F3

Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

3D Object Detection for Autonomous Cars Weakly Supervised by 2D Cues

Bc. Jan Škvrna
Cybernetics and Robotics

January, 2024
Supervisor: Ing. Lukáš Neumann, Ph.D.

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Škvrna Jan** Personal ID number: **483569**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

3D Object Detection for Autonomous Cars Weakly Supervised by 2D Cues

Master's thesis title in Czech:

Detekce 3D objektů pro autonomní auta trénovaná pomocí 2D anotací

Guidelines:

1. Analyse existing methods to train 3D object detectors without 3D annotations [1, 2] and discuss their advantages and shortcomings.
2. Create a method to train a 3D object detector in LIDAR point clouds, which does not require laborious 3D annotations; instead exploit available 2D detections from RGB camera and other scene cues as the training signal to the 3D detector.
3. Exploit temporal consistency of the scene to collect more reliable training signal across multiple image frames and their corresponding LIDAR point cloud.
4. Evaluate the method at least on the standard KITTI [3] dataset and compare the achieved results to state-of-the-art methods in two categories – the standard 3D detectors which use 3D annotations for training and the detectors which do not require them.

Bibliography / sources:

[1] R. McCraith, E. Insafutdinov, L. Neumann and A. Vedaldi, "Lifting 2D Object Locations to 3D by Discounting LiDAR Outliers across Objects and Views," 2022 International Conference on Robotics and Automation (ICRA).
[2] Sergey Zakharov, Wadim Kehl, Arjun Bhargava, and Adrien Gaidon. Autolabeling 3D objects with differentiable rendering of SDF shape priors. 2020 Computer Vision and Pattern Recognition (CVPR).
[3] Geiger, A., Lenz, P., Stiller, C., & Urtasun, R. (2013). Vision meets robotics: The KITTI dataset. The International Journal of Robotics Research, 32(11), 1231-1237.

Name and workplace of master's thesis supervisor:

Ing. Lukáš Neumann, Ph.D. Visual Recognition Group FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **08.09.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until: **16.02.2025**

Ing. Lukáš Neumann, Ph.D. Supervisor's signature	prof. Ing. Tomáš Svoboda, Ph.D. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgement / Declaration

I want to thank my supervisor, Ing. Lukáš Neumann, Ph.D., for his unlimited ideas and perfect supervision of this thesis.

Further, I want to thank my family for endless support during my educational years and my girlfriend Anežka, who has always supported me.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 9.1.2024

.....

Abstrakt / Abstract

Přesná detekce objektů v LiDAR datech je klíčovým předpokladem pro robustní a bezpečné autonomní řízení a také robotické aplikace. Trénování 3D detektorů objektů v současnosti vyžaduje manuální anotaci velkého množství trénovacích dat, což je velmi časově náročné a nákladné. V důsledku toho je množství dostupných anotovaných trénovacích dat omezené a navíc tyto anotované datové sady pravděpodobně neobsahují vzácné případy, jednoduše proto, že pravděpodobnost jejich výskytu v tak malé datové sadě je nízká.

V této práci navrhujeme metodu pro trénování 3D detektoru objektů bez potřeby lidských anotací, a to využitím existujících vizuálních komponent a konzistence světa kolem nás. Metoda proto může být použita pro trénování 3D detektoru pouze pomocí sběru dat ze senzorů v reálném světě, což je extrémně levné a umožňuje trénování s řádově více daty než tradiční plně supervizované metody.

Metoda byla evaluována jak na validačních, tak testovacích datech KITTI, kde překonává všechny předchozí slabě supervizované metody a zmenšuje rozdíl přesnosti s metodami využívajících lidské 3D anotace.

Klíčová slova: slabě, supervizované, trénování, 3D, objekt, detekce, KITTI, konzistence

Accurate object detection in LiDAR point clouds is a key prerequisite of robust and safe autonomous driving and robotics applications. Training the 3D object detectors currently involves the need to manually annotate vasts amounts of training data, which is very time-consuming and costly. As a result, the amount of annotated training data readily available is limited, and moreover these annotated datasets likely do not contain edge-case or otherwise rare instances, simply because the probability of them occurring in such a small dataset is low.

In this thesis, we propose a method to train 3D object detector without any need for manual annotations, by exploiting existing off-the-shelf vision components and by using the consistency of the world around us. The method can therefore be used to train a 3D detector by only collecting sensor recordings in the real world, which is extremely cheap and allows training using orders of magnitude more data than traditional fully-supervised methods.

The method is evaluated on the both KITTI validation and test datasets, where it outperforms all previous weakly-supervised methods and where it narrows the gap when compared to methods using human 3D labels.

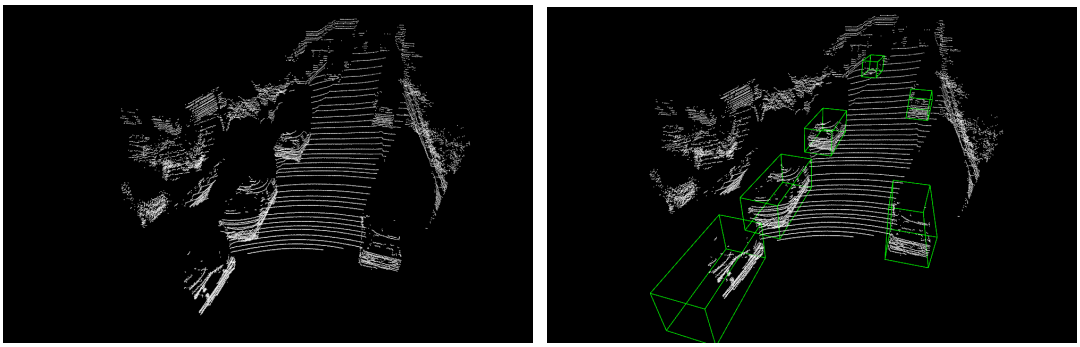
Keywords: weakly, supervised, training, 3D, object, detection, KITTI, temporal, consistency

Contents /

Tables / Figures

Chapter 1
Introduction

Accurate and swift 3D object detection in LiDAR point clouds is a critical component of various real-time applications, ranging from robotics to autonomous driving. These fields have received increasing attention in the last decades due to their potential impact on everyday life. One of the main limitations that currently hinders the exploitation of 3D object detectors in real-world scenarios is the scarcity of labelled training data, owing to the fact that human labelling in 3D is very time-consuming and therefore costly, as labelling one object instance can take up to 100 seconds [1–2].



a) b)

Figure 1.1. Example of the 3D detection on the raw LiDAR scan (a), 3D ground truth labels (b). LiDAR scan and ground truth labels come from the KITTI [3] dataset.

While publicly available datasets like KITTI [3], Waymo open [4], or NuScenes [5] feature hundreds of thousands of annotated scenes, they fall short of capturing the diversity of possible events or environments.

Various tools can aid human annotators in significantly reducing the time spent per instance. All off-the-shelf pre-trained fully-supervised 3D detectors can be used as an estimate of the 3D bounding boxes and then the human must perform only the fine refinement of the 3D bounding boxes. Additionally, specialized methods focused on autolabeling, such as WS3Dv2 [6], have demonstrated the ability to reduce the time spent per instance to 2.6 seconds, though with a tradeoff in accuracy.

On the other hand, vasts amounts of data are readily available, because capturing and storing sensor data is relatively cheap. As almost all modern cars are connected to the internet, equipping them with appropriate sensors and sending them to millions of customers all over the world, enables the possibility to capture very rare events at almost zero cost. However, the data are just not annotated and therefore useless for traditional fully-supervised 3D detection methods.

In this Master's Thesis, we aim to narrow this gap by training a standard 3D object detector, but **without using any human labels** in the process, therefore allowing the detector to be trained using the large quantities of unlabelled data readily available. Instead, We exploit an off-the-shelf 2D detector for the RGB camera (trained on a

generic non-related dataset such as MS COCO [7]) and a number of real-world priors such as multiple generic shapes of cars, temporal consistency between frames or the fact other objects only move subject to constraints given by the laws of physics between individual frames to train the detector. The result of our training process is a traditional 3D object detector that operates on LiDAR point clouds; the only yet crucial difference is the training signal (the training loss) used for the training, which does not rely on human annotations.

We've chosen to apply our method to the KITTI dataset, as it's the standard in all related methods, allowing for an easy comparison of our results. Additionally, this dataset contains all the necessary data for our method.

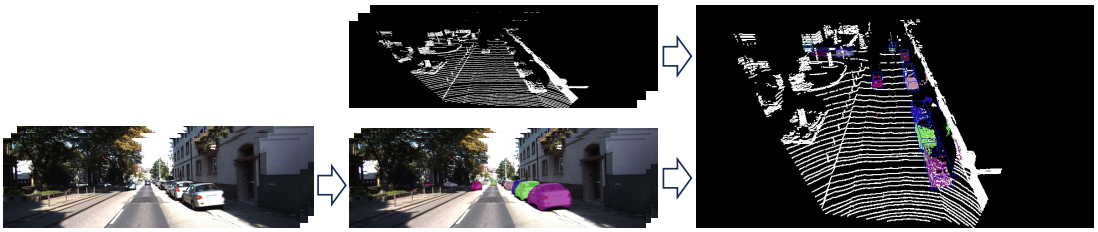


Figure 1.2. Combining raw unlabelled RGB camera and LiDAR sensor data across multiple frames in a temporally consistent manner allows the exploitation of a generic off-the-shelf 2D object detector to train a 3D object (vehicle) detector for LiDAR point clouds.

Chapter 2 describes multiple publicly available datasets, focusing on the amount of data and type of data provided, as well as on the sensor setup. Further, it details fully-supervised methods for 3D object detection and also all other relevant weakly-supervised methods for 3D vehicle detection.

Chapter 3 details the use of an off-the-shelf 2D detector for obtaining 2D instance segmentation masks. The chapter discusses how these masks are used for segmentation in 3D data, including post-processing steps like outlier removal and spatial location estimation.

Chapter 4 covers the exploitation of temporal consistency. It includes tracking vehicles over multiple frames, classifying them as standing or moving, and using this classification for point aggregation or yaw estimation. The frame-to-frame transformations and fine-tuning with the Iterative Closest Point [8] algorithm are also discussed.

Chapter 5 describes the generic car shape template for the fitting process and the fitting process for moving and standing cars. It also explores various loss functions for the fitting process and discusses additional fitting ideas that were less successful.

Chapter 6 focuses on the Scale Detector, which uses information from the fitting process to estimate car spatial dimensions. The chapter also introduces the Bounding Box Reducer for refining scale detection outputs as the Scale Detector tends to estimate inadequately large bounding boxes.

Chapter 7 details how a fully-supervised 3D detector (Voxel-RCNN [9]) is adapted for weakly-supervised training. It includes discussions on the Voxel-RCNN properties and the incorporation of information from previous chapters into the training loop, including two new loss functions added to the training loop.

Chapter 8 provides a range of experiments supporting the decisions made during the method's development.

Further, Chapter 9 compares the method with other fully and weakly supervised methods on the KITTI dataset in both BEV and 3D metrics and illustrates the method's performance on real-world scenes from the KITTI dataset.

Lastly, in Chapter 10, the whole thesis is concluded and the results are commented on.

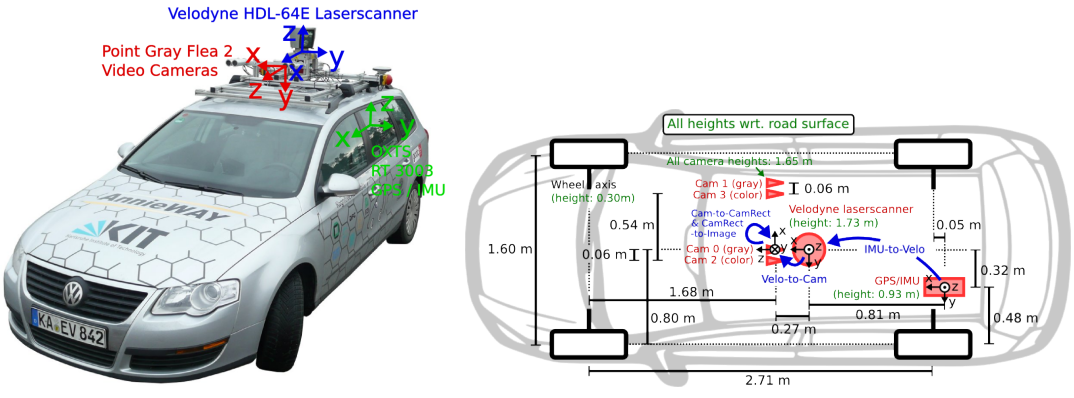
Chapter 2
Related work

First, We describe three main publicly available datasets for autonomous driving in Section 2.1. We focus on the quantity and type of the data, along with the specifics of the sensor setup. Second, We describe various fully-supervised 3D detectors in Section 2.2, starting with the fundamental ones and progressing to the latest state-of-the-art detectors. One of them is chosen and used to be trained under weak supervision in the following chapters. Third, We describe related weakly-supervised methods solving the same challenge as we do, however with different approaches in Section 2.3. This thorough exploration establishes the groundwork for the chapters to follow.

2.1 Datasets

2.1.1 KITTI

One of the most renowned open datasets for autonomous driving is the KITTI dataset [3], captured near Karlsruhe, Germany. The 3D object detection subset contains 7481 training and 7518 testing frames, featuring a total of 80256 labelled objects. This subset is fully annotated in both 2D and 3D, labelling various entities such as cars, vans, trucks, pedestrians, etc. As we leverage the temporal consistency, we use the KITTI raw dataset [10], which contains sequences of unlabelled data from which the 3D object detection subset is derived. The KITTI raw dataset consists of a total of 47885 frames.



a) b)

Figure 2.1. KITTI [3] dataset setup. The figures are taken from [10].

The dataset incorporates four cameras, two RGB and two grayscale, all mounted on the roof and facing in the forward direction of the ego-vehicle. Additionally, a rooftop-mounted LiDAR scanner and a precise IMU unit capturing angular velocities, accelerations, and location via GNSS contribute to the dataset. Detailed sensor calibrations are provided, and a visual representation of the setup is presented in Figure 2.1.

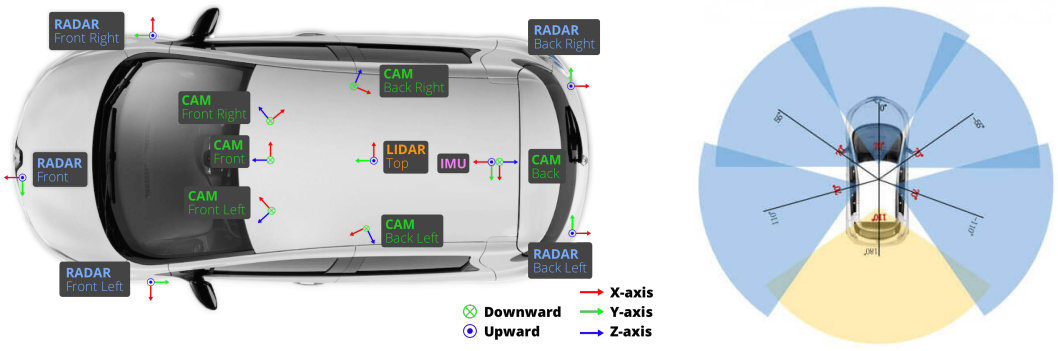
All objects are labelled with regards to the RGB camera with index 2. Thus, as every labelled object has to be visible in the camera, the amount of labelled cars in the LiDAR scan is low, because the LiDAR scanner captures the whole surroundings of the ego-vehicle. Every object label contains 8 parameters: type (car, truck, etc.), truncation, occlusion, alpha (observation angle of an object), 2D bounding box, 3D dimensions, 3D location, and yaw.

2.1.2 NuScenes

The NuScenes dataset [5] draws inspiration from the KITTI dataset [3], expanding it with approximately 7 times more object annotations. Collected in Boston (USA) and Singapore, the dataset exhibits diversity compared to the KITTI dataset, reflecting distinct road and car appearances on each continent. Comprising 1000 scenes, each lasting 20 seconds, these scenes are specifically chosen to showcase a diverse set. Labels for all scenes are provided at a rate of 2 Hz, resulting in around 1.4 million camera images, 390,000 LiDAR scans, and 1.4 million object bounding boxes across 40,000 keyframes (frames sampled at 2Hz). NuScenes stands out as a significantly larger dataset compared to the KITTI dataset.

The dataset employs six RGB cameras, strategically positioned to cover the whole surroundings of the car. A LiDAR scanner is mounted on the roof. Additionally, a precise IMU unit captures angular velocities, accelerations, and location via GNSS, which is enhanced by HD lidar maps. The dataset is further equipped with five radars, strategically placed to cover all angles around the vehicle. Detailed sensor calibrations are provided, and a visual representation of the setup is presented in Figure 2.2.

Since the cameras cover the whole surroundings, all objects around the ego-vehicle are labeled in the NuScenes dataset. There are 23 distinct object classes utilized for labeling, and each label comprises 12 parameters. We omit descriptions for parameters identical to those in the KITTI labels, NuScenes labels introduce an instance token, assigning a unique ID to all objects. Additionally, the label of the object provides the number of lidar points within the 3D bounding box, the number of radar points, and pointers to the next or previous annotation of the object in time. Notably, NuScenes labels lack information on the difficulty of the object, truncation, alpha rotation, and, most significantly, the **2D bounding box** when compared to the KITTI dataset.



a) b)

Figure 2.2. (a) NuScenes dataset [5] setup, (b) camera overlap. The figures are taken from [5].

2.1.3 Waymo Open

Waymo Open Dataset is composed of two datasets. The perception [4] and motion [11] dataset. For our specific task, only the perception dataset is relevant, as it features precise 3D bounding box labels. Captured across multiple cities in the USA, the dataset consists of 2030 segments, each lasting 20 seconds and sampled at 10Hz. In total, it consists of 1.2 million images and LiDAR scans. Thus it has 2x more segments, but a similar number of images compared to NuScenes [5]. If we focus only on the vehicles, It provides high-quality labels for objects in 1200 segments, in total 12.6 million 3D bounding box labels. Similar to NuScenes, precise 3D labels are provided at a rate of 2Hz.

The dataset employs five cameras positioned on the roof of the car, pointing forward and to the sides, collectively achieving a field of view (FOV) exceeding 180°. Additionally, a mid-range LiDAR scanner is mounted on the roof. Four short-range LiDAR scanners are mounted at the front, front-left, front-right and back of the car. The dataset also integrates a precise IMU unit, ensuring accurate capture of angular velocities, accelerations, and location data. Detailed sensor calibrations are provided, and a visual representation of the setup is illustrated in Figure 2.3.

All objects surrounding the ego-vehicle are labelled, meaning a 3D bounding box doesn't necessarily require a 2D bounding box correspondence since the object may not be visible in any camera. The dataset employs only four object classes: vehicle, pedestrian, cyclist, and traffic sign. No label zones are utilized to disregard irrelevant zones (e.g. the opposite side of the highway). Labels consist of 3D dimensions, 3D locations, yaws, 2D bounding boxes, and 2D-3D correspondences of the objects. Each object is assigned a unique ID for tracking purposes. Notably, the dataset labels lack information on the difficulty, truncation, and alpha rotation of the object when compared to the KITTI dataset.

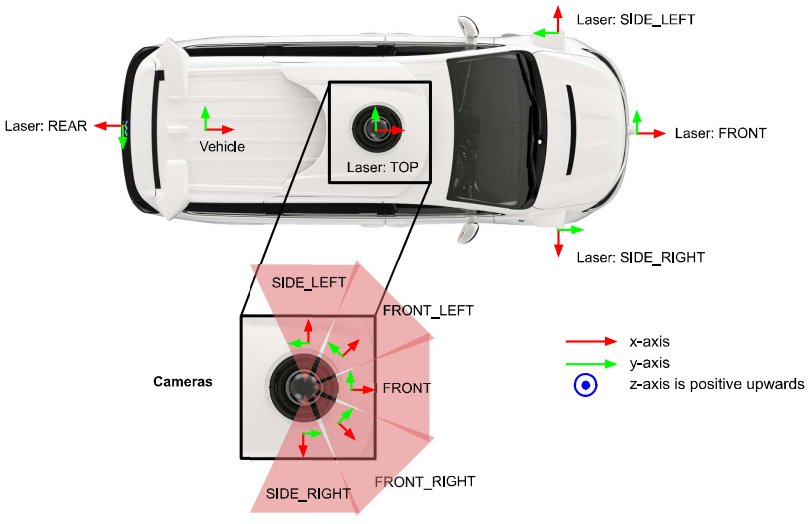


Figure 2.3. Waymo Open dataset [4] setup. The figure is taken from [4].

2.2 Fully supervised 3D detectors

Fully-supervised 3D detector architectures typically fall into two primary streams: point-based and voxel-based methods. Point-based methods directly utilize the unordered point set (LiDAR scans) as input. This approach retains all information present

in the original data, as the point set is not transformed or voxelized. On the other hand, voxel-based methods transform the unordered point set (LiDAR scan) into a voxel grid. Voxelization involves dividing the 3D space into small, regularly spaced volumetric units called voxels. This regular data format allows for efficient storage and memory access. With the precise memory location of each voxel and its neighbours known, 3D convolutions can be applied efficiently. However, it's important to note that during voxelization, the precise location of each individual point is lost, as they are aggregated within the voxel grid.

2.2.1 PointNet

PointNet [12] is a representative example of point-based methods. It introduces a unified architecture applicable to various tasks such as instance segmentation, object classification, and more. Pointnet modules are a foundation for many other methods, e.g. PointNet++ [13] or Voxel-RCNN [9]. A key concept in PointNet is max pooling. This method effectively learns a set of optimization functions or criteria that identify and encode the interesting points, along with the reason for their selection. These encoded points are then utilized in a fully connected layer to predict the desired output, be it instance segmentation, object detection, or other related tasks.

