UNIVERSITÉ DU LUXEMBOURG

Faculty of Science, Technology and Medicine

LiDAR-Camera Fusion for 3D Object Detection in Autonomous Driving Systems

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Tim Kieffer

Supervisor:
Prof. Dr. Raphaël Frank

Reviewer:
Prof. Dr. Radu State

Advisor:
Mehdi Testouri

August 2024

Abstract

This thesis explores the development of a resource-efficient model for 3D object detection utilizing LiDAR and camera sensors, tailored for autonomous vehicles with limited computational resources. Addressing the challenge of deploying models in constrained environments, the proposed model leverages techniques such as segmentation, 3D point filtering, and Principal Component Analysis (PCA) for bounding box creation to enhance detection accuracy. Evaluated on the KITTI and A2D2 datasets, the model demonstrates convincing performance with minimal GPU memory usage, achieving a balance between accuracy and efficiency. This thesis highlights the potential for deploying the model in real-time applications, such as autonomous driving and robotics, where computational efficiency is crucial. Despite its performance, the model faces challenges in generalizing across diverse environments, particularly where objects are occluded or only visible from one side, necessitating further investigation and refinement. Future work aims to implement the model in the RoboCar testing vehicle to validate its real-world applicability and improve its robustness against varying conditions and object classes.

Acknowledgements

I would like to express my deepest appreciation to those who supported and guided me throughout the course of this thesis.

First, I would like to thank my supervisor, Prof. Dr. Raphaël Frank, for their valuable guidance and advice during our monthly meetings as well as for the introduction to this interesting topic of sensor fusion in autonomous driving systems.

I am also deeply grateful to my advisor, Mehdi Testouri, for their mentorship, guidance and support during our weekly meetings. Their expertise and insightful feedback guided me through this project and helped me overcome challenges.

Additionally, I am grateful to my family and friends for their consistent support and understanding throughout this journey.

Thank you all for your contributions to this project. Without your support, this work would not have been possible.

Contents

1 Introduction **1**

2 Related Work **3**
2.1 DeepFusion . . . 3
2.2 TransFusion . . . 4
2.3 BEVFusion . . . 4
2.4 Robust BEVFusion . . . 5
2.5 MSMDFusion . . . 6
2.6 Cross Model Transformer . . . 7
2.7 LoGoNet . . . 7

3 Terminology **9**
3.1 Calibration . . . 9
3.1.1 LiDAR-Camera Transformation . . . 9
3.1.2 Extrinsic Calibration Methods . . . 11
Manual Target-based Calibration . . . 11
Automatic Target-based Calibration . . . 11
Manual Targetless Calibration . . . 11
Automatic Targetless Calibration . . . 11
3.2 Object Detection . . . 12
3.2.1 Traditional Methods . . . 12
3.2.2 Convolutional Neural Networks (CNN) . . . 12
Region-Based Detectors . . . 12
Regression-Based Detectors . . . 14
3.2.3 Performance Metrics . . . 14
Intersection over Union . . . 15
Precision & Recall . . . 15
Average Precision . . . 16
Mean Average Precision . . . 17
3.3 YOLO . . . 17
3.3.1 Original YOLO model . . . 18
3.3.2 Evolution from YOLOv2 to YOLOv7 . . . 18
3.3.3 YOLOv8 . . . 19
Architecture . . . 20
Training . . . 20
Segmentation Model . . . 21
Variants & Performance . . . 22
3.4 Clustering . . . 22
3.4.1 Overview . . . 22
3.4.2 Distance & Similarity . . . 23

3.4.3 Evaluation Indicator . . . 23
3.4.4 KMeans . . . 24
3.4.5 DBSCAN . . . 25

4 Methodology **27**
4.1 Overview . . . 27
4.2 Dataset Presentation . . . 28
4.3 Implementation . . . 30
4.3.1 Data Pre-Processing . . . 30
Filter Point Cloud . . . 30
Segmentation Mask Erosion . . . 32
3D Point Filtering . . . 33
4.3.2 Calibration . . . 34
4.3.3 Fusion . . . 37
4.3.4 3D Bounding Box Creation . . . 39
4.3.5 Tracking . . . 41

5 Evaluation **43**
5.1 Datasets . . . 43
5.1.1 KITTI Dataset . . . 43
KITTI Evaluation . . . 43
5.1.2 Audi Autonomous Driving Dataset . . . 45
5.2 Model Configuration . . . 46
5.2.1 Model Evaluation . . . 47
5.2.2 Data Collection . . . 47
5.3 Model Evolution . . . 49
5.3.1 Baseline Model . . . 50
5.3.2 Modification 1: Include Segmentation . . . 51
5.3.3 Modification 2: Erode Segmentation Mask . . . 53
5.3.4 Modification 3: Apply 3D Point Filtering . . . 54
Distance-Based Filtering . . . 55
Standard Deviation Filtering . . . 56
K-Means Filtering . . . 56
Depth-Based Filtering . . . 57
5.3.5 Modification 4: Implement PCA for 3D Bounding Box Creation . . . 58
5.3.6 Segmentation Refiner . . . 59
5.3.7 Tracking . . . 60
5.3.8 Performance Comparison . . . 61
5.4 Comparison of different YOLO model sizes . . . 64
5.5 Results . . . 66
5.5.1 KITTI Dataset . . . 66
5.5.2 A2D2 Dataset . . . 68
5.5.3 Examples . . . 68
5.5.4 Worth Noting . . . 73
5.6 Comparison with KITTI State-of-the-Art . . . 75
5.7 RoboCar Integration . . . 75
5.8 Future Work . . . 77

6 Conclusion **79**

A Appendix **81**

List of Tables

5.1 KITTI difficulty levels criteria 45
5.2 Performance comparison of different model evolutions 62
5.3 Performance comparison of different model sizes for final model evolution 65
5.4 IoU Scores of Proposed Model on KITTI Dataset 67
5.5 IoU Scores of Proposed Model on A2D2 Dataset 68

List of Figures

3.1 Transformation between LiDAR and camera sensors [1] 10
3.2 RCNN Overview [2] 13
3.3 Intersection over Union equation [3] 15
3.4 Precision-Recall Curves [4] 16
3.5 Illustration of YOLOv1 output prediction [5] 18
3.6 Comparison of different vision tasks [6] 21
3.7 Comparison of YOLO model sizes [7] 22
3.8 KMeans Clustering Illustration 24
3.9 DBSCAN Clustering Illustration 26

4.1 Detection Pipeline Overview 27
4.2 KITTI Sensor Setup Top View [8] 28
4.3 Example Point Cloud and Image 29
4.4 Filtered Point Cloud (BEV) 30
4.5 Filtered Point Cloud 31
4.6 Illustration of Segmentation Mask Erosion 32
4.7 Illustration of filtering 3D points after fusion 33
4.8 Illustration of each fusion step 38
4.9 Illustration of different 3D bounding boxes 40
4.10 Illustration of Object Tracking 42

5.1 A2D2 Sensor Setup [9] 45
5.2 Example of fused image from A2D2 dataset 46
5.3 Intersection-over-Union of runtime parameter combinations 48
5.4 Original Image to demonstrate model evolution 50
5.5 2D Bounding Boxes from YOLOv8 50
5.6 Fused Image from Initial Model 50
5.7 Projected 3D Bounding Boxes from Initial Model 51
5.8 BEV Baseline Model 51
5.9 Segmentation Masks from YOLOv8 51
5.10 Fused Image after adding Segmentation 52
5.11 Projected 3D Bounding Boxes after adding Segmentation 52
5.12 BEV Segmentation 52
5.13 Eroded Segmentation Masks (erosion_factor = 20) 53
5.14 Fused Image with Eroded Segmentation 53
5.15 Projected 3D Bounding Boxes with Eroded Segmentation 54
5.16 BEV Erosion 54
5.17 Projected 3D Bounding Boxes with Distance-Based Filtering 55
5.18 BEV Distance Filtering 55
5.19 Projected 3D Bounding Boxes with Standard Deviation Filtering 56

5.20 BEV Standard Deviation Filtering 56
5.21 Projected 3D Bounding Boxes with KMeans Filtering 57
5.22 BEV KMeans Filtering 57
5.23 BEV Depth Filtering 57
5.24 Projected 3D Bounding Boxes with Depth-Based Filtering 58
5.25 Projected 3D Bounding Boxes with PCA 58
5.26 Projected 3D Bounding Boxes with PCA (no z-axis rotation) 58
5.27 BEV PCA with and without z-axis rotation 59
5.28 FastSAM Segmentation Masks 60
5.29 Projected 3D Bounding Boxes when using FastSAM 60
5.30 BEV FastSAM 60
5.31 Projected 3D Bounding Boxes with Tracking 61
5.32 BEV Tracking 61
5.33 Model Evolution Steps - Best and Average IoU 63
5.34 Model Evolution Steps - Inference Time and GPU Memory Consumption 64
5.35 Model Sizes - Best and Average IoU 66
5.36 Model Sizes - Inference Time and GPU Memory Consumption 67
5.37 Examples - Good Car Prediction 69
5.38 Example - Bounding Box Accuracy when using PCA 69
5.39 Examples - Bad Car Prediction 70
5.40 Examples - Good Pedestrian Prediction 71
5.41 Examples - Bad Pedestrian Prediction 72
5.42 Examples - Cyclist Prediction 73
5.43 Difference between bounding box construction methods 74

A.1 YOLOv8 Architecture [10] 81
A.2 Comparison of Intersection over Union for KITTI and A2D2 testing datasets 82
A.3 Comparison of Inference Time & GPU Memory Consumption for KITTI and A2D2 testing datasets 82
A.4 Example 1 - Bad Car Prediction 3D Visualization 83

1 Introduction

Autonomous Driving Systems (ADS) have rapidly evolved over the past two decades due to significant advancements in deep learning, extensive expertise in vehicle dynamics, and the availability of new and improved sensors such as LiDAR [11]. The growing public interest and market potential for autonomous vehicles (AVs) have further fueled research and development in this area. A critical aspect of ADS is object detection, which is essential for providing a clear and comprehensive perception of the vehicle's surroundings. This perception capability is crucial for the safe and accurate operation of autonomous vehicles, as it directly influences their actions and responses to the environment.

Object detection in ADS involves the precise interaction of various algorithms and methods to ensure the system's reliability. Recent research has shown that combining data from multiple sensors, particularly cameras and LiDARs, can significantly enhance detection performance. Cameras offer rich visual information, such as object textures, while LiDARs provide accurate depth data, though they lack detailed visual information. The fusion of these two sensor types leverages their complementary strengths, resulting in robust and accurate 3D object detection, which is essential for autonomous driving.

This thesis is conducted in collaboration with the Ubiquitous and Intelligent Systems (UBIX)[1] research group at the University of Luxembourg's Interdisciplinary Centre for Security, Reliability, and Trust (SnT). The UBIX group specializes in distributed artificial intelligence across various fields, including smart mobility and digital twins. Further, its focus on real-world applications of AI in distributed systems aligns with the objectives of this research, particularly in enhancing the capabilities of Autonomous Driving Systems.

The RoboCar initiative [12] developed at the University of Luxembourg is one example of ADS research. RoboCar is an open-source research platform[2] designed to facilitate the fast development of experimental ADS. The contribution of this thesis is the implementation of a 3D object detection model that fuses camera and LiDAR data to improve detection performance despite using limited available resources. The goal is to produce better 3D bounding boxes compared to models that rely solely on LiDAR data.

The structure of this thesis is as follows: Chapter 2 reviews related work in the field of 3D object detection, highlighting various methodologies and their key concepts. Chapter 3 introduces general terminology, providing a foundation for understanding the technical aspects of the research. Chapter 4 details the methodology used in developing the proposed model, including data pre-processing, calibration, fusion, and tracking. Chapter 5 presents the evaluation of the model, discussing the datasets used, the runtime param-

[1] https://www.uni.lu/snt-en/research-groups/ubix/
[2] https://github.com/sntubix/robocar

eter configuration process, and performance comparisons with state-of-the-art models. Finally, Chapter 6 concludes the thesis, summarizing the findings and suggesting potential directions for future research.

2 Related Work

In recent research concerning 3D object detection in autonomous vehicles, the usage of data from multiple sensors has led to promising results. In particular, the fusion of camera and LiDAR data has emerged to be a wide research area with many proposed machine learning methods. The main reason to merge the data from both sensors is to combine the benefits of both while eliminating some of their specific limitations. Particularly, cameras capture their environment by providing rich visual information, e.g. textures and shapes of objects. However, their capability to capture accurate depth information is limited and thus it is non-trivial to get useful depth and distance information from its data. On the other hand, LiDAR sensors are very good at capturing precise depth information of their environment, however, they lack in providing detailed visual information. Thus, the combination of both sensors enables researchers to capture visual and depth information about the environment. This can lead to robust and accurate 3D object detection which is very important and useful in the context of autonomous driving.

This literature review will present the concepts and limitations of the current state-of-the-art methods concerning camera LiDAR sensor fusion with the goal of 3D object detection in autonomous driving. This section should provide a comprehensive overview of 7 different methodologies by highlighting their key concepts and differences compared to the other proposed methods.

2.1 DeepFusion

Yingwei Li et. al proposed two techniques, namely *InverseAug* and *LearnableAlign*, that are the building blocks for their 3D multi-modal detection models named **DeepFusion** [13]. This model aims to enhance the fusion of LiDAR and camera data for 3D object detection by fusing LiDAR features with camera features, rather than simply using a *point-level* fusion method, i.e. appending camera features onto LiDAR point clouds (decorating LiDAR points with image pixels). The **DeepFusion** pipeline receives LiDAR features by using existing LiDAR feature extractors. Similarly, the camera features are also obtained by using an existing camera feature extractor. Thus, the model builds on existing methods, however, Yingwei Li et. al underline the importance of alignment quality when data augmentation is applied during the training process to achieve the best model performance. To address this challenge, two innovative techniques, InverseAug and LearnableAlign, were introduced. InverseAug resolves feature alignment problems arising from geometric data augmentation by preserving the augmentations and reversing them prior to feature fusion, ensuring accurate coordinate alignment. LearnableAlign addresses the challenge of aligning LiDAR features with their corresponding camera features, as one LiDAR feature corresponds to a voxel (multiple points) of the LiDAR point cloud. By using a cross-attention mechanism, it dynamically captures

correlations between LiDAR voxels and camera pixels, ensuring that the most relevant camera information is effectively integrated into the fusion process, thus enhancing detection accuracy. The **DeepFusion** model achieved state-of-the-art performance on the *Waymo Open Dataset* in early 2022. This highlights the effectiveness of late-stage feature fusion assuming accurate alignment, motivating the authors to introduce two techniques to address this challenge.

2.2 TransFusion

DeepFusion is more robust against input corruptions and out-of-distribution data than single-modal 3D detection models due to its fusion of LiDAR and camera data. While DeepFusion was tested against laser and pixel noise in its input data, its performance with inferior image conditions, e.g. bad illumination or sensor misalignment, was not explored.
Xuyang Bai et. al proposed a robust solution to LiDAR-camera fusion with a mechanism to handle inferior image conditions, named **TransFusion** [14]. The authors proposed a model that uses *transformers* to fuse LiDAR and camera features. A transformer is a type of neural network architecture that consists of an encoder-decoder structure with an attention mechanism that enables the transformer to focus on the most important and relevant parts of the input. **TransFusion** uses a transformer to first predict bounding boxes from a LiDAR point cloud using object queries, which are sparse representations of potential object locations and characteristics. Then, a second transformer fuses relevant image features with the object queries to get an accurate prediction. Overall, the **TransFusion** model enables relevant image features to be used during the prediction even when there is only a limited number of LiDAR points of the specific object available. Additionally, the **TransFusion** model's robust fusion strategy offers significant advantages in real-world applications of autonomous driving. The model demonstrates resilience against challenges such as bad illumination, sensor misalignment, and degraded image quality shown in the results of the paper. By using transformers during the fusion stage, **TransFusion** represents a significant advancement in the field of 3D object detection by enabling both robust prediction and state-the-art (68.9 mAP on NuScenes) accuracy on large-scale datasets such as the *Waymo Open Datatset* as well as on the *NuScenes* prediction and tracking datasets.

2.3 BEVFusion

Unlike DeepFusion and TransFusion, which use the ground-level perspective of the LiDAR and camera sensors to fuse their features, Zhijian Liu et al. introduced a technique that transforms the input data into a bird's-eye-view (BEV) prior to fusion. The proposed method, **BEVFusion** [15], can preserve the geometric and semantic information of the environment in contrast to other methods which either lose semantic density when decorating LiDAR points with camera features or which are geometrically lossy when projecting the LiDAR point cloud to the camera plane. As **BEVFusion** transforms the camera and LiDAR data into BEV before fusing them, an efficient algorithm to transform camera features into BEV is crucial. The authors followed existing methods to predict the distinct depth distribution of each camera pixel. Each feature pixel is then divided into D distinct points along the camera ray. All these points are then combined

and flattened along the z-axis using the so-called *BEV pooling* operation to produce the BEV grid. This operation is very inefficient as the camera feature point cloud is a lot denser than the LiDAR point cloud and thus also very large. The authors solved this efficiency problem of the BEV pooling operation by using precomputation and interval reduction. First, the 3D coordinates of each point are precomputed and associated with the corresponding BEV grid. By calculating these values in advance, the framework establishes a mapping between each point and its corresponding BEV grid, simplifying the association process during inference. Second, the features within each BEV grid have to be aggregated by some symmetric function, producing many unused computed partial sums when using existing methods. The authors implement a specialized GPU kernel that removes the dependency between outputs and avoids writing partial sums to the DRAM. Overall, these optimizations make the BEV pooling operation 40 times faster, the latency is reduced from more than 500ms to 12ms [15].
After the shared bird's-eye-view conversion of the camera and LiDAR data, the features can be simply fused together by using concatenation for example. There can still be some spatial misalignment between the LiDAR BEV features and the camera BEV features. The authors use a convolution-based BEV encoder to compensate for this misalignment. The **BEVFusion** framework achieved state-of-the-art results on the *NuScenes* 3D detection benchmark (70.2 mAP) while being more efficient than existing solutions. Further, **BEVFusion** also achieves state-of-the-art results in BEV map segmentation while requiring less computation than existing solutions.

2.4 Robust BEVFusion

The majority of LiDAR camera fusion methods rely on the raw LiDAR point cloud data to generate image queries and thus heavily depend on the availability and quality of the LiDAR input. Thus, in a realistic scenario, where the reflection rate of LiDAR points could be low due to object texture or a software or hardware error could interrupt the availability of LiDAR data, these methods wouldn't yield good results.
Tingting Liang et al. proposed a novel method, also called **BEVFusion**, that detaches the LiDAR camera fusion dependency of the current methods [16]. The proposed framework uses two distinct streams for the LiDAR and camera sensors that first encode their respective raw inputs into features within the same BEV space before fusing the BEV-based features and passing them into an existing prediction head. The main idea of the **BEVFusion** framework proposed by Tingting Liang et al. is thus to be robust against the malfunction of one of the two modalities. The camera stream architecture of the framework is composed of three main modules. First, the *image-view encoder* encodes the input image into semantic-rich features. Next, the *view projector* module transforms the 2D image coordinates into 3D ego-car coordinates by predicting the image depth to construct the camera BEV features. Last, the *BEV encoder* module is used to further encode the voxel feature in the BEV space feature by reshaping instead of pooling (used by the previous BEVFusion method) to reduce cost while keeping semantic information.
The LiDAR stream architecture of the robust **BEVFusion** framework can incorporate any network that transforms LiDAR points into BEV features [16]. A popular approach involves using a parameterized voxelization to reduce the Z-dimension of raw points and then using networks with sparse 3D convolution to efficiently generate features in BEV space. The fusion between the camera and LiDAR BEV features is realised by concatenating them with learnable weights. The final component of the framework is the *detection*

head for which the authors decided to use a popular detection head module from earlier works as their **BEVFusion** method is generalizable. During the experiments, the authors compare the results of 3 different detection heady categories, anchor-based, anchor-free-based, and transform-based (TransFusion-L). Overall, the framework achieves state-of-the-art performance on the *NuScenes* dataset (69.2 mAP) when using *TransFusion-L* as their LiDAR stream. Using a dataset that simulates various LiDAR malfunctions, the proposed **BEVFusion** framework significantly outperforms other state-of-the-art methods (15.7% to 28.9% mAP).

2.5 MSMDFusion

Previous methods such as both *BEVFusion* frameworks as well as *TransFusion* do not benefit from the virtual points that are generated when lifting 2D pixels into the 3D BEV space. The quality of virtual points depends on accurate *depth* estimation as it directly determines their position in the 3D BEV space through perspective projection. Further, *depth* also improves the semantics of virtual points when lifted to the 3D BEV space. Despite the *depth* factor significantly affecting the accuracy of 3D detection, it is still under-investigated concerning virtual point generation.
The **MSMDFusion** framework introduced by Yang Jiao et al. [17] seeks to exploit the potential of virtual points and improve the limitations of other methods. The first step of the proposed framework is to extract features from the camera and LiDAR inputs. For the LiDAR point cloud, the points are voxelized and extracted similarly to both *BEVFusion* frameworks. Concerning the camera input, semantically rich image features are extracted before being lifted into the 3D voxel space. As this transfer from the 2D to 3D space is crucial, but also challenging, the authors propose a strategy called *Multi-Depth Unprojection* (MDU) to improve the quality of virtual points and mitigate shortcomings of other methods. The first step of the MDU operation is to estimate the depth of the image pixels, then improve the estimation using a K-nearest neighbours strategy and finally decorate the virtual points with depth-aware semantic features to get the most out of the semantic information before voxelizing them to match the LiDAR output. During the fusion process, the **MSMDFusion** framework uses a *Gated Modality-Aware Convolution* (GMA-Conv) block proposed by the authors. The GMA-Conv operation takes LiDAR points and virtual points (generated from camera images) as inputs and performs a controllable and fine-grained interaction between both modalities. First, LiDAR data is used to select useful information from the camera voxel features as LiDAR voxels are more precise, then the camera and LiDAR voxels are combined using grouped sparse convolutions to facilitate multi-modal interaction. The last step, after the fusion of the voxel features, is to compress their height to convert them into the BEV space and use a conventional BEV 3D detection head to realise the prediction. **MSMDFusion** achieves state-of-the-art performance on the *NuScenes* dataset with 71.5 mAP. The results also show that the **MSMDFusion** framework achieves better (state-of-the-art) performances than both *BEVFusion* frameworks even when using 100 times less generated virtual points showing the effectiveness of the 2 novel techniques MDU and GMA-Conv in increasing the quality and benefiting from the potential of the virtual points.

2.6 Cross Model Transformer

Similar to the *MSMDFusion* framework, the **Cross Modal Transformer** (CMT) framework is also more robust than other existing methods, e.g. it achieves similar results to state-of-the-art vision-based 3D object detectors when LiDAR data is missing. The **CMT** framework was proposed by Junjie Yan et al. [18] with the aim to provide an end-to-end 3D detection pipeline that can be easily extended and only involves simple operations (e.g. no 2D-to-3D transformation of image data for BEV representation).
The **CMT** framework first takes as input multi-view images from the camera and a LiDAR point cloud that are both fed into two individual backbones in order to extract image and point cloud feature tokens respectively. The authors proposed two modules that are used inside the pipeline. First, the *Coordinates Encoding Module* (CEM) is used to encode 3D coordinates into multi-modal tokens. As an image is captured from a perspective by the camera, each pixel can be represented as a line in a 3D space and the *CEM* can encode 3D points from the camera frustum space into position encodings. LiDAR point cloud tokens are sampled along the z-axis in BEV space and encoded by using existing methods (e.g. VoxelNet or PointPillar) to produce the position encodings. Both position encodings from the camera and LiDAR inputs are then used as input in the second module *Position-guided Query Generator*. In this module, the position encodings are used to align the multi-modal tokens in 3D space and generate position-guided queries (following the existing method PETR) as 3D reference points which are then projected into the image and LiDAR spaces respectively to obtain the relative coordinates. The position-guided queries and multi-modal tokens then interact further in each layer of the *transformer decoder* to update the queries until two feed-forward networks predict the 3D bounding boxes and classes of the objects. Thus, the **CMT** framework directly outputs accurate 3D bounding boxes without requiring post-processing. The authors extended the training of the **CMT** framework by introducing a *Masked-Modal Training* training that improves the robustness of the modal by randomly using only one modality during training. This training procedure does not affect the performance of the fusion model and hence is a good technique to improve robustness. Without the robustness training, the model fails to predict anything when only the camera is available. **CMT** achieves state-of-the-art performance on the *NuScenes* dataset (72.0 mAP) while providing better robustness than other methods such as *BEVFusion* or *TransFusion*.

2.7 LoGoNet

Most of the existing camera LiDAR fusion frameworks, including the one presented previously in this section, perform global fusion across the whole scene. Despite the good performance of these methods, they lack precise region-level information that can help to optimise the fusion process. Xin Li et al. propose a novel **Local-to-Global fusion network** (LoGoNet) [19] that fuses camera and LiDAR data at both global and local levels. The *Global Fusion* module is based on existing literature (e.g. *BEVFusion*) while improving the cross-modal alignment through the usage of voxel point centroids instead of voxel centers. For this purpose, the authors developed a *Centroid Dynamic Fusion* (CDF) module that uses a cross-attention module to achieve the fusion of image and their corresponding point cloud features in the voxel feature space. The *Local Fusion* module of **LoGoNet** should provide more local and precise geometric information during the fusion process and thus complement the global fusion. It first uses a

3D backbone to produce initial proposals out of the voxelized point cloud and divides these proposals into uniformly divided grids. Then, the grid centers are projected onto the corresponding images and the image features that are covered by the projection are fused with position-decorated point cloud features using a cross-attention module. This process returns locally fused features while maintaining the rich semantic information of the image features. The *Feature Dynamic Aggregation* (FDA) module of **LoGoNet** has the role to connect global and local features by using a self-attention module in order to produce more informative multi-modal features. It is a key element in the fusion of the multi-modal features that are generated by the *Global Fusion* and *Local Fusion* modules and creates relationships between different grid points. The **LoGoNet** framework achieves state-of-the-art performance on the *Waymo Open Dataset* (81.02 mAPH) and is the first 3D detection framework to achieve performance over 80 APH on all categories. It also surpasses *BEVFusion* by 1.05 mAPH on the *Waymo Open Dataset*.

3 Terminology

This chapter gives a broad overview of the main concepts and terms that are important to understand the proposed method for the fusion of LiDAR and camera data to detect objects. Hence, this chapter aims to provide a foundation for the remaining part of this thesis by presenting all the relevant aspects of the fusion process.

3.1 Calibration

The purpose of the calibration between LiDAR and camera sensors is to obtain the transformation from one coordinate system to the other enabling every point to be represented in either coordinate system. Thus, one can map a 3D LiDAR point into the 2D image coordinate system and conversely get the 3D coordinates, in the LiDAR coordinate system, of an image pixel. This process called **extrinsic calibration** of the sensors, is crucial to achieve good results when fusing LiDAR and camera data for 3D object detection or tracking. This section focuses on the spatial calibration between the two sensors, i.e. the transformation between the two coordinate systems. There exists also the calibration at the temporal level, i.e. time synchronization of the LiDAR and camera sensors, which is not presented in this section.

Besides extrinsic calibration, the **intrinsic calibration** of the camera sensor is also important. It refers to the internal properties of the camera such as focal length, optical center and distortion coefficients. Thus, the intrinsic camera calibration does not require the LiDAR sensor and needs to be performed only once. The intrinsic parameters specify the projection of a 3D point in the camera's field of view onto the 2D image plane.

3.1.1 LiDAR-Camera Transformation

Figure 3.1 illustrates a setup that includes LiDAR and camera sensors which capture a scene. A point P in this observed 3D scene is denoted as P_C in the camera coordinate system and as P_L in the LiDAR coordinate system. The transformation between the coordinate systems is performed using the extrinsic parameters $\mathbf{R}$ and $\mathbf{t}$. The origins of both coordinate systems are denoted by O_C and O_L corresponding to the camera (C) and LiDAR (L) respectively. A point is represented using its 3 coordinates X, Y and Z. The position of a point is defined as $P_C = \begin{bmatrix} X_C & Y_C & Z_C \end{bmatrix}^T$ in the camera coordinate system and as $P_L = \begin{bmatrix} X_L & Y_L & Z_L \end{bmatrix}^T$ in the LiDAR coordinate system. Additionally, the point P is also projected onto the image plane as $p_C = \begin{bmatrix} u & v \end{bmatrix}^T$. Equation 3.1 defines the projection between the 3D points P_C in camera coordinates and the 2D point p_C on the image plane, i.e. the **intrinsic** parameters [1]:

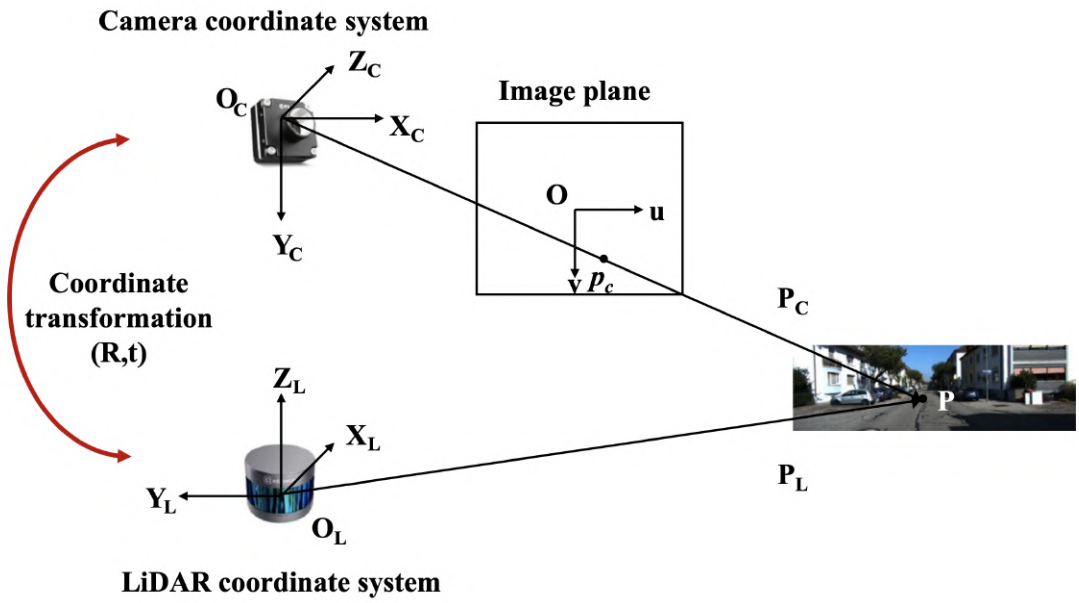


Figure 3.1: Transformation between LiDAR and camera sensors [1]

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \mathbf{K_C} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \tag{3.1}$$

In the above Equation 3.1, Z_C denotes the depth scale factor, f_x and f_y represent the focal length in pixels on the x-axis and y-axis respectively, (u_0, v_0) represent the optical center of the camera and s represents the skew coefficient, which is non-zero if the image axes are not perpendicular. The matrix $\mathbf{K_C}$ is thus the projection matrix from 3D camera coordinates to 2D image coordinates on the image plane.

Equation 3.2 defines the transformation between the point P_L in LiDAR coordinates and the point P_C in camera coordinates, i.e. the **extrinsic** parameters [1]:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \mathbf{R} \begin{bmatrix} X_L \\ Y_L \\ Z_L \end{bmatrix} + \mathbf{t} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \tag{3.2}$$

In the above Equation 3.2, $\mathbf{R}$ represents the rotation matrix and $\mathbf{t}$ represents the translation vector between the LiDAR and camera coordinate systems. Let $\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}$ denote the combination of these values, i.e. the total extrinsic parameters.

In order to obtain the total transformation between a 3D LiDAR point P_L and the corresponding 2D image plane point p_C, one can combine Equations 3.1 and 3.2 [1]:

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K_C} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \tag{3.3}$$

Equation 3.3 defines the mapping of 3D points from the LiDAR coordinate system to 2D points on the camera's image plane (pixel coordinates) and vice versa by using the intrinsic and extrinsic parameters.

