45] algorithm to provide a high-level plan required by the agent to follow in order to reach the goal.

In this master thesis, none of these approaches are used since it is used the SmartElderlyCar autonomous architecture proposal (including the Deep Learning based MOT presented in this work) in order to navigate with the agent, using CARLA as an excellent simulator to bridge in an efficient and right way the real-world and simulation to validate the proposal and then implement it in real situations.

Moreover, this master thesis has used the CARLA 0.9.5 release (4th April 2019). It was not update to the last release (0.9.6, 12th July 2019) since the ROSbridge, required to communicate with the SmartElderlyCar Docker container, was deprecated in order to work with 0.9.6. The *ROSbridge* is a ROS package that aims at proving a ROS bridge for CARLA simulator, that is, it allows message passing between simulator and ROS. For example, vehicles may publish transform information, sensors of different agents publish data stream or it can be published control messages from ROS. It is important to consider that the Ego vehicle (main vehicle in which control algorithms and testing are implemented) is separated from other vehicles.

Chapter 4. SmartElderlyCar project

4.1. Motivation and scope of the project

This chapter focuses on the project in which this master thesis has been applied on. The project, named *SmartElderlyCar* and funded by Ministerio de Economía y Competitividad (Spain), aims to implement an autonomous electric vehicle able to drive in the campus of the University of Alcalá (Spain). Figure 4.1-1 shows the most recent version of the real prototype, still in development, whose software, hardware and structure have been developed by a joint work of the University of Vigo (GROBIS research group) and University of Alcalá (RobeSafe research group). MOT techniques developed in this project are traditional and based on Precision-Tracking [46]. This approach will be considered as baseline for the Deep Learning based Multi-Object Tracking architecture proposed in this work.

To put in context the final application of the proposed tracking, hereafter it is shown the main characteristics of our autonomous navigation architecture (our vehicle, the sensor used, the simulator and the different use cases validated in simulation).

Figure 4.1-1 Real autonomous electric car of Robesafe Research Group (UAH)

4.2. Autonomous Navigation Architecture

Figure 4.2-1 represents the *car software framework* as a modular architecture where individual modules asynchronously process information. These modules are independent

processes that communicate with each other using the ROS inter-process communication system (PCS). In particular, the publish/subscribe paradigm is used in order to provide non-blocking communications.

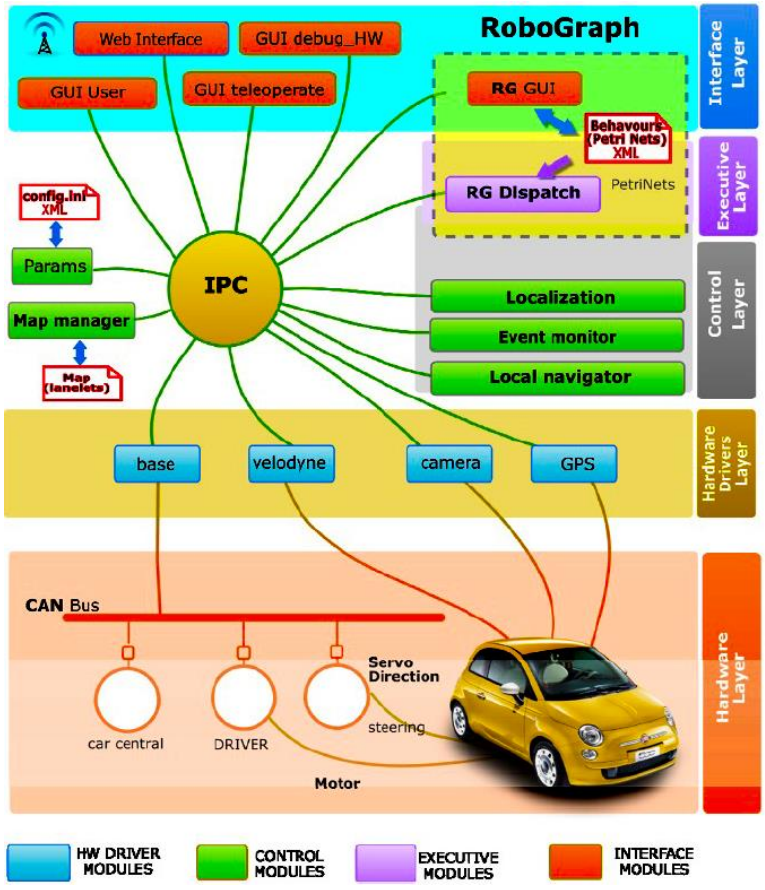


Figure 4.2-1 Proposed autonomous navigation architecture

Each module corresponds to an independent Linux process running on different ECUs (Electronic Control Units). Software modules are organized in four sets:

1. *The hardware driver layer:* Set of programs that control different hardware devices that comprise sensors and actuators.

2. *The control layer:* Set of programs that implements the basic control and navigation functionality. It also contains the reactive control (*local navigator*), localization (*localization*), path planning (*map manager*) and a program that processes most of the exteroceptive sensors to detect relevant events (event monitor).

3. *The executive layer:* Set of programs that coordinates the sequence of actions that need to be executed by other modules to carry out the current behaviour.

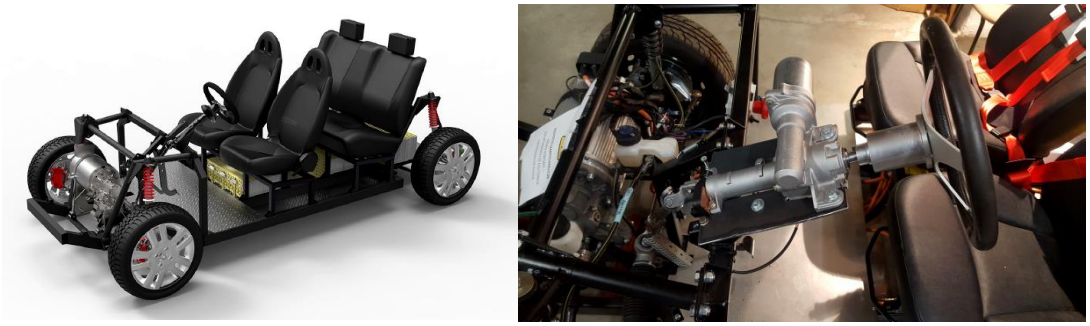
4. *The interface layer:* Set of processes to interact with the users and to connect to other processes for multi-robot applications.

The environment perception is based on the sensor fusion of camera and LiDAR information. The motion control is divided into lower-level reactive control and high-level planning. First, the high-level planning calculates a path consisting of a sequence of lanelets [37] (which can be modified depending on the performed behaviours by the executive layer). The goal of the local navigation system is to safely follow the path, keeping the car within the driving lane, and following the behaviours constraints established by the high-level planning. In order to do that, the car obtains the curvature to guide the car from the current position to a look-at-head position placed in the center of the lane by using the Pure Pursuit approach [39]. Then, this curvature is used as the reference for an obstacle-avoidance method based on the Beam Curvature Method (BCM) [40]. This BCM approach allows to keep the vehicle centered in the lane while is able to avoid unknown obstacles that can partially block the lane.

Finally, the decision making is implemented through Petri nets that take as inputs the map manager information, the local perception (provided through the event monitor module) and the vehicle localization in the map. In concrete, hierarchical interpreted binary Petri nets [41], where a net can stop or start another net of the already started ones. To implement them, the RoboGraph tool is employed.

4.3. Real prototype

The SmartElderlyCar is based on the chassis of an open source EV (Electric Vehicle), the TABBY EVO of the company Open Motors [33] (Figure 4.3-1 (a)). Thanks to the huge effort of the RobeSafe research group, this chassis has been integrated with a tubular car body, a pack of batteries, some sensors (mainly LiDAR, GPS and camera) and a drive-by-wire system enable to prepare it for autonomous navigation. The manual steering wheel has been removed so as to install a commercial electrical power steering (Opel Corsa model) with an encoder to control de vehicle direction electronically, as shown in Figure 4.3-1 (b). In order to do that, an ECU (Electronic Circuit Unit) has been designed based on an olimexino STM32 open-source development board. It receives the angle commands and generates a PWM signal to a full bridge by using a PID closed loop control.



(a) (b)

Figure 4.3-1 (a) Open-source chassis, (b) Electrical power steering wheel

In a setup process the relationship between the desired angle of the steering wheels and the steering wheels angle is calculated. A sensor is included in the power steering to read the angle as well as switch from autonomous to manual control if the driver puts his hand in the wheel. Moreover, the signal generated by the throttle paddle is switched by a signal generated by the ECU to obtain the desired acceleration. If the driver puts his foot in the paddle the system automatically switches to the manual mode. The inputs of the ECU are the angle of the steering wheels and the desired velocity, and these commands are sent through the CAN bus from the high-level control.

4.4. Sensors

For environment perception, the vehicle is mainly equipped with a Velodyne LiDAR (VLP-16) placed on the top of the vehicle and looking at the front of the vehicle, a stereo vision 2-sensor colour camera (ZED), which is mounted on the front windshield of the vehicle at 165 cm height above the ground and oriented to the road and a DGPS-RTK GPS TopCon HiperPro installed on the top in the central rear part of the vehicle. These devices are connected to an on-board embedded computer, working under Ubuntu OS, where the SW architecture is run.

A self-driving car requires more sensors to navigate safely than mentioned above. However, since this thesis is specifically based on the above-mentioned sensors (LiDAR, camera and GPS) it is required to check the key concepts of each technology in addition to the specific features of that models.

4.4.1. Distance sensor: LiDAR

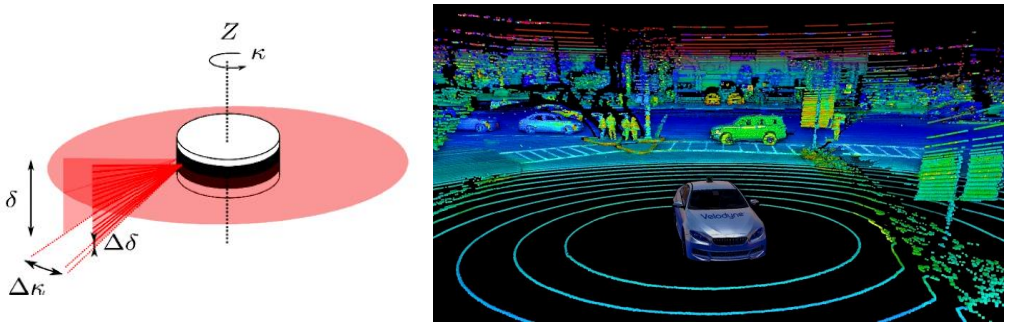
Actives distance sensors are a key concept in mobile robots in terms of localization and environment modelling. They are based on obtaining the distance through the propagation speed of an emitted wave, and the time it takes from the emission until it is received (flight time):

$$d = \frac{c_0 \cdot t}{2} \tag{4.1}$$

Where c_0 is the light speed (when speaking about laser), t is the flight time and d the distance to the object.

The main advantage of laser sensors is that they can obtain more accurate and reliable distance measurements than other types of sensor distance, such as ultrasonic.
LiDAR (Light Detection and Ranging) technology uses a type of sensor that measures distance to a target by illuminating the target with laser light (pulsed laser beam) and measuring the reflected light with a sensor. Its working principle is relatively simple: A diode inside emits the laser beam that is directed through a transmitter lens, hits the target

and part of the light is reflected in a photodiode after passing through a receiver lens. Then, differences in laser return times and wavelengths can be used to perform 3D representations of the target and in general to carry out a 3D reconstruction by using a point cloud of the environment. Figure 4.4-1 (a) shows an overview of the LiDAR system (in horizontal) and Figure 4.4-1 (b) a 3D reconstruction of the environment with a colour scale in such a way that closer objects are painted in green and further objects are painted in dark blue.



(a) (b)

Figure 4.4-1 (a) LiDAR system overview in horizontal (b) 3D reconstruction of the environment

The 3D LiDAR consists of stacked rotary lasers that obtain information from the environment from different angles, which allows to obtain a point cloud. Each layer of lasers is a channel that emits a signal that creates a contour line, and that, together with the contour lines of the other channels, generates a 3D reconstruction of the environment. Therefore, the higher the number of channels, the higher resolution. The SmartElderlyCar uses a Velodyne LiDAR Puck (VLP-16) [34], which provides 16 channels of 360 º horizontal FoV (Field of View) and +/- 15 º vertical FoV. VLP-16 is the smallest and one of the most advanced in the Velodyne 3D LiDAR range, preserving calibrated reflectivity measurements and in real-time at 360 º. Figure 4.4-2 shows the VLP-16 dimensions.

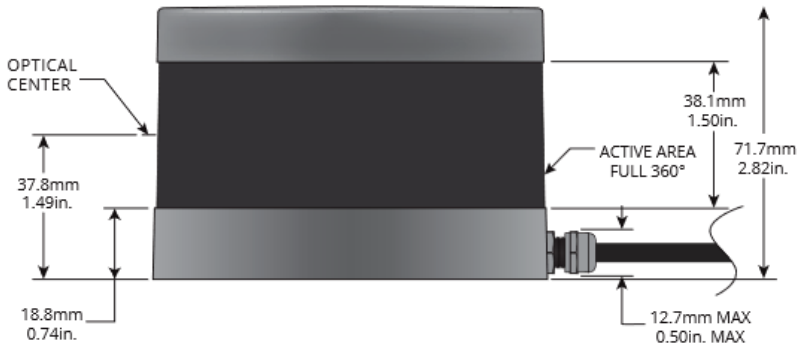


Figure 4.4-2 VLP-16 dimensions overview

On the other hand, Table 4.4-1 shows some of the main specifications of the VLP-16:

Table 4.4-1 Main specifications of VLP-16 sensor

Sensor specifications		**Mechanical/Electrical specifications**	
Channels	16	Typical power consumption [W]	8
Measurement range up to [m]	100	Weight (without cabling) [g]	830
Typical accuracy [cm]	3 *cm*	Operating temperature [º C]	(-10) to (+60)
Vertical FoV [º]	30 º (+15 º to -15 º)	Environmental protection	IP67
Vertical angular resolution [º]	2	**Output features**	
Horizontal FoV [º]	360 º	Simple return mode [points/s]	0.3 million
Horizontal angular resolution [º]	0.1 – 0.4	Dual return mode [points/s]	0.6 million
Rotation rate [Hz]	5 - 20	Ethernet connection [Mbps]	100

4.4.2. Vision sensor: Camera

Artificial or *computer vision* is based on capturing visual information from the environment (both simulation and real-world) to extract relevant visual features. The essential device for obtaining this kind of information is the camera, and one of its main components is the vision sensor (part of the image capture sytem), which is responsible for converting the received light waves into electrical signals, thanks to its photosensitive components. Then, these signals are processed and converted into the images the human beings see. Figure 4.4-3 shows the camera used in this project.

Figure 4.4-3 The ZED camera

Even though there are different types of cameras, in robotics the stereo camera is widely used due to it relies on human binocular vision to capture two images that are processed in

order to obtain the distance and depth of the elements that make up the scene, and thus be able to represent a 3D image. One of its critical points is the correct alignment of the pixels of the image of one camera with the other. In particular, the SmartElderlyCar uses the ZED stereo camera [35] which has two sensors with a baseline of 12 cm. Using its two "eyes" and through triangulation, the ZED provides a three-dimensional understanding of the scene, allowing the target application to become space and motion aware. Table 4.4-2 illustrates some of its main specifications:

Table 4.4-2 ZED camera main specifications

Camera specifications	
Sensors	2 CCD sensors 4M pixels per sensor with large 2-micron pixels Native 16:9 format for a greater horizontal FoV
Stereo baseline [cm]	12
Depth range [m]	0.5 – 20
Technology	Real-time depth-based visual odometry and SLAM
Lens	Wide-angle all-glass dual lens with reduced distortion FoV: 90 º (Horizontal) x 60 º (Vertical) f/2.0 aperture
Connectivity	USB 3.0 port with 1.5m integrated cable Power via USB: 5V / 380 mA
Operating temperature [º C]	0 - 45
Weight [g]	159
Main Third-party support	ROS, Unity, Unreal Engine, OpenCV, MATLAB

4.4.3. Positioning sensor: GPS

One of the most important objectives of a self-driving car is to keep its position and localization throughout the whole navigation. This outdoor information is obtained thanks to the GPS (*Global Positioning System*).

The GPS is a system that allows to obtain the position in any point of the Earth with high precision. Its operation is based on triangulation (Figure 4.4-4) thanks to the network of 24 satellites that orbit the Earth at a height of 20,180 km covering its entire surface. The GPS receiver takes the signal from the satellites indicating its position and time and obtains the time it takes for the signal to arrive to calculate by using triangulation the distance of each satellite to the measurement point.

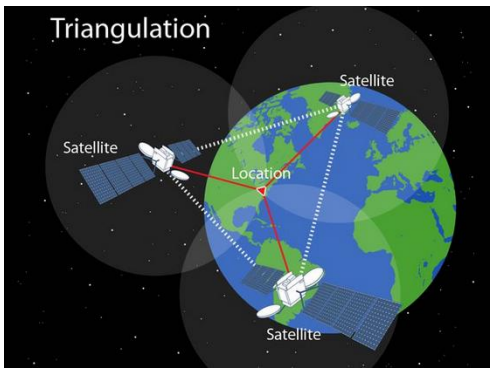


Figure 4.4-4 GPS Triangulation process

Mathematically, 4 satellites would be enough to determine the exact position of Earth, since the geometric place of the space points that are equidistant from each satellite is a sphere, and, therefore, it is the intersection of 4 spheres that allows to obtain a point. However, in a self-driving application, positioning is critical, so high accuracies are required and the detection of a greater number of satellites is required to achieve this high accuracy.

In order to obtain the global positioning, the SmartElderlyCar uses a multi-constellation system (multi-GNSS) with Real-Time Kinematic (RTK) positioning solution. This module is directly integrated in the back of the car and is made up by two elements (Figure 4.4-5 (a)):

Differential Topcon Hiper Pro GPS [36] + Receiver configured as rover and a local base station with the purpose of generating differential corrections for the rover. The rover can obtain data from both GLONASS (Russian global system positioning, homologous to the GPS system managed by USA) and GPS to provide a more robust solution than standard GPS by increasing the number of visible satellites.

(a) (b)

Figure 4.4-5 (a) Differential Topcon Hiper Pro GPS configured as rover and base (b) Choke-Ring Antenna as local base station

Due to the fact that autonomous vehicles demand real-time positioning, the use of differential corrections allows to provide information at a frequency of 10 Hz in order to improve the required accuracy. Furthermore, the local base station (located on the roof of

the Escuela Politécnica Superior – UAH) is based on Choke-Ring Antenna (Figure 4.4-5 (b)), specifically chosen to deal with multipath, connected to a second Topcon Hiper Pro GPS + receiver that provides these differential corrections. Finally, these differential corrections are published over Internet by using standard open source software in the vehicle using a GPRS link via radio. Table 4.4-3 illustrates some of its main features:

Table 4.4-3 Topcon Hiper Pro GPS main specifications

GPS specifications	
Receiver type	Euro-112T (HGGDT)
Standard channels	20 (GPS, Differential and GLONASS)
Operation Time [h]	+14
Power consumption [W]	< 4.2
Antenna type	Central mount UHF
Wireless communication	Bluetooth
Communication ports	x2 (RS2R2)
Output frequency [Hz]	20

Figure 4.4-6 depicts a handcrafted rack placed on the roof of the car with all mentioned sensors integrated and aligned. It can be appreciated that apart from above mentioned sensors, there is a router (in order to provide Internet to the vehicle), a switch (to enable sensor communication), the Velodyne interface that connects the LiDAR with the on-board computer and the Jetson AGX Xavier. A future work of this thesis will be to implement and test the tracking layer into this powerful computer so as to distribute the computational load currently most performed by the on-board computer (an MSI GT62VR-7RE i7-7700HQ), and compare the MOT obtain results with respect to the current HW approach.

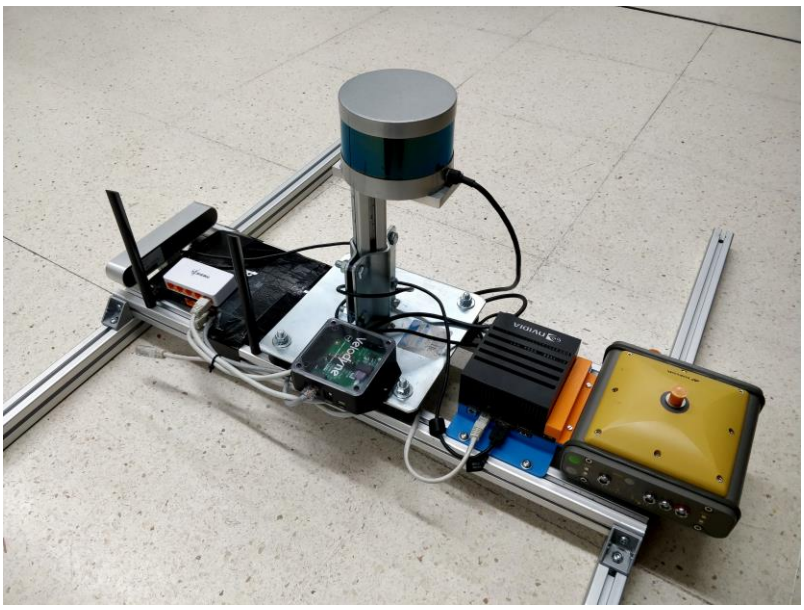


Figure 4.4-6 Handcrafted rack with the main sensors of the SmartElderlyCar

To finish this Sensors section, Figure 4.4-7 illustrates the sensor frames orientation and position which is essential to understand the subsequent sensor fusion dealt in Chapter 5. It must be regarded that this sensor configuration is maintained both in simulation and real-world in order to develop and test SW approaches in simulation and then plug-and-play in the real prototype so as to check new techniques, not requiring additional transformations.

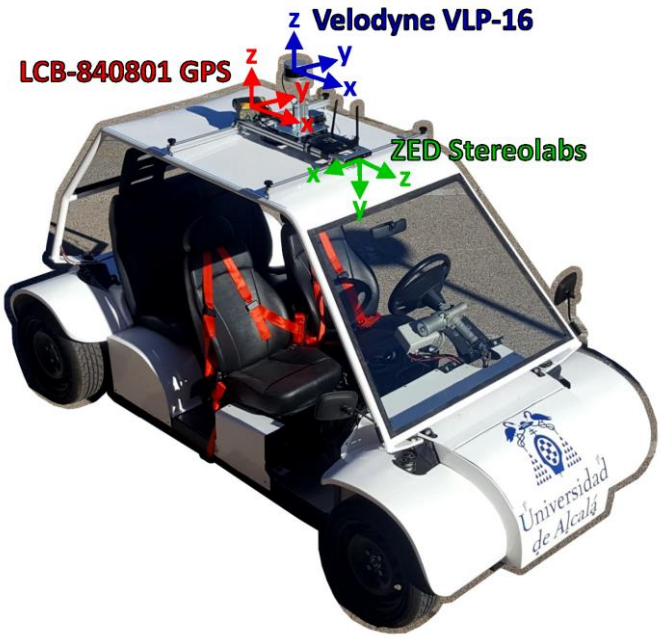


Figure 4.4-7 Frames Orientation and position of the main sensors in the vehicle

4.5. Simulation environments

Most used 3D simulators in the field of robotics are V-REP [38] and Gazebo [43], because of their ease integration in ROS. Other simulation environments are Microsoft Airsim *[44]* (initially designed for drones but recently update so as to include autonomous vehicles), ROS development studio [43] (based 100 % on Cloud, so a system of gym computers allows the parallel training of as many as required) and CARLA [31], which is expected to be the reference open-source simulator for autonomous vehicles based on Unreal engine, with a recent release of its ROSbridge.

The SmartElderlyCar project has been under simulation development for three years (2016-2018) using the V-REP simulator but currently CARLA simulator is used in order to get more challenging situations to improve the robustness and reliability of the prototype (both in simulation and real-world), especially in terms of perception of the environment (which is one of the most attractive features of CARLA). Despite the fact that results of this master thesis are based on [42] and , it is required to show a brief background of the project integration in V-REP (Figure 4.5-1) to understand the validation results carried out with the traditional tracking techniques and the modifications performed in CARLA regarding V-REP.

Figure 4.5-1 V-REP logotype

V-REP is a multiplatform simulation software developed by Coppelia Robotics GmbH. With integrated development, is based on a distributed control architecture: each model/object can be individually controlled via embedded script, a plugin, ROS nodes, BlueZero nodes, remote API clients or a custom solution. This makes V-REP very versatile and suitable for multi-robot applications. A fully functional free version is available for researchers.

The drivable area is modelled both geographic and topologically by using the lanelet approach (Figure 4.5-2) presented in [37] and OpenStreetMap service. Lanes and connections among them are manually delimited, including regulatory traffic information, to generate an enriched map useful for navigation. While in V-REP the simulation was limited to the UAH campus, due to the complexity of mapping other places and manually creating the environment, this master thesis has developed a code in order to transform from OpenDrive syntax (in which CARLA is based on) to the JOSM format (in which lanelets approach is based on) so that CARLA maps fit the needs of the SmartElderlyCar architecture. This code is exposed and commented in Chapter 5. In both cases, to create the JOSM map,

WGS84 are used (made up by latitude, longitude and height) whilst the SmartElderlyCar works in Cartesian coordinates (UTM) relative to an origin (in the UAH campus, it corresponds roughly to the center of the campus, but in CARLA maps sometimes the origin is referred to a corner and other times is referred to the center).

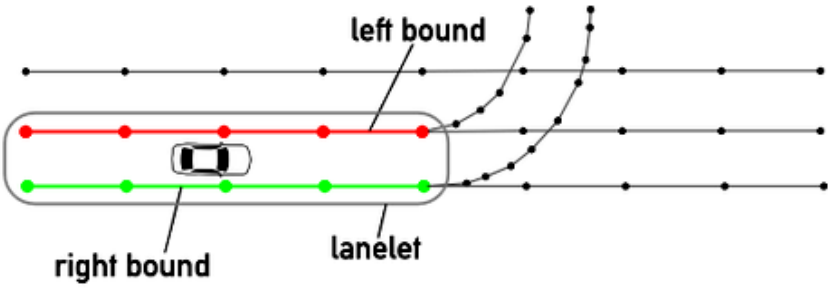


Figure 4.5-2 Map composition based on lanelets

Once the lanelets map is available, the map manager module loads the map and is in charge of planning a new path as a sequence of lanelets. Each lanelet is defined by the borders named ways. Besides the path, the map manager module serves other queries from other modules related to the map, such as providing the contiguous lanes for the overtake maneuver, the lanes of an intersection to the event monitor for the cross intersection maneuver and it should also provide the position of regulatory elements. For the navigation, three different planners are applied. First, a lanelet path is obtained using an A* algorithm [45] from the lanelet maps. Then, as commented above, a global path planner calculates an executable route by the car that tries to go in the middle of the lanes using the Pure Pursuit approach, and finally a local navigation algorithm based on BCM is executed to perform different behaviours following this path and avoiding unexpected obstacles.

4.6. Simulating use cases for the UAH Autonomous Electric Car

To finish this chapter, this section deals with the a paper associated to this master thesis titled “Simulating use cases for the UAH Autonomous Electric Car”, for publication in the ITSC (International Transportation Systems Conference) 2019 and the tracking approach used to perform the different use cases, known as Precision-Tracking [46]. [42] shows a deeper explanation of this paper.