The detailed pipeline of the PointNet is shown in Figure 2.4. The input to the network is a point cloud of shape $n \times 3$, where n denotes the number of points. Points are first transformed into a unified coordination space, as the network should be invariant to the rigid transformation of the input. Subsequently, each point undergoes encoding through an MLP (Multi-Layer Perceptron), transitioning from $n \times 3$ to $n \times 64$. The features of these points are then once again transformed through a rigid transformation and fed into another MLP, producing a final shape of $n \times 1024$ from $n \times 64$.

Up to this point in the pipeline, features of individual points have been extracted independently. To capture global features, max pooling is applied, selecting the maximum value across all points for each feature channel. These global features are then fed into another MLP to obtain output scores corresponding to k classes.

As can be seen, the whole network is invariant with the ordering of the input points. While the Segmentation Network's details haven't been explicitly described, the fundamental idea of PointNet, which forms the basis of subsequent methods, has been outlined. Despite its simplicity, PointNet proved itself fast and effective.

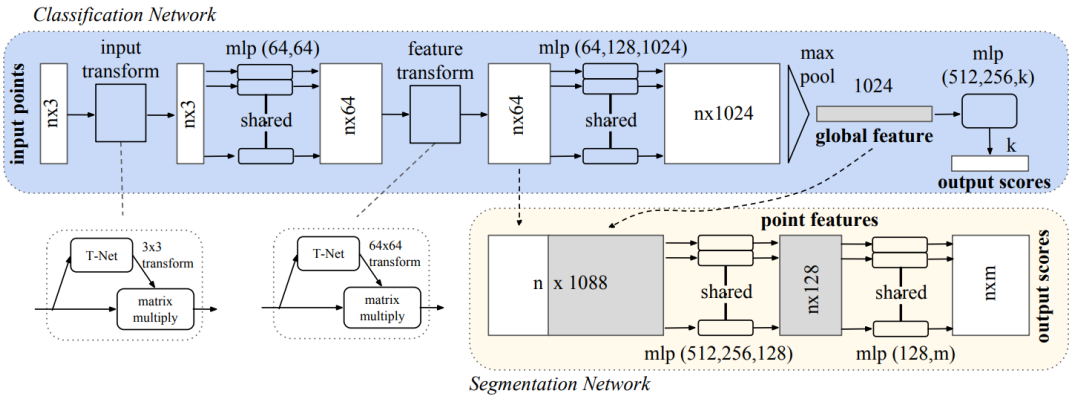


Figure 2.4. Pipeline of PointNet [12] described in Section 2.2.1. The figure is taken from [12].

2.2.2 VoxelNet

PointNet operates on point clouds with approximately 1k points due to limitations in memory and computation. However, to adequately describe the complex scenes around autonomous vehicles, a more extensive point cloud is often required. VoxelNet [14] addresses these limitations, allowing for the use of point clouds with approximately 100k points. While the smaller point clouds used by PointNet are suitable for tasks like 3D object classification or segmentation, a larger number of points becomes essential for capturing the scene in adequate detail around an autonomous vehicle. VoxelNet consists of three key blocks: FLN (Feature Learning Network), CML (Convolutional Middle Layers), and RPN (Region Proposal Network). The pipeline is shown in Figure 2.5.

Within the Feature Learning Network (FLN), a voxel grid is created, and each point is assigned to a specific voxel. To account for potential variations in point density across voxels, a random downsampling mechanism is applied, limiting the number of points within each voxel to a predefined threshold. The points within the voxel are encoded through multiple stacked Voxel Feature Encoders (VFEs), resulting in the extraction of voxel-wise features.

A 4D tensor of size $C \times D' \times H' \times W'$ is created, containing a voxel feature vector of length C per voxel. D' denotes z-axis, H' denotes y-axis, W' denotes x-axis of a given world coordinate space. Subsequently, this 4D tensor is fed into the Convolutional Middle Layers (CML), mainly composed of 3D convolution layers. These layers aggregate voxel features via increasing receptive fields of the 3D convolutions.

The aggregated voxel features are then employed in the Region Proposal Network (RPN) to generate 3D bounding boxes. Proper handling of the sparsity within the 4D tensor is crucial for maintaining computational efficiency. However, the method runs at ~4Hz.

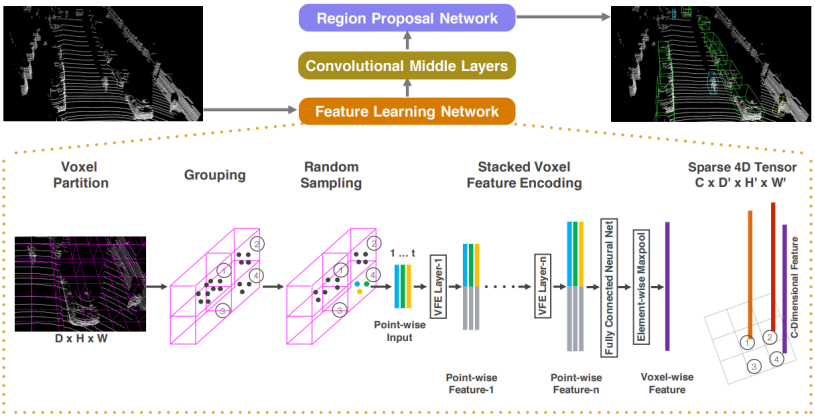


Figure 2.5. Pipeline of VoxelNet [14] described in Section 2.2.2. The figure is taken from [14].

2.2.3 PointPillars

PointPillars [15] key idea is to encode a 3D point set into a Bird's Eye View (BEV) grid, allowing the utilization of fast 2D convolutions on modern GPUs. Positioned within the category of voxel-based methods, PointPillars achieved state-of-the-art performance on the KITTI object detection benchmark at the time of its release. Notably, it operates at a remarkable speed of 62 Hz, making it 2-4 times faster than other methods and highly applicable for real-time applications. The pipeline is shown in Figure 2.6.

PointPillars consists of three main components: first, the point cloud is encoded into a pseudo-image (BEV grid); second, features are extracted from the pseudo-image using a 2D convolutional backbone; third, the detection head regresses 3D bounding boxes. It is worth noting that a simplified version of PointNet is employed in the initial stage of the PointPillars method.

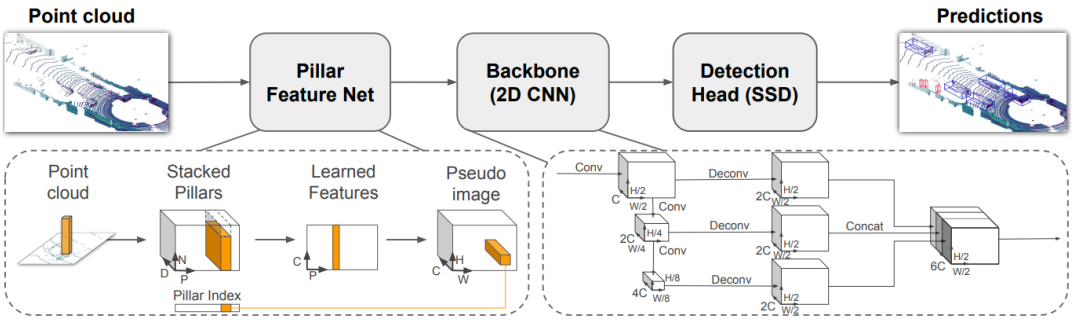


Figure 2.6. Pipeline of PointPillars [15] described in Section 2.2.3. The figure is taken from [15].

2.2.4 PV-RCNN

Point-Voxel RCNN (PV-RCNN) [16] integrates both 3D voxel Convolutional Neural Network (CNN) and PointNet-based [12] networks, combining the strengths of both main streams. Upon its release, PV-RCNN set a new standard in performance on the KITTI [3] and Waymo Open [17] datasets. The pipeline is shown in Figure 2.7.

The PV-RCNN architecture consists of two stages. In the initial stage, a 3D sparse convolutional network operates on voxelized points, generating voxel-wise features and 3D proposals using a Region Proposal Network (RPN) on stacked features in the Bird's Eye View (BEV). As it would be unfeasible to use all points inside the proposal, only specific keypoints are sampled using Furthest Point Sampling (FPS). The features of each keypoint are then aggregated from neighbouring voxels across all 3D CNN layers using the PointNet-based set abstraction. This strategy enables the use of only a few keypoints for describing the entire scene.

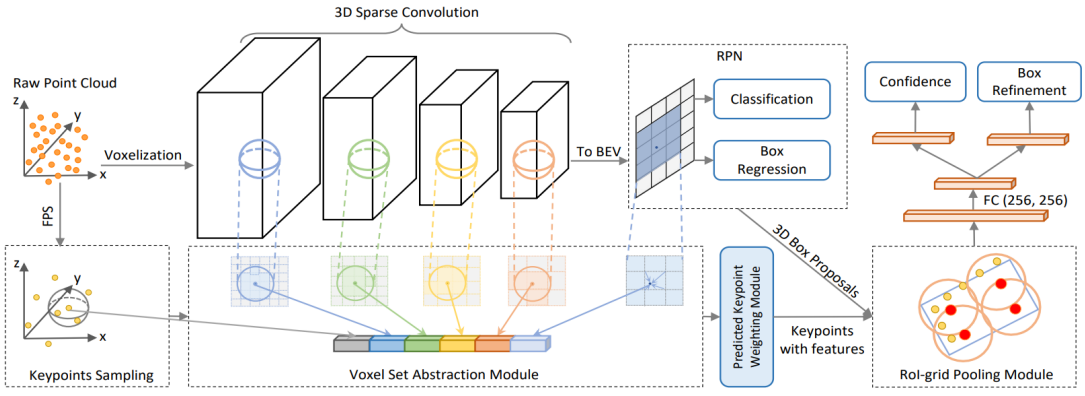


Figure 2.7. Pipeline of PV-RCNN [16] described in Section 2.2.4. The figure is taken from [16].

Moving to the second stage, the 3D proposal undergoes uniform sampling to obtain grid points. For each grid point, Region of Interest (RoI) pooling is applied over neighbouring keypoints. RoI grid features are used to get fine-refined 3D bounding boxes

and confidence scores. This two-stage approach allows the state-of-the-art performance of PV-RCNN.

2.2.5 Voxel-RCNN

Voxel-RCNN [9] falls into the voxel-based stream, demonstrating that maintaining precise point locations isn't necessary. Voxel-RCNN achieves performance comparable to state-of-the-art methods while operating at an impressive 25 Hz (on Nvidia RTX 2080TI), showcasing significant speed improvements over other methods. The pipeline, illustrated in Figure 2.8, consists of three key components.

In the first part, the 3D BackBone network takes voxelized points as input and produces voxel-wise features. These features are stacked along the z-axis, in the second stage, to create a Bird's Eye View (BEV) grid. Then, the Region Proposal Network (RPN) head, utilizing 2D convolutions, generates proposals out of the BEV grid.

As the second part generates proposals equal to the number of anchors (tens of thousands), only a small subset of proposals is selected for the third stage based on the Intersection over Union (IoU) with ground truth during training or predicted score during inference. The third stage uses voxel-wise features from the first stage and a subset of proposals from the second stage.

Proposals from the second stage are divided into sub-voxels, with each sub-voxel functioning as a query point. For each query point, neighbouring voxel features are aggregated, and features are then extracted using a PointNet module. However, as the number of sub-voxels is big, the computation of the Voxel RoI pooling is computationally demanding. To address this, Voxel-RCNN introduces an accelerated PointNet module.

Instead of applying the PointNet module separately to the grouped features for each query point (sub-voxel), the PointNet module is split into two parts. The first part operates on voxel-wise features, and the second part focuses on sub-voxel locations. Those PointNet modules are used to precompute features of all voxel-wise features (First stage) and sub-voxel locations (Second stage proposals). At query time, these precomputed features are merged, significantly enhancing computation efficiency.

The features extracted from Voxel RoI pooling are then utilized in the detection head, which produces finely-refined predictions and scores. Notably, Voxel-RCNN's output scores represent predictions of IoU with the ground truth.

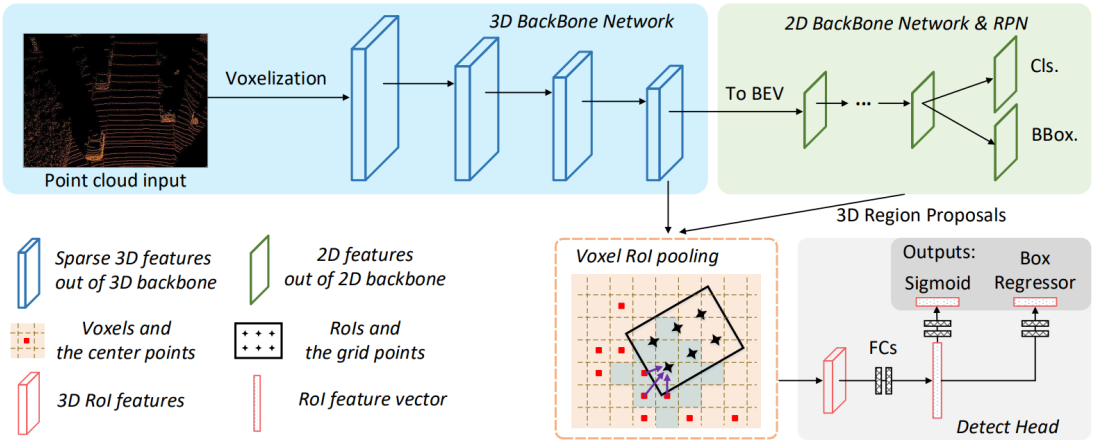


Figure 2.8. Pipeline of Voxel-RCNN [9] described in Section 2.2.5. The figure is taken from [9].

2.2.6 CasA

Cascade Attention Network (CasA) [18] serves as an adaptation of current two-stage architectures like PV-RCNN [16], Voxel-RCNN [9], etc. It adapts the well-known Cascade networks in 2D to 3D, as the vanilla cascade network directly applied to 3D fails due to the different properties of images and point clouds, proposing a novel Cascade Attention Module (CAM). This innovative module enhances performance by approximately 1-2% when integrated into Voxel-RCNN. The pipeline is shown in Figure 2.9.

In the initial step, the point cloud undergoes voxelization and is employed in the Region Proposal Network (RPN). The RPN starts with 3D convolutions, transforms the voxel grid to a Bird's Eye View (BEV) grid, and utilizes 2D convolutions to generate proposals and corresponding scores. These proposals serve as inputs to the Cascade Refinement Network (CRN), which consists of multiple Cascade Attention Modules.

Within each stage of the CRN, the input for the Cascade Attention Module is created by concatenating features from both the previous and current stages. The CRN serves as a fine refinement of the proposals, contributing to the overall enhancement of the detection process.

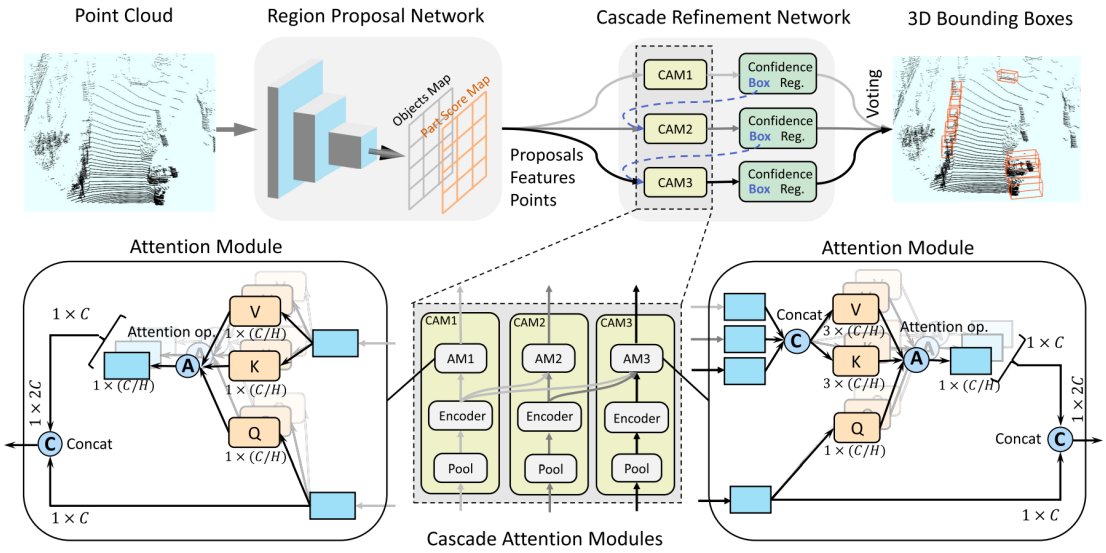


Figure 2.9. Pipeline of CasA [18] described in Section 2.2.6. The figure is taken from [18].

2.3 Weakly supervised 3D detectors

The section starts with weakly supervised methods using no ground truth labels at all and progresses into state-of-the-art methods, which use a small subset of the ground truth labels. Current state-of-the-art methods are described and the performance of each method compared with the fully supervised methods is discussed. As weakly supervised 3D training is challenging, each method proposes a unique solution to the problem and a step forward in the field.

2.3.1 VS3D

VS3D [19] operates without any ground truth labels and consists of an unsupervised 3D object proposal module (UPM) coupled with cross-modal transfer learning. The pipeline is shown in Figure 2.10.

UPM is used to find regions with a high probability of containing an object. This is achieved by leveraging the point cloud density as an indicative factor. As the point density of an object is dependent on the distance, normalized point density is used to mitigate the dependency. The anchor and points are projected into the image plane. The anchor and points are projected onto the image plane, followed by cropping the image to the projected anchor size and resizing it to a constant size. Given the sparsity of point clouds, inpainting [20] is applied to the projected points.

The Regions of Interest (RoI) identified by the UPM must be classified and regressed. Cross-modal transfer learning consists of a teacher and a student architecture. An off-the-shelf 2D detector, pretrained on datasets such as PASCAL VOC [21] or Imagenet [22], serves as the teacher. Each RoI is utilized to crop the RGB image, which is then input into the 2D detector. The 2D detector outputs classification labels and viewpoint estimation, serving as an object yaw estimation. In the student network, all points are projected into a depth image. This depth image is processed by a CNN backbone, followed by a RoIAlign [23] layer and fully connected layers. Similar to the teacher network, the student network performs classification and object yaw estimation.

When initially introduced, VS3D demonstrated state-of-the-art performance on the KITTI [3] dataset. However, it is important to note that the 3D performance gap compared to fully supervised methods remains significant (approximately 40% at 3D IoU 0.3), attributed to the absence of fine-tuning of RoI proposals and poor utilization of the 3D domain.

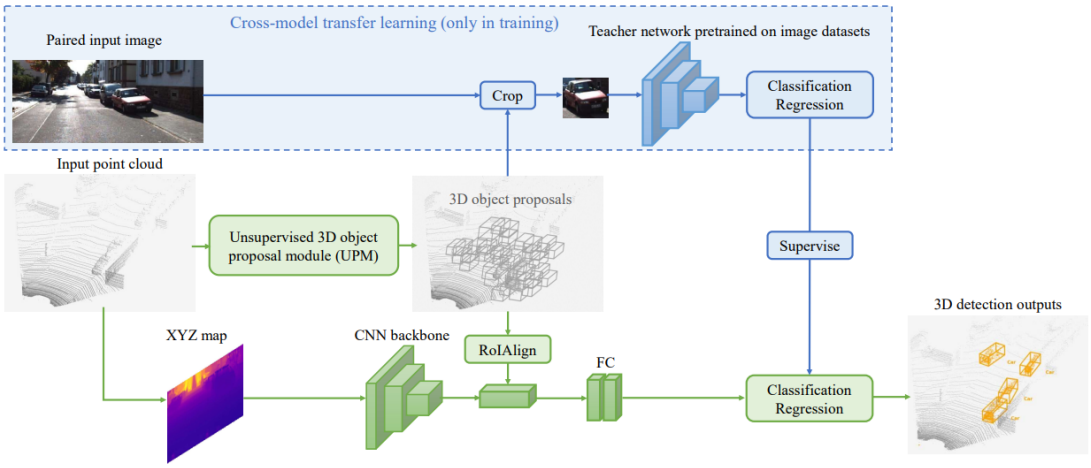


Figure 2.10. Pipeline of VS3D [19] described in Section 2.3.1. The figure is taken from [19].

2.3.2 Zakharov et al.

Zakharov et al. [24] employ an off-the-shelf 2D detector, pre-trained on a generic non-related dataset, with a novel differentiable renderer within the DeepSDF framework [25] to autonomously label 3D bounding boxes of cars. The pipeline is shown in Figure 2.11.

Initially, 2D detections are generated by the 2D detector. The Coordinate Shape Space (CSS) Net then produces a 2D Normalized Object Coordinate Shape (NOCS) map. As there is no ground truth available for NOCS maps, the CSS Net is trained using a synthetic dataset. The NOCS map is then projected back into 3D points and merged with LiDAR points filtered using 2D detection. An object's shape, functioning as a 3D prediction, is derived from the NOCS map. A render of an object's shape must

be aligned with the NOCS map, but at the same time, it has to be aligned with the LiDAR points.

The method is pre-trained on the synthetic dataset, followed by training on the actual KITTI [3] dataset, progressively adding more complex samples iteratively. Remarkably, the method achieves good accuracy on easy and moderate targets without utilizing any human-labeled data. However, no evaluation is provided for hard examples. While outperforming VS3D [19] by a large margin, a noticeable performance gap (approximately 60% at 3D IoU 0.7, 15% at 3D IoU 0.5) remains when compared to fully-supervised methods.