3.1.2 Extrinsic Calibration Methods

Several methods exist for performing extrinsic calibration between LiDAR and camera sensors. These methods can be categorized based on whether they require a calibration target and whether they can be performed automatically or need human intervention. The 4 main categories are presented in the subsection below [1].

Manual Target-based Calibration

This extrinsic calibration method requires human intervention to manually specify the relationship between the LiDAR point clouds and camera images using at least one calibration target, such as a checkerboard-like pattern. The extrinsic parameters, i.e., rotation and translation, are obtained as the calibration target establishes geometric constraints between the pixels in an image and the points in the corresponding point cloud. The checkerboard is captured by both sensors from multiple views to compute the rotation matrix and translation vector.

Automatic Target-based Calibration

As the name suggests, this calibration method only requires minimal human intervention to perform the LiDAR-camera calibration. Multiple targets are used simultaneously to extract various features, which are then used to establish the correspondence between image pixels and LiDAR points. For instance, instead of capturing multiple shots of a single checkerboard at different angles and locations, this method can use a single shot containing several checkerboards. Another possible calibration target is a sphere, which is captured by both sensors. The center of the sphere is matched after being computed from the captured 3D points and pixel contours.

Manual Targetless Calibration

A targetless calibration method is needed for applications that require online adjustments, e.g. autonomous vehicles, to obtain the extrinsic parameters in the real world without using specific calibration targets. The manual targetless method still requires human intervention by using rules and patterns to manually specify the relation between point clouds and images. Usually, these methods involve multiple frames of related points and pixels to compute the extrinsic parameters using an algorithm that estimates the transformation by iteratively refining the alignment between the LiDAR points and image pixels.

Automatic Targetless Calibration

This calibration method neither requires considerable manual intervention nor specific calibration targets to compute the extrinsic parameters. Automatic targetless calibration methods exploit the surrounding environment to extract features, which can then be fed into a neural network that estimates the transformation between LiDAR and camera sensors. Another approach involves using ego-motion analysis when both LiDAR and camera sensors are moving during calibration. The most suitable approach depends

on the surroundings and scenarios, such as the number of features, motion, computing power, and available training data.

3.2 Object Detection

Object detection refers to the task of detecting and classifying multiple objects for some input data, such as an image or a point cloud. The general goal of object detection is to locate the objects within the given input data and create bounding boxes around them that are characterised by a label. These bounding boxes give information about the location of the object inside the coordinate system of the respective input data. The labels specify the corresponding category of each detected object, such as car, pedestrian or bicycle. In autonomous vehicles, object detection is a crucial component that allows the autonomous car to detect other vehicles, persons and traffic signs, among other things in order to determine an optimal and collision-free trajectory.

3.2.1 Traditional Methods

Before 2014, deep learning and the available hardware components were not yet at the stage to perform fast and accurate object detection. Conventional object detection methods included three main steps [20]:

1. **Selection of region** refers to the process of identifying the regions of objects of varying sizes by inspecting the image using a multi-scale sliding window technique. This method is however computationally intensive and often leads to redundant detections.

2. **Feature extraction** is performed after the localisation of objects in order to provide representations of object recognition. However, it is challenging to create an accurate feature descriptor that is able to recognize all types of objects within an image with contrasting backgrounds and variations in perspective and lighting.

3. **Classification** uses the extracted features to give the recognized object a label that specifies its category.

3.2.2 Convolutional Neural Networks (CNN)

As one can see from the above in Section 3.2.1, the traditional object detection methods are computationally costly and involve manually created feature extractors that lack in accuracy. To overcome these drawbacks of traditional methods, deep learning is nowadays used for object recognition. There are two types of object detectors that use deep learning, namely two-stage and one-stage detectors [20].

Region-Based Detectors

Two-stage object detectors divide the task into object localisation and object classification. This means that the detector first creates suggestions for potential regions for objects. Then the detector classifies the object within the generated regions based on the extracted features and according to the corresponding category. Such detectors are also called region-based frameworks which have the advantage of high detection accuracy, however, with the downside of a higher inference speed.

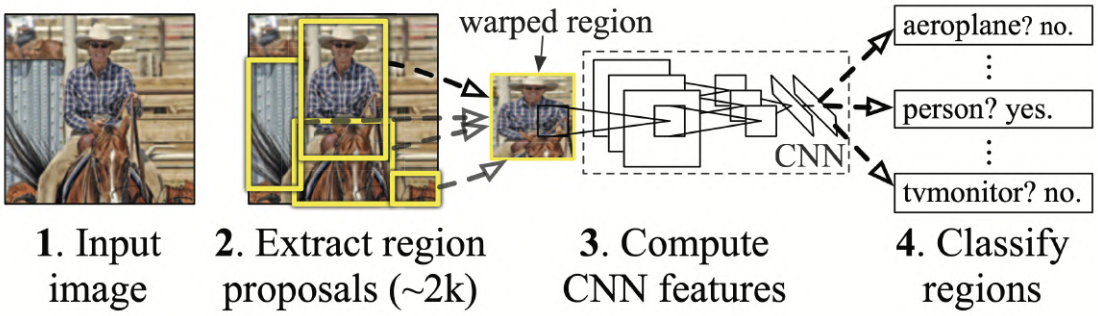


Figure 3.2: RCNN Overview [2]

The *Region-based convolutional neural network* (**RCNN**) proposed by Ross Girshick et al. [2] is the first object detection model to effectively apply deep learning. The general model overview of the model is shown in Figure 3.2. The RCNN model consists of 3 main modules [2], [20]:

1. Selective search is used to extract category-independent **region proposals** that serve as candidate detections that are used by the next module. RCNN identifies roughly 2000 regions in this first module based on differing textures, scales and colours.

2. A large **CNN** processes the region proposals that are firstly rescaled to fixed length vectors to fit the CNN input. The CNN then extracts the features from each of the regions.

3. The third module uses a set of class-specific **Support Vector Machines** (SVM) to detect if an object is present in a region or not. A linear regression model is then used to refine the bounding boxes created by the SVM.

Despite showing great improvements in terms of accuracy and speed, the RCNN model still has the limitations of being inefficient when processing each region separately by the CNN which leads to redundant computations and requires a lot of storage. Ross Girshick proposed the **Fast RCNN** model [21] to improve its previous RCNN object detector. It processes the entire image through the CNN to generate a convolutional feature map, from which region proposals are extracted. Then, a Region of Interest (RoI) pooling layer extracts a fixed-length feature vector from each region proposal. These vectors are then processed by some fully connected layers, one of which produces softmax probability estimates over the available object classes and the other one produces 4 values that define the bounding box corners.

The Fast RCNN model significantly improves the RCNN model in terms of inference speed, efficiency and training performance. However, Fast RCNN still uses selective search to generate the region proposals which is computationally expensive and slow. Shaoqing Ren et al. improved this downside by introducing **Faster RCNN** [22] by introducing a novel *Region Proposal Network* (RPN) that is a fully connected CNN. The RPN generates region proposals directly from the shared feature map that is generated by a CNN, similar to the first step of the Fast RCNN model. The output of the RPN is then fed into the Fast RCNN detector, thus in simple terms, the RPN tells the Fast RCNN module where to look. The Faster RCNN model is able to perform real-time object detection as the RPN module generates region proposals much faster than selective search. Further, Faster RCNN also improves accuracy and end-to-end training.

Regression-Based Detectors

One-stage object detectors combine the tasks of object localization and object classification, therefore they are also called region proposal-free frameworks. Hence, there is no need anymore for region proposals before the detection process and only one pass through a neural network is needed to directly predict bounding box coordinates and class scores from the input image. These one-stage detectors use regression-based CNNs. Regression refers to the process of predicting a continuous value based on one or more input variables, also known as features. In object detection, these continuous values are the bounding box coordinates and size.

The first relevant regression-based object detector was **You Only Look Once** (YOLO) presented by Joseph Redmon et al. [23]. It first predicts the bounding box coordinates for the objects and then determines the probability of their category. In contrast to region-based detectors, YOLO uses features from the whole image to perform its detections. The model achieves this by dividing the image into a grid where each cell predicts bounding box coordinates as well as confidence scores for every available category. In each cell, the category with the highest confidence score is selected and the bounding box with the highest Intersection over Union (IoU) with the other boxes in the cell is selected.

Another relevant regression-based object detector is the **Single Shot MultiBox Detector** (SSD) introduced by Wei Liu et al. [24]. Similar to the YOLO model, the SSD model uses a feed-forward CNN that creates fixes-size bounding boxes and confidence scores for object classes inside these boxes. It then applies a non-maximum suppression step that removes redundant boxes based on IoU to produce the final detections. Unlike YOLO, the SSD model uses anchor boxes, which are predefined boxes of various sizes and aspect ratios placed at multiple locations on the feature maps. SSD is therefore able to extract features of different aspect ratios and sizes, which ensures good recognition accuracy. The SSD model outperforms the basic YOLO model in terms of accuracy and speed.

The YOLO model was further improved over time to fix limitations and improve the model's overall performance. The most recent version as of the writing of this thesis, namely YOLOv8, is further explained in Section 3.3.

3.2.3 Performance Metrics

The effectiveness of object detection frameworks can be measured using several metrics. Such metrics help scientists to compare different models on various datasets by utilizing the 3 main outputs of object detectors, i.e. object label (class), bounding box and confidence score. The most common evaluation metrics are presented in the subsections below. First, the basic concepts that are used in these metrics should be clarified [25]:

- **Ground-Truth (GT)**: The ground-truth data consists of manually annotated bounding boxes and object classes in the images of a dataset. This data is used to train the model and later to evaluate its performance by comparing the model's predictions to the ground-truth annotations.

- **True Positive (TP)**: Instances where the model correctly predicts a bounding box that matches a GT bounding box and class.

- **False Positive (FP)**: Instances where the model predicts a bounding box that either does not exist in the GT data or a misplaced detection of an existent GT bounding box.
- **False Negative (TN)**: Instances where objects present in the GT data are missed by the model, meaning the model fails to detect them.
- **True Negative**: Instances that are not detected by the model and not present in the GT data, i.e. correct prediction of the absence of an object. This concept is not used in object detection as there are infinitely many bounding boxes that should not be detected inside an image.

Intersection over Union

In order to use the above concepts, one has to first define what a *correct* and *incorrect* detection is. When working with bounding boxes, a common way to accomplish this task is to use the **Intersection over Union (IoU)** between the GT and predicted bounding boxes. It measures the similarity of two sets of data as it is based on the Jaccard Index. In the context of object detection, the IoU metric measures the intersecting area of two bounding boxes divided by the area of the union between them as illustrated in Figure 3.3. The IoU of a ground-truth bounding box BB_{GT} and a predicted bounding box BB_P can thus be computed as follows:

$$IoU = \frac{area(BB_P \cap BB_{GT})}{area(BB_P \cup BB_{GT})} \tag{3.4}$$

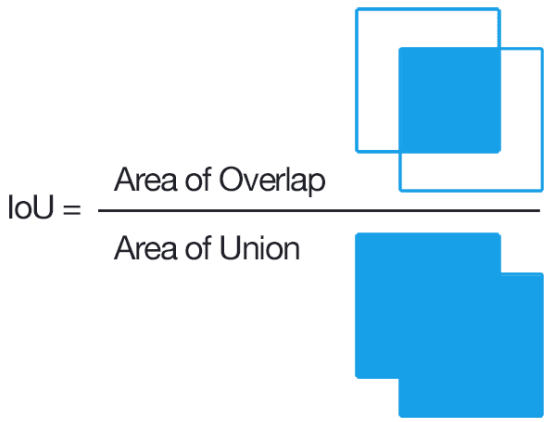


Figure 3.3: Intersection over Union equation [3]

One can now use the IoU value and compare it to a given threshold t to identify when a given detection is classified as being correct or incorrect. Thus, if $IoU \geq t$ then the detection is considered correct, i.e. a *true positive*. If $IoU \leq t$ then the detection is considered incorrect, i.e. a *false positive*.

Precision & Recall

As the concept of *true negatives* are not used in the evaluation of object detection models, most performance metrics are based on the **Precision** and **Recall** concepts.

Precision, defined in Equation 3.5, indicates how many of the total predictions made by the model are actually correct. Thus, precision shows the model's ability to make correct positive predictions. A high precision indicates that when a model makes a prediction, it is usually correct.

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{all\ detections} \tag{3.5}$$

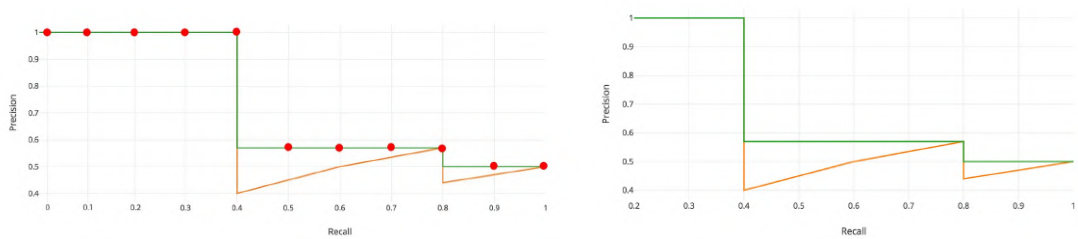
Recall, defined in Equation 3.6, describes the model's ability to predict all relevant objects, i.e. all GT bounding boxes. Hence, recall shows how many relevant objects a model correctly predicts and how many it misses in a given image. A high recall indicates that a model captures most of the relevant objects that are present.

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{all\ ground\ truths} \tag{3.6}$$

Both precision and recall are important metrics for object detection. A good model should capture all ground truths, i.e. high recall ($FN = 0$), while still correctly predicting all these relevant objects, i.e. high precision ($FP = 0$). Thus, for a good model, the precision remains high while the recall increases. One can compute the **precision-recall curve** that represents the relationship between the precision and recall for various confidence values of the bounding boxes generated by the model. The area under the curve (AUC) indicates high precision and recall for a specific confidence threshold. Hence, for various confidence thresholds, a high AUC indicates that the model performs well.

Average Precision

The most common metric to measure the accuracy of object detection models is the **Average Precision (AP)**. This metric is used to assess the detection of different models and thus typically used to compare them. The AUC of the precision-recall curve represents the AP of the model and thus summarizes the model's performance into a single scalar value. There are two common ways to compute the AP: 11-point interpolation and all-point interpolation. These techniques are used to smoothen the precision-recall curve as it often has a zigzag-like shape. Figure 3.4 shows the smoothening of the precision-recall curves using the 11-point interpolation (a) and the all-point interpolation (b).



(a) 11-point interpolation (b) all-point interpolation

Figure 3.4: Precision-Recall Curves [4]

The 11-point interpolation summarizes the shape of the precision-recall curve by computing the maximum precision values at 11 equally spaced recall values $(0, 0.1, ..., 0.9, 1)$ and then averaging them. Equation 3.7 shows how the AP is computed using this technique [25]:

$$AP_{11} = \frac{1}{11} \sum_{R \in \{0, 0.1, ..., 0.9, 1\}} P_{interp}(R) \tag{3.7}$$

where

$$P_{interp}(R) = max_{\tilde{R} \geq R} \; P(\tilde{R})$$

In the above definition, the AP is obtained by using the maximum precision $P_{interp}(R)$, whose recall value is greater than R, instead of using the observed recall level R at each step in order to smoothen the curve.

The all-point interpolation smoothens the precision-recall curve similar to the 11-point interpolation, but by using all recall points instead of just 11 fixed points. Equation 3.8 shows how the AP is computed using the all-point interpolation method [25]:

$$AP_{all} = \sum_{n} (R_{n+1} - R_n) \; P_{interp}(R_{n+1}) \tag{3.8}$$

where

$$P_{interp}(R_{n+1}) = max_{\tilde{R} \geq R_{n+1}} \; P(\tilde{R})$$

In the above definition, the AP is computed by interpolating the maximum precision $P_{interp}(R_{n+1})$, whose recall value is greater than R_{n+1}, observed at each level instead of only a few points.

Mean Average Precision

As the AP is computed for each class separately, there is a need for a metric to represent the model's performance overall available object classes. The **mean Average Precision (mAP)** is the average of all AP values for every class in a database. Equation 3.9 shows how the mAP is computed [25]:

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{3.9}$$

In the above equation, AP_i represents the AP in the ith class and N is the total number of object classes in a specific database for which the model is evaluated.

3.3 YOLO

The object detection model YOLO (You Only Look Once) was originally presented by Joseph Redmon et al. [23] in 2015 with the goal of providing a fast object detection

model that only uses a single neural network. Over time, YOLO became a state-of-the-art model, especially the most recent YOLOv8 model [7] developed by Ultralytics with several features such as detection, segmentation, tracking and classification. These characteristics of YOLOv8 make the model well-suited as the detection model for this Master Thesis.

3.3.1 Original YOLO model

The YOLO models are known for their speed, accuracy and end-to-end object detection approach. The original YOLOv1 model marked a significant change in object detection frameworks by treating object detection as a single regression problem. The model was designed to be both fast and accurate by processing the entire image at once with a single neural network instead of using region proposal methods. The architecture of YOLOv1 included 24 convolutional layers followed by two fully connected layers, with the input image divided into a 7×7 grid. Each grid cell predicted two bounding boxes and class probabilities, which simplified the object detection pipeline and enhanced speed.

A simplified output prediction of the YOLOv1 model, which only uses a 3×3 grid, three classes and one class per grid cell, is shown in Figure 3.5. Each grid cell will output a vector containing 8 values $(P_c, b_x, b_y, b_h, b_w, c1, c2, c3)$ that contain information about the confidence score, bounding box dimensions and class probabilities. This makes a total of $3 \times 3 \times 8$ output predictions for one image using this simplified approach.

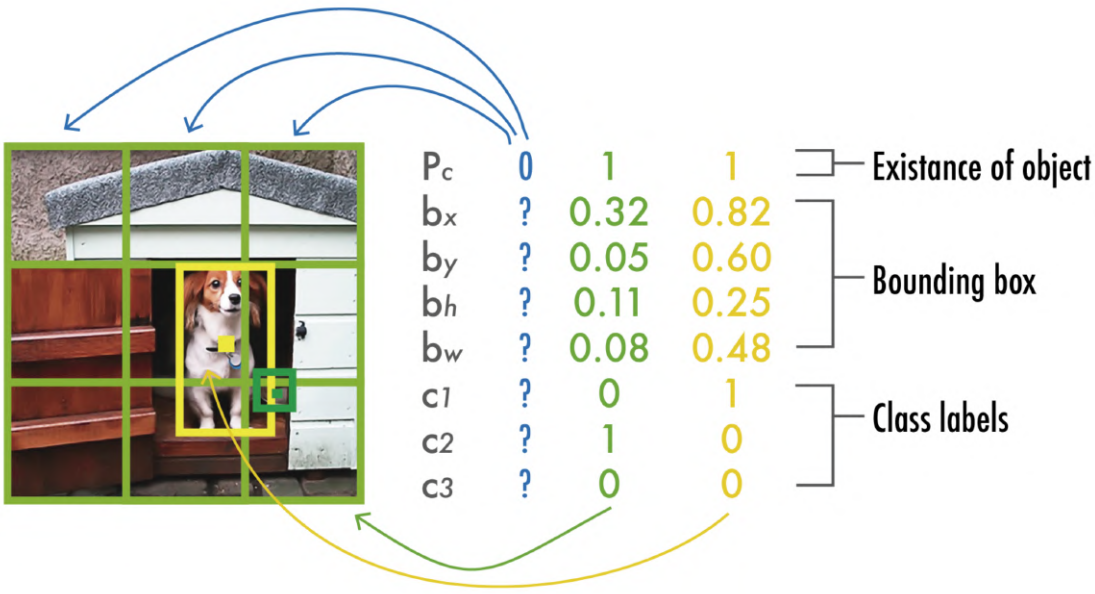


Figure 3.5: Illustration of YOLOv1 output prediction [5]

3.3.2 Evolution from YOLOv2 to YOLOv7

The original YOLOv1 model was improved over time to address the limitations of earlier models and enhance performance. Generally, the goal of newer YOLO models was to be more accurate while still being faster. The itemization below briefly introduces the model version YOLOv2 to YOLOv7 [5]:

- **YOLOv2**: Released in 2016, YOLOv2 aimed to improve accuracy while maintaining speed. It introduced batch normalization to reduce model overfitting, high-resolution classifiers, and anchor boxes to better predict bounding boxes. YOLOv2 used Darknet-19 as its backbone, which consisted of 19 convolutional layers followed by 5 max-pooling layers. Along with YOLOv2, another novel model called *YOLO9000* was introduced. It allowed the model to be trained using a combination of detection and classification data. Further, the YOLO9000 model was capable of detecting more than 9000 categories.

- **YOLOv3**: Launched in 2018, YOLOv3 built upon the success of its predecessors by introducing Darknet-53, a deeper backbone with 53 convolutional layers featuring residual connections. YOLOv3 also improved multi-scale predictions, allowing the model to detect objects at different scales within the same image by detecting 3 boxes at 3 different scales. This improvement helped to get more detailed boxes and thus significantly improved the accuracy of small object detection. YOLOv3 maintained the speed advantage of previous versions while significantly boosting accuracy.

- **YOLOv4**: Released in 2020, YOLOv4 brought substantial advancements in feature extraction capabilities and detection accuracy. At that time, architectures of object detection models were described in three parts, i.e. the backbone, neck and head. The YOLOv4 model used a modified Darknet-53 with Cross Stage Partial (CSP) connections and a Mish activation function as its backbone. YOLOv4 also incorporated techniques like Spatial Pyramid Pooling (SPP), Mosaic data augmentation, DropBlock regularization, and CIoU (Complete Intersection over Union) loss for better performance in complex environments.

- **YOLOv5**: Although not officially part of the original YOLO series by Joseph Redmon, YOLOv5, developed by Ultralytics in 2020, became widely popular due to its user-friendly implementation and practical improvements. It incorporates many improvements from YOLOv4 but was developed in PyTorch instead of Darknet. YOLOv5 focused on simplifying the deployment process and offered different model sizes (nano, small, medium, large, and extra large) to balance between speed and accuracy. YOLOv5 is capable of classification, and instance segmentation and provides an overall easier way to use, train and deploy.

- **YOLOv6 & YOLOv7**: Introduced in 2022, these versions included further architectural enhancements, but they were not developed by Ultralytics. YOLOv6 introduced a new backbone based on RepVGG called EfficientRep that uses higher parallelism which improves processing speed during inference. YOLOv7 focused on extended efficient layer aggregation (E-ELAN), which allows for more efficient training, resulting in overall the fastest and most accurate object detection model at that time.

3.3.3 YOLOv8

YOLOv8 is the latest iteration of the YOLO family introduced by Ultralytics [7] in 2023 and based on their YOLOv5 model with various modifications in terms of model scaling and architecture tweaks. YOLOv8 offers 5 different model sizes similar to its predecessor YOLOv5, but it also provides a semantic segmentation model.

Architecture

The architecture of YOLOv8 can be broken down into 3 main components [26] [5]:

- The **backbone** of YOLOv8 is a Convolutional Neural Network (CNN) that extracts image features using a mathematical operation called *convolution*. Through multiple layers of convolutional filters and pooling operations, the CNN learns to recognize different features in images, capturing patterns and reducing data size while keeping important details. YOLOv8's backbone is built upon YOLOv5's backbone and includes the C2f (Cross Stage Partial bottleneck with two convolutions) module, which improves detection accuracy by combining high-level features with contextual information.

- The **neck** of YOLOv8 merges the features extracted by the backbone from various stages to capture important information across different scales. YOLOv8 employs the Path Aggregation Network (PAN) to enhance feature fusion and improve the model's ability to detect objects at various scales. This neck design helps in better combining low-level spatial details with high-level semantic information, improving the detection performance, especially for small objects.

- The **head** of YOLOv8 uses the information from the neck to produce predictions. It features an anchor-free design that simplifies the model architecture and improves its ability to generalize to different object sizes and shapes, a decoupled head that separately processes objectness, classification, and regression tasks to improve overall prediction accuracy, an IoU (Intersection over Union) loss function that measures the overlap between predicted and ground truth bounding boxes to generate more accurate bounding boxes and better handle overlapping detections, and multi-scale detection to allow the model to detect small, medium, and large objects within the same image. The IoU loss function is crucial for training object detection models as it quantifies how well the predicted bounding boxes match the actual objects, guiding the model to refine its predictions for better precision and recall.

- The **activation function** in a neural network introduces non-linearity, thus allowing the network to learn and represent more complex patterns in the data. In its output layer, YOLOv8 uses the sigmoid activation function for the objectness score, i.e. the confidence score that a bounding box contains an object. It further uses the softmax activation function for class probabilities, i.e. if the object inside the bounding box belongs to a certain class. In its convolutional layers, which are specialized layers designed to automatically and adaptively learn visual features from input images, YOLOv8 uses the SiLU (Sigmoid Linear Unit) activation function along with batch normalization.

The complete architecture of the YOLOv8 model is shown in the Appendix in Figure A.1.

Training

YOLOv8 uses several techniques and methods during its training process [5]:

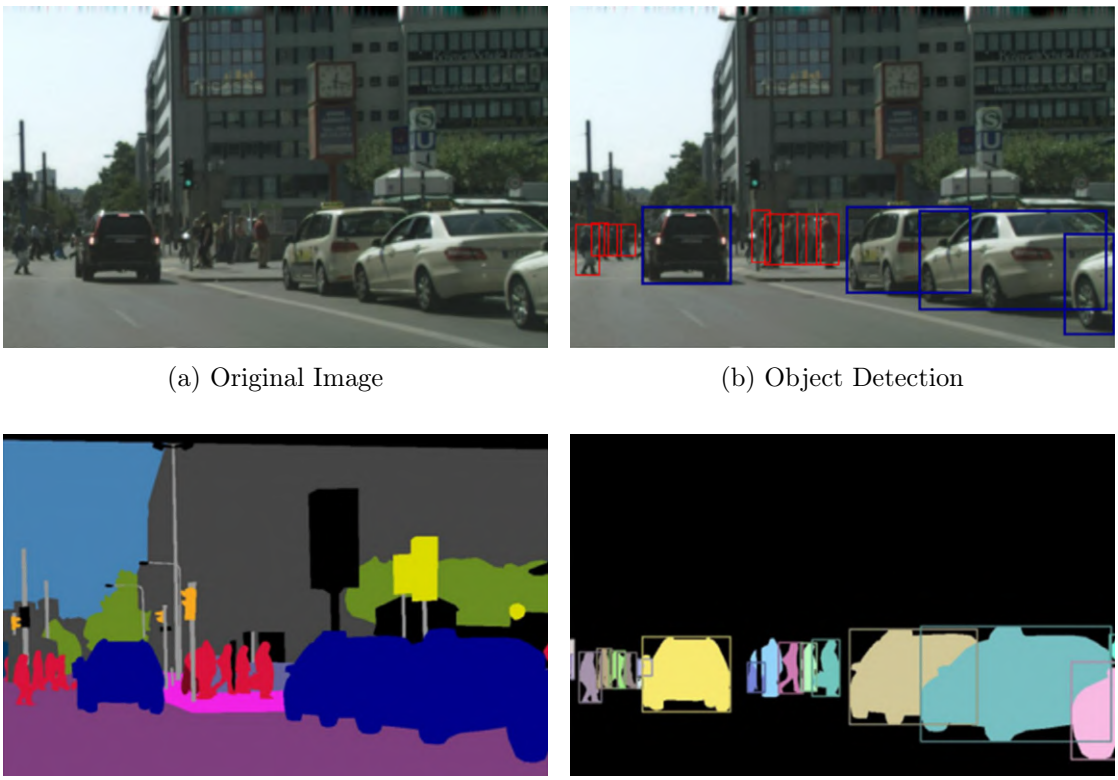
- **Training Strategies**: The training process of YOLOv8 aims to optimize performance by employing several key techniques. It utilizes varying image resolutions during training and includes a mosaic data augmentation that combines multiple

images to create a single image as training input. The training uses less computational resources as the data is processed in batches and it prevents overfitting by using early stopping techniques. These strategies enhance the model's ability to generalize and improve robustness when handling complex input scenarios.

- **Loss Functions**: YOLOv8 uses a combination of the CIoU (Complete Intersection over Union) and DFL (Distribution Focal Loss) loss functions for bounding box loss and binary cross-entropy for classification loss. These loss functions have overall improved the model's object detection performance, specifically when dealing with overlapping objects and small object detection.

Segmentation Model

Semantic Segmentation is a computer vision task that aims to assign a class label to each pixel in an input image. Thus, semantic segmentation partitions the image into meaningful parts, where the shapes of the objects are clearly visible. In contrast to object detection, shown in Figure 3.6(b), which outputs the location and class of a detected object, semantic segmentation, shown in Figure 3.6(c), also outputs the precise boundary lines around an object. One step further is the instance segmentation shown in Figure 3.6(d), which identifies each object individually in the image even if multiple objects are close and have the same label.



(a) Original Image (b) Object Detection

(c) Semantic Segmentation (d) Instance Segmentation

Figure 3.6: Comparison of different vision tasks [6]

YOLOv8 also provides an instance segmentation model, called YOLOv8-seg, which implements a similar structure as the standard object detection model. The backbones of both models are the same, thus the CSPDarknet53 feature extractor. YOLOv8-seg

however replaces the traditional YOLOv8 neck with a C2f module, which enhances the model's ability to capture small but crucial details for segmentation [5]. The segmentation model further contains two segmentation heads that learn to predict instance segmentation masks for various input images [5]. YOLOv8-seg maintains similar detection heads to the standard YOLOv8 model, consisting of five detection modules and a prediction layer. Similar to the standard model, the segmentation model of YOLOv8 is also available in 5 different sizes which provide a trade-off between inference speed and accuracy.

Variants & Performance

The YOLOv8 model offers various variants that differ in size and inference latency. This allows users to select a model based on the desired trade-off between accuracy and latency, tailored to their specific application requirements. Figure 3.7 illustrates the mean average precision (mAP) scores on the COCO dataset [27] of different YOLO models in relation to their size (number of parameters) and inference speed (latency).

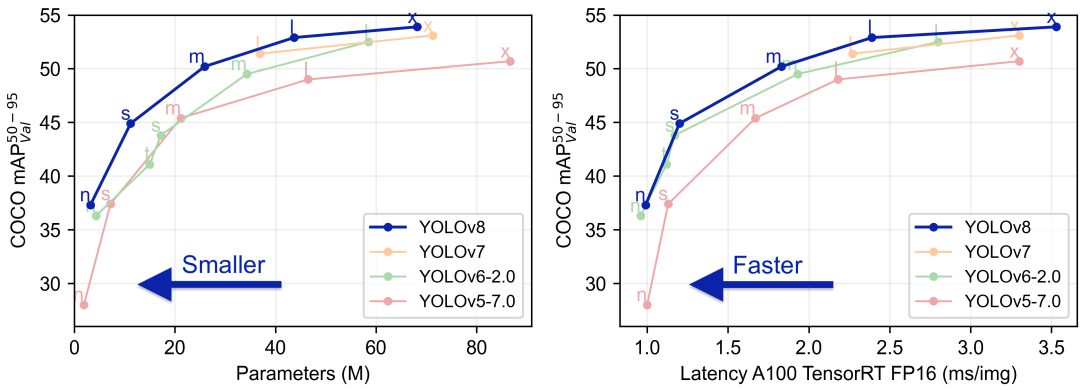


Figure 3.7: Comparison of YOLO model sizes [7]

This Master Thesis uses the YOLOv8 model as it provides fast and accurate detections and object segmentations that can be used as a building block for the LiDAR-camera fusion step.

3.4 Clustering

Clustering is a form of unsupervised learning that aims to group data points based on their similarity, uncovering natural patterns and structures within the data. Unsupervised learning, a type of machine learning, uses algorithms to analyze unlabeled data to gain insights without prior training.

3.4.1 Overview

A clustering algorithm should meet three key criteria: it should maximize similarity within clusters, maximize distinction between clusters, and use clear and meaningful criteria for measuring similarity and dissimilarity [28].