Precision-Tracking [46] is a 3D tracking method used in real spaces that combines 3D shape using a probabilistic framework in which it makes use of the shape information, colour (if available) and motion cues so as to accurately track moving objects in real-time. It allocates computational effort based on the shape of the posterior distribution. Starting with a course approximation to the posterior, precision-tracking approach successively refines this distribution, increasing in tracking accuracy over time. It is able to robustly handle changes

in viewpoint, occlusions and lighting variations for moving objects of a variety of shapes, sizes and distances.

It uses a grid-based method to sample velocities from the state space. While traditional grid-based methods are often used in SLAM (Simultaneous Localization and Mapping) techniques, they are slow for Multi-Object Tracking and so not suitable for self-driving purposes, this approach allows fine-sampling on a large grid thanks to use of the ADH (*Annealed Dynamic Histograms)* method, based on histograms. It starts by sampling from the state space at a coarse resolution using a posterior distribution over velocities. In this way, over time the sampling resolution increases and the probability distribution is strengthened (*annealed*), so the approximate distribution approaches the true posterior. Hence, the current approximation to the posterior may be return, with tracking resolution based on the needs of the application.

In the context of the SmartElderlyCar project, the precision-tracking approach takes as input the 3D clusters proposals after merging the information of a LiDAR point cloud and the semantic segmentation of the scene by using the ERFNet (Efficient Residual Factorized ConvNet) [68], as shown in Figure *4.6-1*.

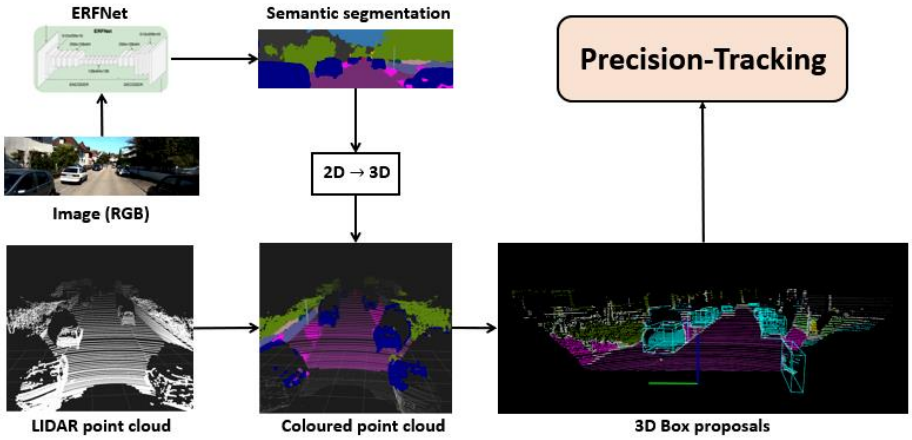


Figure 4.6-1 Precision-Tracking approach in the SmartElderlyCar (Real world)

On the other hand, in simulation, we take advantage of the V-REP characteristics, objects are mainly monocolor (Figure 4.6-2), to implement semantic segmentation.

Figure 4.6-2 From Left to right, V-REP car and pedestrian models

In both cases (simulation and real-world), there must be a projection of the colour information (provided by the image) onto the 3D Point Cloud. It can be performed by using the algorithm shown in Figure 4.6-3. A more detailed overview of the code can be found in Code of Interest C-1.

Algorithm 1 Projecting the semantic segmentation into the 3D Point Cloud

Input: 3D Point Cloud, ROS Semantic Segmentation Image, Camera Info
Output: 3D Coloured Point Cloud

1: $image\_msg$ (ROS input image) $\longrightarrow$ $image\_ptr$ (OpenCV image pointer)
2: $image\_ptr \longrightarrow$ img (OpenCV matrix)
3: ROS input point cloud $\longrightarrow$ pcl::PointXYZRGB pcl_cloud
4: Lookup for target ($/stereo\_cam\_LR$) to source frame ($/lidar\_pts$) transform
5: pcl_ros::transformPointCloud ($*pcl\_cloud, *trans\_cloud, transform$)
6: for pt $= trans\_cloud \rightarrow$ points.begin() $\longrightarrow$ pt $< trans\_cloud \rightarrow$ points.end()
7: $*pt \longrightarrow pt\_cv$ (OpenCV 3D point)
8: uv (OpenCV 2D point) $= cam\_model \cdot$ project3dToPixel(pt_cv)
9: if (uv.x $> 0 \wedge$ uv.x $<$ img.cols $\wedge$ uv.y $> 0 \wedge$ uv.y $<$ img.rows $\wedge$ $*pt.z \geq 0$)

$$(*pt).r = img.at < cv :: Vec3b > (uv)[0];$$
$$(*pt).g = img.at < cv :: Vec3b > (uv)[1];$$
$$(*pt).b = img.at < cv :: Vec3b > (uv)[2];$$

10: else

$$(*pt).r = 40.0;$$
$$(*pt).g = 40.0;$$
$$(*pt).b = 40.0;$$

Figure 4.6-3 Algorithm used to project the semantic segmentation into the 3D Point Cloud

It must be considered that the precision-tracking approach depends crucially on the results obtained during the detection process since they are used as input, as shown in Figure 4.6-1. During the detection process, both the measurement uncertainties of the sensors and the occlusions caused by changes in point-of-view must be dealt with, which causes the laser to not reflect some points correctly. Moreover, sensory is not perfect. One of the key points related with detection problems is that a 16-channels 3D LiDAR is used in the real prototype, which in comparison with other 32 or 64-channels 3D LiDAR, has less FoV and angular resolution although its has a lower a price and requires less computational effort, which makes it attractive to develop algorithm in order to improve detection. On the other hand, CARLA and V-REP they are able of simulating several types of LiDAR, with a greater resolution than offered by 16-channels LiDAR.

In terms of precision-tracking, there is a trade-off in order to choose the minimum number of points that should be considered for an object cluster. Limiting the minimum size to a small number of points (to consider a cluster as a relevant object) has the advantage of being able to determine the presence of an object that is relatively at a great distance. However, if assuming small number of points, it is more likely to fall into a detection error which would cause. In that sense, [42] assumes a minimum cluster size which guarantees a more accurate detection when the object is relatively close.

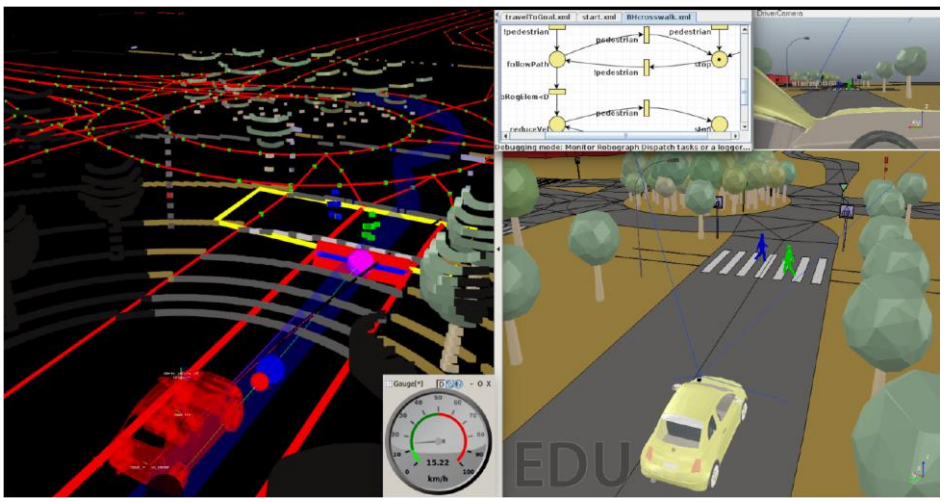


Figure 4.6-4 Pedestrian crossing simulation example

According to [42], several tests were carried out in the simulated Campus of the UAH. In order to perform each use case (Pedestrian Crossing, STOP, Give Way, Traffic Light, Adaptive Cruise Control (ACC) and Overtaking), the control layer takes the information of the lanelets and nearest regulatory elements, provided by the map manager module, so as to generate some velocity command for the low-level control, following the use case commanded by the executive layer through the respective Petri net. Figure 4.6-4 illustrates a simulation example where left image shows the R-VIZ simulator illustrating the point cloud detection and right image shows the V-REP simulator faced by the ego-vehicle sensors.

Table 4.6-1 shows a summary of the main features, including inputs, outputs and number of elements for the main Petri Nets. As observed, the first Petri Net (*Background*) is a net running always in background. This net is waiting for a message from the user requesting to execute some of the car tasks the car can carry out. *Selector PN* decides which behaviour to run, according to the traffic situation, and monitors the execution of the behaviours. Each one of the other Petri nets implements the behaviour that corresponds to a particular traffic situation use case.

Petri Net	Inputs	Input modules	Outputs	Output module	Nº nodes	Nº transitions
Background	Man/auto goToPoint	GUI User GUI User	Run Selector Stop selector	RG Dispatch RG Dispatch	8	9
Selector PN	Reg. Element (STOP ..) Dist. Reg Element End Reg. Element Reg. Element (STOP ..) End Reg. Element FrontCarVel Odom	Map manager Map manager Map manager Event monitor Event monitor Event monitor Base	Run PedestrianCrossing Stop PedestrianCrossing Run GiveWay Stop GiveWay Run STOP ...	RG Dispatch RG Dispatch RG Dispatch RG Dispatch RG Dispatch ...	21	29
FollowLane	Traffic sign (max speed, ...) Force End	Event monitor RG Dispatch	SetMaxVel STOP	Local Navigator Local Navigator	6	8
Pedestrian Crossing	NoPedestrian Pedestrian DistToPedestrianCrossing PedestrianCrossingOver Force End stopped	Event monitor Event monitor Map manager Map manafer RG Dispatch Local Navigator	WatchforPedetrians SetMaxVel StopAtPoint	Event Monitor Local Navigator Local Navigator	10	13
STOP	SafetoMerge NotSafetoMerge DistToStop StopOver Force End stopped	Event monitor Event monitor Map manager Map manager RG Dispatch Local Navigator	CheckSafeMerge SetMaxVel StopAtPoint	Event Monitor Local Navigator Local Navigator	9	12
GiveWay	SafetoMerge NotSafetoMerge DistToStop StopOver Force End stopped	Event monitor Event monitor Map manager Map manager RG Dispatch Local Navigator	CheckSafeMerge SetMaxVel StopAtPoint	Event Monitor Local Navigator Local Navigator	9	12
Traffic Light	CheckForTrafficLight SafetoMerge NotSafetoMerge DistToStop StopOver Force End stopped	Event monitor Event monitor Event monitor Map manager Map Manager RG Dispatch Local Navigator	CheckForTrafficLight CheckSafeMerge SetMaxVel StopAtPoint	Event Monitor Event Monitor Local Navigator Local Navigator	10	12
Adaptive Cruise Control (ACC)	Current Velocity FrontCarVel DistToFrontCar	Map manager Event monitor Map manager	SetMaxVel	Local Navigator	4	6
Overtake	FrontCarVel SafeChangeLeftLane NotSafeChangeLeftLane SafeChangeRightLane NotSafeChangeRightLane Odom OnLeftLane OnRightLane	Event monitor Event monitor Event monitor Event monitor Event monitor Base Local Navigator Local Navigator	CheckLeftLane CheckRightLane SwichtLeftLane SwitchRightLane CheckRightLane	Event Monitor Event Monitor Local Navigator Local Navigator Event Monitor	14	21

Table 4.6-1 Table Main features of the SmartElderlyCar Petri Nets

Chapter 5. Architecture proposal for Deep Learning based Multi-Object Tracking

5.1. Introduction

This chapter aims to show the architecture proposal for Deep-Learning based Multi-Object Tracking. Due to system requirements, the architecture proposal must offer overall two key features: real-time operation and robustness to a number of ambient conditions which typically degrade performance. As mentioned in previous chapters, these object tracking challenges, mainly in Visual Object Tracking (VOT) include partial occlusion, camera modelling errors, photometric changes (lightning, shadows, etc.) or incorrect edge matching.

After trying to run several times deep-learning approaches for object tracking, several limitations were found about previously mentioned deep-learning state-of-the-art approaches for object tracking:

- *GOTURN*: This deep-learning based tracker presents lack of motion information. Since motion information is not integrated in the two-frame model, if the system (for example, the autonomous vehicle) is tracking an object (another car or pedestrians) moving in one direction and gets partially occluded by a similar object moving in the other direction, there is a great change and the tracker will latch onto the wrong object.

- *MV-YOLO:* The code is still not published. In addition, [63] mentions that MV-YOLO does not support currently Multi-Object Tracking but only Single-Object Tracking, what is not appropriate for self-driving applications (the main purpose of this master thesis).

- *MDNet:* The network is not thought for real-time purposes, at least for the moment. It was evaluated on two datasets, Object Tracking Benchmark (OTB) and Visual Object Tracking 2014. As mentioned above, the algorithm is implemented in MATLAB using MatConvNet toolbox, running at around 1 fps with eight cores of 2.20 GHz Intel Xeon E5-2660 and an NVIDIA Tesla K20m GPU. This framerate of 1 fps is not enough for self-driving applications, since what it means is that the network is estimating the position of the object every second, but at certain velocities, one second gives rise to great distances and the vehicle could crash an obstacle.

- *ROLO:* The code is totally deprecated in terms of Python and Tensorflow.

- *Re3:* This approach models both appearance and motion variations using RNNs and achieves comparable results on several object tracking benchmarks. However, Re3 only considers short-term variations [48], is notably affected by partial occlusion and requires manual resetting of RNN states every 32 frames.

Due to limitations of the current deep learning proposals for object tracking (moreover Multi-Object Tracking) unable to be used in an AV application, this master thesis proposes a fusion between VOT performed in 2D images and 3D boxes detected by the LiDAR, following the architecture shown in Figure 5.1-1. VOT is carried out by using CenterNet [49] (one of the most efficient and fastest CNNs right now to detect objects in images, published in 2019, even faster than YoloV3 [51]) and Deep SORT [2] (based on the real-time tracker SORT [27] with a deep association metric). Image bounding boxes are projected into the *Bird Eye View* (BEV) plane using the calibration matrix and taken a fix depth for each bounding box. However, this 3D recovery process is not so accurate. To improve precision, LiDAR 3D boxes proposals obtained from the LiDAR point cloud, after a clustering and KD-Tree process, are projected to the BEV plane. Proposals coming from VOT and LiDAR are fused using a simple algorithm consisting in evaluating the overlapping in the two domains. If the overlapping is higher than a certain *threshold*, the LiDAR proposal pass to the output. Otherwise, the proposal is discarded. Future works will include BEV VOT proposals if the object detection is performed far away from the vehicle and there are not enough points to cluster the point cloud at that distances.

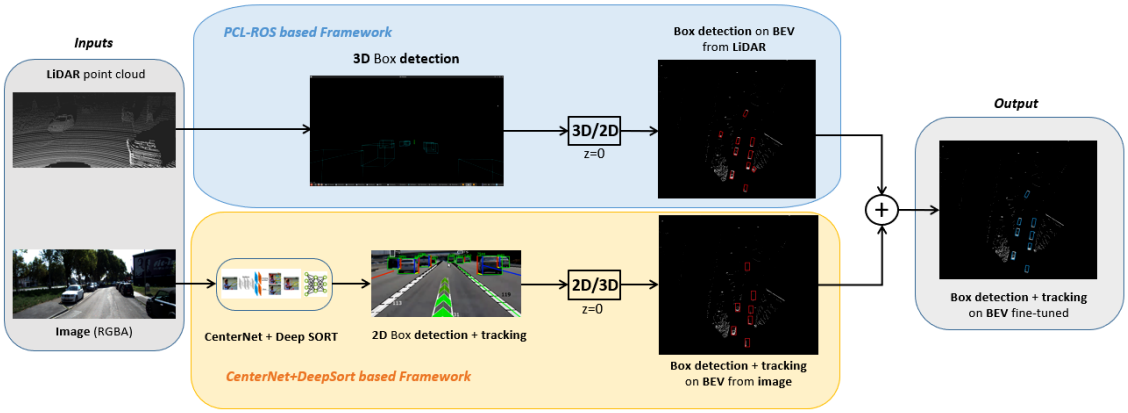


Figure 5.1-1 Architecture proposal for Multi-Object Tracking

The following sections deal with the main modules of this architecture, both those related with the *CenterNet+DeepSORT* based framework and the *PCL-ROS* based framework.

5.2. Centernet

The first step to carry out object tracking, as mentioned throughout this work, is the detection. The *CenterNet* approach [49] is a CNN that detects each object as a triplet (top-left corner, center estimation and bottom-right corner), rather than a pair (only the corners) of keypoints, which improves both precision and recall.

This technique is based on *CornerNet* approach [50]. CornerNet represents each object by a pair of corner keypoints, which bypassed the need of anchor boxes and achieves the state-of-the-art-one-stage object detection accuracy. Nevertheless, the CornerNet performance is restricted by its relatively weak ability of referring to the global information of an object. That is to say, since each object bounding box is constructed by a pair of corners, the algorithm is sensitive to detect the boundary of objects so not being aware of which pairs of keypoints should be grouped into objects. This weakness gives rise to some incorrect bounding boxes, most of which could be easily filtered out with complementary information, such as the aspect ratio.

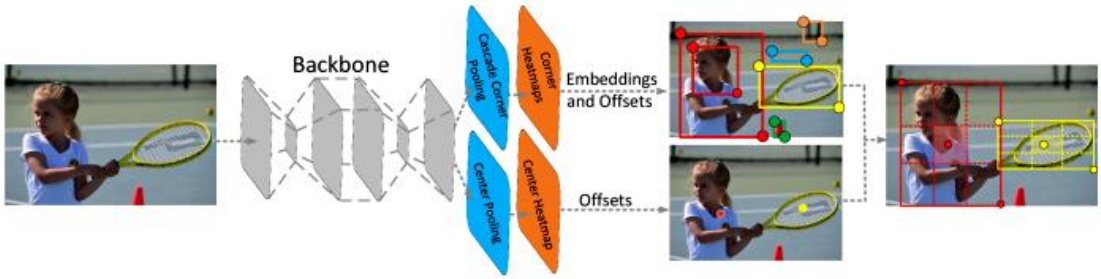


Figure 5.2-1 CenterNet architecture

To address this weakness, CornerNet is equipped with the ability of perceiving the visual patterns within each proposed region in order to identify the correctness of each bounding box by itself. In that sense, CenterNet is a variation of CornerNet that explores the central part of a proposal (region that is close to the geometric center) with one extra keypoint. The statement is very simple: If a predicted bounding box has a high IoU (Intersection over Union) with the groundtruth box, then, the probability that the center keypoint in its central region is predicted as the same class id is high, and vice versa. In other words, if is determined if the proposal is indeed an object by checking if there is a center keypoint of the same class falling within its central region. Since the approach only pays attention to the center information, the cost in minimal.

The overall network architecture is shown in Figure 5.2-1. A convolutional backbone network applies center pooling and cascade corner pooling to output a center keypoints heatmap and two corner heatmaps, respectively. Each object is represented by a center keypoint and a pair of keypoints. CenterNet uses the method proposed in CornerNet in order to generate top-k bounding boxes. Nevertheless, to filter the incorrect bounding boxes, CenterNet leverages the detected center keypoint and follows this procedure:

1. Select top-k center keypoints according to their scores.

2. Use the corresponding offsets to remap these center keypoints to the input image.

3. Define a central region for each bounding box and check if the central region contains center keypoints.

4. If a center keypoint is detected in the central region, the bounding box is preserved and the score of the bounding boxes will be replaced by the average scores of the keypoint triplet. If there are no center keypoints detected in its central region, the bounding box is removed.

However, in order to incorporate this object detector, since to define the bounding box and detect the object largely depends on the size of the central region in the bounding box. For example, smaller central regions lead to a low recall rate for small bounding boxes, while larger central regions lead to a low precision for large bounding boxes. Both in simulation and of course in the real-world, an object must be tracked (and so previously detected) until it disappears from scene. Even if it is at a certain distance (small size in the scene) but still on-road, it must be detected since is relevant. In that sense, CenterNet is excellent because it proposes a scale-aware central region to adaptively fit the size of bounding boxes. The scale-aware central region tends to generate a relatively central region for a small bounding box, while a relatively small central region for a large bounding box.

In conclusion, the main highlights of CenterNet are:

- *Simple:* Use keypoint detection technique to detect the bounding box center point and regress to all other object properties such as bounding box, pose or 3D information.

- *Versatile:* The same framework can work for object detection, multi-person pose estimation with minor modification and 3D bounding box estimation.

- *Fast:* The whole process is included in a single network feedforward.

- *Strong:* The best single model achieves 45.1 AP (Average Precision) on COCO test-dev.

5.3. Deep SORT

Simple Online and Real Time (SORT) [27] tracking is a pragmatic approach to MOT (Multi-Object Tracking) where the main focus is to associate objects efficiently for online and real-time applications. This method represents a lean implementation of a tracking-by-detection framework for the problem of MOT where objects are detected each frame and represented as bounding boxes. SORT performs Kalman filtering (detailed Appendix A**)** in image space and frame-by-frame data association using the Hungarian method [52] with an association metric that measures bounding box overlap. Due to this simple combination, it achieves

favourable performance at high frame rates. For example, on the MOT challenge dataset, SORT with a state-of-the-art detector (Faster R-CNN, [53]) ranks on average higher than Multiple Hypothesis Tracking (MHT) [54]. This method is primarily targeted towards online tracking where only detections from the previous and the current frame are presented to the tracker, in contrast to many batch based (i.e., offline) tracking approaches ([28] [29]).

However, while achieving overall good performance in terms of accuracy and tracking precision, SORT algorithm returns a high number of identity switches since the employed association metric is only accurate when state estimation uncertainty is low (that is, uncertainty in Kalman filter). Therefore, SORT presents a deficiency in tracking through occlusions which they typically appear in frontal-view camera scenes. In order to overcome this drawback, [2] replaces the association metric with a more informed metric that combines appearance information and motion using a deep learning net. Through the integration of this network, the robustness is increased against misses and occlusions while keeping the system efficient, applicable to real-time scenarios and easy to implement.

Deep SORT [2] is one of the most widely used and elegant object tracking framework as an extension to SORT. Now, it is described briefly each of the four core components of this Deep SORT system. Figure 5.3-1 depicts a flowchart for this framework.

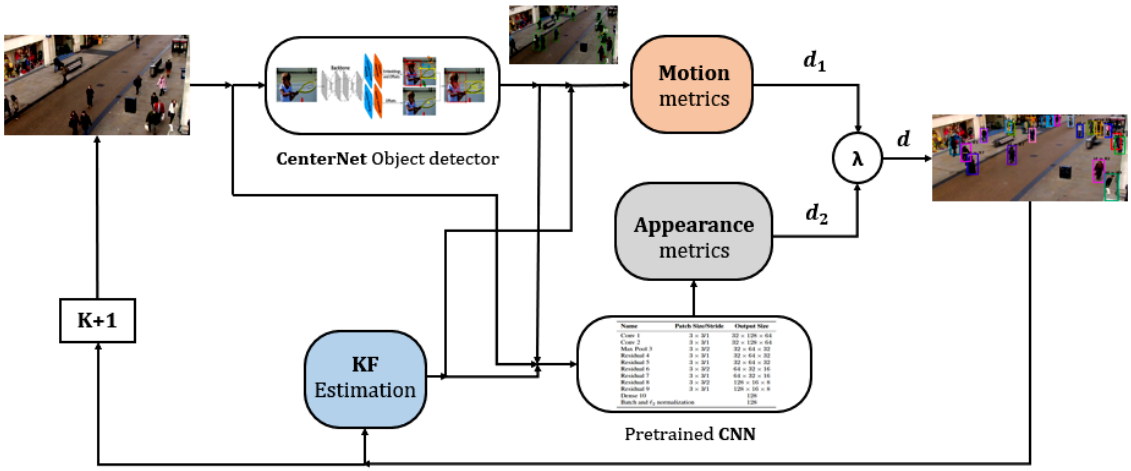


Figure 5.3-1 Flowchart of the CenterNet+Deep SORT framework

5.3.1. Track handling and State Estimation

This component is very similar to the SORT proposal. It is assumed a general tracking scenario where the camera is uncalibrated and there is no ego-motion information available. Therefore, the tracking scenario is defined on the eight-dimensional state space $(u, v, \gamma, h, \dot{x}, \dot{y}, \dot{\gamma}, \dot{h})$ that contains the bounding box center position (u, v), aspect ratio γ, height h and their respective velocities in image coordinates. Then, a standard Kalman filter with constant velocity motion and linear observation model is used, where the bounding box coordinates (u, v, γ, h) are taken as direct observations of the object state. For each track k it is counted the number of frames since the last successful measurement association

a_k. During Kalman filter prediction the counter is increased and reset to 0 if the track has been associated with a measurement. Otherwise, tracked objects whose associated counter exceed a predefined maximum age A_{max} are considered to have left the scene and are deleted from the track set. On the other hand, new track hypotheses are initiated for each detection that cannot be associated to an existing tracked object. Furthermore, these new tracks are classified as tentative during their first three frames. If these new tracks hypotheses are not associated to an existing tracked object within their first three frames, they are deleted.

5.3.2. Assignment Problem

The second component deals with the association of predicted Kalman states and newly arrived measurements. It can be solved using the Hungarian algorithm, which is a combinatorial optimization algorithm that solves the assignment problem in time. In order to solve this assignment problem, motion and appearance information are integrated through combination of two appropriate metrics.

To incorporate motion information, the (squared) Mahalanobis distance between newly arrived measurements and predicted Kalman states is used:

$$d_1(i,j) = \left(d_j - y_i\right)^T S_i^{-1} \left(d_j - y_i\right) \tag{5.1}$$

Where the y_i denote the projection of the i-th track distribution and d_j the j-th bounding box detection. The Mahalanobis distance takes the estimation uncertainty for the association of the detections with the tracker estimations. In addition, using this motion information is possible to exclude unlikely associations thresholding the Mahalanobis distance at 95 % confidence interval, computed from the inverse X^2 distribution:

$$b_{i,j}{}^{(1)} = \mathbf{1}[d_1(i,j) \leq t^{(1)}] \tag{5.2}$$

In this case, the decision to associate the i-th track with the j-th detection is admissible if the Mahalanobis distance is lower or equal than its threshold $t^{(1)} = 9.4877$.

However, the rougher the estimation of the object location (obtained from the Kalman filter) is, the worse works the Mahalanobis distance, since the motion uncertainty is high (for example, in unaccounted camera motion that introduces rapid displacements). Then, a second metric is introduced into the assignment problem. For each bounding box detection d_j, the appearance descriptor r_j is computed. Furthermore, Deep SORT computes a gallery $R_k = \{r_k^{(i)}\}$ from k = 1 to the last 100 associated appearance descriptors for each track k, named L_k.

Then, this second metric measures the smallest cosine distance between the i-th and the j-th detection in appearance space:

$$d_2(i,j) = min\{1 - r_k^T r_k^{(i)} \mid r_k^{(i)} \in R_i\} \tag{5.3}$$

Again, a binary variable is introduced so as to indicate if the association is admissible comparing this result with a suitable threshold:

$$b_{i,j}{}^{(2)} = \mathbf{1}[d_2(i,j) \leq t^{(2)}] \tag{5.4}$$

In combination, both metrics (motion and appearance) complement each other by serving different aspects of the assignment problem. On the one hand, the Mahalanobis distance provides information about possible object locations based on motion (very useful for short-term predictions). On the other hand, the cosine distance considers appearance information that are particularly useful to recover identities after long-term occlusions, where motion is less discriminative. This is an excellent solution for real application, like self-driving, where a car may be partially occluded by another object for a relative long-term for example if they have the same velocity. To build the association problem, both metrics are combined using a weighted sum, where the influence of each metric on the combined association cost can be controlled through hyperparameter λ:

$$c_{i,j} = \lambda d_1(i,j) + (1-\lambda) d_2(i,j) \tag{5.5}$$

Where finally an association is considered admissible if it is within the gating region of both metrics (that is, both decisions $b_{i,j}{}^{(1)}$ and $b_{i,j}{}^{(2)}$ are equal to 1):

$$b_{i,j} = \prod_{m=1}^{2} b_{i,j}^{(m)}$$

5.3.3. Matching Cascade

When an object is occluded for a longer period of time, subsequent Kalman filter predictions increase the uncertainty associated with the object location. For that reason, probability mass spreads out in state space and the observation likelihood decreases. The association metric should consider this spread of probability mass by increasing the measurement-to-track distance. On the other hand, counterintuitively when two tracked objects compete for the same detection, the Mahalanobis distance favors large uncertainty since it effectively reduces the distance in standard deviations of any detection towards the projected track mean. This is a problem: It can lead to increased track fragmentations and unstable tracks. In that sense, Deep SORT introduces a matching cascade algorithm that gives priority to more frequently seen objects to consider in a proper way the probability spread in the association likelihood. Figure 5.3-2 shows the matching cascade pseudo-algorithm.