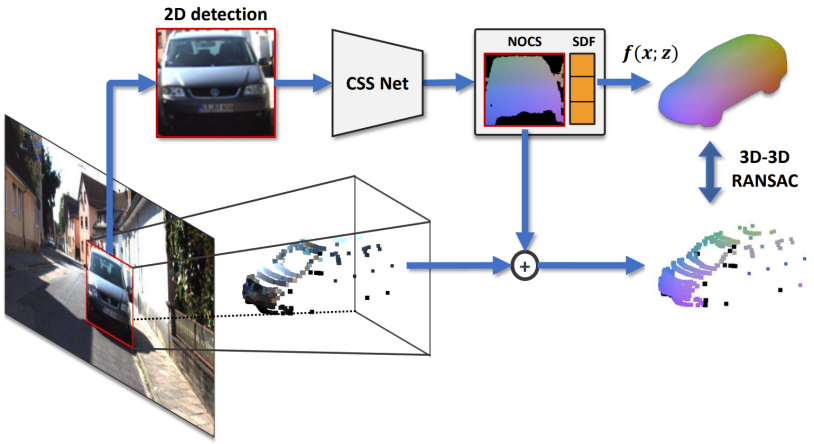


Figure 2.11. Pipeline of Zakharov et al. [24] described in Section 2.3.2. The figure is taken from [24].

2.3.3 McCraith et al.

McCraith et al. [26] leverage an off-the-shelf 2D detector and direct optimization of a template mesh onto LiDAR point clouds. The system employs a model such as Mask RCNN [23], pre-trained on a generic non-related dataset, to generate 2D masks corresponding to objects, particularly cars. The pipeline is shown in Figure 2.12.

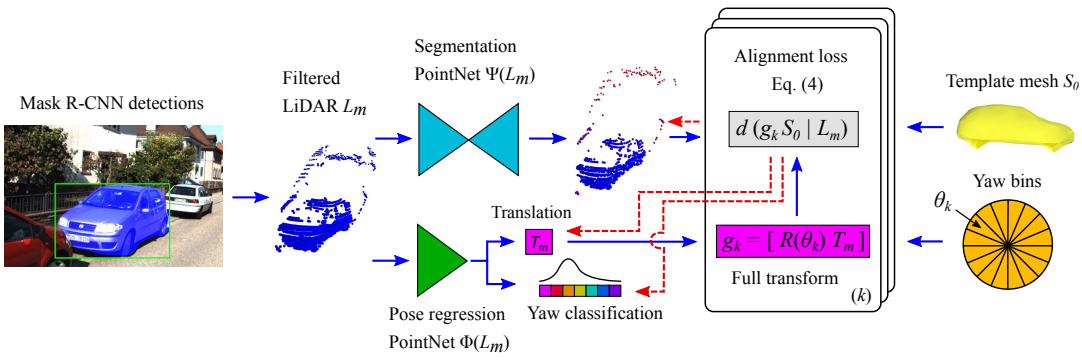


Figure 2.12. Pipeline of McCraith et al. [26] described in Section 2.3.3. The figure is taken from [26].

For each detection, LiDAR points corresponding to the mask are filtered to obtain a raw point cloud containing only points of the object. The Pose Regression PointNet [12] fits the object point cloud with a generic car shape using a one-way Chamfer Distance

loss. The loss calculation uses multiple frames to exploit temporal consistency. Given that the object point cloud often contains numerous outliers, a Segmentation PointNet is employed to assign a number to each point in the object point cloud, representing the variance of the point. This variance is used in the fitting loss. Since yaw prediction can be ambiguous (with $\pm 90^{\circ}$), each prediction from the Pose Regression PointNet undergoes evaluation across all possible rotations, with the one minimizing the loss chosen for back-propagation.

The method shows the significance of proper outlier handling and the exploitation of temporal consistency in achieving good accuracy. Operating without any human-labeled data, the method demonstrates promising results in the easy category of the KITTI [3] dataset. However, a noticeable performance gap (approximately 14% at 3D IoU 0.5) exists on moderate and hard examples.

2.3.4 FGR

Frustum-aware geometric reasoning (FGR) [27] uses 2D bounding boxes ground truth as labelling 2D bounding boxes is 3-16 times faster than 3D bounding boxes. FGR consists of coarse 3D segmentation and 3D bounding box estimation. The pipeline is shown in Figure 2.13.

In the initial stage of coarse 3D segmentation, FGR employs the RANSAC algorithm [28] to remove points in the LiDAR scan corresponding to the ground plane. Subsequently, it extracts points from the LiDAR scan within the frustum area corresponding to each provided 2D bounding box. To address potential occlusions of cars by other vehicles in the scene, the algorithm prioritizes the extraction of the closest cars to the ego-vehicle. Iteratively a random point from the extracted frustum is chosen and its connected component is computed. This process is done until all points within the extracted frustum area have a corresponding connected component. The connected component with the highest point count is then chosen as the representation of the object.

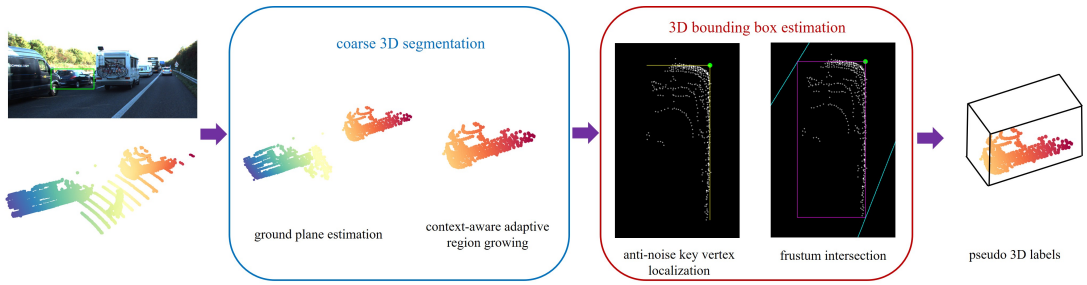


Figure 2.13. Pipeline of FGR [27] described in Section 2.3.4. The figure is taken from [27].

3D bounding box estimation first performs the Anti-noise Key Vertex Localization. It operates in Bird's Eye View (BEV) and finds the key vertex, defining the BEV bounding box. The objective is to find a key vertex with two edges in a way that minimizes the number of points with a greater distance than a specified threshold. In the optimization process, the removal of an outlier creates a change in the optimal bounding box, while the removal of an inlier has a negligible impact on the outcome. This approach effectively handles outliers. Subsequently, an intersection with the frustum area is employed to generate precise 3D bounding boxes from 2D bounding boxes. It's important to note that this process relies on accurate **amodal** 2D bounding boxes.

Generated pseudo ground truth 3D labels are used to train a 3D detector. The method achieved good results on the Easy and Moderate targets on the KITTI [3] dataset when the pseudo labels were used to train PointRCNN [29]. The observed performance gap (3D IoU 0.7) compared to fully-supervised PointRCNN ranges between 3% to 13%.

2.3.5 WS3Dv2

WS3Dv2 [6] uses human-labeled BEV click-point annotations, representing the object's center. It consists of two stages: object center detection and foreground segmentation, and 3D bounding box regression. The pipeline is shown in Figure 2.14.

In the initial stage of WS3Dv2, a network architecture resembling the encoder-decoder structure found in PointNet++ [13] is employed. The primary objective is to identify foreground points within the point cloud. Each point is assigned a probability indicative of its likelihood to be part of the foreground. The network is trained using pseudo-ground truth, generated by creating a Gaussian distribution based on the distance between a point and the nearest object center.

Simultaneously, another network is employed for predicting the object centers. Points with predicted foreground values surpassing a predefined threshold are considered. Each foreground point predicts an object center. Human-annotated point clicks, representing object centers, serve as the ground truth for training.

To ensure the quality of predictions, an Objectness-score-based Non-maximum Suppression strategy is employed, filtering out redundant predicted object centers. Object centers are defined by foreground points. The training process for this stage involves BEV maps featuring point clicks generated by human annotators.

The second stage is a cascade network, initially transforming the cylindrical proposals into a cuboid and then refining the cuboid in each layer. It consists of several set abstraction layers and an MLP branch at the end. The output is 3D bounding boxes and confidences generated by the IoU-based Confidence Estimation branch.

The first stage of WS3Dv2 is trained on 500 frames containing human-labeled BEV click-point maps. The second stage is trained on 534 precisely labeled objects extracted from the same 500 frames. This implies that not all objects within a given scene are used for training in the second stage. The number of labeled objects is significantly lower compared to fully supervised methods. However, WS3Dv2's performance is comparable to fully supervised methods like PointPillars [15] and PointRCNN [29], which were trained on 3712 frames. The performance gap (3D IoU 0.7) is less than 6%. Additionally, the method allows for the incorporation of a human labeler in the loop, providing click-points to achieve an even lower performance gap.

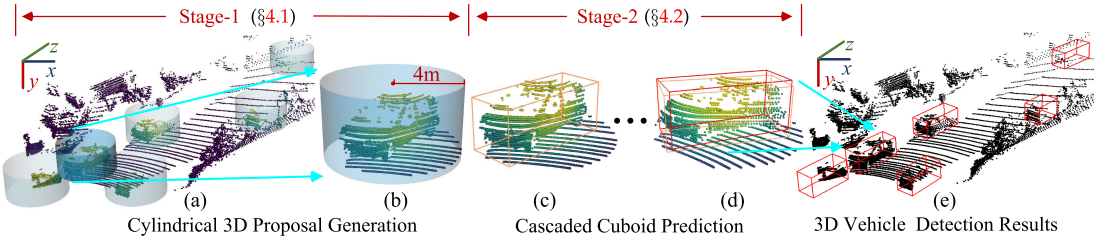


Figure 2.14. Pipeline of WS3Dv2 [6] described in Section 2.3.5. The figure is taken from [6].

2.3.6 MAP-gen

Multimodal attention point generator (MAP-gen) [30] tackles the sparsity problem of the LiDAR scans by introducing a method to generate 3D points based on an image, enriching the sparse LiDAR scans. The method can be divided into three stages: foreground segmentation, point cloud enrichment, and 3D box regression. The pipeline is shown in Figure 2.15.

Points inside the frustum area corresponding to each 2D bounding box are extracted, and the RGB values of the corresponding pixels are used to enhance the extracted LiDAR points. Segmentation of the point cloud is performed using PointNet [12], while the image segmentation is done by PSPNet [31]. During training, only points inside the ground truth 3D bounding box are used as the ground truth for training the PointNet. Similarly, only pixels with corresponding points inside the 3D ground truth bounding box are used for the training of PSPNet.

Pixels in the cropped and segmented image are sampled and enriched by features extracted from PSPNet. The resulting order-invariant sequence contains both segmented and sampled points, with each point containing a 2D location, 2D features from PSPNet, and 3D features from PointNet. The sampled points have unknown 3D features. Each sampled point is queried, and then 3D features are predicted with multimodal attention. An MLP is used to retrieve 3D location from the predicted 3D features. To train this stage, randomly chosen points within the 3D ground truth bounding box are masked and then recovered by multimodal attention.

PointNet is employed to extract global features from the enriched RGB point cloud, while global image features are extracted from PSPNet. An MLP is then used to generate a 3D bounding box by processing the concatenated RGB point cloud and image features.

The method is trained on the same data as WS3Dv2 [6] (500 frames, 534 precisely annotated objects). MAP-gen achieved state-of-the-art performance, reducing the performance gap (3D IoU 0.7) between fully and weakly supervised PointRCNN [29] to 2%. This achievement demonstrates comparable performance with significantly less training data.

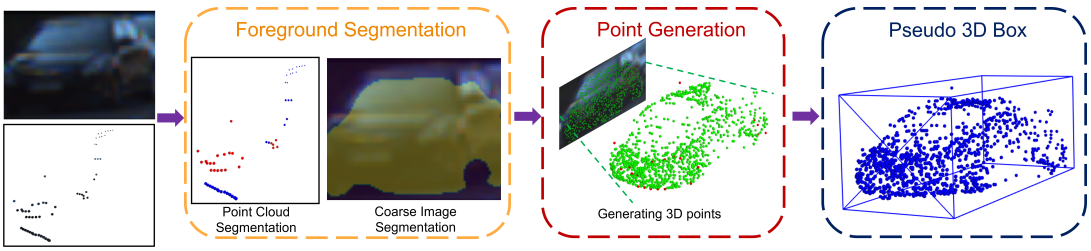


Figure 2.15. Pipeline of MAP-gen [30] described in Section 2.3.6. The figure is taken from [30].

2.3.7 M-trans

Multimodal transformer (M-trans) [32] shares the idea with MAP-gen [30], so they tackle the sparsity of the LiDAR scans. However, the proposed method uses a different architecture than MAP-gen. The pipeline is shown in Figure 2.16.

The method uses 2D ground truth bounding boxes to extract points from the LiDAR scan within the frustum area. Additionally, image patches of the object are extracted. A fixed number of target points is then sampled from the image, which are employed to

generate 3D points. Similar to vanilla transformers [33–34], the features of the object are encoded as embedding vectors. Each 2D and 3D position, along with the image patch, is encoded into features. Target points have unknown 3D position features. All these features are concatenated and fed into an MLP.

The features encoded by the MLP serve as input to the M-trans module, which consists of three different features: C-pts, points with known 3D locations (3D features), T-pts, target points without known 3D locations, and CLS, object-level information. All these features are used in multiple heads. The first head is responsible for point segmentation, using points inside a 3D ground truth bounding box as the foreground ground truth. The second head predicts the 3D point locations for target points, employing a similar strategy as MAP-gen, where some points inside a 3D ground truth bounding box are masked. The third head regresses 3D bounding boxes, using the 3D ground truth bounding boxes directly for training.

The method currently achieves state-of-the-art performance as an auto-labeler, with a performance gap (3D IoU 0.7) lower than 1% when compared to both weakly and fully supervised PointRCNN [29]. It uses the same number of training samples as in MAP-gen, so 500 frames with 534 precisely annotated objects. Thus showing, that it can achieve the same performance while using a small fraction of the labeled data.

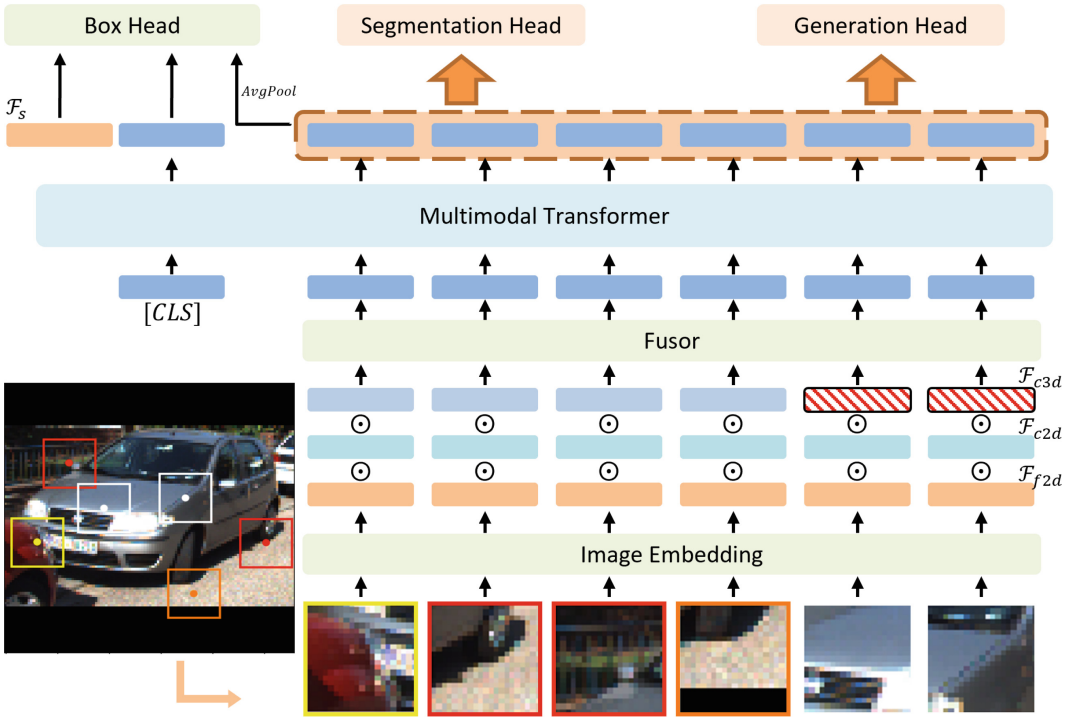


Figure 2.16. Pipeline of M-trans [32] described in Section 2.3.7. The figure is taken from [32].

Chapter 3
Foreground segmentation

This chapter starts with a description of our proposed method. Here the focus is on the process of foreground segmentation of LiDAR points. Our primary goal is to accurately extract and segment LiDAR point clouds that correspond to individual objects, with a focus on cars, in a given scene.

Initially, as outlined in Section 3.1, we utilize the 2D detector, Mask-RCNN [23], for the detection of cars in the image. Following this, in Section 3.2, we focus on how we employ binary masks from the 2D detector to extract points within the frustum area of these masks. Furthermore, Section 3.3 discusses additional steps of postprocessing, including the filtering of points and estimation of their location.

3.1 Instance segmentation

We employed the Detectron2 framework [35], which uses Mask-RCNN [23] for instance segmentation. Detectron2 is known for its high-performing models, excellent documentation, and user-friendly API. It provides access to pre-trained model weights, trained on various datasets like MS-COCO [7] and ImageNet [22].

We have selected *RegNetY* [36] as a backbone for Detectron2 for its optimal balance of speed, memory efficiency, and performance. The model weights were trained using the Simple Copy-Paste Data Augmentation [37]. We have evaluated multiple backbones, trained on different datasets, on the 2D object detection KITTI dataset, and the *RegNetY* trained on MS-COCO proved itself as a sufficient choice for our task. More complex architectures, like ViTDet [38], showed only marginal performance improvements in vehicle detection (cars, vans, trucks) when evaluated on the KITTI dataset. However, the KITTI dataset's use of amodal 2D bounding boxes limits the relevance of this comparison for occluded cars. The evaluation of *RegNetY* is shown in table 3.1.

Difficulty	Easy	Moderate	Hard
Average precision [%]	79.03	71.69	56.35

Table 3.1. Evaluation of the *RegNetY* on the 2D object detection KITTI dataset [3].

The procedure begins with creating the model and loading the configuration, followed by applying the weights. The model operates on either CPU or GPU. Processing one image takes roughly 72 ms, consuming about 1.5 GB of memory per sample in the batch. For efficiency, we process 20 images per batch on an Nvidia A100 GPU.

The KITTI object detection dataset provides one image per camera, in total four images. As detailed in Subsection 2.1.1, all cameras face forward, but we specifically utilize the camera with index two. This camera is an RGB camera positioned almost in the middle of the roof. Once more, we want to emphasize that the KITTI dataset labels only objects visible in camera 2 and with regards to this camera.

Let $L_i \in \mathbb{Z}, 0 \leq L_i \leq 80$ denote object class label, $S_i \in \mathbb{R}, 0.0 \leq S_i \leq 1.0$ denote score, $BB_i \in \mathbb{Z}^4$ denote 2D bounding box, and $M_i \in 0, 1^{H \times W}, M_i \in \{0, 1\}$ denote binary mask for each detected object in the image. A bounding box is decoded as a pair of two points in the pixel coordinates, H and W denote the image's height and width. Outputs are illustrated in Figure 3.1.

Figure 3.1. Example of using Detectron2 [35] framework as instance segmentation tool on KITTI [3] dataset.

We first eliminate all detections with a score S_i below 0.7, an empirically set threshold that effectively reduces false positives while retaining true positives. Then, we filter out detections not classified as cars. We have to deal with non-overlapping object class labels, as MS-COCO labels vehicles as cars, trucks, motorcycles, and buses, on the other hand, KITTI labels vehicles as cars, vans, and trucks. Vans in the scene are usually detected as cars (as it is correct in MS-COCO), creating a false positive in the KITTI labels. The problems with the ambiguities in the KITTI dataset are shown in Figures 3.2 and 3.3.

It is worth noting, that we experimented with the Segment Anything (SAM) [39] model for fine-tuning the masks M initially generated by Detectron2. We encountered a challenge because SAM often segmented specific parts of cars, such as windows, tires, or side mirrors, rather than the entire car. To mitigate this problem, the user in the SAM API can specify a bounding box around the object intended for segmentation. Each mask M_i with a corresponding bounding box BB_i was then with SAM to create a new improved mask. However, the performance difference between using masks from Detectron2 and those refined with SAM was negligible. This outcome indicates, that the quality of the segmentation is not the bottleneck of our method.

Figure 3.2. Examples of the ambiguity caused by the non-overlapping classes in KITTI [3] and MS-COCO [7] dataset. Red bounding boxes denote KITTI ground truth, and blue bounding boxes denote Mask-RCNN [23] output. The upper image shows a van being detected by Mask-RCNN as a car and also it shows the problem with overlapping cars, caused by the amodal KITTI ground truth. The lower image shows an object in the lower left corner, which we believe is a car, however, in the KITTI dataset it is labeled as a van.

Figure 3.3. Examples of the ambiguity caused by the non-consistent labeling in KITTI [3]. Red bounding boxes denote KITTI ground truth, and blue bounding boxes denote Mask-RCNN [23] output. The upper image shows labeled cars (marked by red arrows), that are less visible than the car in the lower image (marked by red arrow), however, the car in the lower image is not labeled.

3.2 Point extraction

At this stage, having successfully detected cars in the images and obtained their corresponding 2D binary masks, our next step is to utilize these masks for extracting points that belong to the detected objects (cars). To achieve this, it is essential first to project the LiDAR scan data into the image coordinate frame.