Generally, the clustering process follows the following steps [28]:

1. **Feature extraction and selection**: Identify and select the most representative features from the original dataset.

2. **Clustering algorithm design**: Develop the clustering algorithm based on the problem's characteristics.

3. **Result evaluation**: Assess the clustering results to determine the algorithm's effectiveness.

4. **Result explanation**: Provide a practical interpretation of the clustering results.

3.4.2 Distance & Similarity

Distance and similarity are fundamental concepts in clustering algorithms. For numerical data, distance metrics like the *Minkowski distance*, which includes both *Euclidean distance* (when $p = 2$) and *Manhattan distance* (when $p = 1$), measure the separation between data points to reveal relationships and patterns.

$$dist(x_1, x_2) = \left(\sum_{j=1}^{d} |x_{1j} - x_{2j}|^p \right)^{\frac{1}{p}} \tag{3.10}$$

where x_1 and x_2 are d-dimensional data points, x_{1j} and x_{2j} are the j-th component of x_1 and x_2 respectively and p determines the type of the distance metric (Euclidean or Manhattan).

For categorical data, similarity metrics like the *Jaccard Similarity* defined in Equation 3.11 are more suitable, as they measure how alike data points are based on qualitative features.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.11}$$

where $|X|$ represents the number of elements in the set X.

These metrics play a key role in determining how clusters are formed and interpreted across different types of data and thus have a big impact on the output of the algorithms.

3.4.3 Evaluation Indicator

Evaluation indicators assess the effectiveness and validity of clustering algorithms, which is crucial for understanding and comparing the performance of algorithms. There exist two main categories of evaluation indicators [28]:

- **Internal Evaluation Indicators** use data from the clustering process to evaluate aspects like cluster cohesion and separation. They may not always provide a definitive comparison between algorithms.

- **External Evaluation Indicators** are often referred to as a *gold standard* as they use external data to objectively measure clustering quality based on known ground truth, making them more suitable for comparing algorithms.

3.4.4 KMeans

KMeans is one of the most popular clustering algorithms and falls into the category of **Partition-Based** clustering, also called centroid-based clustering. The KMeans algorithm requires the number of clusters k to be specified at the beginning of the algorithm as input. The main steps of the KMeans algorithm are the following:

1. **Initialisation**: Randomly select k different centroids, also called *means*, in the set of data points.
2. **Assignment**: Assign each data point to its nearest centroid using a distance metric presented in Section 3.4.2, e.g. Euclidean distance.
3. **Update**: For all clusters, compute the mean of all points inside that cluster and assign it as its new centroid.
4. **Repeat**: Steps 2. and 3. are repeated until the centroids no longer change significantly or a maximum number of iterations is reached.

Figure 3.8 illustrates how the KMeans algorithm clusters a random set of data points. As one can see, the the input parameter k was initialised as $k = 4$ which results in 4 distinct clusters with their centroids marked as red triangles.

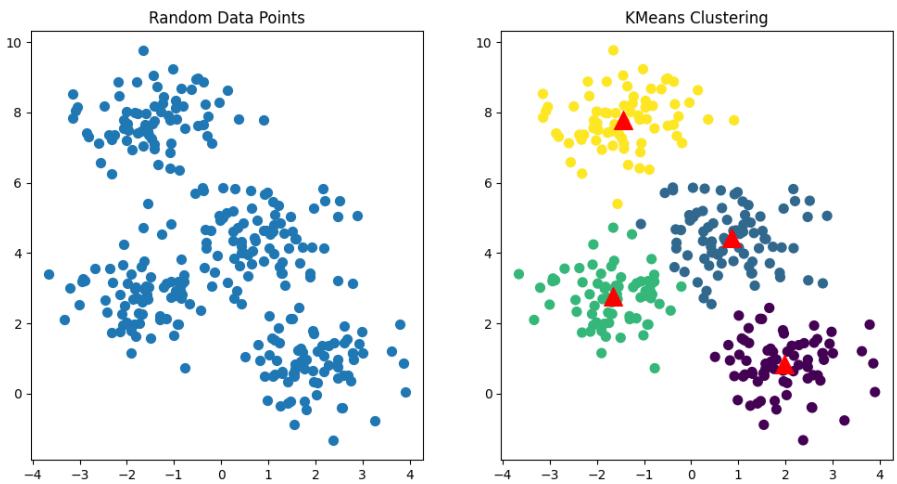


Figure 3.8: KMeans Clustering Illustration

KMeans, like most other partition-based clustering algorithms, has the following advantages [28]:

- **Complexity**: High computing efficiency while having a relatively low time complexity $O(k * n * t)$, where k represents the number of clusters, k the number of total data points and t the number of iterations.
- **Implementation**: Simple to implement and there exist a lot of libraries that have implemented KMeans already.

The algorithm has however also several drawbacks and limitations [28]:

- **Specification**: Requires the number of clusters K to be specified in advance.

- **Initialization**: Sensitive to the initial number and selection of centroids.
- **Convergence**: May converge to a local optimum, thus not necessarily the best result.
- **Suitability**: Not suitable for non-convex data and relatively sensitive to outliers as it assumes clusters are spherical and of similar size.

3.4.5 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is an algorithm that belongs to the category of **Density-Based** clustering. Unlike centroid-based methods such as KMeans, DBSCAN does not require the number of clusters to be specified beforehand and can discover clusters of arbitrary shapes. DBSCAN creates clusters based on two parameters:

- **eps** (epsilon) defines the radius of a neighbourhood around a point, i.e. the maximum distance between two points for them to be considered in the same neighbourhood. If the *eps* is too large, distinct clusters could be merged into one. If it is too small, more points can be considered as outliers which results in fewer clusters.
- **min_samples** defines the minimum number of points in a neighbourhood (within *eps* radius). A higher value for *min_samples* requires denser regions to form a cluster, which can result in overall fewer clusters. A smaller value assigns fewer points to be considered in the same neighbourhood, which can result in overall more clusters.

The formation of clusters in the DBSCAN algorithm includes three different types of data points:

- **Core Points**: points that have at least *min_samples* within its *eps* radius.
- **Border Points**: points that belong to the neighbourhood of a core point, but don't have enough points within their *eps* radius to become a core point themselves.
- **Noise Points**: points that don't meet the requirements to be a core or border point.

The main steps of the DBSCAN algorithm are the following:

1. **Core Point Identification**: Identify all core points in the dataset.
2. **Cluster Expansion**: Recursively add points that are within *eps* radius from a core point or its neighbourhood points to form or expand the cluster.
3. **Noise Handling**: Assign remaining points that are not reachable from any core point as noise.

Figure 3.9 illustrates how the DBSCAN algorithm clusters a random set of data points. The standard values of $eps = 0.5$ and $min_smaples = 5$ were used to form this graph. One can see that there are 4 clusters and some noise points (in violet).

DBSCAN offers similar advantages to other density-based clustering algorithms [28]:

- **Flexibility**: Capable of identifying clusters of arbitrary shapes and sizes.

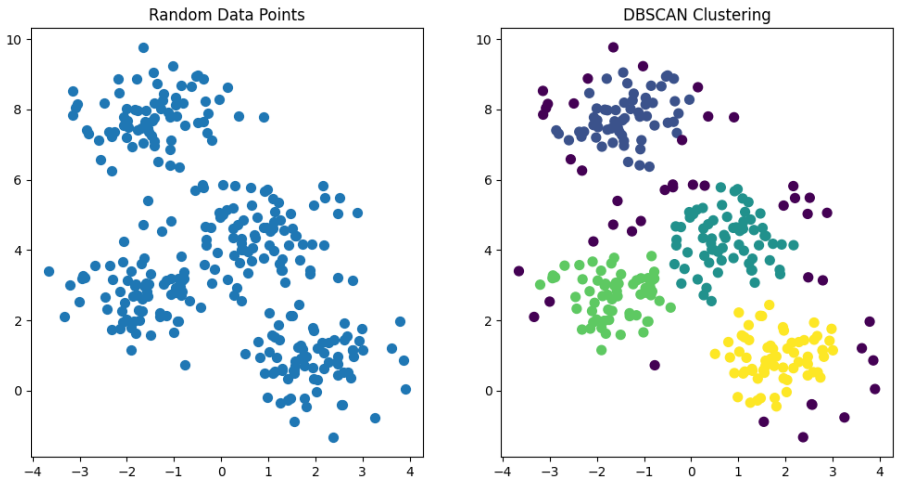


Figure 3.9: DBSCAN Clustering Illustration

- **Robustness to Noise**: Able to handle outliers and noise points effectively without forcing them into clusters.
- **Automatic Cluster Number Determination**: Not required to specify the number of clusters beforehand.
- **Time Complexity**: Generally efficient for large datasets, with a time complexity of $O(n * \log n)$ where n represents the number of total data points.

However, DBSCAN also has limitations and drawbacks [28]:

- **Parameter Sensitivity**: DBSCAN is very sensitive to the *eps* and *min_samples* parameters, which need to be carefully chosen based on the dataset characteristics.
- **Difficulty with Varying Density**: DBSCAN struggles with datasets of varying densities resulting in poor clustering results.
- **Memory Usage**: Achieving $O(n * \log n)$ often relies on big memory consumption for large datasets.

4 Methodology

This chapter details the methodology and implementation of the proposed LiDAR-camera fusion method. It begins with an overview of the detection model, followed by an introduction to the dataset used for both development and validation. The implementation section provides a comprehensive explanation of the main steps in the model, including data preprocessing, calibration, the fusion technique, and 3D bounding box creation.

4.1 Overview

Figure 4.1 shows a simplified overview of the detection pipeline, divided into the three main processing parts *Camera*, *LiDAR* and *Fusion*. The model requires the image, corresponding point cloud and LiDAR-camera calibration files as input.

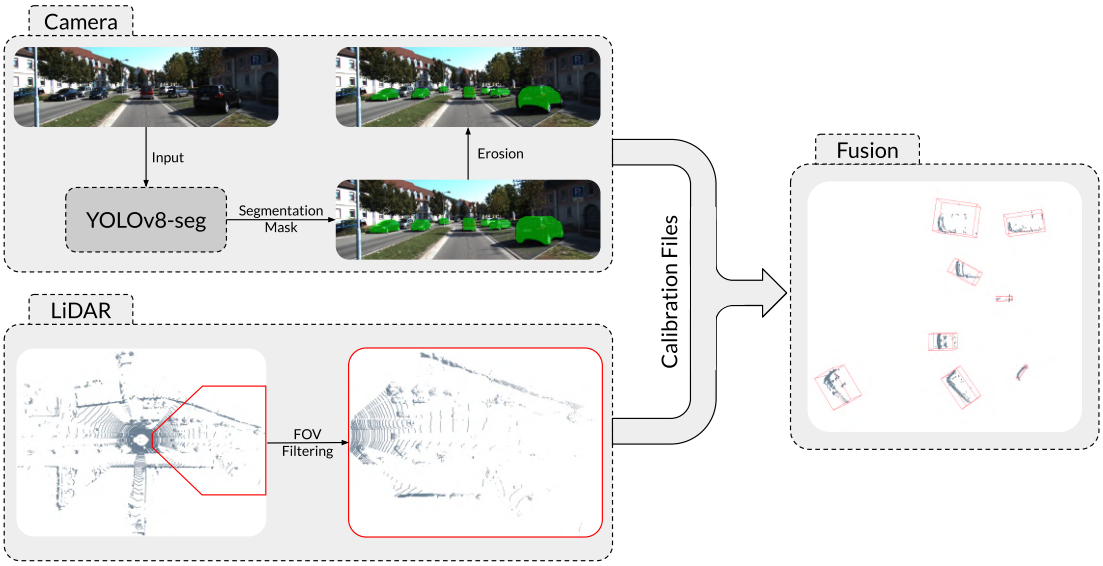


Figure 4.1: Detection Pipeline Overview

As one can see in Figure 4.1, first the image is fed into the YOLOv8-seg model to get the segmentation mask, label, confidence score and tracking ID for each detected object inside the image. The segmentation masks are then eroded to improve the fusion accuracy. The LiDAR point cloud, corresponding to the image, is filtered to only keep points that lie inside the Field-of-View (FOV) of the camera. In the fusion step, these filtered 3D points are projected onto the image plane using the transformation matrix obtained from the calibration files. For each segmentation mask, the projected points

that lie inside of it are identified and their corresponding 3D points are used to produce the 3D bounding boxes as output.

4.2 Dataset Presentation

The main dataset that was used during this project to test and develop the model is the **KITTI** 3D Object Detection dataset [29]. Although the KITTI dataset is already 12 years old as of the writing of this thesis, it is still popular and commonly used as a benchmark to compare different 3D object detection models. The dataset provides 7481 training images and 7518 test images along with their corresponding point clouds and calibration files. The training data contains the labels for each image which are obtained by manually labelling objects in the 3D point cloud and then projecting them back into the image [29].

Figure 4.2 shows the sensor setup of the car. There are two high-resolution colour and grayscale video cameras as well as a Velodyne laser scanner (LiDAR) and a GPS localization system. The cameras and LiDAR capture 10 frames per second (FPS) and the calibration files for the transformation between the different coordinate systems of the cameras, LiDAR and GPS are also given.

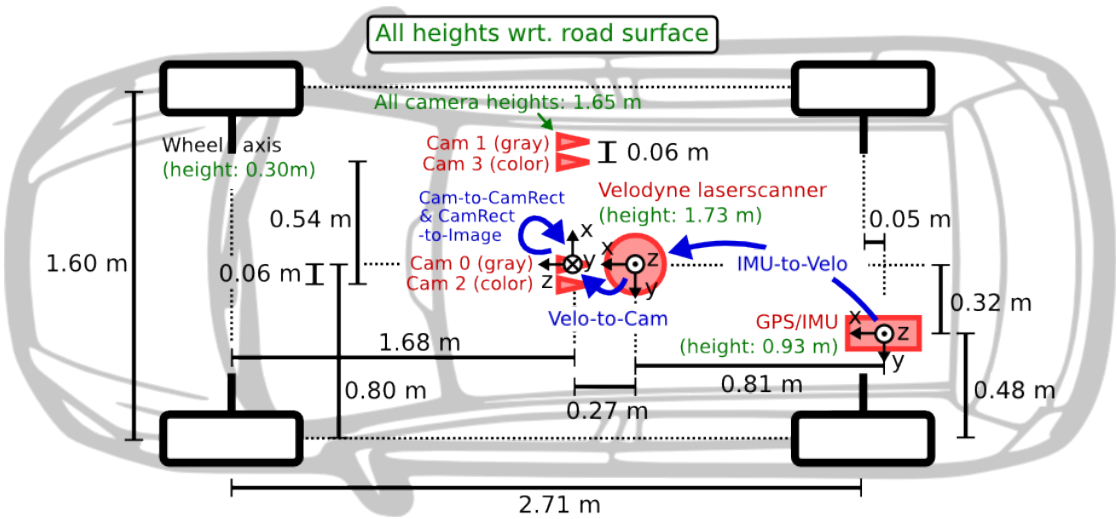


Figure 4.2: KITTI Sensor Setup Top View [8]

For this project, the only relevant data is produced by the center colour camera *Cam 2* and the LiDAR scanner (Velodyne laser scanner) which is also mounted in the center of the car. This is because the RoboCar testing vehicle is equipped with this specific setup: one central LiDAR and one front-facing color camera.

Figure 4.3(a) shows the point cloud that corresponds to the example image, cropped to a size of 1382×512 pixels. This image, shown in Figure 4.3(b) below the point cloud, was captured by *Cam 2*. The point cloud is shown in the Bird's-Eye-View (BEV) and the driving direction of the vehicle is to the top of the image.

(a) Example Point Cloud

(b) Example Image

Figure 4.3: Example Point Cloud and Image

4.3 Implementation

This section explains the implementation details of the model's main parts. The full code can be found in the following GitHub repository.

4.3.1 Data Pre-Processing

The first step of the model is to prepare the LiDAR and camera data used as input for the model. The image does not need any preprocessing as it is directly fed into the YOLOv8 segmentation model.

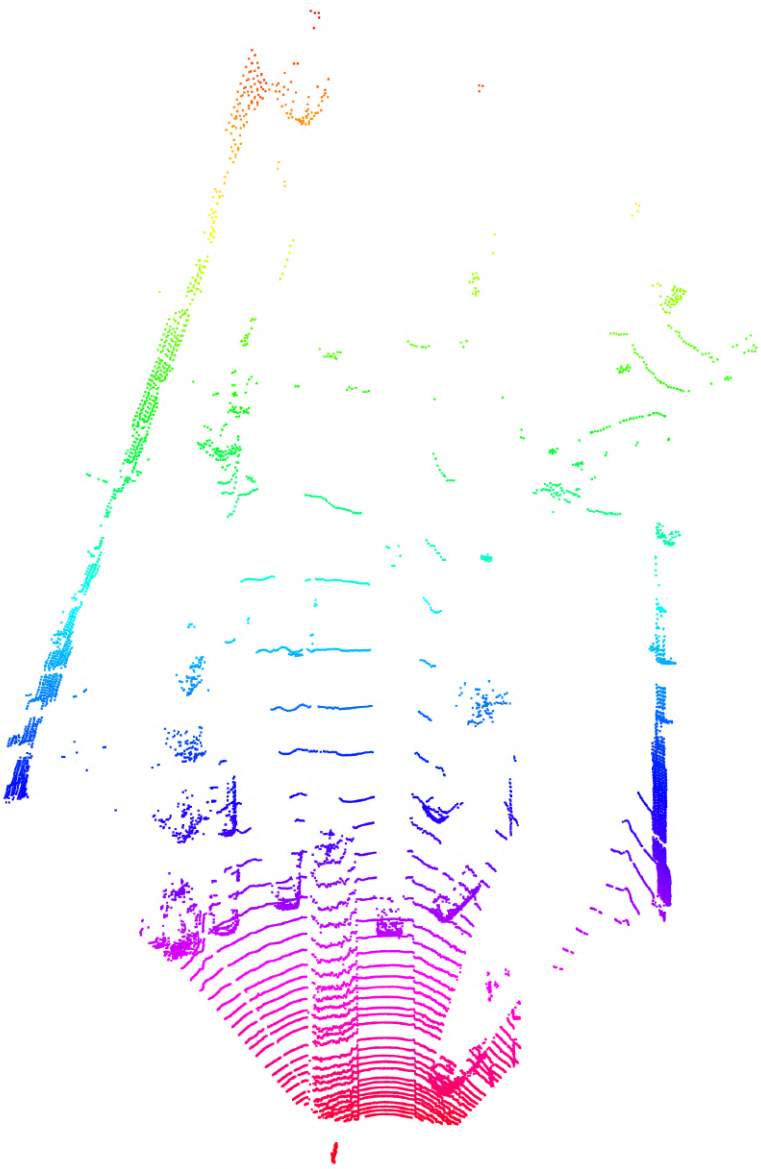
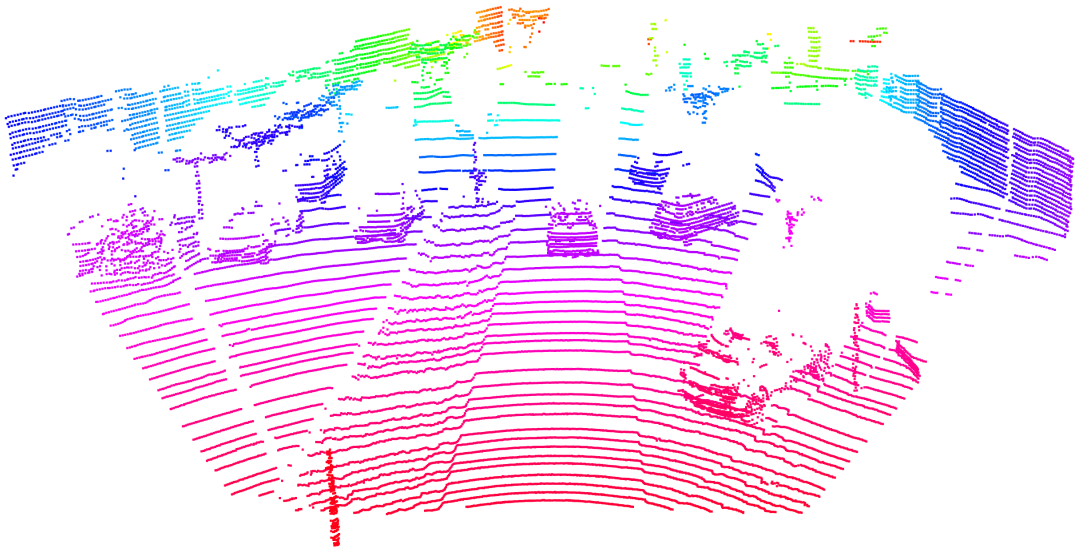


Figure 4.4: Filtered Point Cloud (BEV)

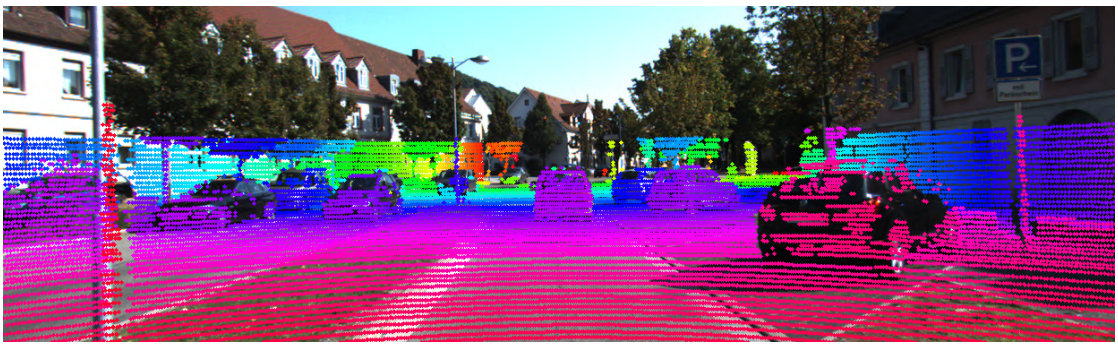
Filter Point Cloud

The point cloud however is filtered by a function such that only the points which lie inside the field of view of the image are kept, while all the others (e.g. the points

behind the car) are discarded. First, the image boundaries are defined, and then the 3D LiDAR points are converted into 2D image coordinates by a function that uses the calibration matrices between the LiDAR and camera sensors further explained in Section 4.3.2. These 2D points are then filtered using a boolean mask that checks if a point lies inside the image boundaries and has a specified minimum clip distance to the sensor to discard noisy and irrelevant data. At the end, the filtered 2D points as well as their corresponding 3D points are returned by the function. When using data from the KITTI dataset [29], this preprocessing step roughly shrinks the size of the point cloud from 122000 points to 18500 points which is a reduction of $\approx 85\%$ and thus improves inference speed. Figure 4.4 shows the point cloud of Figure 4.3(a) after the filtering is performed in BEV.



(a) Filtered Point Cloud (front view)



(b) Fused Image with filtered point cloud

Figure 4.5: Filtered Point Cloud

To better illustrate the filtered point cloud, Figure 4.5(a) shows the filtered point cloud from the front. This point cloud is the 3D version of the fused image, shown in Figure 4.5(b), obtained by completely fusing all the LiDAR points that lie inside the field of view of the camera. The colourmap represents distances from points: closer points are red, while further points transition through purple, blue, turquoise, green, orange and finally again red.

Segmentation Mask Erosion

The segmentation masks obtained from the YOLOv8-seg model are sometimes not entirely accurate, which then results in 3D background points being considered as object points. Figure 4.6 illustrates the erosion process of the segmentation mask by overlaying the original (green) and eroded (blue) masks. The mask as it is produced by the YOLOv8m-seg model is shown in green in the Figure. One can see that for some cars, the mask is too big, containing background information such as the road or walls behind the car. The erosion of this mask, shown in blue in Figure 4.6, shrinks the mask, but keeps its shape and thus helps to discard background points which obstruct the fusion process. For each detected object, the green part that is still visible in the Figure is thus discarded and the blue mask is the final eroded segmentation mask. Generally, each segmentation mask contains enough points such that even after the erosion and fusion steps, an accurate 3D bounding box prediction can still be made.

Figure 4.6: Illustration of Segmentation Mask Erosion

The model performs the erosion by firstly computing the area of the segmentation mask (polygon). An *erosion_factor*, which determines how much a polygon is eroded, has to be specified as input for the function. A higher erosion factor retains more of the segmentation mask's original area and thus results in less erosion. This parameter is scaled to the area of the polygon to ensure the amount of erosion is proportional to the size of the polygon. Equation 4.1 shows how the scaled erosion factor is computed. It prevents that small polygons are eroded more than large ones since the scaled factor will be smaller when the area of the polygon is smaller. A scaled factor of 0 would imply that the polygon is not eroded at all, however, this would mean that the polygon area has to be 0 which is never the case.

$$scaled_erosion_factor = \frac{\sqrt{polygon_area}}{erosion_factor} \tag{4.1}$$

Then, a binary mask is created where the polygon is filled with white pixels and the rest of the image is filled with black pixels. The erosion kernel size is then defined based on the previously computed scaled erosion factor. The erosion kernel is a small matrix that is used to consistently shrink the boundaries of regions in a binary image by removing pixels at the edges. Erosion is then applied to shrink the white area within the mask (polygon) based on the kernel and the contours of the new polygon are detected in the eroded mask. Contours are curves that connect continuous points along the boundary of

a shape that has the same colour, in this case white. The largest contour, based on the area of the polygon it outlines, is selected to ensure that the most significant part of the original polygon is preserved after erosion.

3D Point Filtering

After the fusion of 3D points with the eroded segmentation mask, there is still some noise in the point cloud. For each object, the 3D points that belong to the object are identified, however, due to reflection or obstacles, some of those 3D points do not belong to the object. In the segmentation masks of Figure 4.6(b), one can see that car windows as well as some obstacles, e.g. the street lamp in front of the 3rd car from the left in the image, are also part of the mask. The LiDAR laser can go through car windows or be reflected by them, causing points to be taken into account that are not on the surface of the car itself and thus should not be considered for the 3D bounding box creation.

(a) Fused 3D points with eroded segmentation mask without filtering

(b) Fused 3D points with eroded segmentation mask with depth filtering ($depth_factor = 15$)

Figure 4.7: Illustration of filtering 3D points after fusion

Figure 4.7(a) shows the fusion of 3D points that lie inside the segmentation mask without filtering them. As one can see on the car on the right side, some points that are in the background (blue and green coloured points) are considered to be part of the car. As this would negatively influence the 3D bounding box creation, depth filtering is applied. Figure 4.7(b) shows the fused points after the filtering process, removing some points but retaining the most important ones that truly belong to the detected object.

The filtering process uses the *DBSCAN* clustering algorithm to group points together based on their depth values and keeps the cluster that contains the most points. The DBSCAN algorithm requires two parameters: *eps* and *min_samples*. Therefore, the first

step of the function is to compute *min_samples* as either 1% of the number of points of a detected object or the value 5, whichever is greater.

Depth values are computed using the Euclidean distance of each point from the LiDAR scanner in the xy-plane, excluding the z-coordinate. This approach effectively captures depth information useful for filtering out background and noise points that are generally farther away. Initially, only the x-coordinate (pointing to the front) was considered, however, this approach had its limits when the objects were far on either side of the LiDAR scanner.

The next step is to apply the DBSCAN algorithm to the depth values by specifying a predefined *eps* value (passed as a parameter to the filtering function) and the computed *min_samples* value. The resulting clusters are then sorted in descending order based on their number of points and only the cluster with the most points is selected.

For this largest cluster, noise points, which are identified by the DBSCAN algorithm with the label -1, are discarded. For the largest remaining cluster, the depth values are analyzed to determine the minimum and maximum depths. These depth values are adjusted using a *depth_factor*, which is a parameter passed to the function. The *depth_factor* controls how aggressively the depth range is adjusted: a larger *depth_factor* results in a wider depth range, making the filtering less aggressive and retaining more points. Conversely, a smaller *depth_factor* tightens the depth range, making the filtering more aggressive and removing more points that fall outside this range.

The adjusted depth range is then used to create a depth mask. This mask filters the points in the selected cluster based on their depth values, ensuring that only the points within the specified depth range are kept.

4.3.2 Calibration

The calibration between the LiDAR and camera is a crucial step in order for the model to work accurately. For this purpose, the model uses a calibration object that contains all the necessary calibration matrices (transformation, rotation and projection). The object is constructed by the class *LiDAR2Camera* that contains a utility function from the KITTI development kit [8] that reads the calibration files and parses them into a dictionary.

The calibration matrices are used by a function that transforms 3D points from the LiDAR coordinate system into 2D points in the corresponding image frame. The function thus takes the *LiDAR2Camera* object and the 3D point cloud as input and outputs the corresponding 2D pixel coordinates in the image frame. The *LiDAR2Camera* class as well as this function was inspired by the *Visual Sensor Fusion* repository on GitHub [30].

The 3×3 matrix R, shown in Equation 4.2, represents the rotation from the reference camera coordinate system to the camera coordinate system that is used, i.e. *Cam 2* as explained in Section 4.2. This rotation matrix is given in the calibration file of the KITTI dataset.

$$R = \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{bmatrix} \tag{4.2}$$

The matrix R is converted into a 4×4 homogeneous transformation matrix R_{homo}, shown in Equation 4.3, by adding a row of zeros $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$ and then adding the column $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$.

$$R_{homo} = \begin{bmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \tag{4.3}$$

A homogeneous transformation matrix facilitates the computation, representation and concatenation of multiple transformations as it has a uniform size and can thus be combined easily with other homogeneous matrices. The 3×4 matrix P, shown in Equation 4.4, represents the projection from 3D camera coordinates to 2D image coordinates for the used camera. This projection matrix is given in the calibration files of the KITTI dataset. The left 3×3 portion of the matrix P corresponds to the matrix K_C in Section 3.1.1. The right 3×1 portion of the matrix P includes the translation vector (extrinsic parameters) of the used camera.