Input: Track indices $\mathcal{T} = \{1, \ldots, N\}$, Detection indices $\mathcal{D} = \{1, \ldots, M\}$, Maximum age $A_{\max}$

1: Compute cost matrix $\boldsymbol{C} = [c_{i,j}]$
2: Compute gate matrix $\boldsymbol{B} = [b_{i,j}]$
3: Initialize set of matches $\mathcal{M} \leftarrow \emptyset$
4: Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$
5: **for** $n \in \{1, \ldots, A_{\max}\}$ **do**
6: Select tracks by age $\mathcal{T}_n \leftarrow \{i \in \mathcal{T} \mid a_i = n\}$
7: $[x_{i,j}] \leftarrow \text{min\_cost\_matching}(\boldsymbol{C}, \mathcal{T}_n, \mathcal{U})$
8: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i,j) \mid b_{i,j} \cdot x_{i,j} > 0\}$
9: $\mathcal{U} \leftarrow \mathcal{U} \setminus \{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\}$
10: **end for**
11: **return** $\mathcal{M}, \mathcal{U}$

Figure 5.3-2 Matching Cascade algorithm to evaluate the age of the tracked objects

Its inputs are the set of track T and detection D indices in addition to the maximum age A_{max} (maximum time considered for a tracked object to have left the scene and delete from the track set). Lines 1 and 2 compute the association cost matrix and admissible associations matrix. Then, using the Hungarian algorithm, it is iterated over track age *n* (from 1 to A_{max}) to solve the linear assignment problem for tracked objects of increasing age. In line 6 a subset of n tracks, named T_n is selected since it has not been associated with a detection in the last *n* frames. In line 7 the linear assignment problem between unmatched detections *U* and not-associated tracks T_n. In lines 8 and 9 the set of matches and unmatched detections is updated, which is returned in line 11. It is important to consider that the matching cascade gives priority to tracked objects of smaller age, that is, those that have been seen more recently.

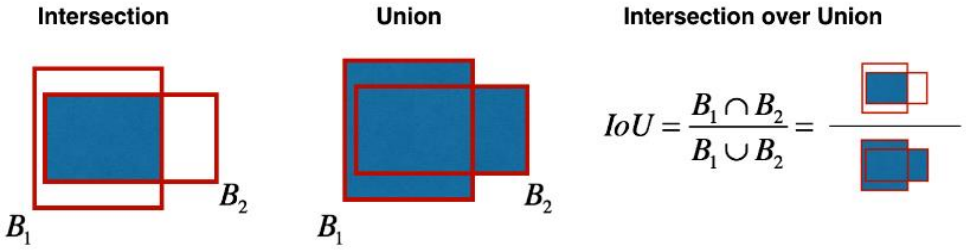


Figure 5.3-3 Intersection over Union representation

Finally, the Intersection over Union (IoU) (Figure 5.3-3) is performed, as proposed in SORT, on the set of unconfirmed and unmatched tracks of age n = 1. This helps to increase robustness against erroneous initialization of the Kalman filter and take into account for sudden appearance changes, such as partial occlusion with static scene geometry.

5.3.4. Deep Appearance Descriptor

At this moment, the above steps represent the SORT algorithm, that is, an object detector providing detections (CenterNet in the present work), Kalman filter tracking them and

providing missing tracks and the Hungarian algorithm solving the association problem. However, despite the effectiveness of Kalman filter, it fails in many of the real-world scenarios where associated VOT problems take place, such as occlusions, lighting or different point-of-view, in other words, one fundamental thing that SORT algorithm misses, which human beings use all the time in tracking, is a visual understanding of the detected bounding boxes. In order to overcome this drawback, [2] introduces another distance metric based on the *appearance* of the object. Using deep features allows Deep SORT technique to track much better in cases where people are occluding or are very close in the image. The idea is to obtain a vector that can describe all the features of a given image.

Original version of [2] employed a CNN trained on a large-scale person re-identification dataset (Mars dataset [57]) that contains over 1,150,000 images of 1,261 pedestrians, so it is well suited for deep metric learning in a people tracking context. A classifier over this mentioned dataset was trained till it achieves a reasonably good accuracy and then strip the final classification layer. That feature vector becomes the *appearance* descriptor of the object. Moreover, currently not only supports people re-identification but also the COCO-dataset labels (cars, bicycles, etc.) which provides a more robust tracking in terms of MOT for self-driving applications.

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
Batch and ℓ_2 normalization		128

Figure 5.3-4 Overview of the Deep Appearance descriptor CNN architecture

This CNN architecture is shown in Figure 5.3-4. The input constitutes a 32 x 128 x 3 image corresponding to each of the bounding boxes (crop) detected in the image. It is actually a wide residual network [58] with two convolutional layers followed by six residual blocks. The *Dense 10* layer will be the appearance feature vector for the given crop. The network has 2,800,864 parameters and one forward pass of 32 bounding boxes which takes 30 ms on a Nvidia GeForce GTX 1050 mobile GPU. Considering that the available GPU of the SmartElderlyCar is better (1070 GTX), this network is well suited for the vehicle online tracking purposes. A final batch and l_2 normalization projects features onto the unit hypersphere to be compatible with the cosine appearance metric. Once trained, it is just required to pass all the crops of the detected bounding box (in this case performed by CenterNet) from the image to this network and obtain a *128 x* 1 dimensional feature vector (for each detected bounding box).

In conclusion, a simple distance metric, combined with a powerful DL technique is all it took for Deep SORT to be an elegant and one of the most widespread object trackers. In summary, given an input image, CenterNet detects the relevant objects in the image according to COCO-dataset. By using this bounding box information between the projection of the track distribution (Kalman Filter) and bounding boxes detection, Mahalanobis distance is computed (Motion metrics); on the other hand, using a deep appearance descriptor based on a pretrained CNN, appearance metric is computed as well. By using a weighted sum of these motion metrics and appearance metrics, Deep SORT is able to predict in a very accurate way feature pose of the tracked objects.

Moreover, Figure 5.3-5 shows the algorithm used to perform VOT using CenterNet and Deep SORT approaches. A more detailed overview of the code can be found in Code of Interest C-2 and Code of Interest C-3.

Algorithm 2 Performing Visual Object Tracking using CenterNet and Deep SORT

Input: ROS RGBA image (*img_msg*)
Output: Predicted pose of tracked objects

1: *img_msg* $\longrightarrow$ *image* (OpenCV RGB image)
2: Use CenterNet to obtain preliminary bounding boxes proposal
3: Discard bounding box proposal if score lower than visualization theshold:
if any(bbox[:,4] > opt.vis_thresh):
bbox = bbox[bbox[:,4] > opt.vis_thresh, :]
bbox[:,2] = bbox[:,2] - bbox[:,0]
bbox[:,3] = bbox[:,3] - bbox[:,1]
return bbox[:,:4], bbox[:,4], bbox[:,5]
4: Predict the pose of tracked objects using Deep SORT algorithm:
max_cos_distance = 0.2 ; nn_budget = 100
metric = NNDistanceMetric('cosine', max_cos_distance, nn_budget)
Initialize a tracker with the proposed metric
Generate the deep appearance vector of the bounding box
Run no Non-Maximum Suppresion
Update the tracker
Output bbox identities

Figure 5.3-5 Algorithm used to perform VOT using CenterNet and Deep SORT

5.4. LiDAR clustering

As commented in Chapter 3, working with the whole point cloud can be hard computationally to perform detection and tracking on individual objects. For that reason, the whole point cloud is usually divided into smaller clouds of points (also known as *clusters*) in which each one contains nearby space points that belong to the same object.

In this master thesis, a clustering based on the 3D point cloud data is performed. Despite the fact that it requires a great computational effort than if it was done with 2D data, the results (overall the 3D centroid of the object) are more accurate when incorporating the

vertical component. The implemented method is based on grouping 3D points by using the searching algorithm KD-Tree combined with the cluster extraction based on the Euclidean distance (*Euclidean Cluster Extraction*).

5.4.1. KD-Tree

KD-Tree (K-dimension tree) is a data structure used to organize points in a k-dimensional Euclidean space. It is a binary search tree that only employs perpendicular planes to each dimension, where each node contains a point, leaving all the points crossed by planes. It is usually used for point searches, such as the Nearest Neighbour [59]. As this master thesis works with 3D point cloud data, 3D trees are used (K = 3).

The tree-shaped organization is equivalent to a hierarchical structure divided into levels formed by parent nodes and child nodes, where at each level, the nodes are divided by a plane perpendicular to an axis. The most efficient way to build it is by using the *QuickSort* algorithm, based on taking the median in one dimension and then ordering the rest of the elements (on the right and on the left with respect to a given node) depending on whether they have a greater or lesser value. Figure 5.4-1 shows an example of 3D tree structure:

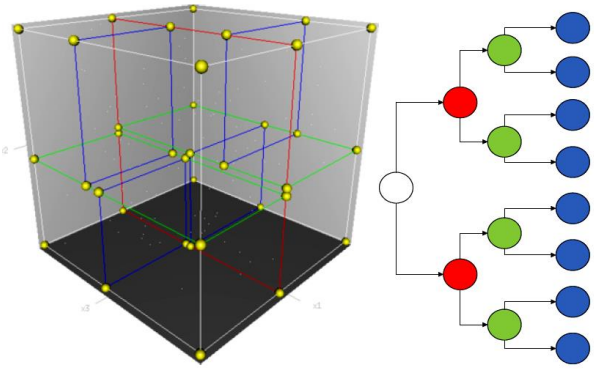


Figure 5.4-1 3D KD-Tree example

In this example, the primitive cell is limited by the white cube. The first division is performed by the red plane (first dimension), dividing the original cell into two subcells. Each of these subcells is divided by the green plane (second dimension), resulting in 4 subcells which are dividing again by the blue plane (third dimension). In this case, with 8 subcells, the leaves of the tree are defined, since all the points, which represent the nodes of the tree, have been covered.

5.4.2. Euclidean cluster extraction

As commented, the *clustering* method is based on dividing the whole point cloud into smaller clouds according to the Euclidean distance based on the nearest neighbours as a result of the 3D division performed by the KD-Tree algorithm. The Euclidean Cluster Extraction works as follow:

1. A KD-Tree is created to process the point cloud data.

2. A list of indices is created.

3. For each point in the cloud:
    a. The set of neighbouring points belonging to a sphere of radius lower than the defined threshold is searched.
    b. For each neighbouring point it is checked whether it has already been evaluated. If not, it is added.

4. The algorithm ends when all points are part of the cluster list.

In order to implement this algorithm, a *EuclideanClusterExtraction* object with the *PointXYZRGB* point type from the PCL library is used. The threshold will be set using the *setClusterTolerance* parameter, which varies depending on the object model detected, in the same way that the minimum and maximum size of each of the clusters (*setMinClusterSize* and *setMaxClusterSize* parameters, respectively). Naturally, the number of detected points for a car will be higher than for a cyclist or pedestrian.

Algorithm 3 Computing the LiDAR clustering using Euclidean Cluster Extraction and KD-Tree

Input: 3D ROS LiDAR Point Cloud (*lidar_msg*)
Output: Relevant clusters

1: *lidar_msg* $\longrightarrow$ *pcl_ptr* (PointCloud<pcl::PointXYZRGB> pointer)
2: *cloud_filtered* (PointCloud<pcl::PointXYZRGB>) = xyz_filter(pcl_ptr)
3: Create the KD-Tree object to process the XYZ filtered cloud:
    tree $\rightarrow$ setInputCloud (**cloud_filtered*)
4: Create Euclidean Cluster Extraction object for cluster extraction:
    ec.setClusterTolerance(1);
    ec.setMinClusterSize(2);
    ec.setMaxClusterSize(25000);
    ec.setSearchMethod(tree);
    ec.setInputCloud(*cloud_filtered*)
    ec.extract(*cluster_indices*)

Figure 5.4-2 Algorithm used to compute the 3D LiDAR clustering using Euclidean Cluster Extraction and KD-Tree techniques

Figure 5.4-2 shows the algorithm developed to obtain the relevant clusters from the detected 3D LiDAR Point Cloud. A more detailed overview of the code can be found in Code of Interest C-4.

5.5. Sensor fusion

Although VOT performs a very accurate 2D tracking, it presents a strong inaccuracy when projecting the bottom position of the proposed bounding box onto the 3D space. To solve

this problem, a sensor fusion is performed. As commented in Chapter 2, *Sensor fusion* is an approach that combines sensory data derived from different sources such the resulting information has less uncertainty that would be possible when these sources were used individually. In terms of AVs, main sensors to perform sensor fusion are camera and LiDAR.

In that sense, this work presents a sensor fusion on BEV between VOT performed in 2D image and projected to the BEV plane and the LiDAR 3D boxes proposals and then projected to the BEV, as shown in Figure 5.1-1. First of all, in order to perform the sensor fusion, BEV projection from 2D tracked objects is carried out.

Algorithm 4 Computing the Image to BEV projection

Input: Bounding Box coordinates of the detected obstacle
Output: BEV pose of the obstacle in LiDAR frame

1: Set the inverse of the camera projection matrix (4x4)
2: Set camera height in the vehicle
3: Store Bounding Box coordinates in camera frame:
$\text{tracked\_obstacle.x}_1 = \text{bb\_coordinates}[0]$
$\text{tracked\_obstacle.y}_1 = \text{bb\_coordinates}[1]$
$\text{tracked\_obstacle.x}_2 = \text{bb\_coordinates}[2]$
$\text{tracked\_obstacle.y}_2 = \text{bb\_coordinates}[3]$
4: Compute BEV pose of the obstacle:
$\text{centroid\_x} = \frac{tracked\_obstacle.x1 + tracked\_obstacle.x2}{2}$
$\text{pixels} = \begin{bmatrix} centroid\_x \\ tracked\_obstacle.y\_2 \\ 1 \\ 1 \end{bmatrix}$
$\text{p\_camera} = \text{inv\_proj\_matrix} \cdot \text{pixels}$
$\text{K} = \frac{camera\_height}{p\_camera[1]}$
$\text{p\_camera\_meters} = \text{p\_camera} \cdot \text{K}$
$\text{tracked\_obstacle.pose} = \text{tf\_Cam\_to\_LiDAR} \cdot \text{p\_camera\_meters}$

Figure 5.4-3 Algorithm used to project the 2D VOT proposals onto the BEV plane

Figure 5.4-3 shows the algorithm used to obtain the BEV pose of the 2D VOT proposals. A more detailed overview of the code can be found in Code of Interest C-5. Basically, based on the inverse of the camera projection matrix, the position of the camera in the vehicle (mainly its height) and the top-left and bottom-right corners of the 2D VOT proposals; the pose of the 2D proposals are recovered in the BEV plane. It is important to take into account that there is a transformation matrix, both in orientation and position, between the LiDAR and camera frame, as shown in Figure 4.4-7. After a proper calibration process, through the $tf\_Cam\_to\_LiDAR$ parameter, camera information is referred to the BEV LiDAR coordinates system.

Furthermore, in order to perform the sensor fusion of BEV VOT and LiDAR proposals, both ROS topics must be reasonably synchronized when being processed by the fusion *callback*. In that sense, ROS *message filters* package are used, which contain some message filters algorithm out of which the time synchronization messages are the most interesting for this purpose.

This synchronization filter is based on a policy that determines how to synchronize the different channels. There exist two main policies: *ExacTime* and *ApproximateTime*. In the case of sensor fusion. In this case, as in Figure 5.4-4, the *ApproximateTime* policy with a interval of time of 200 ms in which both messages may be synchronized is used since it performs better results than other configurations.

```
typedef message_filters::sync_policies::ApproximateTime<sensor_msgs::PointCloud2, centernet::centernet_list> MySyncPolicy2;
message_filters::Synchronizer<MySyncPolicy2> sync2_(MySyncPolicy2(200), velodyne_cloud_sub_, centernet_sub_);
sync2_.registerCallback(boost::bind(&tracking_lidar_camera, _1, _2));
```

Figure 5.4-4 Main callback using the Approximate Time policy

At this point, both BEV proposals are synchronized in the same *callback*. Then, sensor fusion is performed. Figure 5.4-5 shows the algorithm used to carry out the sensor fusion between both proposals. A more detailed overview of the code can be found in Code of Interest C-6. Since the BEV VOT approach is generally much more restrictive, that is, the CenterNet+DeepSORT framework only identifies and track (if relevant) COCO-objects while LiDAR considers an object any relevant group of 3D points, giving rise to a lot of irrelevant 3D small point clouds. For that reason, the first for loop focuses on the BEV VOT proposals and the inner loop in the BEV LiDAR proposals, in order to reduce the computation time. In the contrary case, many of irrelevant LiDAR objects would try to be associated with relevant BEV VOT objects. Then, a maximum difference is initialized to 4 m in such a way that the first preliminary merged object would be represented by the fusion of the BEV VOT proposal and a BEV LiDAR proposal if the Euclidean distance between the BEV LiDAR centroid and the BEV VOT pose (obtained from the bottom position of the 2D bounding box) is lower than this initial 4 m. Moreover, this new max Euclidean distance is updated, and the next BEV LiDAR proposal should be closer to the BEV VOT proposal in order to update the preliminary merged object. When the BEV VOT proposal is compared with all BEV LiDAR proposals, there is a LiDAR candidate with its respective distance to the BEV VOT proposal. If this distance is lower than 3 m, the merged object (also known as Merged VOT object) stores the LiDAR pose and orientation while keeping the VOT identification (object ID and type). Future works will include the BEV VOT pose not exclusively based on the BEV bottom position of the bounding box but as a sum of the projected bottom position and its intrinsic centroid based on the object type and orientation.

Algorithm 5 Sensor fusion between BEV LiDAR and BEV VOT proposals

Input: BEV LiDAR proposals, BEV VOT proposals
Output: Merged proposal

for i = 0 $\longrightarrow$ BEV_VOT.size - 1
Initialize maximum difference to 4 m
Store current BEV obstacle values:
    ycx = BEV_VOT[i].t_x
    ycy = BEV_VOT[i].t_y
if (BEV_LiDAR.size $>$ 0 $\wedge$ (BEV_VOT[i].type = (car $\vee$ pedestrian)))
    Compute BEV_VOT[i] global coordinates
    for j = 0 $\longrightarrow$ BEV_LiDAR.size - 1
    Find closest BEV_LiDAR[j] w.r.t BEV_VOT[i] using Euclidean Distance
    if (smallest Euclidean Distance $<$ 3 m):
    Merge BEV_LiDAR[j] pose and dimensions with BEV_VOT[i] identification

Figure 5.4-5 Algorithm to perform the sensor fusion between BEV VOT and BET LiDAR proposals

Chapter 6. Experimental results

6.1. Introduction

This chapter aims to present the obtained results in order to validate the architecture proposal for Deep Learning based MOT. It is divided in the following sections:

- *Quantitative results:* First, the architecture is validated using the KITTI tracking benchmark. Second, a comparison of BEV pose estimation is performed among the Precision-Tracking, VOT and Merged VOT strategies in CARLA simulator. Finally, some results are shown in the context of our Campus using our real autonomous vehicle.
- *Qualitative results:* Some visual results are shown to illustrate the effectiveness of the MOT approach.

The proposed VOT framework is implemented in a Docker image Ubuntu 18.04 using Python and ROS Melodic Morenia. KITTI and CARLA test cases are done in a desktop computer with i7-8700, 3.2 GHz CPU, 32 GB DDR4 2400 MHz RAM, 500 GB SSD NVME and NVIDIA 2070 RTX. Moreover, in the real prototype test cases, they are performed on an on-board laptop MSI i7-8700, 2.8 GHz CPU, 16 GB DDR4 2400 MHz RAM, 500 GB SSD and NVIDIA 1070 GTX.

In terms of CenterNet, the model was by the CNN was modified from the original *ctdet_coco_dla_2x.pth* model (*dla_34* architecture) to *ctdet_coco_resdcn18.pth* (*resdcn_18* architecture) with *ctdet* task (general object detection task) and a visualization threshold of 0.6. With this configuration, CenterNet+DeepSORT framework runs at 38-45 fps in the above mentioned MSI laptop. For a deeper info about CenterNet models and parameters configuration, the author is referred to the Model ZOO found in CenterNet GitHub.

In addition, sensor fusion has been encapsulated in a Docker image Ubuntu 14.04 using C++ and Ros Indigo Igloo.

6.2. Quantitative results

This section shows the quantitative results for each validation tool used, that is, KITTI tracking benchmark, CARLA simulator and the SmartElderlyCar real prototype.

6.2.1. KITTI tracking benchmark

The KITTI object tracking benchmark [69] consists of 21 training sequences and 29 test sequences, where 31 sequences contains images of size 1242 x 375 while other sequences contain images of similar resolution (i.e. 12*xx* x 37*x*). Moreover, a 64-channels LiDAR

information is provided for each frame. Despite the fact that it labels 8 different classes, only the classes *Car* and *Pedestrian* are evaluated in the benchmark, as only for those classes enough instances for a comprehensive evaluation have been labelled.

KITTI has been performed the labelling process in two steps: First, a set of annotators is hired, in order to label 3D bounding boxes as *tracklets* in point clouds. Since for a pedestrian tracklet a single 3D bounding box tracklet (dimensions have been fixes) often fits badly, it additionally labels the left/right boundaries of each object by making use of Mechanical Turk. It also collects labels of the object occlusion state and computes the object truncation via *backprojecting* a car/pedestrian model into the image plane.

The goal the object tracking task in terms of KITTI benchmark is to estimate object tracklets for the classes car and pedestrian. It evaluates 2D 0-based bounding boxes in each image. For evaluation, it only considers detections/objects larger than 25 pixel (height) in the image and do not count *Vans* as false positives for cars or *Sitting persons* as wrong positives for Pedestrians due to their similarity in appearance.

The main performance metrics used have been MOTP and MOTA [70]:

- *MOTP:* Total position error for matched object-hypothesis pairs over all frames, averaged by the total number of matches made. It shows the ability of the tracker to estimate precise object positions, independent of its skill at recognizing object configurations, keeping consistent trajectories, etc. In other words, it is represented by the summary of overall tracking precisions in terms of bounding box overlap between ground-truth and reported location:

$$MOTP = \frac{\sum_{i,t} d_{i,t}}{\sum_t c_t} \tag{6.1}$$

  Where c_t is the number of correct matches found at frame t and $d_{i,t}$ is the distance between predicted detection and groundtruth detection for each correct match which is taken as the IoU between the two bounding boxes.

- *MOTA:* Summary of overall tracking accuracy in terms of false positives, false negatives and identity switches:

$$MOTA = 1 - \frac{\sum_t (m_t + fp_t + mme_t)}{\sum_t g_t} \tag{6.2}$$

  Where m_t, fp_t and mme_t are the number of misses, of false positives and of mismatches respectively for time t. In that sense, the MOTA can be seen as composed of 3 error ratios, as shown in (6.3): $\bar{m}$ represents the ratio of misses in the sequence, computed over the total number of objects present in all frames, $\overline{fp}$ represents the ratio of false positives and $\overline{mme}$ represents the ratio of mismatches.

$$\bar{m} = \frac{\sum_t m_t}{\sum_t g_t} \quad ; \quad \overline{fp} = \frac{\sum_t fp_t}{\sum_t g_t} \quad ; \quad \overline{mme} = \frac{\sum_t mme_t}{\sum_t g_t} \tag{6.3}$$

Summing up over these different error ratios gives rise to the total error rate E_{tot}, where $MOTA = 1 - E_{tot}$, that is, $1 - E_{tot}$ represents the resulting tracking accuracy.

In that sense, the results obtained, both applying the traditional Precision-Tracking technique and the CenterNet+DeepSORT proposal, in the KITTI tracking dataset compared to other state-of-the-art approaches have been:

Table 6.2-1 Results in KITTI tracking validation/test of different state-of-the-art approaches

	MOTA	**MOTP**
MOTBeyondPixels [78]	84.24 %	85.73 %
IMMDP [79]	83.04 %	82.74 %
3D-CNN/PMBM[80]	80.39 %	81.26 %
extraCK [81]	79.99 %	82.46 %
MASS [82]	85.04 %	85.53 %
Precision-Tracking (ours)	40.93 %	79.13 %
CenterNet+DeepSORT (ours)	82.57 %	81.53 %

All the approaches except ours have been validated on the test datasets for the car class as displayed in the benchmark online (April 2019). In our case we have used the validation dataset (20 % of the training dataset) because our implementation has not been loaded yet into the KITTI public online benchmark. It can be appreciated that the precision-tracking technique, even though it performs in terms of precision (MOTP) in a similar way than other state-of-the-art approaches (it is able to estimate in a very accurate way the object pose), this precision is limited to a good detection, task in which precision-tracking technique fails considerably as mentioned throughout this work, due to the presence of small number of coloured points at further distances. On the other hand, the architecture proposed in this work based on Deep Learning shows some very promising results which are on pair with other current state-of-the-art approaches keeping a good processing time. It presents both good accuracy, since thanks to the scale-aware paradigm CenterNet is able to detect objects at pretty far distance, and precision due to the Deep SORT tracking as a preliminary stage to carry out the sensor fusion. Furthermore, in order to improve the MOTP metric, future works will deal with other state-of-the-art BEV VOT projection approaches, also considering the tracked object type in order to adapt the currently predefined BEV VOT bounding boxes to the object orientation so as to improve the posterior sensor fusion.

6.2.2. CARLA simulator

In spite of the fact that KITTI tracking benchmark is a correct way to validate a tracking architecture proposal, all features are fixed and cannot be modified in order to perceive how the tracking architecture faces different situations, such as sudden rain, pedestrians crossing the road on not-allowed zones or other vehicles performing anomalous behaviours. On the other hand, validation is carried out on the images (2D) using the projection of the ground truth poses but not directly on the 3D space.

Figure 6.2-1 Manual control of dynamic obstacles in CARLA

In that sense, CARLA represents a very powerful simulator in order to validate the tracking system by using its modifiable perception environment, manual control of dynamic obstacles, like cars, pedestrians or bicycles (Figure 6.2-1), and a fast configuration of the vehicle sensors in order to check the best combination to perform the VOT as a preliminary stage to carry out the sensor fusion. As shown in Figure 6.2-2, CARLA test were performed using an RGB camera 1280 x 720, 32-channels LiDAR and semantic segmentation information so as to get a coloured point cloud and carry out the Precision-Tracking approach.

```
"type": "sensor.camera.rgb",
"id": "front",
"x": 0.32, "y": 0.0, "z": 1.65, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
"width": 1280,
"height": 720,
"fov": 100

"type": "sensor.other.gnss",
"id": "gnss1",
"x": -0.28, "y": 0.0, "z": 1.65

"type": "sensor.lidar.ray_cast",
"id": "lidar1",
"x": 0.0, "y": 0.0, "z": 1.95, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
"range": 5000,
"channels": 32,
"points_per_second": 320000,
"upper_fov": 2.0,
"lower_fov": -26,
"rotation_frequency": 20

"type": "sensor.camera.semantic_segmentation",
"id": "semantic",
"x": 0.0, "y": 0.0, "z": 1.65, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
"width": 1280,
"height": 720,
"fov": 100
```

Figure 6.2-2 CARLA sensors configuration

In order to compare CARLA object location, which represents the groundtruth, with the BEV pose estimation of Precision-Tracking, VOT and Merged VOT approaches, some code is implemented to return in a topic named */carla/hero/location_list* the pose of all manually introduced CARLA objects, transforming the GNSS coordinates provided by CARLA to UTM-

global coordinates with respect to the map origin. For example, if there are five pedestrians, this topic must return the pose and orientation of those pedestrian at the same timestamp.