Let $P \in \mathbb{R}^{4\times n}$ be a raw point cloud, where n represents the number of points in the LiDAR scan. Each point in this point cloud has four values (x, y, z, α), where x, y, z denotes the location in the Euclidean space, while α denotes a point's reflectance. In our process, since reflectance α is not utilized, we assign it a constant value of 1 to all points. This adjustment effectively transforms the point cloud P into homogenous coordinates.

The subsequent step involves transforming the point cloud P from the LiDAR coordinate frame to the camera 0 coordinate frame. This is achieved using the transformation matrix $T_l^{\text{c0}} \in \mathbb{R}^{4\times4}$. Given that the camera has a limited field of view (FOV), we remove all points, that are behind the camera, as they obviously cannot be seen. Therefore, we obtain a filtered point cloud $P_f = \{p \in \mathbb{R}^4 \mid p \subset P,\ p(z) \geq 0\}$.

After filtering, the point cloud P_f is rectified using the matrix $R \in \mathbb{R}^{4\times4}$ as the projection matrices are defined for rectified images. Next, the rectified point cloud is transformed from the camera 0 coordinate frame to the camera 2 coordinate frame, employing the transformation matrix $T_{\text{c0}}^{\text{c2}} \in \mathbb{R}^{4\times4}$. The final step involves projecting the rectified point cloud P_f from this rectified camera 2 coordinate frame into the image coordinate frame. This projection is done using the matrix $K_{\text{rc2}}^i \in \mathbb{R}^{3\times4}$, as it describes the camera's intrinsics using the pinhole camera model. The projection matrix is structured as follows:

$$K_{\text{rc2}}^i = \begin{pmatrix} f_u & 0 & c_u & -f_u b_x \\ 0 & f_v & c_v & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1}$$

Where f are the focal lengths, c are coordinates of the principal point, and b_x is the baseline. Ignoring the point filtering, the transformation matrix from the LiDAR coordinate frame to the image coordinate frame can be computed as follows:

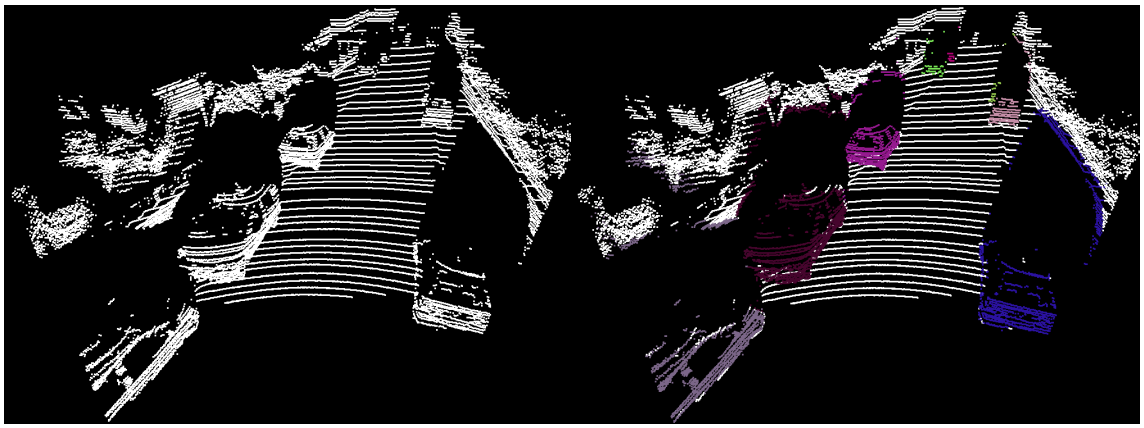
$$K_l^i = K_{\text{rc2}}^i T_{\text{c0}}^{\text{c2}} R T_l^{\text{c0}} \tag{2}$$

Utilizing this transformation matrix, we can project a point $\mathbf{X} = (x,\ y,\ z,\ 1)$ from the LiDAR coordinate frame to a corresponding point $\mathbf{Y} = (u/z,\ v/z,\ 1)$ in the image coordinate frame. Here, x,y, and z represent the point's location in Euclidean space, while u and v correspond to the pixel location in the image coordinate frame. This projection effectively transforms the three-dimensional spatial coordinates into two-dimensional pixel coordinates, aligning the LiDAR data with the image captured by the camera.

$$Y = K_l^i X \quad \rightarrow \quad P_e = K_l^i P \tag{3}$$

We enrich the point cloud P_f by appending u, v values to each point, so each point has a corresponding location in the image. Thus each point has known 3D and 2D coordinates. Considering that only objects within the FOV of camera 2 are labeled, we filter out points, where u or v exceed the image's width or height. This filtering provides us an enriched and filtered point cloud $P_e \in \mathbb{R}^{6\times n}$, where n denotes the number of points.

With the given enriched point cloud P_e we can perform the instance segmentation leveraging the provided binary masks M provided by Detectron2. However, given that our implementation is in Python, iterating over all points in P_e with a simple loop is impractical due to poor Python for loop performance. Instead, we employ NumPy slicing to significantly speed up the 3D instance segmentation. For each mask M_i, we extract all points where the mask M_i equals 1 at pixel coordinates u, v determined by the point. Thus, we get object point cloud $F_i \subset P_e$ for each mask M_i, illustrated in Figure 3.4.



a) b)

Figure 3.4. Example of the foreground segmentation. (a) raw point cloud (LiDAR scan) cropped to camera FOV, (b) foreground segmentation performed on all objects visible in the camera. Each distinct color represents a different object instance.

3.3 Postprocessing

For each object point cloud F_i our goal is to estimate its approximate 3D location. However, the point cloud F_i often contains many outliers due to the imperfect masks M generated by Detectron2 [35] or see-through surfaces such as windows. As a result, simply using a mean of the points in F_i doesn't give good results. Instead, we employ the median of all points in F_i, which has proven to be a more robust estimator in handling these outliers.

To further enhance the robustness of our estimator, we can shrink the mask M_i. This is based on our belief that the central area of the mask is likely to contain fewer outliers compared to the edges. For this purpose, we employ binary dilatation from the SciPy package to shrink the mask.

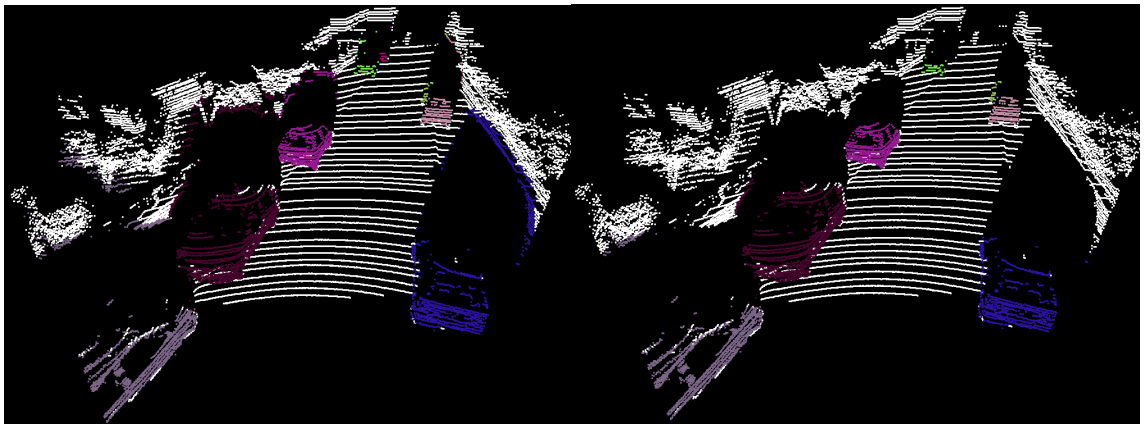
However, a crucial step is to determine the number of iterations k_i, needed for the mask M_i shrinking. It's important that k_i is always greater than zero to ensure that the mask is indeed shrunk, but at the same time, it also must not be so large, otherwise the mask will disappear. We have empirically found and tested an estimator for the k_i for each mask M_i. This calculation is done as follows:

$$k_i = \text{int}(2 + (\sqrt{\sum M_i})/10) \tag{4}$$

As the mask M_i is a binary mask, the sum of all elements corresponds to the total number of foreground pixels. Since the area shrinks in a quadratic manner, we need to consider the square root of the total number of foreground pixels in the mask M_i. Here, *int* represents the operation of casting values to the integer type. Utilizing the

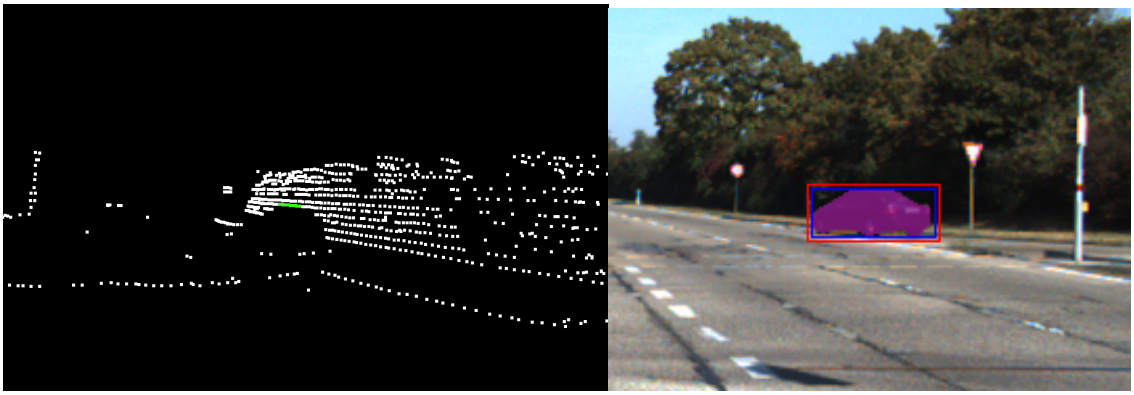
more robust location estimator is especially useful in tracking, described in the following Chapter 4.

To eliminate obvious outliers, we remove all points whose distance from the estimated location exceeds 4 meters. This simple yet effective filtering is demonstrated in Figure 3.5. However, this process of location estimation and further filtering can fail as shown in Figure 3.6.



a) b)

Figure 3.5. Example of the simple filtering of the segmented points. (a) raw segmented points, (b) filtered segmented points.



a) b)

Figure 3.6. Example of an unsuccessful segmentation of a moving car. The segmentation failed due to a low ratio of inliers to outliers. The imbalance led to the location estimation being placed in the background, and the following filtering wrongly removed all the foreground points. (a) Green segmented points, clearly being in the background (in this picture trees), (b) correct segmentation of a car in the corresponding image.

Chapter 4
Temporal consistency

In this chapter, our focus is on leveraging temporal consistency. We begin in Section 4.1 by describing the issue of ambiguity in sparse point clouds. Next, in Section 4.2, we focus on how can we utilize calibration data from the KITTI dataset [3] to get frame-to-frame transformations.

Subsequently, Section 4.3 details our method for tracking cars across multiple adjacent frames, a key aspect of exploiting temporal consistency. In Section 4.4, we explore the aggregation of points from multiple frames for standing cars. Additionally, Section 4.5 focuses on estimating the yaw of moving cars based on their trajectories.

The chapter concludes with Section 4.6, where we discuss the challenges posed by inaccuracies in the localization provided by calibration data. We describe how these challenges are addressed using the Iterative Closest Points (ICP) algorithm [8], providing a fine refinement of the frame-to-frame transformations.

4.1 Ambiguity challenge

In the previous Chapter 3, we discussed foreground segmentation of point clouds corresponding with objects (specifically cars) within a scene. A notable challenge with LiDAR scans is their sparsity, which increases with distance. For instance, an object a few meters away from the ego-vehicle may have thousands of corresponding points in the LiDAR scan. However, an object approximately 40 meters away might only have tens of points. This significant reduction in point density at greater distances poses a challenge for the fitting process. When attempting to fit a generic car mesh to such a sparse point cloud, multiple solutions are proposed, leading to ambiguity. This issue of ambiguity is shown in Figure 4.1.

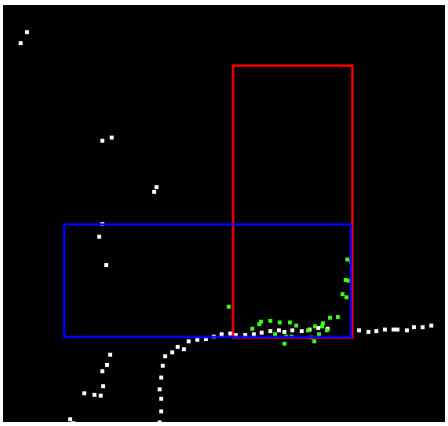


Figure 4.1. Example of the ambiguity fitting challenge in the Bird's Eye View (BEV). Green points correspond to a car, and red and blue bounding boxes propose a solution, however, it is not obvious which one is the correct one.

To address the ambiguity defined by the sparsity of the LiDAR scans, our strategy involves aggregating a larger number of points of the car across multiple adjacent frames or estimating the yaw of the car from its trajectory. Given that our pre-processing is done offline, which gives us access to future frames, we can utilize this additional data. With the dataset's frame rate of 10 Hz, we can observe the car across multiple frames, significantly increasing point density or we can track the car and thus reducing ambiguity in the fitting process. This is further discussed in this chapter.

4.2 Frame-to-frame transformations

The KITTI dataset includes calibration data for each frame, which includes IMU (Inertial Measurement Unit) readings. These readings contain precise positional data in the world coordinate frame, given by the Global Navigation Satellite System (GNSS) and further enhanced with real-time kinematics. This combination can provide positional accuracy to within a few centimeters. For processing this data, we employ the Pykitti library [40], which provides frame-to-frame transformations. Pykitti processes all sequences in the dataset by taking the first frame of each sequence as the world frame. It then computes transformation matrices for each subsequent frame relative to this world frame.

In this process, we are focused on transforming any given frame within the sequence to a reference frame. For any i-th frame in the sequence, our first step is to transform frame i into the world frame. This is done using the transformation matrix $T_i^w \in \mathbb{R}^{4\times4}$. Subsequently, we transform from the world frame to the reference frame using another transformation matrix $T_w^r \in \mathbb{R}^{4\times4}$. The transformation matrix $T_i^r \in \mathbb{R}^{4\times4}$, which converts frame i to the reference frame, is calculated as follows:

$$T_i^r = T_w^r T_i^w = (T_r^w)^{-1} T_i^w \tag{1}$$

However, it's important to note that this transformation is defined within the IMU coordinate space. Our entire process, on the other hand, is done with all LiDAR scans in the camera 2 coordinate space. So, we cannot apply this transformation directly to LiDAR scans. First, we transform from the camera 2 coordinate space to the IMU coordinate space using the transformation matrix $T_{\text{c2}}^{\text{imu}} \in \mathbb{R}^{4\times4}$. Following this, we do the frame-to-frame transformation using T_i^r. Finally, we transform back to the camera 2 coordinates using the transformation matrix $T_{\text{imu}}^{\text{c2}} \in \mathbb{R}^{4\times4}$. Note, that only $T_{\text{imu}}^{\text{c2}}$ is defined in the calibration files. The final transformation matrix operating in the camera 2 coordinate frame is calculated as follows:

$$(T_i^r)_{\text{c2}} = T_{\text{imu}}^{\text{c2}} T_i^r T_{\text{c2}}^{\text{imu}} = T_{\text{imu}}^{\text{c2}} T_w^r T_i^w T_{\text{c2}}^{\text{imu}} = T_{\text{imu}}^{\text{c2}} (T_r^w)^{-1} T_i^w (T_{\text{imu}}^{\text{c2}})^{-1} \tag{2}$$

4.3 Car tracking

To effectively aggregate additional data from adjacent frames, as previously mentioned, we exploit temporal consistency. This involves tracking each vehicle across all frames, to ensure that multiple instances of cars are correctly distinguished. As it is crucial to avoid mixing data of different instances. Additionally, we need to determine whether each instance represents a standing or moving car. This distinction is crucial as it allows us to gather different types of data depending on the car's state.

We set a fixed number of frames to be processed both before and after the reference frame, which is the sample from the training set. Empirically, we determined that 30 frames on both sides of the reference frame (totaling 60 frames) provide sufficient coverage. This equates to 3 seconds before and after the reference frame, considering the dataset's sampling rate of 10 Hz

The initial step involves processing all images corresponding to frames using Mask-RCNN [23]. To speed up the computation, we set the batch size to 20. For each frame and its corresponding image, we extract the spatial locations of cars $L \in \mathbb{R}^{3\times n}$ using the robust location estimator described in Section 3.3. For each car location $L_i \in \mathbb{R}^3$, the corresponding segmented point clouds F_i and binary masks M_i are also extracted. We omit any detected cars with fewer than two segmented points.

The spatial location L_i of each car is then transformed into the reference coordinate frame using the transformation matrix T_i^r, as detailed in Section 4.2. All transformed locations in the reference frame behind the ego-vehicle are removed. Additionally, the segmented point clouds F are also transformed into the reference coordinate frame.

For each frame i, we now have n spatial car locations L_n, segmented point clouds F_n, both transformed into the reference coordinate frame, and corresponding binary mask M_n. However, at this stage, the correspondences between car instances across different frames are not yet established.

To resolve this, we implement a tracking algorithm that we have developed. This algorithm's primary goal is to track the cars over the sequence of frames, to establish the necessary correspondences. The output of this algorithm is an array, where each entry represents a unique car instance identified over some subset of the frame sequence. Each car instance contains k spatial locations L_k, segmented point clouds F_k, binary masks M_k, and additional information I_k (such as bounding box and score). Here, k represents the number of frames in which the car was observed and successfully tracked. The pseudo-code for our tracking algorithm is presented on page 27.

In cases where a car instance has only been seen once previously, we simply use the last known location as the estimated location. However, when we have access to at least two locations for a car instance from the past, we implement a motion model to predict its current location. The motion model we employ is outlined as follows:

$$L_i^t = L_i^{t-1} + (L_i^{t-1} - L_i^{t-2})$$

Where i denotes the i-th tracked car and t denotes the frame index. This motion model is especially important for tracking moving cars.

In the matching process, we aim to pair each car instance i, with a given location, in the current frame with the closest corresponding tracked car z, with a predicted location, in the Euclidean space. To make our tracker robust, we require two key conditions to be satisfied: Firstly, the tracked car z must be the nearest to the current car i in terms of an Euclidean distance. Secondly, the current car i must also be the nearest to the tracked car z.

Additionally, even when both conditions are satisfied, we introduce a further constraint: the distance between the matched cars must be less than 5 meters. This constraint is to prevent unrealistic matches that could occur. If all of these conditions are satisfied, then we take them as a match and we can establish the correspondence.

It's worth noting a limitation in our tracking method: it cannot recover from a lost object. This means that if a vehicle disappears in a frame and then reappears in subsequent frames, it will be treated as two distinct instances. However, the occurrence of this situation is very rare.

Algorithm 4.1 Car Tracking Algorithm

```
Input: Car locations for each frame in the reference coordinate frame
Output: Car instances with locations over the frames in the reference co-
ordinate frame

begin
tracked = []
final = []
for i in frames:
    current = car locations in frame i
    if number of current > 0:
        for each tracked estimate location
        for k in current:
            if number of tracked > 0:
                perform matching of k-th current with tracked
                if matched:
                    add location of k-th current to matched tracked
                else:
                    move k-th current to tracked
            else:
                move all current to tracked cars

        move all unmatched tracked to final
    else:
        move all tracked to final
move all tracked to final
return final
end
```

The next step in our process involves filtering out all instances of tracked cars that are not visible in the reference frame. This step is needed because such instances are just not labeled in the KITTI dataset, otherwise, they will pose as false positives. After this filtering, we categorize the remaining tracked cars into two groups: standing and moving. This categorization is based on the distance the vehicle has traveled: if the total path length of a tracked vehicle exceeds 5 meters, we classify it as a moving car. If the distance is less than this threshold, the car is considered as standing.

4.4 Standing cars

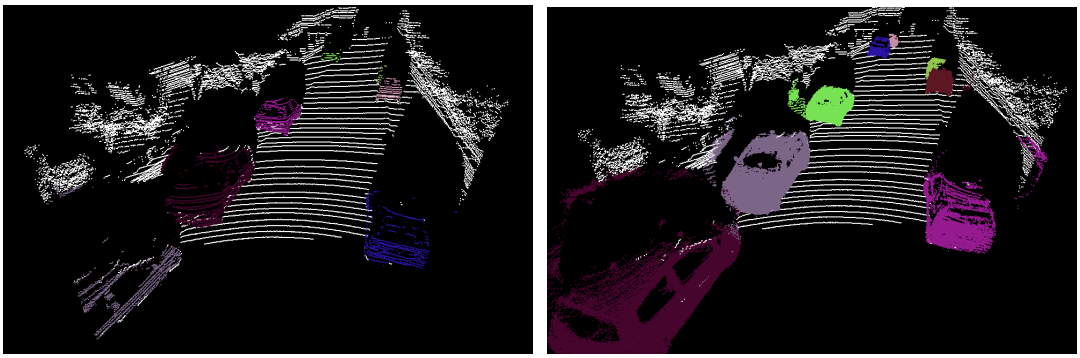
For each instance of a standing tracked car i, observed across n frames, we have the spatial location $L_{i,n}$, segmented point cloud $F_{i,n}$, binary mask $M_{i,n}$ and information $I_{i,n}$. Both the spatial locations and segmented point clouds are in the reference coordinate frame. By compensating for the movement of the ego-vehicle using the known transformation matrices T_i^r we ensure that the segmented point cloud remains in a consistent location across all frames. This is enabled by the fact, that the car is standing. This consistency allows us to concatenate all individual point clouds $F_{i,n}$ from different frames into a single aggregated point cloud $\hat{F}_i$.