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \end{bmatrix} \tag{4.4}$$

The next step is to combine projection matrix P with the homogeneous transformation matrix R_{homo} by a dot product to obtain the 3×4 matrix PR, shown in Equation 4.5.

$$\begin{aligned} PR &= P \cdot R_{homo} \\ &= \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \end{bmatrix} \cdot \begin{bmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} PR_{00} & PR_{01} & PR_{02} & P_{03} \\ PR_{10} & PR_{11} & PR_{12} & P_{13} \\ PR_{20} & PR_{21} & PR_{22} & P_{23} \end{bmatrix} \end{aligned} \tag{4.5}$$

The 3×4 matrix $V2C$, shown in Equation 4.6, represents the transformation from the LiDAR coordinate system to the reference camera coordinate system. It thus compromises the rotation matrix (left 3×3 portion) and translation vector (right 3×1 portion) between the LiDAR and reference camera coordinate systems referred to as R and t respectively in Section 3.1.1. Hence, the matrix $V2C$ corresponds to the matrix T in Section 3.1.1. This transformation matrix is again given in the dataset's calibration file.

$$V2C = \begin{bmatrix} V2C_{00} & V2C_{01} & V2C_{02} & V2C_{03} \\ V2C_{10} & V2C_{11} & V2C_{12} & V2C_{13} \\ V2C_{20} & V2C_{21} & V2C_{22} & V2C_{23} \end{bmatrix} \tag{4.6}$$

The transformation matrix $V2C$ is converted into a 4×4 homogeneous transformation matrix $V2C_{homo}$ by adding the row $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$.

$$V2C_{homo} = \begin{bmatrix} V2C_{00} & V2C_{01} & V2C_{02} & V2C_{03} \\ V2C_{10} & V2C_{11} & V2C_{12} & V2C_{13} \\ V2C_{20} & V2C_{21} & V2C_{22} & V2C_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

The 3×4 matrix PR is then combined with the 4×4 matrix $V2C_{homo}$ by a dot product to obtain the 3×4 matrix T, shown In Equation 4.8, that transforms LiDAR coordinates directly to the image frame of the used camera.

$$\begin{aligned} T &= PR \cdot V2C_{homo} \\ &= \begin{bmatrix} PR_{00} & PR_{01} & PR_{02} & P_{03} \\ PR_{10} & PR_{11} & PR_{12} & P_{13} \\ PR_{20} & PR_{21} & PR_{22} & P_{23} \end{bmatrix} \cdot \begin{bmatrix} V2C_{00} & V2C_{01} & V2C_{02} & V2C_{03} \\ V2C_{10} & V2C_{11} & V2C_{12} & V2C_{13} \\ V2C_{20} & V2C_{21} & V2C_{22} & V2C_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \end{bmatrix} \end{aligned} \tag{4.8}$$

Now that the transformation matrix from LiDAR coordinates to the image plane is computed, the 3D points passed as input are converted into homogeneous points. Thus, a point $\begin{bmatrix}x & y & z\end{bmatrix}$ in Cartesian coordinates is represented as $\begin{bmatrix}x & y & z & 1\end{bmatrix}$ in homogeneous coordinates. This again facilitates the transformation process as now the dot product of the 3×4 transformation matrix T and the 4×1 transposed homogeneous coordinates of the 3D points $\begin{bmatrix}x & y & z & 1\end{bmatrix}^T$ can be computed as shown in Equation 4.9.

$$T \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{4.9}$$

This results in the 3×1 transposed homogeneous coordinates of the 2D points in the image frame $\begin{bmatrix}u & v & w\end{bmatrix}^T$. Finally, the u and v coordinates are normalized by dividing them by the w coordinate to convert from homogeneous to Cartesian coordinates and the x' and y' coordinates are extracted to get the 2D pixel coordinates as shown in Equation 4.10.

$$x' = \frac{u}{w} \quad , \quad y' = \frac{v}{w} \tag{4.10}$$

This step essentially performs the perspective division, projecting the 3D points onto the 2D image plane and finally receiving the 2D pixel coordinates $\begin{bmatrix}x' & y'\end{bmatrix}$ from the 3D LiDAR coordinates $\begin{bmatrix}x & y & z\end{bmatrix}$

4.3.3 Fusion

The fusion of the LiDAR point cloud and the corresponding camera image involves some data processing methods presented in Section 4.3.1. The fusion function takes as input the 3D point cloud as well as the corresponding 2D points obtained through the point cloud filtering, explained in Section 4.3.1, and projection explained in Section 4.3.2. It further needs the segmentation mask obtained from the YOLOv8-seg model and the original image that matches the point cloud. The fusion function is called for every detected object in the image and returns the 3D coordinates of the corners of the 3D bounding box of the detected object.

The first step of the fusion is to erode the segmentation mask as explained in subsection *Segmentation Mask Erosion* of Section 4.3.1. After the erosion, it is checked if the eroded segmentation is too small for further processing. If this is the case, the function returns and the current object is not fused with any points.

The main fusion process involves the identification of the 3D LiDAR points that are compromised in the eroded segmentation. To achieve this in an efficient way, first a single-channel greyscale mask of the same size as the image is created containing only black pixels, i.e. a 2D array with only zero values. The eroded segmentation polygon is then drawn on top of this grayscale image with an intensity of 1, meaning that every pixel in the region of the polygon gets the value 1 instead of 0 in the 2D array of the image. This effectively marks the region of the segmentation polygon in the image.

Next, the 2D coordinates, received as input, are converted from floating point to integer values for the next processing step. These integer coordinates are then used to check which points fall within the filled polygon on the mask. During this check, a boolean array is created that contains truth values which indicate if a pixel is inside the polygon or not. Using this boolean array, which basically contains the indices of the points that are inside the polygon, the 3D points are selected by simply indexing the array of 3D points. This indexing only works as the array of 2D and 3D points have the same lengths and order, meaning that a 2D point at index i in the 2D points array corresponds to the 3D point at index i in the 3D points array.

After the fusion is completed, all the 3D points that belong to the currently processed object are stored in an array. However, as mentioned in subsection *3D Point Filtering* of Section 4.3.1, there are still some points that do not belong to the object and thus should not be considered for the creation of the 3D bounding box. These points are then filtered with the depth filtering approach that employs DBSCAN, which is also explained in the subsection mentioned before. After the filtering, it is checked if there are still enough points to create an accurate 3D bounding box. This is the case if there are at least 4 points left and the points are not co-planar, i.e. have the same z-coordinates.

The entire fusion process, along with visualizations of each step, is illustrated in Figure 4.8. The sequence begins with the original image, followed by the projection of segmentation masks in the second step. These masks are then eroded in the third step. Thereafter, the 3D points are fused and projected onto the image. In the fifth image, these points are filtered and finally, the bounding boxes are created and projected onto the image.

Step 1: Original image that the proposed model receives as input.

Step 2: Retrieve the segmentation masks of every detected object from the YOLOv8 model.

Step 3: Erode every segmentation mask based on an erosion factor that influences the amount the mask is shrunk.

Step 4: Fuse every 3D LiDAR point of which its projection lies inside the eroded segmentation mask.

Step 5: Filter the fused 3D LiDAR points using the depth-based filtering that employs DBSCAN.

Step 6: Create 3D bounding boxes that enclose the filtered 3D LiDAR points using PCA without z-axis rotation.

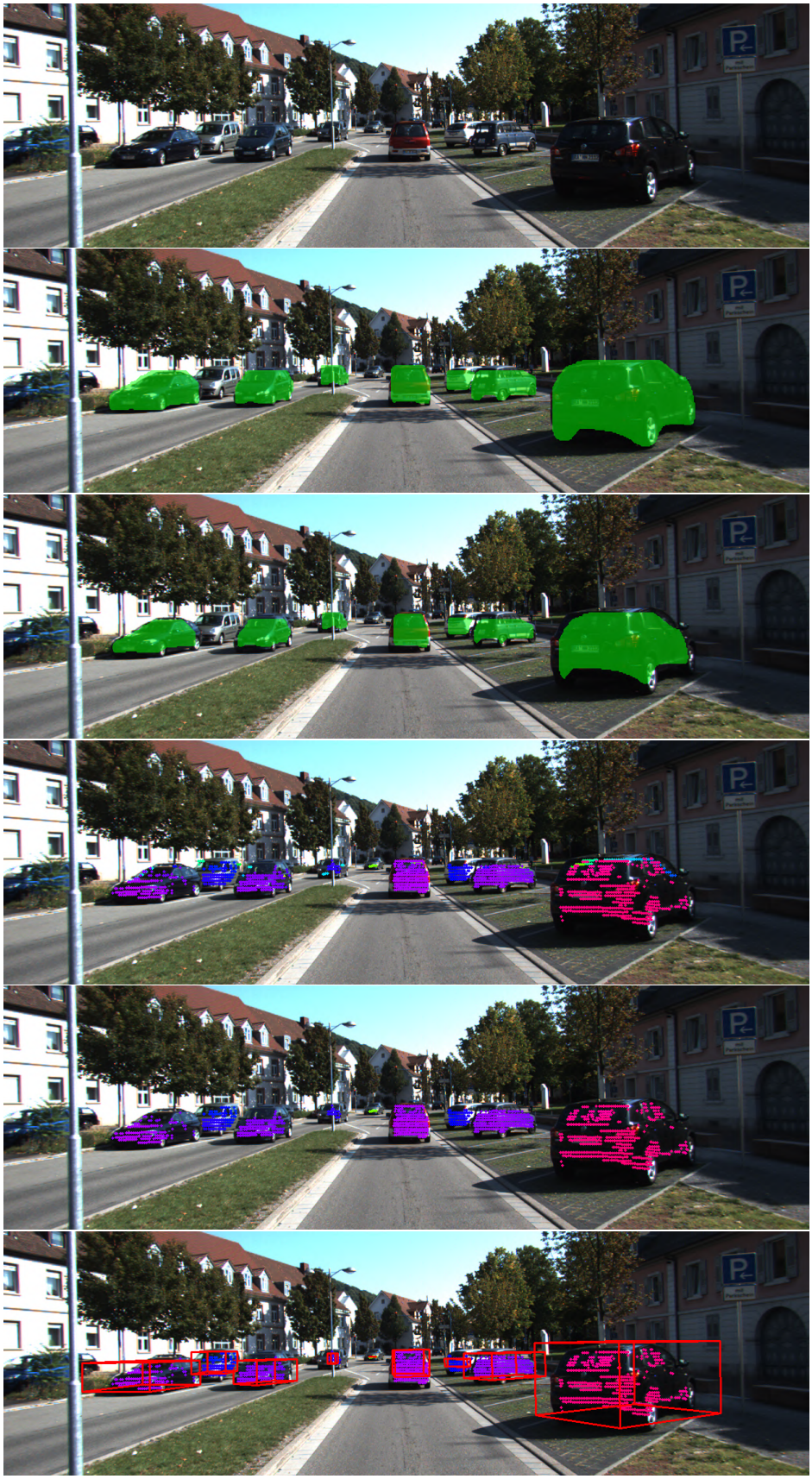


Figure 4.8: Illustration of each fusion step

4.3.4 3D Bounding Box Creation

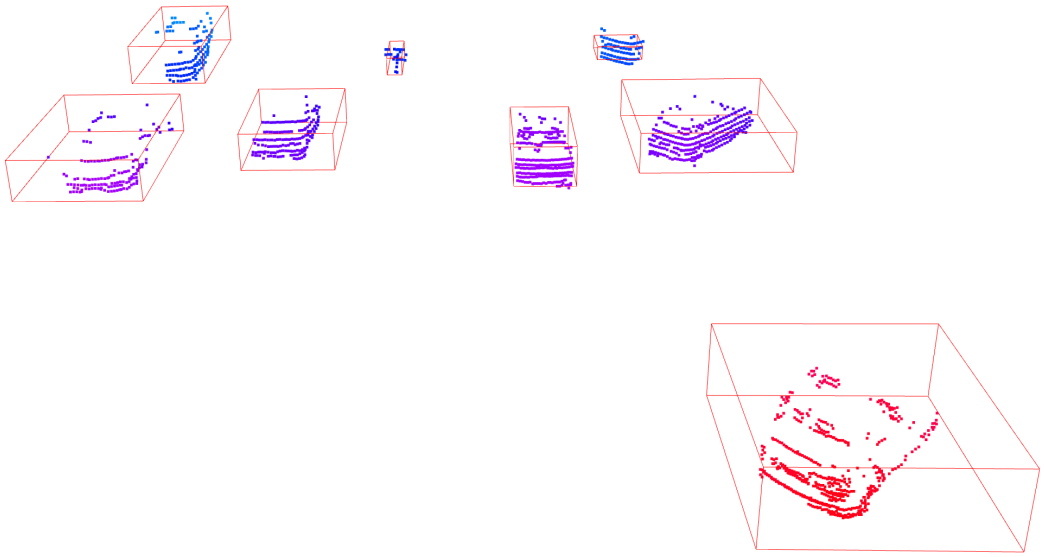
The fusion process itself is finished when all the 3D points of a detected object are identified and filtered. The last crucial step that remains to be performed by the model is the creation of the 3D bounding boxes. The model differentiates between the *person* class and the other classes, i.e. *car*, *motorcycle*, *bicycle*, *truck*, *bus*, *train*, for the bounding box creation.

The 3D bounding box of the *person* class is computed using *Principal Component Analysis* (PCA) without any further processing. PCA is a statistical technique that can be used to find the smallest possible bounding box that encloses a given set of 3D points. This technique thus gives oriented bounding boxes that have the smallest possible volume but still enclose all the 3D points. The difference between a 3D bounding box that is created without PCA and one that is created using PCA is shown in Figure 4.9 (a) and Figure 4.9 (b) respectively. As one can see, the bounding boxes that were created using PCA are more accurate and better represent the shape of the detected object.

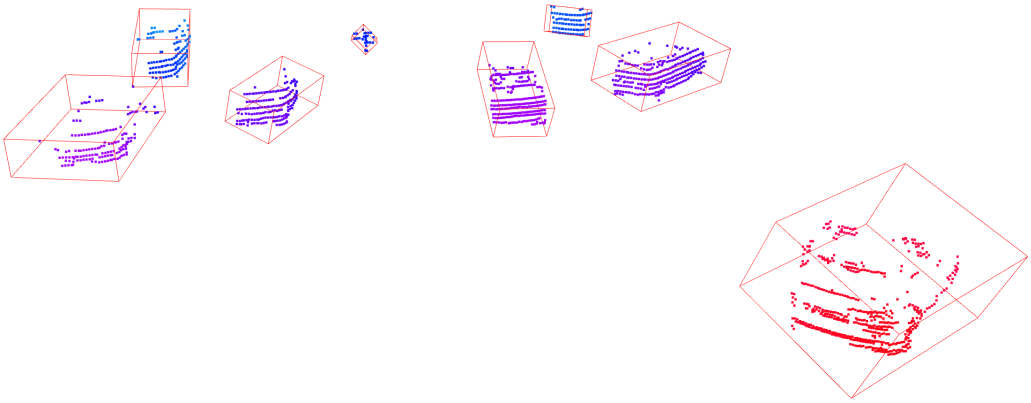
The model uses the *Open3D* library [31] in the function that creates the 3D bounding boxes. First, a point cloud is created from the array of 3D points provided as input. This array is the output of the fusion process after the depth filtering. Then, the built-in Open3D function *get_oriented_bounding_box()* handles the creation of the oriented bounding box using PCA. The 8 corners of this bounding box are then extracted and converted to 2D points using the function explained in Section 4.3.2. The conversion to 2D points is used to later create 2D edges that can be visualized on top of an image. However, the 2D points are only used for visualization purposes, the 3D coordinates of the bounding box are later used to compute the necessary parameters such as the center and dimensions of the bounding box.

The model's approach to create bounding boxes for classes other than *person* is similar but has one key difference, the rotation around the z-axis of the bounding boxes is discarded. The first steps to create a point cloud out of the array of 3D points and to create the oriented bounding box using the built-in Open3D function stay the same. However, the next step is to extract the rotation matrix from the oriented bounding box that describes the orientation of it along the 3 different axes. This matrix is then modified to only keep the x- and y-axis rotations and discard the z-axis rotation. Using the center and dimensions of the previously computed oriented bounding box as well as the adjusted rotation matrix, a new or oriented bounding box is created. From this step onward, the function is again similar to the usual PCA bounding box creation, thus, the 8 corners of the box are extracted, converted into 2D coordinates and both coordinates are returned. The difference between the oriented bounding boxes with and without the z-axis rotation are shown in Figure 4.9 (b) and Figure 4.9 (c) respectively.

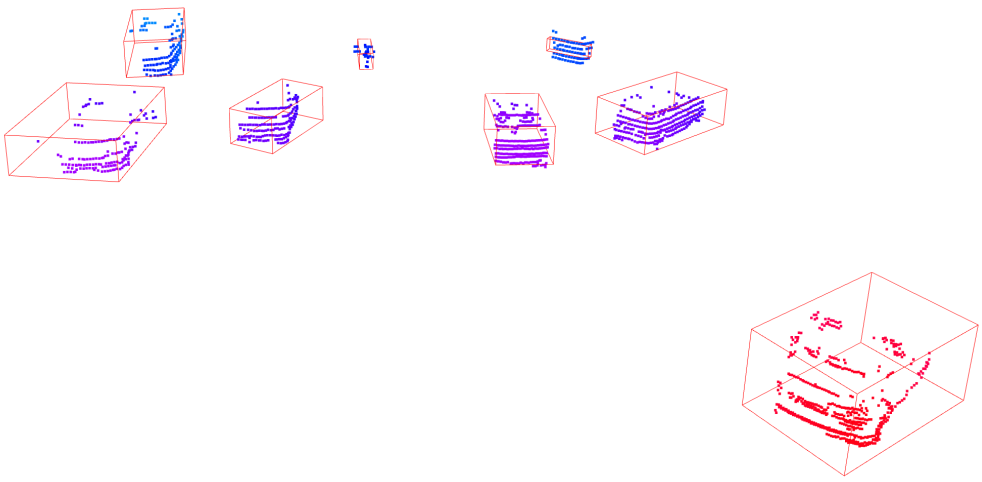
The model is only using PCA without z-axis rotation for classes other than *Person* as during the development of the model, it showed that vertical bounding boxes, such as *Pedestrian* bounding boxes, are not improved when discarding the z-axis rotation. Usually, they get smaller and thus provide a worse detection than before. The normal PCA bounding box creation function also needs fewer computational steps and is therefore faster and more resource-efficient.



(a) 3D bounding boxes



(b) 3D bounding boxes using PCA



(c) 3D bounding boxes using PCA without z-axis rotation

Figure 4.9: Illustration of different 3D bounding boxes

4.3.5 Tracking

The proposed model has the ability to track detected objects across multiple consecutive frames by using the tracking capabilities of the YOLOv8 model. This simplifies the implementation as instead of only detecting objects, one can also obtain tracking IDs from YOLOv8's output. Two tracking algorithms are supported:

- **BoT-SORT** [32] combined traditional methods, like the Hungarian Algorithm or IoU matching, with deep learning to improve the tracking accuracy of occluded objects and in crowded regions.
- **ByteTrack** [33] is a simpler tracking algorithm that combines high- and low-confidence detections to achieve a balance between reliability and robustness by using traditional methods such as motion and IoU matching.

The *BoT-SORT* algorithm is more complex and provides better accuracy in difficult scenarios while *ByteTrack* exceeds in efficiency and is effective in scenarios with clear high-confidence detections. Since the proposed model should provide the lowest possible inference time, to ensure live inference, the *ByteTrack* tracking algorithm was chosen due to its computational efficiency while still providing suitable accuracy.

Figure 4.10 shows 5 consecutive frames from top to bottom which illustrate the tracking capability of the proposed model. As one can see, three cars are correctly detected and given a tracking ID. The IDs are given in the order of detections, thus the objects with the highest confidence score get the first ID and so on. The YOLOv8 model includes a parameter (persist=True) that allows it to maintain the tracking IDs over multiple frames even when the model receives single frames (images) as input. This parameter is crucial for accurate and consistent tracking when given single images as input as it is the case for the proposed model during live inference.

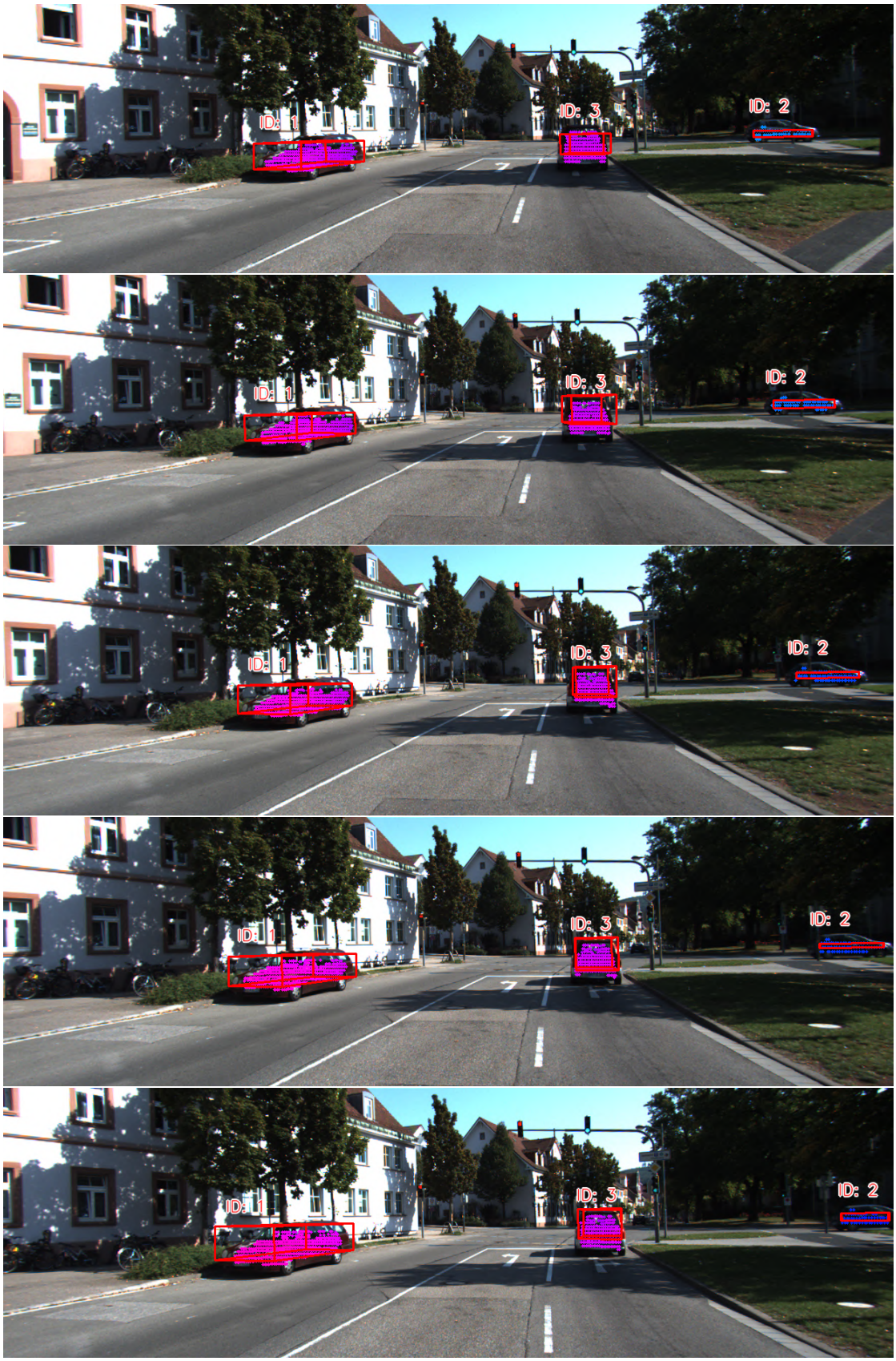


Figure 4.10: Illustration of Object Tracking

5 Evaluation

This chapter provides the evaluation of the proposed model. It starts with an introduction to the datasets used, specifically the KITTI and Audi Autonomous Driving (A2D2) datasets. Following this, the runtime parameter configuration process of the model is described by presenting the evaluation data collection techniques. Next, the evolution of the proposed model is illustrated with examples and the performance of each evolution step as well as of different YOLOv8 model sizes is compared. Finally, the proposed model is tested on both datasets, its integration in RoboCar is explained and some future work is presented.

5.1 Datasets

The general introduction of the **KITTI** 3D Object Detection dataset [29] containing example images and point clouds was given in Section 4.2. This section provides information on how the datasets for the evaluation and testing procedures were obtained.

5.1.1 KITTI Dataset

The KITTI dataset consists of 7481 training images and 7518 test images, each accompanied by their corresponding point clouds and calibration files. However, labels are provided only for the training set, which can be downloaded from the dataset's website [34]. Although the testing data can also be downloaded, it does not include the corresponding labels. This allows the KITTI team to ensure that the models, submitted by researchers, are tested on previously unseen data and that the obtained results can be compared to the other submitted models. As the KITTI team updated their policy to only accept *submissions with significant novelty* that are part of a *peer-reviewed paper* and to not accept *student research projects* anymore [34], I decided to evaluate and test my model on a split of the training dataset.

The training dataset was split into a validation dataset and a testing dataset that contain 3740 and 3741 matching images and points clouds respectively. The splitting was performed by using every second image, point cloud and calibration file in the KITTI training set for the new validation set and the remaining images for the testing set. This approach gives enough data to validate the model and later test and compare it to the state-of-the-art (SOTA) models available on the KITTI website. However, since the proposed model of this thesis and the SOTA models were not tested on the exact same dataset, the results may be not entirely valid for comparison.

KITTI Evaluation

The KITTI dataset provides a development kit [8] that contains tools and explanations on how to evaluate the model's 3D object detection results on the KITTI dataset.

For each processed frame, the model has to store the output in a text file that contains the following information for each detected object:

- **Object class**: describes the type of the detected object. The evaluation script only takes into account the classes *Car*, *Pedestrian* and *Cyclist*. The other classes are marked as a separate class *DontCare* and are ignored for the valuation process.
- **2D bounding box**: contains the 2D pixel coordinates of the top-left and bottom-right corners of the 2D bounding box that is projected onto the image from the 3D bounding box.
- **3D bounding box dimensions**: consist of the height, width and length of the 3D bounding box in meters.
- **3D bounding box ground center**: contains the center of the 3D bounding box in LiDAR coordinates.
- **Confidence score**: defines the probability that the detected object belongs to the specified class as a floating number (higher is better).

The object class and confidence score are both taken from the YOLOv8-seg model output. The object class is only refactored to the exact string that the KITTI evaluation script expects. The bounding box information is obtained from the fusion process of the model.

The evaluation script from the KITTI development kit did not work correctly as it opened the KITTI website after completion where the results should be displayed. However, this resulted in an error that could not be solved as the documentation provided to run the evaluation script is very limited. For this reason, I decided to use a GitHub repository [35] that is built upon the KITTI development kit and provides a better interface and documentation. This repository follows the official KITTI evaluation criteria [34]:

- Objects that are smaller than 25 pixels are considered too far away and thus not counted for the evaluation.
- The class *Van* is not counted as false positive for *Car* and *Sitting Person* is not counted as false positive for *Pedestrian* due to their similarity.
- In order to be counted as a true positive for the object class *Car*, the IOU of the detected 3D bounding box and the ground truth box has to be at least 70%. For the classes *Pedestrian* and *Cyclist*, the IOU only has to be 50% to be counted as a true positive.

The KITTI dataset also provides 3 different difficulties for their 3D object detection benchmark. Each object in each frame of the dataset is ranked to a specific difficulty based on some criteria [34]. Table 5.1 provides an overview of the criteria used to categorize objects into the different difficulty levels. The minimum bounding box height refers to the projected 2D bounding box of the object and thus refers to how far away an object is. The maximum occlusion level describes how visible an object is, i.e. if it is fully visible in the image or if it is blocked by another object or by a bush for example. The last criterion is the maximum truncation which defines the amount that an object is cut off by the image boundaries.

Difficulty	Min. bounding box height	Max. occlusion level	Max. truncation
Easy	40 Px	Fully visible	15%
Moderate	25 Px	Partly occluded	30%
Hard	25 Px	Difficult to see	50%

Table 5.1: KITTI difficulty levels criteria

Hence, for each of the 3 object classes that are used in the evaluation process, there are 3 different difficulty levels. As the KITTI dataset uses *Average Precision* (AP) as the metric to evaluate the detection performance of models, the final evaluation will have 9 different AP values. Since the YOLOv8-seg model was not trained to detect cyclists, I decided to discard this class for the evaluation of my model. I tried to use detected bicycles as the *Cyclist* class for the evaluation, however, this resulted in very bad results as often stationary bicycles on the side of the road were then counted as cyclists.

5.1.2 Audi Autonomous Driving Dataset

The Audi Autonomous Driving Dataset (A2D2) [36] provides a newer and more complete dataset for 3D object detection by providing over 40000 frames with semantic segmentation image and point cloud labels. Out of these labelled frames, 12497 front-camera images along with their matching 3D point clouds also contain labelled 3D bounding boxes using 14 classes. The data was collected by using an Audi Q7 e-tron SUV equipped with six cameras and five Velodyne (LiDAR) sensors shown in Figure 5.1. For this thesis, only the *Cam front-center* and the *LiDAR front-center* are used as the model was designed to work with only one camera and one LiDAR.

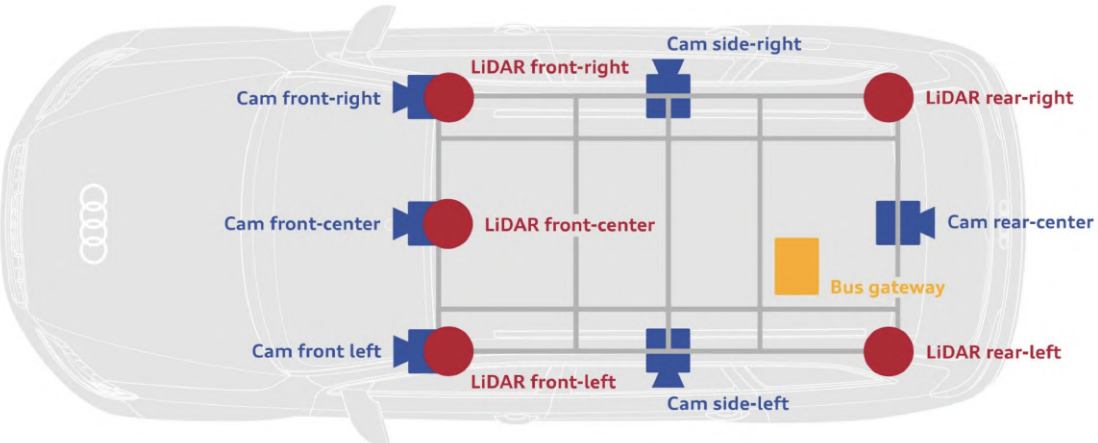


Figure 5.1: A2D2 Sensor Setup [9]

One main difference between the LiDAR scanner used for the A2D2 dataset is that it only has 16 channels, whereas the LiDAR scanner used for the KITTI dataset has 64 channels. This is probably due to the fact that the A2D2 dataset uses in total 6 LiDAR scanners and the KITTI dataset only uses one LiDAR scanner. The 16-channel LiDAR

scanner has the disadvantage that it only uses 16 laser beams to scan the environment which results in lower resolution and data density. This means that fewer data points are collected per scan and thus the resulting point cloud is coarser with less detail. The 64-channel LiDAR scanner used by KITTI produces a much denser point cloud with much finer and more details as it uses 64 laser beams to scan the environment. Figure 5.2 shows an example of a fused image using the front-center camera and LiDAR sensors of the A2D2 dataset. As one can see the point cloud is much sparser than it is the case for the KITTI dataset when comparing it to Figure 4.5(b) from Section 4.3.

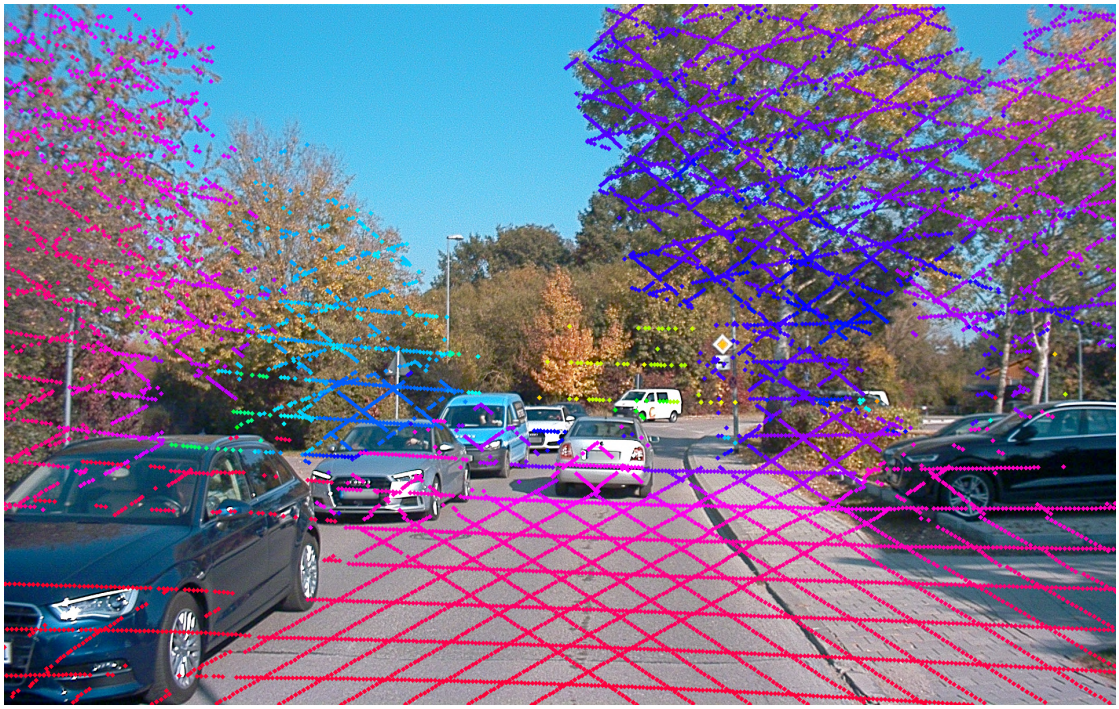


Figure 5.2: Example of fused image from A2D2 dataset

The complete A2D2 dataset is used to evaluate the proposed model on another dataset than KITTI. This should demonstrate how the proposed model performs on a different dataset that has other sensors to capture the images and point clouds. Further, when the performance of the model is more or less the same as for the KITTI dataset, one can conclude that it generalizes well. The results of both testing datasets are shown in Section 5.5.