To perform the comparison, it is calculated the Euclidean distance between the CARLA object BEV pose and the BEV pose estimation for each of the above-mentioned techniques if both BEV bounding boxes are associated. This association is carried out in similar way than exposed in Figure 5.4-5: For each technique, a given CARLA object is associated to the closest BEV proposal if the Euclidean distance between its BEV pose is lower than a certain threshold (2 m in this work). Finally, if they are associated, it is stored in a text file in order to perform posterior analysis and comparison, in a similar way that results are compared in the KITTI tracking benchmark. Each row is divided into 11 rows, whose associated fields are shown in Table 6.2-2. It is remarkable that in this case errors will be directly calculated in the BEV plane instead of in images, as happen in the KITTI evaluation.

Table 6.2-2 Structure of each element of the comparison file

Column	Meaning
1	Approach ID (-1 = VOT, -2 = Merged VOT, -3 = Precision-Tracking)
2	CenterNet detector size (how many objects are detected in the scene)
3	CARLA size (how many objects have been introduced in the CARLA world)
4	Tracked object ID (associated ID to an object, used to identify ID mismatching)
5	Tracked object CARLA ID (associated CARLA object to that tracked object)
6	x-groundtruth (CARLA object x position)
7	y-groundtruth (CARLA object y position)
8	Estimated BEV x object position
9	Estimated BEV yobject position
10	Euclidean distance (between CARLA groundtruth and estimated BEV pose)
11	Timestamp

Following tables and graphics are calculated in terms of Single Object Tracking (SOT) to validate the BEV proposals in a more flexible way. Figure 6.2-3 shows the Euclidean distance between the CARLA groundtruth and the estimated position using BEV VOT, Merged VOT and Precision-Tracking estimated pose versus the X-local distance (road axis).

An interesting metric to compare the performance in the BEV pose estimation is to calculate the global *Root Mean* Square (RMS) error in the whole trajectory and the RMS for each interval:

$$RMS_{error} = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} e_i} \tag{6.4}$$

Where n is the number of associations that were produced with the CARLA object for each technique and ed_i the Euclidean distance between the CARLA object pose and the BEV proposal for the i-th association. Table 6.2-3, Table 6.2-4 and Table 6.2-5 show the RMS error and number of samples (correctly associated BEV proposals) for each technique. It can be appreciated that the maximum registered distance to track an object is 37.34 m. Moreover, until distances of 12-13 m all approaches perform a similar behaviour, with a Euclidean distance under 0.2436 m. However, for further distances, VOT starts increasing its error. Moreover, for distances further than 24 m, due to the point cloud scarcity, Precision-Tracking, based on semantic segmentation, is not able to track the object since there are very few coloured points, what is represented by 0 samples and a Not Applicable (N/A) RMS. Finally, the blue line, representing the Merged VOT proposal, works in a pretty accurate way, maintaining an RMS (Root Mean Square) error in the whole trajectory below 0.2278 m even for distances further than 32 m. Additionally, global RMS

Table 6.2-3 RMS error and Number of samples in function of the distance (Precision-Tracking)

Interval	Number of samples	RMS
0 m – 4 m	1	0.2164
4 m – 8 m	0	N/A
8 m – 12 m	2	0.2436
12 m – 16 m	1	0.1212
16 m –20 m	10	0.2268
20 m – 24 m	1	0.2668
24 m – 28 m	0	N/A
28 m – 32 m	0	N/A
32 m – 36 m	0	N/A
36 m – 40 m	0	N/A
Whole trajectory	15	0.2259

Table 6.2-4 RMS error and Number of samples in function of the distance (VOT)

Interval	Number of samples	RMS
0 m – 4 m	9	0.0472
4 m – 8 m	21	0.2177
8 m – 12 m	21	0.1784
12 m – 16 m	22	1.1189
16 m –20 m	23	1.5045
20 m – 24 m	20	0.9613
24 m – 28 m	18	2.2384
28 m – 32 m	7	2.6679
32 m – 36 m	5	1.8596
36 m – 40 m	7	1.9294
Whole trajectory	149	1.2811

Table 6.2-5 RMS error and Number of samples in function of the distance (Merged VOT)

Interval	Number of samples	RMS
0 m – 4 m	9	0.2156
4 m – 8 m	21	0.1911
8 m – 12 m	21	0.1856
12 m – 16 m	22	0.1884
16 m –20 m	23	0.1546
20 m – 24 m	20	0.2278
24 m – 28 m	15	0.2201
28 m – 32 m	3	0.1963
32 m – 36 m	3	0.1408
36 m – 40 m	5	0.2295
Whole trajectory	146	0.1963

As expected, precision-tracking technique performs a good BEV estimation if a precision tracker is associated to the CARLA object, although it is able to perform that identification very few times (15) in comparison with VOT (149) and Merged VOT (149). The number of samples of VOT is always greater than Merged VOT since at further distances (over 31 m) BEV VOT pose and BEV LiDAR pose distance is greater than the required *threshold*, so sensor fusion is not performed.

Figure 6.2-4 and 6.2-5 show a visual comparison of the BEV pose estimations for the different techniques (CARLA groundtruth trajectory versus BEV estimated trajectory) in straight and curved trajectory respectively. Figure 6.2-5 and Figure 6.2-7 focus on the Merged VOT proposal, illustrating the effectiveness of this method.

Figure 6.2-8, Figure 6.2-9 and Figure 6.2-10 show 3D scatter plots to represent the Euclidean distance in terms of CARLA (X,Y) groundtruth. It can be appreciated that the Precision-Tracking approach presents very few points as a result of not semantic segmenting properly the object at further distances due to the points scarcity in that particular part of the point cloud. VOT points show how Y-lateral displacements increases the error in a parabolic way (it can be appreciated better for low values of X local distances). Furthermore, Merged VOT points show a proper behaviour, since despite the presence of very few outliers, most of points present a Z-value (that is, the Euclidean distance) under 0.2764 m even including Y-lateral displacements. All graphics show the position of the ego vehicle in order to appreciate properly the pose and orientation of the trajectories.

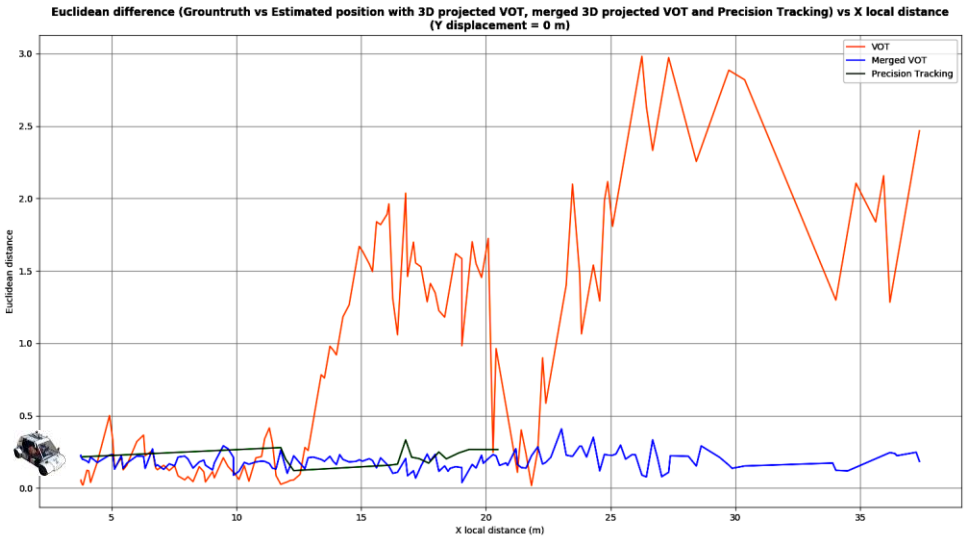


Figure 6.2-3 Euclidean distance vs X local distance with Y displacement = 0 m (including all approaches)

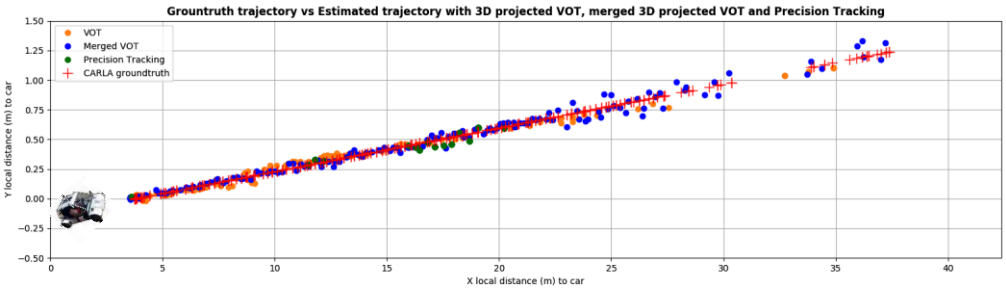


Figure 6.2-4 BEV of Groundtruth trajectory vs Estimated trajectory in straight line (including all approaches)

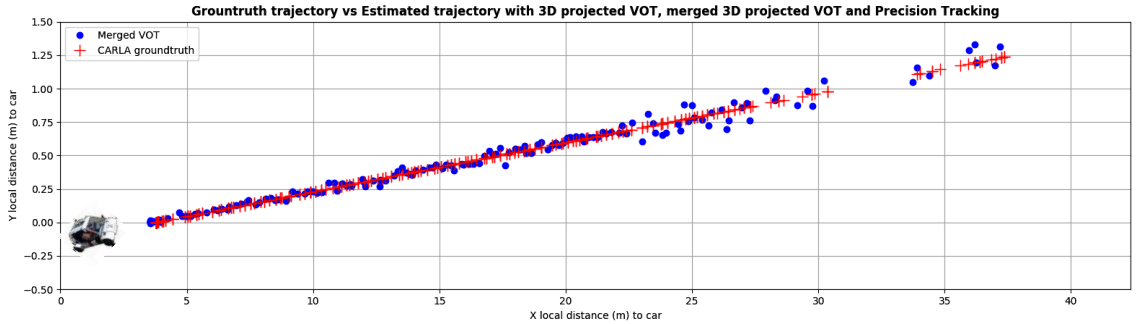


Figure 6.2-5 BEV of Groundtruth trajectory vs Estimated trajectory in straight-line tracking (only Merged VOT approach)

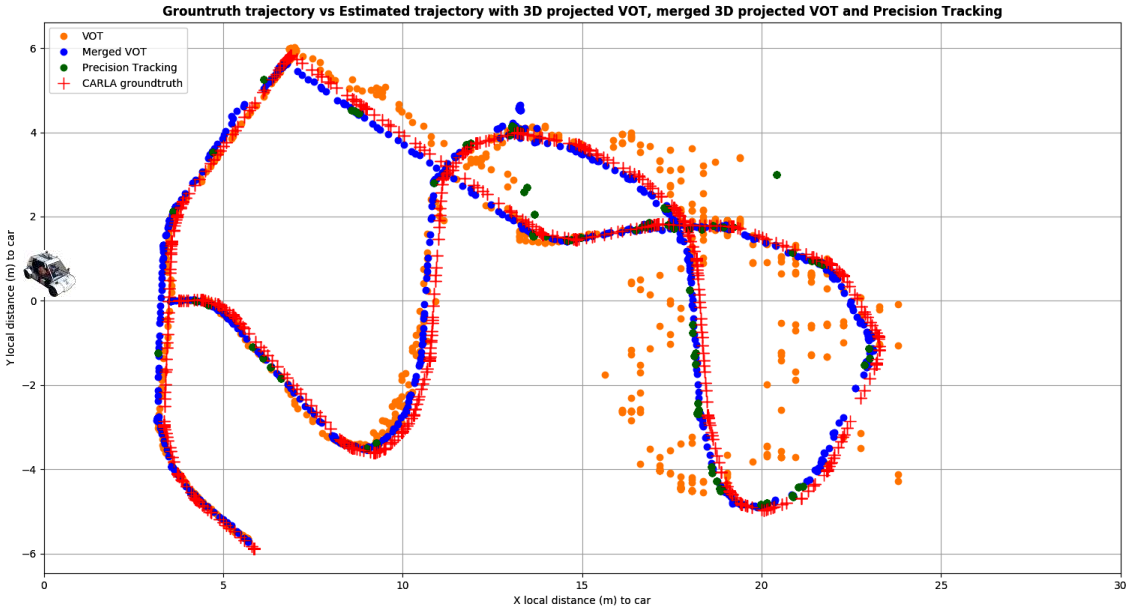


Figure 6.2-6 BEV of Groundtruth trajectory vs Estimated trajectory in curved-line tracking (all approaches)

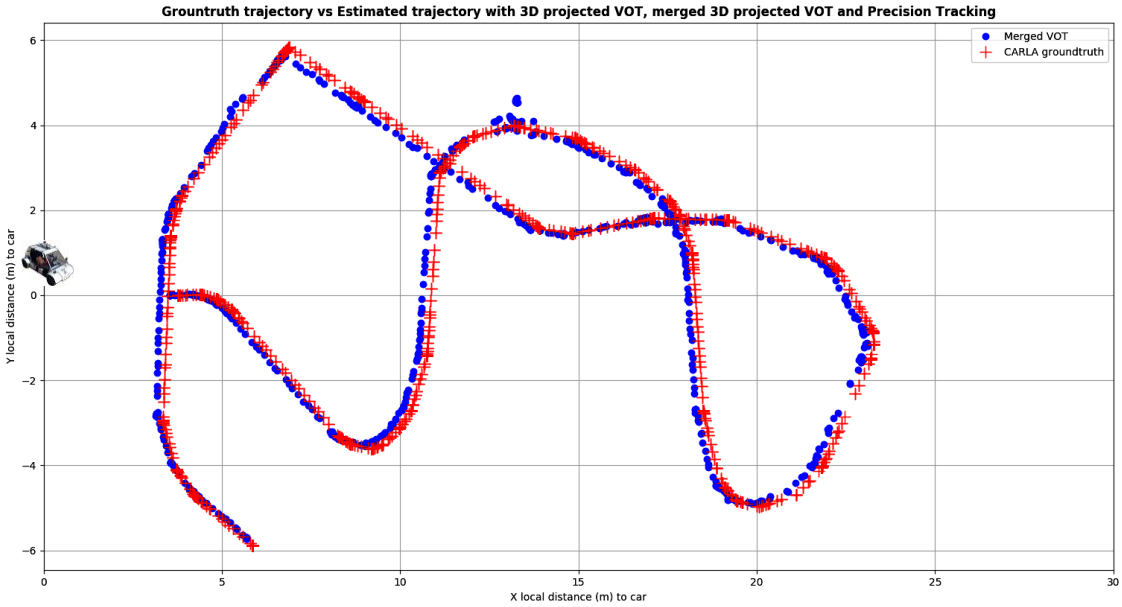


Figure 6.2-7 BEV of Groundtruth trajectory vs Estimated trajectory in curved-line tracking (only Merged VOT approach)

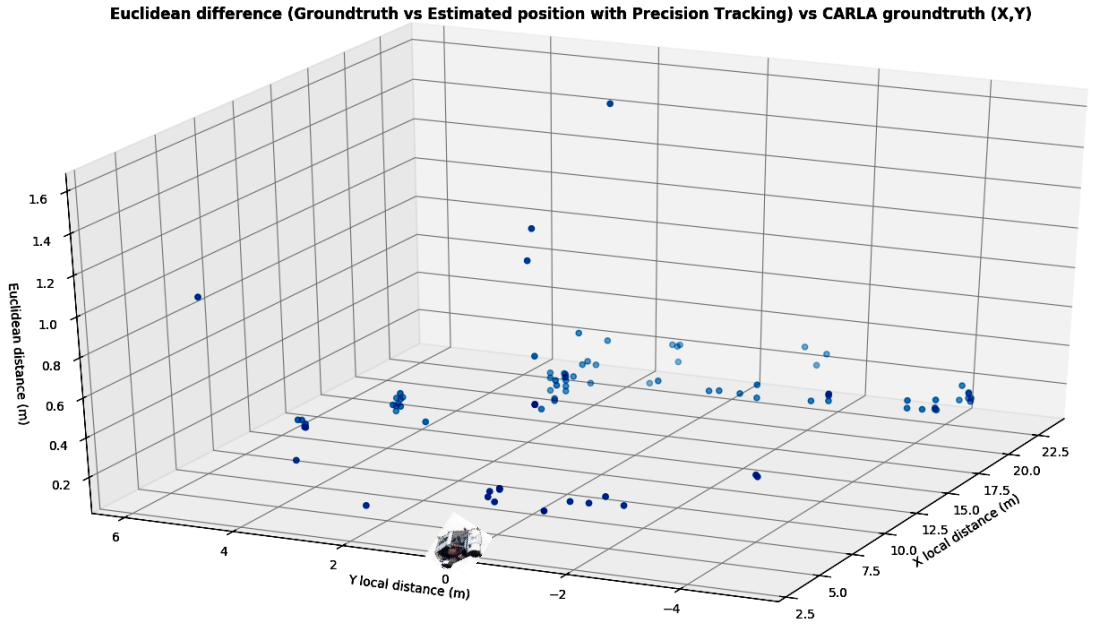


Figure 6.2-8 3D scatterplot representing the Euclidean distance vs CARLA groundtruth (X,Y) (Precision-Tracking approach)

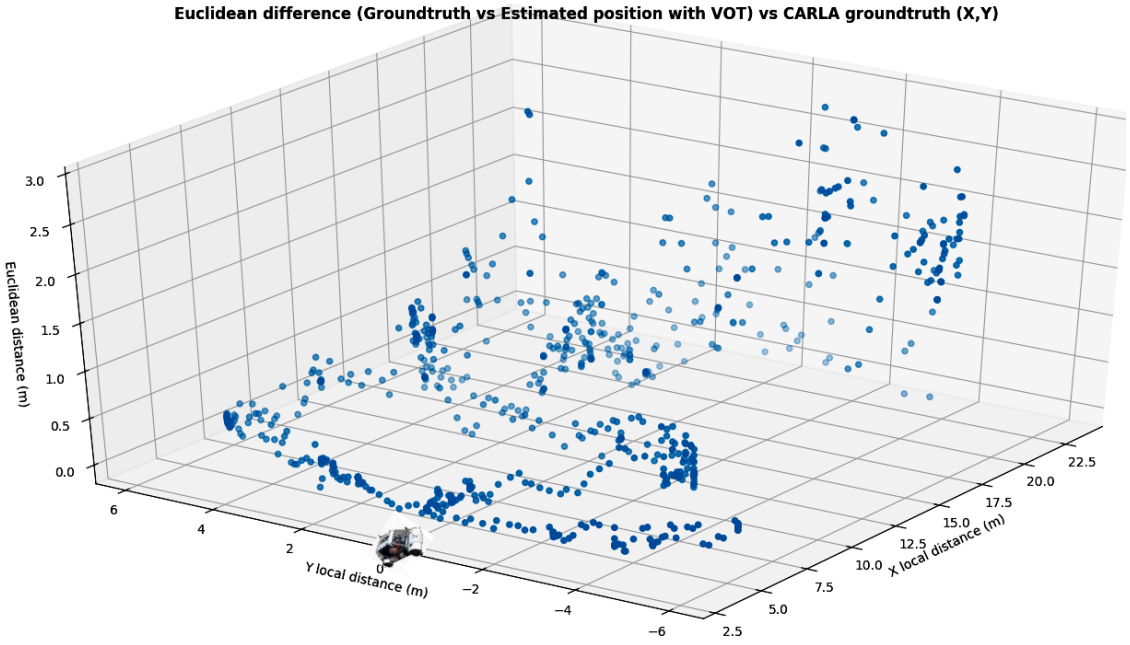


Figure 6.2-9 3D scatterplot representing the Euclidean difference vs CARLA groundtruth (X,Y) (VOT approach)

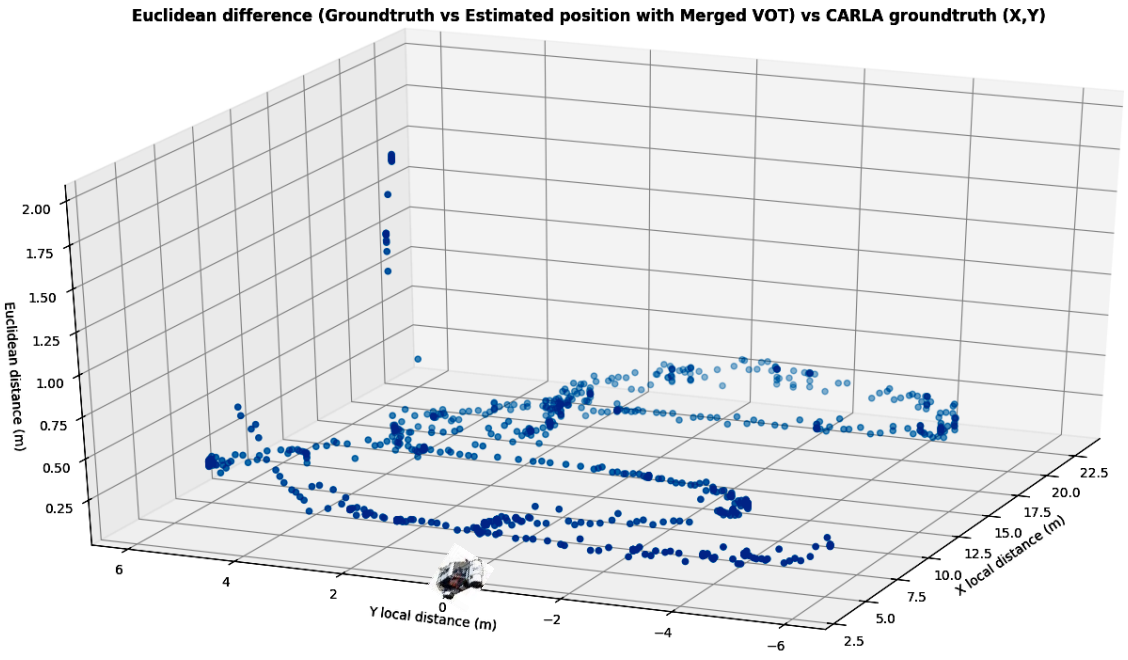
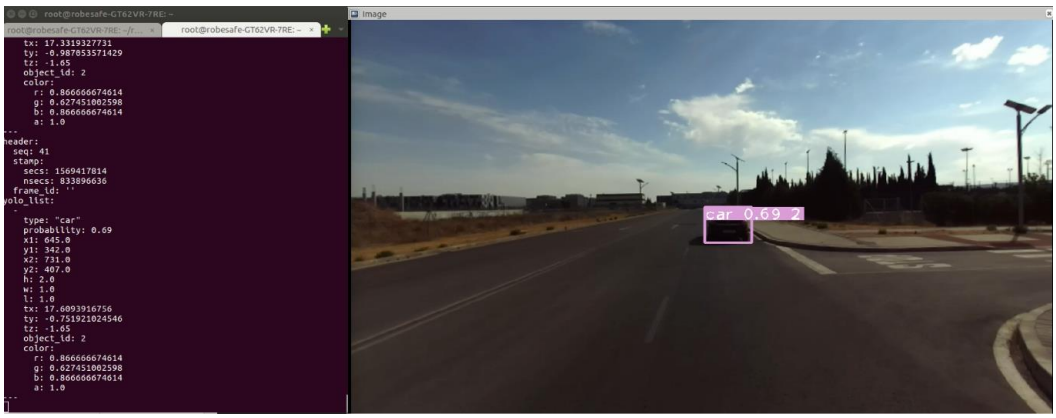


Figure 6.2-10 3D scatterplot representing the Euclidean difference vs CARLA groundtruth (X,Y) (Merged VOT approach)

6.2.3. SmartElderlyCar

In spite of the fact that in the Campus there is not a groundtruth, several tests were carried out in order to appreciate a preliminary performance of the architecture proposal, obtaining successful results. Following figures represent four subsequent frames for the same scene where SOT is performed. It can be appreciated that the ID is kept throughout all frames (ID = 2) and the projection matrix, though is not the best solution, offers coherent numbers (the lateral displacement spans from -0.75 m in the #1st frame, since the car is on the left with respect to the ahead vehicle, to -0.028 m in the #4th since both vehicles are almost aligned).