This process enables us to give a much denser point cloud $\hat{F}_i$, which helps mitigate the ambiguity shown in Section 4.1. It improves the performance of our method signif-

icantly, especially for the Hard examples in the KITTI dataset, which are represented by cars in the distance. We can virtually move closer to those hard examples, extract the points, and then move back.

$$\hat{F}_i = \sum_{k=1}^{n} F_{i,n} \tag{3}$$

Where i represents the index of the instance, n denotes the number of frames in which instance i has been tracked. The sum denotes concatenation. The point aggregation is shown in Figure 4.2.



a) b)

Figure 4.2. Example of the point aggregation exploiting the temporal consistency for standing cars. (a) Segmentation is employed only in the reference frame, (b) Segmentation is employed in 30 scans before and after the reference frame, and then the segmented points are aggregated together. Downsampling is not used for visualization purposes.

As the number of points in each F_i can be excessively large, to keep the computational requirements at a reasonable level, we implement two downsampling methods: random and voxel. We achieve the best results by applying these methods independently to F_i and then concatenating the resulting downsampled point clouds. The reasoning behind this approach is as follows.

Random downsampling, in theory, uniformly reduces the point density across all areas. We assume that areas with a higher density of points are less likely to be outliers and therefore should be given preference. Voxel downsampling, on the other hand, retains areas that are observed at least once, thus more effectively preserving the overall shape of the car, at the cost of including outliers. By combining these two downsampling methods, we get a point cloud that not only preserves the general shape of the car but also keeps points from locations with a high likelihood of being inliers.

The random downsampling process is written using NumPy slicing, while voxel downsampling is performed with the PyntCloud library, with a set voxel size of 0.15 meters. The results of this downsampling process are illustrated in Figure 4.3.

It's important to highlight a potential problem in our classification of standing cars. Consider a scenario where a car is waiting at a red light and then begins to accelerate slowly. If this slow movement does not surpass our 5-meter threshold, the car may still be classified as standing. However, because the car is not standing, the aggregated point cloud becomes blurred due to the small movement of the car. Consequently, this leads to wrong results in the fitting process. On the other hand, these occasions are very rare, so they do not pose a significant problem for our method. This problem is shown in Figure 4.4.

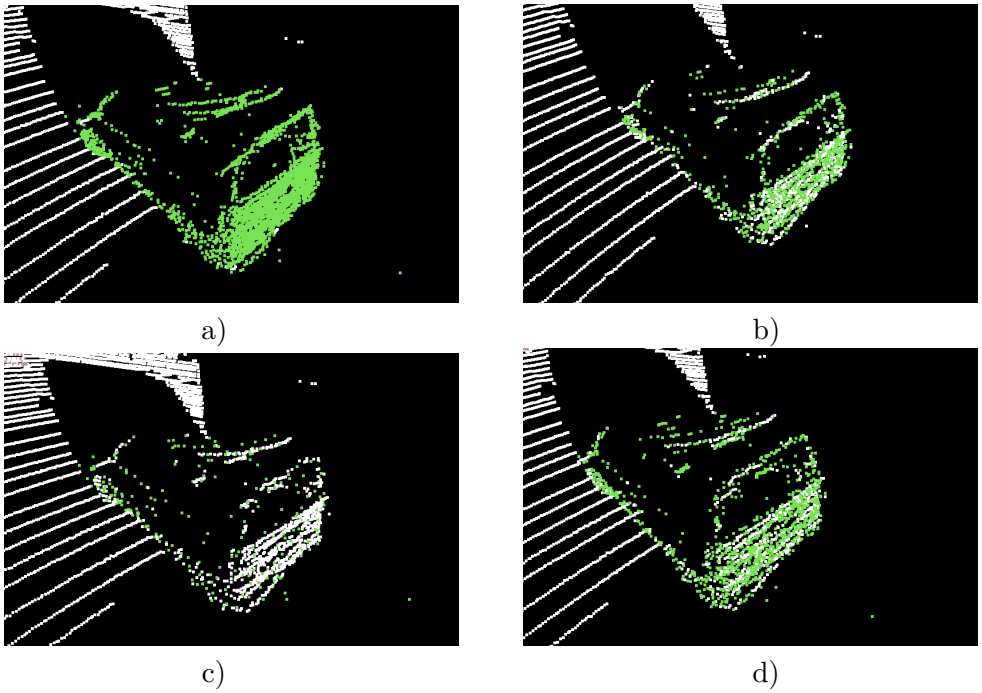


Figure 4.3. Example of the downsampling of the segmented cars. (a) raw segmented car, (b) random downsampling to 1000 points, (c) voxel downsampling with voxel size 0.15m, (d) concatenation of random and voxel downsampled points.

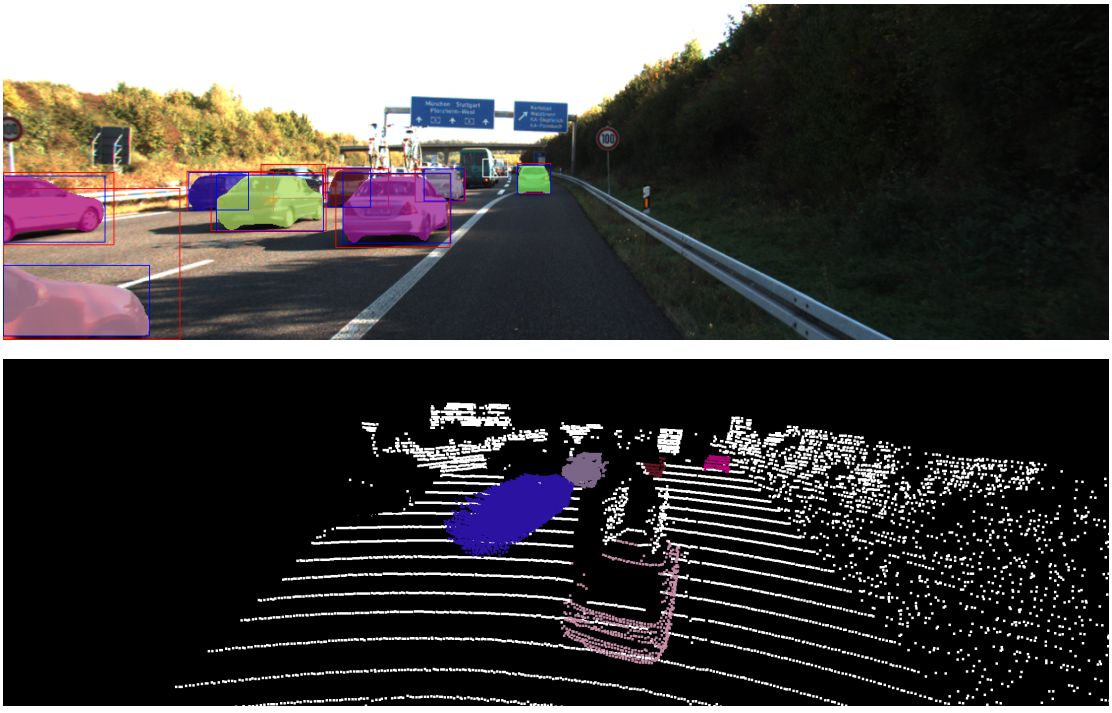


Figure 4.4. Example of the wrong classification of a standing car. In the upper image, the cars are moving slowly in a traffic jam, and the car with a green mask in the left part of the image is classified as standing. In the lower image, we can see, that aggregated points over the frames are blurred of the blue car, as the car is moving.

4.5 Moving cars

For each instance of a moving tracked car i, observed across n frames, we have the spatial location $L_{i,n}$, segmented point cloud $F_{i,n}$, binary mask $M_{i,n}$ and information $I_{i,n}$. The process of aggregating point clouds for the moving cars are significantly challenging problem, as the compensation for the movement of the ego-vehicle is not sufficient. A highly accurate tracking of the car's movement in space is required to establish the precise rigid transformation of the car between frames.

We experimented with using the Iterative Closest Point (ICP) method to align the adjacent pair of segmented point clouds $F_{i,n}$ and $F_{i,n-1}$ frame by frame, trying to find the perfect fit. However, this approach often failed, mainly because the segmented point clouds can differ significantly between frames.

Instead, we have discovered that accurately estimating the yaw (rotation around the vertical axis) of the car significantly simplifies the fitting process. This estimation of yaw also effectively addresses the ambiguity challenge previously described in Section 4.1.

As we have successfully tracked the given instance, a reliable estimate of its trajectory is available. This makes it easy to estimate the car's yaw – its rotation around the vertical axis – from this trajectory. However, for a robust estimation of yaw, we require that the instance must have been tracked for a minimum of three frames. The trajectories used for the yaw estimation are shown in Figure 4.5.

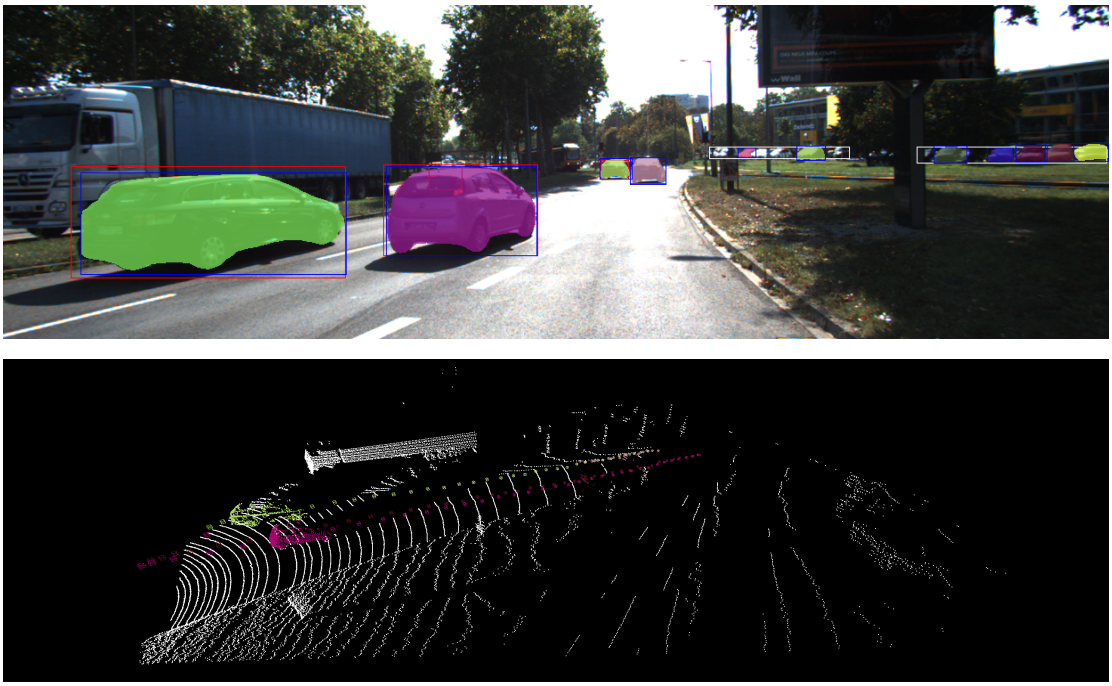


Figure 4.5. Example of the tracking of the moving cars. The upper image shows the scene, while the lower case shows the trajectories of the cars from the past and the future.

To calculate the yaw angle, we consider each pair of spatial locations $L_{i,k}$ and $L_{i,r}$ where r denotes the index of the reference frame and k represents a set of indices such that $k = \{k \in \mathbb{Z} \mid k = 1...n,\ k \neq r\}$, with n being the total number of locations. We select five pairs of locations both before and after the reference frame to compute the yaw. The computation of the yaw angle for each pair of locations is based on the following equation:

$$\theta = \text{arctan2}\Big(L_{i,n}(z) - L_{i,n-1}(z), L_{i,n}(x) - L_{i,n-1}(x)\Big) \tag{4}$$

We employ NumPy implementation of arctan2. It's important to clarify, that our method operates within the camera 2 coordinate frame. Consequently, when viewing from the bird's eye view (BEV), the y axis vanishes, and only x and z are present. This is a key distinction from operating in the LiDAR coordinate space, where the z axis would be absent in the BEV.

To ensure the robustness of our yaw estimation, we require that the distance between points in each pair must be at least 3 meters. After computing the yaw estimations for all valid pairs, we compute the median value out of these estimations as our final yaw estimation. This approach of using the median helps in mitigating the impact of any outliers in the estimations.

4.6 Iterative closest point refinement

While working with the frame-to-frame transformations and subsequent aggregation of points from standing cars, we encountered instances where the aggregated point cloud, denoted as $\hat{F}_i$, appeared blurred. Initially, we thought that this blurring might be caused, by the effect, that the LiDAR scanner needs approximately 100 ms to do the complete scan. During this interval, any movement of the ego-vehicle could distort the captured scene. However, during further investigation, we realized that this was not the case.

The actual problem lies in the imprecise IMU. Despite the theoretical claim that this system can achieve precision up to a few centimeters, in practice, it does not consistently maintain this level of accuracy. Various factors can distort the localization such as a low number of satellites in view or signal reflection from buildings. Those factors lead to a less accurate position provided by the IMU, which in the end causes the blurring of the aggregated point cloud $\hat{F}_i$.

In order to achieve an optimal alignment of the frames, we decided to refine the transformation matrices T_i^r using the Iterative Closest Point (ICP) [8] algorithm, specifically its point-to-plane variant. We use the IMU frame-to-frame matrices as a prior estimate, providing a solid baseline for the alignment. We then apply the ICP algorithm to the LiDAR scans from the adjacent frames. This step enables us to finely refine the transformation matrices, which will reduce the blurring of the point cloud $\hat{F}_i$.

In the initial version of our approach, we tried to directly refine the transformation matrix T_i^r for each frame i, where i is defined as $i = \{i \in \mathbb{Z} \mid i = -n...0...n,\ i \neq r\}$, with n representing the number of frames before and after the reference frame. However, as the time difference between the two frames increased, the overlap of the LiDAR scans became small. This reduced overlap posed a significant challenge for the ICP algorithm, which then often failed and proposed even worse solutions than the IMU frame-to-frame transformations. The problem is shown in Figure 4.6.

In the second version, we approach the fine refinement process by focusing on each pair of adjacent frames. For this, we use the transformation matrix T_{i-1}^i, where i is defined as $i = \{i \in \mathbb{Z} \mid i = -n + 1...0...n\}$. Again, we used the IMU frame-to-frame matrices as a prior. As these frame pairs are directly adjacent, the differences between their LiDAR point clouds are negligible. This significantly increases the probability of ICP converging to a better solution. Having the refined transformation matrices for each pair of adjacent frames, we can compute the refined transformed matrix T_i^r as follows:

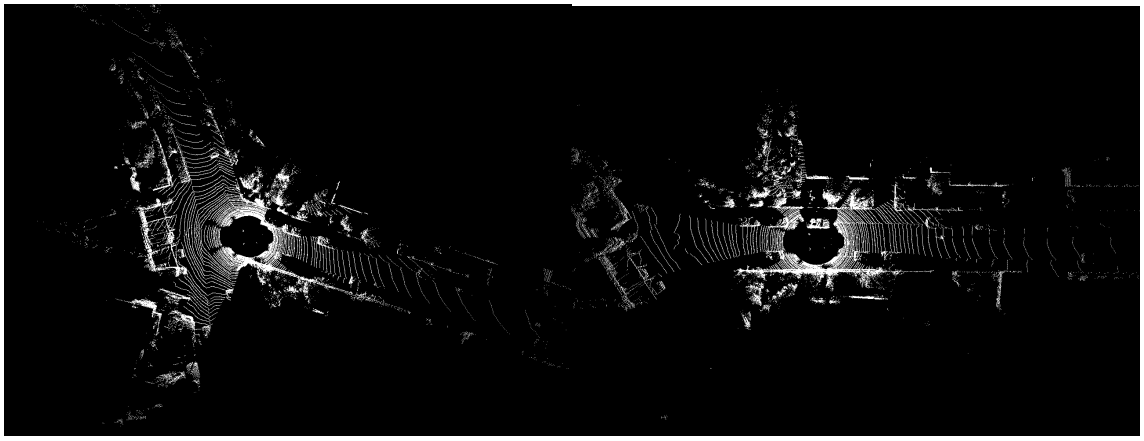
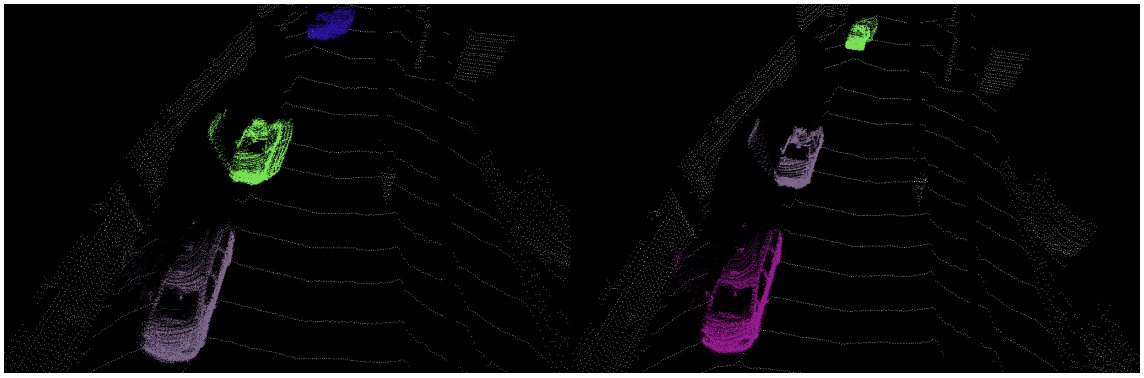


Figure 4.6. Example of two LiDAR scans are taken from the same sequence in the KITTI [3] dataset. The LiDAR scans are captured three seconds apart, which represents the maximum time we employ the tracking. As those LiDAR scans are different, aligning them poses a challenge.

$$T_i^r = \prod_{k=i+1}^{r} T_{k-1}^k \tag{5}$$

This second version proved itself to be robust and showed a significant performance boost for our method.

For the implementation of the ICP algorithm, we employ the version available in the Open3D Python library. The first step in our process estimates normals for each point cloud. We do this using a radius of 0.5 meters and setting a limit of a maximum of 30 neighbors for each point. Once the normals are estimated, we then execute the ICP algorithm, setting the maximum correspondence distance to 0.1 meters. The fine refinement by ICP is shown in Figure 4.7



a) b)

Figure 4.7. Example of the same frame from the KITTI [3] dataset. In both images, the points are aggregated from 30 frames before and after the reference scan. (a) uses the frame-to-frame transformation provided by the KITTI dataset. (b) uses the frame-to-frame transformation provided by the KITTI dataset as a prior and then Iterative Closest Point [8] for fine refinement.

There are some cases in the KITTI dataset, that the localization from IMU data is distorted, and ICP cannot recover from that. This situation is shown in Figure 4.8 and happens only in one captured sequence.

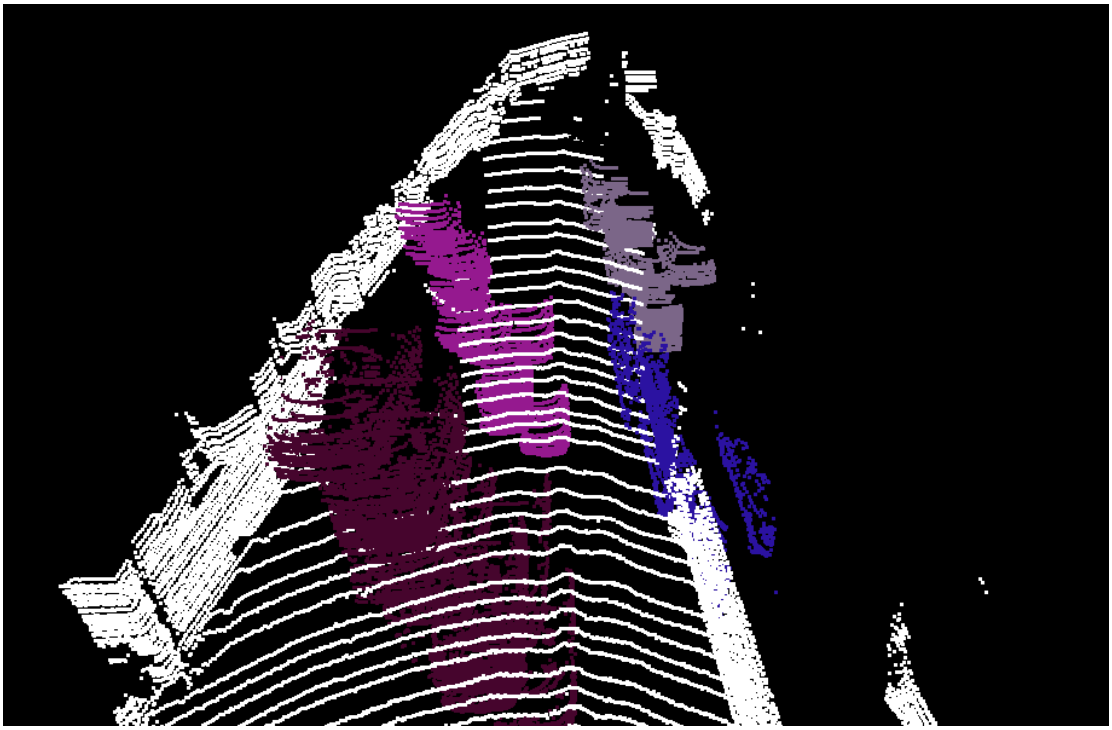


Figure 4.8. Example of a situation, where the ICP fails because the localization from IMU data from the KITTI [3] is distorted.