5.2 Model Configuration

The proposed model was not trained during this thesis as the only machine-learning part is the YOLOv8-seg model which is pre-trained on the *Microsoft Common Objects in Context* (COCO) dataset [27] that provides 328000 labelled images for instance segmentation with 91 different categories. Since the most important categories for object detection in autonomous vehicles, e.g. car, pedestrian, bicycle and truck, are present in this dataset, there is no need to retrain the YOLOv8-seg model. All results in the following sections, including parameter configuration, model evolution and the final results, were accomplished using the medium size of the YOLOv8 model if not otherwise stated.

However, the proposed model includes two runtime parameters that influence the erosion and filtering processes, detailed in Section 4.3.1. These parameters can significantly influence the model's performance:

1. **Erosion Factor**: This parameter influences the *Segmentation Mask Erosion* process. Thus, it defines how much a mask is eroded and therefore defines how many points are taken into account during the fusion process.

2. **Depth Factor**: This parameter influences the aggressiveness of the *3D Point Filtering*. It defines the allowable range of distances from the LiDAR scanner to a 3D point within which a point can be considered part of the detected object. Thus, it adjusts the maximum and minimum distances, determining whether a point should be included in the object cluster or filtered out. By widening or narrowing this range, the depth factor specifies how aggressive the filtering process is.

5.2.1 Model Evaluation

During the configuration process, the runtime parameter combinations for the proposed model were evaluated using the 3D Intersection-over-Union (IoU) of the ground truth (GT) bounding boxes from the validation set and the predicted bounding boxes. This value is computed by using the ground center, size and y-axis rotations (yaw) of each bounding box in 3D camera coordinates. The oriented bounding boxes are then converted to axis-aligned boxes to compute their IoU dividing their intersection volume by their union volume.

For each detected object in a frame, every GT bounding box of this frame is compared to the bounding box of the detected object. The detected box with the highest IoU is selected as the correct match for the GT box. These IoU values are then saved such that one obtains a list of IoU values between matching GT and prediction boxes for each frame of the dataset provided as input. In the cases where the proposed model predicts objects with no GT label, the prediction box is simply discarded as the IoU computation is only performed for GT boxes and their corresponding prediction boxes.

For each frame, the script saves the best IoU in the specific frame, thus only one IoU value that corresponds to the most accurate predicted bounding box, and also an average IoU score that is computed from the IoU of all matching GT and prediction boxes. This value is usually lower as it takes into account every predicted box which includes sometimes inaccurate predictions.

5.2.2 Data Collection

The runtime parameter configuration process is performed by testing the model with all possible combinations for erosion and depth factors in a specified range and collecting the IoU scores for each combination. First, this range was defined through experiences made during the development of the model, thus, by testing out which values are best for some specific image and point cloud pairs. By looking at how the model's performance evolved for these values, the range was enlarged to get the best possible IoU. This process was then repeated until another enlargement of the range of possible values for the runtime parameters did not improve the accuracy of the model.

During the parameter configuration, the model was run on the validation set obtained from the KITTI dataset as explained in the previous Section 5.1.1. Since this set contains

3740 matching images and point clouds that all have to be processed, the model was run on the UBIX PC platform to minimize the runtime. The UBIX PC has the following hardware specifications:

- **CPU**: Intel Core i9-13900K (24 Cores with 5.5GHz)
- **GPU**: NVIDIA GeForce RTX 4090 (24GiB GDDR6X RAM)
- **RAM**: 98GiB

The execution of the model takes around 2016MiB of GPU RAM as the YOLOv8-seg model can be run on the GPU for faster inference time. The execution for one combination of the parameters on the whole validation set takes around 820 seconds (13 minutes and 40 seconds).

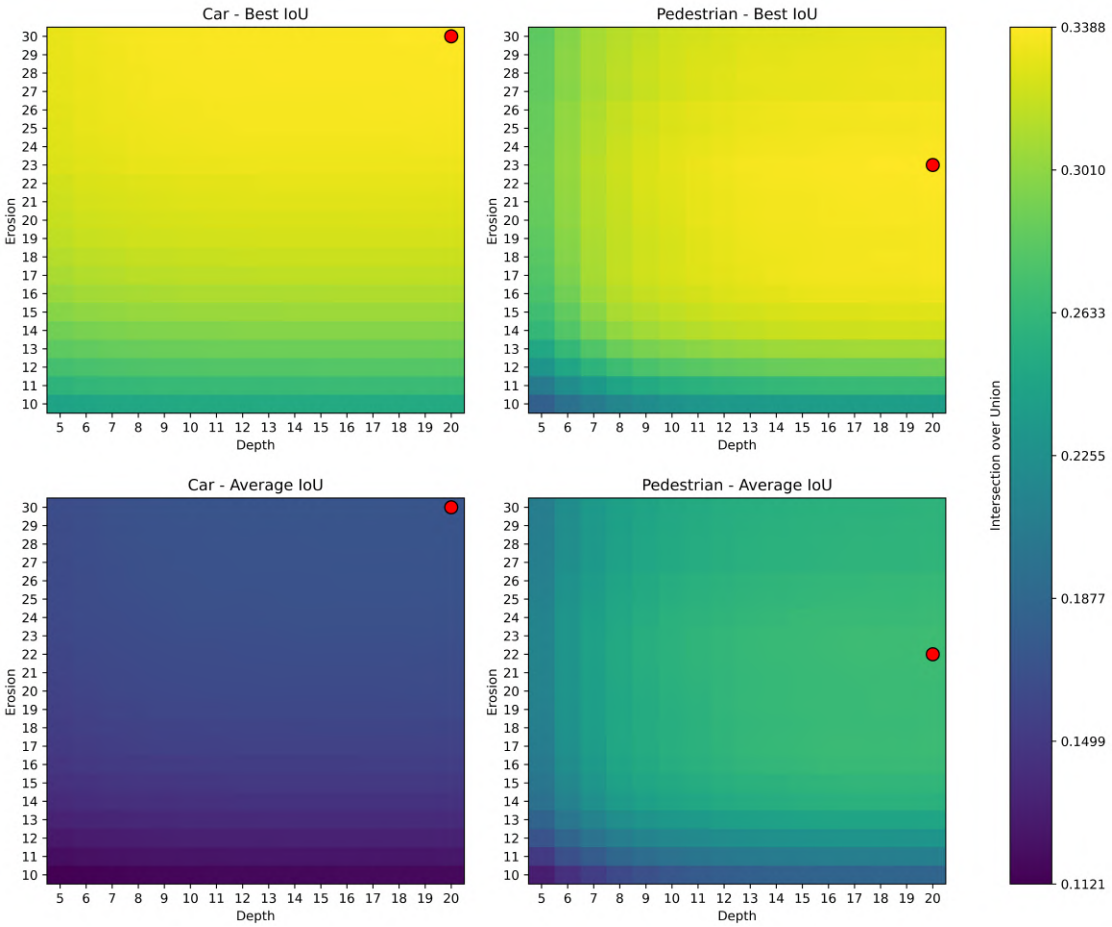


Figure 5.3: Intersection-over-Union of runtime parameter combinations

Figure 5.3 illustrates the IoU values for the different combinations of parameters, i.e. erosion and depth factor, in a heatmap graph. The erosion factor ranges from 10 to 30 and the depth factor ranges from 5 to 20. The heatmaps on the left side illustrate the IoU values for the *Car* class and the ones on the right side correspond to the IoU scores for the *Pedestrian* class. The upper row represents the *best IoU* score in each frame, thus it represents the IoU scores for the most accurate 3D bounding box in each frame of the validation set. The heatmaps in the lower row of Figure 5.3 show the *average IoU*

values of all the 3D bounding boxes in each frame, thus representing the general IoU scores over all detected objects for the specific class in all frames of the validation set. Both values are saved for each frame and then averaged over all validation set frames to obtain an IoU score that represents the overall performance of the model. The x-axis represents the depth factor and the y-axis represents the erosion factor. The scale on the right side indicates which colour corresponds to which IoU value. The darker the colour gets, the worse the model performed for the specific combination of parameters. The *Cyclist* class was discarded from the evaluation as the model classifies all detected bicycles as cyclists which results in a lot of wrong predictions for this class and thus a very low IoU score.

When plotting the results in a line chart, one can better observe that the curve flattens out at the end. Thus, I decided that further runtime parameter configuration would not bring much more benefit, even though for the *Car* class, the best IoU scores were achieved for the last tested combination of parameters. Based on the collected data, the seemingly best combination of parameters can be identified, illustrated by the yellow colour. The model achieves an IoU score for the validation set of around 33.8% for both object classes when considering only the best predicted bounding boxes for each frame and of around 27% for *Pedestrian* and 17.5% for *Car* when averaging all predicated boxes. For each heatmap, the red dot indicates the location of the best IoU value, thus the best combination of runtime parameters.

Figure 5.3 further shows that as the erosion factor increases, i.e. the segmentation masks of the detected objects are less eroded, the IoU also increases. The depth factor plays a bigger role for the *Pedestrian* class since it impacts its IoU positively when it increases. For cars, the depth factor only marginally increases the IoU score. Hence, the erosion factor has an overall bigger impact on the model's performance for detecting cars than the depth factor. In the heatmaps in the second row of Figure 5.3 one can see that when taking all 3D bounding boxes of every detected object over the whole validation set into account, the IoU score approximately halves for the *Car* class, but it decreases less for the *Pedestrian* class. An explanation of this could be that the bounding boxes of pedestrians are generally smaller and more representative points can be collected. In the case of cars, sometimes the angle of detection only allows to fuse points of one surface of the car and thus the predicted bounding box does not represent the whole vehicle.

Overall, from the results illustrated in Figure 5.3 one can conclude that an erosion factor of around 25 is generally good for both object classes, even when not considering the depth factor. Furthermore, a higher depth factor, above 15, generally performs better. Thus, a combination of parameters that works well for both object classes would be an erosion factor of 25 and a depth factor of 20.

5.3 Model Evolution

This section demonstrates how different modifications to the model impact its performance. Figure 5.4 shows the original image that will be used to illustrate the model's evolution. The image was taken from the KITTI dataset and contains cars, persons and bicycles and can thus well demonstrate the strengths and weaknesses of each modification done to the baseline model. Further, it shows a typical situation that an autonomous car will face in a residential environment.

Figure 5.4: Original Image to demonstrate model evolution

5.3.1 Baseline Model

The first model that was developed during this thesis is inspired by the code of the GitHub repository *Visual-Sensor-Fusion* [30]. The YOLOv8 object detection model is used to get the 2D bounding boxes for each detected object in a frame. As depicted in Figure 5.5 the most important objects, i.e. nearby cars, persons and bicycles, are detected by the YOLOv8 model. Only vehicles that are far away are not detected, however, these are not relevant in this frame. The 2D bounding boxes are directly fed to the fusion

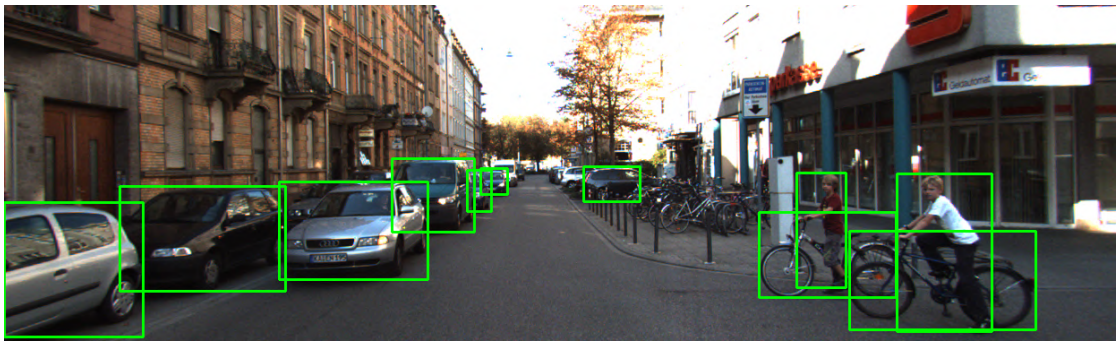


Figure 5.5: 2D Bounding Boxes from YOLOv8

function that checks if a point lies inside of it. Every point that is identified to belong to a detected object in this way is thus used to create the 3D bounding boxes without any pre-processing like filtering. As illustrated in Figure 5.6 the model does not fuse

Figure 5.6: Fused Image from Initial Model

only 3D LiDAR points that really belong to the detected objects. This results in a lot of noise points being considered for the 3D bounding box creation. Figure 5.7 shows the

3D bounding boxes projected onto the image.

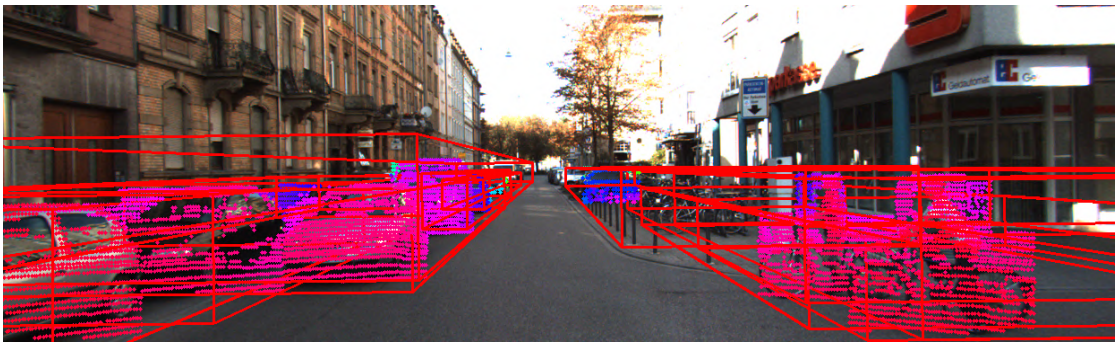


Figure 5.7: Projected 3D Bounding Boxes from Initial Model

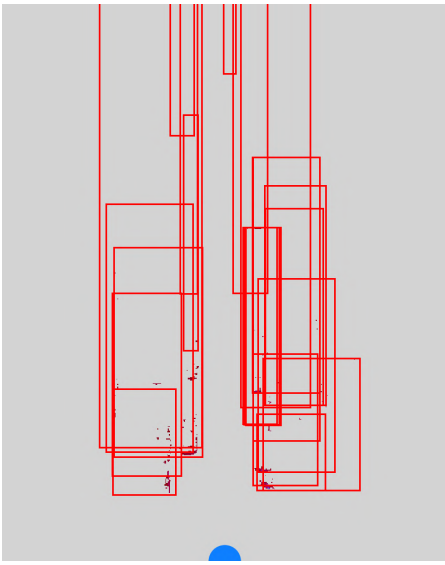


Figure 5.8: BEV Baseline Model

For each detected object, every 3D point that, when projected onto the image, lies inside the 2D bounding box obtained from the YOLOv8 model is considered for the 3D bounding box creation. This means that the boundary points of the set of 3D points for a specific object are used to define the corners of the 3D bounding box. As one can see, this results in bounding boxes that are very big and do not correspond at all to the size of their corresponding detected objects. Figure 5.8 gives a better overview of these boxes by representing them in the Bird's-Eye-View (BEV). The blue dot at the bottom of the image represents the car and the 3D point cloud of the fused points is displayed as small purple points.

5.3.2 Modification 1: Include Segmentation

The first modification done to the baseline model was to add segmentation. This was achieved by using the YOLOv8-seg model to detect objects inside the input image. This

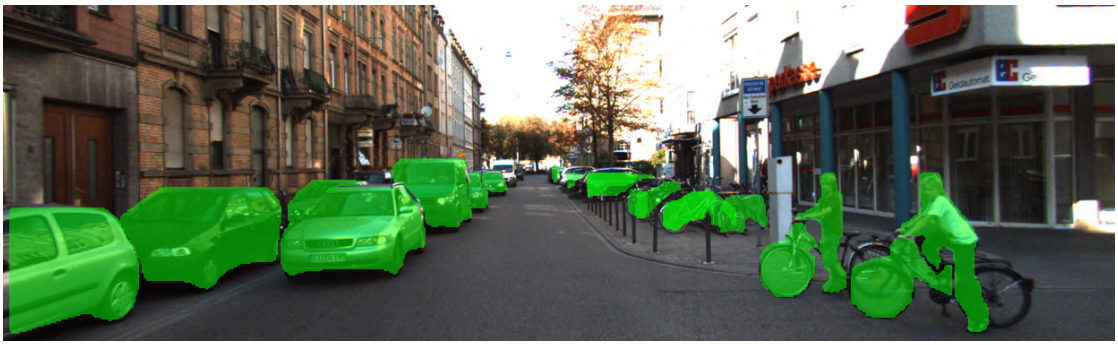


Figure 5.9: Segmentation Masks from YOLOv8

model returns a segmentation mask for every detected object along with its class and confidence score. The segmentation masks obtained for the example image are shown in Figure 5.9 in green. One can observe that the masks for the car and person classes are pretty good already. Further, the bicycles in the background on the right side of the image, which were not detected by the YOLOv8 detection model, are now identified by the segmentation model. However, the bicycles are not entirely segmented and the quality of their segmentation is not very good. The fusion process of this model iteration is very similar to the baseline model.

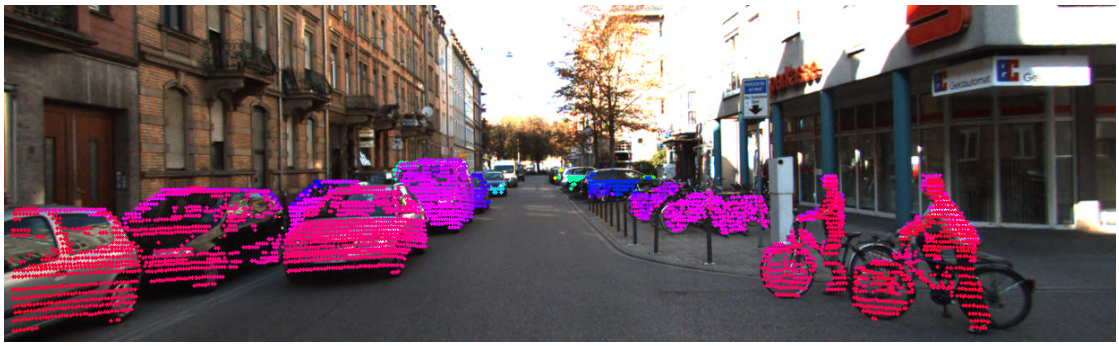


Figure 5.10: Fused Image after adding Segmentation

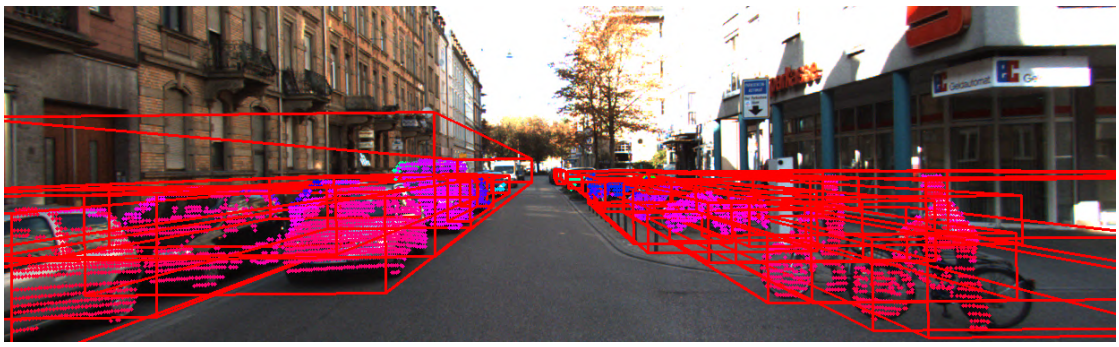


Figure 5.11: Projected 3D Bounding Boxes after adding Segmentation

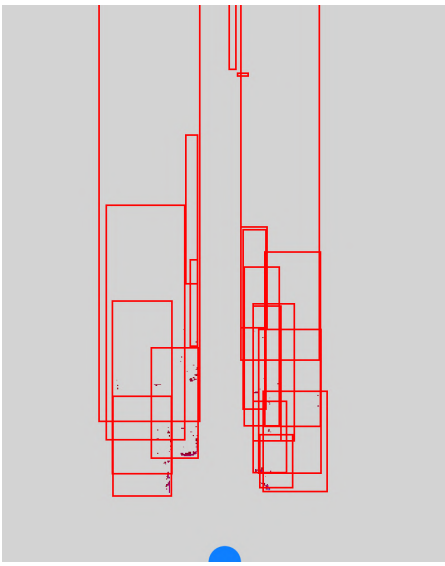


Figure 5.12: BEV Segmentation

A boolean list containing truth values that indicate if a point lies inside the segmentation or not is created by using a greyscale mask of the segmentation as explained in Section 4.3.3. The corresponding 3D LiDAR points are then stored and projected onto the image to create Figure 5.10. The stored 3D LiDAR points are again used without any pre-processing to create the 3D bounding boxes around the detected objects exactly as for the baseline model. Thus, the only difference between the modified model and the baseline model is the segmentation of detected objects. The results of this modification, when comparing Figures 5.6 and 5.10, are very promising as a lot fewer noise points are taken into account for each detected object. However, Figure 5.11 demonstrates that the 3D bounding boxes do not improve much. Again a better

overview and also comparison of the 3D bounding boxes between the baseline model and this modification is shown in Figure 5.12.

5.3.3 Modification 2: Erode Segmentation Mask

In order to refine the segmentation masks of the YOLOv8-seg model, the segmentation polygon can be eroded. By applying a specified erosion factor, the polygon is shrunk as explained in Section 4.3.1. As illustrated in Figure 5.13 the green eroded segmentation masks reduce the area of the original masks shown in Figure 5.9 by reducing their area. This reduction prevents noise points around the mask borders from being included during the fusion process. An erosion factor of 20 was chosen for this comparison based on the runtime parameter configuration presented in Section 5.2. Figure 5.14 shows the fused points when projected onto the image. When comparing to Figure 5.10, one can see that fewer points around the borders of the detected object have a different colour, meaning that fewer points from the background are included in the fusion. The 3D bounding boxes, as shown in Figure 5.15, do not improve much when the segmentation masks are eroded.

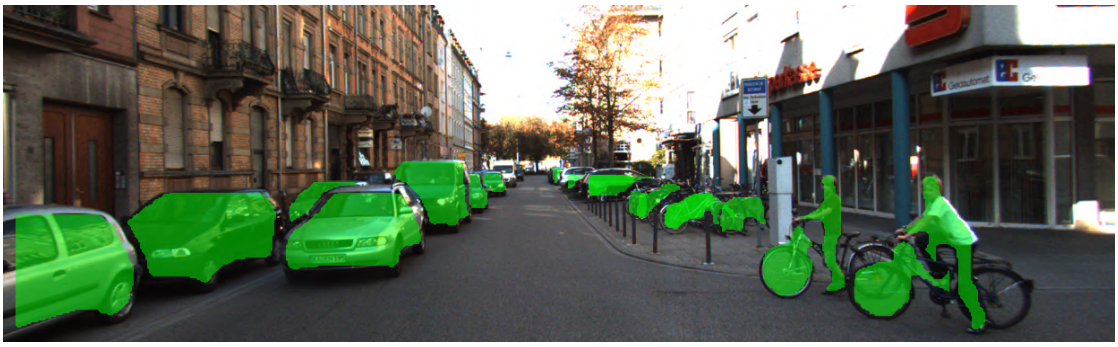


Figure 5.13: Eroded Segmentation Masks (erosion_factor = 20)

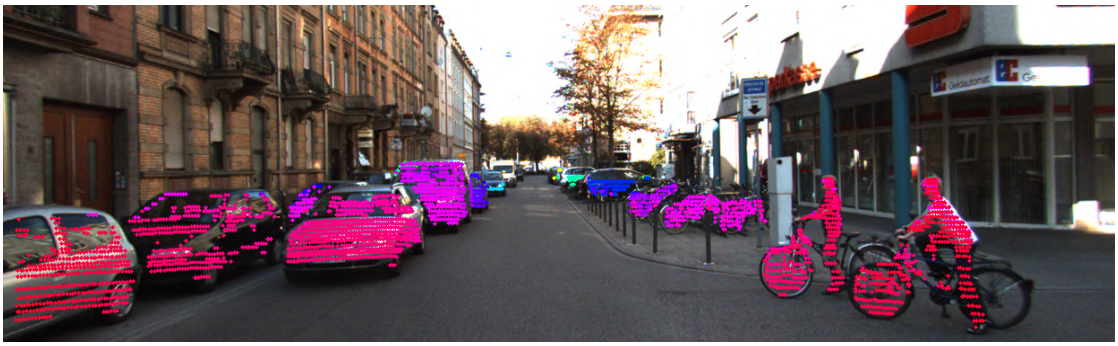


Figure 5.14: Fused Image with Eroded Segmentation

However, there is still a minor improvement when looking at the first cars on the left side. Figure 5.16 shows the impact of decreasing the erosion factor. As already mentioned the figures were produced using an erosion factor of 20, however, to create Figure 5.16 (a), an erosion factor of 10 was used. As the erosion factor gets smaller, the segmentation masks also get smaller, thus they are eroded more. For this example frame and model evolution step, a lower erosion factor produces better visual results.

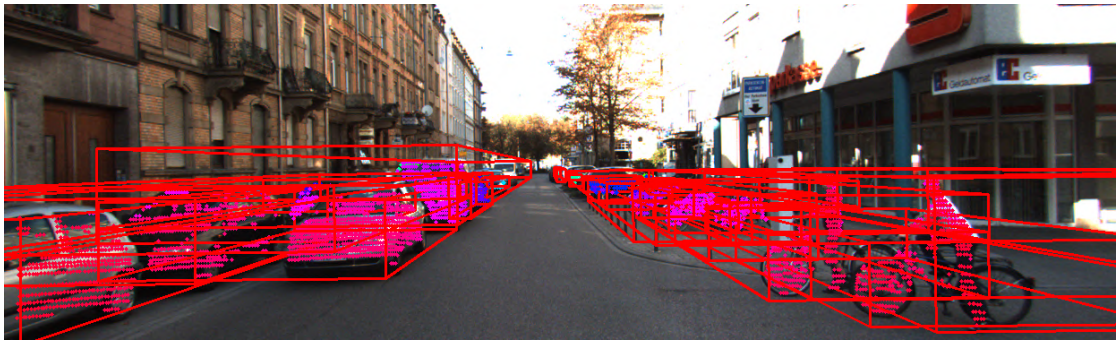
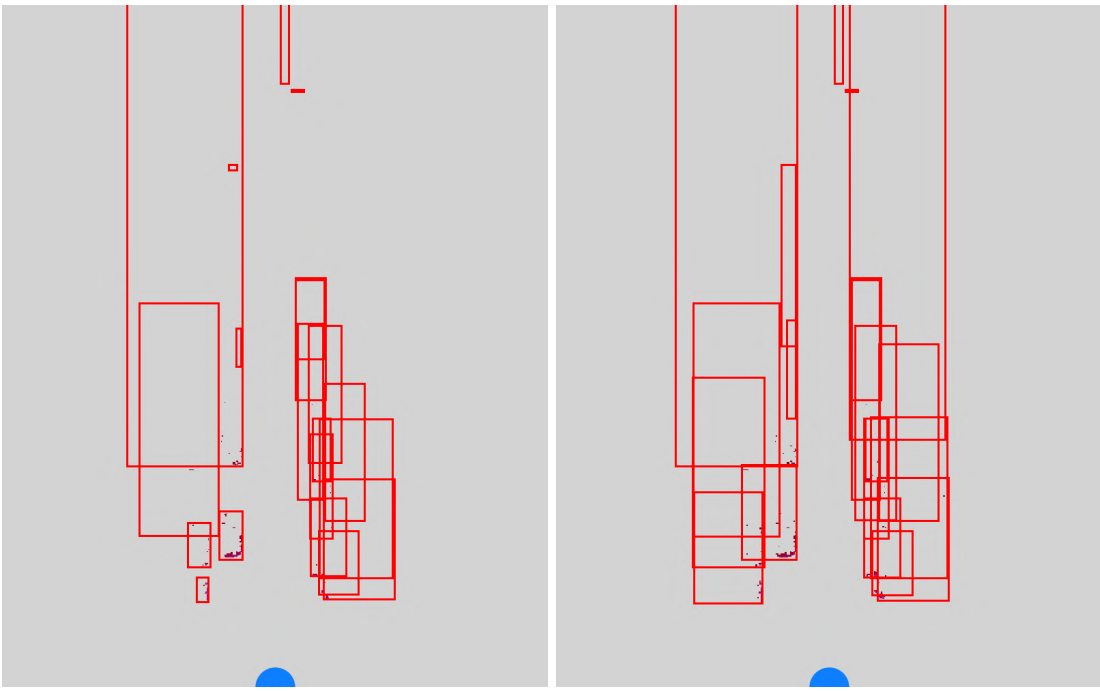


Figure 5.15: Projected 3D Bounding Boxes with Eroded Segmentation



(a) erosion_factor = 10 (b) erosion_factor = 20

Figure 5.16: BEV Erosion

5.3.4 Modification 3: Apply 3D Point Filtering

The next modification, after including eroded segmentation masks, involves filtering the 3D points collected from the fusion process. In total 4 different filtering methods were tested to get the best results. This step is crucial as it determines how well the 3D bounding boxes are created for each detected object. The starting point for every tested filtering method is the set of 3D points obtained after the fusion using the eroded segmentation masks as illustrated in Figure 5.14, i.e. with $erosion_factor = 20$. For every filtering method, a parameter can be specified that influences the aggressiveness of the filtering. For this comparison, multiple values were tested and the best one for this specific example image and point cloud pair was selected.

Distance-Based Filtering

This filtering method uses a statistical method called the *Interquartile Range* (IQR) to filter outliers from a set of 3D points. First, the Euclidean distances of every point to the LiDAR scanner in the xy-plane are computed. The z-coordinate is disregarded as usually, outlier points are in the background and thus the distance is the most important factor. Then, the first quartile (Q1) and the third quartile (Q3) are computed. Q1 is the value below which 25% of the data falls, indicating the distance value at the lower quarter of the data set. Conversely, Q3 is the value below which 75% of the data falls, representing the distance value at the upper quarter of the data set. The IQR is then computed as the range between Q3 and Q1, representing the middle 50% of the data. Next, the lower and upper bounds for valid distances are calculated, as shown in Equation 5.1 using the IQR, Q1, Q3 and the filter factor passed as a parameter to the function.

$$\begin{aligned} lower_bound &= Q1 - filter_factor * IQR \\ upper_bound &= Q3 + filter_factor * IQR \end{aligned} \tag{5.1}$$

A lower filter factor leads to more aggressive filtering, keeping points closer to the median, and a higher filter factor allows more points to be kept.