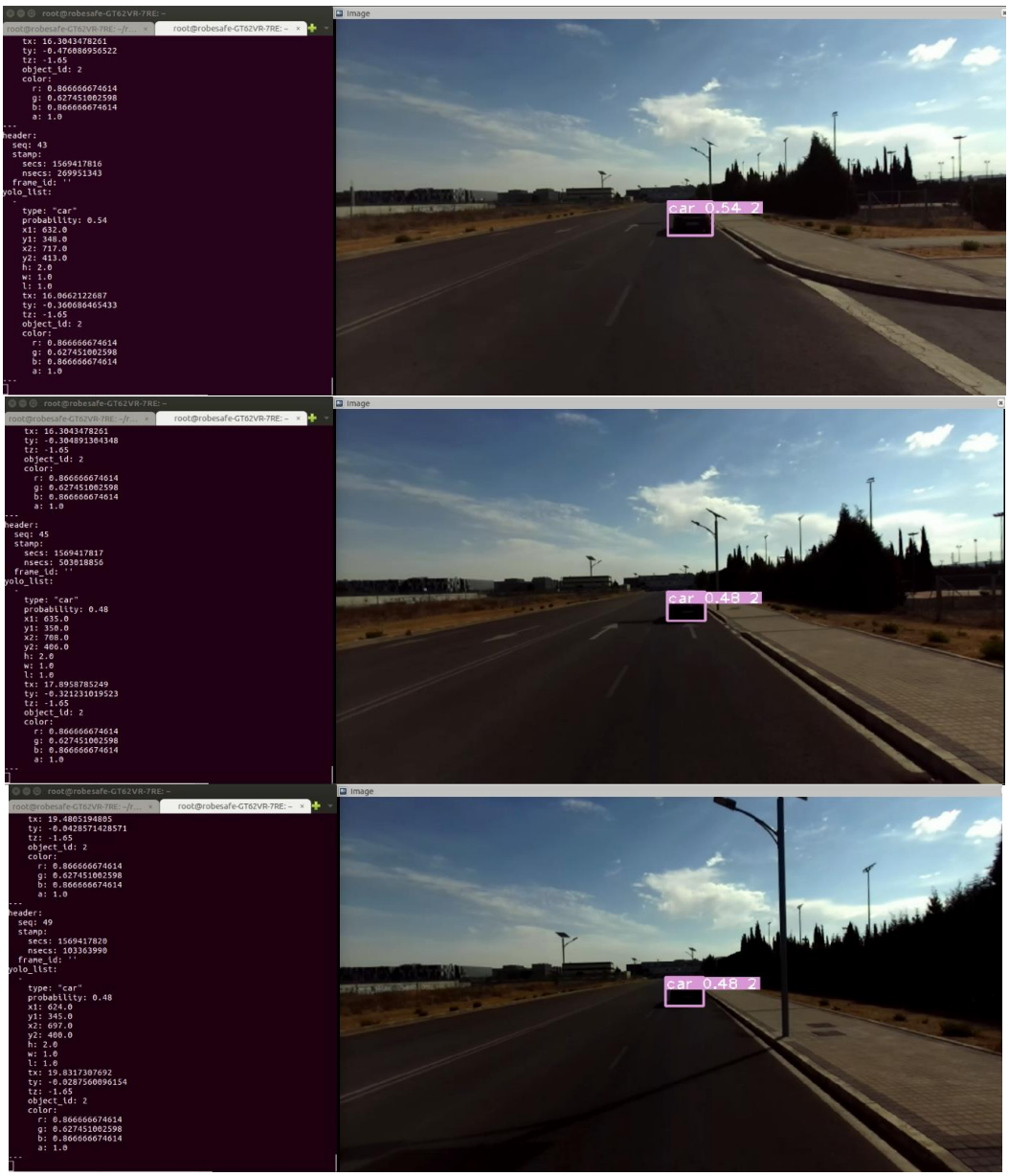


Figure 6.2-11 Quantitative results in SmartElderlyCar navigation

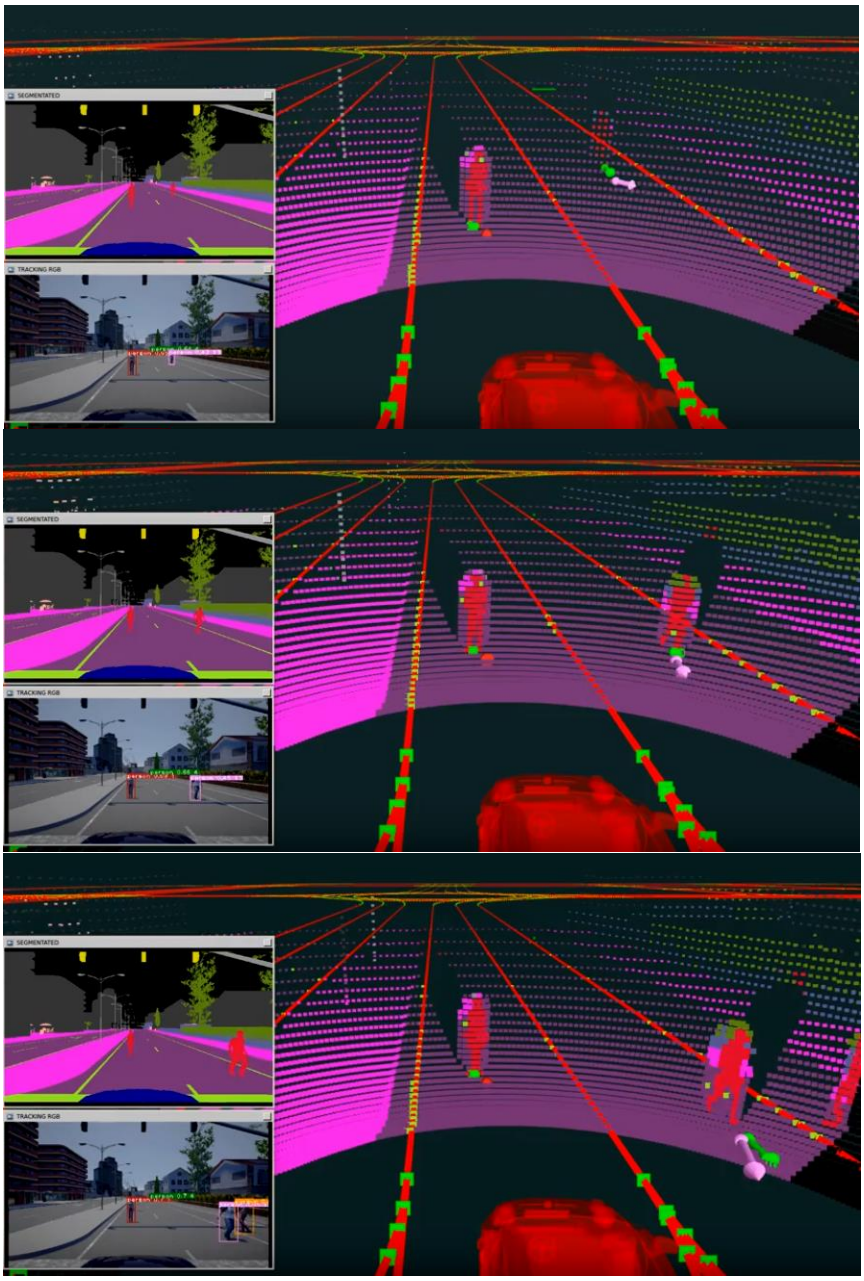
6.3. Qualitative results

This section shows the qualitative results of the architecture proposal, illustrating how MOT is performed both in CARLA simulator and in the Campus with our real autonomous vehicle.

6.3.1. CARLA simulator

All frames show different pedestrians which are included in the CARLA simulator. The top-left small window shows the semantic segmentation provided by CARLA. On the other hand, the bottom-left small window shows the output image as a result of the object detection and tracking in the scene performed by the CenterNet+DeepSORT framework. The general background shows the coloured point cloud (in *R-VIZ* simulator) obtained by projecting the

semantic segmentation information onto the velodyne point cloud (as shown in Figure 4.6-3). Each pedestrian (red cluster) has a green arrow, illustrating the predicted position using the Merged VOT approach and an arrow with the colour obtained from the 2D tracking (VOT). It can be appreciated that Merged VOT approach estimates better the centroid of the objects than the other approaches.



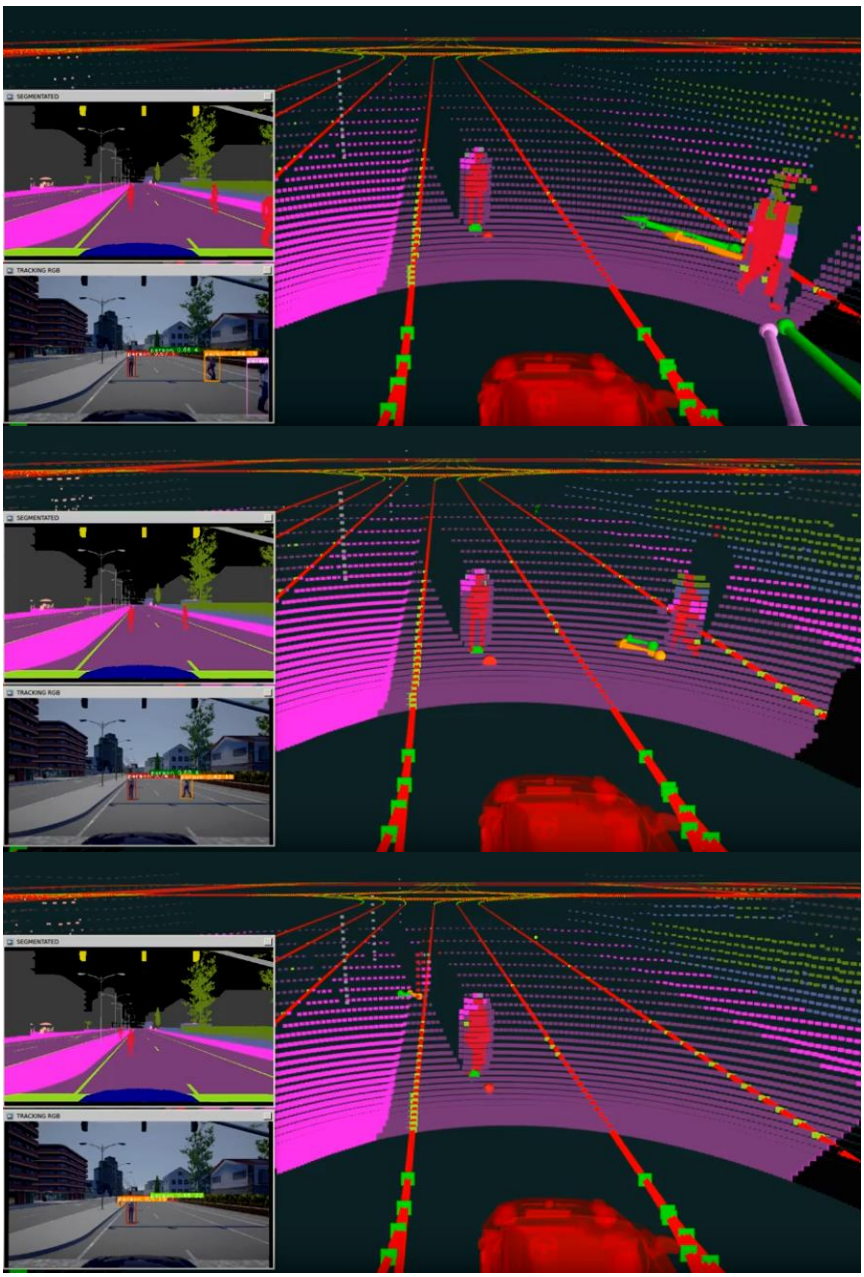


Figure 6.2-12 Qualitative results in CARLA simulator

6.3.2. SmartElderlyCar

This section shows the MOT paradigm in a real-world situation. Tests were performed in the Escuela Politécnica Superior (UAH) surroundings. We show results directly on the images to understand the robustness of our architecture. Even though most of objects (mainly cars) are partially occluded, Deep SORT proposal deals with that issue in an accurate way.


car car car 0.49
car 0.48 41
car car 0.44 3
car car 0.41 443
car 0.46 41
car 0.4 44

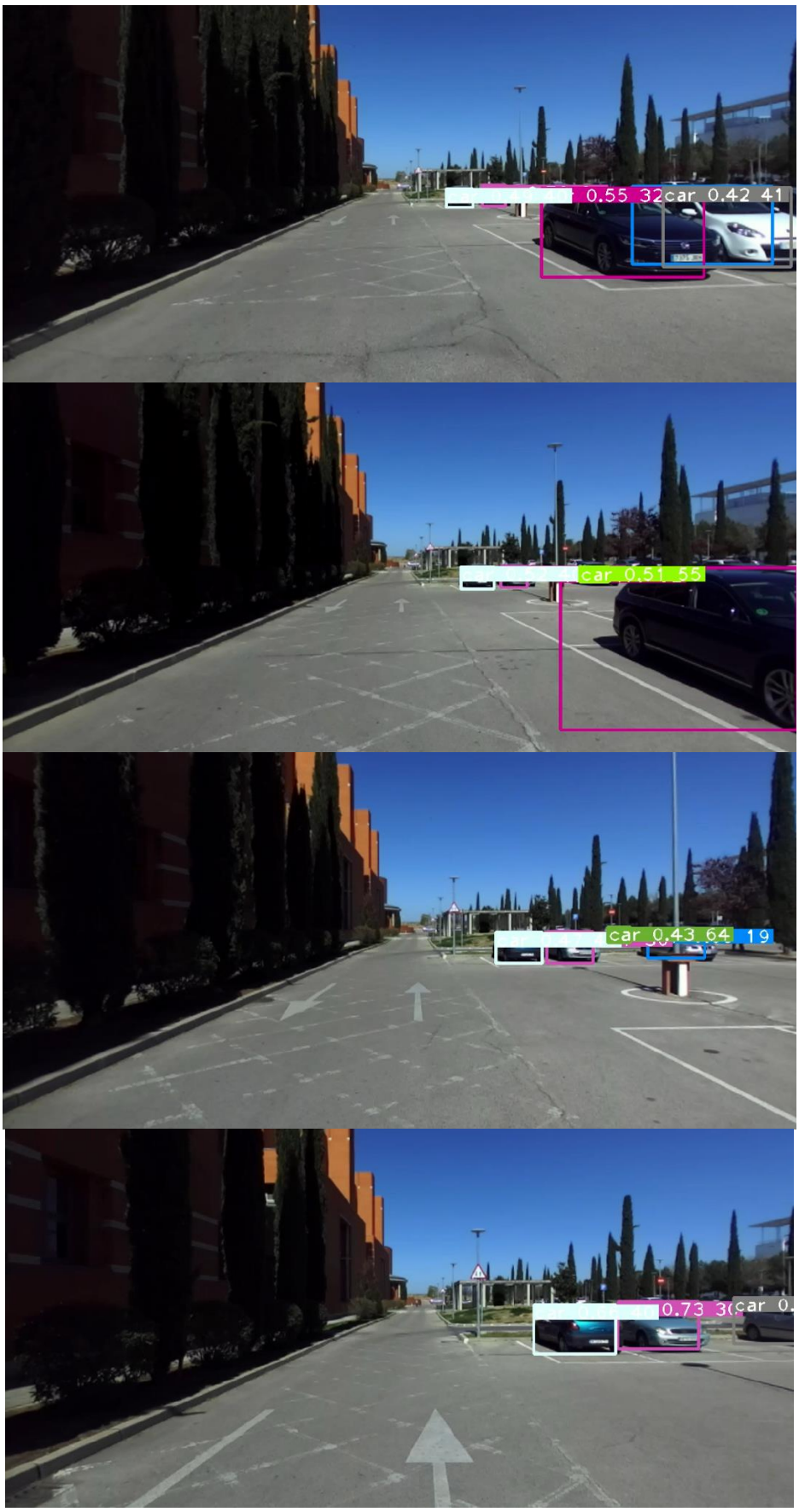


Figure 6.2-13 Qualitative results in a real-world situation with the SmartElderlyCar

Chapter 7. Conclusions and future works

7.1. Conclusions

This master thesis shows the development of a Deep Learning based Multi-Object Tracking approach based on CenterNet as object detector and Deep SORT as object tracking algorithm and its implementation on KITTI benchmark and on the SmartElderlyCar project both in CARLA simulator and real-world. In addition, sensor fusion was performed between the deep learning approach and a 64/32 channels LiDAR 3D, obtaining better results than other state-of-the-art approaches for object tracking in autonomous driving applications.

It must be considered the architecture proposal depends crucially on the results obtained during image detection, since although the sensor fusion performs a very accurate estimated pose of the objects, the tracking is performed in 2D. So, if the object detector does not work in a proper way, there could be identification mismatching or even loss of information, leading to a dangerous situation for the vehicle. In that sense, implemented object detector (CenterNet) works in a very accurate way. Even it has been trained using real world images (COCO dataset), it performs a good detection with synthetic data (CARLA simulator). On the other hand, Deep SORT approach works in a very appropriate way, handling the occlusion, lighting, and point-of-view problems as shown in the qualitative results illustrating the effectiveness in the use of deep learning approaches in terms of object tracking (Deep appearance descriptor). The combination of this object detector and the deep learning-based tracking algorithm gives rise to a correct performance in terms of 2D tracking.

Validation results have been obtained in KITTI dataset showing better results than using the traditional Precision-Tracking strategy and being on pair with other state-of-the-art proposals. Validation has been performed as well in CARLA simulator since it is able to get the groundtruth of the objects in an easy way in order to stablish a proper comparison. As expected, the tracking results obtained by the sensor fusion proposal significantly improve the tracking performed by Precision-Tracking algorithm and Visual Object Tracking.

The validation has been performed in CARLA since it is able (after few modifications) of obtaining the groundtruth of the objects in order to stablish a proper comparison. As expected, the tracking results obtained by the sensor fusion proposal significantly improve the tracking performed by precision-tracking algorithm and only Visual Object Tracking. Despite the fact the estimation of precision-tracking is relatively accurate, it is not able to track at further distances than 23 m since it most depends on the projection of the semantic segmentation into the point cloud. Since at that distance, even with a 32-channels LiDAR, the number of points considerably is decreased, the number of coloured points is not enough in order to identify that coloured cluster as a determined object. On the other hand,

CenterNet+DeepSORT framework is able to detect and track the object in an accurate way until distances of 34 m, when the object size in the image is not big enough to be detected by CenterNet object detector, even with its scale-aware paradigm. As expected, the further an object is, the greater the Euclidean distance between the groundtruth of CARLA position and the estimated position is. However, merging this BEV proposal of the CenterNet+DeepSORT framework with the closest BEV LiDAR cluster gives rise to Euclidean distances lower than 0.37 m throughout the whole trajectory.

Therefore, it can be concluded that the proposed objectives at the beginning of this master thesis, mainly the study of deep learning-based tracking approaches, implementation of a Deep Learning based Multi-Object Tracking architecture and validation, have been met.

7.2. Future Works

In order to improve the present work, the following improvements may be performed:

- Improvement of 3D projection in the tracking layer in order to decrease the error in terms of X-axis and Z-axis.

- Improvement in the architecture validation for the case of multiple objects in CARLA.

- Update of the CARLA ROSbridge in order to improve the communications between the CARLA world and the SmartElderlyCar project.

- Update of the CARLA version (0.9.5 to the most recent) to improve the simulation experience and efficiency.

- Improvement of sensor fusion not only taking into account the Euclidean distance but also previous behaviours of relevant objects (such as a pedestrian has the intention to cross the road or a car is going to perform an anomalous behaviour) based on the orientation of the object and biometric features.

- Validation in KITTI using the test dataset and other classes beyond the cars and in other tracking datasets.

- Integration of the different SmartElderlyCar software layers in Docker containers in order to improve the development and testability of the project.

- Update the ROS version of Docker containers to ROS2 so as to investigate new paradigms of real-time operation and Docker integration.

- Implementation of a multi-camera system (both in CARLA and in the real prototype) to carry out a tracking in 360 º.

- Development of a groundtruth for real-world objects so that the tracking of multiple objects in real applications can be validated.

Appendix A: Kalman Filter

The *Kalman filter* [30] (Rudolf E. Kalman, 1960) is an algorithm that uses a set of measurements observed throughout the time, containing statistical noise and other inaccuracies, producing an estimation of unknown variables that are likely to be more accurate than those variables based on a single measurement alone, estimating a joint probability distribution over the variables for each timeframe.

Kalman filter can be used in any field where there is uncertain information about some dynamic system, performing a quite well guess about what system is going to do next. Kalman filter is able to "messy" realities interfering with the clean motion it guessed about, figuring out what actually is happening.

A. 1. Introduction to the Kalman Filter

Kalman filter is ideal for systems that are continuously changing since it is light on memory (not requiring to keep any history other than the previous system state), so they are very fast, making them well-suited for embedded systems and real-time applications. It can be applied to multiple situations and data, such as the amount of fluid in a tank, the position of a user's finger on a touchpad, the temperature of a car engine or any number of things needed to keep track of.

However, since the purpose of this master thesis is related with tracking, or keep track of an object in a scene, it is reasonable to explain the Kalman filter from a certain (simple) dynamic robot perspective. This robot presents a state $\overrightarrow{x_k}$ which is just a vector that shows its position and velocity:

$$\overrightarrow{x_k} = (\overrightarrow{p}, \overrightarrow{v}) \tag{A.1}$$

In spite of the fact that an external user does not know the actual position and velocity, there are a whole range of possible combinations of position and velocity which might be true, but some of them are more likely than others. The Kalman filter assumes that both variables (in this case velocity and position) are random and *Gaussian distributed*. Each variable has a mean value μ, which is the center of the random distribution (moreover, the most likely state), and a variance σ^2 which is the uncertainty. Figure A.1-1 illustrates how the position and velocity are uncorrelated, which means that the state of one variable does not affect to the state of the others.

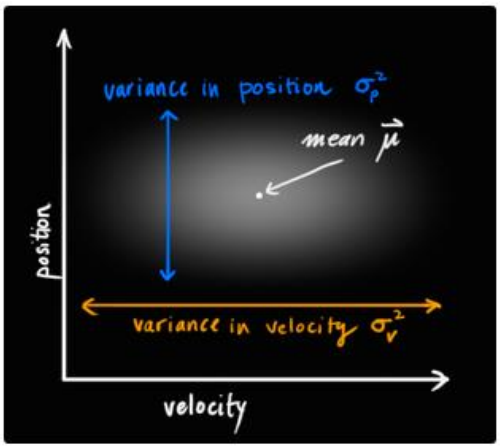
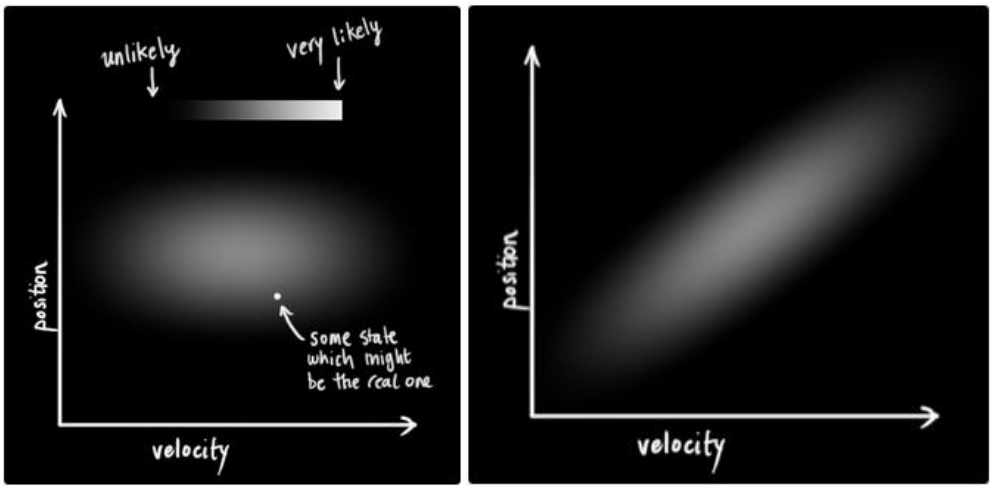


Figure A.1-1 Mean μ and variance σ^2 of the velocity and position

On the other hand, Figure A.1-2 shows the difference between two uncorrelated (a) and correlated (b) variables. In this case, the likelihood of observing a particular position depends on what velocity the robot has.



(a) (b)

Figure A.1-2 (a) Velocity and position are uncorrelated (b) Both variables are correlated

This kind of situation (correlated position and velocity) might arise if, for example, the system is estimating a new position based on an old one. If the velocity was high, the robot probably moved farther, and the position will be more distant. This kind of relationship is important to keep track of, since it reports more information: The behaviour of a variable can be determinant about what the others could be. That is one of the main goals of the Kalman, to squeeze as much information from an uncertain measurement as it is possible.

This correlation is capture by a covariance matrix. In statistics and probability theory, covariance is a measure of the joint probability of two random variables, so the covariance matrix is a matrix whose element in the i, j position is the covariance between the i-th and

j-th elements of a random vector. Given n random variables (each with finite variance and expected value) of a column vector:

$$\boldsymbol{X} = (X_1, X_2, \dots, X_n)^T \tag{A.2}$$

Then the covariance matrix $\boldsymbol{K_{XX}}$ (also Σ_{XX}) can be defined as the matrix whose (i, j) entry is the covariance. It must be noted that the covariance matrix is symmetric, so it does not matter to swap i and j.

$$\boldsymbol{K_{X_iX_j}} = cov\left[X_i, X_j\right] = E\left[(X_i - E[X_i])\left(X_j - E\left[X_j\right]\right)\right] \tag{A.3}$$

Where the operator **E** denotes the expected value (the expected value of a random variable is the long-run average value of repetitions of the same experiment it represents) of its argument. In matrix form, for n random variables, it can be expressed as shown in Figure A.1-3:

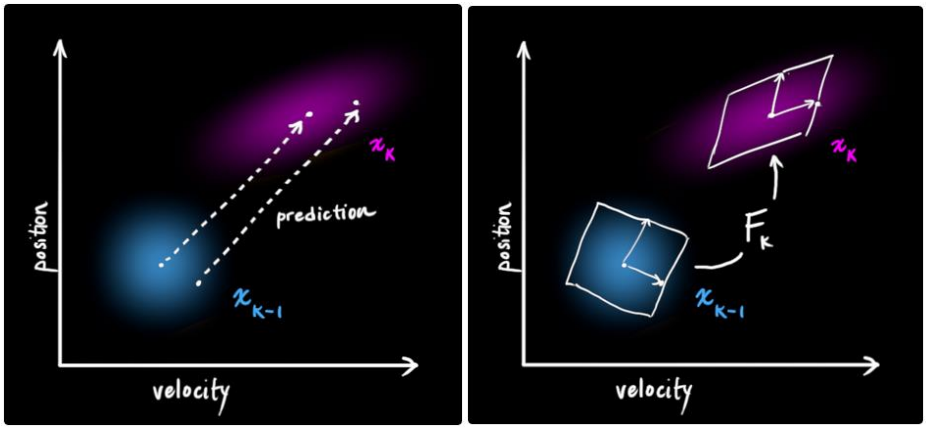
$$\mathrm{K_{XX}} = \begin{bmatrix} \mathrm{E}[(X_1 - \mathrm{E}[X_1])(X_1 - \mathrm{E}[X_1])] & \mathrm{E}[(X_1 - \mathrm{E}[X_1])(X_2 - \mathrm{E}[X_2])] & \cdots & \mathrm{E}[(X_1 - \mathrm{E}[X_1])(X_n - \mathrm{E}[X_n])] \\ \mathrm{E}[(X_2 - \mathrm{E}[X_2])(X_1 - \mathrm{E}[X_1])] & \mathrm{E}[(X_2 - \mathrm{E}[X_2])(X_2 - \mathrm{E}[X_2])] & \cdots & \mathrm{E}[(X_2 - \mathrm{E}[X_2])(X_n - \mathrm{E}[X_n])] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{E}[(X_n - \mathrm{E}[X_n])(X_1 - \mathrm{E}[X_1])] & \mathrm{E}[(X_n - \mathrm{E}[X_n])(X_2 - \mathrm{E}[X_2])] & \cdots & \mathrm{E}[(X_n - \mathrm{E}[X_n])(X_n - \mathrm{E}[X_n])] \end{bmatrix}$$

Figure A.1-3 Covariance matrix example

As commented above, Kalman filter assumes that system variables are *Gaussian distributed*, so two pieces of information are required at time k: The best estimate $\widehat{\boldsymbol{x}}_{\boldsymbol{k}}$ (i.e., the mean, elsewhere named μ), and its covariance matrix $\boldsymbol{P_k}$ (note that here it is written as P and not as K in order to not to confuse with the instante k:

$$\widehat{\boldsymbol{x}}_{\boldsymbol{k}} = \begin{bmatrix} position \\ velocity \end{bmatrix} \quad ; \quad \boldsymbol{P_k} = \begin{bmatrix} \sum_{pp} & \sum_{pv} \\ \sum_{vp} & \sum_{vv} \end{bmatrix} \tag{A.4}$$

Where subscripts p and v mean position and velocity respectively. Then, the main goal is to consider the current state (at time $k-1$) so as to predict the next state at time k. However, Kalman filter assumes that the system does not know which the real state of its variables is, but the prediction function does not care. Moreover, the prediction function works on all possible states giving rise to a new distribution, as shown in Figure A.1-4 (a).