Chapter 5
Rigid shape model fitting

This chapter is dedicated to the process of fitting a generic shape mesh to segmented (aggregated) point clouds. We begin in Section 5.1, where we describe the meshes and their conversion process into point clouds. Following this, Section 5.2 discusses the use of raycasting on these generic shape meshes to produce raycasted point clouds.

In Section 5.3, the focus shifts to the fitting process specifically for standing cars. On the other hand, Section 5.4 discusses the process of the fitting process for moving cars. Section 5.5 then proceeds into how raycasted templates are employed within the fitting process. In Section 5.6, we explore various loss functions that are crucial to the fitting process.

The chapter concludes with Section 5.7, where we provide an idea behind the Histogram Yaw Estimation, particularly in the context of standing cars.

5.1 Meshes

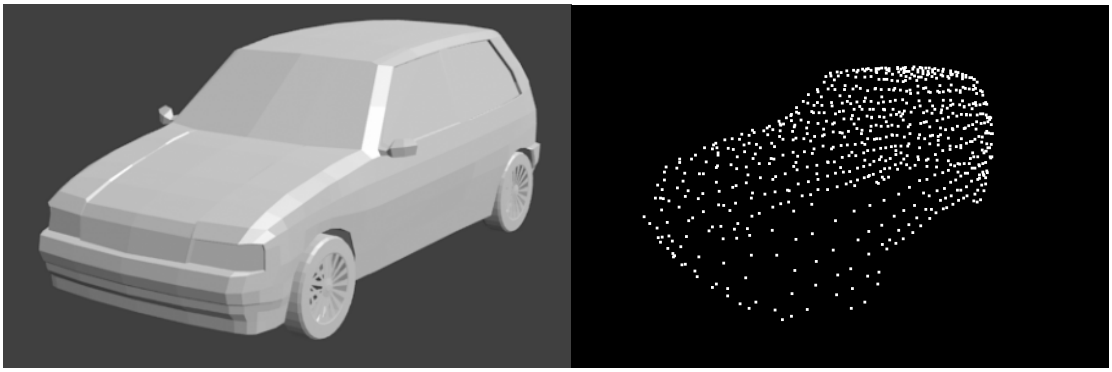
To effectively fit a generic shape mesh to various car types, it's crucial to choose a mesh that represents the common categories of cars. In real-world scenarios, most cars can be classified into a few categories, such as hatchbacks, sedans, SUVs, MPVs, and pickups. We aren't focused on the pickups, as we employ our method on the KITTI [3] dataset, which takes place in Germany, where the percentage of pickups among all cars in Germany is very low.

Initially, we decided to use a single mesh corresponding to a hatchback. This choice is reasonable as hatchbacks share similarities with SUVs and MPVs and also offer clear differentiation of the car's orientation. This is important because sedans can appear symmetrical, especially when rotated by 180 degrees around the y-axis (yaw). However, it is worth noting that the 3D objection detection evaluation script for the KITTI dataset does not differentiate the car's orientation. For our purposes, we have selected the Fiat Uno model as our hatchback representation, taken from [41]. The chosen mesh effectively represents the generic shape of a car and is shown in Figure 5.1.

The subsequent step in our method involves converting the mesh into a point cloud, which is needed for the fitting process. To achieve this, we randomly sample 1000 points from the mesh. Notably, we have excluded the bottom (floor) of the car from this sampling process, as these points are not visible and thus not useful for the fitting. The resulting sampled point cloud, denoted as $P_h \in \mathbb{R}^{3\times 1000}$, is shown in Figure 5.1.

Once P_h is generated, we load it into memory and proceed to shift it so the centre of the mesh is located at the origin of the coordinate frame. Following this alignment, we apply an upward shift to P_h. We have empirically determined that a shift of 0.2 meters is optimal for this particular mesh to achieve the best performance during the fitting process. This adjustment is necessary to compensate for a slight bias introduced by the estimator of spatial location.

After shifting P_h, our next step is to scale it to match real-life spatial dimensions. We use the average dimensions of cars in the KITTI dataset as a prior estimate for this

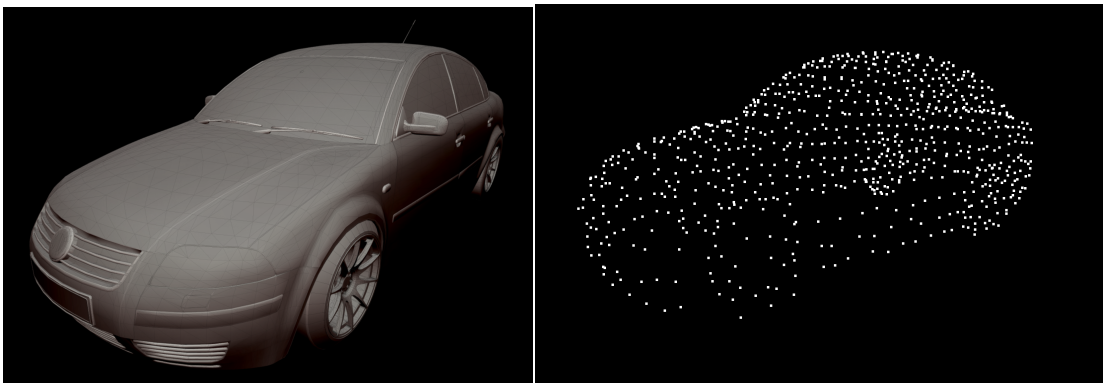


a) b)

Figure 5.1. (a) Mesh of a Fiat Uno, which is utilized as a template for the fitting process. The image is taken from [41]. (b) point cloud generated by 1000 randomly sampled points on the Fiat Uno mesh (a).

scaling. The prior estimate is based on the dataset, where the average car dimensions are 1.53 meters in width, 1.63 meters in height, and 3.88 meters in length.

As we have continued our research, especially in the development of the Scale Detector described in Chapter 6, it became apparent that adding another car type is necessary to increase the performance of our method. Specifically, we need to include a sedan model, as fitting a hatchback template to a sedan is not precise enough. To address this, we selected a mesh model of a Volkswagen Passat, taken from [42]. The mesh for this sedan, along with its corresponding sampled point cloud $P_s \in \mathbb{R}^{3\times1000}$, is shown in figure 5.2.



a) b)

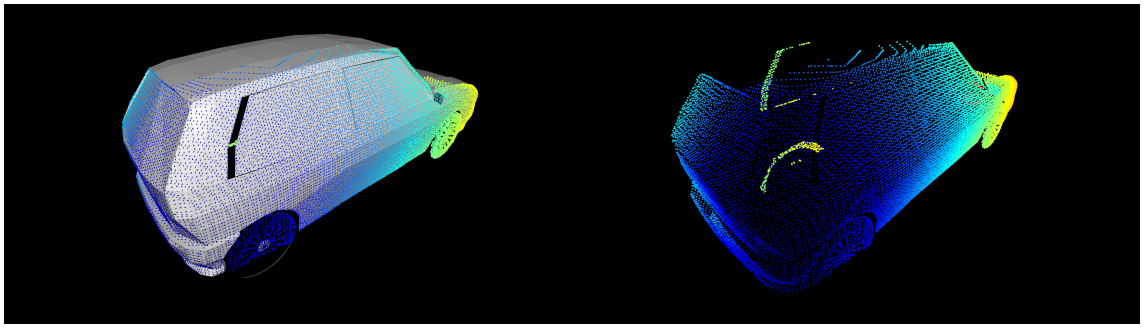
Figure 5.2. (a) Mesh of a Volkswagen Passat used as a second template for fitting. The image is taken from [42]. (b) point cloud consisting of 1000 randomly sampled points on the mesh (a).

5.2 Raycasting on meshes

In our fitting process, using the entire mesh of a car may not always be practical, as it's impossible to see the entire vehicle from a single viewpoint, as a significant part of the car is occluded by itself. The portion of the car that is visible to us is determined by the viewing angle, which is defined by the car's location and yaw.

To employ this strategy, we take our two selected mesh models — the Fiat Uno and the Volkswagen Passat — and create a corresponding point cloud $P_{\text{rh}} \in \mathbb{R}^{3\times n}$ or

$P_{\text{rs}} \in \mathbb{R}^{3\times n}$, where n denotes the number of points, to a given viewing angle. We achieve this by rotating each car model around the y-axis. For every one-degree increment in this rotation, we perform raycasting on the mesh. This process generates a partial point cloud P_{rh} or P_{rs} representing the portion of the mesh visible from that specific viewing angle. However, it's important to note that due to imperfections in the mesh models, which are not completely water-tight, this raycasting process can sometimes produce outliers. The results of this raycasting process, including potential outlier points, are illustrated in Figure 5.3.



a) b)

Figure 5.3. Example of the raycasting performed on the mesh. (a) Only the points, that are visible based on the viewing angle are sampled, (b) Point cloud resulting from the raycasting process on the mesh.

The raycasting is performed with Open3D [43] Python library. As the LiDAR point cloud generated by the raycasting is usually dense, we employ voxel downsampling with a voxel size of 0.025 metres, also using the Open3D Python library.

5.3 Fitting of standing cars

In this section, our focus is on fitting a generic car template to standing cars. At this point, we have the segmented aggregated point clouds $\hat{F}$, as outlined in Chapter 4, and point clouds P_h and P_s, which correspond to the hatchback and sedan car categories, described in Section 5.1. For standing cars, we primarily utilize $\hat{F}$, as there is no available estimate of the yaw angle for the standing vehicles. During our evaluation process, we have found that using both point clouds P_h and P_s simultaneously does not yield a better performance of our method. Therefore, in the fitting process, we employ only the hatchback template point cloud P_h.

For each i-th car represented by dense point cloud $\hat{F}_i$, the first step is to decide whether we have sufficient data to perform a reliable fitting. We exclude any i-th car with corresponding point cloud $\hat{F}_i$ that has fewer than 1000 points in the $\hat{F}_i$. The reasoning behind this threshold is that point clouds with fewer points often face the ambiguity challenge, as discussed in 4.1. It is important to note that at this stage, the point clouds $\hat{F}$ are not yet downsampled. As the point clouds are aggregated over many frames, they can have tens of thousands of points for each car.

The subsequent step involves downsampling the point cloud $\hat{F}_i$ using the combination of random and voxel downsampling methods we previously outlined in Section 4.4. After downsampling, our next task is to estimate the spatial location of the i-th car, which will serve as a prior for the fitting process. For this estimation, we use our robust spatial location estimator, described in Section 3.3. This estimator calculates the median of

all points, each axis separately, in the point cloud $\hat{F}_i$, providing a reliable estimate of the car's location.

The 3D detection problem for cars is generally defined by seven degrees of freedom (7-DOF): three for spatial location, three for the car's dimensions, and one for yaw. In this phase of the process, we simplify the problem to 4-DOF by setting the car's dimensions to the average dimensions derived from the KITTI dataset. Despite having a prior estimate for the spatial location, which theoretically reduces the optimization space, performing the optimization over a 4D space is still infeasible. The reason is that our optimization process, consisting of a brute-force approach, would require an impractically large number of iterations.

In the fitting process, we chose to exclude optimization along the y-axis (the vertical axis) based on two key considerations. First, our robust spatial location estimator reliably determines the y-axis location. Second, and more importantly, the y-axis estimated location remains unaffected by the yaw rotation, for which we currently have no prior estimation. Our optimization strategy is executed in two distinct stages: an initial coarse optimization and a subsequent fine optimization.

The coarse optimization phase optimizes over the parameters x, z, and θ, where θ represents the yaw angle of the car. We define a region $C_c \in \{-2, ..., 2\} \times \{-2, ..., 2\} \times \{0, ..., 360\}$ for the coarse optimization. The first and second dimensions denote the x and z ranges in metres. The third dimension denotes the range of θ in degrees. We sample the region with 20 steps for each dimension, resulting in x and z steps being 0.21 metres and the θ step being 19 degrees.

Within this region, we define a parameter set $Q = \{q \in \mathbb{R}^3 \mid q \in C_c\}$ and parameters $p \subset Q$, and considering the given estimated spatial location $e \in \mathbb{R}^3$ along with the average KITTI car dimensions as a prior, we can represent a car in 3D using a total of 7 parameters. These parameters consist of the spatial location, dimensions, and yaw of the car as follows:

$$x = e(x) + p(x) \quad y = e(y) \quad z = e(z) + p(z) \quad \theta = p(\theta) \quad l = 3.88 \quad h = 1.63 \quad w = 1.53 \tag{1}$$

Where l, h, w denotes the length, height, and width of the car in meters. The parameters x, y, z represent the car's spatial location, while θ denotes the yaw. To align the template point cloud P_h with these state parameters, P_h is shifted and rotated according to the parameters. It's important to note that P_h is already scaled to the average car dimensions, so further scaling is not required at this stage. We use the complete point cloud P_h, rather than its raycasted version, which is later reasoned in this chapter.

The final step in our coarse optimization process involves computing the loss for all parameters $p \subset Q$. We then select the parameters with the lowest loss value, p_{opt}, as our optimal solution. The specific loss functions used in this process are discussed later in the chapter. However, the relatively large step size for θ can lead to imperfect fitting. To refine this, we conduct fine fitting focused only on the yaw of the car.

We define a region $C_f \in \{0, ..., 360\}$ for the fine fitting, which is one-dimensional, as the only parameter we optimize over is the yaw. Thus, we can afford a finer step size of 1 degree with 360 samples. Within this region, we define a parameter set $Q_f = \{q_f \in \mathbb{R}^1 \mid q_f \subset C_f\}$, together with the previously determined optimal parameters p_{opt} and parameter $p_f \subset Q_f$, the estimated spatial location $e \in \mathbb{R}^3$, and the average KITTI car dimensions as a prior. We can represent a car in 3D with the same 7 parameters as follows:

$$x = p_{\text{opt}}(x) \quad y = e(y) \quad z = e(z) + p_{\text{opt}}(z) \quad \theta = p_f(\theta) \quad l = 3.88 \quad h = 1.63 \quad w = 1.53 \tag{2}$$

Where l, h, w denotes the length, height, and width of the car in meters, x, y, z represents the spatial location, and the θ denotes the yaw of the car. For each parameter p_f, we adjust the template point cloud P_h by shifting and rotating it according to the values in p_f and p_{opt}. After this transformation, we compute the loss for each parameter p_f.

We then select the optimal parameter, denoted as $p_{\text{fopt}} \subset Q_f$, which is the one having the lowest loss, following the same criterion used in the coarse fitting process. Although the fine fitting process improves the performance, it is mainly useful in correcting small errors in the yaw estimation. This limitation is a bottleneck in cases where the initial bounding box estimation is completely wrong, as shown in Figure 5.4. In such scenarios, the fine fitting process is not able to help at all.

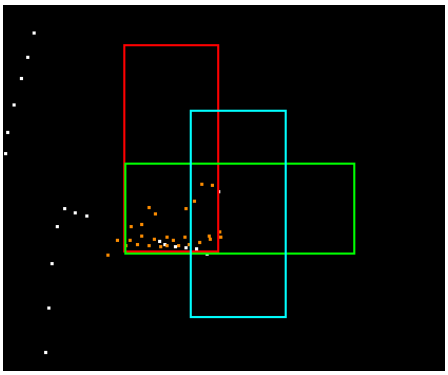
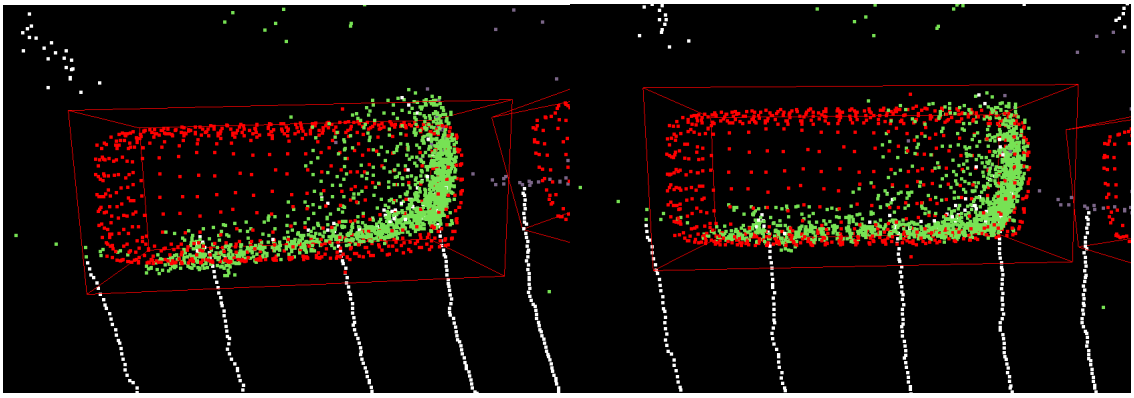


Figure 5.4. Example of fine fitting limitation, as it can't recover from completely wrong yaw prediction provided by the coarse fitting. Orange points represent the car, the red bounding box represents the ground truth, the green bounding box represents the output of the coarse fitting and the light blue bounding box represents how would the output of coarse fitting look if we set the correct yaw. This figure illustrates that as the fine fitting can only change yaw, it can't recover from the completely wrong coarse fitting output.

This fitting process effectively generates all the necessary parameters to define the 3D detection of a car, encoded in the form of a 3D bounding box. These 3D bounding boxes can then be exported into .txt files, functioning as KITTI [3] dataset labels. These labels can be either used for the evaluation of our method or as training data for a 3D detector. The improvement that fine fitting brings to the process is illustrated in Figure 5.5, where the results of coarse and fine fitting are compared.

To further illustrate the capability of our fitting process, we present examples of successful fittings for standing cars in Figure 5.6. Additionally, Figure 5.7 shows examples of instances where the fitting process encounters problems, highlighting the challenges and limitations of the fitting process.



a) b)

Figure 5.5. Example of the difference between the coarse and fine fitting. (a) A coarse fit to a car, (b) a fine fit to the same car. Green points are aggregated points corresponding to a car, red points correspond to a template point cloud P_h fitted to the standing cars, and red bounding boxes denote the 7-DOF optimal parameters.

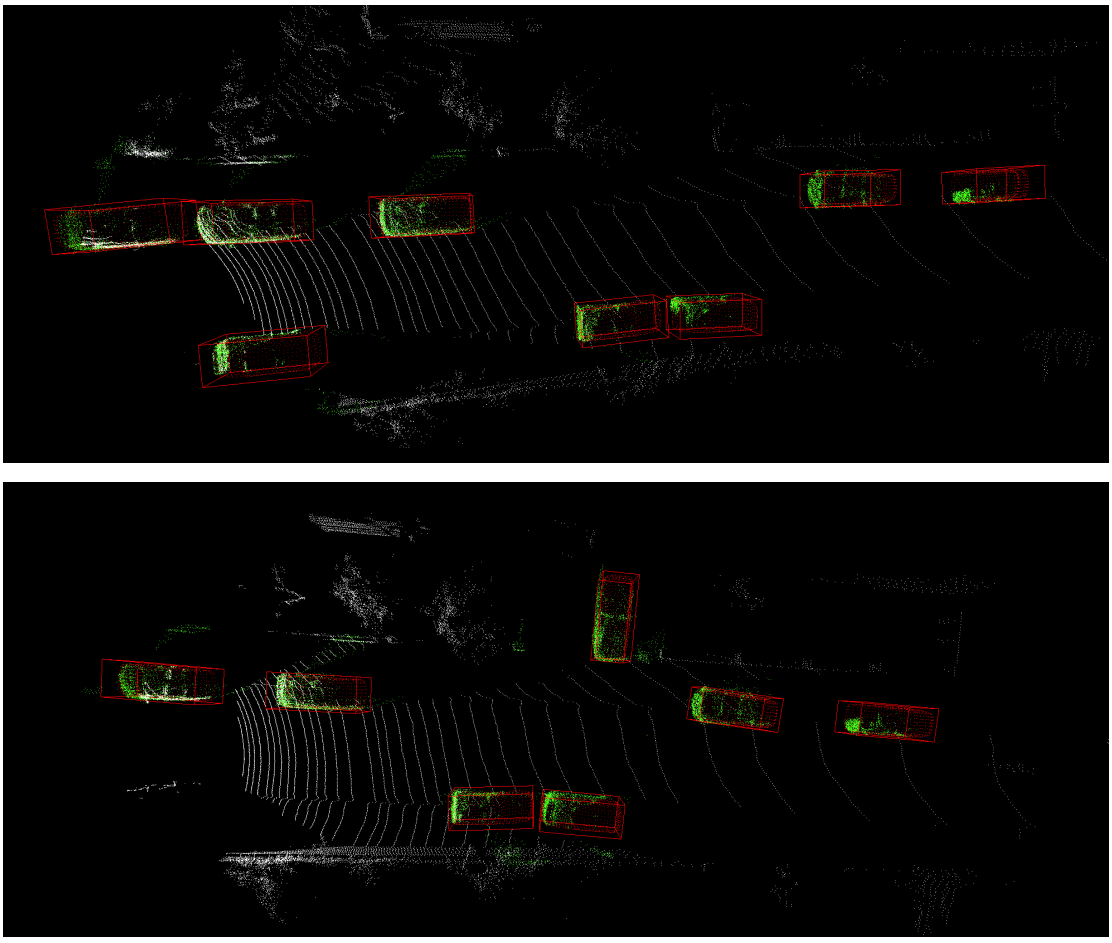
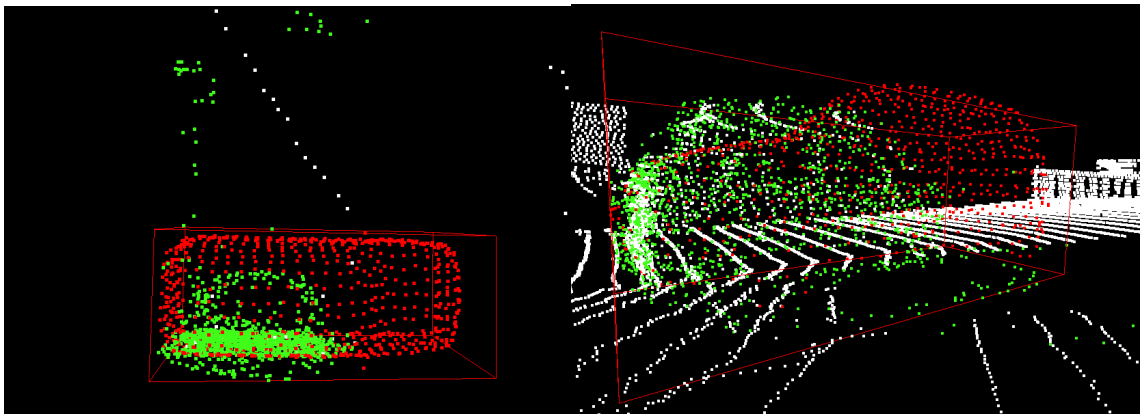


Figure 5.6. Examples of frames containing only standing cars and how our fitting performs well on those frames in terms of Bird's Eye View (BEV). Green points are aggregated points corresponding to cars, red points correspond to a template point cloud P_h fitted to the standing cars, and red bounding boxes denote the 7-DOF optimal parameters.



a) b)

Figure 5.7. Examples of problematic situations in the standing cars fitting process. (a) The fitted template has an incorrect yaw, as the correct fit should be rotated by 90 degrees. (b) even though the BEV looks perfect, the correct fit should be rotated by 180 degrees, as the car orientation is wrong. However, the KITTI [3] object detection dataset evaluation does not care about this ambiguity. Green points are aggregated points corresponding to a car, red points correspond to a template point cloud P_h fitted to the standing cars, and red bounding boxes denote the 7-DOF optimal parameters.

