Real-time Detection and Tracking of Moving Objects Using Deep Learning and Multi-threaded Kalman Filtering

A joint solution of 3D object detection and tracking for Autonomous Driving

Henrik Söderlund
Real-time Detection and Tracking of Moving Objects Using Deep Learning and Multi-threaded Kalman Filtering

A joint solution of 3D object detection and tracking for Autonomous Driving

Henrik Söderlund

Department of Electronics and Applied Physics
Umeå University

This thesis is submitted for the degree of
Master of Science in Electronics with specialization in Robotics and Control

June 2019
To my daughter and my fiancée.
Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original. Except for what is specified under Section 1.6 – Collaboration, and Acknowledgements, the thesis has not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work, except for what is described under Section 1.6 – Collaboration. I also declare that I have taken part of- and am following the IEEE Code of Ethics [1] in this work.

Henrik Söderlund
June 2019
Acknowledgements

First and foremost, I would like to thank my supervisor at Volvo Cars, Nivir Roy, for his confidence in me. I would also like to thank Sven Rönnbäck and Pedher Johansson for serving as members on my thesis committee. Their comments and guidance were very beneficial in the completion of this manuscript. Thank you, Leonid Freidovich, for your expertise and guidance during my studies. Special thanks go to my thesis partner, Mikel Broström, for sharing this period with me and for the interesting conversations we had about life.

Finally, I must express my very profound gratitude to my family for always being there for me, supporting me throughout my years of study and through the process of researching and writing this thesis. Thank you for giving me strength and encouragement in my lowest moments. This accomplishment would not have been possible without you. Thank you. I love you.

Henrik Söderlund, June 2019, Mölndal
Abstract

Perception for autonomous drive systems is the most essential function for safe and reliable driving. LiDAR sensors can be used for perception and are vying for being crowned as an essential element in this task. In this thesis, we present a novel real-time solution for detection and tracking of moving objects which utilizes deep learning based 3D object detection. Moreover, we present a joint solution which utilizes the predictability of Kalman Filters to infer object properties and semantics to the object detection algorithm, resulting in a closed loop of object detection and object tracking.

On one hand, we present YOLO++, a 3D object detection network on point clouds only. A network that expands YOLOv3, the latest contribution to standard real-time object detection for three-channel images. Our object detection solution is fast. It processes images at 20 frames per second. Our experiments on the KITTI benchmark suite show that we achieve state-of-the-art efficiency but with a mediocre accuracy for car detection, which is comparable to the result of Tiny-YOLOv3 on the COCO dataset. The main advantage with YOLO++ is that it allows for fast detection of objects with rotated bounding boxes, something which Tiny-YOLOv3 can not do. YOLO++ also performs regression of the bounding box in all directions, allowing for 3D bounding boxes to be extracted from a bird’s eye view perspective. On the other hand, we present a Multi-threaded Object Tracking (MTKF) solution for multiple object tracking. Each unique observation is associated to a thread with a novel concurrent data association process. Each of the threads contain an Extended Kalman Filter that is used for predicting and estimating an associated object’s state over time. Furthermore, a LiDAR odometry algorithm was used to obtain absolute information about the movement of objects, since the movement of objects are inherently relative to the sensor perceiving them. We obtain 33 state updates per second with an equal amount of threads to the number of cores in our main workstation.

Even if the joint solution has not been tested on a system with enough computational power, it is ready for deployment. Using YOLO++ in combination with MTKF, our real-time constraint of 10 frames per second is satisfied by a large margin. Finally, we show that our system can take advantage of the predicted semantic information from the Kalman Filters in order to enhance the inference process in our object detection architecture.
### Table of contents

List of figures ........................................... xiii
List of tables ............................................ xv
Nomenclature ............................................. xvii

#### 1 Introduction ........................................ 1
  1.1 Background ........................................... 2
  1.2 Ethics in Autonomous Driving ....................... 3
  1.3 Aim .................................................. 4
  1.4 Delimitation ......................................... 5
  1.5 Structure ............................................ 5
  1.6 Collaboration ........................................ 6

#### 2 Detection and Tracking of Moving Objects ......... 9
  2.1 Deep Learning ........................................ 9
     2.1.1 Types of Learning ............................... 9
     2.1.2 Artificial Neural Networks .................... 10
     2.1.3 Convolutional Neural Networks ................ 14
  2.2 Object Detection .................................... 18
     2.2.1 One-stage vs. Two-stage Detectors ............. 18
     2.2.2 2D Object Detection ............................ 19
     2.2.3 3D Object Detection ............................ 24
  2.3 Object Tracking ..................................... 27
     2.3.1 Model-based Object Tracking .................. 27
     2.3.2 Model-free Object Tracking .................... 33
     2.3.3 Point Cloud Registration ........................ 34
     2.3.4 LiDAR Odometry ................................ 41
  2.4 Conclusions .......................................... 42
# Table of contents

## 3 Proposed Detection and Tracking Solution

3.1 Object Detection ................................................. 45
   3.1.1 Data Pre-processing ........................................ 45
   3.1.2 YOLO++ ..................................................... 47
3.2 Object Tracking .................................................. 55
   3.2.1 Multi-threaded Object Tracking ............................. 55
   3.2.2 Data Association ........................................... 56
   3.2.3 Components of the Extended Kalman Filter ............... 57
   3.2.4 LiDAR Odometry ............................................ 59
3.3 Joint Solution .................................................... 60

## 4 Evaluation Strategy

4.1 Evaluation Metrics .............................................. 65
   4.1.1 Mean Average Precision .................................... 65
   4.1.2 CLEAR MOT & MT/PT/ML .................................. 67
4.2 The KITTI Dataset ............................................... 68
4.3 Experimental Setup .............................................. 70

## 5 Results

5.1 Detection Performance .......................................... 73
5.2 Tracking Performance ........................................... 77

## 6 Discussion

6.1 Result Analysis .................................................. 83
6.2 Problems ......................................................... 85
6.3 Social and Ethical Aspects ...................................... 86
6.4 Future Work ...................................................... 87
6.5 Conclusions ...................................................... 88

References .............................................................. 89
List of figures

2.1 Example of a multilayer neural network architecture ..................... 10
2.2 Convolution illustration example ............................................ 15
2.3 Max pooling illustration example ............................................ 17
2.4 Detection of objects in 2D and 3D .......................................... 19
2.5 SSD framework ........................................................................ 21
2.6 Bounding boxes with dimension priors and location prediction .......... 23
2.7 Feature Pyramid Network ......................................................... 24
2.8 Bird’s eye view detection overview .......................................... 26
2.9 The kinematic bicycle model .................................................... 32
2.10 A SLAM framework for dynamic environments .......................... 34
2.11 Point-to-point error between two surfaces ................................. 36
2.12 Point-to-plane error between two surfaces .................................. 37
2.13 Plane-to-plane error between two surfaces .................................. 40
3.1 A point cloud projected onto the bird’s eye view plane ................... 46
3.2 Anchor extraction ..................................................................... 54
3.3 The thread creation process of multi-threaded object tracking .......... 56
3.4 The data association process of object tracking ............................. 57
3.5 The state update process of the Extended Kalman Filters ................ 59
3.6 Overview of the LiDAR Odometry pipeline .................................. 59
3.7 The proposed joint solution of DATMO ...................................... 61
3.8 Illustration of the prediction probability grid map .......................... 63
4.1 The IoU metric illustrated ......................................................... 66
4.2 Problem with IoU and rotated bounding boxes ............................. 67
4.3 The KITTI Passat setup ........................................................... 69
4.4 Overview of connected ROS nodes .......................................... 71
5.1 Hyperparameter optimization done on a simplified dataset .............. 74
| 5.2  | The Object Detector detecting cars in an image                      | 75  |
| 5.3  | YOLO++ training loss                                              | 76  |
| 5.4  | The Object Tracker tracking multiple cars in a point cloud        | 77  |
| 5.5  | Result of tracking a car turning right                            | 78  |
| 5.6  | Results of multiple tracked objects                               | 79  |
| 5.7  | Erroneous tracking of a car traveling too fast                    | 80  |
| 5.8  | Velocity and innovation MSE from erroneous tracking               | 81  |
| 5.9  | Object Tracker run time performance                               | 81  |
List of tables

2.1 YOLOv1 architecture ........................................... 20
2.2 YOLOv2 architecture ........................................... 23

3.1 Feature extraction network .................................... 48
3.2 Detection pipeline 1 ............................................ 49
3.3 Detection pipeline 2 ............................................ 49
3.4 Observation-To-Thread Matrix ................................. 56

4.1 KITTI sensor setup ............................................. 68
4.2 KITTI 3D Object Detection Evaluation 2017 ............... 69
4.3 KITTI Object Tracking Evaluation 2012 .................. 70

5.1 Mean average precision results of hyperparameter experiments .......... 74
5.2 Performance comparison for BEV object detection ................ 74
## Nomenclature

**Acronyms / Abbreviations**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS</td>
<td>Autonomous Drive System</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>AP</td>
<td>Average Precision</td>
</tr>
<tr>
<td>BEV</td>
<td>Bird’s Eye View</td>
</tr>
<tr>
<td>BN</td>
<td>Batch Normalization</td>
</tr>
<tr>
<td>CM</td>
<td>Convolutional-Maxpooling</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DATMO</td>
<td>Detection and Tracking of Moving Objects</td>
</tr>
<tr>
<td>EKF</td>
<td>Extended Kalman Filter</td>
</tr>
<tr>
<td>FEN</td>
<td>Feature Extractor Network</td>
</tr>
<tr>
<td>FPN</td>
<td>Feature Pyramid Network</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>ICP</td>
<td>Iterative Closest Point</td>
</tr>
<tr>
<td>IoU</td>
<td>Intersection over Union</td>
</tr>
<tr>
<td>LiDAR</td>
<td>Light Detection and Ranging</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>LOAM</td>
<td>Lidar Odometry and Mapping</td>
</tr>
<tr>
<td>LReLU</td>
<td>Leaky Rectified Linear Unit</td>
</tr>
<tr>
<td>mAP</td>
<td>Mean Average Precision</td>
</tr>
<tr>
<td>ML</td>
<td>Mostly Lost</td>
</tr>
<tr>
<td>MOT</td>
<td>Multiple Objects Tracking</td>
</tr>
<tr>
<td>MOTA</td>
<td>Multiple Object Tracking Accuracy</td>
</tr>
<tr>
<td>MOTP</td>
<td>Multiple Object Tracking Precision</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>MT</td>
<td>Mostly Tracked</td>
</tr>
<tr>
<td>MTKF</td>
<td>Multi-threaded Kalman Filtering</td>
</tr>
<tr>
<td>NMS</td>
<td>Non-Maximum Suppression</td>
</tr>
<tr>
<td>OTM</td>
<td>Observation-to-Thread Matrix</td>
</tr>
<tr>
<td>PT</td>
<td>Partially Tracked</td>
</tr>
<tr>
<td>RANSAC</td>
<td>Random Sample Consensus</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit</td>
</tr>
<tr>
<td>ROS</td>
<td>Robotics Operating System</td>
</tr>
<tr>
<td>SAE</td>
<td>Society of Automotive Engineering</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous Localization and Mapping</td>
</tr>
<tr>
<td>SSD</td>
<td>Single Shot Detector</td>
</tr>
<tr>
<td>YOLO</td>
<td>You Only Look Once</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Among the many capabilities that an Autonomous Drive System (ADS) should have, perception of the environment is one of the most fundamental requirements. In fact, understanding the scene around the ADS is the first step towards achieving full autonomy. The perception starts in the sensors, which provides raw data for the ADS to interpret and extract contextual information from, giving meaning to the data. In a perception system for an autonomous vehicle, two main tasks can be identified: accurate Simultaneous Localization and Mapping (SLAM), and the Detection and Tracking of Moving Objects (DATMO) [2].

An example scenario in which both SLAM and DATMO are needed may be if a self-driving car is trying to cross an intersection in heavy traffic. In this particular scenario the autonomous vehicle needs to detect individual moving objects in its vicinity such as cars, pedestrians, and bicyclists. However, in order to navigate through the intersection efficiently and safely the vehicle must also predict the objects’ individual movement over time. The understanding of the dynamic nature of the environment offers three main advantages for an autonomous vehicle [3]:

1. Removing dynamic objects from the internal map can help to improve the estimation accuracy of the pose of the vehicle.

2. Predicting the motion of moving objects facilitates safe motion and path planning in order to prevent accidents from occurring.

3. Inferring semantic information of a detected object between different states can help the object detection process.

Because of the increasing popularity of vehicle autonomy in the car industry [4], research within SLAM and DATMO has gained a lot of attention in the last decade [5–13, 2, 3]. But all this research would not be possible without the recent breakthroughs that have unleashed
the long-awaited arrival of artificial intelligence, including a heavy increase in computing power, increasing data quantities to work on and refined algorithms.

1.1 Background

The Society of Automotive Engineering (SAE) have defined five levels in the evolution of autonomous driving, here each level describes the extent to which a car takes over tasks and responsibilities from its driver [14]:

1. Driver Assistance: drive assistance systems support the driver but do not take control.

2. Partly Automated Driving: system can also take control but the driver is the main responsible.

3. Highly Automated Driving: system can take control under certain situations for extended periods of time.

4. Fully Automated Driving: the vehicle drives independently most of the time but the driver must remain able to drive.

5. Full Automation: the vehicle assumes all driving functions.

Some of the most advanced self-driving vehicles in existence today are in the fourth stage [15]. This means that they are fully autonomous but just under certain conditions such that they are constrained to drive in pre-determined areas. In order to reach level 5 autonomy, machine vision capabilities and related technology play an important role in not only the safety of autonomous vehicles, but in their ability to account for unexpected variables while driving - a key milestone for autonomous vehicles to achieve [16].

Perception for autonomous drive systems is the most essential function for safe and reliable driving. LiDAR sensors can be used for perception and are vying for being crowned as an essential element in this task. They deliver large sets of 3D measurements that describe the environment, which makes their interpretation of huge interest in the Autonomous Drive System (ADS) community. Because it is impossible to handle all the individual outcomes explicitly in this type of data, the field of Deep Learning has become the main workhorse for this task. A fundamental sub-problem to solve in ADS, where point clouds are of great use, is to accurately and effectively detect, localize and track moving and potentially exposed objects, such as pedestrians, bicyclists, cars, etc., in highly dynamic environments. This is crucial for ADSs as their reasoning is based on this fundamental ground truth. If the
perception is not reliable the decisions can not be either. If the ADS can predict where a moving object is headed (and at which velocity) in relation to itself, it can self-adjust in real-time and prevent severe accidents from happening.

1.2 Ethics in Autonomous Driving

Self-driving cars promise to deliver a number of benefits to society, e.g. road accident prevention, optimal fuel usage, comfort, and convenience [17]. However, one also has to take the ethical complications into consideration. It is up to the engineer of a certain work to follow and uphold a set of socially acceptable ethical values [18]. Roboethics [19] is an applied ethics which should develop tools that can be shared and accepted by different social groups and beliefs. It focuses on the ethics of the robots’ designers, manufacturers and users instead of the actual robots. The purpose of Roboethics, applied for autonomous vehicles, is to solve the problem of moral uncertainty. How should autonomous vehicles be programmed to act when the person, who authorizes the choice of ethics, lacks the moral high ground? There have to be predefined and universally accepted ethics settings, which solve complex ethical problems [20]. An example of a problem, which requires an ethics setting, is the "helmet problem":

An autonomous car is facing an imminent crash. It could select one of two targets to swerve into: either a motorcyclist who is wearing a helmet or a motorcyclist who is not. What’s the right way to program the car? — [21]

The helmet problem raises a typical ethical question, which requires making a value judgment in order to answer it. One way to answer this problem could be to use an ethics setting that values minimizing overall harm, which would lead to the car swerving into the motorcyclist who is wearing a helmet since that rider has a higher chance of survival. Another setting could be to value responsible behavior, which argues that the car should swerve into the helmet-less motorcyclist, since it is not a responsible behavior to choose not to wear a helmet when motorcycling [20].

The helmet problem is similar to the classical "trolley problem", where there is a binary decision to make based on a particular value judgment. The problem is that the solution to which ethics setting to choose is not clear and the choice may vary throughout different social groups and beliefs.

Awad et al. [22] investigated ethical values in a recent study, where the authors invited people from all over the World to answer a questionnaire about which side they would pick given a certain case regarding the trolley problem. This investigation came to be called
the Moral Machine experiment. The authors identified three strong preferences that can be served as building blocks for Roboethics in the future [22]:

- The preference of sparing human lives compared to other life.
- The preference of sparing more lives instead of less.
- The preference of sparing young lives instead of old.

Something that also can be concluded is that no matter the preference, a life will be harmed (given this seemingly unrealistic situation) and the ADS may have conflicting ethics about the matter. It is thus up to us humans and the engineers of ADSs to define the ethics and be held responsible for them [19, 18].

1.3 Aim

The aim is to develop a novel solution for simultaneous real-time detection and tracking of moving objects, based exclusively on LiDAR data. Object detection will be done on objects that are capable of moving independently in the environment, such as cars, pedestrians or bicyclists. The detected objects will be tracked in such a way that reliable predictions (in such a way that the tracking does not deteriorate due to sudden changes in motion) of future states of these objects can be determined.

In order to validate the performance of the proposed network architecture for a 3D object detection in terms of speed and accuracy, a comparison will be made against state-of-the-art methods using the mean Average Precision (mAP) metric. The comparison will be made based both on a bird’s eye perspective (2D accuracy) and in the full 3D perspective (3D accuracy). Furthermore, the performance of Multiple Objects Tracking (MOT) should be investigated with respect to run-time performance. The CLEAR MOT (Classification of Events, Activities and Relationship for MOT) and MT/PT/ML (Mostly Tracked / Partly Tracked / Mostly Lost) metrics should be used for evaluation of the tracking method against the state-of-the-art. The idea is to investigate the performance of a joint detection and tracking of moving objects. Is it viable for safe and precise motion planning in real-time? Can the semantic information be inferred among states to aid the object detection process using object tracking?
1.4 Delimitation

This research focuses on the real-time aspect of object detection and tracking, and how the two subsystems can work jointly, aiding each other for increased performance. The project is limited to the use of LiDAR sensors for perception. The study and implementations will be based around this delimitation.

The object detection will be based on a point cloud projection solution and will be limited to detect objects within the categories Car, Cyclist and Pedestrian. The training method will be based on supervised learning and the input data and labels will be provided by the KITTI dataset [23]. The object tracking method will be developed to run on a GPU. The object tracking will be based on a model-based approach for tracking objects within the categories Car and Cyclist, while for objects within the category Pedestrian, a model-free approach will be used. This is because cars and bicycles are constrained in their movement, making it more predictable and thus viable for modelling. Pedestrians, however, are not constrained in their movement (except for when taking obstacles in the environment into consideration) and are thus not very easy to model. The object tracking method will be developed to run on a multi-core CPU.