(a) (b)

Figure A.1-4 (a) New distribution after prediction (b) Transformation matrix between original estimate and new prediction position

If the prediction process is represented as a matrix $\boldsymbol{F_k}$, it takes every point in the original estimate and moves it to a new prediction position (which is the point where the system would move in the plane, in case of two variables, if that original estimate was the right one). In the case of position and velocity, using basic kinematic formulas, it can be expressed as:

$$p_k = p_{k-1} + \Delta t \cdot v_{k-1} \quad ; \quad v_k = v_{k-1} \tag{A.5}$$

Rewriting (A.5) in a matrix form:

$$\hat{x}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{x}_{k-1} = \boldsymbol{F_k} \hat{x}_{k-1} \tag{A.6}$$

Nevertheless, despite the fact that (A.6) expresses how the next state can be calculated based on a prediction matrix $\boldsymbol{F_k}$ and the current frame, the system still does not know how to update the covariance matrix. In order to calculate the new covariance matrix, every point in the distribution must be multiplied by a matrix $\boldsymbol{A}$, giving rise the identity (matrix properties):

$$Cov(x) = \Sigma \quad ; \quad Cov(Ax) = A\Sigma A^T \tag{A.7}$$

And combining (I.7) with equation (I.6), it results in:

$$\hat{x}_k = \boldsymbol{F_k} \hat{x}_{k-1} \quad ; \quad P_k = F_k P_{k-1} F_k^T \tag{A.8}$$

A. 2. External influence

Even though (A.8) captures the covariance matrix and state variables in instant k, there might be some changes that are not related to the state itself, that is, the world could be affecting the system. For example, in the case of an autonomous vehicle performing an

Adaptive Cruise Control (ACC), the vehicle could accelerate. If the system knows what is going on in the real-world, this additional behaviour can be stacked in a vector $\vec{u}_k$, in order to incorporate to the prediction as a correction. In the case of the previous robot, the vector $\vec{u}_k$ is in the form of an expected acceleration a due to the control commands or throttle settings:

$$p_k = p_{k-1} + \Delta t \cdot v_{k-1} + 1/2 \cdot \mathrm{a} \cdot \Delta t^2 \quad ; \quad v_k = v_{k-1} + \mathrm{a} \cdot \Delta t \tag{A.9}$$

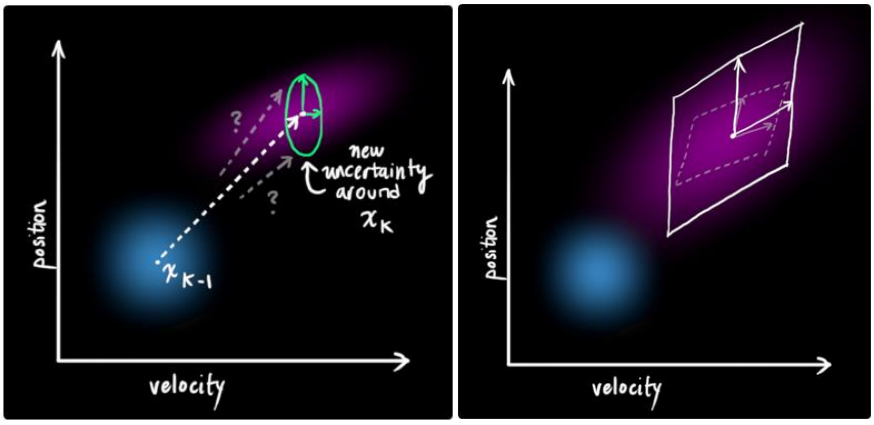
Or in the matrix form:

$$\hat{x}_k = \boldsymbol{F_k}\hat{x}_{k-1} + \begin{bmatrix} \Delta t^2/2 \\ \Delta t \end{bmatrix} a = \boldsymbol{F_k}\hat{x}_{k-1} + B_k\vec{u}_k \tag{A.10}$$

Where B_k is called the control matrix and $\vec{u}_k$ the control vector.

A. 3. External influence

While previous point mentioned what happens if and additional component has an influence on the system, this point deals with what happens if the prediction model is not 100 % accurate, that is, external variables which the system does not know about. In the case of self-driving, the wheels could slip or bumps on the ground. If it happens, and the system is not prepared for those extra forces, the prediction could be off. The uncertainty associated with the "world" (i.e., variables the system is not keeping track of) can be modelled by adding some new uncertainty after every prediction step (Figure A.3-1 (a)).



(a) (b)

Figure A.3-1 (a) New uncertainty after the prediction step (b) New Gaussian distribution with a different covariance

Every state in the original estimate could have moved to a range of new states. Since variables are expected to be under *Gaussian distributions*, each point in $\hat{x}_{k-1}$ is moved to somewhere inside a *Gaussian distribution* with the covariance Q_k, that is, the system would be treating the untracked fluences as noise with covariance Q_k, what produces a new

Gaussian distribution with a different covariance but the same mean, as shown in (Figure A.3-1 (a)). Then, the new covariance is corrected by simply adding Q_k, giving rise to a complete expression for the prediction step:

$$\hat{x}_k = F_k\hat{x}_{k-1} + B_k\vec{u}_k \quad ; \quad P_k = F_kP_{k-1}F_k^T + Q_k \tag{A.11}$$

In other words, the new best estimate ($\hat{x}_k$) is a prediction made from the previous best system estimate ($\hat{x}_{k-1}$) plus a correction for known external influences ($\vec{u}_k$), and the new uncertainty (P_k) is predicted from the old uncertainty (P_{k-1}) with some additional uncertainty from the environment.

A. 4. Refining the estimate with measurements

As commented throughout this master thesis, autonomous vehicles are full of sensors. One can read velocity, other position or even calculating the odometry of the vehicle by using visual information, but in summary all of them report information about the state of the vehicle. Since the units and scale of the reading might not be the same as the units and scale of the state the system is keeping track of, these sensors are modelled with a matrix H_k (Figure A.4-1).

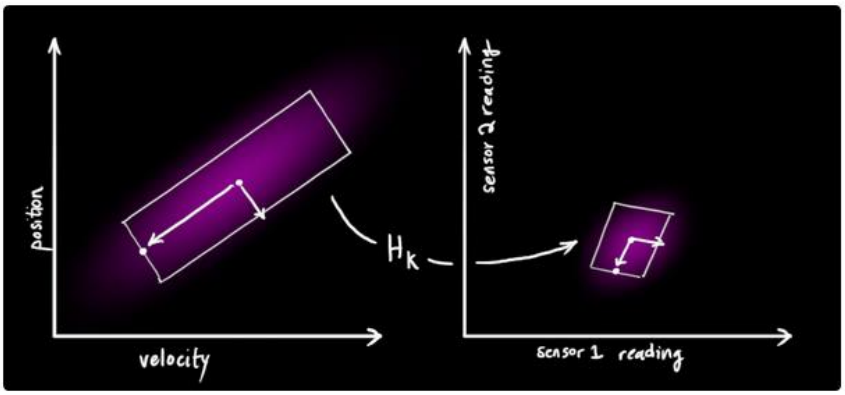


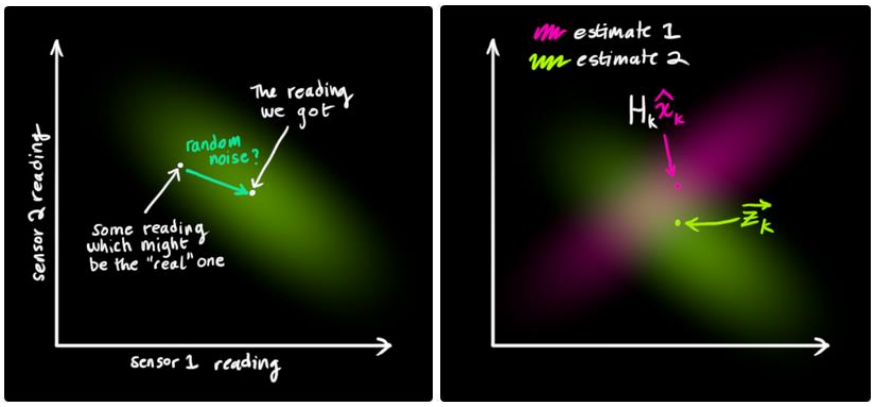
Figure A.4-1 Sensors modelling using matrix transformation

So the expected distribution of sensor readings is as following:

$$\vec{\mu}_{expected} = H_k\hat{x}_k \quad ; \quad \Sigma_{expected} = H_kP_kH_k^T \tag{A.12}$$

One thing that Kalman filters are great for is dealing with sensor noise, since even the best sensor is at least somewhat unreliable and every state in the original estimate might result in a range of sensor readings. From each sensor reading it can be appreciated was in a particular state. However, since there is uncertainty, some states are more likely than others (Figure A.4-2 (a)). The covariance of this uncertainty (covariance of the sensor noise) is called R_k, whose distribution has a mean equal to the reading it was observed by the sensors ($\vec{z}_k$). Then, there are two distributions: One surrounding the mean of the transformed

prediction, and one surrounding the actual sensor the system got, as shown in Figure A.4-2 (b). Then, the system must be able to guess about the readings it would see based on the predicted state with a different guess based on the sensor reading that the system actually observed. In order to estimate these new possible readings (position and velocity estimation in this case) there are two associated probabilities: First, the probability that the sensor mean reading $\vec{z}_k$ is a (mis-) measurement of (z_1, z_2) and second, the probability that the previous estimate (z_1, z_2) is the reading the system should see.



(a) (b)

Figure A.4-2 (a) Random noise between the current read and the possible real one (b) Transformed prediction distribution and sensor measurement distribution

By using basic concepts of statistics, if a system has two probabilities and it is required to know the change that both are true, they must be multiplied together, which is actually the overlap of both distributions. It is a lot more precise than either of previous estimates of the system. The mean of this distribution is the configuration for which both estimates (transformed prediction and actual sensor reading) are most likely, and therefore the best guess of the true configuration given all the information the system has collected. Then, this overlap looks like another *Gaussian distribution* (furthermore, the multiplication between two *Gaussian distributions* with separate means and covariance matrices results in a new *Gaussian distribution* with its own mean and covariance matrix).

A. 5. Combining *Gaussians*

According to the 1D *Gaussian* bell curve with variance σ^2 and mean μ is defined as:

$$N(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \text{(A.12)}$$

$$N(x, \mu_0, \sigma_0) \cdot N(x, \mu_1, \sigma_1) \stackrel{?}{=} N(x, \mu', \sigma') \quad \text{(A.13)}$$

If two normal distributions (μ_0, σ_0 and μ_1, σ_1) are multiplied, the result is the unnormalized intersection. Substituting (A.12) into (A.13) (then renormalizing, so that the total probability is 1), it is obtained:

$$\mu' = \mu_0 + \frac{\sigma_0^2 \cdot (\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2} \quad ; \quad \sigma'^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2} \tag{A.14}$$

Then, it can be observed a common factor $k = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2}$, (A.14) can be rewritten as:

$$\mu' = \mu_0 + k \cdot (\mu_1 - \mu_0) \quad ; \quad \sigma'^2 = \sigma_0^2 - k \cdot \sigma_0^2 \tag{A.15}$$

In addition, (A.15) can be expressed as a matrix version. If Σ is the covariance matrix of a *Gaussian distribution* and $\vec{\mu}$ its mean along each axis, then:

$$K = \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \quad ; \quad \vec{\mu}' = \vec{\mu}_0 + K(\vec{\mu}_1 - \vec{\mu}_0) \quad ; \quad \Sigma' = \Sigma_0 - K\Sigma_0 \tag{A.16}$$

Where K is a matrix called the Kalman gain. Finally, after explaining the effect of uncertainty, external forces and refining the estimate with sensor measurement, the summary of the Kalman filter is as following: There are two distributions: The predicted measurement with $(\mu_0, \Sigma_0) = (H_k\hat{x}_k, H_k P_k H_k^T)$ and the observed measurement $(\mu_1, \Sigma_1) = (\vec{z}_k, R_k)$. Plugging these measurements into equation (A.16) is required to find their overlap (i.e., the best guess of the true configuration given all the information the system has collected, as mentioned above):

$$H_k\hat{x}_k{}' = H_k\hat{x}_k + K(\vec{z}_k - H_k\hat{x}_k) \quad ; \quad H_k P_k{}' H_k^T = H_k P_k H_k^T - K H_k P_k H_k^T \tag{A.17}$$

From (A.16), the Kalman gain can be expressed as:

$$K = H_k P_k H_k^T \left(H_k P_k H_k^T + R_k\right)^{-1} \tag{A.18}$$

Then, simplifying H_k in (I.16) and (I.17), the final equations of the Kalman filter that include the update step are as following:

$$\hat{x}_k{}' = \hat{x}_k + K'(\vec{z}_k - H_k\hat{x}_k) \quad ; \quad P_k{}' = P_k - K' H_k P_k \tag{A.19}$$

$$K' = P_k H_k^T \left(H_k P_k H_k^T + R_k\right)^{-1} \tag{A.20}$$

Where $\hat{x}_k{}'$ is the new best estimate and $P_k{}'$ the new uncertainty (based on predicted state and sensor readings). In order to improve this estimation, this best estimation can be feed it (along with $P_k{}'$ back into another round of predict or update as many times as is required). In conclusion, the main Kalman filter equations are (A.11), (A.18) and (A.19). This allows to model any linear system accurately. For non-linear systems, Extended Kalman Filter (EKF) must be used, which works by simply linearizing the predictions and measurements about their mean. Figure A.5-1 shows the Kalman Filter Information Flow, which sums up what is stated in this appendix.

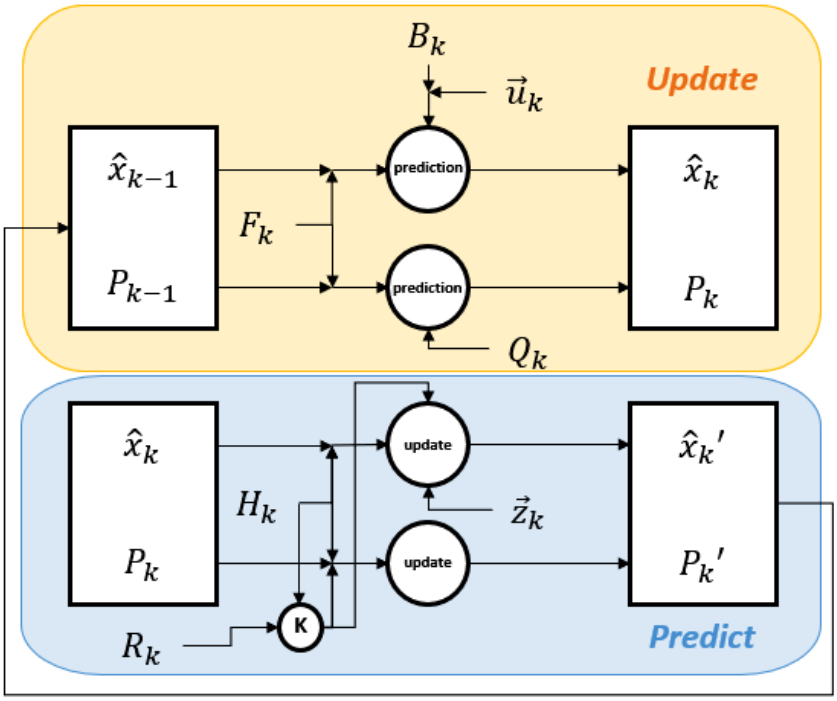


Figure A.5-1 Kalman Filter Information Flow

Appendix B: Artificial Intelligence

This appendix aims to summarize the main artificial intelligence concepts addressed in this master thesis. Artificial Intelligence has been witnessing an exponential growth in bridging the gap between the capabilities of machines and humans. Although the terminologies *Artificial Intelligence* (AI), *Machine Learning* (ML) and *Deep Learning* (DL) are usually used interchangeably, they do not quite refer to the same issues. Figure B.1-B-1 depicts how DL is a subset of ML, which is also a subset of AI. AI is the all-encompassing concept that initially erupted, followed by ML and lastly DL that is promising to escalate the advances of AI to another level.

B. 1. Artificial Intelligence concept

Artificial Intelligence is intelligence shown by machines, in contrast to the natural intelligence demonstrated by humans. It represents a broader concept that consists of everything from GOFAI (Good Old-Fashioned AI, also known as symbolic AI, collection of all methods in AI based on high-level symbolic representations of problems, that is, logic and search) to futuristic technologies such as deep learning.

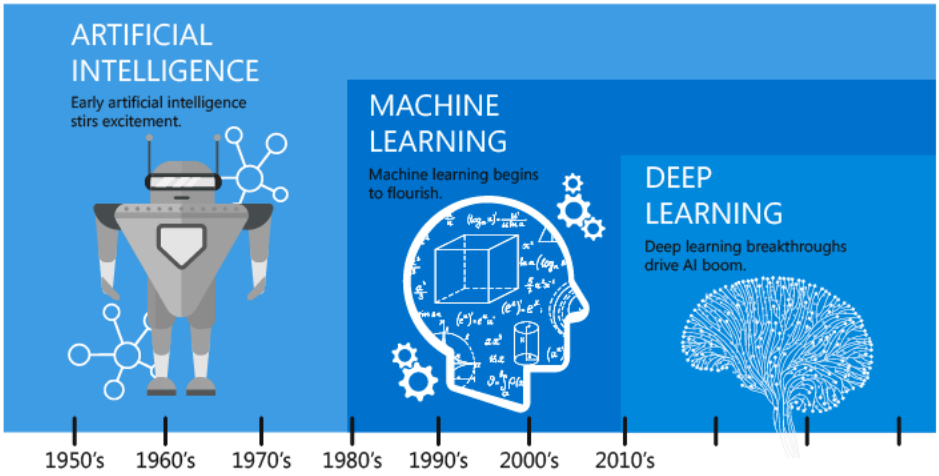


Figure B.1-B-1 Development of artificial intelligence and its subsequent fields in the last six decades

If a machine performs a task based on a certain algorithm (set of stipulated rules that solve problems), like an intelligent behaviour is what is called artificial intelligence. An example can be found in robot that can move and manipulate objects in warehouses.

AI powered machines are usually classified into two groups:

- *General AI:* Also known as "Strong AI", "Full AI" or "True AI", machines can intelligently solve general problems, for example recognizing if someone has raised

the hands or moving a simple box. The machine has the capacity to understand or learn any intellectual task that a human can.

- *Narrow AI:* The machine or technology outperforms humans in some very narrowly defined task, focusing on a single subset of cognitive abilities and advances. For example, nowadays computers are able to surpass humans in the playing of complex games like GO or chess, making intelligent business decisions or classifying images and tracking objects.

B. 2. Machine Learning concept

As mentioned above, *Machine Learning* (ML) was born in the 80s, whose intention was to enable machines to learn by themselves using the provided data and make accurate predictions. Indeed, it can be considered a technique for realizing AI as method of training algorithms in such a way they can learn how to make decisions.

Then, training in machine learning is based on giving a lot of data to the algorithm and allowing it to learn more about the processed information.

B. 3. Deep Learning concept

In the same way that ML corresponds a subset of AI, *Deep Learning* (DL) is a subset of ML. Indeed, it is a technique for realizing machine learning, being DL the next evolution of machine learning.

Deep learning algorithms are inspired by the information processing patterns found in the human brain. In the same way that a human brain can identify patterns and classify various types of information (for example, object detection, the first step to perform object tracking), DL algorithms can be taught to carry out the same tasks for machines. Whenever the human receives a new information (car on the road, pedestrian crossing 50 meters ahead), the brain tries to compare it to a known item before making sense of it, which is the same concept employed by DL algorithms.

While ML requires to be provided manually of features for classification, DL can automatically discover them. Furthermore, DL requires high-end machines and considerable huge amounts of training data in order to achieve accurate results.

In conclusion, Deep Learning is the most successful direction in the field of ML. Since proposed, it has given rise to revolutionary progress and breakthrough in several aspects of information processing like text, image, video or voice. The advantage of DL is mainly reflected in the powerful ability in feature expression. Through the multi-level learning and mapping, deep neural networks can obtain high-level abstract features from colours, edges and other low-level features gradually. While ML techniques required professional manual design with traditional feature, DL performs feature extraction in an automatic way.

Below it is explained a brief explanation of the main DL concepts used for Multi-Object Tracking, that is Convolutional Neural Networks (CNNs) and RNNs (Recurrent Neural Networks), in order to appreciate the particularities of the state-of-the-art approaches of this master thesis such as GOTURN, MV-YOLO, MDNet, ROLO, Re3 or Deep SORT.

B. 4. Convolutional Neural Networks (CNNs)

The advancements in Computer Vision with Deep Learning has been developed and improved with time, primarily over one particular algorithm: Convolutional Neural Network.

A *Convolutional Neural Network* [22] (CNN/ConvNet) is a Deep learning algorithm which can take in an input image, assign importance (learnable weights and biases) to some objects/aspects in the image, being able to differentiate one from the other (that is, assigning a different semantic identification, such as dog, car or bycicle). The pre-processing process required in a CNN is much lower than other classification algorithms. While in primitive methods filters (Machine Learning algorithms) are hand-crafted, with enough training, CNNs have the ability to learn these characteristics/features. Figure B.4-1 illustrates how ML algorithms require a particular stage for feature extraction before classifying the input variable, in DL algorithms, and furthermore in a CNN, the feature extraction and classification is performed at the same level, so required pre-processing process is decreased.

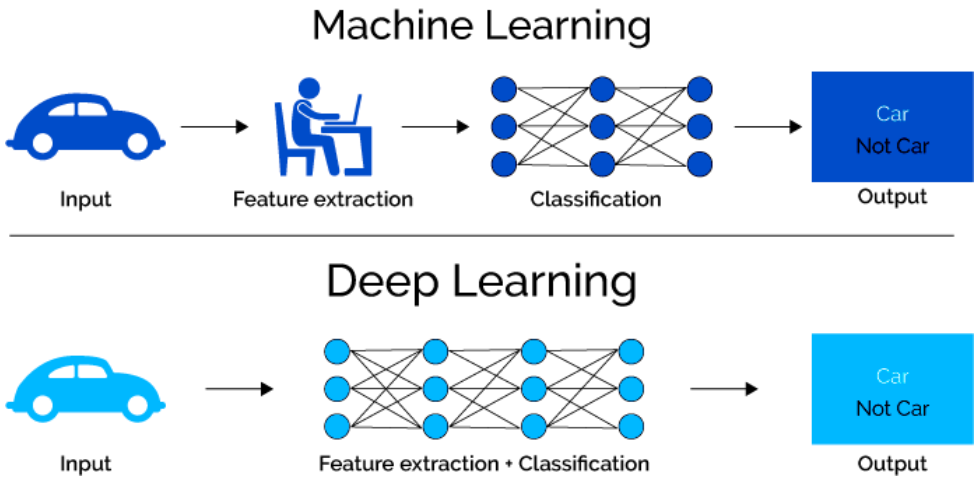


Figure B.4-1 Comparison between stages required by ML and DL for classification

The architecture of a CNN is analogous to the connectivity pattern of Neurons in the Human Brain, inspired by the organization of the Visual Cortex. In that sense, individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the whole visual area.

An image is actually a matrix of pixel values, it could be flattened (e.g., 4x4 image matrix into a 16x1 vector) and feed it to a Multi-Level Perceptron, which is not a recent technique [23]. However, in cases of extremely basic binary images, the method could show an average precision score while performing prediction and estimation of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

On the other hand, a ConvNet is capable of successfully capturing the Spatial and Temporal dependencies in an image through the application of relevant filters. In that direction, a CNN architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In conclusion, the network can be trained to understand the sophistication of the image better.

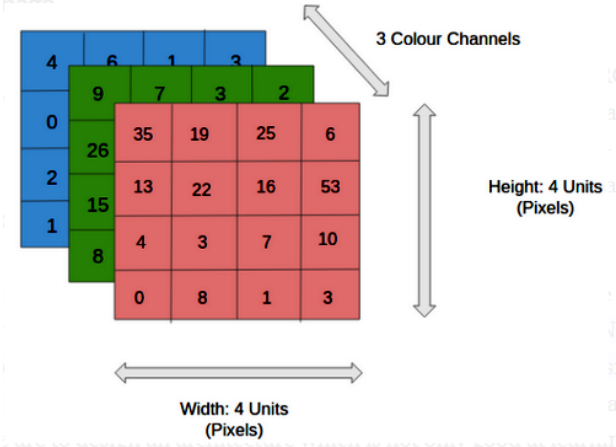


Figure B.4-2 RGB image channels and pixel correspondence

Figure B.4-2 represents an example of RGB image which has been separated by its three colour planes, that is, Red, Green and Blue. There are a number of such colour spaces in which images exist, such as RGB, HSV, CMYK, Grayscale, etc.

As mentioned above, the classification and prediction of the objects in the scene could be relatively well-addressed if the image is binary and the dimensions are small. However, when speaking about computation time it is easy to realize how computationally intensive things would get once images reach dimensions, for example 8K (768 x 4320). Then, the role of the CNN is to reduce the images into a simpler form which is easier to address, without leasing critical features which are essential for getting a good estimation and classification. It is a key concept when a novel architecture is proposed, since an architecture does not have to be only good at learning features but also scalable to massive datasets.

B. 4.1. Convolution Layer - The Kernel

Figure B.4-3 represents the convolution process of 5x5x1 (only a colour channel) with a 3x3x1 kernel in order to obtain a convolved feature with 3x3x1 dimensions. In terms of image processing, convolution is the process of adding each element of the image to its local

neighbours, weighted by a kernel. The matrix operation being performed (also known as convolution process) is not a traditional matrix multiplication but the result of applying the kernel over a zone of the image is the sum of dot product between both matrices. The kernel (also known as mask or convolution matrix) is a small matrix (smaller than image dimensions) used for sharpening, edge detection, blurring and more. Then, the element involved in carrying out the convolution operation in the first part of the Convolutional Layer is called the Kernel/Filter (K), represented in yellow in Figure B.4-3.

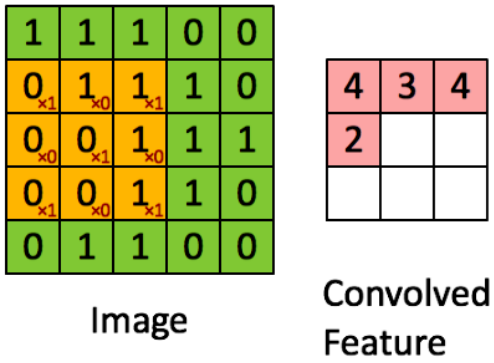


Figure B.4-3 Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature

When speaking about CNNs, there are two key concepts related with the application the kernel, that is, padding and stride:

- *Stride:* Distance between spatial location where the convolution kernel is applied. In default scenarios, the distance usually is 1 in each dimension (also the default value in TensorFlow), that is, each time the kernel moves from left to the right it is only displaced one position, and the same when it has finished a horizontal movement and moves down. When the stride is larger than one, it is usually called stride convolution to make the difference explicit with *non-stride* (standard) convolutions.

- *Padding:* Since the result of the convolution between the kernel and the input image results in a shrank output image, it is easy to realize that corner pixels will only get covered one time while middle pixels will get covered more than once, giving rise to a main downside, that is, loosing information on corner of the image. To overcome this padding is introduced to the image. Padding is an additional layer added to the border of an image, in order to take into account more times the edge and corner pixels. Figure B.4-4 represents an example of padding (4x4x1 input image is padded with 0s to create a 6x6x1 image) and stride (1,1, that is, standard movement both in X and Y direction), resulting in an output image with identical dimensions than input image.

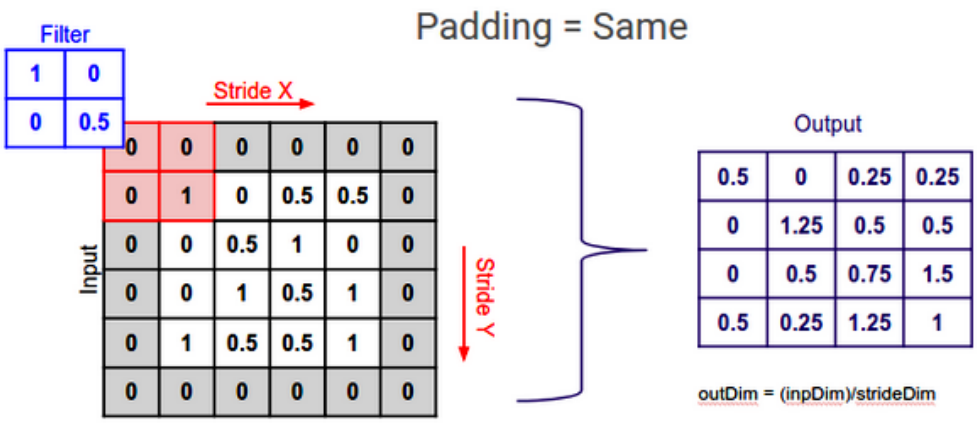


Figure B.4-4 Example of padding and stride in the input image

In the case of Figure B.4-3, the kernel shifts 9 time since the stride length = 1, so non-strided, every time performing a matrix multiplication operation between K and the porting T of the image over which the kernel is hovering.