5.4 Fitting of the moving cars

In this section, we'll detail how the fitting process for moving cars differs from that of standing cars. One key difference is that for moving cars, we do not use the segmented aggregated point clouds $\hat{F}$. This is because the aggregation of points for moving objects is very challenging. Instead, we use the segmented point clouds F, as described in Chapter 3. Additionally, the fitting process utilizes the yaw estimation derived from the car's trajectory described in Section 4.5. For this fitting, we also use only the template point cloud P_h.

While we also perform downsampling on F for moving cars, it's a rare occasion because these point clouds are not aggregated over multiple frames, so they are not dense. Unlike with standing cars, we do not discard moving cars based on the number of points in the cloud. The estimation of the spatial location is performed in the same manner as with standing cars.

For moving cars, we again exclude optimization over the y-axis. Given the yaw estimation from the trajectory, we also omit optimization over yaw. In rare cases, where a moving car is not visible for at least three frames, we apply the same fitting process used for standing cars.

The fitting process for moving cars is conducted over a region $C_m \in \{-2, ..., 2\} \times \{-0.5, ..., 2.5\}$. This fitting process optimizes only the parameters x and z. The first dimension denotes the range of x, while the second dimension denotes the range of z, both are in metres. We sample the region with 20 points for each dimension. The adjustment in the z-axis range is based on our observation that our robust spatial location estimator tends to place estimated locations slightly closer than the actual positions. Together with the fixed yaw, this allows us to lower the z-axis range.

We define a parameter set $Q = \{q \in \mathbb{R}^2 \mid q \subset C_m\}$ and a parameters $p \subset Q$, with the estimated spatial location $e \in \mathbb{R}^3$, the estimated yaw $\hat{\theta} \in \mathbb{R}^1$, and the average KITTI car dimensions as a prior. We describe a moving car in 3D using the following 7 parameters:

$$x = e(x)+p(x) \quad y = e(y) \quad z = e(z)+p(z) \quad \theta = \hat{\theta} \quad l = 3.88 \quad h = 1.63 \quad w = 1.53 \quad (3)$$

Where l, h, w denotes the length, height, and width of the car in meters, x, y, z represents the spatial location, and the θ denotes the yaw of the car. We adjust the template point cloud P_h by shifting and rotating it to match the parameters p. Then we proceed to find the parameters p_{opt} with the lowest loss.

We have discovered that when the yaw of the car is accurately estimated, it leads to a significant simplification of the fitting process as this approach directly addresses the ambiguity challenge described in Section 4.1. Despite the sparsity of the point cloud F_i corresponding to i-th car to which we try to fit our template P_h, we achieve surprisingly accurate results in fitting moving cars. Moreover, this fitting process is significantly faster than that for standing cars due to the reduced dimension of the parameter space.

As for standing cars, the fitting process generates all the necessary parameters to define the 3D detection of a car, encoded in the form of a 3D bounding box. These 3D bounding boxes can then be exported into .txt files, functioning as KITTI [3] dataset labels. These labels can be either used for the evaluation of our method or as training data for a 3D detector. Examples of successful fitting of moving cars are illustrated in Figure 5.8, and instances where the fitting presents challenges are shown in Figure 5.9.

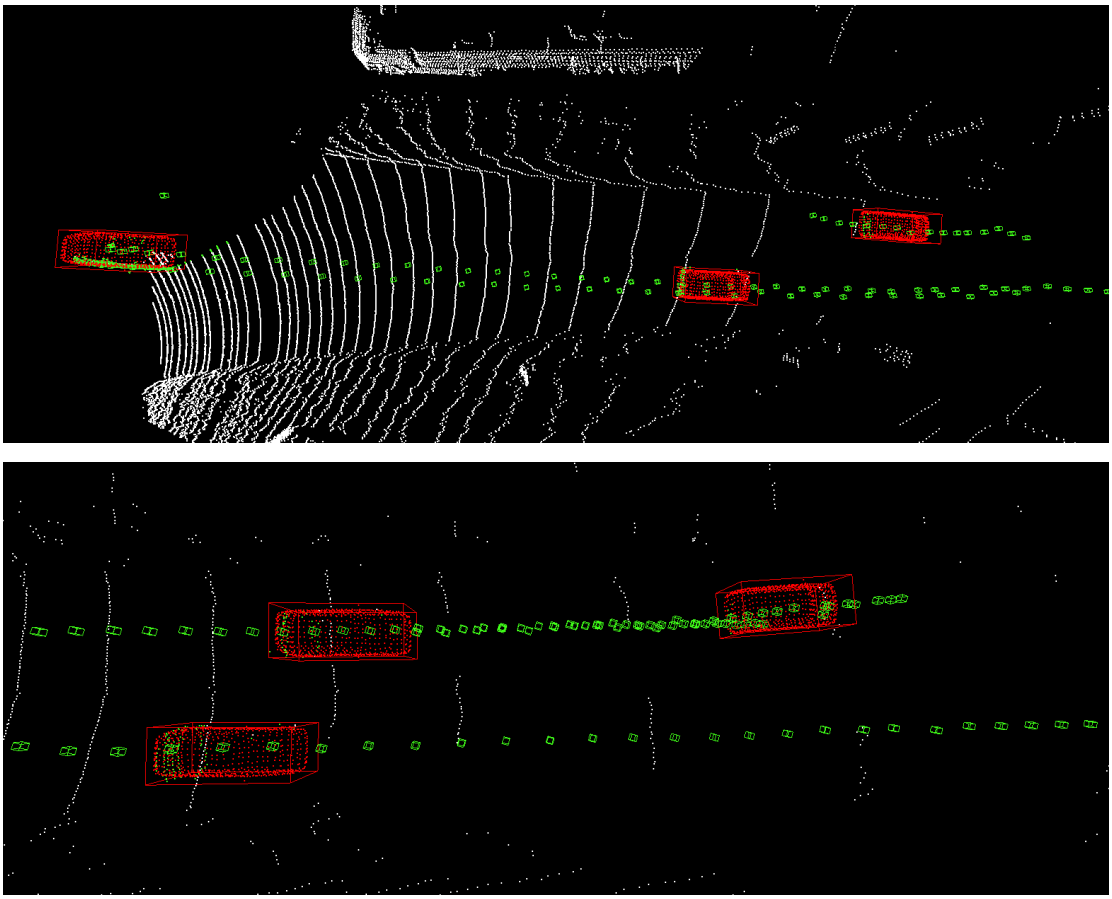
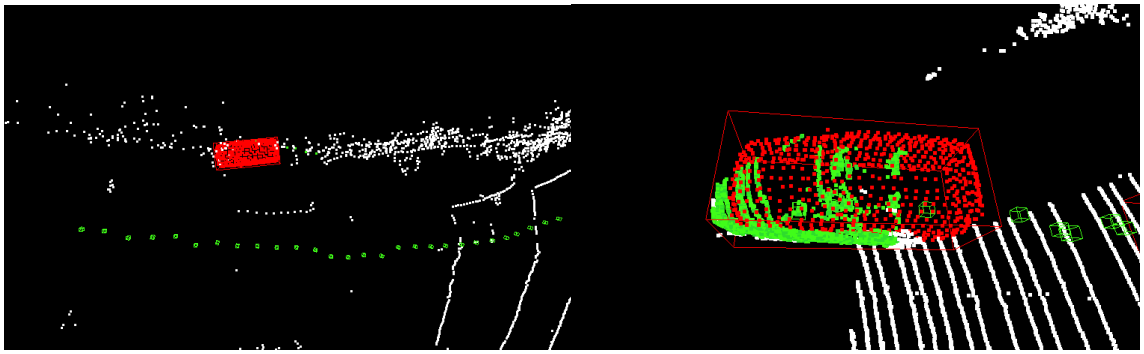


Figure 5.8. Examples of frames containing only moving cars and how our fitting performs well on those frames in terms of Bird's Eye View (BEV). Green points correspond to cars, small green cubes denote the trajectories, red points correspond to a template point cloud P_h fitted to the moving cars, and red bounding boxes denote 7-DOF optimal parameters.



a) b)

Figure 5.9. Examples of problematic examples for the moving cars fitting. (a) The fitted template P_h is in a completely wrong location because of the high outlier-to-inlier ratio, even though the trajectory is correct, (b) the fitted template P_h is not perfectly aligned with the segmented point cloud F_i. Green points correspond to cars, small green cubes denote the trajectories, red points correspond to a template point cloud P_h fitted to the moving cars, and red bounding boxes denote 7-DOF optimal parameters.

5.5 Raycasted templates in the fitting

As previously detailed in Section 5.1, we have generated raycasted point clouds P_{rh}, which include only the points visible given a specific viewing angle. To effectively utilize these raycasted point clouds in our fitting process, we first need to determine the viewing angle of a detected car. This angle is calculated based on the location and yaw of the car.

For each parameters p, provided at each step of the optimization, along with the estimated spatial location $e \in \mathbb{R}^3$, we can calculate the viewing angle α as follows:

$$\alpha = \text{atan2}(e(z) + p(z), e(x) + p(x)) + p(\theta) - \pi/2 \tag{4}$$

Once we have determined the viewing angle α, we select the corresponding raycasted point cloud P_{rh}. During the optimization process, we shift and rotate this selected point cloud in the same manner as we would with the complete point cloud P_h. By using the P_{rh} in the fitting process, we increase the potential similarity between P_{rh} and F_i. The application of this technique, employing the raycasted point cloud P_{rh} in the fitting process, is shown in Figure 5.10.

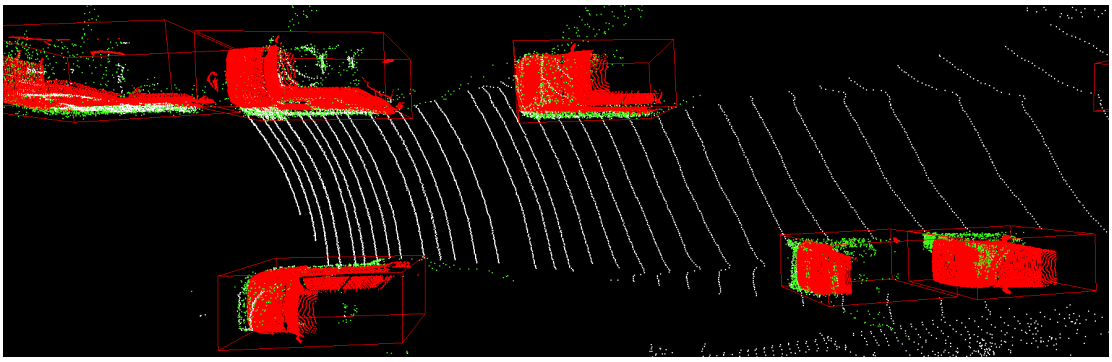


Figure 5.10. Example of the raycasted template point cloud P_h employed in the fitting process on cars in a Bird's Eye View (BEV) scene. Green points are aggregated points corresponding to a car, red points correspond to a template point cloud P_{rh} fitted to the standing cars, and red bounding boxes denote the 7-DOF optimal parameters.

However, during the evaluation, we observed that using the raycasted point cloud P_{rh} in the fitting process does not increase the performance of our method at all when using the best-performing loss function (Template Fitting Loss). The different loss functions are described further in the chapter. Consequently, in our fitting process, we have chosen to use P_h in place of P_{rh}.

5.6 Loss functions

This section is focused on various loss functions that play a crucial role in the fitting process of two point clouds. In the context of our method, these point clouds are the segmented (aggregated) point cloud F_i ($\hat{F}_i$) and the generic shape point cloud P_h. These loss functions are essential for estimating the alignment between F_i, which represents the actual car in the scene, and P_h, our template model.

5.6.1 Chamfer Distance Loss

Initially, we have chosen to use the well-known Chamfer Distance Loss. The formula of the Chamfer Distance Loss is as follows:

$$L(F_i, P_h) = \sum_{x \in F_i} min_{y \in P_h} \|x - y\|_2^2 + \sum_{y \in P_h} min_{x \in F_i} \|x - y\|_2^2 \tag{5}$$

In words, the Chamfer Distance Loss consists of finding the nearest neighbor for each point in F_i from the points in P_h and calculating the L2 distance between them. Then, it sums all these distances for all points in F_i and repeats the process from the perspective of P_h.

In our method, we found that Chamfer Distance Loss does not perform well. This loss function is typically used when comparing very similar point clouds that might only differ by a rigid body transformation. The Chamfer Distance Loss handles outliers very poorly. It assigns a high cost to outliers in F_i, as they tend to be far from their nearest neighbors in P_h. This high cost can cause the fitting process to inadequately adjust for these outliers, which gives a poor performance as there should be no attention paid to outliers.

It is interesting to note, though, that the Chamfer Distance Loss shows better performance with the raycasted template point cloud P_{rh} than with the complete template point cloud P_h.

To improve the loss function, we propose normalizing each part of the loss by the size of its corresponding point cloud. The modified formula for the Chamfer Distance Loss, employing this normalization, is as follows:

$$L(F_i, P_h) = \Big(\sum_{x \in F_i} min_{y \in P_h} \|x - y\|_2^2 \Big) / |F_i| + \Big(\sum_{y \in P_h} min_{x \in F_i} \|x - y\|_2^2 \Big) / |P_h| \tag{6}$$

5.6.2 Median Chamfer Distance Loss

In our efforts to improve the Chamfer Distance Loss to reduce its sensitivity to outliers, we have chosen to use the median instead of the sum. This approach follows the usage of the median within this thesis for its robustness. The modified formula for this Median Chamfer Distance Loss, which uses the median rather than the sum of point distances, is as follows:

$$L(F_i, P_h) = \text{median}_{x \in F_i}\Big(min_{y \in P_h}\|x - y\|_2^2\Big) + \text{median}_{y \in P_h}\Big(min_{x \in F_i}\|x - y\|_2^2\Big) \tag{7}$$

Surprisingly, the shift from using a sum to a median in our loss calculation provided a significant boost in performance. This improvement is primarily due to the median's ability to disregard outliers if the ratio of outliers to inliers remains low. At the same time, the median also discards points that fit perfectly, which is actually a plus because our iteration step is relatively coarse and we do not care if the distance between a point is 1 or 2 centimetres, but rather we care if the distance is 1 or 50 centimetres. It's interesting to note that, as the original Chamfer Distance Loss, the Median Chamfer Distance Loss shows better performance when used with the raycasted template point cloud P_h.

With the same approach as in the Chamfer Distance Loss, we normalize each part of the Median Chamfer Distance Loss by the size of the corresponding point cloud. We have thus modified the formula for the Median Chamfer Distance Loss to include this normalization, and it is structured as follows:

$$\begin{aligned} L(F_i, P_h) = &\Big(\text{median}_{x \in F_i}(min_{y \in P_h}\|x - y\|_2^2)\Big)/|F_i| \\ &+ \Big(\text{median}_{y \in P_h}(min_{x \in F_i}\|x - y\|_2^2)\Big)/|P_h| \end{aligned} \tag{8}$$

It is worth noting that it is possible to use only one-sided Median Chamfer Distance Loss, which means either the median over F_i or P_h is taken. However, none of those adaptations ever achieved a reasonable performance compared to the double-sided versions.

5.6.3 Template Fitting Loss

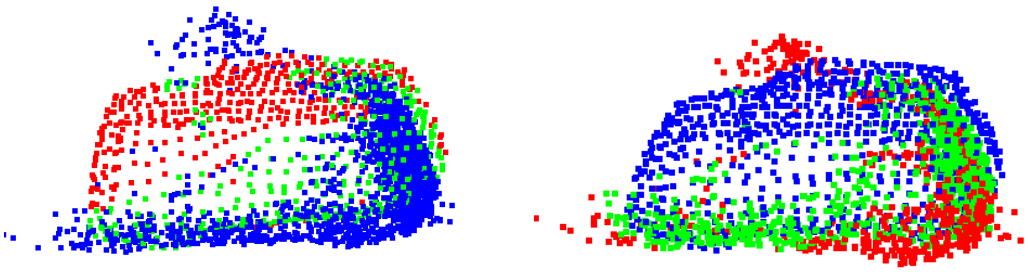
To further refine the Chamfer Fitting Loss, we have developed a new loss, which we call the Template Fitting Loss. The inspiration comes from the RANSAC algorithm [28], which utilizes the ratio of inliers to outliers as an optimization criterion. We have chosen a different approach from the conventional L2 norm. Instead, we employ a saturation function instead of the L2 norm. This means that if the distance between points is less than a certain threshold, the value is set to 1, indicating an inlier. On the other hand, if the distance exceeds this threshold, the value becomes 0, indicating an outlier. The loss behaves simply as an estimator of the inlier-to-outlier ratio. The formula for the Template Fitting Loss, employing this saturation-based inlier counting, is as follows:

$$L(F_i, P_h) = \sum_{x \in F_i} g\Big(min_{y \in P_h}\|x - y\|_2^2\Big) + \sum_{y \in P_h} g\Big(min_{x \in F_i}\|x - y\|_2^2\Big) \tag{9}$$

$$g(d) = \begin{cases} 1 & \text{if } d \leq \epsilon \\ 0 & \text{if } d > \epsilon \end{cases} \tag{10}$$

Where ϵ denotes the distance threshold, for which we have empirically found that the optimal value is 0.2 meters.

The logic behind this loss function is as follows: When aligning two point clouds, F_i and P_h, our goal isn't to achieve perfect alignment of just a small subset of points. Instead, we aim for a reasonable alignment across the majority of the points. In this context, we don't care if the distance between points is 1 or 2 centimetres. What



a) b)

Figure 5.11. Example of the Template Fitting Loss. (a) represents the points from the template P_h, blue points are from F_i, red points are outliers and green points are inliers from P_h. (b) represents the points from the F_i, blue points are from P_h, red points are outliers and green points are inliers from F_i.

matters more is if the distance between points is 1 centimetre (inlier) or 50 centimetres (outlier). Our goal is to maximize the number of inliers, those points where the distance is relatively small. This focus on the inliers instead of distance is employed by the saturation function $g(d)$, which differentiates between inliers and outliers. The visualization of this loss function is shown in Figure 5.11.

The Template Fitting Loss has achieved the best performance among all other loss functions discussed in this thesis. It achieves the best performance when the complete template point cloud P_h is used. Again it is a good idea to normalize each part of the loss by its corresponding point cloud size. This results in an updated formula as follows:

$$L(F_i, P_h) = \Big(\sum_{x \in F_i} g(min_{y \in P_h} \|x - y\|_2^2) \Big) / |F_i| + \Big(\sum_{y \in P_h} g(min_{x \in F_i} \|x - y\|_2^2) \Big) / |P_h| \quad (11)$$

It's worth noting that our loss function essentially represents the inlier-to-outlier ratio. Therefore, during the optimization process, which involves minimization, we need to invert the loss by multiplying it by -1. This adjustment is needed so the optimization process minimizes the number of outliers.

To further enhance the speed of our method, we have successfully utilized the FAISS library [44], known for its speed in similarity search and clustering of dense vectors. The integration of FAISS into our method is as follows:

Given that the point cloud F_i remains constant during the optimization, we take advantage of this property. We create a quantizer IndexFlatL2 from the FAISS library to represent F_i. Subsequently, we build an IndexIVFFlat using this quantizer. The number of cells n_c for the IndexIVFFlat is determined by a specific formula, which is designed to optimize the balance between search speed and accuracy. This formula for selecting the number of cells n_c in the IndexIVFFlat is as follows:

$$n_c = \lfloor \sqrt{n} \quad (12)$$

Where n denotes the number of points in the F_i. Since building the index is a time-consuming step, we perform it only once at the beginning of the optimization. The points are then divided into multiple cells in the index. We can adjust a parameter to control how many neighboring cells are searched around a prior estimate. Setting this parameter to the total number of cells follows the behaviour of a k-nearest neighbors

algorithm. However, reducing the parameter makes the search approximate, which is more efficient. For our method, searching only the cell of the prior estimate has proven to be sufficiently precise while also being the fastest.