A simulation environment will be set up using Robotics Operating System (ROS) to connect the subsystems and to achieve real-time testing and visualization capabilities. Real-time is defined as a minimum of 10 frames per second in this context as the LiDAR used to produce the data-set spin at 600 rpm which produces a complete point cloud every 100ms [24].

1.5 Structure

The thesis is constructed in the following manner: Chapter 2 covers the theory and related works gathered during the prestudy of the project. We start with Deep Learning, then we go through Object Detection followed by Object Tracking, and finish the chapter with decisive decisions about how the project would be connected to the related works. In Chapter 3 we present our proposed detection and tracking solution, where we first go through the object detection method and then the object tracking method. We finish the chapter by presenting our proposed joint solution, which connects the methodology to the problem statement that can be found under Aim. Chapter 4 covers the chosen evaluation strategy, including an explanation of the evaluation metrics used, the dataset that we base our solution around, and the experimental setup for performing the evaluation. Chapter 5 covers the results of the project and the evaluation, and Chapter 6 covers the discussion around the results and the
methodology. Chapter 6 also reflects upon the problem statement and covers a more in-depth discussion around the social and ethical aspects.

## 1.6 Collaboration

This master thesis is part of a collaborative project with Volvo Cars at the Department of Autonomous Vehicle Perception. The master thesis was done in pair with another student, Mikel Broström from the Department of Computing Science, Umeå University [25]. The thesis was written in the same document, but on different parts. Though, some parts have been written together. The thesis comes in two versions – this one and [25] – which are identical (or almost identical) in the text, except for Chapter 6, Discussion. The main reason for this is because of the differences in requirements for two degrees.

Henrik Söderlund’s contributions to this thesis are focused on – but not limited to – the object tracking part of the joint object detection and object tracking solution. These contributions are listed below:

- Chapter 1 - Introduction, excluding Background.
- Theory, related works and conclusions of Object Tracking in Chapter 2.
- Bounding box predictions in Chapter 3, Section 3.1.2.
- Choice of anchors in Chapter 3, Section 3.1.2.
- Object Tracking in Chapter 3, Section 3.2.
- Tables in Chapter 3, Section 3.1.
- Joint Solution in Chapter 3, Section 3.3.
- Modified IoU in Chapter 4, Section 4.1.1.
- Sections 4.1.2, 4.2 and 4.3 (including the figure) in Chapter 4.
- Hyperparameter Optimization in Chapter 5, Section 5.1.
- Results in Chapter 5, Section 5.2.

While Henrik Söderlund’s contributions have been listed above, it is difficult to pinpoint the entirety of Henrik Söderlund’s contributions to this thesis since the thesis work has been performed in unison with the other part, Mikel Broström. Thus, minor contributions may be found in all parts of this thesis, for both individuals involved.
The contributions of Mikel Broström are focused on – but not limited to – the object detection. These contributions are listed below:

- Background in Chapter 1.
- Deep Learning, in Chapter 2, Section 2.1.
- Theory, related works and conclusions of Object Detection in Chapter 2.
- Object Detection in Chapter 3, Section 3.1, excluding Bounding box predictions and Choice of anchors.
- Figures 3.3 to 3.5 in Chapter 3, Section 3.2.
- Sections 4.1.1 (excluding Modified IoU) and 4.3 (excluding the figure) in Chapter 4.
- Results in Chapter 5, Section 5.1, excluding Hyperparameter Optimization.
- Analysis about Figure 5.9 in Chapter 5, Section 5.2.

It is important to note that Chapters 1 to 5 are identical for both this thesis and [25]. The differences appear in Chapter 6, the discussion, whereas in this particular version, only the discussion of Henrik Söderlund is presented and is not done in collaboration with Mikel Broström. Though, similarities in the discussion of both versions may appear since the results are the same.
Chapter 2

Detection and Tracking of Moving Objects

To reach full autonomy, autonomous vehicles will have to be able to operate in scenarios, which are difficult to handle, such as crowded streets or heavy traffic. When solving problems such as self-localization and mapping, the environment can not be assumed to be static and the ADS will have to deal with the dynamic aspects of the environment [2]. Detection and tracking of moving objects is crucial for safe and intelligent navigation in dynamic environments. Another important aspect is that object tracking can help the inference of semantic information of an object among states [3].

2.1 Deep Learning

Deep learning is one of the machine learning methods based on feature learning; techniques that allows a system to automatically comprehend the representations needed for detection tasks from training data [26].

2.1.1 Types of Learning

There are three types of learning [27]: supervised, unsupervised and semi-supervised.

In supervised learning the system learns a function that maps an input to an output based on an ordered set of tuples \( \mathcal{X} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) where \( x_i \) is an input instance and \( y_i \) is its corresponding ground truth [26]. The goal in supervised learning is to minimize the inconsistency between a predicted value and its actual output [28]. When the task is to approximate a mapping function \( f \) from an input variable \( x \) to a discrete
Detection and Tracking of Moving Objects

output variable \( y \) it is called classification. However, when the task is to approximate \( f \) to a continuous output variable \( y \) it is called regression [28].

In unsupervised learning the goal is to learn relationships among elements in a data set \( D = \{ x_1, x_2, ..., x_n \} \) and classify the raw data without relying on a ground truth. Since it is not clear which patterns should be learned there is no obvious error metric which leads to search indirect hidden structures, patterns or features in the data [28].

Semi-supervised learning combine both of the previous approaches by typically making use of a small amount of labeled data with a large amount of unlabeled data [28].

2.1.2 Artificial Neural Networks

A neural network is a supervised learning method based on circuits of perceptrons that exchange messages between each other. A perceptron is a function that maps the dot product of a weight vector \( w \in \mathbb{R}^L \) and its corresponding input vector \( x \in \mathbb{R}^L \) plus a bias to an output value \( y_j \) [29]:

\[
y_j = f \left( \sum_{i=1}^{L} w_{ij} x_i + b_j \right), \quad j = \{1, 2, \ldots, M\},
\]

where \( f : \mathbb{R} \rightarrow \mathbb{R} \) is an activation function. Perceptrons are grouped in layers as can be seen in Figure 2.1.

![Figure 2.1 Example of a multilayer neural network architecture](image-url)
2.1 Deep Learning

There are three kind of layers [29]: the layer where the input data enters the system is called input layer, the output layer is responsible of producing the end result and every layer in between the input and output layers are called hidden layers. A layer where all the nodes are connected to all perceptrons in the next layer is said to be fully connected. The output $O_l \in \mathbb{R}^M$ of an arbitrary layer $l$ is computed as [29]

$$O_l = f^l(w^l x + b^l).$$

The output of the first layer becomes the input to the second layer, the second to the third and so on successively. A hidden layer $l$ with $N$ perceptrons and $M$ input values can be defined as a function $\mathbb{R}^M \to \mathbb{R}^N$ where $N$ is the number of perceptrons in the layer and $M$ is the number of inputs. A neural network with $n$ layers can be seen as a series of nested functions where the output of the first layer becomes the input to the second, the second to the third and so on successively. This can be described mathematically as

$$O = f^n(w^n \ldots (f^2(w^2 f^1(w^1 x + b^1)) + b^2) \ldots ) + b^n.$$

In its simplest form, a neural network can perform binary classification with a single perceptron, but increasing the amount of perceptrons and constructing the network in specific architectures, they can be universal approximators [30] to almost any continuous set function making them suitable for different machine learning tasks. The term ANN is used to designate all types of neural networks even though there exist several different types of them: Modular Neural Networks, Convolutional Neural Networks, Recurrent Neural Networks, etc [31].

**Back-propagation**

The learning problem for a neural network is the search process of a set of weights $w$ that minimizes a loss function $L(X, w)$ for a set of input-output pairs $X$. A loss function calculates the difference between a predicted output $\hat{y}_i$ and its actual value $y_i$. A classic error function for back-propagation is the mean squared error [32],

$$L(X, w) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i(X, w) - y_i)^2,$$

where $y_i$ is the target value for an input pair $(x_i, y_i)$ and $\hat{y}_i$ is the computed output of the network on input $x_i$. Other error functions can be used but its convenient mathematical properties make it a good choice for generalized learning methods [32].
Back-propagation means that the calculation of the gradient proceeds backwards through the network, with the gradient of the weight in the last layer being calculated first, then the penultimate and so on. The computations of the gradient from one layer are reused in the computations of the preceding layer allowing for efficient computation of the gradient at each layer compared to the naive approach of calculating each layer separately [32].

Hence, back-propagation attempts to minimize the chosen loss function $L$ with respect to neural network’s weights by calculating, for each weight $w_{ij}^k$, the value of $\frac{\delta L}{\delta w_{ij}^k}$. This derivative can be calculated with respect to individual input-output pairs combining them at the end [32]:

$$\frac{\delta L(X, w)}{\delta w_{ij}^k} = \sum_{d=1}^{N} \frac{\delta L_d}{\delta w_{ij}^k}.$$ 

Finally the weights can be updated according to the learning rate $\alpha$ and the total gradient [32]:

$$\Delta w_{ij}^k = -\alpha \frac{\delta L(X, w)}{\delta w_{ij}^k}.$$ 

**Activation Function**

The activation function $f(x)$ defines the output of a perceptron given its input. In most applications this function is non-linear because otherwise the output to the neural network would be a linear function which would only be suitable for linear classification/regression problems [29]. Moreover, most of the times neural networks want to compute something more complicated than that. This is especially relevant in Deep Learning approaches when the goal is to make sense of something very complex with high dimensionality as in pictures.

An important feature that needs to be considered is that it must be differentiable to be able to perform back-propagation optimization for gradient error calculations. There are several activation functions but the use of each of them depend heavily on the goal of each layer in the ANN as they have different properties. Some common activation functions are presented below [29]:

- **Sigmoid**, $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$, is a non-linear activation function that ranges from 0 to 1 with an S-shaped transient. This activation function can in some cases, lead to a vanishingly small gradient, effectively preventing the weight from changing its value and hence learning. Another problem that the sigmoid functions has is that the function is not symmetric around the origin which leads to positive values being forwarded but in some cases this is undesirable.
• **Tanh**, \( f(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \), is a non-linear activation function that ranges from \(-1\) to \(1\) and has a transient similar to the sigmoid function. The symmetry problem can be solved by this activation function. However it still has the vanishing gradient problem.

• **ReLU**, \( f(x) = \text{ReLU}(x) = \max(0, x) \), is a conditional linear activation function. The main advantage of ReLU (Rectified Linear Unit) is that it does not activate all the perceptrons at the same time as it returns the value zero for negative inputs. This makes the network sparse and efficient. However, the vanishing problem still persists as for negative values the gradient is zero, preventing the network from getting updated during back propagation.

• **Leaky ReLU**, \( f(x) = \text{LReLU}(x) = \max(\alpha x, x) \), \( \alpha \leq 1 \), is an improved version of ReLU. It solves the vanishing gradient problem by inserting a small linear component for negative values.

• **Softmax**, \( f(x) = \sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{M} e^{x_k}} \), for \( j = 1, 2, \ldots, M \) and \( x = (x_1, \ldots, x_M) \in \mathbb{R}^M \), is a type of sigmoid function used in classification tasks. It squeezes the outputs for each variable in the feature vector \( x \) between 0 and 1 dividing it by the sum of all variables. This makes it so that the sum of all variables in \( x \) will result in 1 after being run through the softmax activation function. This activation function is ideally used in the output layer of the classifier in order to obtain probabilities to define the class of each input.

### Batch Normalization

Deep neural networks are challenging to train. One usual reason is the distribution change of the inputs to layers deep in the network when weights are updated, known as internal covariate shift [33].

In the model update process, layers get updated backwards from the output to the input assuming that weights in the layer prior to the current layer are fixed causing the model to forever chase a moving target. This slows down the training by the need of using lower learning rates and careful parameter initialization. In order to solve this problem, **batch normalization** can be used. A technique that coordinates the update of multiple layers in the model by re-scaling the data to have zero mean and a standard deviation equal to one. It has the effect of stabilizing the learning process and dramatically reduce the training epochs required [33].
2.1.3 Convolutional Neural Networks

A CNN is a special case of neural networks described above. The design of a CNN is motivated by the functioning of the visual cortex of the brain, a part of the cerebral cortex which processes visual information [34]. From Hubel and Wiesel’s work on animals’ visual cortex [35], we know it contains a complex arrangement of cells, responsible for detecting light in small, overlapping sub-regions of the visual field called receptive fields. These cells act as local filters where the more complex the cell the larger its receptive fields is.

The animal visual cortex being the most powerful visual processing system in existence; it seems natural to emulate its behavior. Many neurally-inspired models can be found in the literature [36–38] but in all cases they consist of three fundamental layers that are always present: convolutional layers, subsampling layers and fully connected layers.

The Convolutional Layer

The convolutional layer is the core building block of CNNs. This type of layer consists of a set of filters with learnable parameters that are used to extract features from input data. They can be seen as the weights and biases of a CNN. The layers are built up so that the first layer detects a set of low-dimensional patterns in the input such as edges, blobs of color, etc., the second layer detects patterns of patterns, and so on [34]. The convolutional layer learns features in the same way as a multi-layer perceptron network (or ANN) – through back-propagation.

A convolution is done by sliding a kernel with fixed size over the input matrix. The elements that fall inside of the kernel at each step are combined through matrix multiplication of the kernel and the region in the input matrix that the kernel overlaps. There are other parameters that may be used as well; the zero-padding, which adds zeros around the input matrix in order for the input matrix size to be preserved (since a convolution reduces the dimension of the input matrix), and the stride, which determines how many elements the kernel should jump over between steps. The bigger the stride the smaller the output volume spatially. An important parameter to specify for a convolutional layer is the number of filters, which determines the depth of the convolutional layer. Each filter learns to look for different visual features in the input. The convolutional layer accepts an input of size $W_1 \times H_1 \times D_1$. It requires four parameters: the number of filters $K$, the kernel size $F$, the stride $S$, and the zero-padding $P$. The layer produces an output of size $W_2 \times H_2 \times D_2$ where [29] (see Figure 2.2 for a graphical representation of this):
\[ W_2 = \frac{(W_1 - F + 2P)}{S} + 1, \]  
\[ H_2 = H_2 = \frac{(H_1 - F + 2P)}{S} + 1, \]  
\[ D_2 = K. \]  

Figure 2.2 In this example, the input volume of size \([W_1 \times H_1 \times D_1]\) is convolved with a \(k \times k \times K\) kernel obtaining an output volume \([W_2 \times H_2 \times K]\).

As the kernel is slid over the input volume it produces an activation map that gives the responses of that kernel at every spatial position. CNNs learn kernels that activate when they see some type of visual feature such as an edge or line with some specific orientation on the first layer, and eventually higher-level patterns on deeper layers of the network. Each of the filters in each convolutional layer with its respective number of kernels produce a separate activation map. Stacking these activation maps along the depth dimension lead to that deeper layers in the network can perform more complex associations. There are two types of convolution [29]:

- **2D Convolution**: In 2D CNNs, convolution is performed to extract features from 2D space only. Formally, the value of an unit at \((x, y)\) in the \(i\)-th layer in the \(j\)-th feature map in, denoted as \(v_{ij}^{xy}\), is given by

\[
v_{ij}^{xy} = f\left(b_{ij} + \sum_m \sum_{p=0}^{P_i-1} \sum_{q=0}^{Q_i-1} w_{ijm}^{pq} v_{(i-1)m}^{(x+p)(y+q)}\right),
\]
where \( f \) is an activation function, \( b_{ij} \) is the bias for the feature map, \( m \) is the number of filters in the \((i-1)\)th layer, \( w_{ijm}^{pq} \) is the value at the position \((p, q)\) of the kernel connected to the \(k\)th feature map, and \( P_i \) and \( Q_i \) are the height and width of the kernel, respectively.

- **3D Convolution**: When the same concept is applied to spatial locations in 3D, the previous equation can be expanded to

\[
v_{ij}^{xyz} = f \left( b_{ij} + \sum_{m} \sum_{p=0}^{P_i-1} \sum_{q=0}^{Q_i-1} \sum_{r=0}^{R_i-1} w_{ijm}^{pqr} v_{(i-1)m}^{(x+p)(y+q)(z+r)} \right),
\]

where \( R_i \) is the size of the 3D kernel along the third spatial dimension and \( w_{ijm}^{pqr} \) is the \((p, q, r)\)th value of the kernel connected to the \(m\)th feature map in the previous layer.

**Subsampling Layer**

Subsampling (pooling) layers are mainly used for two reasons: to progressively reduce the spatial size from one layer to another in order to reduce the amount of parameters, and to make features robust against noise. The pooling layers operate independently on every depth slice over equally sized non-overlapping region using the \textit{max} operation. The most common is a \(2 \times 2\) max pooling layer applied with a stride of two which discards 75\% of the activations. An example of this operation can be seen in Figure 2.3. The subsampling layer accepts an input of size \(W_1 \times H_1 \times D_1\). It requires two parameters: the spatial extent, i.e., kernel size \(F\), and the stride \(S\). The layer produces an output of size \(W_2 \times H_2 \times D_2\) where [29]

\[
W_2 = (W_1 - F) / S + 1, \\
H_2 = (H_1 - F) / S + 1, \\
D_2 = D_1.
\]

**Fully Connected Layer**

While the high-level feature output from the convolutional layers can be flattened and connected to the output layer, adding a fully connected layer is a way of learning non-linear combinations of these features. Fully connected layers are hence often used as the final layers of a CNN. They sum a weighting of the previous layer of features, indicating the precise parameter inputs to determine a specific output target result [29].
2.1 Deep Learning

Figure 2.3 Left: in this example, the input volume of size $[224 \times 224 \times 3]$ is pooled with a filter of size 2, stride 2 into an output volume $[112 \times 112 \times 3]$. Right: the most common pooling operation: max with $2 \times 2$ filters and stride 2.

Why Convolutional Neural Networks?

While neural networks have been around for the past 50 years, there are several reasons why CNNs have become the main workhorse for object detection and classification [39]. Some of their main advantages are [34]:

- **CNNs have fewer memory requirements**: Regular neural networks do not scale well for inputs like multi-channel images. A single fully connected perceptron in a first hidden layer of a regular neural network, for a $200 \times 200$ image with three color channels, would have $200 \cdot 200 \cdot 3 = 120000$ weights. Moreover, several of such perceptrons would be needed in order to perform any relevant type of learning. Clearly, this is ineffective memory- and computation-wise. CNNs take advantage of the fact that the input data is can be interpreted as a multi-channel image and performs operations that reduce the dimensionality of this input while preserving features that may be extracted for classification within the input image.

- **They are easier and better to train**: Because of the lower architectural complexity of a CNN compared to a standard neural network, its training time is proportionately decreased. Moreover, because of a lower number of parameters, the susceptibility to
noise is lower during the training process. Hence, the performance of a standard neural network will always be poorer than a CNN for image classification purposes.