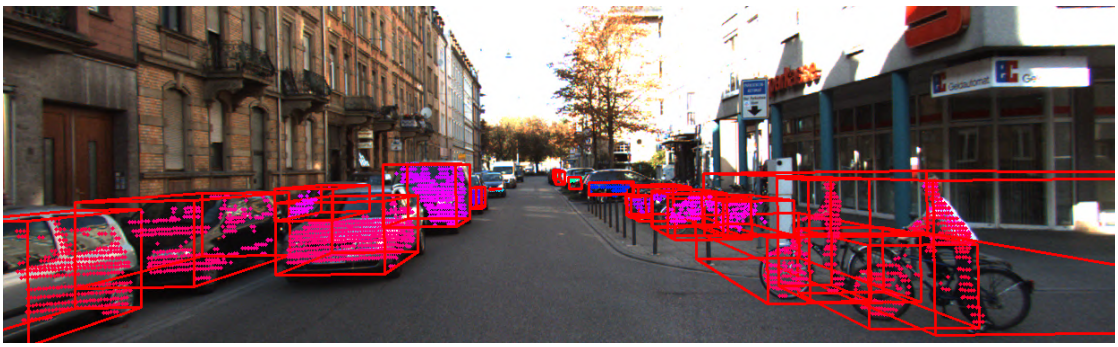


Figure 5.17: Projected 3D Bounding Boxes with Distance-Based Filtering

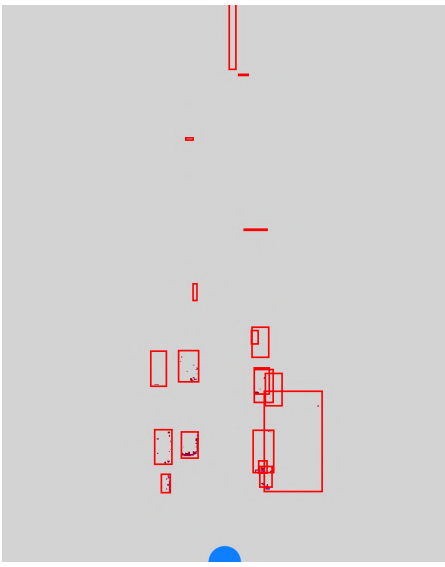


Figure 5.18: BEV Distance Filtering

Using these upper and lower bounds, a boolean mask is created, which is applied to the original set of 3D points to extract inliers which are then returned by the function. Figure 5.17 depicts the 3D bounding boxes as well as 3D points that were filtered using this function. One can see that the filtering works pretty well for cars. However, pedestrians and bicycles are not accurately filtered resulting in bounding boxes that are a lot too big compared to the detected objects. This indicates that the filtering method does not perform well in filtering out background points that are far away, like for example points that went through the spokes of the bicycles, even when reducing the filter factor to increase the aggressiveness of the filtering. Figure 5.18 shows the BEV representation of the points and bounding boxes allowing for a better comparison.

Standard Deviation Filtering

The idea of this method is to filter 3D points based on their distances from a mean point. First, the mean point of the set of 3D points is calculated and the distance from every point to this mean is computed and stored in an array. Then, the standard deviation of these distances, which measures the spread of the points around the mean, is calculated. A threshold factor, specified as a parameter of the function, determines how many standard deviations away from the mean a point can be before it is considered an outlier.

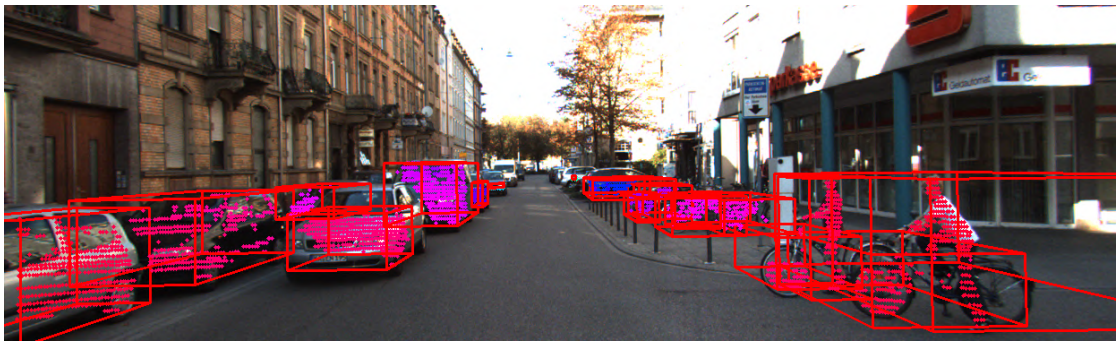


Figure 5.19: Projected 3D Bounding Boxes with Standard Deviation Filtering

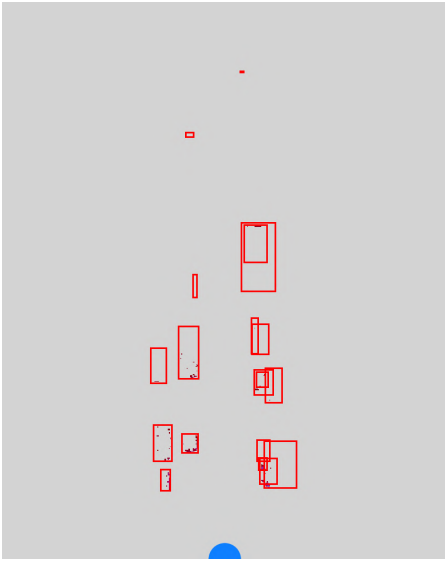


Figure 5.20: BEV Standard Deviation Filtering

This is done by multiplying the threshold factor with the standard deviation. Finally, points that have distances less than this scaled standard deviation are selected as inliers and returned. Figure 5.19 shows that this filtering method is better suited for pedestrians and bicycles as these bounding boxes are a lot better when compared to the distance-based filtering method. However, the filtering is sometimes too aggressive for the *Car* object class as one can see by the smaller bounding box for the first car on the left side of the road. Figure 5.20 gives a better representation of the bounding boxes for comparison. When increasing the threshold factor to reduce the aggressiveness of the filtering, the bounding boxes of the cars improve, but the ones of the cyclists become worse similar to the ones for the distance-based filtering method.

K-Means Filtering

This filtering method uses the *KMeans* clustering algorithm, explained in Section 3.4.4. The idea was to cluster all the 3D points of a detected object and choose the largest cluster. The z-coordinate of the points was not very relevant for the clustering since most outliers were background points that were simply far away from the detected object. Thus, the function first computes the Euclidean distances of each point to the LiDAR sensor and fits the KMeans algorithm, from the *scikit-learn* library [37], with these depth values.

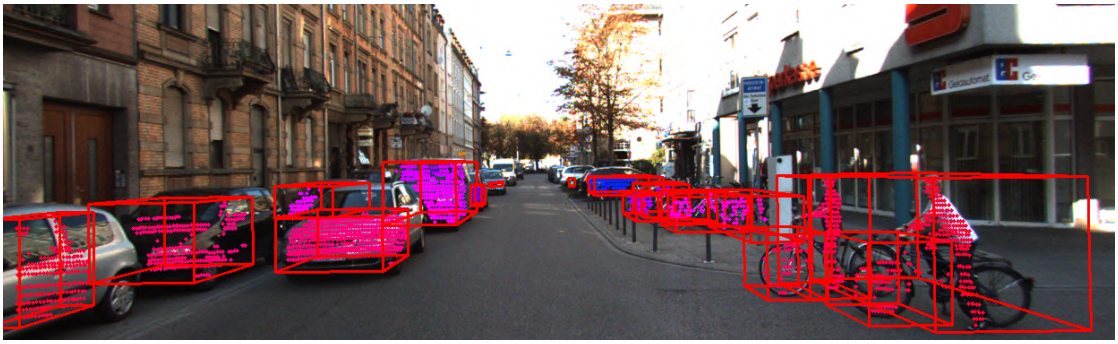


Figure 5.21: Projected 3D Bounding Boxes with KMeans Filtering

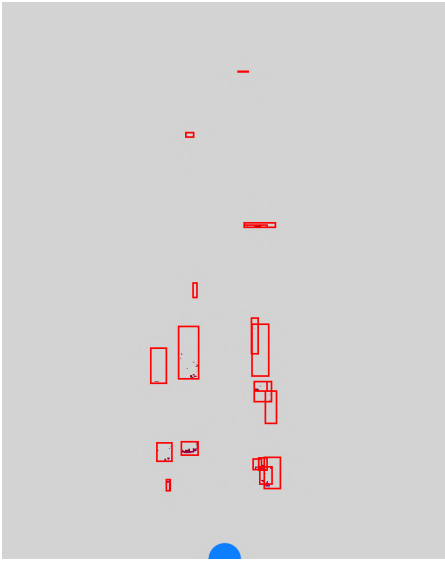


Figure 5.22: BEV KMeans Filtering

Then, the number of points in each cluster is counted and the largest cluster is selected. Finally, the points belonging to this cluster are selected and returned. Figure 5.21 shows the filtering results when using the KMeans approach. As one can see, the bounding boxes are generally better than the previous filtering methods. Figure 5.22 shows that the bounding boxes are similar to the ones of the standard deviation filtering method with slightly worse bounding boxes for the *Car* class. However, the function needs the number of clusters as input. Since this value is different for various situations, it is very difficult to generalize this filtering method and thus it is not suitable for this application. For this specific example image, the results may be better than for other image and point cloud pairs.

Depth-Based Filtering

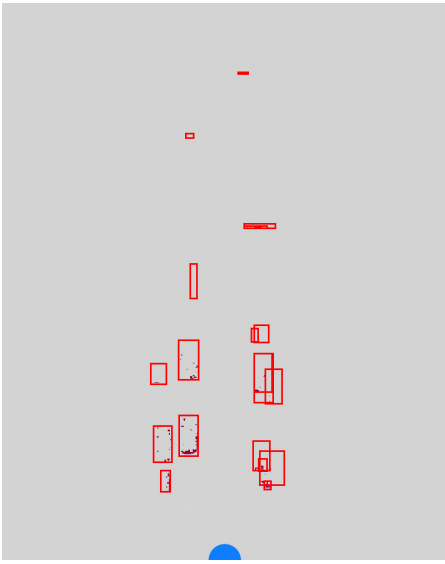


Figure 5.23: BEV Depth Filtering

The idea of this filtering method was to cluster the 3D points based on the density of their depths. This means that again the z-coordinates are not considered and the depths are computed as the Euclidean distances in the xy-plane. Section 4.3.1 provides a detailed explanation of the depth-based filtering function. Figure 5.24 shows the bounding boxes that encapsulate the 3D points filtered by this method using a depth factor of 30. One can see that for the *Car* object class the bounding boxes are more accurate than for the other filtering methods. For the *Pedestrian* class the boxes are slightly better than with the KMeans and Standard Deviation methods. However, for the *Bicycle* class, the boxes are sometimes a little worse than for the other two mentioned filtering methods. This is again better illustrated in Figure 5.23 with the

BEV representation of the boxes. Since the bounding boxes created from the points filtered with the depth filtering approach are generally more accurate, I decided to use this filtering method in the proposed model. Further, this method can better adjust to various situations compared to the KMeans approach for example.

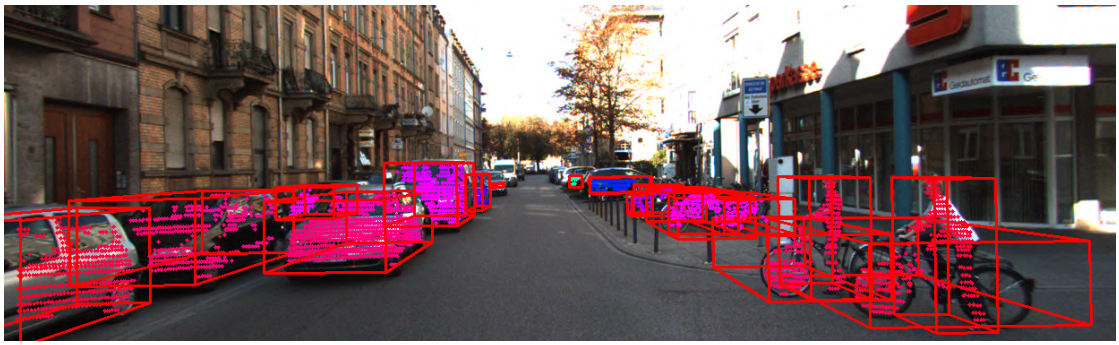


Figure 5.24: Projected 3D Bounding Boxes with Depth-Based Filtering

5.3.5 Modification 4: Implement PCA for 3D Bounding Box Creation

The last evolution step of the model was to add *Principle Component Analysis* (PCA) during the bounding box creation as already explained in Section 4.3.4. The usage of PCA enables the bounding boxes to be oriented and should thus improve the overall accuracy of the 3D bounding boxes. Figure 5.25 demonstrates how the bounding boxes

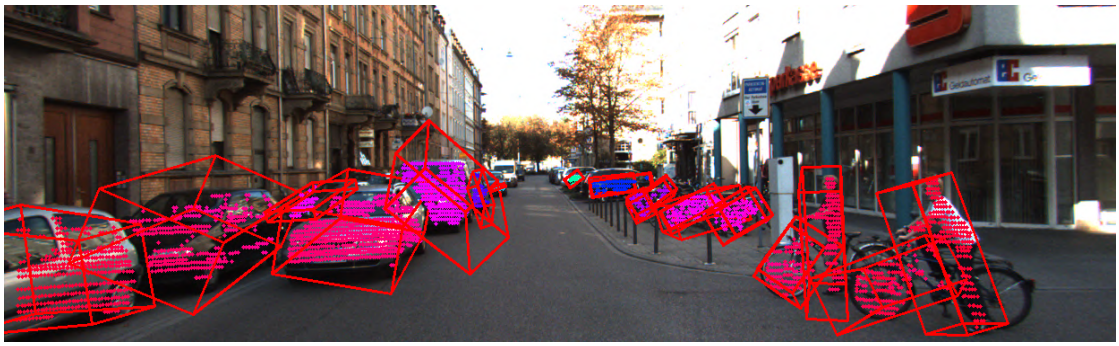


Figure 5.25: Projected 3D Bounding Boxes with PCA

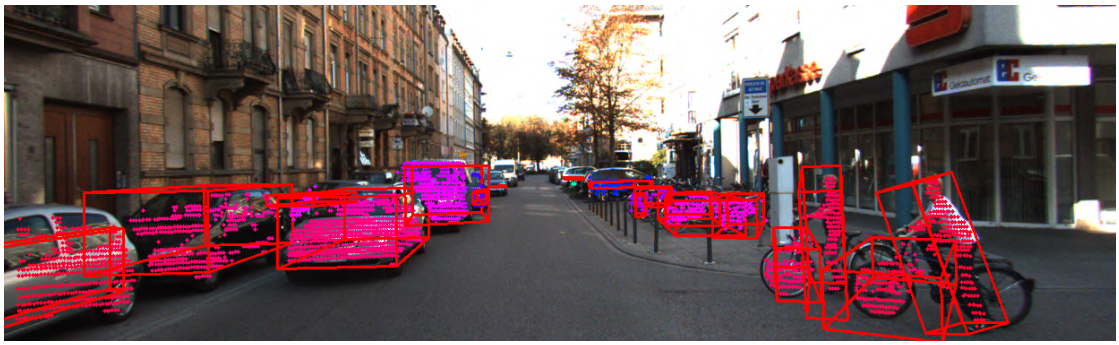
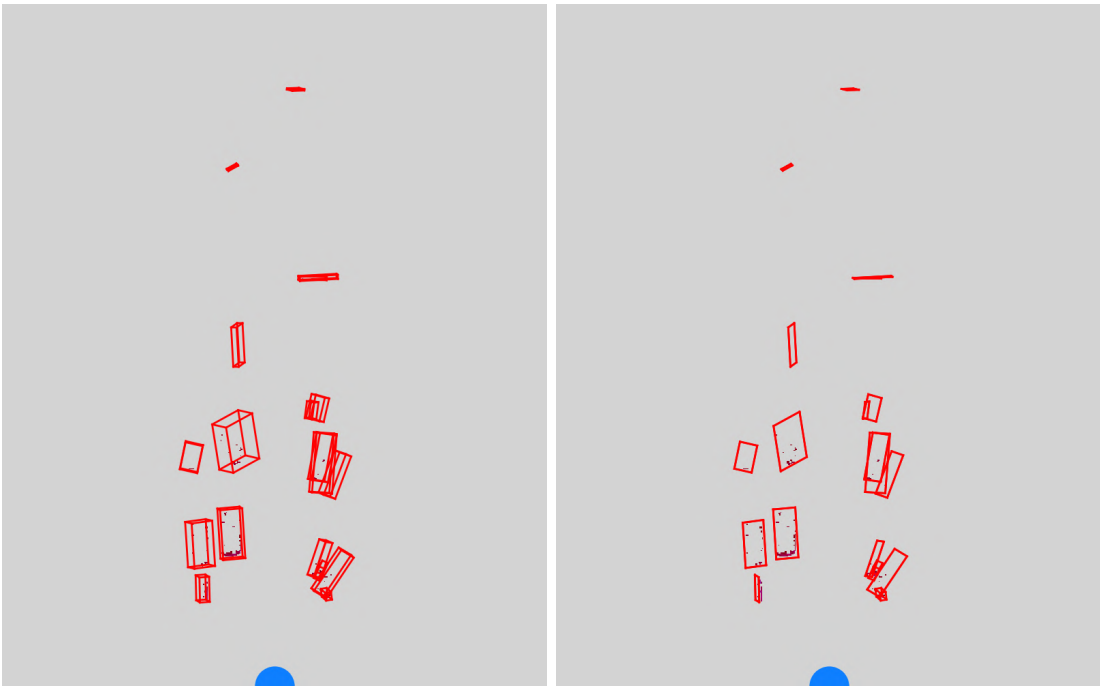


Figure 5.26: Projected 3D Bounding Boxes with PCA (no z-axis rotation)

would be created when simply using PCA for the filtered points of each detected object.

As one can see, the bounding boxes are oriented in all 3 axes. However, the z-axis rotation is not necessary as it does not improve the bounding boxes. For this reason, the proposed model uses PCA without z-axis rotation as shown in Figure 5.26. Only for the *Pedestrian* class, the z-axis rotation is still kept as omitting it results in less accurate bounding boxes. For the other classes, the resulting boxes are more accurate and only



(a) BEV PCA with z-axis rotation (b) BEV PCA without z-axis rotation

Figure 5.27: BEV PCA with and without z-axis rotation

take the xy-plane into account for their orientation. In this example image, the power of PCA does not show very well, one can refer to Figure 4.9 of Section 4.3.4 to get a better visualization of the impact of using PCA for the bounding box creation. Further, Figure 5.27 shows the difference when using PCA with rotation in all 3 axes (a) and only in the xy axes, i.e. without z-axis rotation, (b).

5.3.6 Segmentation Refiner

Even after the erosion and filtering, there still remain outliers in some cases, e.g. when the LiDAR laser goes through car windows or is reflected from surfaces. To prevent this, the idea was to use a *Segment Anything* (SAM) model that can give a better segmentation mask of the detected objects and maybe also ignore specific parts such as car windows when creating the mask. However, as the standard SAM model from Ultralytics [38] requires heavy computational and time resources, I decided to use their faster *FastSAM* model [39] which is based on the YOLOv8-seg model that the proposed model is based on. As the FastSAM model would segment anything when the whole image is input, the normal YOLOv8 model first runs to get the bounding boxes of the detected objects. Then, the FastSAM model is used to get a segmentation of the object inside each box. FastSAM provides two model sizes, however, the smaller size (FastSAM-s) was less accurate than the normal YOLOv8-seg model and thus it would not make sense to use it.

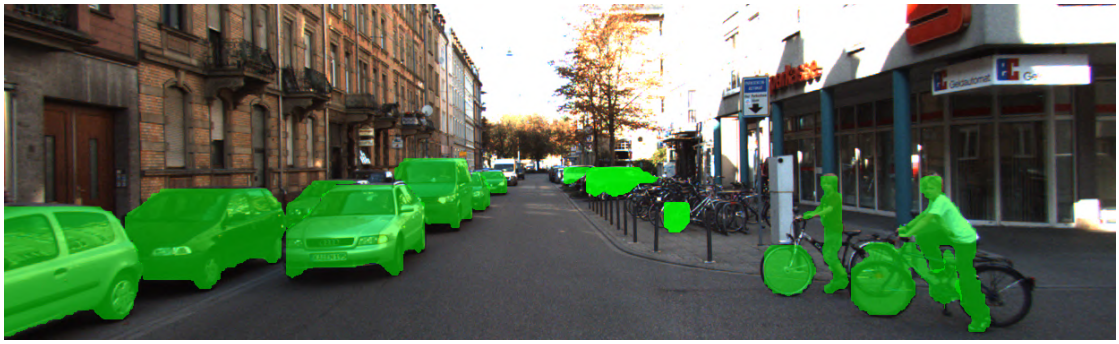


Figure 5.28: FastSAM Segmentation Masks

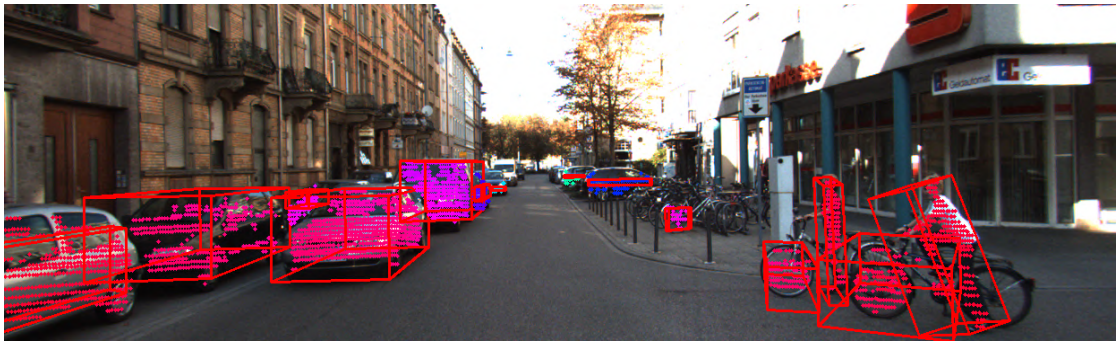


Figure 5.29: Projected 3D Bounding Boxes when using FastSAM

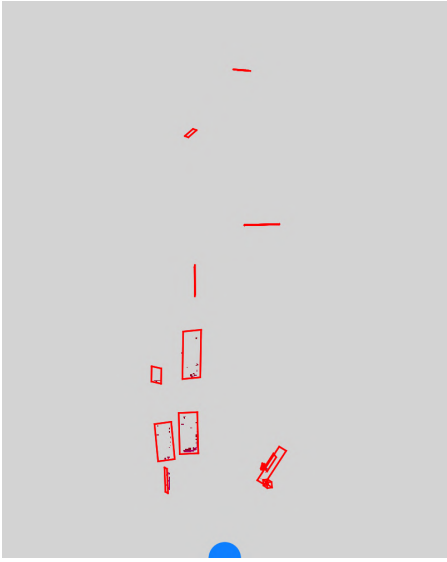


Figure 5.30: BEV FastSAM

The segmentation masks from the bigger FastSAM-x model are shown in Figure 5.28. One can see that the masks are a bit better than those of the YOLOv8-seg model shown in Figure 5.9. The next steps of the model are the same as the last model evolution, i.e. the masks are eroded, 3D points are fused, filtered with the depth-based method and the 3D bounding boxes are created using PCA. The final result for this example image when using this segmentation refinement is shown in Figure 5.29. One can see that the results do only very slightly improve for this example, the bounding boxes for the *Pedestrian* and *Car* classes stay mostly the same and the bicycles in the background are not entirely detected, which is irrelevant for this image. This can again be better observed in Figure 5.30 by comparing the bounding boxes to the previous model modification.

5.3.7 Tracking

The YOLOv8 model also provides the possibility of adding tracking while detecting objects as explained in Section 4.3.5. The detection performance of the model is impacted by the addition of tracking as illustrated by Figure 5.31.

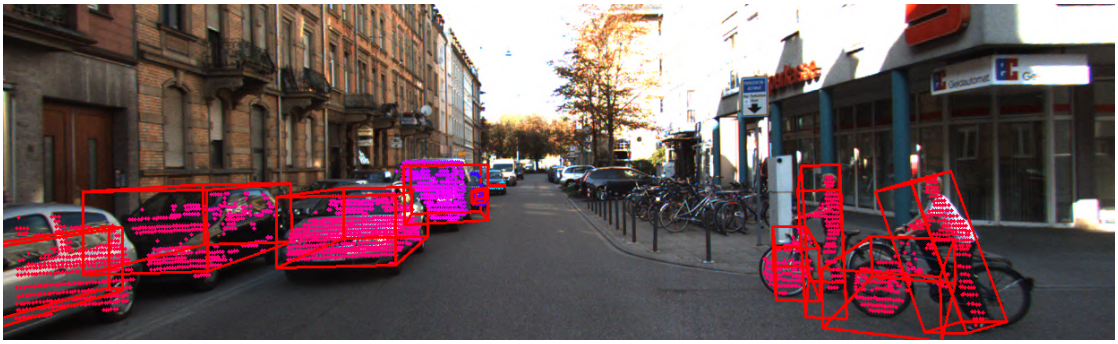


Figure 5.31: Projected 3D Bounding Boxes with Tracking

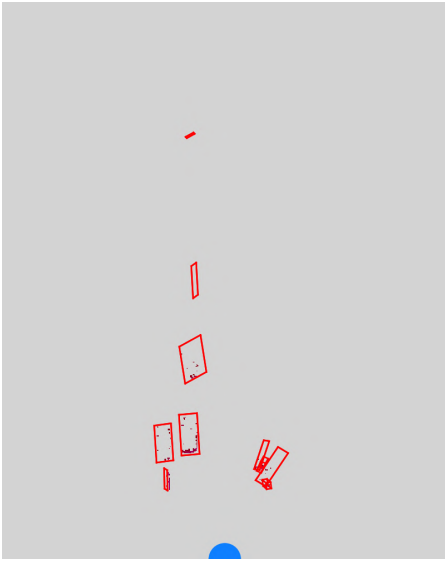


Figure 5.32: BEV Tracking

The figures were produced by simply adding tracking to the last model evolution, thus the model also includes erosion, depth-based filtering and PCA. One can observe that some objects are not detected by this model, in the example image the bicycles and cars in the background on the right side are not captured. Figure 5.32 further shows that the remaining bounding boxes of the detected objects correspond to the same that the model without tracking creates. Overall one can thus say that the addition of tracking slightly decreases the model's detection capabilities. This could be due to the way that the tracking algorithm manages object identities which could lead to objects being filtered out as stated in a discussion on the official Ultralytics GitHub page. However, in most situations, the model should still be able to accurately detect important objects, such as vehicles or pedestrians on its trajectory.

5.3.8 Performance Comparison

Every model evolution step was run on the validation set derived from the KITTI dataset, as explained in Section 5.1.1, to compare the different evolution steps of the proposed model. All model evolution steps were run with the runtime parameters $erosion_factor = 20$ and $depth_factor = 20$, when available in the respective model evolution, to provide a valid comparison. The validation set contains 3740 matching image and point cloud pairs with the 3 most relevant object classes *Car*, *Pedstrian* and *Cyclist* as already explained before. The intersection over union (IoU) for the *Car* and *Pedestrian* classes were used for the comparison as the proposed model cannot classify the *Cyclist* class accurately, as explained in Section 5.2.2.

Table 5.2 provides the results of each model evolution step presented in the previous subsections. It contains the best and average IoU scores for both *Car* and *Pedestrian* classes. The best IoU score represents the most accurate bounding box for each frame, whereas the average IoU score represents all bounding boxes in one frame. Both values are averaged over all frames in the validation set. Additionally, Table 5.2 also contains the *Inference Time* to process one frame and the *GPU Memory Consumption* used dur-

Model Evolution	**IoU (%)**				**Inference Time (s)**	**GPU Memory Consumption (MiB)**
	Car		**Pedestrian**			
	Best	**Avg.**	**Best**	**Avg.**		
Initial Model	11.79	5.90	1.65	1.08	**0.2237**	1778
Segmentation	14.23	6.80	2.94	1.81	0.2338	**1650**
Erosion	18.06	8.64	3.57	2.24	0.2986	1720
Distance-Based Filtering	29.29	13.86	16.97	13.27	0.2347	1728
Std. Deviation Filtering	35.57	17.64	15.73	11.68	0.2336	1888
K-Means Filtering	31.08	14.52	19.86	15.34	0.2838	1754
Depth-Based Filtering	**40.40**	**20.28**	25.36	20.37	0.2416	1722
PCA	32.66	16.37	**35.50**	**28.13**	0.2425	1772
Tracking	31.99	15.51	32.45	14.47	0.2353	1858
Segmentation Refiner	33.42	17.49	24.92	18.70	0.6420	4440

Table 5.2: Performance comparison of different model evolutions

ing the processing of the testing dataset. The inference time represents an average value as it was measured by taking the total execution time of the respective model step to process the whole testing dataset and dividing it by the number of frames in the set, i.e. 3740. This inference time thus also includes the processing time to compute the best and average IoU scores for each frame and to write those values inside a text file that is later used to extract the results and average them over all validation set frames. During live inference, the model should be faster as it simply returns the 3D bounding box information. Further, the inference time always depends on the number of detected elements in the current frame and on the inference speed of the YOLOv8 model.

As one can see in Table 5.2 as well as in Figure 5.33, the average IoU values for all classes improve as more processing steps are included in the model. The darker-coloured lines on the graph represent the best IoU values, and the lighter colours represent the average IoU values for each of the 3 classes. The IoU scores for the *Cyclist* class, represented in green on the graph, are as already mentioned very bad. This is due to the fact that every bicycle is classified as a cyclist. Thus, most of the bounding boxes for this class are either misplaced or too small.

Generally, the model's performance increases as more processing is added, e.g. erosion, filtering, and PCA. However, one can observe some interesting outcomes, for exam-

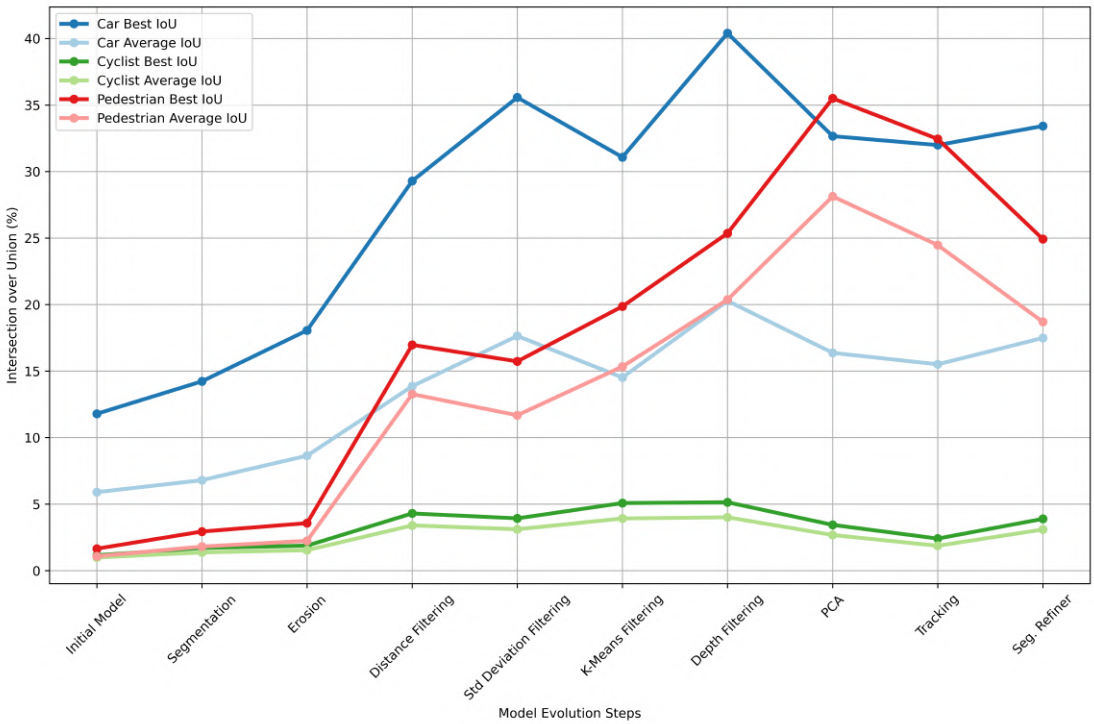


Figure 5.33: Model Evolution Steps - Best and Average IoU

ple, standard deviation filtering is better suited for cars than for pedestrians. Contrarily, the KMeans filtering method performs better for pedestrians than for cars. The reason for this could be that there are usually fewer points for pedestrians and their bounding boxes are smaller than those of cars. Another interesting observation from these results is that bounding boxes that are created without PCA are more accurate for the *Car* class, but implementing PCA results in a higher IoU for the *Pedestrian* class. The addition of tracking to the final model evolution step (i.e. with depth-based filtering and PCA) slightly decreases the IoU scores compared to the model that uses depth-based filtering and PCA without tracking. Additionally, the usage of a segmentation refiner in the final model (without tracking) does not improve the accuracy and actually diminishes the IoU score for pedestrians compared to the model that uses depth-based filtering with and without PCA.