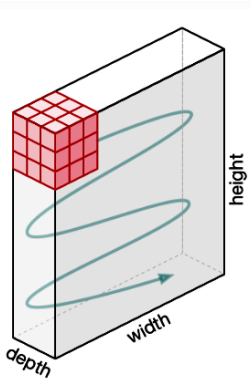


Figure B.4-5 Movement of a 3D kernel over the image

The filter moves to the right with a certain stride value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same stride value (if X stride value and Y stride value are identical) and repeats the process until the whole image is traversed, as shown in Figure B.4-5.

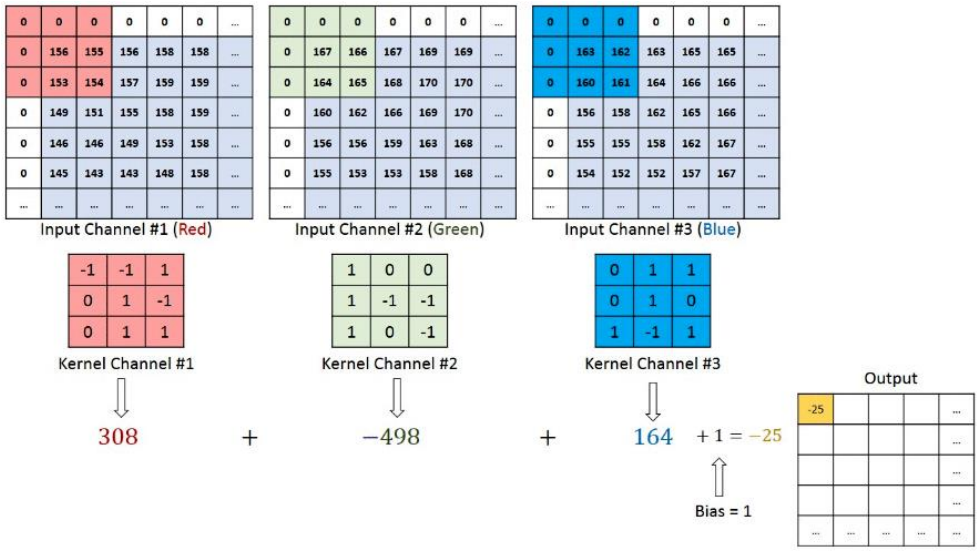


Figure B.4-6 Convolution operation of a MxNx3 image matrix with a 3x3x3 kernel

It is important to consider that in the case of images with multiple channels (e.g. RGB), the kernel has the same depth as that of the input image. Then, matrix multiplication is carried out between K_n and I_n stack ($[K_1, I_1]$; $[K_2, I_2]$; $[K_n, K_n]$) and all the results are summed with the (optional) bias to give the user a squashed one-depth channel Convoluted Feature Output, as shown in Figure B.4-6.

The objective of the convolution operation is to extract the *high-level* features (complex shapes) from the input image. However, CNNs need not be limited to only one Convolutional Layer. Conventionally, the first CNN is responsible for capturing the *low-level* features (gradient orientation, edges, colour, dots, etc.), while subsequent conv-layers the architecture adapts to the high-level features (built on top of low-level features to detect objects and larger shapes in the image) as well, giving rise to a network which has the wholesome understanding of images in the dataset, similar to how the human would. Figure B.4-7 shows an example of low, middle and high-level feature extraction.

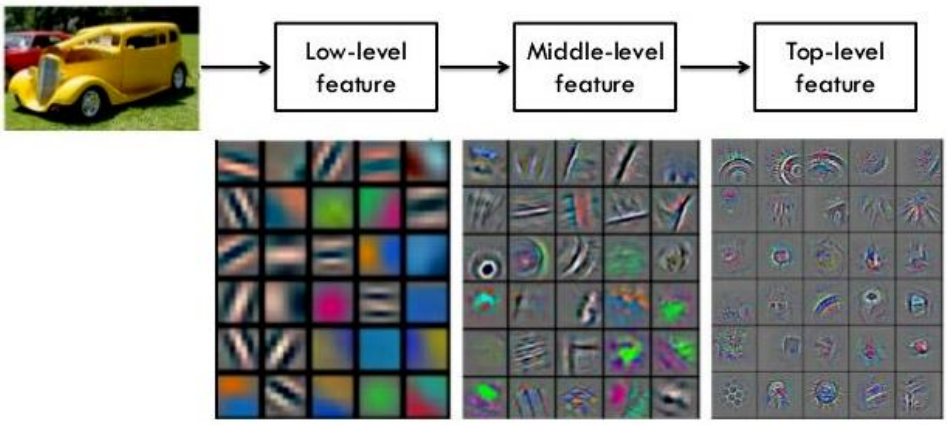


Figure B.4-7 Low, Middle and High-Level feature extraction

B. 4.2. Pooling layer

In a similar way to the conv-layer, the *pooling layer* is responsible for reducing the spatial size of the convolved feature. This is very important to decrease the computational time (and power) required to process the data through dimensionality reduction. Moreover, it is useful for extracting dominant features that are positional and rotational invariant (keeping the process of effectively training of the model).

As shown in Figure B.4-8, the main types of pooling are *Max Pooling* and *Average Pooling*. While Max Pooling returns the maximum value from the portion of the image covered by the kernel, average pooling returns the average of all the values covered. In addition, Max Pooling performs as *Noise Suppressant*, that is, it discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Then, it is said that Max Pooling performs a lot better than Average Pooling.

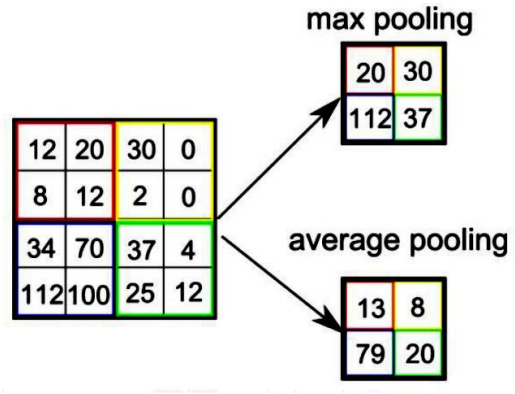


Figure B.4-8 Types of pooling

The conv-layer and the pooling-layer together form the basic *i-th* layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

B. 4.3. Classification – Fully Connected Layer

After going through the above layers (conv and pooling), the basic model of CNN to understand the features is explained. Then, the final output must be flattened and feed it to a standard Neural Network for classification purposes (Figure B.4-9).

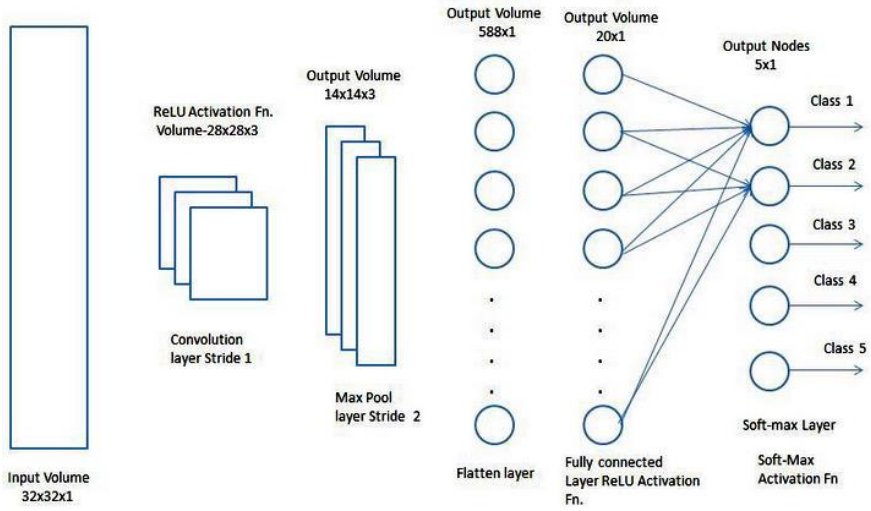


Figure B.4-9 Example of Fully Connected Layer (FC Layer)

Adding a FC layer is a (usually) cheap way of learning non-linear combinations of the high-level features, as represented by the output of the conv-layer. The FC layer is learning a possibly non-linear function in that space. After converting the input image into a suitable for previous layers of the network, now the final image has to be flattened into a column vector. Then, the flattened output is fed to a Feed-Forward Neural Network, where backpropagation [25] is applied to every iteration of training. Over a series of epochs (an epoch can be defined when an entire dataset is passed both forward and backward through the neural network only once), the model is able to distinguish between low-level and dominating features in images and classify them using the Softmax Classification technique [26].

B. 5. Recurrent Neural Networks (RNNs)

While CNNs are able to analyse only a single frame of the image (detecting and classification the objects found in the scene), a *Recurrent Neural Network* (RNN) is a kind of neural network that runs on a sequential data. It basically makes predictions based on the current input and the previous state of the network. In RNN, the main element that stores the condition or values of the previous state is known as the RNN hidden layer. This hidden layer works as a control unit for the network. So as to make good predictions, the RNN hidden state makes sure a smooth transition between previous and current status, by making sure that the difference between the previous state and the current one of the network does not exceed a given limit.

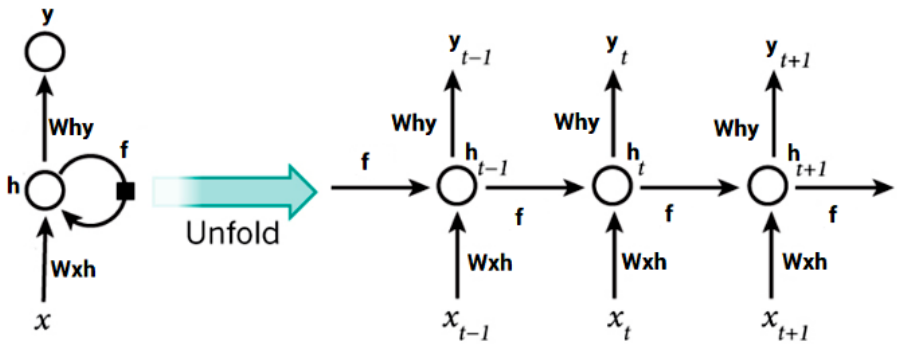


Figure B.5-1 (Left) A standard RNN (Right) Unrolled RNN in time

Figure B.5-1 shows the function of a standard RNN. It can be appreciated that y_t is just a function of hidden state of the RNN, that is, h_t (value of the hidden state at time t) multiplied by the weight matrix W_y plus the bias term b_y (overall a linear transformation):

$$y_t = W_y h_t + b_y \tag{B.1}$$

On the other hand, the value of the hidden state h_t is estimated as:

$$h_t = g_h(W_I x_t + W_R h_{t-1} + b_h) \tag{B.2}$$

Where W_I represents the weight matrix (also known as learnable parameters) for linear transformation of input variable x_t at time t and W_R represents the weight matrix the carry out the transformation of hidden state of previous time step $t-1$, shared across the layers in time. Weighted information from both hidden state of previous time step h_{t-1} and current input x_t are added together with the bias (offset) term b_h, and then passed through a non-linear activation function g_h to estimate h_t, i.e. the value of the new hidden state of the RNN at time t. Non-linear activations are one the key concepts of the Deep-Learning breakthrough. Both in CNNs, RNNs or other types of DL networks, if is not applied an activation function then the output signal would be a simple linear function (polynomial of one degree), which is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data, i.e., a Neural Network without activation functions would simply be a Linear Regression Model. For that reason, *Non-linear functions* (as g_h) are those which have degree more than one (considered as *Universal Function Approximators*) and are required by Neural Network Models to learn and represents almost anything and any arbitrary complex function which maps inputs to outputs. One thing to note in Figure B.5-1 is that the activity of the unit h_t depends not only on the input x_t but also on the activity at the previous timestep h_{t-1}, so the activity at the memory unit at time t is passed on to the unit activity at time $t+1$. This is the key concept by which RNNs have begun to be intensively studied to perform tasks that require information from previous moments, as is the case with Multi-Object Tracking, where previous moments must be taken into account for assignment and tracking of multiple objects.

B. 5.1. Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a special purpose RNN. An LSTM cell is usually comprised of three gates which basically controls the flow of the information. These gates are:

- *Forget gate:* A memory forgetting mechanism:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{B.3}$$

- *Input gate:* A memory saving mechanism:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{B.4}$$

- *Output gate:* A memory focusing mechanism, which saves the long-term memory into workable memory:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{B.5}$$

Is important to consider that x_t represents the input to the current RNN layer (where LSTM cell is located) at time t and h_{t-1} represents the previous state of this RNN layer.

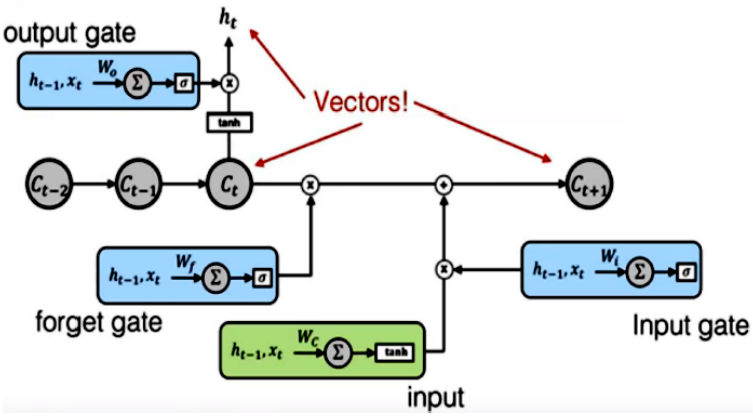


Figure B.5-2 A basic LSTM cell

From the mathematical derivation of the gates, it can be appreciated that each gate performs a function of a typical fully-connected (FC) layer of a neural network. These gates take h_{t-1} (previous hidden state) and x_t (current input) as inputs for estimating the current state. Figure B.5-2 illustrates a basic LSTM cell, where each gate has small memory block. Combining these gates, they form a strong memory unit which is helpful in solving non-trivial tasks of sequential processing and data associations in temporal domain. $\tilde{C}_t$ represents the current input to the LSTM cell at time t:

$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_c) \tag{B.6}$$

Where W_C represents the weight matrix which computes the new input for the LSTM memory cell based on x_t and h_{t-1}. On the other hand, the output of the LSTM memory cell is calculated by applying non-linearity to the current memory state C_t and then multiplying with the states of the output gate:

$$h_t = o_t tanh(C_t) \tag{B.7}$$

Finally, the new state of the LSTM memory cell for frame $t+1$ is a weighted combination of the current in current input to the LSTM memory cell $\tilde{C}_t$ and the current state C_t:

$$C_{t+1} = f_t C_t + i_t \tilde{C}_t \tag{B.8}$$

In other words, the activity of the cell at time $t+1$ (the next frame) is equal to the memory or the amount memory to be forgotten from the previous time step (C_t) plus the input at the new timestep which is multiplied by how much the requirements want to accept from this new input got at timestep t, or what is the same, the new memory of the LSTM cell C_{t+1} is basically a weighted sum between the amount of information that the input and forget gates allow to flow.

Appendix C: Code of Interest

In this appendix it is shown some interesting parts of the code created and modified in order to perform most of exposed tasks throughout this master thesis. For a deeper explanation, the reader should go to the corresponding algorithm in the code in addition to different bibliographic source that have been used.

Code of Interest C-1 Code to obtain the 3D coloured point cloud

```
for (pcl::PointCloud<pcl::PointXYZRGB>::iterator pt = coloured->points.begin(); pt < coloured->points.end(); pt++)
    {
          cv::Point3d pt_cv((*pt).x, (*pt).y, (*pt).z);
          cv::Point2d uv;
          uv = cam_model_.project3dToPixel(pt_cv);

          // CARLA simulation

          if(uv.x>0 && uv.x < image.cols && uv.y > 0 && uv.y < image.rows &&(*pt).z>=0)
          {
                  (*pt).r = image.at<cv::Vec3b>(uv)[0];
                  (*pt).g = image.at<cv::Vec3b>(uv)[1];
                  (*pt).b = image.at<cv::Vec3b>(uv)[2];
          }
          else  // color black cloud to white/grey
          {
                  (*pt).r = 20.0;
                  (*pt).g = 20.0;
                  (*pt).b = 20.0;
          }
    }

    // Publish coloured PointCloud

    sensor_msgs::PointCloud2 pcl_colour_ros;
    pcl::toROSMsg(*coloured, pcl_colour_ros);
    pcl_colour_ros.header.stamp = pcl_msg->header.stamp ;
    pcl_pub.publish(pcl_colour_ros);
}

int main(int argc, char **argv){
    ros::init(argc, argv, "pcl_coloring");
    ros::NodeHandle nh_("~"); // LOCAL
    // Parameters
    nh_.param<std::string>("target_frame", target_frame, "/ego_vehicle/camera/semantic_segmentation/semantic");
    nh_.param<std::string>("source_frame", source_frame, "/ego_vehicle/lidar/lidar1");

    // Subscribers
    message_filters::Subscriber<PointCloud2> pc_sub(nh_, "pointcloud", 1);
    message_filters::Subscriber<CameraInfo> cinfo_sub(nh_, "camera_info", 1);
    message_filters::Subscriber<Image> image_sub(nh_, "image", 1);

    pcl_pub = nh_.advertise<PointCloud2> ("velodyne_coloured", 1);

    typedef message_filters::sync_policies::ApproximateTime<PointCloud2, CameraInfo, Image> MySyncPolicy;
    // ExactTime takes a queue size as its constructor argument, hence MySyncPolicy(10)
    message_filters::Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), pc_sub, cinfo_sub, image_sub);
    sync.registerCallback(boost::bind(&callback, _1, _2, _3));
```

Code of Interest C-2 Code to concatenate non-discarded CenterNet bounding boxes

```
def bbox_to_xywh_cls_conf(bbox):

    type_object_list = [1,2,3] # person_id = 1, bicycle_id = 2, car_id = 3

    bbox_of_interest = []

    k = 0

    for i in type_object_list:
        bbox_object = bbox[i]
        r,c = bbox_object.shape
        aux = np.zeros((r,1))
        aux.fill(i)
        bbox_object = np.concatenate([bbox_object,aux],1)

        if (k==0):
            bbox_of_interest = bbox_object
        else:
            bbox_of_interest = np.concatenate([bbox_of_interest,bbox_object])

        k = k+1

    bbox = bbox_of_interest

    if any(bbox[:,4] > opt.vis_thresh):

        bbox = bbox[bbox[:,4] > opt.vis_thresh, :]
        bbox[:,2] = bbox[:,2] - bbox[:,0]
        bbox[:,3] = bbox[:,3] - bbox[:,1]

        return bbox[:,:4], bbox[:,4], bbox[:,5]

    else:

        return None, None, None
```

Code of Interest C-3 Code to update the tracked objects based on Deep SORT

```
def update(self, bbox_xywh, confidences, ori_img):
        self.height, self.width = ori_img.shape[:2]

        # generate detections
        try :
            features = self._get_features(bbox_xywh, ori_img)
        except :
            print('a')
        detections = [Detection(bbox_xywh[i], conf, features[i]) for i,conf in enumerate(confidences) if conf>self.min_confidence]

        # run on non-maximum supression
        boxes = np.array([d.tlwh for d in detections])
        scores = np.array([d.confidence for d in detections])
        indices = non_max_suppression( boxes, self.nms_max_overlap, scores)
        detections = [detections[i] for i in indices]

        # update tracker
        self.tracker.predict()
        self.tracker.update(detections)

        # output bbox identities
        outputs = []
        for track in self.tracker.tracks:
            if not track.is_confirmed() or track.time_since_update > 1:
                continue
            box = track.to_tlwh()
            x1,y1,x2,y2 = self._xywh_to_xyxy_centernet(box)
            track_id = track.track_id
            outputs.append(np.array([x1,y1,x2,y2,track_id], dtype=np.int))
        if len(outputs) > 0:
            outputs = np.stack(outputs,axis=0)


        return outputs
```

Code of Interest C-4 Code to perform the 3D LiDAR Point Cloud cluster extraction

```
// Auxiliar variables

pcl::PointCloud<pcl::PointXYZRGB>::Ptr vlp_cloud_Ptr (new pcl::PointCloud<pcl::PointXYZRGB>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZRGB>);

// Read in the cloud data

pcl::PCDReader reader;

// If Full Cloud is used ...

pcl::fromROSMsg(*lidar_msg, *vlp_cloud_Ptr); // vlp_cloud_Ptr stores the LiDAR point cloud
*cloud_filtered = *vlp_cloud_Ptr;

// Filter cloud

pcl::PointCloud<pcl::PointXYZRGB> cl_filter = xyz_filter(cloud_filtered);
*cloud_filtered = cl_filter;

sensor_msgs::PointCloud2 cloud;
pcl::toROSMsg(*cloud_filtered, cloud);
//cloud.header.frame_id = "velodyne";
cloud.header.frame_id = "ego_vehicle/lidar/lidar1";
cloud.header.stamp = lidar_msg -> header.stamp;

// Publish only LiDAR cloud

pointcloud_only_laser_pub_.publish(cloud);

// Cluster Segmentation

// Creating the KdTree object for cluster extraction (Cloud filtered)

pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZRGB>);
tree -> setInputCloud (cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
ec.setClusterTolerance (1);
ec.setMinClusterSize (2);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered); // Clusters will be obtained from this filtered cloud
ec.extract (cluster_indices);
```

Code of Interest C-5 Code to project the bottom position of the CenterNet+DeepSORT bounding box onto the 3D space

```
def Image_to_RealWorld(self, color, object_score, object_id, obj_coordinates, object_type, ori_im):

    tracked_obstacle = yolo_obstacle()

    proj_matrix = np.matrix([[0.0018621149884284534333, 0, -1.1917535925942101973, 0],
    [0, 0.0018621149884284534333, -0.67036139583424323599, 0], [0, 0, 1, 0], [0, 0, 0, 1]])

    camera_height = 1.95 # ZED camera in SmartElderlyCar

    tracked_obstacle.type = object_type

    # Bounding Box coordinates in camera frame

    tracked_obstacle.xl = obj_coordinates[0]
    tracked_obstacle.yl = obj_coordinates[1]
    tracked_obstacle.x2 = obj_coordinates[2]
    tracked_obstacle.y2 = obj_coordinates[3]

    # 3D box dimensions that ties the object

    tracked_obstacle.h = 2.0
    tracked_obstacle.w = 1.0
    tracked_obstacle.l = 1.0

    # Object score

    tracked_obstacle.probability = object_score

    # Object ID

    tracked_obstacle.object_id = object_id

    # Object colors (since the colours in Yolo detection are in BGR,
    # so they must be converted to RGB for ROS topic )

    tracked_obstacle.color.a = 1.0

   tracked_obstacle.color.r = color[0]/255
   tracked_obstacle.color.g = color[1]/255
   tracked_obstacle.color.b = color[2]/255

   # Image world to Real world

   centroid_x = (tracked_obstacle.xl+tracked_obstacle.x2)/2

   pixels = np.matrix([[centroid_x], [tracked_obstacle.y2], [1], [1]])

   p_camera = proj_matrix*pixels

   K = camera_height/p_camera[1]

   p_camera_meters = p_camera*K

   tracked_obstacle.tx = float(p_camera_meters[2])
   tracked_obstacle.ty = float(-p_camera_meters[0])
   tracked_obstacle.tz = float(-p_camera_meters[1])

   # Append single tracked obstacle to tracked obstacle list

   self.tracked_obstacle_list.yolo_list.append(tracked_obstacle)
   self.tracked_obstacle_list.header.stamp = rospy.Time.now()
```

Code of Interest C-6 Code to perform the sensor fusion between BEV VOT proposals and BEV LiDAR proposals

```
for (int i=0; i<centernet_msg->centernet_list.size(); i++) // CenterNet is more restrictive than LiDAR
{
        float max_diff_lidar_vot = 4; // Initialize maximum allowed difference

        ycx = float(yolo_msg->yolo_list[i].tx); // Distance to the object-camera in Velodyne frame
        ycy = float(yolo_msg->yolo_list[i].ty);

        if (only_laser_objects.size() > 0 && (!strcmp(centernet_msg->yolo_list[i].type.c_str(),"car") ||
        !strcmp(centernet_msg->centernet_list[i].type.c_str(),"person")))
        {
                //cout<<endl<<"Yolo msg list size: "<<yolo_msg->yolo_list.size()<<endl;
                double time = centernet_msg->header.stamp.toSec();
                object_id = centernet_msg->centernet_list[i].object_id;

                geometry_msgs::PointStamped local_centroid;
                geometry_msgs::Point32 global_centroid;

                local_centroid.point.x = ycx;
                local_centroid.point.y = ycy;
                local_centroid.point.z = 0;

                global_centroid = Local_To_Global_Coordinates(local_centroid);

                // Find closest LiDAR object with respecto to VOT

                for (int j=0; j<only_laser_objects.size(); j++)
                {
                        lcx = float(only_laser_objects[j].centroid_x);
                        lcy = float(only_laser_objects[j].centroid_y);

                        diff_lidar_vot = float(sqrt(pow(ycx-lcx,2)+pow(ycy-lcy,2)));

                        if (diff_lidar_vot < max_diff_lidar_vot) // Find the closest cluster
                        {
                                max_diff_lidar_vot = diff_lidar_vot;
                                index_most_similar_lidar_vot = j;

                        }
                 }

                 if (max_diff_lidar_vot < 3)
                 {
                        only_laser_objects[index_most_similar_lidar_vot].type = centernet_msg->yolo_list[i].type;
                        only_laser_objects[index_most_similar_lidar_vot].r = centernet_msg->yolo_list[i].color.r;
                        only_laser_objects[index_most_similar_lidar_vot].g = centernet_msg->yolo_list[i].color.g;
                        only_laser_objects[index_most_similar_lidar_vot].b = centernet_msg->yolo_list[i].color.b;
                        only_laser_objects[index_most_similar_lidar_vot].a = centernet_msg->yolo_list[i].color.a;
                 }
          }
   }
```

Appendix D: User's manual

This appendix presents the installation guide and execution process in order to reproduce the obtained results. Due to the numerous tools involved in this work, only the most relevant are commented, assuming that the reader has some knowledge of the Linux OS.

As commented throughout this work, the architecture proposal for Deep Learning based Multi-Object Tracking has been implemented in ROS, making use of the its different packages, in order to provide proper intercommunication with the SmartElderlyCar project.

D. 1. Docker installation

First, Docker must be installed, in this case in Linux:

https://docs.docker.com/install/linux/docker-ce/ubuntu/)

1. Install packages to allow *apt* to use a repository over HTTPS:

```
sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
```

2. Add Docker's official GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –
sudo apt-key fingerprint 0EBFCD88
```

3. Set up the stable repository for the Ubuntu host release:

```
sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"
```

4. Install Docker Engine – Community:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

This installation guide must be carried out if a new computer wants to use Docker services.

In the case of the SmartElderlyCar project, it is encapsulated in a Docker Image with Ubuntu 14.04 and ROS Indigo Igloo, and then cloning the project in the corresponding Docker

container from the RobeSafe GitHub. This user's manual does not cover the SmartElderlyCar packages installation due to the magnitude of the project.

On the other hand, in the case of the tracking module, it has been encapsulated in a Docker Image with Ubuntu 18.04. It was performed by typing on the Ubuntu host terminal:

docker pull ubuntu:18.04

Where docker is the name of the Docker group, pull is a command to pull image from repositories, as explained in previous chapters, ubuntu is the general repository of the image and 18.04 its particular, since it also can be found other Ubuntu versions such as 16.04 Xenial or 19.04 Disco.

In order to run this docker image for the first time, since it has not registered the host user in the original Docker File, the container must be run as *root*, as shown Figure D.1-1, using a .sh file named *launch_docker_version.sh*.