- **They are rugged to shifts and distortion in the input:** CNNs are shift invariant since the same weight configuration is used across space. Although this could be achieved by a standard neural network it would need multiple units with identical weight parameters at different locations of the input, increasing the memory and training time burdens. CNNs are also rugged to distortions such as changes in shape, partial occlusions, horizontal and vertical shifts, etc.

However, CNNs are only suitable for generalized object detection tasks as the precise spatial relationships between higher-level features are lost in the consecutive down-sampling process [40].

### 2.2 Object Detection

Computer vision is an interdisciplinary field that has been gaining a lot of interest in recent years with self-driving cars in the centre stage [41].

In the early stages of object detection, in 2D as well as 3D (see Figure 2.4), most of the state of the art approaches consisted of extraction of hand-engineered features, which were fed to a standard classifier such as an SVM. However, this kind of approach is at the present time outperformed by Deep Learning approaches, where the classifiers are trained from the data using CNNs. While this method is conceptually simple to understand, it is unclear what architecture and feature representation could lead to good object detection performance as the behavior of the CNN learning process is difficult to anticipate [29].

#### 2.2.1 One-stage vs. Two-stage Detectors

Two-stage object detectors first propose a set of regions of interest by a selective search algorithm [42] or a region proposal network. Then a classifier only processes the region candidates. Examples of this type of 2D object detectors are the R-CNN family [43–45]. However, its fastest version to date, Faster R-CNN, obtains an inference time of 198 ms on a K40 GPU [45], making it far from being a viable real-time object detection solution.

One-stage detectors, on the other hand, skips the region proposal stage and runs detection directly over a high number of possible locations. This makes it faster and simpler than two-stage detectors but with an inevitable accuracy loss. An inevitable trade-off necessary to achieve real-time capabilities thus far [46].
2.2 Object Detection

A key aspect in computer vision is object detection which aids ADS in the process of pose estimation, path tracking algorithms, mapping, etc. The detection problem expands the classic classification problem where the goal is to label an image with the drawing of a bounding box around the object of interest to delimit it within the image.

**YOLO**

The YOLO [47] model is the very first attempt at building a fast real-time object detector (see Table 2.1 for architecture details). It looks at the complete image just once instead of using regions to localize objects within the image [47]. In YOLO a single convolutional network predicts the class probabilities over a limited set of bounding boxes allowing for direct end-to-end optimization and fast inference speed. It takes an image and split it into an $s \times s$ grid. For each of the grid cells, $B$ bounding boxes are predicted for which the CNN calculates: 1) The coordinates defined by 4 values: the center of the bounding box in the x- and y axes and the width and height of the bounding box. All of the variables are normalized by the image width and height, which makes each variable range between $(0, 1]$. 2) A confidence score that indicates the probability that the cell contains an object defined by $P(Object) \cdot \text{IoU}_{truth}^{pred}$. 3) The C class probabilities defined by $P(C_i \mid Object)$.

The final layer of the CNN is then modified to output a prediction tensor of size $s \times s \times B \cdot (5 + C)$. This is done using two fully connected layers over the entire final feature map. Only
Detection and Tracking of Moving Objects

The bounding boxes that contain class probabilities above a certain threshold are selected and further used to locate objects within the image. This is combined with Non-Max Suppression (NMS) in order to eliminate duplicated selections [47].

Table 2.1 YOLOv1 architecture.

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Activation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>64</td>
<td>7 × 7 / 2</td>
<td>LReLU &amp; BN</td>
<td>224 × 224 × 64</td>
</tr>
<tr>
<td>Maxpool</td>
<td>128</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 128</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 256</td>
</tr>
<tr>
<td>Maxpool</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3 / 2</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Connected</td>
<td>112</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 64</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Connected</td>
<td>7 × 7 × 30</td>
<td></td>
<td></td>
<td>7 × 7 × 30</td>
</tr>
</tbody>
</table>

The multi-part loss function is dependent on the location $x, y$ and size $w, h$ of bounding boxes together with the objectness $p(c)$ (or confidence) and class probabilities $C$. Two gain factors ($\lambda_{\text{coord}}$ and $\lambda_{\text{noobj}}$) are used to control the contribution of each part to the total loss. The function used to optimize during the training is

$$
\begin{align*}
\lambda_{\text{coord}} \sum_{i=0}^{B} \sum_{j=0}^{B} 1_{ij}^{obj} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{\text{coord}} \sum_{i=0}^{B} \sum_{j=0}^{B} 1_{ij}^{obj} \left[ (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
+ \sum_{i=0}^{B} \sum_{j=0}^{B} 1_{ij}^{obj} (C_{ij} - \hat{C}_{ij})^2 + \lambda_{\text{noobj}} \sum_{i=0}^{B} \sum_{j=0}^{B} 1_{ij}^{noobj} (C_{ij} - \hat{C}_{ij})^2 \\
+ \sum_{i=0}^{B} \sum_{c \in \text{classes}} 1_{i}^{obj} (p_{i}(c) - \hat{p}_{i}(c))^2,
\end{align*}
$$
2.2 Object Detection

which computes the Mean Squared Error (MSE) of the difference between true parameters and predicted parameters. No motivation for the choice of this loss function is given by the authors [47]. The reason the location loss and size loss parameters are gathered in their corresponding summations is to make it easier to read. The MSE is computed for each parameter independently.

1_{i}^{obj} denotes if cell i contains an object and 1_{ij}^{obj} if the j-th bounding box predictor in the grid cell i is a candidate for the prediction. The λ parameters are used to increase the loss from bounding box coordinate and to decrease the loss from confidence predictions in boxes that don’t contain objects.

YOLO is fast but not good at recognizing small or irregularly shaped objects due to a limited number of bounding boxes at a single, coarse-grained, feature map [48].

**Single Shot Detector**

Single Shot Detector (SSD) [49] is one of the first attempts at using convolutional neural networks in pyramidal feature hierarchies [50] for efficient detection of objects of various sizes. Using the fine-grained feature maps from earlier levels for detecting small objects and the coarse-grained feature maps for detecting large objects (see Figure 2.5). The detection happens at every pyramidal layer. However, SSD does not split the image into a grid like YOLO but predicts offsets of predefined anchor boxes for every location of the feature map. Here each box has a fixed size and position relative to its corresponding cell.

![Figure 2.5 SSD framework [49]. Left: The input to SDD is comprised of images with their corresponding bounding boxes. Center: In fine-grained feature-maps the default boxes of different aspect ratios corresponds to a smaller area. Right: For coarse-grained feature maps these boxes are bigger and thus more suitable for larger objects.](image_url)

Moreover, the anchor boxes on different levels are rescaled so that one feature map is only responsible for objects at one particular scale. The width, height and the center location of an anchor box are all normalized to be (0, 1). Every location (i, j) of the l-th feature layer
of size $m \times n$ has a linear scale value associated to it proportional to its layer level as well as 6 different width-to-height ratios. Giving a total of 6 anchor boxes per feature cell where the scale at each level is

$$s_l = s_{\text{min}} + \frac{s_{\text{max}} - s_{\text{min}}}{L-1}(l-1)$$

where the level index $l = 1, \ldots, L$, the aspect ratios $r \in \{1, 2, 3, 1/2, 1/3\}$, with an additional scale $s'_r = \sqrt{s_l s_{l+1}}$ when $r = 1$. The width and height for each box can then be computed as $w'_r = s_l \sqrt{r}$ and $h'_r = s_l / \sqrt{r}$ respectively where the center location $(x'_i, y'_i) = (\frac{i+0.5}{m}, \frac{j+0.5}{n})$.

At every location of each feature map, the model outputs four anchor box offsets, $c$ class probabilities for every one of $k$ anchor boxes obtaining $k \cdot m \cdot n(c+4)$ outputs [49].

The loss function is very similar to the one used in YOLO. Defined by the sum of a localization loss and a classification loss with some minor modifications.

**YOLOv2**

In this version of the YOLO family several modification are applied to the original YOLO in order to make prediction more accurate and faster [51] (see Figure 2.2 for architecture details). Batch normalization is added on all the convolutional layers leading to a significant acceleration of the learning process and improved mAP. Instead of predicting bounding box offsets as SSD, YOLOv2 predicts location coordinates relative to the location of the grid cell normalized to $(0, 1)$ [49].

Given an anchor size $p_w, p_h$ at a certain grid cell with its left corner at $(c_x, c_y)$, the model predicts the offset scale, $(t_x, t_y, t_w, t_h)$ (see Figure 2.6) and a confidence prediction representing the IoU between the predicted box and any ground truth box. The corresponding predicted bounding box $b$ has center $(b_x, b_y)$ and size $(b_w, b_h)$ [49].

The detection is still performed at the final coarse-grained layer missing many of the smaller objects although it passes fine-grained features from a previous layer to the output detection layer [49] (see Table 2.2).

**RetinaNet**

RetinaNet tackles the extreme imbalance between background, that contains no objects, and foreground that holds objects of interest by reshaping the standard cross entropy loss function, such that it down-weights the loss assigned to well-classified examples. The result is that it prevents the large number of easy negatives from overwhelming the detector during training [52].

Following the same approach as SSD, RetinaNet used a featurized image pyramid [50] for object detection at different scales (see Figure 2.7). It up-samples higher-level features by
2.2 Object Detection

Figure 2.6 Bounding boxes with dimension priors and location prediction. The width and height of the box is predicted as offsets from cluster centroids. The center coordinates of the box are predicted relative to the location of the filter using a sigmoid function (Based on [51]).

Table 2.2 YOLOv2 architecture.

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Activation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>32</td>
<td>3 × 3/1</td>
<td>LReLU &amp; BN</td>
<td>224 × 224 × 32</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>2 × 2/2</td>
<td></td>
<td>112 × 112 × 32</td>
</tr>
<tr>
<td>Convolutional</td>
<td>64</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>112 × 112 × 64</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>2 × 2/2</td>
<td></td>
<td>56 × 56 × 64</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 128</td>
</tr>
<tr>
<td>Convolutional</td>
<td>64</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 64</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>56 × 56 × 128</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>2 × 2/2</td>
<td></td>
<td>28 × 28 × 128</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>28 × 28 × 256</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>28 × 28 × 128</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>28 × 28 × 256</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>2 × 2/2</td>
<td></td>
<td>14 × 14 × 256</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 512</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 256</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 512</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 256</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>14 × 14 × 512</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>2 × 2/2</td>
<td></td>
<td>7 × 7 × 512</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 512</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>1 × 1</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 512</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>LReLU &amp; BN</td>
<td>7 × 7 × 1024</td>
</tr>
<tr>
<td>Avgpool</td>
<td>1000</td>
<td>1 × 1</td>
<td></td>
<td>7 × 7 × 1000</td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td>Global</td>
<td></td>
<td>1000</td>
</tr>
</tbody>
</table>
2 and merges the lower level features that undergo a $1 \times 1$ convolutional layer by element-wise addition [52], obtaining rich semantics at all levels of the architecture, matching the speed of previous one-stage detectors while surpassing the accuracy of all existing two-stage state-of-the-art detectors [52].

Figure 2.7 The RetinaNet network architecture uses a Feature Pyramid Network on top of the feed-forward ResNet architecture (Based on [50]).

### YOLOv3

YOLOv3 is the latest contribution to the YOLO family model and is inspired by recent advances in object detection [48]. It uses successive $3 \times 3$ and $1 \times 1$ convolutional layers just like the original architecture but has residual blocks added. Inspired by the featurized image pyramid, predictions are made at three different scales. While YOLOv1 and v2 uses a sum of squared errors for all its loss terms, YOLOv3 predicts the classification and confidence loss for each bounding box using sigmoid cross entropy [48]. Sigmoid cross entropy measures the probability error in classification tasks where each class is independent and not mutually exclusive. Due to this, one can perform multi-label classification where an object can be a human and a child at the same time [53]. This makes YOLOv3 mutually inclusive. YOLOv3 also adds inter-layer connections between higher resolution and deeper feature maps in the same way as in RetinaNet [48].

YOLOv3 achieves higher accuracy than SSD but lower than RetinaNet [54]. However, it is faster than both SSD and RetinaNet, which makes it the natural way to go for a real-time object detection solution [54].

#### 2.2.3 3D Object Detection

While it is conceptually simple to extend the 2D approach to volumetric data, it is not obvious what architectures and input data will yield good performance. LiDAR-based 3D object
2.2 Object Detection

detection is inevitable for autonomous driving as it gives 360 degrees of visibility with and extremely accurate depth information [55] which is essential for 3D ground truth based decisions. There are three popular representations to handle unstructured point clouds [54]: 1) projecting a point cloud onto one or more 2D planes, 2) using the point clouds directly without any structured form and 3) using a 3D voxel grid.

3D Point-based Approaches

PointNet is the first end-to-end 3D point-based classification [56]. This 3D CNN consumes point clouds directly, respecting the permutation invariance of points in the input. This is of key importance as the model shouldn’t assume any spatial relationships between points. Based on properties of point set in \( \mathbb{R}^n \) this means that the N 3D point sets need to be invariant to N! permutations of the input set in the feeding order [57].

The network learns a collection of point functions that selects representative points from an input point cloud. The final fully connected layers of the network aggregate these values into the global descriptor for the entire shape as mentioned. However, PointNet does not capture local structures induced by the metrics space points live in [58]. This was solved in [59] by introducing a hierarchical neural network called PointNet++ that applied PointNet recursively on a nested partitioning of the input point set. The bottleneck of both methods is that they consume point clouds directly which are usually comprised of approximately 100 000 points, making both the training and inference computationally and memory expensive [56, 59] making them unsuitable for real-time applications.

Frustum PointNet reduced the search space following the dimension reduction principle [60]. First, the 3D bounding frustum is extracted by extruding 2D bounding boxes from a 2D image detector. Then, within the 3D space trimmed by each of the frustums 3D object segmentation and 3D bounding box regression is performed using a PointNet scheme. However, this approach use both images and point clouds in a sensor fusion manner, which is outside the scope of this thesis. Moreover, the referenced model runs with a too low frame-rate of 7 FPS on an NVIDIA GTX 1080i GPU [60].

Voxel-based Approaches

Voxel-based approaches use a 3D box grid over the input point cloud data. Zhou et al. [61] proposed an end-to-end point cloud object detection model comprised of three blocks: a Feature Learning Layer (FLN), a Convolutional Middle Layer (MDL) and a Region Proposal Network (RPN) [61].
The first block takes in non-empty preprocessed point clouds and passes them through a stack of VFE layers; each of them consisting of: a linear layer, batch normalization layer and a rectified linear unit that maps each point into a feature space. This feature space is then augmented by its locally aggregated feature obtaining an output feature set for each voxel, which is then passed on to the next VFE. Because of the combination of these two features, the VFE stacking encodes point interactions within a voxel, learning descriptive shape information represented as a sparse 4D tensor [61].

This tensor is passed to a stack of convolutional middle layers: a 3D convolution, batch normalization layer and ReLU layer adding more context to the shape description. The volumetric representation obtained from the MDL is consumed by the RPN, a highly optimized algorithm for efficient object detection, and yields the detection result in the form of a probability score map and a regression map. Despite of its high accuracy, the model ends up with an inference time of approximately 5fps on a Titan X GPU [61].

### 2D Projection Approaches

Recently, increasing attention has been drawn towards approaches that projects 3D point cloud onto one or more 2D planes [54, 62, 63]. Most of them are based on bird’s eye view (BEV) on which 2D object detection is performed, allowing the generation of 3D bounding box proposals (see Figure 2.8 for a typical architecture overview). This minimizes the inference time by, at least, one order of magnitude [54] compared to the two previous methods as the input to evaluate is heavily simplified.

Chen et al. [54] proposed a point cloud projection on the X-Y plane into a 3 channel bird eye view image encoded by height, intensity and density [64]. This allows the network to learn to extract information out of each of the channels. BEV representations are more computationally friendly compared to 3D voxel grids. However, information loss is inevitable, which necessarily results in a lower accuracy but with a lower inference time. This dimension
reduction is reasonable in the context of autonomous driving as the objects of interest are on the same ground [54].

With this approach, object detection in point clouds can be treated as object detection in 2D images with the consequent transformation of the coordinates from the detected objects in the bird eye-view to its 3D point cloud counterpart. Since the projections contain spatial information about the location of points in 3D, one can train a neural network to regress 3D bounding boxes from 2D images.

2.3 Object Tracking

Tracking objects in an environment can be done in several ways, but the most widely used method is using Kalman filters for state estimation, given a dynamical model of the object. If the dynamical model of the object to track is unknown or if it is not obvious, one will have to estimate the model of the object whilst tracking it. This can be done using Random Sample Consensus (RANSAC). Since the system, which tracks the dynamic objects, is also moving, it is critical to keep track of how the sensor moves in time. If one can keep track of how the sensor moves between consecutive time-steps, the relative movement between the sensor and the moving objects can be determined [3]. Since we are using the point cloud input from a 3D laser scanner, point cloud registration can be used to find the sensor pose relation between two consecutive scans and thus one can obtain the motion of the sensor over time, given the time interval between the two scans [2]. This process can be referred to as lidar odometry [8].

2.3.1 Model-based Object Tracking

Moosmann et al. [2] proposes a joint solution of both self-localization and DATMO which tracks arbitrary objects using a track before detect approach together with dynamic data partitioning. Input to the algorithm is a set of range measurements, i.e., a point cloud \( \mathcal{P}_t = \{(x_t, y_t, z_t)^T\}_{t=0}^m \). From \( \mathcal{P}_t \) object hypotheses are generated based on object detection using point cloud segmentation. Each object hypothesis \( \mathcal{S}_t \) is turned into a tracklet \( \tau_t^{(k)} \) which comprises the set of tracklets \( \mathcal{T}_t^{(k)} = \{\tau_t^{(k)}\}_{t=0}^n \). The superscript \( (·)^{(k)} \) denotes the current time index and the subscript \( (·)_t \) denotes the measurement time. The tracklets are predicted and updated along \( k \) frames and merged with existing tracks. Point cloud registration is performed on segmented points of moving objects and the unsegmented point cloud of the previous time-step to find the corresponding relative motion. This is combined with a Kalman Filter in order to predict and update the motion of all tracked objects. The output of the
algorithm is a set of tracks of moving objects and a track of the static points in the sensor coordinate system [2].

Most model-based object tracking approaches use some variant of Kalman filtering together with an appropriate pre-defined model. When the task is to track a two- or four wheeled vehicle, a standard bicycle model suffices for state predictions, since it has been proven to be an appropriate kinematic model for most ground vehicles [65].

The Kalman Filter

Kalman filters have a wide variety of applications. Some of the most common applications are navigation and control for vehicles, radar tracking for anti-ballistic missiles, process control, etc. Using the Kalman filter, one can fuse together multiple sensors that give information about a common quantity. This is usually referred to as Sensor Fusion. In the context of this project, we have one physical sensor – a LiDAR – and a virtual sensor, which is the obtainment of bounding boxes and classes from the 3D object detection. What is important for a Kalman filter to work is that one needs a model of the entity to track. The choice of the model will be covered in the next section.