Figure 5.34 provides insights about the GPU memory consumption of the various model evolution steps (in orange) and their average inference times (in blue). As already mentioned, the inference time represents an average value to process one frame. The *GPU Memory Consumption* of each model evolution step was taken directly from the *NVIDIA System Management Interface* of the GPU and represents the highest memory usage during the processing of the validation dataset for the corresponding model.

One can observe that the GPU memory consumption and inference time of the model are somewhat correlated. The baseline model is the only model step that does not use the YOLOv8 segmentation model, but it still consumes more memory than most of the other model steps. However, it still has the lowest average inference time. The slight variations in the inference time and GPU memory consumption can mostly be neglected as the tests were run on a platform that has no strictly controlled environment to ensure

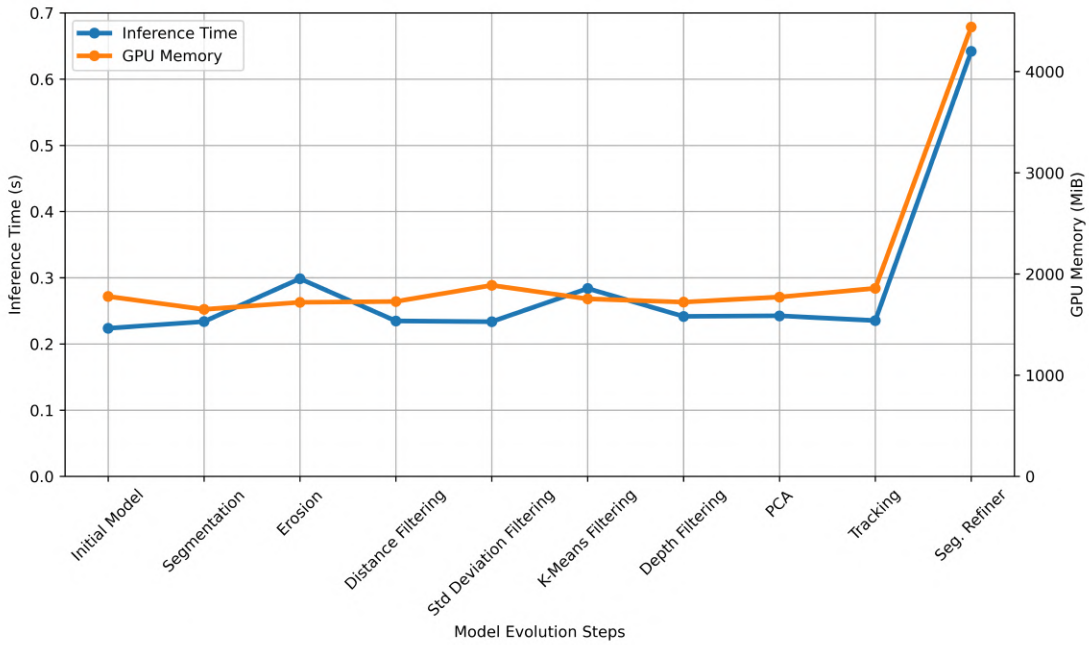


Figure 5.34: Model Evolution Steps - Inference Time and GPU Memory Consumption

the same setting for every model execution. One can further observe that the addition of tracking does not impact the GPU memory consumption or the inference time a lot. Differently, the segmentation refiner that implements Ultralytics' FastSAM model consumes around 2.5 times the amount of GPU memory than the last model step (with depth-based filtering and PCA) which also results in an around 2.6 times slower inference time while not providing any benefit concerning the IoU score.

Overall, this comparison shows that the hypothesis that the model evolution step with the YOLOv8 segmentation model, erosion, depth-based 3D points filtering and PCA for the 3D bounding box creation indeed performs best, however, only for the *Pedestrian* class. Surprisingly, the 3D bounding boxes for the *Car* class are most accurate when constructing them without using PCA. A possibility for the final model could thus be to only use PCA for the *Pedestrian* class and construct the bounding boxes for the *Car* class without PCA.

5.4 Comparison of different YOLO model sizes

The YOLOv8 segmentation model provides 5 different model sizes, i.e. nano (n), small (s), medium (m), large (l) and extra large (x). As the model size grows, its performance improves but the inference time also increases. In order to find the perfect tradeoff between performance and speed for my proposed fusion model, every model size was tested on the latest model evolution (i.e. using depth-based filtering and PCA). Table 5.3 shows the results of the proposed model when using the different YOLO-seg model sizes. The Intersection over Union (IoU) scores of the model are again divided into the *Car* and *Pedestrian* classes. The inference time to process one frame is calculated by dividing the total execution time of the model on the validation dataset by the number of frames in the dataset.

Model Size	IoU (%)				Inference Time (s)	GPU Memory Consumption (MiB)
	Car		Pedestrian			
	Best	Avg.	Best	Avg.		
Nano (yolov8n-seg)	32.30	15.90	32.74	25.21	**0.1567**	**754**
Small (yolov8s-seg)	**32.67**	16.34	34.44	27.14	0.1854	1232
Medium (yolov8m-seg)	32.66	**16.37**	35.50	28.13	0.2472	1772
Large (yolov8l-seg)	32.65	16.30	36.08	**29.00**	0.3192	2294
Extra Large (yolov8x-seg)	32.66	16.34	**36.16**	28.85	0.4047	3578

Table 5.3: Performance comparison of different model sizes for final model evolution

As illustrated in Figure 5.35, the IoU values for the *Pedestrian* and *Cyclist* classes improve as the model size gets bigger. However, the IoU for the *Car* class stays constant after the *small* model size. The proposed model with the biggest YOLOv8 segmentation model size (i.e. extra large) does not provide any improvement in terms of 3D bounding box accuracy for all of the 3 classes.

The *Inference Time*, depicted in blue in Figure 5.36, increases nearly linearly for the last 4 model sizes. The YOLOv8 nano model size is over 2.5 times faster than the extra large model size and can thus achieve around 6.4 frames per second (FPS). The biggest model size only achieves around 2.5 FPS, however, as already said before the inference times also include the computation of the IoU values as well as the creation of the text files to store the detection results. Thus, when only outputting the 3D bounding box data, the models should be faster.

The *GPU Memory Consumption* of the different model sizes follows a similar trend as the inference time, as depicted in Figure 5.36 as the orange line. The memory consumption increases nearly linearly for the first 4 model sizes and massively increases for the largest model size. The smallest model size (nano) consumes over 4.7 times less GPU memory than the largest model size (extra large), which is nearly double the amount as for the inference time.

The results illustrated in Table 5.3, Figure 5.35 and Figure 5.36 support my assumption that the *medium* model size is best suited for my proposed model as it provides good IoU scores for both classes, compared to the other model sizes, while still being fast and resource-efficient enough to provide live inference in an autonomous car running *RoboCar* [12]. However, the *small* model size is even faster (0.06 seconds) and consumes less GPU memory (540MiB) while only having around 1% less IoU score for the *Pedestrian* class and the same IoU for the *Car* class. Thus, depending on the mode of application, the model could also use the second smallest YOLOv8 segmentation model.

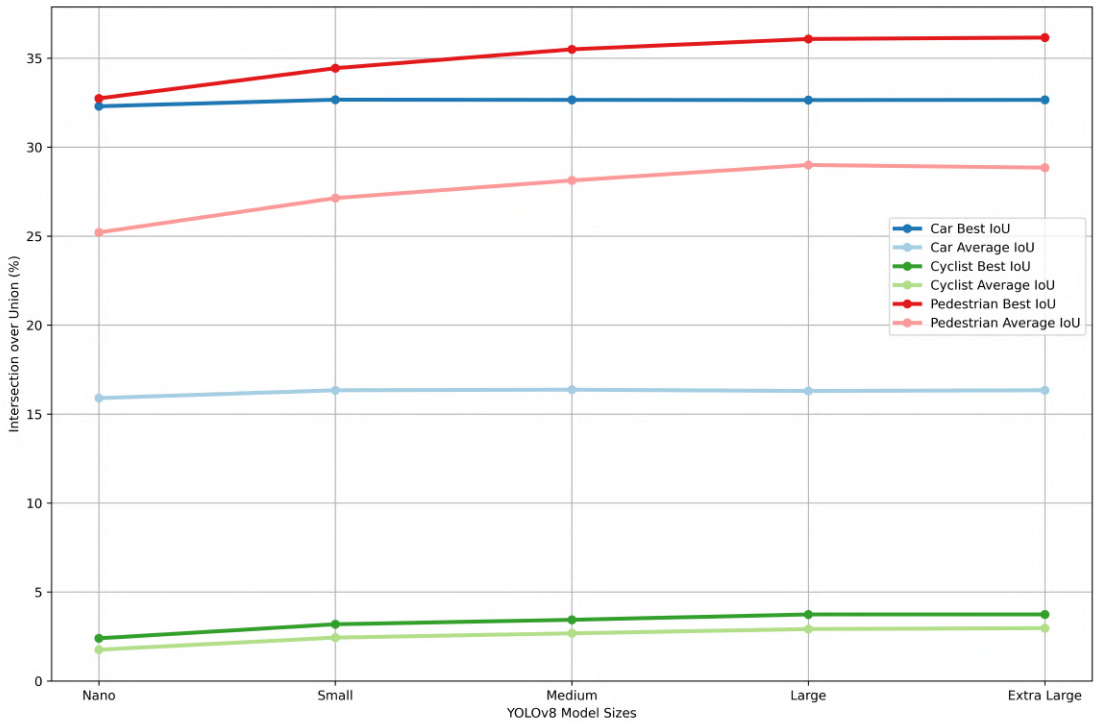


Figure 5.35: Model Sizes - Best and Average IoU

5.5 Results

The proposed model corresponds to a modified version of the last evolution step (Modification 4) of the model evolution in Section 5.3. From the results of Section 5.3.8, one can deduce that the usage of PCA for the 3D bounding box creation is only beneficial for the *Pedestrian* class. Thus this final proposed model does not implement PCA for the bounding box creation of the *Car* class. It also utilizes the runtime parameters derived from the model configuration step of Section 5.2 and thus has the following characteristics:

- Uses the **YOLOv8 Segmentation Model** (medium size) to generate segmentation masks of detected objects
- Applies **Erosion** to the segmentation masks using erosion_factor = 25
- Employs the depth-based **3D Point Filtering** method using depth_factor = 20
- Implements **PCA** for 3D bounding box creation only for the *Pedestrian* class

5.5.1 KITTI Dataset

Table 5.4 summarizes the Intersection over Union (IoU) scores for the proposed model when run on the testing dataset (split from the KITTI training dataset). The IoU values are divided into the best and average IoU scores for both *Car* and *Pedestrian* classes. The average IoU scores are obviously worse as they represent all the IoU values of every ground truth (GT) and predicted bounding box. The best IoU scores represent the most accurate 3D prediction box for each frame, i.e. the one with the highest IoU value with its matching GT box. One can observe that for the *Car* class, the average IoU score

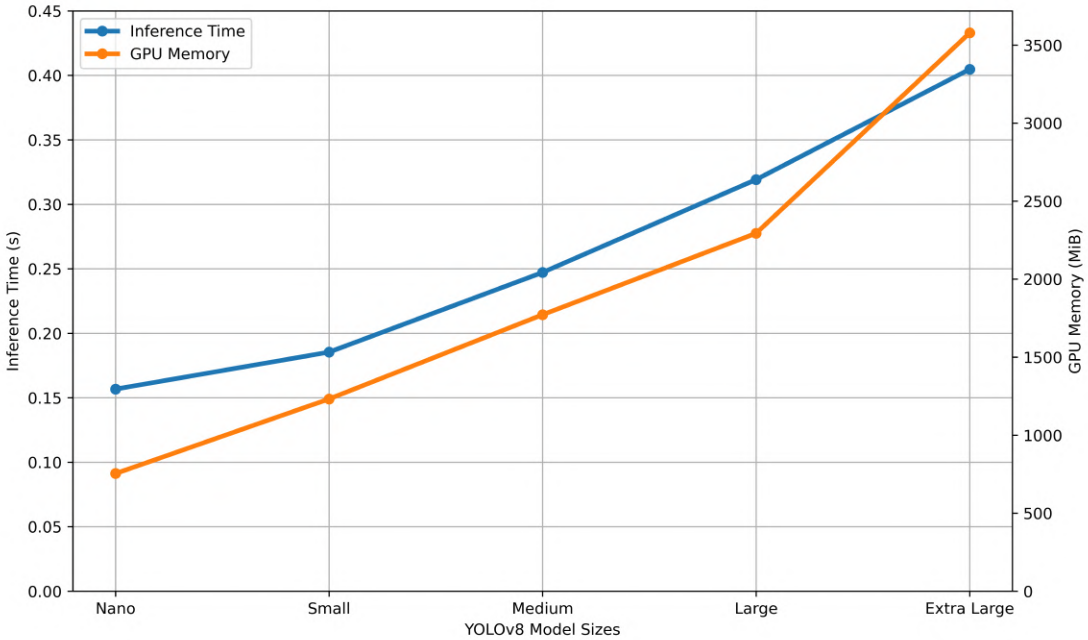


Figure 5.36: Model Sizes - Inference Time and GPU Memory Consumption

is nearly half of its best IoU. This indicates that for some cars the 3D bounding boxes are good, but there are a lot of worse boxes that decrease the average IoU value over all frames of the testing dataset. For the *Pedestrian* class, the average only decreases by roughly one quarter and is also higher than the average IoU for the *Car* class. This shows that the model can better predict *Pedestrians*, probably since their bounding boxes are smaller and usually the captured LiDAR points are more representative of the overall shape and size of the detected object (pedestrian in this case).

IoU *Car* (%)		**IoU *Pedestrian* (%)**		**Inference Time (s)**	**GPU Memory Consumption (MiB)**
Best	**Average**	**Best**	**Average**		
43.51	22.07	35.63	26.99	0.1907	1874

Table 5.4: IoU Scores of Proposed Model on KITTI Dataset

The average *Inference Time* of 0.1907 seconds for one frame was computed by saving the processing time for each frame and then taking the average over all the frames in the testing set. This inference time would allow a live inference of around 5 frames per second (FPS) which could be optimized by using the smaller *yolov8s-seg* model as shown in Figure 5.36. The *GPU Memory Consumption* of 1874 was taken directly from the *NVIDIA System Management Interface* of the GPU and represents the highest memory usage during the processing of the testing dataset by the proposed model. This demonstrates that the proposed model is relatively memory-efficient compared to SOTA models, which often employ more complex and memory-intensive machine-learning techniques.

5.5.2 A2D2 Dataset

The results for this section were obtained by running the exact same model (with slight modifications to utilize the A2D2 calibration files for the fusion) as in the previous Section 5.5.1 on the Audi Autonomous Driving Dataset (A2D2). As depicted in Table 5.5, the proposed model does, as expected, not perform well on the A2D2 dataset. The greater *GPU Memory Consumption* of 2156 (an increase of 15%) as well as the higher *Inference Time* of 0.2547 (an increase of 33%) compared to the proposed model run on the KITTI testing dataset split could be explained by the higher image size (despite the point clouds being smaller) in the A2D2 dataset. On average the images of the A2D2 dataset are over 3 times larger than those of the KITTI dataset due to a higher camera resolution. The whole A2D2 dataset, which contains 12499 frame with annotated 3D bounding boxes, was used for this test. The proposed model could thus process around 4 frames per second on average for this dataset. Figure A.3 visualizes the inference time and GPU memory consumption when the proposed model is run on both the KITTI and A2D2 testing datasets. For the *Car* class, the proposed model achieves less than half

IoU *Car* (%)		**IoU *Pedestrian* (%)**		**Inference Time (s)**	**GPU Memory Consumption (MiB)**
Best	**Average**	**Best**	**Average**		
18.95	10.66	10.31	6.70	0.2547	2156

Table 5.5: IoU Scores of Proposed Model on A2D2 Dataset

of the IoU scores than for the KITTI dataset. The model even performs worse for the *Pedestrian* class where the average IoU score is only a quarter of the average IoU value for the KITTI dataset. Figure A.2 illustrates the best and average IoU score for both the *Car* and *Pedestrian* classes on both testing datasets as only these objects are evaluated in both datasets. The worse results of the model, when tested on the Audi dataset, could be because the LiDAR scanner that was used to capture the point clouds for the A2D2 dataset has only 16 channels, whereas the one used to capture the point clouds for the KITTI dataset has 64 channels as mentioned in Section 5.1.2. Thus, the point cloud of the A2D2 dataset is much sparser than it is the case for the KITTI dataset which results in overall fewer 3D LiDAR points that can be fused for each detected object. As the proposed model was developed and tuned on the KITTI dataset (with a 64-channel LiDAR scanner) it cannot handle the sparser point clouds as well, producing worse 3D bounding boxes.

5.5.3 Examples

This section presents some of the best and worst predictions made by the proposed model based on the IoU scores of the ground truth (GT) and prediction 3D bounding boxes. The purpose of this section is to show that the overall IoU scores on the KITTI testing dataset do not clearly represent the performance of the model in some situations. One may think that based on the scores of Table 5.4, the model performs worse than it actually does. Figure 5.37 shows some of the best predictions for the *Car* class of the proposed model on the testing dataset. One can observe in these images that for each car, multiple sides are shown, i.e. either the front or back and from one side. This enables the model to construct accurate 3D bounding boxes as PCA is discarded for the *Car* class. The average IoU scores of these three images are 73.48%, 75.25% and 81.75% respectively.

Figure 5.37: Examples - Good Car Prediction

Figure 5.38 depicts the result when the model applies PCA for the bounding box creation to the second image of Figure 5.37. One can see that the rotation of the boxes is not correct as the fused points only represent half of the car (back and left side). The usage of PCA decreases the average IoU from 75.25% to 35.68%. With the implemented approach, it is thus not possible to construct an accurate bounding box except if some points that capture the other sides of the car would also be captured.

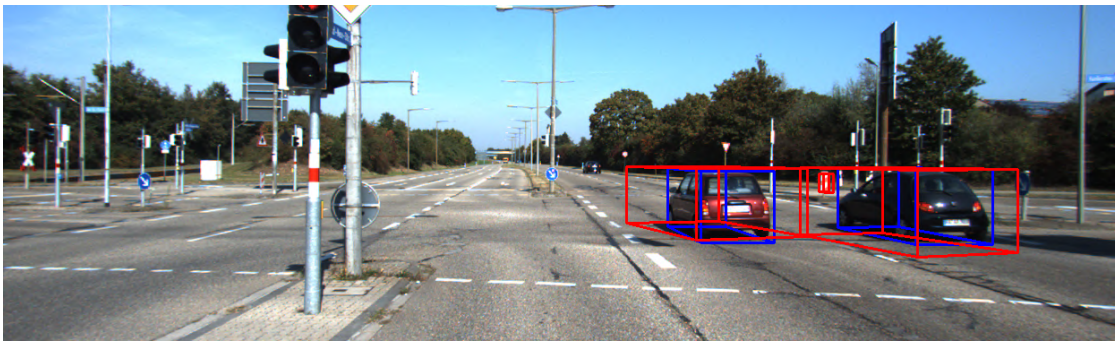


Figure 5.38: Example - Bounding Box Accuracy when using PCA

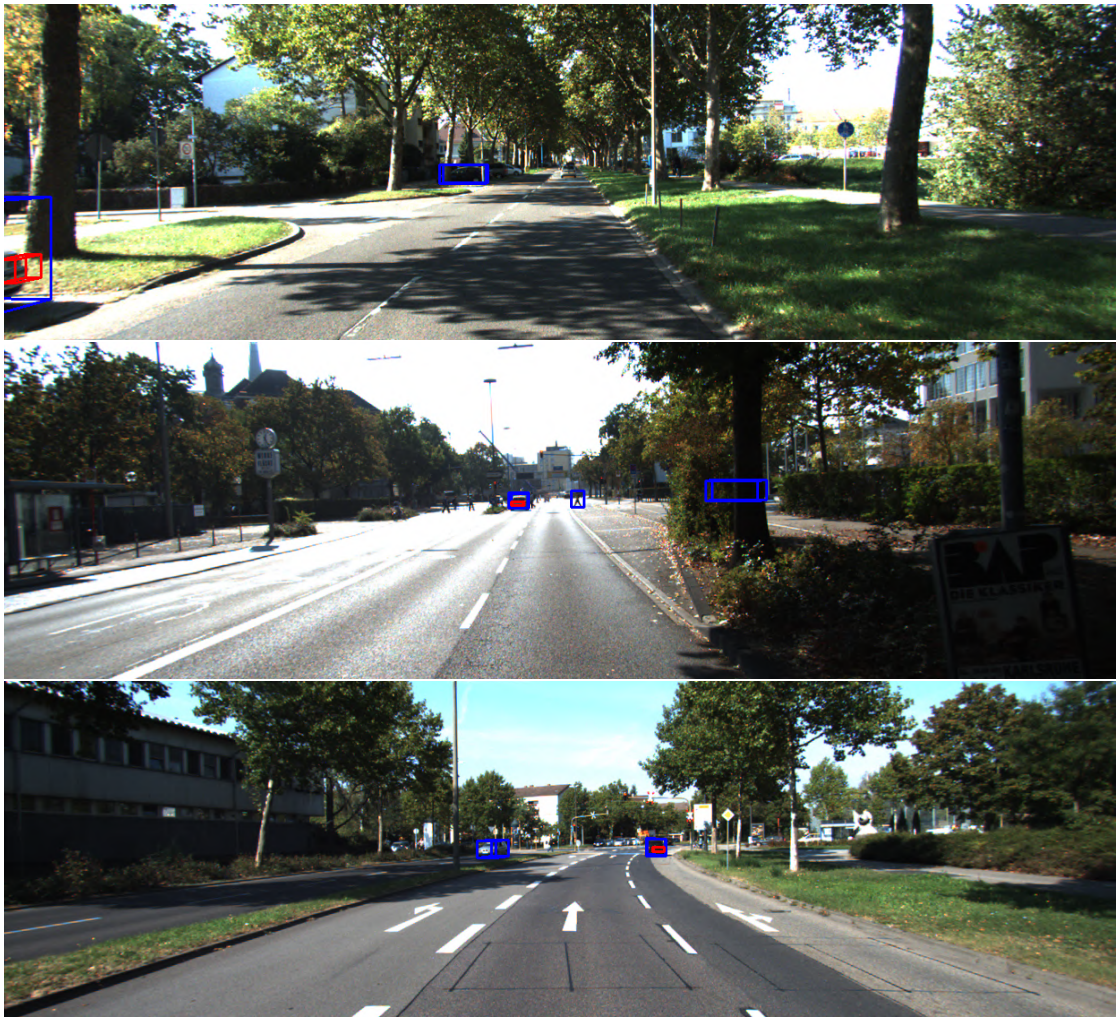


Figure 5.39: Examples - Bad Car Prediction

Some of the worst predictions based on the IoU score are shown in Figure 5.39. In the first image, the car on the left side is barely visible, however, KITTI labelled this car with a complete bounding box, probably using the LiDAR points that are outside of the field of view (FOV) of the camera. This is better shown in Figure A.4 in a 3D representation. The red bounding box is again the one constructed by the proposed model.

One can see that the first blue bounding box captures the whole car, even while it is nearly completely outside the camera's view. This prediction would thus probably be classified with the difficulty *Hard* in the KITTI evaluation script. For the proposed model, it is impossible to construct a bigger bounding box as the 3D LiDAR points outside of the image FOV are filtered out and cannot be fused. Further, the YOLOv8 model can only detect objects that are visible inside the image. For all three images, the cars that are far away are usually not predicted by the proposed model as the number of 3D LiDAR points left after the erosion and filtering processes is too small. Another observation in the second image of Figure 5.39 is that there is a GT bounding box that apparently does not belong to any object, but simply captures a tree. The average IoU values for these three images are 0.07%, 0.17% and 0.23% respectively. These values thus decrease the overall IoU scores for the proposed model on the KITTI testing set.

Figure 5.40: Examples - Good Pedestrian Prediction

Figure 5.40 demonstrates the proposed model's performance to accurately detect the *Pedestrian* class. The first image shows an accurate prediction for both the car and pedestrian even though it is far away. The second image shows that pedestrians that are clearly visible are usually well detected by the model. In the last image of Figure 5.40, the prediction boxes (red) for the person on the left side and in the background on the right side are more accurate than the GT boxes (blue) in my opinion.

Further, the proposed model detects pedestrians that are not labelled by the KITTI dataset, this could however be due to the fact that the KITTI dataset also provides a class for a sitting person, however, I only show the normal *Pedestrian* class. The proposed model however detects a person in the windows of the store on the left side, this thus results in a False Positive (FP), as there is only a picture of a person. The average IoU scores for these 3 images are 71.60%, 72.33% and 75.39% respectively.

Some of the worst predictions for the *Pedestrian* class on the KITTI testing dataset are displayed in Figure 5.41. The first image demonstrates two mistakes made by the proposed model. First, the prediction bounding box is a lot too big for the person on the left side. Second, the model predicts two bushes in the background as pedestrians. The

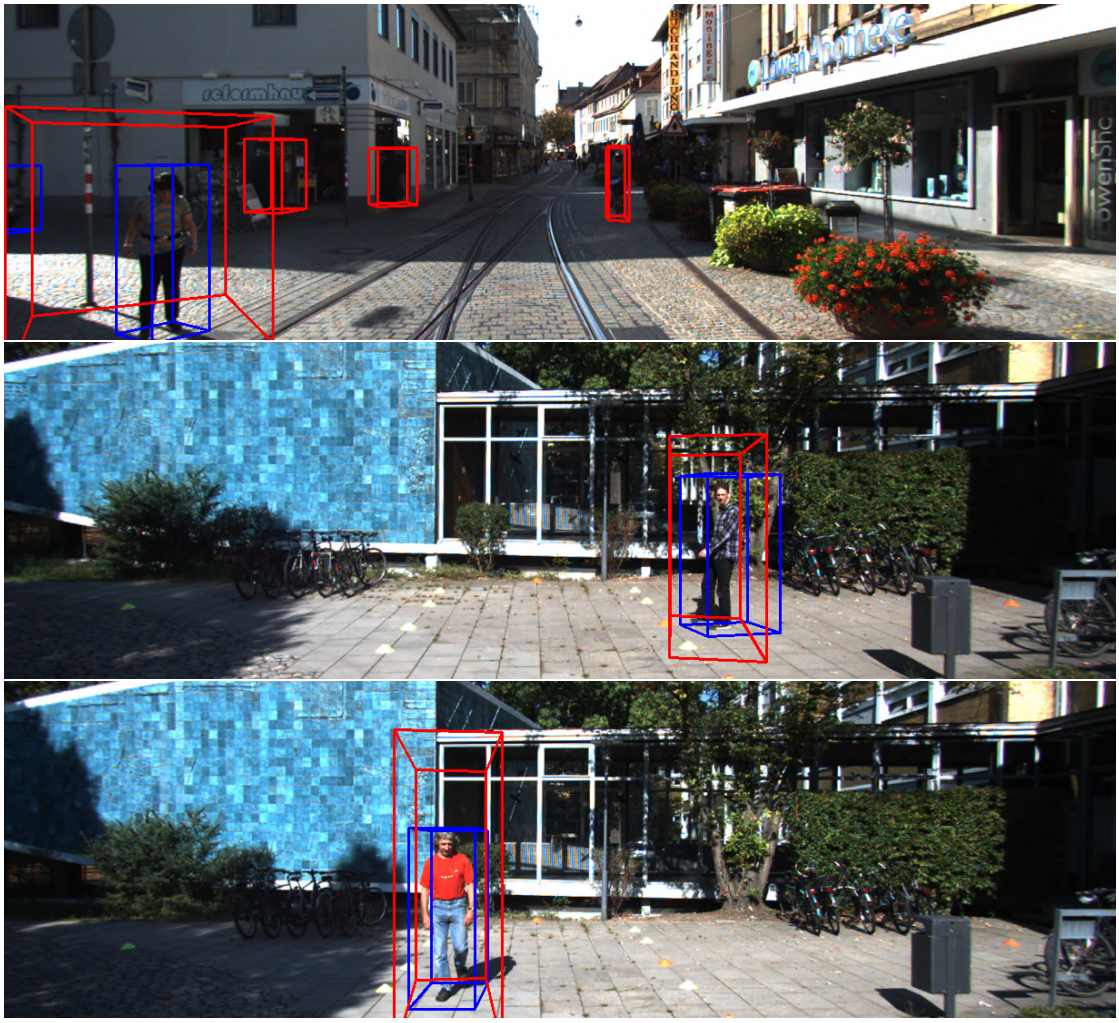


Figure 5.41: Examples - Bad Pedestrian Prediction

prediction bounding box on the right side is actually a cyclist whereas the model only predicts the persons on the op the bicycle. The other two images show again prediction bounding boxes that are a lot larger than the predicted person itself. These bad predictions thus result in average IoU scores of 1.35%, 1.52% and 1.78% respectively. The first image of Figure 5.41 (the rightmost bounding box) shows another problem of the model, i.e. the incapability of predicting cyclists since YOLOv8 only includes the *Person* and *Bicycle* classes separately.

This means that for each cyclist, the proposed model creates two bounding boxes. This is better illustrated in Figure 5.42. The first image shows on the left side the GT box of a cyclist and the corresponding two prediction boxes that the proposed model creates, i.e. one for the bicycle and one for the person riding the bicycle. Similar bounding box creations can be observed in the second image on the right side and in the third image on the left side next to the car. The second image however also shows another GT box that seems to capture no object, but again a tree. The average IoU scores for the *Cyclist* class for these images are as expected bad, i.e. 1.39%, 1.93% for the first two images. However, the third image achieves an IoU score of 30.93% for the cyclist as the proposed model classifies every bicycle detection as a cyclist.