```
#!/bin/bash

directorio=$HOME/compartido_con_docker:$HOME/compartido_con_docker

docker run -it --net host --name=CenterNet_DeepSORT --privileged -u root -v /
tmp/.X11-unix:/tmp/.X11-unix -v $directorio -v $directorio_rosbag -e
DISPLAY=unix$DISPLAY $1 /bin/bash
```

Figure D.1-1 Bash launch file to run the CenterNet_DeepSORT docker image

Now, in order to run the image, in the Ubuntu host terminal it must be typed:

./launch_docker_version.sh CenterNet_DeepSORT:last

It is important to consider that this reposity (CenterNet_DeepSORT) and tag (last) were not the originals but ubuntu and 18.04. It can be easily modified renamed using the following command:

docker tag ORIGINAL_DOCKER_IMAGE_ID CenterNet_DeepSORT:last

At this point, the user is inside the Docker container.

D. 2. ROS installation

Before installing the CenterNet+DeepSORT framework it is advisable to install ROS in such a way this framework is installed in the *source* (src) directory of ROS and later modifications can be compiled in an easy way.

In this docker image it is installed ROS Melodic Morenia since it is the last LTS (Long Term Support) of ROS1. In addition, ROS2 was installed and the bridge between ROS1 and ROS2 was configured, but it is still in development.

To install ROS Melodic Morenia:

http://wiki.ros.org/melodic/Installation/Ubuntu

1. Setup the sources.list

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

2. Setup the keys

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

3. Installation

sudo apt update
sudo apt install ros-melodic-desktop-full
4.Initialize *rosdep* (it enables to easily install system dependencies and run some core components in ROS):

sudo rosdep init
rosdep update

5. Create a ROS Workspace:

mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make

D. 3. Anaconda, CUDA and NVIDIA driver

Anaconda is the easiest way to install Python 3.7 and other Python requirements in other to run the tracking module. For Linux distribution, it can be downloaded from:

https://www.anaconda.com/distribution/

Then, after the download is complete, in the docker terminal must be typed:

chmod +x Anaconda3-2019.07-Linux-x86_64.sh (in order to give execute permissions)
sh Anaconda3-2019.07-Linux-x86_64.sh

On the other hand, CUDA and CuDNN are required in order to optimize the neural network computation. According to the Ubuntu version of the docker image (18.04 Bionic), the most advisable CUDA version is 10.1, as shown in:

https://developer.nvidia.com/cuda-downloads?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1804&target_type=deblocal

It is recommended to use the *deb (local)* installer type, so the steps are as following:

wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-ubuntu1804.pinsudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600

wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers /cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb

sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda

To check the CUDA version, run the following command:

nvcc --version

There should appear the installed CUDA version as 10.1.

Finally, in order to run graphic interfaces inside the Docker, both host machine and Docker image must be synchronized in terms of NVIDIA driver. In contrary case, when typing nvidia-smi command, there should be an error message as follows:

Failed to initialize NVML: Driver/library version mismatch

To synchronize both graphical interfaces (host machine and Docker), the driver must be downloaded from NVIDIA driver releases:

https://www.nvidia.es/Download/index.aspx?lang=es

It must be stored in the shared volume (in this master thesis called *compartido_con_docker*) to get access to this host file inside the Docker image. Now open a new shell with CTRL+ALT+F2 and go to the shared volume where the NVIDIA driver. Then type:

sudo ./NVIDIA-Linux-x86_64-VERSION.run --no-kernel-module

If the system reports that a previous NVIDIA driver was installed, just continue the installation process. Finally, typing ***nvidia-smi*** command in the docker terminal it should report a message as follows (e.g., in the case of 390.67 NVIDIA Driver):

```
robesafe@robesafe-X555LD:~$ nvidia-smi
Wed Jul 25 12:58:43 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.67                 Driver Version: 390.67                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce 820M        Off  | 00000000:04:00.0 N/A |                  N/A |
| N/A   48C    P8    N/A /  N/A |    210MiB /   964MiB |     N/A      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0                    Not Supported                                       |
+-----------------------------------------------------------------------------+
```

Figure D.3-1 Check NVIDIA driver in the case of 390.97

D. 4. CenterNet+DeepSORT framework installation

After ROS installation, CenterNet+DeepSORT is installed. To do this, the first step is to download the files from:

https://github.com/kimyoon-young/centerNet-deep-sort.git

And copy then in the *src* directory of the previously created ROS Workspace *catkin_ws*. Once the files have been copied, inside this centernet directory:

conda env create -f CenterNet.yml
pip install -r requirements.txt

Then, a virtual environment named *CenterNet* is created with the specified requirements. To run this virtual environment:

conda activate CenterNet

In summary, Figure D.4-1 shows an example of how should be configured the Python

```
# For CARLA use
export ROS_MASTER_URI=http://localhost:11311
export ROS_IP=localhost

# Modify to operate with the SEC

# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/root/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/root/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/root/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/root/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<

#export PYTHONPATH=$PYTHONPATH:/root/anaconda3/envs/CenterNet/bin/python3.6

#export PATH=$PATH:/usr/local/cuda-9.2/bin

export PYTHONPATH=/root/ros2_ws/install/lib/python3.6/site-packages:/root/ros_ws/devel/lib/python2.7/dist-packages:/opt/ros/dashing/lib/python3.6/site-packages:/root/anaconda3/envs/CenterNet/bin/
python3.6

export PATH=/root/anaconda3/condabin:/opt/rti.com/rti_connext_dds-5.3.1/lib/x64Linux3gcc5.4.0:/opt/rti.com/rti_connext_dds-5.3.1/bin:/opt/ros/dashing/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/local/cuda-9.2/bin

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-9.2/lib64

export ros1_tracking=1

if [ "$ros1_tracking" -eq 1 ]; then
        cd ~
        cd ros_ws/src/centerNet-deep-sort/
        conda activate CenterNet
        source /opt/ros/melodic/setup.bash
        source /root/ros_ws/devel/setup.bash
        python tracking_yolov3_centernet_deepsort.py
fi
```

Figure D.4-1 Bashrc configuration (Docker image of tracking module)

D. 5. Install CARLA 0.9.5

CARLA requires Ubuntu 16.04 or later. First, install the build tools dependencies:

sudo apt-get update
sudo apt-get install wget software-properties-common
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-7 main"
sudo apt-get update
sudo apt-get install build-essential clang-7 lld-7 g++-7 cmake ninja-build libvulkan1 python python-pip python-dev python3-dev python3-pip libpng16-dev libtiff5-dev libjpeg-dev tzdata sed curl unzip autoconf libtool rsync
pip2 install --user setuptools
pip3 install --user setuptools

To avoid compatibility issues between Unreal Engine and the CARLA dependencies, the best configuration is to compile everything with the same compiler version and C++ runtime library (CARLA uses clang 6.0 and LLVM's libc++):

sudo update-alternatives --install /usr/bin/clang++ clang++ /usr/lib/llvm-7/bin/clang++ 170
sudo update-alternatives --install /usr/bin/clang clang /usr/lib/llvm-7/bin/clang 170

After configuring the tool dependencies, build Unreal Engine:

git clone --depth=1 -b 4.22 https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.22
cd ~/UnrealEngine_4.22
./Setup.sh && ./GenerateProjectFiles.sh && make

Finally, Build CARLA from GitHub repository:

https://github.com/carla-simulator/carla

git clone https://github.com/carla-simulator/carla
./Update.sh
export UE4_ROOT=~/UnrealEngine_4.22 (it can be also added to the ~/.bashrc file)
make launch # Compiles the simulator and launches Unreal Engine's Editor.
make PythonAPI # Compiles the PythonAPI module necessary for running the Python examples.
make package # Compiles everything and creates a packaged version able to run without UE4 editor.
make help # Print all available commands.

D. 6. Execution CARLA + SmartElderlyCar (SEC)

The main commands so as to perform a MOT using CARLA and the SmartElderlyCar project are

Shell 1 (Launch CARLA simulator on Desktop computer host)

cd ~/carla/Dist/0.9.5-96-g67cfd574/LinuxNoEditor
DISPLAY= ./CarlaUE4.sh /Game/Carla/Maps/Town03

Shell 2 (Run roscore; SEC Docker, *Desktop computer*)

./launch_docker_version SmartElderlyCar_project:last

Shell 3 (Run ROSBridge for CARLA and SEC communications; Host, *Desktop computer*)

roslaunch carla_ros_bridge carla_interface_map.launch

On the MSI computer, run the tracking module

./launch_docker_version CenterNet_DeepSORT:last
cd ~/ros_ws/src/centernet_deepsort_master
python3 tracking_centernet_deepsort.py

Make sure that *ROS_MASTER_URI* points to the Desktop computer IP address and *ROS_IP* points to the laptop IP (it can be checked by typing *ifconfig* on the Ubuntu terminal and saving the provided *ined addr*).

Shell 4 (Include dynamic objects in the CARLA world, such as pedestrians or cars; Host, *Desktop computer*)

python manual_control_pedestrian.py --filter=walker --point -1 73 2 --orientation 65
python manual_control_vehicle.py --filter=vehicle.tesla.model3 --point 244 95 2 --orientation 270

Shell 5 (Launch map_module to load Lanelets map; SEC Docker, *Desktop computer*)

roslaunch smart_elderly_car map_module.launch

Shell 6 (Launch navigation_module to perform local navigation; SEC Docker, *Desktop computer*)

roslaunch smart_elderly_car navigation_module.launch

Shell 7 (Launch Robograph to activate decision-making layer; SEC Docker, *Desktop computer*)

roslaunch smart_elderly_car robograph.launch

Appendix E: Specifications

This appendix details the main hardware and software used in this master thesis.

E.1. Hardware

- ASUS Intel Core i5-4210U
    - 1.70 GHz CPU
    - 8 GB DDR3 1333 MHz RAM
    - 500 GB HDD
    - NVIDIA GeForce 320 M (CUDA technology available)
- MSI GT62VR-7RE i7-7700HQ
    - 2.8 GHz CPU
    - 16 GB DDR4 2400 MHz RAM
    - 500 GB SSD
    - NVIDIA 1070 GTX (CUDA technology available)
- Desktop computer i7-8700
    - 3.2 GHz CPU
    - 32 GB DDR4 2400 MHz RAM
    - 500 GB SSD NVME
    - NVIDIA 2070 RTX (CUDA technology available)
- *Tabby Evo* Open Source Electric Car
    - Maximum speed: 100 km/h
    - Autonomy: 80 km
    - Power: 19 kW
    - Weight: 380 kg
- *Velodyne LiDAR Puck (VLP-16)*
    - Channels: 16
    - Maximum range: 100 m
    - Maximum number of points/s: 600,000
    - Accuracy: 3 cm
- StereoLabs ZED camera
    - Sensors: 2 CCD 4M pixels per sensor with large 2-micron pixels
    - Field of View: 90 º (Horizontal) x 60 º (Vertical)

- ❖ Maximum output resolution: 4416x1242
- ❖ Technology: Real-time depth-based visual odometry and SLAM

- Topcon Hiper Pro GPS

  - ❖ I/O ports: *2x serie* (RS232)
  - ❖ Output frequency: 20 Hz
  - ❖ Communication: Bluetooth 1.1
  - ❖ Search channels: 20 GPS L1+L2 (GD), GPS L1 + GLONASS (GG)
    20 GPS L1+L2+GLONASS (GGD)

E.2. Software

- Ubuntu 14.04.5 LTS (Trusty Tahr)
- Ubuntu 18.04.3 LTS (Bionic Beaver)
- Docker 19.03
- ROS Indigo Igloo
- ROS Melodic Morenia
- Point Cloud Library V1.7
- CARLA simulator 0.9.4
- ROS packages
- Microsoft Office 365 ProPlus

Appendix F: Budget

This appendix describes the theoretical cost of the whole project.

F. 1. Material cost

In this section, the cost of the different materials (software and hardware) are detailed (21 % VAT is included).

Table F.1-1 Material costs

Concept		Units	Unit cost [€]	Total cost [€]
Hardware	MSI Laptop	1	1360.00	1360.00
	Windows PC i5 1.7 GHz	1	450.00	450.00
	Desktop computer	1	2,365.00	2,365.00
	Velodyne LiDAR Puck (VLP-16)	1	7560.0	7560.0
	StereoLabs ZED camera	1	416.00	416.00
	Topcon GPS Hiper Pro	1	3,000.00	3,000.00
	Tabby Evo Open Source vehicle	1	20,250.00	20,250.00
Software	Docker 19.03	1	0.00	0.00
	CARLA simulator	1	0.00	0.00
	ROS Indigo Igloo	1	0.00	0.00
	ROS Melodic Morenia	1	0.00	0.00
	Point Cloud Library V1.7	1	0.00	0.00
	ROS packages	1	0.00	0.00
	Microsoft Office 365 ProPlus	1	0.00	0.00
Material total costs [€]				35,401.00 €

F. 2. Professional fees

In this point the different professional fees are calculated as gross incomes (not including VAT). They include all the professional activities related with the project.

Table F.2-1 Professional fees

Activity	Salary (€/month)	Time (months)	Total cost (€)
Engineering	1,250.00	3	3,750.00
Typing	1,000.00	1	1,000.00
Professional fees total costs [€]			4,750.00

F. 3. Total costs

Total costs have been calculated by adding the material total costs and professional fees total costs.

Table F.3-1 Total costs

Material total costs [€]	35,401.00
Professional fees total costs [€]	4,750.00
Transport [€]	260.00
Total cost [€]	40.411

References

[1] Parekh, Himani S., Darshak G. Thakore, and Udesang K. Jaliya. "A survey on object detection and tracking methods." International Journal of Innovative Research in Computer and Communication Engineering 2.2 (2014): 2970-2979

[2] WOJKE, Nicolai; BEWLEY, Alex; PAULUS, Dietrich. Simple online and realtime tracking with a deep association metric. En 2017 IEEE International Conference on Image Processing (ICIP). IEEE, 2017. p. 3645-3649

[3] CIAPARRONE, Gioele, et al. Deep Learning in Video Multi-Object Tracking: A Survey. arXiv preprint arXiv:1907.12740, 2019

[4] WOJKE, Nicolai; BEWLEY, Alex. Deep cosine metric learning for person re-identification. En 2018 IEEE winter conference on applications of computer vision (WACV). IEEE, 2018. p. 748-756

[5] ÅGREN, Sanna. Object tracking methods and their areas of application: A meta-analysis: A thorough review and summary of commonly used object tracking methods. 2017

[6] PATEL, Sandeep Kumar; MISHRA, Agya. Moving object tracking techniques: A critical review. Indian Journal of Computer Science and Engineering, 2013, vol. 4, no 2, p. 95-102

[7] PAREKH, Himani S.; THAKORE, Darshak G.; JALIYA, Udesang K. A survey on object detection and tracking methods. International Journal of Innovative Research in Computer and Communication Engineering, 2014, vol. 2, no 2, p. 2970-2979

[8] ATHANESIOUS, J.; SURESH, P. Implementation and comparison of kernel and silhouette based object tracking. International Journal of Advanced Research in Computer Engineering & Technology, 2013, p. 1298-1303

[9] JIA, Xu; LU, Huchuan; YANG, Ming-Hsuan. Visual tracking via adaptive structural local sparse appearance model. En 2012 IEEE Conference on computer vision and pattern recognition. IEEE, 2012. p. 1822-1829

[10] KWON, Junseok; LEE, Kyoung Mu. Visual tracking decomposition. En 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE, 2010. p. 1269-1276

[11] DEL PINO, Iván, et al. Low resolution lidar-based multi-object tracking for driving applications. En Iberian Robotics conference. Springer, Cham, 2017. p. 287-298

[12] REKLEITIS, Ioannis; BEDWANI, Jean-Luc; DUPUIS, Erick. Autonomous planetary exploration using LIDAR data. En 2009 IEEE International Conference on Robotics and Automation. IEEE, 2009. p. 3025-3030

[13] PETROVSKAYA, Anna; THRUN, Sebastian. Model based vehicle detection and tracking for autonomous urban driving. Autonomous Robots, 2009, vol. 26, no 2-3, p. 123-139

[14] TEICHMAN, Alex; LEVINSON, Jesse; THRUN, Sebastian. Towards 3D object recognition via classification of arbitrary object tracks. En 2011 IEEE International Conference on Robotics and Automation. IEEE, 2011. p. 4034-4041

[15] WANG, Dominic Zeng; POSNER, Ingmar. Voting for Voting in Online Point Cloud Object Detection. En Robotics: Science and Systems. 2015. p. 10.15607

[16] DEWAN, Ayush, et al. Motion-based detection and tracking in 3d lidar scans. En 2016 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2016. p. 4508-4513

[17] DE SILVA, Varuna; ROCHE, Jamie; KONDOZ, Ahmet. Fusion of LiDAR and camera sensor data for environment sensing in driverless vehicles. 2018

[18] CHO, Hyunggi, et al. A multi-sensor fusion system for moving object detection and tracking in urban driving environments. En 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2014. p. 1836-1843

[19] JALAL, Anand Singh; SINGH, Vrijendra. The state-of-the-art in visual object tracking. Informatica, 2012, vol. 36, no 3

[20] ROMERA, Eduardo, et al. Bridging the day and night domain gap for semantic segmentation. En 2019 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2019. p. 1312-1318

[21] FENG, Xiaoyu; MEI, Wei; HU, Dashuai. A review of visual tracking with deep learning. En 2016 2nd International Conference on Artificial Intelligence and Industrial Engineering (AIIE 2016). Atlantis Press, 2016

[22] ALBAWI, Saad; MOHAMMED, Tareq Abed; AL-ZAWI, Saad. Understanding of a convolutional neural network. En 2017 International Conference on Engineering and Technology (ICET). IEEE, 2017. p. 1-6

[23] SVOZIL, Daniel; KVASNICKA, Vladimir; POSPICHAL, Jiri. Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems, 1997, vol. 39, no 1, p. 43-62

[24] KARPATHY, Andrej; JOHNSON, Justin; FEI-FEI, Li. Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078, 2015

[25] HECHT-NIELSEN, Robert. Theory of the backpropagation neural network. En Neural networks for perception. Academic Press, 1992. p. 65-93

[26] MEMISEVIC, Roland, et al. Gated softmax classification. En Advances in neural information processing systems. 2010. p. 1603-1611

[27] BEWLEY, Alex, et al. Simple online and realtime tracking. En 2016 IEEE International Conference on Image Processing (ICIP). IEEE, 2016. p. 3464-3468

[28] KIM, Chanho, et al. Multiple hypothesis tracking revisited. En Proceedings of the IEEE International Conference on Computer Vision. 2015. p. 4696-4704

[29] HAMID REZATOFIGHI, Seyed, et al. Joint probabilistic data association revisited. En Proceedings of the IEEE international conference on computer vision. 2015. p. 3047-3055

[30] KALMAN, Rudolph Emil. A new approach to linear filtering and prediction problems. Journal of basic Engineering, 1960, vol. 82, no 1, p. 35-45

[31] DOSOVITSKIY, Alexey, et al. CARLA: An open urban driving simulator. arXiv preprint arXiv:1711.03938, 2017

[32] QUIGLEY, Morgan, et al. ROS: an open-source Robot Operating System. En ICRA workshop on open source software. 2009. p. 5

[33] TE chassis "Open Motors" https://www.openmotors.co/product/tabbyevo/

[34] P. VLP-16, "Velodyne lidar," http://velodynelidar.com/vlp-16.html

[35] ZED camera, "StereoLabs ZED", https://www.stereolabs.com/zed/#

[36] Hiperpro, "Topcon gps," http://www.topcon.com.sg/survey/hiperpro.html.

[37] BENDER, Philipp; ZIEGLER, Julius; STILLER, Christoph. Lanelets: Efficient map representation for autonomous driving. En 2014 IEEE Intelligent Vehicles Symposium Proceedings. IEEE, 2014. p. 420-425

[38] ROHMER, Eric; SINGH, Surya PN; FREESE, Marc. V-REP: A versatile and scalable robot simulation framework. En 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2013. p. 1321-1326

[39] KO, Nak Yong; SIMMONS, Reid G. The lane-curvature method for local obstacle avoidance. En Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No. 98CH36190). IEEE, 1998. p. 1615-1621

[40] FERNÁNDEZ, Joaquín Lopez, et al. Improving collision avoidance for mobile robots in partially known environments: the beam curvature method. Robotics and Autonomous Systems, 2004

[41] FERNÁNDEZ, Joaquín L., et al. Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph. En 2008 IEEE International Conference on Robotics and Automation. IEEE, 2008. p. 1372-1377

[42] Carlos Gómez-Huelamo, Luis M. Bergasa, Rafael Barea, Elena López-Guillén, Sandra Carrasco, Pablo Sánchez, "Simulating use cases for the UAH Autonomous Electric Car", in IEEE Conference on Intelligent Transportation Systems (ITSC), Auckland, New Zealand, October 2019. Accepted on July 2019

[43] D. S. Michal and L. Etzkorn, "A comparison of player/stage/gazebo and microsoft robotics developer studio," in Proceedings of the 49th Annual Southeast Regional Conference, pp. 60–66, ACM, 2011.

[44] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in Field and service robotics, pp. 621–635, Springer, 2018

[45] BRANDES, Ulrik. A faster algorithm for betweenness centrality. Journal of mathematical sociology, 2001, vol. 25, no 2, p. 163-177

[46] HELD, David, et al. Robust real-time tracking combining 3D shape, color, and motion. The International Journal of Robotics Research, 2016

[47] RUSU, Radu Bogdan; COUSINS, Steve. 3d is here: Point cloud library (pcl). En 2011 IEEE international conference on robotics and automation. IEEE, 2011. p. 1-4

[48] LI, Bi, et al. Learning to Update for Object Tracking With Recurrent Meta-Learner. IEEE Transactions on Image Processing, 2019, vol. 28, no 7, p. 3624-3635

[49] DUAN, Kaiwen, et al. CenterNet: Object Detection with Keypoint Triplets. arXiv preprint arXiv:1904.08189, 2019

[50] H. Law and J. Deng. Cornernet: Detecting objects as paired keypoints. In Proceedings of the European conference on computer vision, pages 734–750, 2018

[51] REDMON, Joseph; FARHADI, Ali. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018

[52] MILLS-TETTEY, G. Ayorkor; STENTZ, Anthony; DIAS, M. Bernardine. The dynamic hungarian algorithm for the assignment problem with changing costs. 2007

[53] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in NIPS, 2015

[54] REID, Donald. An algorithm for tracking multiple targets. IEEE transactions on Automatic Control, 1979, vol. 24, no 6, p. 843-854

[55] LIN, Tsung-Yi, et al. Microsoft coco: Common objects in context. En European conference on computer vision. Springer, Cham, 2014. p. 740-755

[56] LIU, Wei, et al. Ssd: Single shot multibox detector. En European conference on computer vision. Springer, Cham, 2016. p. 21-37

[57] L. Zheng, Z. Bie, Y. Sun, J. Wang, C. Su, S. Wang, and Q. Tian, "MARS: A video benchmark for large-scale person re-identification," in ECCV, 2016

[58] S. Zagoruyko and N. Komodakis, "Wide residual networks," in BMVC, 2016, pp. 1–12

[59] BEIS, Jeffrey S.; LOWE, David G. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. En cvpr. 1997. p. 1000

[60] MERKEL, Dirk. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014, vol. 2014, no 239, p. 2

[61] HELD, David; THRUN, Sebastian; SAVARESE, Silvio. Learning to track at 100 fps with deep regression networks. En European Conference on Computer Vision. Springer, Cham, 2016. p. 749-765

[62] JIA, Yangqing, et al. Caffe: Convolutional architecture for fast feature embedding. En Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014. p. 675-678

[63] ALVAR, Saeed Ranjbar; BAJIĆ, Ivan V. MV-YOLO: Motion vector-aided tracking by semantic object detection. En 2018 IEEE 20th International Workshop on Multimedia Signal Processing (MMSP). IEEE, 2018. p. 1-5

[64] NING, Guanghan, et al. Spatially supervised recurrent convolutional neural networks for visual object tracking. En 2017 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2017. p. 1-4

[65] ZHANG, Zizhao, et al. Mdnet: A semantically and visually interpretable medical image diagnosis network. En Proceedings of the IEEE conference on computer vision and pattern recognition. 2017. p. 6428-6436

[66] FARHADI, Daniel Gordon1 Ali; FOX, Dieter. Re 3: Real-Time Recurrent Regression Networks for Visual Tracking of Generic Objects. IEEE Robot. Autom. Lett, 2018, vol. 3, no 2, p. 788-795

[67] SANDERS, Andrew. An Introduction to Unreal Engine 4. AK Peters/CRC Press, 2016

[68] ROMERA, Eduardo, et al. Erfnet: Efficient residual factorized convnet for real-time semantic segmentation. IEEE Transactions on Intelligent Transportation Systems, 2017, vol. 19, no 1, p. 263-272

[69] GEIGER, Andreas, et al. Vision meets robotics: The KITTI dataset. The International Journal of Robotics Research, 2013, vol. 32, no 11, p. 1231-1237

[70] BERNARDIN, Keni; ELBS, Alexander; STIEFELHAGEN, Rainer. Multiple object tracking performance metrics and evaluation in a smart room environment. En Sixth IEEE International Workshop on Visual Surveillance, in conjunction with ECCV. 2006. p. 91

[71] ZHANG, Da, et al. Deep reinforcement learning for visual object tracking in videos. arXiv preprint arXiv:1701.08936, 2017

[72] BABENKO, Boris; YANG, Ming-Hsuan; BELONGIE, Serge. Robust object tracking with online multiple instance learning. IEEE transactions on pattern analysis and machine intelligence, 2010, vol. 33, no 8, p. 1619-1632

[73] KRISTAN, Matej, et al. The visual object tracking vot2015 challenge results. En Proceedings of the IEEE international conference on computer vision workshops. 2015. p. 1-23

[74] BERTINETTO, Luca, et al. Fully-convolutional siamese networks for object tracking. En European conference on computer vision. Springer, Cham, 2016. p. 850-865

[75] KAHOU, Samira Ebrahimi, et al. RATM: recurrent attentive tracking model. En 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). IEEE, 2017. p. 1613-1622

[76] GAN, Quan, et al. First step toward model-free, anonymous object tracking with recurrent neural networks. arXiv preprint arXiv:1511.06425, 2015

[77] REDMON, Joseph, et al. You only look once: Unified, real-time object detection. En Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 779-788

[78] SHARMA, Sarthak, et al. Beyond pixels: Leveraging geometry and shape cues for online multi-object tracking. En 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018. p. 3508-3515

[79] XIANG, Yu; ALAHI, Alexandre; SAVARESE, Silvio. Learning to track: Online multi-object tracking by decision making. En Proceedings of the IEEE international conference on computer vision. 2015. p. 4705-4713

[80] SCHEIDEGGER, Samuel, et al. Mono-camera 3d multi-object tracking using deep learning detections and pmbm filtering. En 2018 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2018. p. 433-440

[81] GÜNDÜZ, Gültekin; ACARMAN, Tankut. A lightweight online multiple object vehicle tracking method. En 2018 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2018. p. 427-432

[82] KARUNASEKERA, Hasith; WANG, Han; ZHANG, Handuo. Multiple Object Tracking With Attention to Appearance, Structure, Motion and Size. IEEE Access, 2019, vol. 7, p. 104423-104

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR

Universidad de Alcalá