The Kalman filter implements a belief system for computations of continuous states. This means that the Kalman filter can predict a future state based on a priori information [66]. Given an a priori state vector $x_{k-1} = (x^{(1)}_{k-1}, x^{(2)}_{k-1}, \ldots, x^{(n_x)}_{k-1})^T$ at timestep $k-1$ and input vector $u_k = (u^{(1)}_k, u^{(2)}_k, \ldots, u^{(n_u)}_k)^T$, one can compute the a posteriori state vector at timestep $k$. Kalman filters work on linear state space models of the form

$$
\begin{align*}
    x_k &= F_k x_{k-1} + G_k u_k + w_{k-1}, \\
    z_k &= H_k x_k + v_k.
\end{align*}
$$

Here $F$ is the $n_x \times n_x$ state transition matrix, $G$ is the $n_x \times n_u$ control input matrix and $w$ is the Gaussian distributed process noise vector. $z$ is the output vector with length $n_z$, $H$ is the $n_z \times n_x$ observation matrix and $v$ is the measurement noise vector [66].

The Prediction Step

The state vector at timestep $k$ given the estimated state vector at $k-1$ can be predicted using [66]

$$
\hat{x}_{k|k-1} = F_k \hat{x}_{k-1} + G_k u_k,
$$

where $\hat{x}_{k|k-1}$ is the predicted state vector and $\hat{x}_{k-1}$ is the estimated state vector from the previous timestep. The state covariance matrix must also be predicted. It gives a confidence
2.3 Object Tracking

measurement of how accurate the state prediction is. The predicted state covariance matrix is computed as [66]

\[ P_{k|k-1} = F_k P_{k-1} F_k^T + Q_k, \]

(2.5)

where \( P_{k-1} \) is the previous estimated state covariance matrix and \( Q_k \) is the \( n_x \times n_x \) process noise covariance matrix.

Let an observation \( z_k = (z_k^{(1)}, z_k^{(2)}, \ldots, z_k^{(n_y)})^T \) be predicted using

\[ \hat{z}_k = H_k \hat{x}_{k|k-1}, \]

(2.6)

where \( \hat{z}_k \) is the predicted observation at timestep \( k \). As can be seen in the equation above, observations can be predicted from the predicted state using the observation matrix \( H_k \).

The Validation Gate

Once the prediction step is done, one wants to acquire observations to be used for the update step of the algorithm. After acquiring an observation \( z_k \) from a sensor, the error between the prediction and the actual observation can be computed as

\[ \nu_k = z_k - \hat{z}_k, \]

(2.7)

where \( \nu_k \) is often referred to as the innovation. The innovation covariance matrix \( S \) provides the error of the observation fitting the state. The innovation covariance matrix can be computed as

\[ S_k = H_k P_{k|k-1} H_k^T + R_k, \]

(2.8)

where \( R \) is the \( n_z \times n_z \) measurement noise matrix.

In order to validate the association of the observation to the state, i.e., how well the observation can be used to estimate a new state, one has to perform a so called innovation test, which utilizes a validation gate based on a statistical measure. To implement the innovation test, the Mahalanobis distance can be used:

\[ d_k^2 = \nu_k^T S_k^{-1} \nu_k, \quad d_k^2 \in \chi^2(m), \]

which is a unit-less, \( \chi^2 \) distributed number. Depending on the degrees-of-freedom (DOF) denoted \( m \), a threshold value \( \chi^2_{95}(m) \) can be chosen which guarantees a 95\% confidence of association for all distances below the threshold. For 2 DOF, the resulting validation gate would be:

\[ d_k^2 \leq \chi^2_{95}(2) = 5.991. \]
The amount of DOF is based on the amount of random variables present in the model.

**The Update Step**

Since there is a measure of how uncertain our state prediction is, a gain can be computed that decides how much each state variable should be updated based on the innovation. The gain is often referred to as the Kalman gain $K_k$ and is computed using \[66\]

$$K_k = P_{k\mid k-1} H_k^T S_k^{-1}. \quad (2.9)$$

If an observation passes through the validation gate, we update the predicted state vector using \[66\]

$$\hat{x}_k = \hat{x}_{k\mid k-1} + K_k \nu_k,$$

and the state covariance matrix using \[66\]

$$P_k = (I - K_k H_k) P_{k\mid k-1}. \quad (2.10)$$

The computed $\hat{x}_k$ is called the estimated state and $P_k$ is called the estimated state covariance matrix. These will be used as a priori information in the next timestep.

If there is no observation passing through the validation gate, the estimated state and state covariance matrix are updated solely on the predicted counterparts. This is equivalent to setting the Kalman gain to zero.

**Trimming the Kalman Filter**

The trim of a Kalman filter is based on the choice of the noise covariance matrices $Q$ and $R$. For each variable in the state vector, $Q$ will have an associated variation value along the diagonal. Similarly, $R$ will contain the variation information for the innovation, also along the diagonal. By plotting the confidence interval associated with each state variable, together with its signal, one can detect if the confidence interval is too large in comparison to the variation of the actual signal of the state variable. If this is the case, one can decrease the value of the corresponding row in the diagonal of $Q$. The same procedure is done for $R$ using the innovation [67].

If one can obtain a good knowledge of the noise characteristics of the state space model, in such a way that $w$ and $v$ can be accurately estimated, the Kalman Filter can be considered optimal and is perfectly trimmed.
The Extended Kalman Filter

While the Kalman filter works well for linear state space models, it is inapplicable to problems governed by nonlinear models. This is where the Extended Kalman Filter (EKF) becomes useful. The EKF assumes that the next state probability and the measurement probabilities are governed by nonlinear functions $f$ and $h$, respectively [66]. The state space model is thus of the form:

$$x_k = f(u_k, x_{k-1}) + w_{k-1},$$
$$z_k = h(x_k) + v_k.$$

The initial state $x_0$ is a random vector with known mean $\mu_0 = E[x_0]$ and the initial covariance is $P_0 = E[(x_0 - \mu_0)(x_0 - \mu_0)^T]$.

The belief estimate of the EKF is calculated through an approximation to the true belief. This is done via a linearization called first order Taylor expansion. Take $f$ for example – $f(u_k, x_{k-1})$ can be approximated by its value at the mean $\mu_{k-1}$ of the posterior $x_{k-1}$ and the input $u_k$, and the linear extrapolation is achieved by the Taylor expansion [66]:

$$f(u_k, x_{k-1}) \approx f(u_k, \mu_{k-1}) + J^F_k (x_{k-1} - \mu_{k-1}),$$

where $J^F_k$ is the Jacobian which gives the gradient of each parameter in $f(u_k, x_{k-1})$ at timestep $k$:

$$J^F_k = \frac{\partial f(u_k, \mu_{k-1})}{\partial \mu_{k-1}}.$$

The exact same linearization procedure is applied to the measurement function $h$. Here the Taylor expansion is developed around the mean $\hat{\mu}_k$ of the predicted state $\hat{x}_k$. $h(x_k)$ is approximated as [66]

$$h(x_k) \approx h(\hat{\mu}_k) + J^H_k (x_k - \hat{\mu}_k),$$

where $J^H_k = \frac{\partial h(\hat{\mu}_k)}{\partial \hat{\mu}_k}$.

In contrast to the steps involved in the Kalman Filter algorithm, the EKF is implemented by simply swapping the $F$ and $H$ matrices in equations (2.4) to (2.10) for $J^F$ and $J^H$, respectively.
The Kinematic Bicycle Model

The bicycle model is a widely used and a good approximation of most vehicle kinematics [65, 68]. The kinematic bicycle model comprises two wheels: one at the front which handles steering, and one at the back, which acts as a constraint (see Figure 2.9) [68].

The control inputs to the model correspond to the acceleration denoted $a$ and the steering angle of the front wheel denoted $\delta_f$. Kong et al. [69] proposes a bicycle model which includes slipping based on a slip angle $\beta$:

\[
\dot{x} = v \cos (\psi + \beta), \\
\dot{y} = v \sin (\psi + \beta), \\
\dot{\psi} = \frac{v}{\ell_r} \sin (\beta), \\
\dot{v} = a,
\]

where $\beta$ is the slip angle at the center of mass, computed as [69]

\[
\beta = \tan^{-1} \left( \frac{\ell_r}{\ell_f + \ell_r \tan(\delta_f)} \right).
\]

In the kinematics model, $x$ and $y$ are the coordinates of the center of mass of the vehicle in an inertial frame $O = (X, Y)$. $v$ is the linear velocity and $\psi$ is the inertial heading of the vehicle. $\ell_r$ and $\ell_f$ are the distances from the center of mass to the rear- and front axles, respectively. $a$ is the acceleration of the vehicle along the same axis as the velocity $v$ [69].
Without the slip integrated to the model, the model can be simplified to

\[
\begin{align*}
\dot{x} &= v \cos(\psi), \\
\dot{y} &= v \sin(\psi), \\
\dot{\psi} &= \frac{v}{\ell_r + \ell_f} \tan(\delta_f), \\
\dot{v} &= a,
\end{align*}
\]

which is sufficient in combination with an EKF for tracking, since the slipping of the objects will be captured by the corrections of the filter based on a continuous stream of observations. Here, \( \tan(\delta_f) \) is equivalent to \( \frac{R}{\ell_r + \ell_f} = \frac{R}{L} \), where \( R \) is the turning radius of the bicycle. This means that the angular velocity of the model is dependent in a proportion of velocity and the turning radius \( \frac{v}{R} \). This makes sense since the higher the turning radius and lower the velocity, the lower the angular velocity will be, and vice versa.

### 2.3.2 Model-free Object Tracking

Dewan et al. [3] proposes a model-free approach for detecting and tracking dynamic objects in urban environments. Instead of relying on detecting changes in the environment caused by motion, the authors segment distinct objects using motion cues. The method is claimed to be superior compared to the model based method proposed by Moosmann et al. [2].

The proposed model by [3] uses RANSAC to estimate motion models for both the sensor and the dynamic objects. Point correspondences between two consecutive scans are found by uniformly sampling keypoints in the previous point cloud and matching their SHOT [70] descriptors against all points in the current point cloud. The point pairs with the minimum descriptor distance are matched together.

Bahraini et al. [6] proposes the Multilevel Random Sample Consensus (ML-RANSAC) algorithm that enhances the speed of RANSAC. ML-RANSAC decreases the number of generated hypotheses using a compatibility matrix and sequentially improves the performance of data association and estimation. Combining ML-RANSAC with an object detection algorithm the authors in [6] proposes a framework presented in Figure 2.10 that solves the SLAM and DATMO problems. The algorithm successfully tracks moving objects while running SLAM, via robust data association techniques. The main advantage is that ML-RANSAC can use dynamic objects as landmarks for localization and mapping [6].
Figure 2.10 A proposed framework for SLAM in dynamic environments using ML-RANSAC [6].

Random Sample Consensus

Some moving objects are very difficult to associate a specific model to. Therefore, the model may instead be estimated online based on previous knowledge of the motion behavior of the object. One way to estimate the model of an object is to use Random Sample Consensus (RANSAC).

The main concept of RANSAC is to form several simple hypotheses of a model from a batch of data and identify the best matching hypothesis to the measurements. RANSAC was developed to reduce the effects of spurious measurements and has played a major part within the computer vision community due to its robustness and efficiency [71].

The RANSAC algorithm comprises two repeated steps. The first step is the generation of hypotheses. A randomly minimal sample subset is selected from the input data to form a set of hypotheses. The second step is hypothesis validation, which verifies if the data is consistent with the estimated model, which was obtained from the first step. The hypotheses that lie outside of the confidence interval of the estimated model will be removed [6].

RANSAC is best described in pseudo code. Algorithm 1 is an example of how the RANSAC algorithm can be written [72]. In Algorithm 1 we have that $n$ is the minimum amount of points necessary to fit the model, $k$ is the maximum number of iterations, $t$ is the inlier threshold, and $d$ is the cutoff threshold for a good fit.

2.3.3 Point Cloud Registration

The Iterative Closest Point algorithm (ICP) has since it’s introduction become the most extensive algorithm for aligning 3D shapes [73]. In mobile robotics, the availability of range
Algorithm 1 Random Sample Consensus

**Require:** data, model, n, k, t, d

1: bestModel $\leftarrow$ None
2: bestFit $\leftarrow \infty$
3: while $i < k$ do
4: sample $\leftarrow$ draw n random points from data
5: Fit model to sample
6: inliers $\leftarrow$ data within $t$ of model
7: if inliers $>$ bestFit then
8: Fit model to all inliers
9: bestFit $\leftarrow$ fit
10: bestModel $\leftarrow$ model
11: if inliers $>$ $d$ then
12: return model
13: return bestModel

sensors that can quickly capture the 3D environment has enabled the use of optimization-based methods like ICP for scan-to-scan registration [74]. Since the early 90s when the ICP algorithm was first proposed [75], a wide variety of variations have been proposed over the years. ICP-based methods are still considered the state of the art when it comes to scan matching [74].

**Point-to-point ICP**

ICP finds the optimal affine transformation between two consecutive point sets such that the Euclidean distance between each associated point pair are minimized. The point pair associations have to be made before proceeding with the algorithm; one common way of associating points in this case is by finding point pairs, which comprises the points that are closest to each other [75].

Let $\{x_i\}_{i=0}^{n} \in \mathcal{X}$ be the point set in the target surface $\mathcal{X}$ and $\{p_i\}_{i=0}^{m} \in \mathcal{P}$ be the point set in the source surface $\mathcal{P}$. The point $y_i \in \mathcal{X}$ which is closest to a point $p_i \in \mathcal{P}$ can be computed as [75]

$$y_i = C(p_i, \mathcal{X}) = \arg\min_{x \in \mathcal{X}} \|x - p_i\|^2,$$

where the whole set of closest points $\mathcal{Y} = \{y_0, y_1, \ldots, y_m\}$ can be computed as $\mathcal{Y} = C(\mathcal{P}, \mathcal{X})$.

Let $\mathcal{L}$ be a set of indices such that $\ell_i < \delta$ holds, where $\ell_i = \|y_i - p_i\|$ and $\delta > 0$ is a threshold value. Let $\mathcal{Y}_c$ be a set of points $y_c \in \mathcal{Y}$, $c \in \mathcal{L}$ and let $\mathcal{P}_c$ be a set of points $p_c \in \mathcal{P}$, $c \in \mathcal{L}$. We want to find a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{d}$ such that the
following objective function is minimized [76]:

$$E(R, d) = \sum_{j \in \mathcal{L}} \|y_j - (Rp_j + d)\|^2. \quad (2.11)$$

An illustration of the distance error between the two surfaces is presented in Figure 2.11.

Let $\mu_y$ be the centroid of the points in $\mathcal{Y}_\mathcal{L}$ and $\mu_p$ be the centroid of the points in $\mathcal{P}_\mathcal{L}$, then the translation vector becomes $d = \mu_y - R\mu_p$. Let $y_j^* = y_j - \mu_y$ and $p_j^* = p_j - \mu_p$, $\forall j \in \mathcal{L}$. Since the translation vector $d$ can be described as a function of the rotation matrix $R$, we can now rewrite Equation (2.11) as [76]

$$E(R) = \sum_{j \in \mathcal{L}} \|y_j^* - Rp_j^*\|^2,$$

and the objective is to find the rotation matrix $R$ which minimizes $E(R)$:

$$R \leftarrow \arg\min_R \{E(R)\}.$$ 

One can find $R$ by computing the singular value decomposition (SVD) of $M$:

$$M = U \Sigma V^T \implies R = VU^T,$$

where $M$ is the cross-covariance matrix of $\mathcal{Y}_\mathcal{L}$ and $\mathcal{P}_\mathcal{L}$, such that

$$M = \text{cov}(\mathcal{Y}_\mathcal{L}, \mathcal{P}_\mathcal{L}) = \sum_{j \in \mathcal{L}} y_j^*(p_j^*)^T.$$
The source point set in $\mathcal{P}$ is then updated by the affine transformation

$$p_i \leftarrow Rp_i + d, \quad \forall p_i \in \{p_i\}_0^m.$$  

ICP is considered to have converged when the objective function $E(R, d)$ in (2.11) reaches below a certain threshold $\tau > 0$ such that $E(R, d) < \tau$ [75].

**Point-to-plane ICP**

Point-to-plane ICP was first introduced by Chen and Medioni [77] and is a more robust and accurate variant of the original ICP [74].

Point-to-plane ICP is usually solved by using standard nonlinear least squares methods, such as the *Levenberg-Marquardt method*. This means that each iteration is slower than the point-to-point version of ICP, which is solved by closed-form solutions [73]. While point-to-plane ICP is slower than point-to-point ICP, it has been proven to converge much faster in the sense that it takes less iterations for point-to-plane ICP to converge than its counterpart [78].

In point-to-plane ICP, we are trying to minimize the sum of the squared distance between each source point and the tangent plane at the corresponding target point (see Figure 2.12) [73]:

$$E(R, d) = \sum_{j \in \mathcal{L}} ((y_j - (Rp_j + d)) \cdot n_j)^2, \quad (2.12)$$

where $n_j \in \mathcal{N}$ is a unit normal vector at the destination point $y_j$ and $(\cdot)$ is the dot product. $\mathcal{N}$ is the set of all unit normal vectors corresponding to each point in $\mathcal{Y}$, respectively.

![Figure 2.12 Illustration of the point-to-plane error between two surfaces (Based on [73]).](image)
We rewrite the objective function $E(R, d)$ in Equation (2.12) as

$$E(\beta) = \sum_{j \in L} (y_j \cdot n_j - f_j(\beta))^2,$$

where $\beta = [\phi, \theta, \psi, x, y, z]^T$ and $f_j(\beta) = (R(\phi, \theta, \psi)p_j + d(x, y, z)) \cdot n_j$, $\forall j \in L$. The rotation matrix $R(\phi, \theta, \psi) = R_z(\psi)R_y(\theta)R_x(\phi)$ is a function of the Euler angles $[\phi, \theta, \psi]$ and the translation vector $d(x, y, z)$ is a function of the offset variables $[x, y, z]$. The goal is to find the parameter vector $\beta$ that minimizes $E(\beta)$:

$$\beta = \arg\min_{\beta} \{E(\beta)\}. \quad (2.13)$$

We have now defined a Nonlinear Least Squares Minimization problem and can use the Levenberg-Marquardt (LM) algorithm to find a solution to Equation (2.13). At each iteration step of the LM algorithm, $\beta$ is replaced by a new estimate $\beta + \delta$. $\delta$ can be determined from the linear approximation of $f_j(\beta + \delta)$:

$$f_j(\beta + \delta) \approx f_j(\beta) + J_j \delta$$

where $J_j$ is the gradient vector of $\beta$ and is computed as:

$$J_j = \frac{\partial f_j(\beta)}{\partial \beta},$$

here $J_j$ is considered as a row vector.