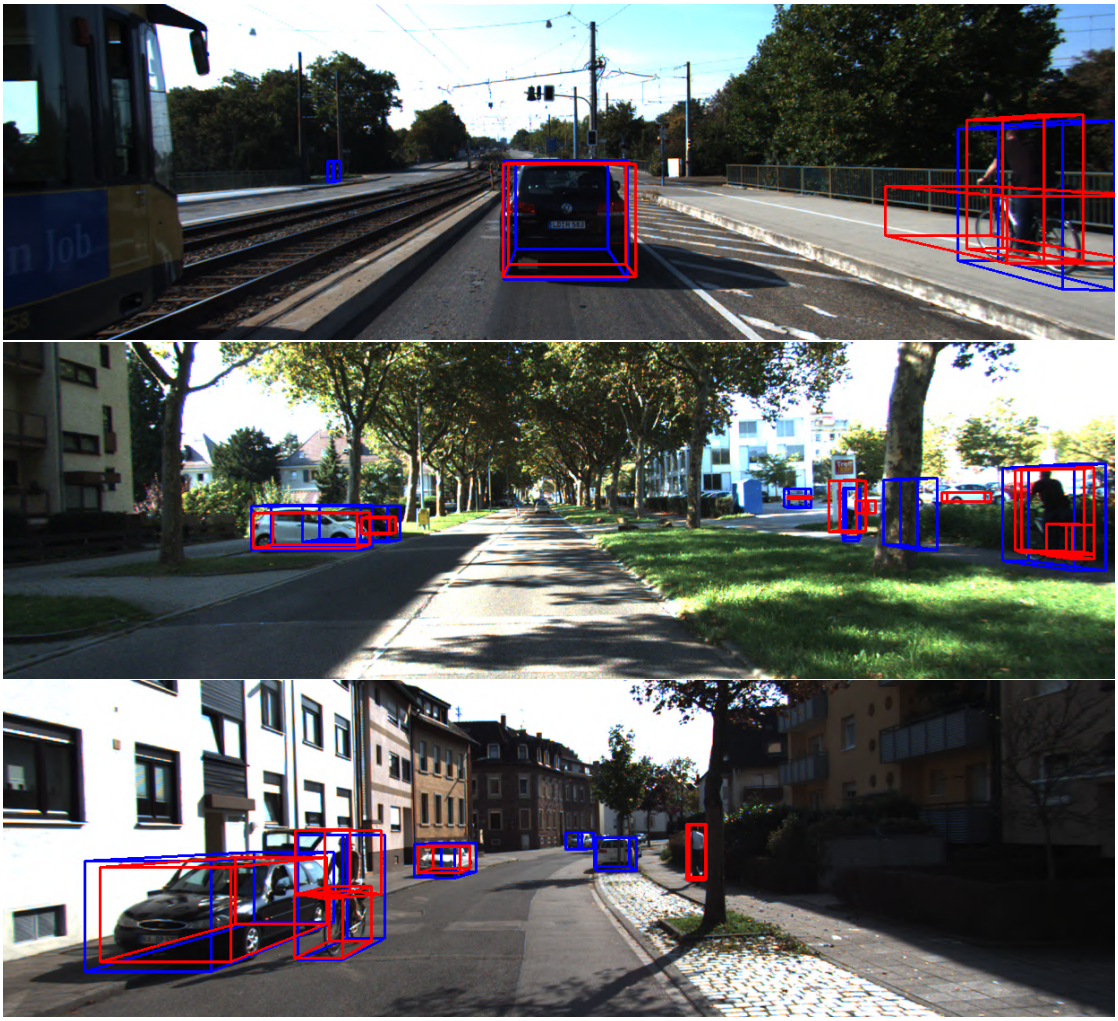


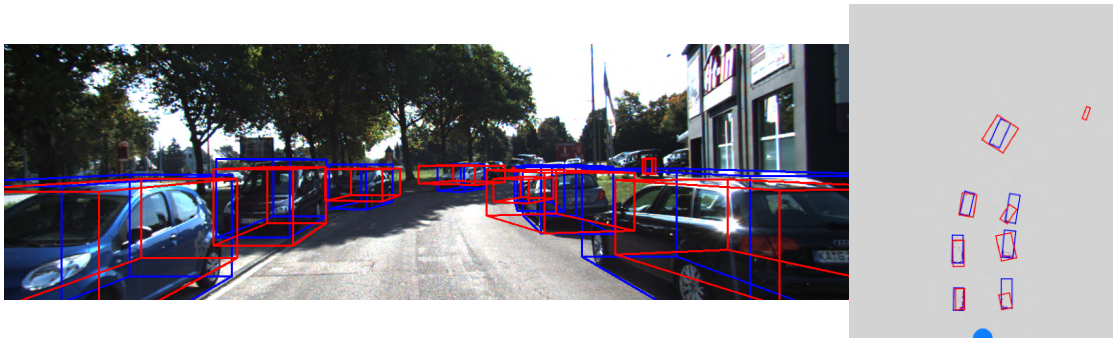
Figure 5.42: Examples - Cyclist Prediction

This section thus explains a bit why the overall IoU score of the proposed model on the KITTI dataset is not very good while some predictions still are very accurate. There are of course situations where the model does not accurately predict objects as shown in this section. The performance of the model depends on the situation and the environment, e.g. if the objects are occluded or fully visible, at which angle the LiDAR and camera sensors capture the objects, etc.

5.5.4 Worth Noting

During the evaluation of the model and gathering of the results and visualizations for this section, I noticed that the model's performance is sometimes diminished by the evaluation method. Since the Intersection-over-Union (IoU) of the predicted and Ground Truth (GT) bounding boxes is computed using their ground centers, dimensions and rotation around the y-axis (yaw), as explained in Section 5.2.1, some details of the prediction bounding boxes are lost.

The proposed model outputs the 8 corners of each predicted 3D bounding box, from which the ground center, dimension and yaw are derived. When no PCA is used for



(a) Bounding box construction using center, dimension and yaw

(b) Bounding box construction using 8 corners returned by model

Figure 5.43: Difference between bounding box construction methods

the creation of the prediction boxes, there is a very minimal difference between the original box derived directly from the 8 corners that the model returns and the box that is constructed using its computed center, dimension and yaw. However, when using PCA, there are in some cases significant differences in quality between both methods of constructing the 3D bounding boxes. One reason for this is that when a bonding box is constructed using its center, dimension and yaw, then the angles between its edges are always 90 degrees, which is not the case when constructing the box using its 8 corners.

Figure 5.43 visualizes the difference between the 3D bounding boxes constructed using the center, dimension and yaw (a) and directly using the 8 bounding box corners (b) returned by the model. The GT bounding boxes are drawn in blue and always constructed by using their center, dimension and yaw. The red boxes are the predicted bounding boxes. As shown in Figure 5.43 (a), the bounding boxes are generally less accurate than those in Figure 5.43 (b). The most notable difference is observed in the last bounding box on the left side of the road. The bounding box constructed using the 8 corners returned by the model is much more accurate compared to the one constructed using the derived center, dimension, and yaw. However, the latter method must be used to compare the prediction boxes with the ground truth (GT) boxes, leading to IoU scores that are worse than they should be.

This highlights that IoU scores can sometimes be misleading when assessing the model's overall 3D detection performance. It is important to note that this example represents only one case out of the entire KITTI training dataset of 7481 labelled images and point clouds. Additionally, the final model does not use PCA for constructing the

bounding boxes for cars, so the issue described is mostly irrelevant to the current evaluation. However, this might explain why the IoU scores for the proposed model—when using PCA to construct 3D bounding boxes for every object class—are worse compared to a model that does not use PCA.

5.6 Comparison with KITTI State-of-the-Art

The comparison of the proposed model with the State-of-the-Art (SOTA) model on the KITTI dataset is unfortunately not possible as the KITTI evaluation script requires an IoU of 70% for the *Car* class and of 50% for the *Pedestrian* and *Cyclist* classes for a detection to be considered as a True Positive (TP). Since the proposed model does not achieve enough predictions with such high IoU scores, it cannot be fairly compared to the KITTI SOTA models, i.e. without lowering the thresholds.

5.7 RoboCar Integration

This thesis aimed to develop a 3D object detection model that can be integrated into the open-source research platform RoboCar [12]. RoboCar was developed at the University of Luxembourg and provides a platform, mostly designed for the 2018 KIA Soul EV, to test and operate Autonomous Driving Solutions (ADS). The platform is built on top of the ROS2 [40] framework which contains a set of libraries and tools for building robot applications. The test vehicle (2018 KIA Soul EV) running RoboCar is equipped with a 64-channel Ouster OS1 LiDAR scanner that has a range of 100m. Further, the vehicle features an Intel RealSense D435 camera that captures RGB images with a resolution of 1920×1080.

RoboCar includes six core modules crucial for various autonomous driving functions. One of these is the *perception* module, which contains all necessary components to process data from perception sensors, such as the LiDAR and camera. The current 3D detection algorithm relies on a clustering algorithm followed by an object-filtering method. The camera is currently only used to detect traffic lights and provide information about their current colour status.

My proposed LiDAR-camera fusion model should fit inside the *perception* module of RoboCar allowing better 3D object detection using both the LiDAR and camera sensors. For this purpose, I designed a ROS2 package, similar to the existing traffic light detector package, to integrate the proposed model into RoboCar. The package includes a ROS2 node that subscribes to the *sensors/camera/compressed* and *sensors/lidar/point_cloud* topics to receive the most recent camera images as well as the point clouds produced by the LiDAR sensor. Further, the node publishes the model's output on the *perception/fused/objects3d* topic such that the motion planner can process this information to update the vehicle's trajectory.

The *ObjectDetector* ROS2 node contains two callback functions that handle incoming data on the subscribed topics, i.e. point clouds and images. The LiDAR callback handles incoming point clouds by converting them from the ROS2 *PointCloud2* message type to a numpy array of 3D coordinates. It then assures that this array is stored thread-safely in the class variable *lidar_data* by using mutual exclusion. The camera callback handles incoming compressed images by converting them into a numpy array and decoding them

into an image using OpenCV [41]. It then copies the latest point cloud data from the class variable *lidar_data* to a local variable, again using mutual exclusion, and passes both the image and point cloud to the function that processed the data. This function passes the camera and LiDAR data to my proposed 3D object detector model which fuses both and returns the detected object from the image frame and corresponding LiDAR point cloud.

For each detected object, the following information are returned and stored in a list that contains the detection information of all objects in the frame:

- **Type**: Integer value that represents the class of the detected object (e.g. 0=person, 2=car)
- **Ground Center**: 3D coordinates of the ground center of the 3D bounding box
- **Direction**: 3D vector that indicates the orientation of the 3D bounding box
- **Dimensions**: Length, width and height of the 3D bounding box
- **Velocity**: 3D vector that represents the speed of the 3D bounding box between two consecutive frames
- **Points**: List of points in 3D coordinates that represent the 8 corners of the 3D bounding box

This list of detection data corresponds to the RoboCar message type *Objects3d* and is published on the *perception/fused/objects3d* topic for further processing.

To test the proposed model inside the RoboCar framework, I created a dummy node that publishes point clouds and corresponding images at 20Hz which is the frequency used by the LiDAR sensor of the testing vehicle. The point clouds are published on the *sensors/lidar/point_cloud* topic and the images on the *sensors/camera/compressed* topic, which correspond to the same topics that the *ObjectDetector* node subscribes to. The dummy node publishes data from the KITTI dataset to simulate the behaviour of the proposed model in RoboCar. The average execution time of the proposed model, when running it in the setup described above, is 0.15192 seconds of which the YOLOv8 model takes on average 0.07373 seconds. However, there are a lot of fluctuations in the YOLOv8 execution time as it can sometimes be as fast as 0.0116 seconds, but also be as slow as 0.2364 seconds. These differences then also impact the total execution time of the proposed model which can in some cases be as low as 0.07913 seconds depending on the YOLOv8 execution time as well as on the number of detected objects in the frame as each object has to be fused with its corresponding LiDAR point. On average the proposed model can thus process around 6.5 frames per second (FPS) and in the best case around 12.6 FPS. Thus, the proposed model does not use the full potential of the LiDAR scanner and camera which can provide point clouds and images at 20Hz.

Due to time requirements and the complexity of calibrating the camera with the LiDAR sensor to compute the transformation matrices from the 3D LiDAR coordinate system to the 2D image plane, we were not able to run the proposed model in the testing vehicles running RoboCar to gather real-world data.

5.8 Future Work

Based on the findings of this project, there are several improvements that could be made to the proposed model to achieve more accurate 3D bounding boxes. One of the biggest challenges during the development of the proposed model was to find a solution that generally works for different object classes and various scenarios. Thus, handling each object class differently would probably give better results. The runtime parameters (erosion and depth factor) of the model could be tuned separately for each object class. This would improve the overall performance of the model as Figure 5.3 of Section 5.2 shows. The best Intersection over Union (IoU) scores for the *Car* and *Pedestrian* classes were achieved by applying a different amount of erosion to the segmentation masks. This again shows that it is difficult to construct one model that can accurately detect all relevant object classes.

A further improvement of the model would be to use different processing techniques (e.g. 3D points filtering, PCA) for different object classes. As it is shown in Figure 5.33 of Section 5.3, the *Car* and *Pedestrian* classes are again individually influenced by different 3D points filtering techniques as well as by the usage of PCA or not to construct the bounding boxes. The final proposed model does already differentiate between the *Pedestrian* and the other classes to use PCA or not for bounding box creation. However, the implementation of further individualized processing techniques for different object classes could not be completed due to time requirements.

Furthermore, the model could include the *Cyclist* class, which KITTI uses among others to evaluate models on their dataset, by combining the bounding boxes of a person and bicycle if they are close to each other. A machine learning approach to combine the bounding boxes could provide good results as the accuracy of the *Cyclist* classification has to be high in order to prevent False Negatives (FN).

Once the performance of the model is good by introducing the before-mentioned improvements, the model could be deployed into the RoboCar testing vehicle to collect data and analyse how the live 3D object detection works in this new environment. As the proposed model heavily relies on LiDAR-to-Camera calibration, real-world testing could be combined with the development and integration of a robust calibration technique to facilitate the deployment of the model to other autonomous driving systems.

Since the proposed model does not consume much GPU memory, which results in a wide range of applications, compared to SOTA models that implement machine learning techniques, the inference time of the proposed model could also be improved to allow even more diversity. The inference time of the model heavily depends on YOLOv8's processing time, thus, finding a method to accelerate it (e.g. by using NVIDIA's *TensorRT* framework) would massively improve the overall inference time of the model.

In the case that the geometric processing techniques, such as erosion, 3D points filtering, and PCA, are at their limits to provide the best possible model performance, one could combine more advanced machine learning techniques that improve the fusion process and bounding box creation. This would include the research of SOTA sensor fusion models and selecting the best-suited and most promising techniques to include in the proposed model.

Considering the YOLOv8 model is capable of detecting 80 different object classes (when trained on the COCO dataset [27]), one could extend the model to predict more objects related to vehicles and autonomous driving. These could then again be used to improve the generalizability of the model by handling the processing of objects in these classes differently as already mentioned. Additionally, the proposed model could be extended to other use cases (not related to autonomous driving) as it only requires a camera, a LiDAR scanner and their calibration. One example could be to fit small robots used in restoration or to deliver packets in crowded urban areas with a camera and LiDAR sensor to detect obstacles.

6 Conclusion

This thesis explored the development of a resource-efficient model for 3D object detection using LiDAR and camera sensors, aimed at improving the performance of autonomous driving systems with limited computational resources. The proposed model achieves notable efficiency in GPU memory usage, differentiating it from state-of-the-art (SOTA) approaches that often demand extensive hardware capabilities. The decision to focus on a lightweight solution was driven by the constraints of deploying such models in real-world vehicles where computational resources are limited such as in the RoboCar test vehicle which uses a NVIDIA GTX 1060 GPU with 6GB of GDDR5 memory.

Throughout this work, several key modifications and techniques were employed to enhance the accuracy and reliability of the model as shown in Section 5.3. The integration of segmentation and 3D point filtering, along with the implementation of Principal Component Analysis (PCA) for bounding box creation, significantly contributed to the model's performance. Despite these advancements, the challenges of accurately detecting various object classes remained obvious, underscoring the complexity of constructing a one-size-fits-all solution.

Evaluation on the KITTI and A2D2 datasets demonstrated the model's capability to operate effectively with a setup of only one LiDAR and camera, highlighting its potential for real-time applications in autonomous driving scenarios. The model's resource efficiency and simple setup expand its potential use cases beyond autonomous driving systems to for example small robots, where computational efficiency is crucial.

In conclusion, while the proposed model offers a promising foundation for efficient 3D object detection, further refinements and exploration of machine learning techniques could enhance its performance and generalization across various environments and object classes. Future work, involving real-world deployment in the RoboCar test vehicle to gather additional insights or refining the model's calibration and detection capabilities, can improve the model's performance such that it could be used in real-world applications.

A Appendix

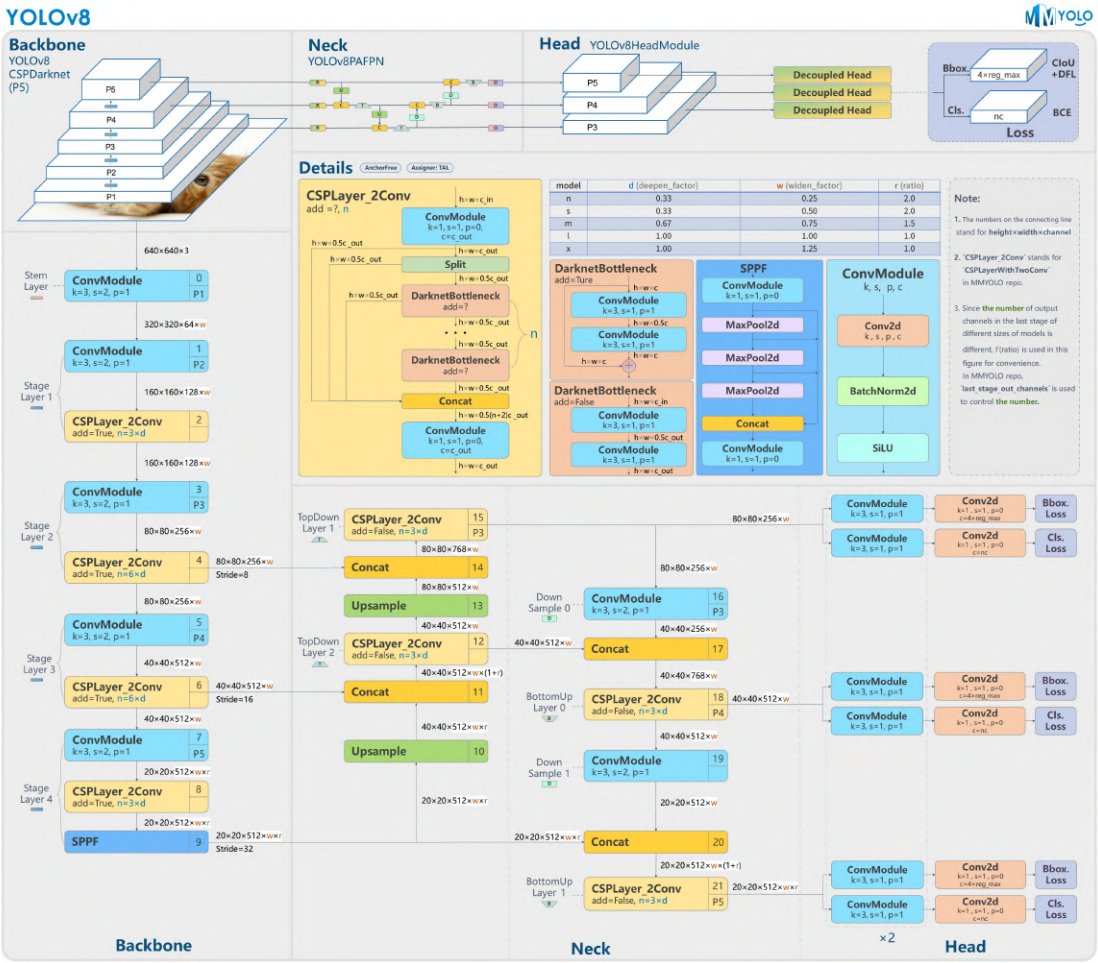


Figure A.1: YOLOv8 Architecture [10]

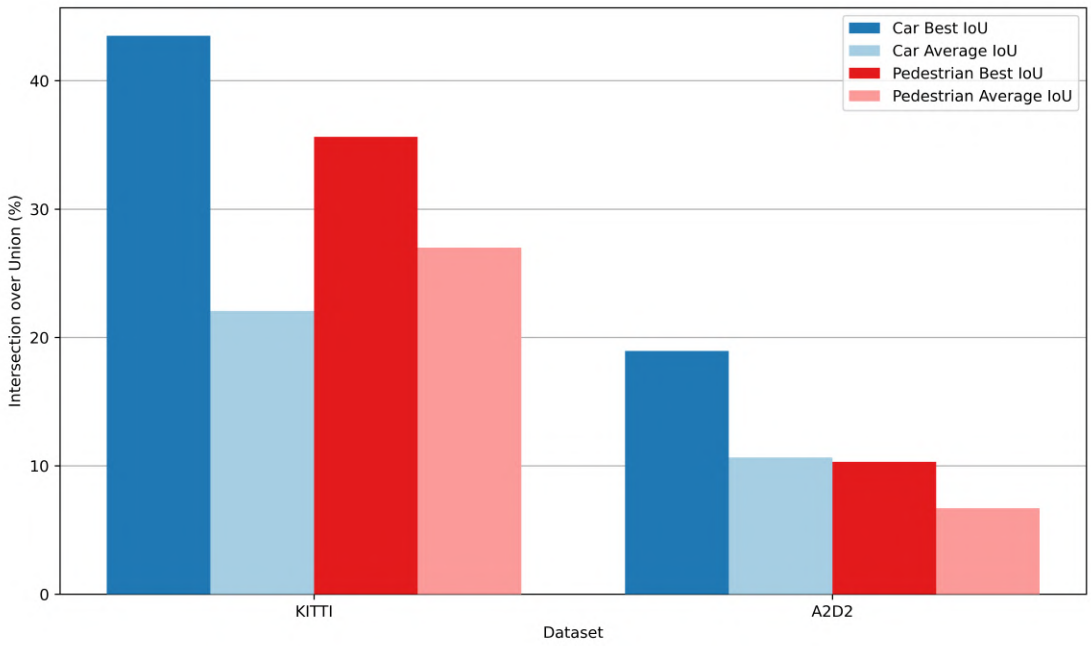


Figure A.2: Comparison of Intersection over Union for KITTI and A2D2 testing datasets

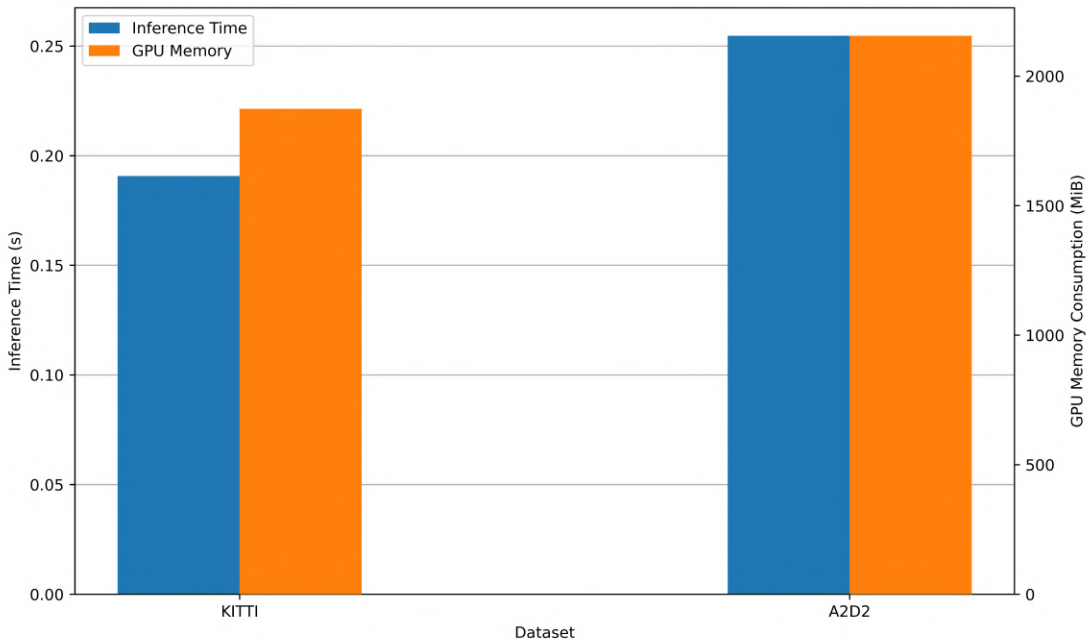


Figure A.3: Comparison of Inference Time & GPU Memory Consumption for KITTI and A2D2 testing datasets

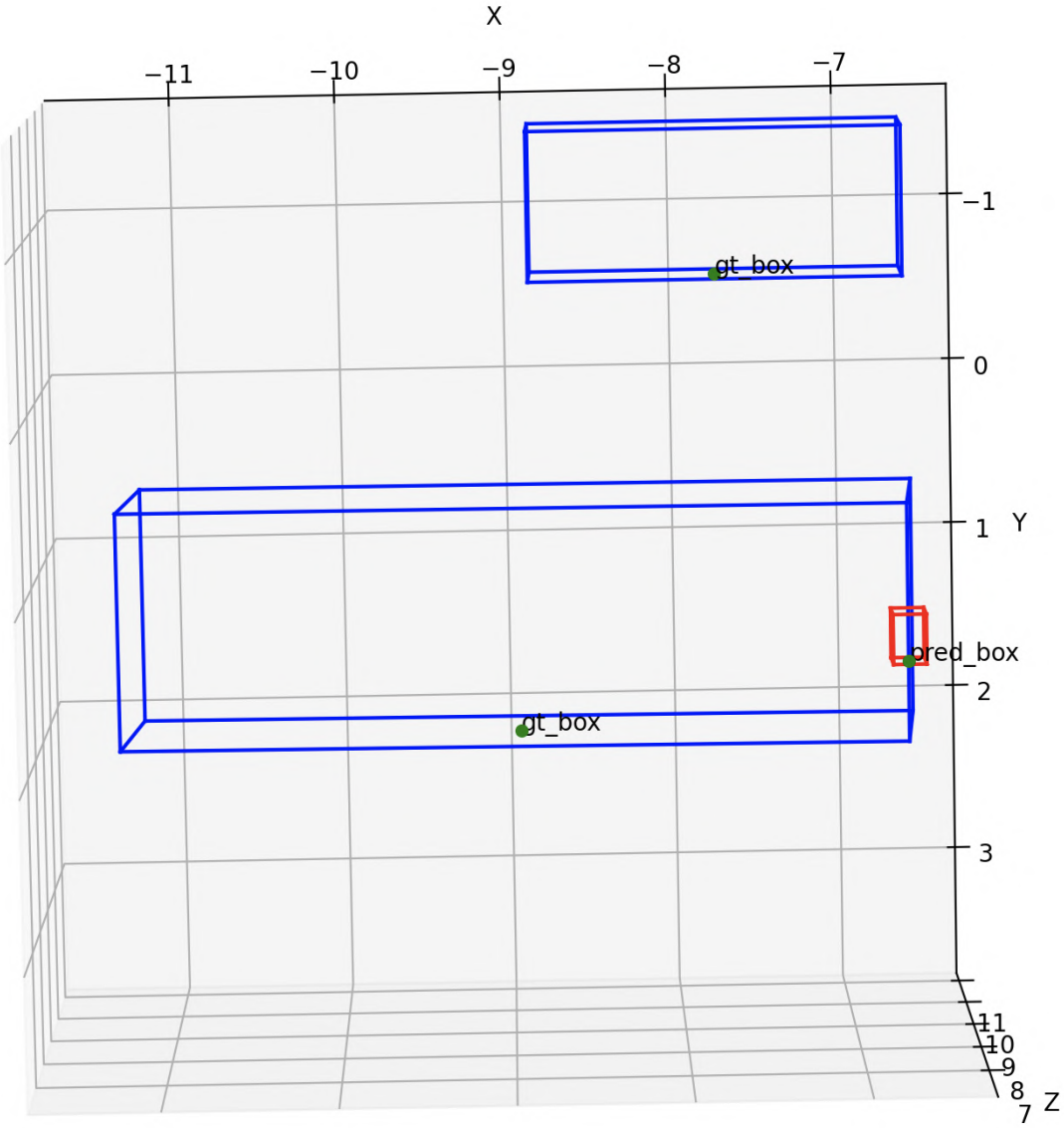


Figure A.4: Example 1 - Bad Car Prediction 3D Visualization

Bibliography

[1] X. Li, Y. Xiao, B. Wang, H. Ren, Y. Zhang, and J. Ji, "Automatic targetless lidar–camera calibration: a survey," *Artificial Intelligence Review*, vol. 56, no. 9, pp. 9949–9987, 2023.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.

[3] A. Rosebrock, "Intersection over union (iou) for object detection - pyimagesearch," Nov 2016. [Online]. Available: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

[4] J. Hui, "map (mean average precision) for object detection - jonathan hui - medium," Mar 2018. [Online]. Available: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173

[5] J. Terven and D. Cordova-Esparza, "A comprehensive review of yolo: From yolov1 to yolov8 and beyond," *arXiv preprint arXiv:2304.00501*, 2023.

[6] S. Hao, Y. Zhou, and Y. Guo, "A brief survey on semantic segmentation with deep learning," *Neurocomputing*, vol. 406, pp. 302–321, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220305476

[7] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics yolov8," 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[8] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.

[9] Jun 2024. [Online]. Available: https://www.a2d2.audi/a2d2/en/sensor-setup.html

[10] M. Contributors, "MMYOLO: OpenMMLab YOLO series toolbox and benchmark," https://github.com/open-mmlab/mmyolo, 2022.

[11] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A Survey of Autonomous Driving: Common Practices and Emerging Technologies," *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020.

[12] M. Testouri, G. Elghazaly, and R. Frank, "Robocar: A rapidly deployable open-source platform for autonomous driving research," 2024.

[13] Y. Li, A. W. Yu, T. Meng, B. Caine, J. Ngiam, D. Peng, J. Shen, B. Wu, Y. Lu, D. Zhou, Q. V. Le, A. Yuille, and M. Tan, "DeepFusion: Lidar-Camera Deep Fusion for Multi-Modal 3D Object Detection," 2022.

[14] X. Bai, Z. Hu, X. Zhu, Q. Huang, Y. Chen, H. Fu, and C.-L. Tai, "TransFusion: Robust LiDAR-Camera Fusion for 3D Object Detection with Transformers," 2022.

[15] Z. Liu, H. Tang, A. Amini, X. Yang, H. Mao, D. Rus, and S. Han, "BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation," 2022.

[16] T. Liang, H. Xie, K. Yu, Z. Xia, Z. Lin, Y. Wang, T. Tang, B. Wang, and Z. Tang, "BEVFusion: A Simple and Robust LiDAR-Camera Fusion Framework," 2022.

[17] Y. Jiao, Z. Jie, S. Chen, J. Chen, L. Ma, and Y.-G. Jiang, "MSMDFusion: Fusing LiDAR and Camera at Multiple Scales with Multi-Depth Seeds for 3D Object Detection," 2023.

[18] J. Yan, Y. Liu, J. Sun, F. Jia, S. Li, T. Wang, and X. Zhang, "Cross Modal Transformer: Towards Fast and Robust 3D Object Detection," 2023.

[19] X. Li, T. Ma, Y. Hou, B. Shi, Y. Yang, Y. Liu, X. Wu, Q. Chen, Y. Li, Y. Qiao, and L. He, "LoGoNet: Towards Accurate 3D Object Detection with Local-to-Global Cross-Modal Fusion," 2023.

[20] R. Kaur and S. Singh, "A comprehensive review of object detection with deep learning," *Digital Signal Processing*, vol. 132, p. 103812, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1051200422004298

[21] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 6, pp. 1137–1149, 2016.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[25] R. Padilla, S. L. Netto, and E. A. Da Silva, "A survey on performance metrics for object-detection algorithms," in *2020 international conference on systems, signals and image processing (IWSSIP)*. IEEE, 2020, pp. 237–242.

[26] J. Torres, "Yolov8 architecture: A deep dive into its architecture - yolov8," Jan 2024. [Online]. Available: https://yolov8.org/yolov8-architecture/

[27] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015. [Online]. Available: https://arxiv.org/abs/1405.0312

[28] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of data science*, vol. 2, pp. 165–193, 2015.

[29] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[30] mjoshi07, "Github - mjoshi07/visual-sensor-fusion: Lidar fusion with vision," 2022. [Online]. Available: https://github.com/mjoshi07/Visual-Sensor-Fusion

[31] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[32] N. Aharon, R. Orfaig, and B.-Z. Bobrovsky, "Bot-sort: Robust associations multi-pedestrian tracking," *arXiv preprint arXiv:2206.14651*, 2022.

[33] Y. Zhang, P. Sun, Y. Jiang, D. Yu, F. Weng, Z. Yuan, P. Luo, W. Liu, and X. Wang, "Bytetrack: Multi-object tracking by associating every detection box," 2022.

[34] 2012. [Online]. Available: https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d

[35] cguindel, "Github - cguindel/eval_kitti: Tools to evaluate object detection results using the kitti dataset." 2017. [Online]. Available: https://github.com/cguindel/eval_kitti

[36] J. Geyer, Y. Kassahun, M. Mahmudi, X. Ricou, R. Durgesh, A. S. Chung, L. Hauswald, V. H. Pham, M. Mühlegg, S. Dorn, T. Fernandez, M. Jänicke, S. Mirashi, C. Savani, M. Sturm, O. Vorobiov, M. Oelker, S. Garreis, and P. Schuberth, "A2D2: Audi Autonomous Driving Dataset," 2020. [Online]. Available: https://www.a2d2.audi

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[38] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[39] X. Zhao, W. Ding, Y. An, Y. Du, T. Yu, M. Li, M. Tang, and J. Wang, "Fast segment anything," 2023.

[40] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[41] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.