The partial derivatives of $f_j(\beta)$ for each parameter in $\beta$ and for a certain point pair $(p_j, y_j)$, $j \in L$ are computed as follows:

$$\frac{\partial f_j(\beta)}{\partial \phi} = \left(\frac{\partial R(\phi, \theta, \psi)p_j}{\partial \phi}\right) \cdot n_j, \quad \frac{\partial f_j(\beta)}{\partial x} = \left(\frac{\partial d(x, y, z)}{\partial x}\right) \cdot n_j,$$

$$\frac{\partial f_j(\beta)}{\partial \theta} = \left(\frac{\partial R(\theta, \phi, \psi)p_j}{\partial \theta}\right) \cdot n_j, \quad \frac{\partial f_j(\beta)}{\partial y} = \left(\frac{\partial d(x, y, z)}{\partial y}\right) \cdot n_j,$$

$$\frac{\partial f_j(\beta)}{\partial \psi} = \left(\frac{\partial R(\phi, \theta, \psi)p_j}{\partial \psi}\right) \cdot n_j, \quad \frac{\partial f_j(\beta)}{\partial z} = \left(\frac{\partial d(x, y, z)}{\partial z}\right) \cdot n_j.$$

Using the linear approximation in Equation (2.13), one can rewrite the updated objective function as

$$E(\beta + \delta) \approx \sum_{j \in L} (y_j \cdot n_j - f_j(\beta) - J_j \delta)^2.$$
Taking the derivative of $E(\beta + \delta)$ with respect to $\delta$ and setting the result to zero provides a formula to compute $\delta$ which moves $E(\beta)$ towards an optimum:

$$\delta = (J^T J + \lambda I)^{-1} J^T [y \cdot n - f(\beta)],$$

where $I$ is the identity matrix and $\lambda > 0$ is the damping factor. In a multiple minima case, the solution only converges to a global minimum if the initial guess $\beta_0$ is somewhat close to the true solution.

### Generalized-ICP

Generalized-ICP (G-ICP) is based on adding a probabilistic model to the minimization objective [74]. Segal et al. [74] has shown that G-ICP outperforms both standard ICP and point-to-plane ICP. In G-ICP, the surface structure from both scans are considered instead of from just one, as it is done with point-to-plane ICP. G-ICP can thus be thought of as a "plane-to-plane" variant of ICP [74].

Assume that when measuring a point, we are taking a random sample from a surface. Let the points from the source point set $\{p_j\}_{0}^{L} \in \mathcal{P}_L$ and the target point set $\{y_j\}_{0}^{L} \in \mathcal{Y}_L$ inherit the normal distribution $p_j \sim \mathcal{N}(\hat{p}_j, C^P_j)$ and $y_j \sim \mathcal{N}(\hat{y}_j, C^Y_j)$.  $\hat{p}_j$ and $\hat{y}_j$ are the expected values of the measured points. $C^P_j$ and $C^Y_j$ are the co-variance matrices of the points in their corresponding point sets [74].

If we assume correct correspondences and that we can compute the correct rotation $R^*$ and translation $d^*$, it is given that

$$\hat{y}_j = R^* \hat{p}_j + d^* = T^*[\hat{p}_j, 1]^T,$$

where $T^*$ is the corresponding rigid transformation matrix.

We define the residual of an arbitrary rigid transformation as $\Delta_j = y_j - T^* p_j$. Since $y_j$ and $p_j$ are assumed to be drawn from independent Gaussians, we can assume that the residual $\Delta^*_j$ of the correct transformation belongs to the following normal distribution [74]:

$$\Delta^*_j \sim \mathcal{N}(\hat{y}_j - (T^*)^* \hat{p}_j, C^Y_j + (T^*)^* C^P_j (T^*)^T)$$

$$= \mathcal{N}(0, C^Y_j + (T^*)^* C^P_j (T^*)^T).$$
Using Maximum Likelihood Estimation (MLE), one can iteratively compute the MLE estimator $T$ as \[ T = \arg\max_T \sum_{j \in \mathcal{L}} \log(P(\Delta_j)) = \arg\min_T \sum_{j \in \mathcal{L}} \Delta_j^T (C_j^Y + T C_j^P T^T)^{-1} \Delta_j, \]

which defines the key step of the G-ICP algorithm. Every sampled point is considered to be distributed with high co-variance along its local plane, i.e., orthogonal to its surface normal. Given $\mu_j$ and $\nu_j$ – the normal vectors at $y_j$ and $p_j$, respectively – one can compute $C_j^Y$ and $C_j^P$ by rotating the co-variance matrix \[ \Sigma_x = \begin{pmatrix} \varepsilon & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \]

so that the $\varepsilon$ term represents the uncertainty along the surface normal, where $\varepsilon$ is a small constant value. Let $R_x$ denote a rotation matrix which transform the basis vector $e_x \rightarrow x$, where $x$ is an arbitrary vector in $\mathbb{R}^3$. We compute the co-variance matrices as \[ C_j^Y = R_{\mu_j} \Sigma_x R_{\mu_j}^T; \]
\[ C_j^P = R_{\nu_j} \Sigma_x R_{\nu_j}^T. \]

An illustration of the plane-to-plane error together with the co-variance ellipses is shown in Figure 2.13.

Figure 2.13 Illustration of the plane-to-plane error between two surfaces, as described by [74].
The optimization problem in Equation (2.3.3) can be solved using the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [76].

### 2.3.4 LiDAR Odometry

LiDAR Odometry is a relatively recent field of research [8, 5, 79, 80]. The idea is to utilize the relation between *a priori* and *a posteriori* scans in order to find the scan-to-scan motion of the sensor. This grants the system the knowledge of vehicle motion without the use of an IMU.

Zhang *et al.* [8] propose a LiDAR Odometry algorithm that runs at high frequency but low fidelity. Their solution is two-fold in that it both removes distortions of the input point cloud and estimates the motion of the LiDAR between two consecutive sweeps. The point cloud registration process is similar to point-to-plane ICP, but with capabilities to connect an IMU for increased precision. To match consecutive point clouds, feature extraction is done to increase the robustness and speed of the point cloud registration process. Feature extraction is done on points located on edges and planar surfaces. The feature points are then associated to edge-line segments and planar surface patches, respectively. The proposed LiDAR Odometry algorithm is a part of a SLAM solution called LiDAR Odometry and Mapping (LOAM), proposed by the same authors. The performance of LOAM has been evaluated through the KITTI dataset [23]. With a translation error of 0.59% and a rotational error of 0.0014 [deg/m] while also running at 10 FPS on a 2 core 2.5 GHz CPU, it has achieved great results compared to other SLAM methods. LOAM is at the moment of writing placed 2nd in the KITTI Visual Odometry / SLAM Evaluation [81].

Shan *et al.* [5] builds upon LOAM to increase the real-time performance and accuracy in noisy environments. They call their solution lightweight and ground-optimized, meaning that they optimized the solution for robust real-time SLAM running on an embedded system. The method separates the ground from the rest of the point cloud data and discards points that may yield unreliable features via point cloud segmentation. This is because the ground plane is considered to contain the most stable set of points to extract planar features from. They authors also introduce a two-step optimization process for pose estimation. In the first step, three of the pose states $[t_z, \theta_{roll}, \theta_{pitch}]$ are obtained from planar features extracted from planar features like the ground segment. In the second step, the rest of the pose states $[t_x, t_y, \theta_{yaw}]$ are obtained from edge features extracted from the segmented point cloud, which is separated from the ground. The authors call their solution LeGO-LOAM [5]. LeGO-LOAM achieves a feature extraction speed that is 10x faster than LOAM. Moreover, the LiDAR Odometry algorithm runs 8x faster that its predecessor [5, TABLE IV].
Except for LiDAR Odometry solutions that utilize optimization techniques like ICP (also called geometry-based approaches), there are also solutions that make use of Deep Learning to find the scan-to-scan relationship between consecutive point clouds. Cho et al. [79] propose a novel approach of deep LiDAR Odometry, which incorporates the ICP algorithm into a Deep Learning framework. The system comprises feature networks (FeatNet), which extracts the feature vectors of consecutive scans, and a pose network (PoseNet), which estimates the relative motion of the scans from the feature vectors. These networks can be trained using either supervised or unsupervised learning, depending on the availability of ground truth labelling. The authors call their solution DeepLO. In contrary to DeepLO, Li et al. [80] proposes an end-to-end trainable Deep Learning solution for LiDAR Odometry, which does not have to make use of a feature selection/matching and pose estimation pipeline. By performing sequential learning, the proposed solution can exploit more information of the scan-to-scan relatin, compared to processing a single scan. The authors call their solution LO-Net and have demonstrated that LO-Net outperforms existing learning based approaches and that it achieves a similar accuracy to the state-of-the-art geometry-based approach, LOAM.

2.4 Conclusions

Based on the prestudy we see that a good approach for real-time 3D point cloud object detection is projecting the point clouds on the ground plane, encoding them into a 3 channel representation. This makes it possible to run the compressed representation through a real-time 2D object detection such as YOLO which theoretically could obtain up to approximately 100 FPS on a high end GPU. However, the standard YOLO architecture just predicts the location and the size of the object. Hence, the architecture needs to be expanded by enabling it to learn rotations for bounding boxes.

While model-free object tracking seem to be faster compared to the model-based counterpart, it is only beneficial if the objects to be tracked lack a well-defined model. Since this project only focuses on the detection and tracking of cars, cyclists and pedestrians, a model-based approach similar to Moosmann et al. [2] can still be used efficiently. Model-based approaches allow for the behavior of objects within the same category to be consistent throughout the scene and more accurate predictions can be made, especially for cars and cyclists. Pedestrians are a bit more tricky to predict using a pre-defined model, since the movement behavior is not consistent between individuals; thus it makes more sense to use a model-free approach when tracking pedestrians.
By using a LiDAR Odometry algorithm like what is proposed by Shan et al. [5], one can obtain robust and accurate sensor odometry similar to what can be obtained from an IMU, only by using a single LiDAR sensor. This information can be used to obtain world absolute information about the movement of objects in the scene, since the movement of objects are inherently relative to the sensor perceiving them. According to what has been investigated, the choice of which LiDAR Odometry algorithm to use breaks down to two choices: to either use a geometry-based approach or a Deep Learning based approach. Since LOAM [8] has been considered to be the state-of-the-art since its release on the KITTI benchmark [81], and LO-Net [80] achieves similar results to LOAM, the solution proposed by Shan et al. [5] seems like the correct choice based on the many improvements the authors made upon LOAM with regards to speed and robustness.
Chapter 3

Proposed Detection and Tracking Solution

3.1 Object Detection

In this section we present an end-to-end trainable 2D object detection network intended to find objects in projected point clouds. We expand the existing YOLOv3 architecture such that it outputs YOLO’s standard output plus an angle and a height for each detected object. We call the solution YOLO++.

3.1.1 Data Pre-processing

The point clouds with their respective object labels extracted from the KITTI 3D object detection dataset undergo heavy preprocessing before being fed to our architecture. The process can broken down into three different sub-steps listed below:

2D Point Cloud Projection

We base this method on the projection proposed by Simon et al. [54]. The main difference is that the resolution chosen for the pictures is approximately 20% lower, with the aim of obtaining a speed-up in the inference process. Each point cloud is read in its binary format and reorganized into a 2D array containing elements comprised of 3D coordinates and reflectance values. The elements in this array are filtered in the x and y plane such that the unique values within \([-40, 40], [0, 70.4] \text{ meters}\) are kept. The resulting elements are encoded in a \([800 \times 704] = [W \times H]\) image (which yields a resolution of \([10 \times 10] \text{ cm per pixel}\) comprised of 3 channels: density, height and intensity. When more than one point is
mapped to the same pixel one of them is extracted arbitrarily. The density is calculated by
\( \min(1.0, \log(N + 1)/64) \) where \( N \) are the number of coordinates mapped from the binary
point cloud to a certain pixel in its projection. We chose only to project points within the
interval \([-3, 1.5]\) along the z-axis. This is done to remove points that are of no interest,
since all the objects to detect lie within this pre-defined height. Intensity, or reflectance, is
taken directly from the LiDAR output. All of the channels are normalized between 0 and 1
to match the feature scaling of the neural network. An example of the result of this process
can be seen in Figure 3.1.

![Figure 3.1 A point cloud projected onto the bird’s eye view plane.](image)

This representations makes it possible for the network to learn, not only the features in
bird’s eye view corresponding to each object type but also to regress height for them as the
height information is embedded in the point cloud representation.

### Data Splitting

The KITTI 3D object detection dataset is comprised of a training and testing set. However,
the testing portion of the dataset lacks any labels for the images as it is intended to do the
testing over their evaluation server. Moreover, minor modifications of existing algorithms or
student research projects are not allowed [82] which makes the splitting of the training set
unavoidable. Hence, we chose to split the 3 channel projected point clouds from the training
dataset with their respective labels into a testing and training dataset of ratio 0.8 and 0.2 respectively.

**Data Serialization**

The performance of the import pipeline is of vital importance when working on large datasets as it has a direct consequence on the training time of the model. Hence, both the training and testing data sets created in the step above are serialized in the TFRecord format. This is a simple format for storing data in a sequence of binary records, taking less space and making them more efficient to read from disk [83]. For each bounding box, the data is stored in the following format: \([x_0, y_0, x_1, y_1, d, \phi_{re}, \phi_{im}, c]\), where \(x_0, y_0, x_1, and y_1\) are the left, top, right, and bottom positions of the boundary, respectively. \(d\) is the depth of the object with respect to the bird’s eye view plane, i.e., the height of the object in the 3 dimensional space. \(\phi_{re}\) and \(\phi_{im}\) are the components of the rotation vector in the complex plane and \(c\) is the class, encoded as a number.

**3.1.2 YOLO++**

In order to detect objects with YOLOv3 from the preprocessed point cloud accurately, its architecture needs to be expanded. The reason behind this is that YOLOv3 outputs unrotated bounding boxes. This is suboptimal as the orientation of the detected objects is of core importance for reliable perception based decisions such as path planing, where the exact location and orientation of the objects need to be taken into consideration. Moreover, we chose to expand the faster version of YOLOv3 – called Tiny YOLOv3 – as lower inference times are more important than accuracy in this case. Its architecture consists of 2 functional blocks: the Feature Extractor Network and the Detection Layer.

The feature extractor consists of a stack of convolutional and Maxpooling layers. The purpose of the convolutional layers is to add more context to the shape description while the Maxpooling layer is used to down-sample the feature map, passing the size-invariant map forwardly. With no fully-connected layers this architecture makes it possible to deal with feature spaces of any size [51].

The detection is performed at different stages in the YOLO network; each stage may be referred to as a Detection Layer. The reason behind this is that there are objects of different sizes on the image but as the network goes deeper its feature map gets smaller, making it harder to detect smaller objects. Hence, it is better to detect objects at different feature maps of different scales so that small objects do not get lost in the generalization of the training process [50]. This concept is further improved by a Feature Pyramid Network (FPN)
structure to increase the accuracy in the detection process. It uses the up-sampled feature map concatenated with the Feature Extractor Network allowing the network to capture the objects’ information from both low- and high level features. This simplifies the learning of stable features, especially in very deep networks such as this one [50].

In the following subsections, our contributions are colored in blue to make it easier for the reader to distinguish between the related works and our additions upon them.

**Network Details**

The pre-processed point clouds are sent into the network model through a series of pair-wise Convolutional and Maxpooling layers (CM blocks), presented in Table 3.1. This is referred to as the Feature Extractor Network (FEN). While similar to Tiny YOLOv3, the FEN is reduced in the amount of layers it contains but increased in the amount of filters it uses for feature matching. This is because larger grid sizes are needed for the detection layer to find the objects present in the images, but to compensate for the performance loss of losing a convolutional layer, the filter size is doubled for each layer.

Table 3.1 Feature extraction network of YOLO++.

<table>
<thead>
<tr>
<th>Block</th>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Activation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-1</td>
<td>Convolutional</td>
<td>64</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>800 x 704 x 64</td>
</tr>
<tr>
<td></td>
<td>Maxpool</td>
<td></td>
<td>2/2</td>
<td></td>
<td>400 x 352 x 64</td>
</tr>
<tr>
<td>CM-2</td>
<td>Convolutional</td>
<td>128</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>400 x 352 x 128</td>
</tr>
<tr>
<td></td>
<td>Maxpool</td>
<td></td>
<td>2/2</td>
<td></td>
<td>200 x 172 x 128</td>
</tr>
<tr>
<td>CM-3</td>
<td>Convolutional</td>
<td>256</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>200 x 172 x 256</td>
</tr>
<tr>
<td></td>
<td>Maxpool</td>
<td></td>
<td>2/2</td>
<td></td>
<td>100 x 88 x 256</td>
</tr>
<tr>
<td>CM-4</td>
<td>Convolutional</td>
<td>512</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>100 x 88 x 512</td>
</tr>
<tr>
<td></td>
<td>Maxpool</td>
<td></td>
<td>2/2</td>
<td></td>
<td>50 x 44 x 512</td>
</tr>
</tbody>
</table>

The first detection pipeline, presented in Table 3.2, is connected to the output of the CM block, presented in Table 3.1. Its output is a \([W/32] \times [H/32] \times B \cdot (8 + C)\) tensor where \(B\) is the number of bounding boxes predicted using anchors and \(C\) is the number of classes to predict. Since we use three anchors per detection layer and aim to predict objects within three categories (Car, Cyclist and Pedestrian), we have that \(B \cdot (8 + C) = 3 \cdot (8 + 3) = 33\).

The second detection pipeline is connected to the output of DP1-5 belonging to the first detection pipeline. In the second detection pipeline, we are aiming at fine tuning the object detection and to find smaller objects in the point cloud based on lower level features. We are up-sampling the \([W/32] \times [H/32] \times X\) tensor by \(2\times\) along the first two dimensions and we are corencatenating the upsampled tensor by the output of the convolutional layer.
3.1 Object Detection

CM-3 in order to pass on the feature information to the second pipeline. This results in an output tensor $\lceil \frac{W}{16} \rceil \times \lceil \frac{H}{16} \rceil \times B \cdot (8 + C)$ that is double in resolution compared to the first detection layer.

Table 3.2 Detection pipeline 1.

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Activation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1-1</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 512$</td>
</tr>
<tr>
<td>D1-2</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 1024$</td>
</tr>
<tr>
<td>D1-3</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 512$</td>
</tr>
<tr>
<td>D1-4</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 1024$</td>
</tr>
<tr>
<td>D1-5</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 512$</td>
</tr>
<tr>
<td>D1-6</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 1024$</td>
</tr>
<tr>
<td>F-1</td>
<td>Convolutional</td>
<td>33</td>
<td>1/1</td>
<td>ReLU</td>
<td>$50 \times 44 \times 33$</td>
</tr>
</tbody>
</table>

Table 3.3 Detection pipeline 2.

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Activation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2-1</td>
<td>Route from D1-5</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 512$</td>
</tr>
<tr>
<td>D2-2</td>
<td>Convolutional</td>
<td>256</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$50 \times 44 \times 256$</td>
</tr>
<tr>
<td>D2-3</td>
<td>Upsample 2×</td>
<td></td>
<td></td>
<td></td>
<td>$100 \times 88 \times 256$</td>
</tr>
<tr>
<td>D2-4</td>
<td>Concatenate CM-3</td>
<td></td>
<td></td>
<td></td>
<td>$100 \times 88 \times 512$</td>
</tr>
<tr>
<td>D2-5</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 512$</td>
</tr>
<tr>
<td>D2-6</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 1024$</td>
</tr>
<tr>
<td>D2-7</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 512$</td>
</tr>
<tr>
<td>D2-8</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 1024$</td>
</tr>
<tr>
<td>D2-9</td>
<td>Convolutional</td>
<td>512</td>
<td>1/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 512$</td>
</tr>
<tr>
<td>D2-10</td>
<td>Convolutional</td>
<td>1024</td>
<td>3/1</td>
<td>LReLU &amp; BN</td>
<td>$100 \times 88 \times 1024$</td>
</tr>
<tr>
<td>F-2</td>
<td>Convolutional</td>
<td>33</td>
<td>1/1</td>
<td>ReLU</td>
<td>$100 \times 88 \times 33$</td>
</tr>
</tbody>
</table>

Bounding Box Predictions

Bounding box predictions are based on manipulating pre-defined anchors $p_w$, $p_h$, $pd$, $pre$, $pim$. Not only does this help the training process to converge, it also makes the network give more robust and accurate predictions since the only parameters the network will have to learn are scaling factors applied to the anchors.

The center coordinates of the bounding boxes are predicted through a sigmoid function, which squeezes the output between 0 and 1. Since there are at least one bounding box prediction at each cell (the amount of predicted bounding boxes per cell are the same amount as the number of anchors), the location prediction is local within it. To get a bounding box
location that is global throughout the whole image, the prediction simply has to be added with the cell index and multiplied by the stride. The local location predictions for each bounding box can thus be written as

\[ b_x = \sigma(t_x) + c_x, \]
\[ b_y = \sigma(t_y) + c_y. \]

The dimensions of the bounding boxes, \( b_w, b_h, b_d \), are calculated by applying a logarithmic space transform to each dimension and then multiplying it by its respective anchor size. This means that the network only has to predict a scaling of the anchor size, making convergence of the training process much more stable [48]. The bounding box sizes are predicted as

\[ b_w = p_w e^{b_w}, \]
\[ b_h = p_h e^{b_h}, \]
\[ b_d = p_d e^{b_d}. \]

The predicted angles, represented as complex numbers, are predicted by rotating the anchors, which has a pre-defined orientation \([p_{re}, p_{im}]^T\) by \([t_{re}, t_{im}]^T\), where \( t_{re} \) and \( t_{im} \) are sent into the tanh activation function beforehand to squeeze the output between the bounds of the unit circle. Through various experiments, presented in Section 5.1, we have shown that this gives better angle predictions than predicting the angles directly from the network. The rotation is done through complex multiplication, which means that the angles of the bounding boxes are predicted as

\[ b_{re} = p_{re} \tanh(t_{re}) - p_{im} \tanh(t_{im}), \]
\[ b_{im} = p_{re} \tanh(t_{im}) + p_{im} \tanh(t_{re}), \]
\[ b_\theta = \arctan2(b_{im}, b_{re}). \]

Both objectness score and class scores are also passed through a sigmoid function. For the objectness score this means that the probability that the object is contained inside the bounding box is between 0 and 1. For class scores this has also a deeper meaning, namely that it allows for mutually inclusive classes.
Output processing

Detection is made at 2 different feature sizes: \((100 \times 88)\) and \((50 \times 44)\). For each of the cells in the feature maps, 3 bounding boxes are created using 3 anchors, which gives a total of \(((100 \times 88) + (50 \times 44)) \times 3 = 33000\) bounding boxes. However, there is always one single box that fits the detected object best, which leads to the need of some kind of filtering. For this purpose non-maximum suppression is used, which can be summarized in the following steps:

1. Discard all the proposed regions with objectness score under \(X\).
2. While there are any remaining boxes:
   (a) Pick the prediction with the largest objectness score.
   (b) Discard any remaining box with \(IoU \geq Y\) with the output box picked in (a).

Loss function

Our loss functions is based on the one presented in YOLOv3 [48] and the Euler regression part based on M. Simon et al. in [54]. The resulting loss function is comprised of two main terms \(\mathcal{L} = \mathcal{L}_{YOLO} + \mathcal{L}_{Euler}\) where \(\mathcal{L}_{YOLO}\) contains:

1. **Classification loss**: the sigmoid cross entropy of the class conditional probabilities for each class
   \[
   \sum_{i=0}^{s^2} \sum_{c \in \text{classes}} \mathbb{1}_{i}^{\text{obj}} \left[ -p_i^{(c)} \log(\sigma(\hat{p}_i^{(c)})) \right].
   \]

2. **Localization loss**: the error in the 3D predicted location and size of the bounding box
   \[
   \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{i}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] + \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{i}^{\text{obj}} \left[ (w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2 + (d_i - \hat{d}_i)^2 \right].
   \]

3. **Confidence loss**: the sigmoid cross entropy of the class conditional probabilities for each class
   \[
   \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{i}^{\text{obj}} \left[ -c_{ij} \log(\sigma(\hat{c}_{ij})) \right] + \lambda_{\text{noobj}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{i}^{\text{noobj}} \left[ -c_{ij} \log(\sigma(\hat{c}_{ij})) \right].
   \]
Proposed Detection and Tracking Solution

\( L_{Euler} \) is based on a complex angle \(|z|e^{ib_0}\) for box orientation that is always assumed to be located on the unit circle with \(|z| = 1\) and \(|\hat{z}| = 1\):

\[
L_{Euler} = \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left| (e^{ib_0} - e^{i\hat{b}_0})^2 \right|
\]

\[
= \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left[ (t_{re} - \hat{t}_{re})^2 + (t_{im} - \hat{t}_{im})^2 \right].
\]

The final loss adds the classification, localization, confidence and Euler angle loss together into one massive single loss function:

\[
\lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]
\]

\[
+ \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left[ (w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2 + (d_i - \hat{d}_i)^2 \right]
\]

\[
+ \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left[ -C_{ij} \log(\sigma(\hat{C}_{ij})) \right] + \lambda_{\text{noobj}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{noobj} \left[ -C_{ij} \log(\sigma(\hat{C}_{ij})) \right]
\]

\[
+ \sum_{i=0}^{s^2} \sum_{c \in \text{classes}} \mathbb{1}_i^\text{obj} \log(\sigma(\hat{p}_i^c)) \right]
\]

\[
+ \lambda_{\text{coord}} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^\text{obj} \left[ (t_{re} - \hat{t}_{re})^2 + (t_{im} - \hat{t}_{im})^2 \right].
\]

Choice of Anchors

An anchor can be seen as the a priori belief for the size of a detected object. YOLOv2 [51] suggest to use K-Means clustering on the training set bounding boxes to automatically find good priors, making it easier for the network to learn to predict good detections. We, however, chose to use K-Medoids on the Car, Cyclist and Pedestrian labels in the KITTI dataset in an attempt to get results more robust to noise and outliers than K-Means, which has been shown by Jin et al. [84].

For each category, three bounding box centroids are extracted from the dataset (see Figure 3.2). This sums up to nine anchors in total. In the object detector, we have two detection layers, where each comprises three anchors for detection. The choice was made to use the first detection layer for detecting cars, thus the associated anchors were taken from the
3.1 Object Detection

Car centroids. The second detection layer, which is better at more fine grained detection, was chosen to specialize in detecting cyclists and pedestrians, thus the associated anchors were taken from the Cyclist and Pedestrian centroids. Moreover, the most common orientations were also extracted from the dataset for each category. However, to no surprise, the most common orientations for Car, Cyclist, and Pedestrian were either heading away from, or towards the sensing vehicle.
Figure 3.2 **Left**: The result of performing k-medoids ($K = 3$) on labelled bounding boxes in the categories **Car**, **Cyclist**, and **Pedestrian**. **Right**: The extracted anchors from clustering labelled bounding boxes. There are three anchors in total.
3.2 Object Tracking

We propose a multi-threaded solution for object tracking with a novel concurrent data association process. Each tracking component (thread) has an associated Extended Kalman Filter that is used for prediction and estimation of the object state over time. While observations, i.e., detected objects, are confined within a specified space $[-40, 40]$, $[0, 70.4]$ relative to the sensor in the Y and X axes respectively (see the sensor configuration in Figure 4.3 under Section 4.2), the tracking solution keeps track of detected objects outside of this area, given that they once entered it and provided observations to the tracker. The main purpose of the tracker is to predict the movement of objects within the scene and to infer semantic information between states, in a way that it can aid the object detection process.

3.2.1 Multi-threaded Object Tracking

The Multi-threaded Object Tracking (MTKF) solution creates and deletes threads in a continuous flow as objects enter and exit the scene. The main thread is responsible for the creation/deletion process based on the data association method at the thread update step. The thread update step comprises two phases.

At the first phase, for all threads that are running, their corresponding Extended Kalman Filters are being run through their prediction step. Based on the predicted states of each EKF, an association cost is being computed for each observation at the validation step and stored in a particular matrix that is shared between all threads, which we call an Observation-To-Thread Matrix (OTM). The association cost is based on the Mahalanobis distance, which is a $\chi^2$-distributed value computed from the innovation and the innovation co-variance of the EKF. Since innovation is the difference between the predicted observation and each observation evaluated, the Mahalanobis distance is a suitable metric to use for filter-to-observation association. After this, the thread waits for all other threads to complete. The threads are being informed by the main thread when it is time to move on to the next phase of the data association process.

At the second phase, each thread investigates the OTM to find the matching observation before running the update step of the EKF. If a thread has not been associated with an observation for a given period of time, the thread dies and the object stops being tracked. If an observation does not get associated to a thread, a new thread is created to track this observation. A simplified illustration of the thread creation process can be seen in Figure 3.3.
3.2.2 Data Association

As stated previously, the data association is performed in two phases. At the first phase, the Mahalanobis distances between the predicted observation of the EKF and each measured observation \( i \) are calculated within each thread \( j \). The computed distance, also called the association cost \( V_{ij} \), is inserted into the cell \((i,j)\) of the Observation-To-Thread Matrix (see Table 3.4) which works as an observation-to-thread map for threads to find the optimal association for the update step of their corresponding EKFs. This process occurs concurrently with a barrier that synchronizes the threads between the phases.

Table 3.4 The proposed Observation-To-Thread Matrix.

<table>
<thead>
<tr>
<th>THREAD</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( \ldots )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( V_{11} )</td>
<td>( V_{12} )</td>
<td>( V_{13} )</td>
<td>( \ldots )</td>
<td>( V_{1n} )</td>
</tr>
<tr>
<td>2</td>
<td>( V_{21} )</td>
<td>( V_{22} )</td>
<td>( V_{23} )</td>
<td>( \ldots )</td>
<td>( V_{2n} )</td>
</tr>
<tr>
<td>3</td>
<td>( V_{31} )</td>
<td>( V_{32} )</td>
<td>( V_{33} )</td>
<td>( \ldots )</td>
<td>( V_{3n} )</td>
</tr>
<tr>
<td>( m )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( m )</td>
<td>( V_{m1} )</td>
<td>( V_{m2} )</td>
<td>( V_{m3} )</td>
<td>( \ldots )</td>
<td>( V_{mn} )</td>
</tr>
</tbody>
</table>

Once every thread reaches this barrier the second phase starts, which is the process of finding the best thread-to-observation fit via the OTM. This is done by finding the shortest
3.2 Object Tracking

Mahalanobis distance $d^2$ for each thread $j$,

$$i^* = \arg\min_{d^2} \{V_{i,j}\},$$

and then checking that it really is the best overall fit in the matrix by traversing the row of the optimal observation $i^*$,

$$j^* = \arg\min_{d^2} \{V_{i^*,j}\}.$$

If a thread has the lowest association cost to a given observation compared to all other threads, i.e., $j = j^*$, it claims the observation and updates its internal EKF state accordingly. Otherwise it disregards the observation. Worth mentioning is that the second phase is run in parallel between all threads making the data association process extremely fast. An illustration of the data association process can be seen in Figure 3.4.

![Figure 3.4 The data association process of the Object Tracker.](image)

3.2.3 Components of the Extended Kalman Filter

The Extended Kalman Filter was chosen as the method used to track detected objects over time. The motivation is that we are trying to predict the movement of the objects based on non-linear quantities. The detected objects output by the proposed object detector, YOLO++, were considered as observations comprising three parameters: $x, y, \theta$ and $L$. $(x, y)$ correspond to the observed location of the object in the XY plane, $\theta$ the observed angle of the object, and $L$ ($= \ell_r + \ell_f$) the length of the object.
Based on the bicycle model introduced in Chapter 2.3, the state vector was chosen as \([x, y, v, \theta, \alpha, L]^T\), where \(v\) correspond to the linear velocity- and \(\alpha\) the steering angle of the object. Using the bicycle model, the state \(x_k\) may be predicted as

\[
\hat{x}_k = \begin{bmatrix}
x_{k-1} \\
y_{k-1} \\
v_{k-1} \\
\psi_{k-1} \\
\delta_{k-1} \\
L_{k-1}
\end{bmatrix} + \Delta t \begin{bmatrix}
1 & 0 & \cos(\theta_{k-1}) \Delta t & -v_{k-1} & \sin(\theta_{k-1}) \Delta t & 0 & 0 \\
0 & 1 & \sin(\theta_{k-1}) \Delta t & v_{k-1} & \cos(\theta_{k-1}) \Delta t & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & \tan(\alpha_{k-1}) \Delta t & L_{k-1} & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix},
\]

where \(u_k\) and \(\omega_k\) correspond to the linear- and angular velocity of the perceiving vehicle, computed from LiDAR Odometry. Based on the predicted state \(\hat{x}_k\), the state Jacobian \(J^F_k\) can thus be computed as

\[
J^F_k = \begin{bmatrix}
1 & 0 & \cos(\theta_{k-1}) \Delta t & -v_{k-1} & \sin(\theta_{k-1}) \Delta t & 0 & 0 \\
0 & 1 & \sin(\theta_{k-1}) \Delta t & v_{k-1} & \cos(\theta_{k-1}) \Delta t & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & \tan(\alpha_{k-1}) \Delta t & L_{k-1} & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}.
\]

The innovation, i.e., the difference between the predicted observation and the associated observation, is computed like Equation (2.7), but with the innovation of the angle computed as \(\theta \ominus \hat{\theta}\), where \(\ominus\) denotes the angular difference.

The observation Jacobian \(J^H_k\) is chosen to transfer the parameters of the state to the observation space. \(J^H_k\) is computed as

\[
J^H_k = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}.
\]

The update process of the Extended Kalman Filters is illustrated in Figure 3.5, where the LiDAR Odometry is taken as an input during the prediction step and the observations, i.e., the detected objects, are evaluated during the validation step.
3.2 Object Tracking

3.2.4 LiDAR Odometry

To integrate LiDAR Odometry into our Object Tracking solution, we followed what is proposed by Shan et al. [5] to increase real-time performance, which is in this case of outermost priority.

An overview of the LiDAR Odometry pipeline proposed by Shan et al. [5] is shown in Figure 3.6. The segmentation block takes a point cloud from a single scan and projects it onto a range image. The range value \( r_i \) for a particular point \( p_i \) represents the Euclidean distance from the sensor to the corresponding point. A column-wise evaluation of the range image is performed [85], which is considered as a form of ground plane estimation. After the ground points have been extracted, the segmentation is performed on all points except the ground points. An image-based segmentation method [86] is applied which associates points to unique clusters based on the range image. The ground segment is considered as its own cluster. The advantages of performing segmentation of the point cloud into many clusters is that it can improve processing efficiency and accuracy of the feature extraction [5].

Figure 3.6 Overview of the LiDAR Odometry pipeline which comprises a segmentation step and a feature extraction step before being sent into the LiDAR Odometry algorithm.
The feature extraction method is the same as LOAM [8], but instead of extracting features from raw point clouds, the feature extraction is done on the ground points and the segmented points separately. Planar features are extracted from both the ground points and segmented points while edge features are only extracted from segmented points, since the ground is considered as a plane. Feature extraction is done on the range image by evaluating each row, which is associated to one horizontal sweep. Let $S$ be a set that comprises the points located in a single row of the range image. The roughness of a point $p_i \in S$ can be evaluated using [8]

$$c = \frac{1}{|S| \cdot \|r_i\|} \sum_{j \in S, j \neq i} (r_j - r_i),$$  \hspace{1cm} (3.1)

where $r_i$ is the range value of point $p_i$ from the range image. Plane points will result in a low roughness value while edge points will result in a high roughness value. A certain threshold $c_{th}$ determines whether a point is considered a plane point (falls below $c_{th}$) or an edge point (falls above $c_{th}$). Only the most significant features, i.e., the edge points with the highest $c$ and plane points with the lowest $c$ are extracted [8].

Similar to Shan et al. [5], we use a two-step optimization process for pose estimation. In the first step, three of the pose states $[t_z, \theta_{roll}, \theta_{pitch}]$ are obtained from planar features extracted from planar surfaces like the ground. In the second step, the rest of the pose states $[t_x, t_y, \theta_{yaw}]$ are obtained from edge features extracted from a segmented point cloud, which is separated from the ground. The motivation of this procedure is that while $[t_x, t_y, \theta_{yaw}]$ may be obtained from the first optimization step, it is much less accurate due to the lack of a unique solution, especially when taking the ground plane into consideration. Moreover, Shan et al. [5] observed a 35% reduction in computation time at a similar accuracy compared to estimating the pose in one optimization step.

### 3.3 Joint Solution

The final solution takes in point clouds and projects them onto the bird’s eye view plane in the form of 3 channel images comprising density, height, and intensity. The images are then fed into YOLO++, which detects 3D objects. These detections are matched to a unique Extended Kalman Filter (EKF) through the data association process, where each EKF runs on a dedicated thread. Each EKF can then predict its next state making it possible to generate the joint probability distribution for each object. These joint probability distributions can then be discretized and weighted onto a grid map (see the grid coming out of the Object Tracking in Figure 3.7) in order to combine them with the feature maps generated by YOLO++ to achieve a joint solution, which utilizes object tracking to boost the reliability of object detection.
3.3 Joint Solution

The procedure of discretizing a joint probability distribution onto a grid is done in the following manner: Given a predicted state covariance matrix $P_{k|k-1}$, one may find the covariance ellipse by finding the eigenvectors, $v^\top$ and $v^\perp$, that correspond to the largest and the smallest eigenvalues, $\lambda^\top$ and $\lambda^\perp$, of $P_{k|k-1}$, respectively. The angle $\alpha$ of the covariance ellipse can be obtained from $v^\top$ using the arctan2 function, $\alpha = \arctan2(v^\top_y, v^\top_x)$, where $v^\top_y$ and $v^\top_x$ are the y and x components of $v^\top$, respectively. The major and minor axis components, $a$ and $b$, are computed from the eigenvalues $\lambda^\top$ and $\lambda^\perp$ as

$$a = \chi_i \sqrt{\lambda^\top}, \quad b = \chi_i \sqrt{\lambda^\perp},$$

where $\chi_i$ is a $\chi^2$-distributed value which acts as a scaling factor for the covariance ellipse. $\chi_i$ belongs to a set of equidistant values, $\chi = \{\chi_i\}_0^N$, $\chi_0 > \chi_1 > \cdots > \chi_N$, which is used
to assign the appropriate confidences to the corresponding grid cells. The first value in $\chi$, which is $\chi_0$, corresponds to a probability of 99% that the object appears within the covariance ellipse. The closer a grid cell is to the midpoint of the covariance ellipse, the higher the mapped confidence value should be.

Given the angle $\alpha$, the midpoint $(c_x, c_y)$, and the axis components $(a, b)$ of the covariance ellipse, the discretization process is done in the following way: For each value $\chi_i$ found in $\chi$, starting from the beginning, one may find out if a grid cell $G_{ij}$, defined by its center point $(x_{ij}, y_{ij})$, lies within the covariance ellipse by checking if the following inequality is fulfilled:

$$\frac{\cos(\alpha)(x_{ij} - c_x) + \sin(\alpha)(y_{ij} - c_y)^2}{a^2} + \frac{\sin(\alpha)(x_{ij} - c_x) - \cos(\alpha)(y_{ij} - c_y)^2}{b^2} \leq 1.$$ 

(3.2)

If the inequality is fulfilled, we map the $\chi^2$ probability based on $\chi_i$ to the corresponding grid cell. We call the output a prediction probability grid map. See Figure 3.8 for an illustration of the procedure.

For each feature layer in the Object Detector, the prediction probability grid map may be applied during the prediction step of object detection. Since both the objectness and the class probability are being sent through the sigmoid function, one may apply the values of each cell prediction probability grid map to the corresponding cells in the feature layers by taking the maximum probability value and applying it to the objectness $\hat{p}^{(c)}_{ij}$ and class probability $\hat{C}_{ij}$ as

$$\hat{p}^{(c)}_{ij} := \max(G_{ij}, \hat{p}^{(c)}_{ij}), \quad \hat{C}_{ij} := \max(G_{ij}, \hat{C}_{ij}).$$

We have shown that the predicted semantics by the Extended Kalman Filters (EKF) may be combined with the predicted feature maps during object detection. This can potentially be of huge help in the inference process of low-accuracy, high-inference speed networks such as ours. This is due to the fact that each filter knows what type of object it is tracking and where it could potentially be in the next timestep. Consequently, where the network fails to find an object, the discretized and weighted semantics inferred by the EKFs can be used to complement the network’s low accuracy by providing a posteriori information about tracked objects’ locations, sizes and orientations from the previous timestep. These states are derived from the tracked objects through their corresponding probability distributions.
Figure 3.8 Illustration of the prediction probability grid map. We show how we take two covariance ellipses and discretize them onto a probability grid. We show the probability grid map in both a flat- and a 3-dimensional representation to illustrate the shape of the gaussians generated.
Chapter 4

Evaluation Strategy

This chapter covers the methodology used to evaluate- and to produce results from our solutions. We present the metrics used for evaluation, together with the dataset. We introduce the system overview of the whole pipeline, which is based on the Robotics Operating System (ROS) and we present our test cases.

4.1 Evaluation Metrics

Evaluation metrics are a crucial part in an object detection framework in order to understand the accuracy obtained by the system once it is trained.

4.1.1 Mean Average Precision

mAP (mean Average Precision) is the standard metric to measure the accuracy of object detectors [87, 88]. In order to understand this metric we first need to go through three concepts: precision, recall and IoU. Precision measures the accuracy, the percentage of the predictions that are correct, and is described by

\[ P = \frac{TP}{TP + FP}, \]

where \( TP \) is the total number of true positives and \( FP \) is the total number of false positives. Recall measures the percentage of the total correctly classified

\[ R = \frac{TP}{TP + FN}, \]
where $FN$ is the number of false negative. IoU is given by the ratio of the overlapping area between the ground truth box and the predicted box by the area that both boxes cover.

$$IoU = \frac{\text{Area of overlap}}{\text{Area of union}}.$$  

True positives and false positives are defined using IoU with a specific threshold value which determine if a predicted bounding box is TP, FP or FN. Bounding boxes are considered as FP when their IoU is under a certain threshold or if there are duplicated bounding boxes. They are considered FN when the predicted bounding boxes has an IoU over this threshold but has the wrong classification. When the object detection is performed in 3D the IoU is simply extended from area to volume (see figure 4.1).

By collecting all of the predictions on the evaluation set the precision can be plotted against the recall value for each of them obtaining a saw-tooth like pattern. However, to calculate the AP a step function needs to be created out of the the saw-tooth pattern first. This is done by replacing the precision value at each recall level $r$ by the maximum at that specific step [87]:

$$p_{\text{interp}}(r) = \max_{\hat{r} : \hat{r} \geq r} p(\hat{r}).$$

The AP is then calculated by taking the area under the precision-recall curve. This is done by segmenting the recalls evenly to $N$ parts, $\mathcal{R} = \{0, r_2, r_3, ..., r_{N-1}, 1\}$, and computing the AP as [87]:

$$AP = \frac{1}{N} \sum_{r \in \mathcal{R}} p_{\text{interp}}(r).$$

In the object development kit that can be found under KITTI's homepage, it can be seen that the KITTI evaluation benchmark uses $N = 41$. An AP of 1 defines a perfect object detector. mAP for object detection is the average of the AP calculated for all the classes.
**Modified IoU**

Dealing with rotated bounding boxes with their centers located at various different positions, one cannot simply use the conventional way of computing the intersecting area of two rectangular bounding boxes. Instead, one will have to convert the boxes from the \([x_0, y_0, x_1, y_1]\) format into a polygon described by four points, each located at the corners of the bounding box. The corners can then be rotated using \(t_{re}\) and \(t_{im}\) by simple complex multiplication. The area of intersection between two polygons can then be computed using any known method. However, this is not enough since the area of intersection between two rectangular shapes can be the same for different configurations (see Figure 4.2 for an illustrative example).

![Figure 4.2](image)

Figure 4.2 The problem with computing the IoU of two rotatable bounding boxes is that multiple configurations can result in the same IoU.

The solution to this problem is to expand the IoU formula with a scale factor based on the angular error between the true- and the predicted bounding box. The scale factor may be written as

\[
\alpha = \frac{1}{4} \left( (t_{re} - \hat{t}_{re})^2 + (t_{im} - \hat{t}_{im})^2 \right), \quad 0 \leq \alpha \leq 1,
\]

since \(t_{re}\) and \(t_{im}\) – including the predicted parameters – are limited between −1 and 1. As the predicted angle approaches the true angle, \(\alpha\) goes to zero. Since we want the IoU to be higher the closer the predicted angle is to the true angle, the new IoU equation can simply be modified to

\[
\text{IoU} = (1 - \alpha) \frac{\text{Area of overlap}}{\text{Area of union}}.
\]

### 4.1.2 CLEAR MOT & MT/PT/ML

Object tracking should also be evaluated universally in order to compare different methods more easily. The most common metrics used for object tracking evaluation are the CLEAR MOT and the MT/PT/ML metrics. The metrics descriptions are listed below [89, 90]:

\[
\text{IoU} = (1 - \alpha) \frac{\text{Area of overlap}}{\text{Area of union}}.
\]
• **MOTA** (Multiple Object Tracking Accuracy): Measures the tracker’s performance at detecting objects and keeping their trajectories, independent of the precision of object location estimations.

• **MOTP** (Multiple Object Tracking Precision): The ability of the tracker to estimate precise object positions, independent of recognition performance of object configurations, keeping consistent trajectories, etc.

• **MT** (Mostly Tracked): The percentage of ground truth trajectories that are covered by tracker output for more than 80% in length.

• **PT** (Partially Tracked): The percentage of ground truth trajectories that are the remainder of MT and ML. PT is computed as $PT = 1 - MT - ML$.

• **ML** (Mostly Lost): The percentage of ground truth trajectories that are covered by the tracker output for less than 20% in length.

### 4.2 The KITTI Dataset

The KITTI dataset is obtained from the KITTI Vision Benchmark Suite [24], which provides researchers with labelled stereo vision data, point cloud data and odometry data. The KITTI dataset has been recorded from a VW Passat B6 wagon (see Figure 4.3), driving in and around Karlsruhe, Germany with the following sensor setup [24]:

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Sensor Model</th>
<th>No. Mounted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inertial Navigation System (GPS/IMU)</td>
<td>OXTS RT 3003</td>
<td>1</td>
</tr>
<tr>
<td>Laserscanner</td>
<td>Velodyne HDL-64E</td>
<td>1</td>
</tr>
<tr>
<td>Grayscale cameras, 1.4 Megapixels</td>
<td>Point Grey Flea 2 (FL2-14S3M-C)</td>
<td>2</td>
</tr>
<tr>
<td>Color cameras, 1.4 Megapixels</td>
<td>Point Grey Flea 2 (FL2-14S3C-C)</td>
<td>2</td>
</tr>
<tr>
<td>Varifocal lenses, 4-8 mm</td>
<td>Edmund Optics NT59-917</td>
<td>4</td>
</tr>
</tbody>
</table>

The KITTI Vision Benchmark Suite provides a number of benchmarks from a wide variety of categories ranging from Visual Odometry / SLAM and Object Tracking to 3D Object Detection and Semantic Segmentation. The 3D Object Detection Evaluation from 2017 provides a list of methods, published by researchers, which are evaluated against each other (the list is not final and new methods can be submitted during the time of writing this thesis). The top five (5) participants can be seen in Table 4.2. The method performances presented are only based on detection of objects within the category Car.
4.2 The KITTI Dataset

Figure 4.3 The setup used to obtain data for the KITTI dataset, a VW Passat B6 equipped with a variety of sensors [82].

uses IoU (3D bounding box overlap) as evaluation metric. The evaluation runs three different tests on each method: Easy, Moderate, and Hard. The parameters for each test are listed below [82]:

- **Easy**: Min. bounding box height: 40 Px, Max. occlusion level: Fully visible, Max. truncation: 15%.

- **Moderate**: Min. bounding box height: 25 Px, Max. occlusion level: Partly occluded, Max. truncation: 30%.

- **Hard**: Min. bounding box height: 25 Px, Max. occlusion level: Difficult to see, Max. truncation: 50%.

Table 4.2 KITTI 3D Object Detection Evaluation 2017. Top five (5) participants from detection of cars. All of the methods used point cloud data as input [82].

<table>
<thead>
<tr>
<th>Method</th>
<th>Submission</th>
<th>Easy</th>
<th>Moderate</th>
<th>Hard</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD</td>
<td>2019</td>
<td>86.61%</td>
<td><strong>77.63%</strong></td>
<td><strong>76.06%</strong></td>
<td>0.08 s, GPU @ 2.5Ghz</td>
</tr>
<tr>
<td>EMP</td>
<td>2019</td>
<td>87.85%</td>
<td>77.20%</td>
<td>72.28%</td>
<td>0.5 s, GPU @ 2.5Ghz</td>
</tr>
<tr>
<td>Patches</td>
<td>2019</td>
<td><strong>87.87%</strong></td>
<td>77.16%</td>
<td>68.91%</td>
<td>0.15 s, GPU @ 2.0Ghz</td>
</tr>
<tr>
<td>UberATG-MMF</td>
<td>2018</td>
<td>86.81%</td>
<td>76.75%</td>
<td>68.41%</td>
<td>0.08 s, GPU @ 2.5Ghz</td>
</tr>
<tr>
<td>Part-A²</td>
<td>2019</td>
<td>85.41%</td>
<td>76.66%</td>
<td>68.66%</td>
<td>0.08 s, GPU @ 2.5Ghz</td>
</tr>
</tbody>
</table>

Similarly to the 3D Object Detection Evaluation, the Object Tracking Evaluation from 2012 provides a comparison between different object tracking methods. The top five (5)
participants with respect to the evaluation metric MOTA can be seen in Table 4.3. The method performances presented are only based on tracking of objects within the category Car. Since there is no single ranking criterion, the methods can not be ranked from best to worst overall. The benchmark uses the common MOT metrics CLEAR MOT [89] and MT/PT/ML [90] for evaluation.

Table 4.3 KITTI Object Tracking Evaluation 2012. Top five (5) participants from detection of cars, with respect to MOTA. The methods use different sources of input, they are not be listed due to the space limit [82].

<table>
<thead>
<tr>
<th>Method</th>
<th>Submission</th>
<th>MOTA</th>
<th>MOTP</th>
<th>MT</th>
<th>ML</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH-TRK</td>
<td>2018</td>
<td>90.77%</td>
<td>83.28%</td>
<td>80.62%</td>
<td>3.38%</td>
<td>0.1s, 1 Core @ 2.5Ghz</td>
</tr>
<tr>
<td>IWNCC</td>
<td>2019</td>
<td>86.86%</td>
<td>85.39%</td>
<td>75.38%</td>
<td>2.92%</td>
<td>0.01s, 1 Core @ 2.5Ghz</td>
</tr>
<tr>
<td>TuSimple</td>
<td>2016</td>
<td>86.62%</td>
<td>83.97%</td>
<td>72.46%</td>
<td>6.77%</td>
<td>0.6s, 1 Core @ 2.0Ghz</td>
</tr>
<tr>
<td>DBTO3D</td>
<td>2018</td>
<td>84.82%</td>
<td>85.43%</td>
<td>68.31%</td>
<td>4.31%</td>
<td>0.06s, GPU @ &gt;3.5Ghz</td>
</tr>
<tr>
<td>3DT</td>
<td>2018</td>
<td>84.52%</td>
<td>85.64%</td>
<td>73.38%</td>
<td>2.77%</td>
<td>0.02s, GPU</td>
</tr>
</tbody>
</table>

The 3D object detector proposed is trained and validated on 7481 point clouds with their corresponding ground truth labels. The dataset used for training and validation comprises labelled objects within the categories Car, Van, Truck, Pedestrian, Person (sitting), Cyclist, Tram and Misc (trailers, segways, etc.) [23]. However, the proposed detector was limited to detecting objects from the categories Car, Pedestrian and Cyclist.

4.3 Experimental Setup

Our joint solution was connected using Robotics Operating System (ROS) [91]. ROS is an open-source meta-operating system, which is developed specifically for robotics purposes. It enables different solutions implemented as modules to be connected, reused and developed upon with ease. Furthermore, it introduces the notion of Nodes, which can communicate through topics in a peer-to-peer network called a ROS graph. While ROS is not inherently a real-time framework, it is possible to integrate ROS with real-time code, which makes ROS a suitable choice of framework to base our joint solution upon.

Each major subsystem is represented by a ROS node (see Figure 4.4). The input to the system is a time stamped point cloud, which triggers both the object detection and tracking subsystems represented by the upper two central blocks and lower two central blocks respectively. The results gathered from the two subsystems, explained in the previous subsections under this same chapter are sent to the Visualization node, which is in charge of plotting everything up in RViz, a high performance 3D visualization tool for ROS. What is
plotted up synchronously in RViz is the raw point cloud, the detected objects, their headings, their estimated paths, and a label for each object where its category is specified and its ID in the form of an integer. The object detection semantics are aided by the discretization of the joint probability distributions obtained from the Extended Kalman Filters.

**System Overview**

The quantitative results of the object detection solution was evaluated using 20% of the training set from the KITTI 3D object detection dataset. The reasons for this are explained under Data Splitting in Chapter 3.

The scenario, in which the Object Tracker is evaluated upon, is related to the problem formulation in Chapter 1. We mentioned a problem scenario where a self-driving car was trying to cross an intersection in heavy traffic safely and efficiently. In order to relate to this problem scenario, we chose sequence 2011_09_26_drive_0057 from the KITTI dataset [23], which places the ADS in a similar situation. The qualitative results of the multiple object tracking solution were based on this scenario.

Figure 4.4 Overview of connected ROS nodes.
Chapter 5

Results

This chapter covers the qualitative and quantitative results obtained for both object detection and object tracking. Both detection and tracking were done on cars only due to the time limit.

5.1 Detection Performance

As stated in previous chapters it is highly unclear what architecture and hyperparameter configuration leads to good object detection performances as the behavior of the CNN learning process is difficult to anticipate. In order to mitigate this fact, by optimizing the hyperparameters of the object detection model, a low-resolution custom dataset was generated, which tries to mimic the general shape of the cars found in the projected point clouds. The dataset is comprised of 5000 images of size 200 × 200 where each of them contain four L shaped objects rotated with uniform distributed angles, which are positioned sparsely throughout the images. Each training session was run at 5000 epochs with varying adjustments in both hyperparameters and model configuration. Various experiments and their outcome are presented in Table 5.1. Some of the results from the best configuration found during hyperparameter optimization are presented in Figure 5.1. These hyperparameters were then used in YOLO++ with the aim of obtaining good initial results on the projected point clouds as we initially just aimed at detecting cars. We evaluate our model on the car class only, on the three difficulty levels (easy, moderate and hard) described under Section 4.2, and compare it with several state-of-the-art BEV object detectors with different modality methods (see Table 5.2). All training and evaluation was performed on an Nvidia P100 GPU.

Figure 5.2 shows some results of detected cars in projected point clouds. The point clouds presented in Figure 5.2 are taken from the training set to show that the network can in fact learn to re-locate these objects again after training. Running the Object Detector on the test
Table 5.1 Results of hyperparameter optimization using our simplified dataset. We analyze the performance gain or loss of tweaking batch size, the number of epochs, learning rate, usage of angle anchors, and usage of the tanh activation function for angle prediction.

<table>
<thead>
<tr>
<th>Experiment no.</th>
<th>Batch Size</th>
<th>Epochs</th>
<th>Learning Rate</th>
<th>Angle Anchors</th>
<th>tanh</th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>3000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>58%</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>5000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>70%</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>5000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>70%</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>5000</td>
<td>0.0005</td>
<td>x</td>
<td>x</td>
<td>67%</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>5000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>65%</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>5000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>62%</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>5000</td>
<td>0.001</td>
<td>x</td>
<td>x</td>
<td>67%</td>
</tr>
</tbody>
</table>

Figure 5.1 The model was trained on a custom, simplified dataset in order to optimize the hyperparameters.

Table 5.2 Class-wise average precision and inference speed on BEV of different methods based on their modality. The average precision was computed on the three KITTI difficulty levels.

<table>
<thead>
<tr>
<th>Method</th>
<th>Modality</th>
<th>FPS</th>
<th>Car</th>
<th>Pedestrian</th>
<th>Cyclist</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Easy</td>
<td>Mod.</td>
<td>Hard</td>
<td>Easy</td>
</tr>
<tr>
<td>MV3D</td>
<td>Lidar+Mono</td>
<td>2.8</td>
<td>86.02</td>
<td>76.90</td>
<td>68.49</td>
</tr>
<tr>
<td>F-PointNet</td>
<td>Lidar+Mono</td>
<td>5.9</td>
<td>88.70</td>
<td>84.00</td>
<td>75.33</td>
</tr>
<tr>
<td>AVOD</td>
<td>Lidar+Mono</td>
<td>12.5</td>
<td>86.80</td>
<td>85.44</td>
<td>77.73</td>
</tr>
<tr>
<td>AVOD-FPN</td>
<td>Lidar+Mono</td>
<td>10.0</td>
<td>88.53</td>
<td>83.79</td>
<td>77.90</td>
</tr>
<tr>
<td>VoxelNet</td>
<td>Lidar</td>
<td>4.3</td>
<td>89.35</td>
<td>79.26</td>
<td>77.39</td>
</tr>
<tr>
<td>Complex-YOLO</td>
<td>Lidar</td>
<td>50.4</td>
<td>85.89</td>
<td>77.40</td>
<td>77.33</td>
</tr>
<tr>
<td>YOLO++</td>
<td>Lidar</td>
<td>20.0</td>
<td>34.01</td>
<td>29.31</td>
<td>21.98</td>
</tr>
</tbody>
</table>

set, however, yields much worse results. The networks fails at generalization, which can be further observed in Figure 5.3.

The training process was run during enough epochs (4500) to have the loss on the validation set converge and without over-training the network on the training data. Based on Figure 5.3, one can observe that the location- and angle loss on the validation set does not converge to optimal values whereas the total loss is influenced by this.
Figure 5.2 The Object Detector detecting cars in a bird’s eye view projection of a point cloud. The point cloud data is from the KITTI Dataset [82].
Figure 5.3 YOLO++ training loss. We present the loss of both the training set and the validation set during 4500 epochs of training. At the top we have the location- and size loss. In the middle we have the confidence- (objectness) and class loss. At the bottom left we have the angle loss and at the bottom right we have the total loss. We draw a trend line on the validation loss to help the reader understand the data.
5.2 Tracking Performance

The tracking solution was tested on a specific sequence of the KITTI dataset that depicts a crossing with limited visibility and multiple object occlusion moments. This specific sequence was chosen because it covers most of the cases that are relevant for checking the reliability and robustness of our solution. The results are promising as can be seen in Figure 5.4. It has to be clarified that we assume that all the cars within the LiDAR 3D space are driving on the same 2-dimensional plane, omitting the z-axis, since the object detector does not regress locations of bounding boxes in this direction.

![Image of tracking multiple cars in a point cloud](image)

Figure 5.4 The Object Tracker tracking multiple cars in a point cloud. Each car gets a unique label that is inherited between states. The car surrounded by a circle (car-7) is the one presented in Figure 5.5. The point cloud data is from the KITTI Dataset [23], sequence 2011_09_26_drive_0057. The corresponding camera view [23] is shown as a reference.

The heading of all objects is predicted with very good results (see Figure 5.5) with an approximate maximum deviation of 0.1 degrees from the ground truth. The performance is not surprising since the object locations comes from the ground truth data of sequence 2011_09_26_drive_0057 from the KITTI Dataset [23]. The position estimation of the
objects with their dedicated Extended Kalman Filter is also good as we were still working on the ground truth data. Hence, there is no wonder that the estimations are so accurate compared to the ground truth values. However, what can be pointed out is that the position deviation from the ground truth seems to increase with the distance.

![Figure 5.5 The result of tracking a car turning right. The x and y coordinates relative to the sensor and the heading of the tracked car together with their $2\sigma$ confidence intervals are logged and displayed in the figure. The point cloud data is from the KITTI Dataset [82].](image)

The results presented in Figure 5.6 show a strong correlation between the acceleration and the Mean Squared Error (MSE) of the innovation. This can be seen by superposing the velocity plot on the innovation MSE plot; where the error seem to be proportional to the slope of the velocity curves for each car. Heading and location of the tracked object seem to have minor to no effect on the innovation MSE. The innovation MSE is computed by taking the sum of all components of the innovation vector squared. This gives a general idea of how the prediction error evolves based on particular events.

In Figure 5.6 we can also see how the threads’ creation/deletion process works as well. When a specific thread gets no observation as a result of an unsuccessful association process,
5.2 Tracking Performance

Figure 5.6 Results of multiple tracked objects. At the top we have x- and y locations, in the middle we have heading and steering angle, to the bottom left is linear velocity and to bottom right is innovation Mean Squared Error (MSE). The innovation MSE is computed by taking the sum of all components of the innovation vector squared. This gives a general idea of how the prediction error evolves based on particular events.

the object state is frozen giving linear predictions for that time step. If this thread to observation pairing is unsuccessful 10 time steps in a row (see the different state values
Results

predicted for car-7 from the seven second mark to eight second mark in Figure 5.6), the thread and its associated state model is killed. This gives the chance for another thread to claim that same object as can be seen in Figure 5.7. In this figure a car gets assigned the label car-20 by the thread that claimed its existence, but gets lost due to the car’s high speed resulting in a stalled state model. The thread associated to car-20 gets killed after 10 unsuccessful associations and a new thread emerges that claims that same object, tracking the same car but under the label car-21 instead. As the car enters the scene, it is already travelling at about 50 km/h. What can be seen in Figure 5.8a, is that the initially associated Extended Kalman Filter (EKF) can not keep up to this sudden change in velocity (the initial estimation of the velocity in the EKF is zero). Thus, the EKF loses track of the car since the subsequent observations lie outside of the association zone around the predicted state of the EKF. This is particularly noticeable by looking at Figure 5.8b, where the innovation MSE spikes at the moment of failure.

Figure 5.7 Erroneous tracking of a car traveling too fast. Initially, the car gets assigned the label car-20, but gets lost due to the car travelling too fast. The thread associated to car-20 dies and a new thread emerges, tracking the same car but under the label car-21 instead.

The tracker is synced to run a single iteration when a new complete point cloud is generated, which is every 100 ms. We show that this is the case by plotting the overall tracker run time for each iteration. The execution times are of course not exactly 100 ms as we are running our tracker system on a Linux machine with Ubuntu installed which implies limited thread execution control (see Figure 5.9b). As can be seen in Figure 5.9a the Extended Kalman Filter updates are performed at a maximum speed of around 0.03 ms when the system is maximized with an equal amount of threads to the number of cores of our system. We ran the Object Tracker on an 8 core Intel® CPU at 1.90 GHz.
5.2 Tracking Performance

Figure 5.8 Velocity and innovation MSE from erroneous tracking of a car travelling at high-speed.

Figure 5.9 Object Tracker run time performance. The number of threads at each iteration is represented by a blue dot. While there is a correlation between the number of threads and the individual thread execution time, the impact the number of threads has on the overall run time is next to none. The experiment was done on an 8 core Intel® CPU at 1.90 GHz.
Chapter 6

Discussion

This chapter covers the individual discussion around the thesis. The discussion covers a result analysis, what problems were encountered during the project and how they were resolved, the social and ethical aspects regarding the problem area and how this project is affected by them, future work, and conclusions based on the initial problem statement.

6.1 Result Analysis

The Object Tracker achieves a run time performance of 33 state updates per second when the amount of threads is the same as the amount of cores in the CPU. There is a correlation between mean thread execution time and the amount of running EKF-threads, which makes sense since there is a barrier during the thread execution in which all threads have to be ready. Thus, each thread has to wait for all other threads to complete in order to continue. This barrier divides the first and the second phase of the data association process.

The overall tracker run time is consistent, meaning that it does not increase significantly as the number of threads increases. We achieve an overall run time around 10 frames per second (FPS), which meets the real-time requirement. The fact that the frame rate is not always the same, but fluctuates around 10 frames per second, is that the operating system that the tracker was run on does not set the priority on the threads at all times. Thus, the overall tracker run time will fluctuate.

The accuracy of the object tracking seem promising based on the qualitative results, even through the tracker was only tested on the ground truth object detection data and only on one sequence of the KITTI dataset [23]. Hence, there is no wonder that the estimations are so accurate compared to the ground truth values. However, what has been noticed is that sometimes the tracking still fails, particularly on cars that enter the scene at a too high
velocity (see Figure 5.7). This is due to the fact that the EKFs can not be trimmed to track fast moving objects while also keeping high precision tracking for slow moving objects. A solution to this would be to use an online trimming technique. Another solution would be to initiate the EKFs using a model-free approach for estimation of velocity, since velocity can not in principle be estimated without enough history.

Unfortunately, no quantitative results were gathered for the object tracker due to the insufficient results of the Object Detector. The Object Detector, while being fast (20 FPS), performed much worse on the BEV evaluation (see Table 4.2) than the state-of-the-art methods based on the same modality. This is due to a wide variety of reasons. Firstly, and most importantly, the data representation was inadequate for training the neural network to give accurate bounding box predictions. The data representation used – projection of point clouds into an RGB map comprising height, intensity and density – is very complex to train upon and the architecture chosen, which was based on Tiny-YOLOv3, is insufficient for this type of data. Secondly, no other data splitting portions were used except for a split of 80% for training and 20% for validation. Some related work perform a split of 50/50, which in theory is inferior to the split used in this project, but should have been tested upon nevertheless. Thirdly, the resolution of the feature maps for each detection layer play an important role in the balance between inference time and accuracy. We chose scales, which down-samples the input image size by 16 and 8 times for each respective detection layer. Down-sampling the input even more before it reaches the feature maps speeds up the inference time, but gives less predictions. In this type of architecture, less predictions means less accuracy, since the Feature Extractor Network (FEN) used is not complex enough (see Table 3.1). It remains to be seen how the tracking behaves when the observations come from our neural network based object detector. What is clear is that the precision of tracking will deteriorate due to the low object detection accuracy.

From the hyperparameter optimization, by comparing the mAP resulting from each experiment, we can see that the usage of angle anchors and the activation function tanh gives better results in general. This is mostly due to the fact that the network will only have to learn to regress a complex rotation vector within a confined space. A batch size of 16 seems to be sufficient for this particular dataset, since increasing it to 32 does not give any significant improvements in performance. The number of epochs on the other hand does have a significant impact on the outcome of the training process. What has been concluded is that one has to find a "sweet spot" of the number of epochs to use for training to avoid both undertraining and overtraining of the neural network. The learning rate has to be chosen in such a way that convergence is still maintained, but is not too slow.
6.2 Problems

The joint solution has not been tested on a system with enough computational resources, but is ready for deployment. Using the prediction probability grid map obtained from the joint probability distribution discretization, one may boost the confidence of the detection layers in the Object Detector by a large margin. Consequently, where the Object Detector fails to find an object, the semantics inferred by the EKFs may be used to complement the network’s low accuracy by providing \textit{a posteriori} information about tracked objects’ locations, sizes and orientations from the previous timestep. However, heavy testing would be needed in order to confirm this. Even further testing would be needed in order to come up with an efficient and synchronized architecture for this to work.

6.2 Problems

Since the results of object detection were inadequate, there was not enough motivation for the Object Detector and the Object Tracker to be combined. Thus, no quantitative analysis could be done using the CLEAR MOT and MT/PT/ML metrics, even though such an evaluation was prepared to be run on the Object Tracker.

Replication of Complex-YOLO [54] was aimed at initially since their architecture performs a low amount of predictions, but with high performance, which makes it able to run at 50 FPS with state-of-the-art performance on the KITTI benchmark [24]. The replication attempt failed since no source code was available and some details were not specified in their paper. The attempted replication of the Complex-YOLO solution gave worse predictions than the YOLO++ FEN architecture (see Table 3.1), which is not surprising since the feature maps has a higher resolution for this architecture, compared to Complex-YOLO.

The KITTI dataset contains a couple of problems in itself. The dataset was recorded in ideal scenarios, meaning that it was recorded on a sunny day with minimal amounts of traffic. This means that whatever method is evaluated on the KITTI benchmark, it will not mean that that method is good in real life conditions. There has to be a dataset including snow, rain, night, etc. in order to properly evaluate and validate methods that should have any use in Autonomous Driving for consumer use. Moreover, the KITTI dataset includes mostly cars, a whopping 75% of labelled objects belong to the Car category [54]. This affects the training done on other categories since there is not enough data to be able to generalize sufficiently.


6.3 Social and Ethical Aspects

The example scenario covered as a motivating example in this thesis was if a self-driving car is trying to cross an intersection in heavy traffic. The self-driving car would have to navigate through the intersection efficiently and safely while predicting the surrounding objects’ individual movement over time. Though, this is not enough when the social and ethical questions come into play. Some objects may not be as important to keep track of as others. A child, for example, is of the outermost importance to keep track of. Not only because children are unpredictable in their behavior, but also because they are fragile and have to be protected. The ethics setting to use for this scenario could be very different depending on which society it is implemented in. It is therefore important that the ethics are defined carefully as to not conflict with different societies and beliefs.

When a human is able to take control of the vehicle at command, which defines a level 4 system of autonomous driving, some of the ethics are relieved from the system and placed upon the driver. This has to be the case because the ethics settings of a fully autonomous vehicle will be far from defined and agreed upon when the first level 5 ADS is put on the market. Level 5 autonomy, i.e., full automation puts a lot of questions about how society should handle it. By this time, the car is no longer a car, but a robot able to make own decisions based on its ethics settings. Societies will have to unite to define the ethics settings in a political aspect, letting societies vote upon agreed ethics to be applied to fully autonomous vehicles. The moral machine experiment [22] gives some insight into how ethical preference can vary throughout different age groups, education levels, and culture.

Instead of assigning one unique class to each object, YOLOv3 assumes that classes are mutually inclusive, meaning that if an object scores high on both the class Woman and Person, the object is assigned both classes, instead of just one. This is an important factor when ethical choices come into play. Take the helmet problem from Chapter 1 for example, we had a choice between swerving into a motorcyclist who is wearing a helmet or a motorcyclist that is not. With an assumption of mutually exclusive classes, only one of the many applicable classes would be assigned to each motorcyclist. In this case, the obvious choices would be Helmet or No helmet. But what if one of the motorcyclists is pregnant? Let us say that the YOLOv3 could learn to predict if a human is pregnant or not based on their body shape. Then it would be able to assign the class Pregnant to the corresponding motorcyclist. Now a whole other set of ethical questions come into play. Should the car swerve into the motorcyclist who is wearing a helmet or the motorcyclist who is not wearing a helmet but is pregnant? To assume multiple classes for a given object is vital in order to make the proper value judgements.
6.4 Future Work

It is the novel way of performing concurrent data association presented in this thesis that enables us to achieve real-time performance on multiple object tracking. Implementing the Object Tracker on a GPU would be a next step in order to increase the performance of the object tracking. Instead of running each EKF in a thread, they may be run in parallel through a GPU graph, which should increase the run time speed even further.

The LiDAR Odometry is not complete for absolute tracking, instead we use it for updating the movement of objects relative to the sensor to enable predictions to be made in directions perpendicular to the direction of motion of the individual objects. This can be done since we can predict information regarding the states of tracked objects using the LiDAR Odometry as an input to the EKFs. In future works, the LiDAR Odometry should be fully integrated to allow for tracking relative to an absolute coordinate system instead of relative to the sensing vehicle.

Both detection and tracking were done on cars only due to the time limit. The desire is to perform detection and tracking on pedestrians and cyclists also, but that requires a good data representation and a good dataset to base the training upon. The tracking of cars was done using the bicycle model. Further research could go into whether there is a better model that may be used without sacrificing too much of the real-time capabilities. We still believe that tracking of pedestrians should make use of a model-free method in order to give reliable predictions, since the movement pattern of pedestrians can be unpredictable.

The object detection solution was only evaluated based on the bird’s eye view perspective. It remains to be seen how the method performs in the full 3D perspective. Since our solution involves height regression of 3D objects, it is possible to evaluate it in full 3D. There was simply not enough time in the project to perform the full 3D evaluation on our method.

The object detection and object tracking were unfortunately not connected in this work in order to test the joint solution together with inferred semantics. This would be the next step in this project, but requires adequate computational resources to be feasible to test.

During the writing of this thesis, additional works have been published regarding joint solutions of both object detection and object tracking. The successor of Complex-YOLO [54], called Complexer-YOLO [92], performs both object detection and object tracking on semantic point clouds in real-time. Instead of projecting point clouds to an RGB image consisting of height, intensity and density, semantic segmentation is performed on camera images associated with the point clouds and is then mapped onto the point cloud in a voxel based representation. Multiple object tracking is done using the Labeled Multi-Bernoulli
Filter. We believe this approach is the way to go for future research within the field of DATMO.

Another recently published work tries to bridge the gap between LiDAR based- and camera based 3D object detection for autonomous driving. Wang et al. [93] proposes a Pseudo-LiDAR solution from visual depth estimation using stereo cameras. The Pseudo-LiDAR representation can be used by already established LiDAR based 3D object detection methods and provides a similar performance compared to the use of point clouds from an actual LiDAR sensor. The author believes that it is the data representation that is the key to bridging the gap between the performance of camera based solutions versus LiDAR based solutions for object detection. This is something that should be further investigated since cameras are cheaper than LiDARs in general, even though LiDARs are beginning to become cheaper on the market. Though, in a safety aspect, LiDARs are probably better since Pseudo-LiDAR involves an additional software block in the detection pipeline, which is prone to bugs and is thus the more unreliable alternative.

6.5 Conclusions

The aim was to develop a solution for simultaneous detection and tracking of moving objects in real-time based only on LiDAR data. While the object detection solution needs more work to provide reliable results, the object tracking solution is viable for safe and precise motion planning in real-time. With the object detector tuned to provide a good performance, it would be safe to say that the joint solution would also be viable for use in motion planning.

The joint solution was not tested in this project due to the time limit. Based on the method of joint probability distribution discretization, the semantic information could be inferred between states to aid the object detection process using object tracking. By combining the predictions based on the Extended Kalman Filters from the Object Tracker and the Feature Maps from the Object Detector, one could potentially boost the reliability of objectness and class predictions of the Object Detector. However, more testing have to be done in order to argue whether the proposed method of inferring semantic information is feasible or not.
References


[84] Xin Jin and Jiawei Han. K-Medoids Clustering, pages 564–565. Springer US, Boston, MA, 2010.