Instead of a standard k-nearest neighbor search, we use the range_search function. This function, for a given query point q, returns all points within a specified range. We set this range equal to our distance threshold ϵ.

To compute the loss, we query each point from P_h in the index to find all neighbors within ϵ. The output format is as follows: for each query point in P_h that has at least one neighbor in F_i represented as an Index, it returns a list of neighbor indexes from F_i.

For the part of the loss function that sums over all points in P_h, we simply count the number of arrays returned as points in P_h with no neighbors in F_i within the distance threshold do not return any array.

For the part of the loss function summing over all points in F_i, we merge all returned arrays of indexes and count the unique indexes. This count indicates the number of points in F_i that have at least one neighbor in P_h within the threshold. We must ensure that each point in F_i is counted only once, even if it appears multiple times in the merged array.

This FAISS implementation allows for multi-core processing and uses approximate k-nearest neighbors, resulting in linear time growth with an increasing number of points. This is a significant improvement over the quadratic increase seen with the exact k-nearest neighbors algorithm. Thanks to this efficiency, we can increase the number of optimization steps per parameter from 20 to 40 and use denser point clouds (around 2000 points instead of 1000), further increasing the performance of our method.

5.6.4 Occlusion Loss

To address the ambiguity challenge outlined in Section 4.1, we have developed an additional loss function, which we call the Occlusion Loss. The idea behind this loss is straightforward. In a LiDAR point cloud, a line drawn from the LiDAR scanner to any point in the point cloud implies that the space along this line is free of objects. If it weren't, the scanner wouldn't be able to detect the point.

The first step in implementing this idea is to generate a 3D voxel grid representing the space around the ego-vehicle. For each voxel in this grid, we need to determine its state: whether it's free or occupied. Initially, we set all voxels to the state occupied. We determine these states by drawing lines from each point in the LiDAR scan to the origin (the position of the LiDAR scanner). To do this, we utilize the Bresenham line algorithm [45], which provides the indices of voxels intersected by these lines. For each voxel, we keep the number of total intersections with all lines.

A voxel is classified as free if the number of intersections exceeds a given threshold. It is classified as occupied if there is at least one LiDAR scan point inside the voxel or the number of intersections is lower than a given threshold.

It is important to note that the number of intersections per voxel is highly influenced by its distance from the ego-vehicle. To address this, we divide the number of intersections of each voxel by the squared L2 distance to the ego-vehicle. The scaled 3D voxel grid is shown in Figure 5.12.

One of the disadvantages of the Occlusion Loss method is that we cannot aggregate the number of voxel intersections over multiple frames due to the movement of cars.

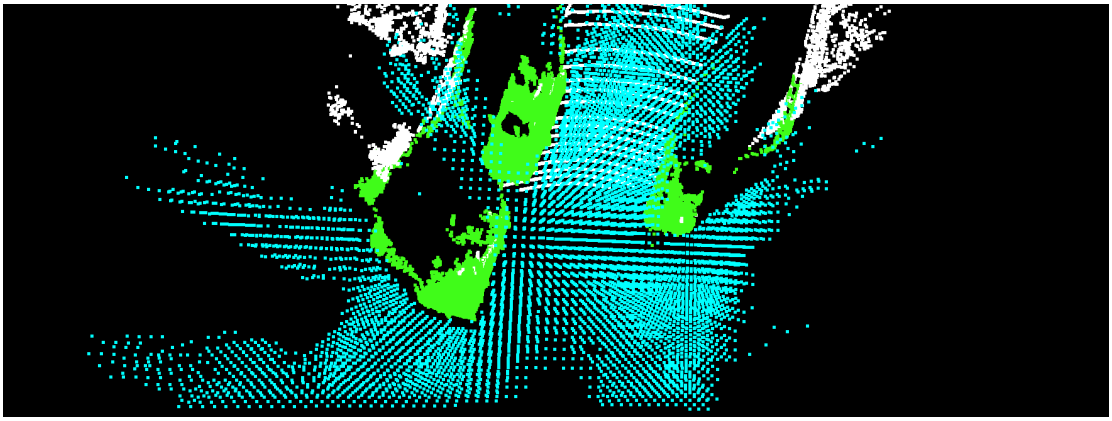


Figure 5.12. Example of the 3D voxel grid used for the occlusion loss. The blue points denote free voxels, so the template point cloud P_h or P_{rh} should not intersect those. Green points correspond to cars, and white points are from the LiDAR scan.

Given a 3D voxel grid that categorizes each voxel based on occupancy, we want to use it in the fitting process. Here, we take the shifted and rotated template point cloud P_h or P_{rh}, corresponding to each parameter set, and calculate the loss as follows.

For every point in the template point cloud P_h or P_{rh}, we assign it to a corresponding voxel in the grid. If a voxel is marked as free, it implies that the template point cloud should not intersect with this voxel. If that happens, we add 1 to the loss for each free voxel intersected. This loss is then normalized by dividing it by the total number of points in P_h or P_{rh} and scaled with a weight. This scaling allows the Occlusion Loss to be integrated with other loss functions previously mentioned, as it serves as supplementary information and is never used in isolation.

However, it is important to note that the Occlusion Loss has not shown improvement in performance in our experiments. The main problem lies in the usage of the generic size of the template P_h or P_{rh}, which may not match the exact dimensions of the actual car. Even though we can provide a perfect fit, the overlapping of the template will cause the intersections with free voxels, resulting in increased loss. We believe that with more attention, this loss could be useful. However, we chose to go in another direction.

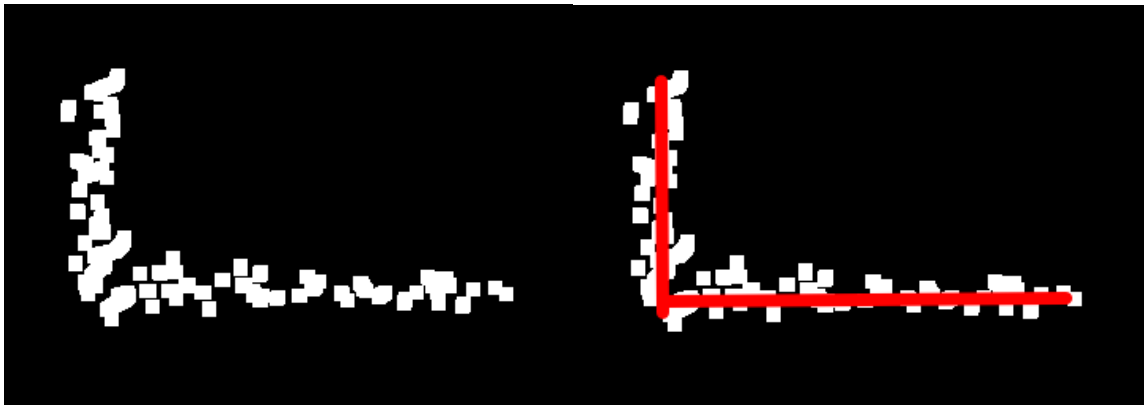
5.7 Histogram Yaw Estimation

Another way to estimate the yaw of the car we did a little bit of research is something we call Histogram Yaw Estimation. This technique is specifically designed for use with segmented aggregated point clouds $\hat{F}$ of standing cars.

To illustrate the concept, let's consider a simple example of what a segmented aggregated point cloud $\hat{F}_i$ might look like, as shown in Figure 5.13. For a given point cloud $\hat{F}_i$, we can encode the cloud into two lines that represent the dominant angles among the points. This encoding allows us to compare two point clouds based on these dominant angles. In our case, the comparison is between $\hat{F}_i$ and P_{rh}.

We compute the encoding as a histogram. We randomly sample point tuples (p, q), where both points are from the same point cloud. Then, we compute the angle β between (p, q) with the following equation:

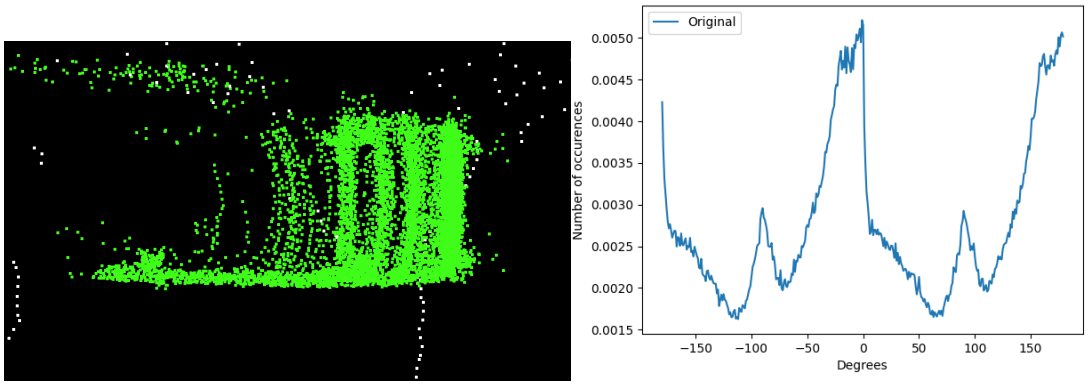
$$\beta = \text{atan2}\Big(p(z) - q(z), p(x) - q(x)\Big) \tag{13}$$



a) b)

Figure 5.13. Example of the reasoning behind Histogram Yaw Estimation. (a) hand drawn simple example of segmented aggregated point cloud $\hat{F}_i$, (b) key lines describing the point cloud $\hat{F}_i$.

The computed angles β are then utilized to create a histogram representing the point cloud $\hat{F}_i$ or P_{rh}. As stated before, the histogram should contain two dominant angles, one being the correct yaw of the car. A real histogram computed for one car from the KITTI [3] dataset is shown in Figure 5.14, where the dominant angles are shown.



a) b)

Figure 5.14. Example of the (b) histogram corresponding to (a) a car. Green points correspond to segmented aggregated point cloud F_i. The correct yaw is around 0 degrees, which is also the dominant angle in the histogram. The dominant angle is also around 180 degrees because this method cannot distinguish if the car is facing forward or not.

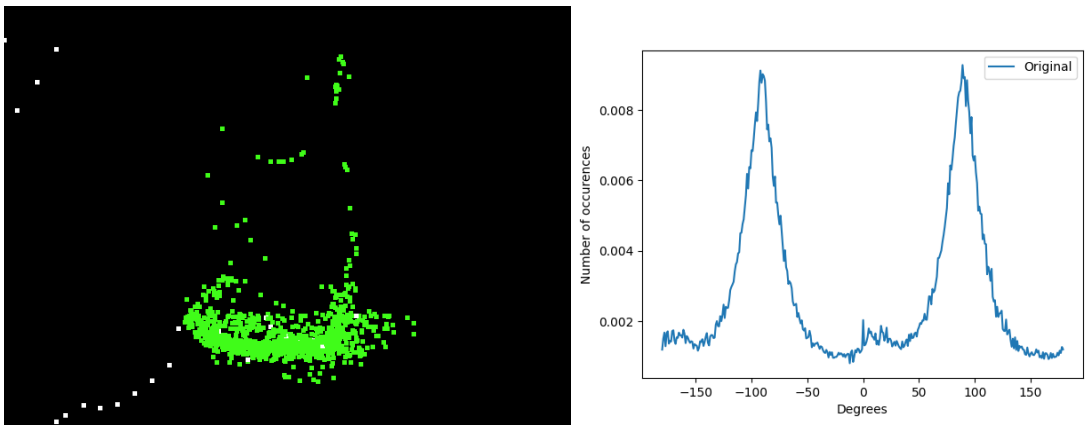
Now, let us integrate the Histogram Yaw Estimation into our fitting process. The first step involves taking the segmented aggregated point cloud $\hat{F}_i$ and computing its corresponding histogram. We also calculate the viewing angle α for $\hat{F}_i$ using the estimated spatial location $e \in \mathbb{R}^3$. The viewing angle is needed for shifting the histogram to compensate for the angle at which the car is seen. The viewing angle α is calculated as follows:

$$\alpha = -\text{atan2}\Big(e(z), e(x)\Big) + \pi/2 \tag{14}$$

After computing the histogram and the viewing angle, we iterate over all raycasted template point clouds P_{rh} (one for each yaw degree). For each P_{rh}, we compute its

histogram, shift it by α to account for the viewing angle, and then compute the similarity with the histogram corresponding to $\hat{F}_i$. The P_{rh} corresponding to a yaw angle with the most similar histogram is selected. Various functions are employed to estimate the histogram similarity, including L1 and L2 distance per bin, Bhattacharyya distance [46], and Earth Mover's Distance (EMD). As these histograms are compensated for the viewing angle, they are location-invariant; thus, maximizing similarity leads to an estimation of the yaw. With this estimated yaw, similar to the process for moving cars, we then optimize over two parameters to refine the location estimation.

Even though this method looks promising, we have encountered problems in practice. Our results with Histogram Yaw Estimation have not been as successful as hoped, often leading to failed yaw estimation, which caused the fitting process to fail. Consequently, a brute-force-like fitting approach remains more robust for our needs. We have not assigned a lot of focus to this method. However, we believe that this yaw estimation should receive more attention in the future as it might provide promising results. An example of a situation where Histogram Yaw Estimation did not perform well is illustrated in Figure 5.15.



a) b)

Figure 5.15. Example of a failed histogram yaw estimation. (a) Segmented aggregated point cloud F_i, green points correspond to a car, (b) histogram corresponding to F_i. The correct yaw is approximately 0 yaw. However, the histogram suggests ± 90 degrees, which is wrong. This is caused by seeing only the back of the car.

Chapter 6
Scale Detector

In this chapter, we describe the Scale Detector, our method designed for estimating the dimensions (scale) of standing cars. This method is crucial for increasing performance on the tight 0.7 3D IoU evaluation threshold.

We begin in Section 6.1 by describing the process of point aggregation from multiple frames. Section 6.2 is then dedicated to the fitting process utilized within the Scale Detector. The fitting process is very similar to the ones described in Chapter 5.

Finally, in Section 6.3, we discuss the refinement of the fitting process outputs. This step is needed to create reasonable outputs and account for the bias in the KITTI dataset [3] labels.

6.1 Point aggregation

The main motivation for the Scale Detector's development is the objective to estimate the dimensions of cars, a task that currently hinders our method's performance, especially on the 0.7 3D IoU threshold. The problem in this estimation process is the limited visibility of the entire car. Our car segmentation process only applies to cars in front of the ego-vehicle, meaning we only have a clear view of up to two sides of any given car. Unfortunately, this limited perspective is insufficient for a robust estimation of a car's dimensions.

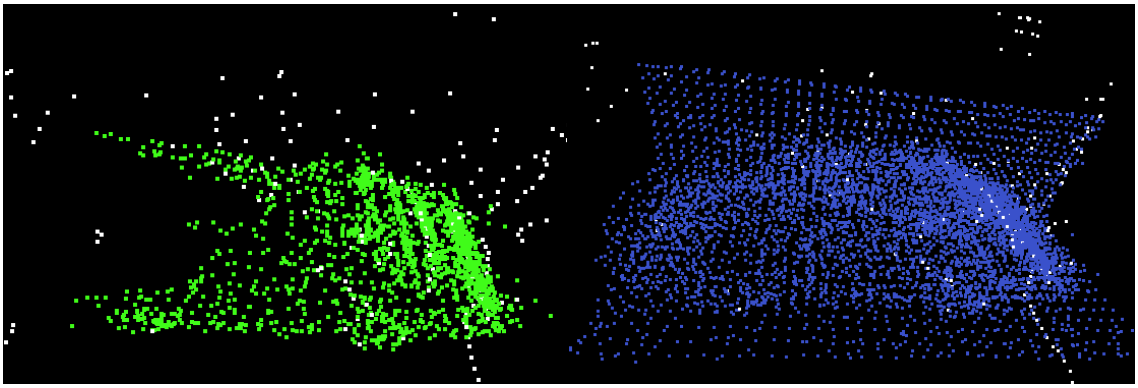
Seeing at least three sides of a car would significantly simplify the dimension estimation process, providing a fuller understanding of the car's shape and size. This is demonstrated in Figure 6.1, which shows the difference in estimation based on the number of visible car sides.

a) b)

Figure 6.1. Example of the Scale Detector motivation. (a) Car viewed only from two sides in Bird's Eye View (BEV), (b) Car viewed from three sides in BEV. Estimating the car's dimensions in (a) is not straightforward, as we might not capture some critical points, creating ambiguity in estimating the dimensions. However, in (b), the ambiguity in estimating the length of the car has vanished, thanks to the visibility of the third side.

While the LiDAR scanner can capture a 360-degree view around the ego-vehicle, we must focus only on the field of view (FOV) corresponding to the camera. In segmenting points from the LiDAR scan, we utilize the 2D binary masks provided by Detectron2 [35]. In Scale Detector, we instead rely on the 3D detections obtained from the fitting process described in Chapter 5. For each car instance, we segment all points within its 7-DOF 3D bounding box in the LiDAR scan, resulting in a segmented point cloud $S_i \in \mathbb{R}^{3\times n}$ for the i-th car, where n represents the number of points.

Simply employing this segmentation in the reference frame would not gain additional information. Therefore, similar to the approach in Chapter 4, we aggregate these segmented point clouds over multiple adjacent frames. This aggregation results in a segmented aggregated point cloud $\hat{S}_i \in \mathbb{R}^{3\times m}$ for the i-th car, where m represents the total number of aggregated points. This method allows us to view the car from various angles, enabling the segmentation even when the car is not visible in the camera's FOV. We need to expand the dimensions of the bounding box by 50% to increase the capture area. However, it's important to note that this strategy only works for standing cars. The process of point aggregation and how it views the car from different perspectives is shown in Figure 6.2.



a) b)

Figure 6.2. Comparison of (a) Segmented aggregate point cloud $\hat{F}_i$ used for fitting in Chapter 5, (b) Segmented aggregated point cloud $\hat{S}_i$ used in Scale Detector.

However, this strategy produces many outliers, which we have been unable to filter out. As expected, the point clouds $\hat{S}$ are very dense. Thus, we need to perform the downsampling in the same manner as in Section 4.4.

6.2 Fitting process in Scale Detector

The fitting process is similar to that described in Chapter 5. The key difference, however, lies in the target of our fitting process. Instead of fitting the template point cloud P_h to the segmented aggregates point cloud $\hat{F}_i$, we fit P_h to the aggregated segmented point cloud $\hat{S}_i$ associated with the Scale Detector.

One of the objectives is to estimate the height of the car. As we have discovered, as we are fitting the generic shapes of the car, optimizing over the height does not provide good results. The computation of the car's height, denoted as h, is as follows:

$$h = \max_{s\in\hat{S}_i}(s(y)) - \min_{s\in\hat{S}_i}(s(y)) \tag{1}$$

In the scale detection fitting process, we limit the car's height h to between 75% and 125% of the average car height in the KITTI dataset [3]. This is because the height variance among cars is high. Our fitting process already provides a relatively accurate estimation of spatial location and yaw. However, when adjusting the length or width of the car, it becomes necessary to optimize the spatial location again. Additionally, we employ slight adjustments in the yaw to enhance the fit potentially.

We've observed the best results when the length and width parameters are tied together based on the assumption that longer cars tend to be wider, and vice versa.

For this purpose, we define a region $C_s \in \{0, 1\} \times \{0.67, ..., 1.5\} \times \{-r_x, ..., r_x\} \times \{-r_z, ..., r_z\} \times \{-25, ..., 25\}$. The first dimension represents the two different template meshes used: the hatchback template P_h and the sedan template P_s. We have discovered that using both templates increases the performance of our method. The second dimension is the scale of the car. Here, the length scale s_l varies from 0.67 to 1.5, corresponding to lengths ranging from 2.6 meters to 5.8 meters, as the scaling is used on the average KITTI car dimensions. The number of steps for the scale is 8. The width scaling s_w, which is proportionally linked to the length scaling, is calculated using the following formula:

$$s_w = 1 + (s_l - 1) \cdot 0.75 \tag{2}$$

Thus, the s_w ranges from 0.75 to 1.38, corresponding to widths ranging from 1.15 to 2.11 meters. The next two dimensions are x and z. Where the range for x is $-r_x$ to r_x and z is $-r_z$ to r_z with a number of steps equal to 10. The r_x and r_z is computed as follows:

$$r_z = \cos(\theta) + \sin(\theta)/2 \tag{3}$$

$$r_x = \sin(\theta) + \cos(\theta)/2 \tag{4}$$

The reasoning is that the need to change the spatial location due to the length is more significant than changing the width, so the range on the axis belonging to the length must be greater. The last parameter to optimize is the yaw that ranges within ± 25 degrees around the yaw estimate from the fitting process described in Chapter 5. The number of steps for θ is 10. This variation of the yaw accounts for both possible orientations (headings) of the car.

Regarding the loss function, we employ the same one used in the fitting process described in Chapter 5. Both the complete point clouds P_h, P_s and their raycasted versions P_{rh}, P_{rs} can be used in the Scale Detector. However, we found that using the complete point clouds P_h and P_s provides better overall performance.

Once we find the parameters $p_{\text{sopt}} \subset Q_s$, where $Q_s = \{q \in \mathbb{R}^5 \mid q \subset C_s\}$, that results in the lowest loss, we compute a one-sided loss from the template to the segmented aggregated point cloud. Given that we primarily use the Template Fitting Loss, which calculates the inlier-to-outlier ratio, we require that this ratio must exceed 0.7 for the scale detection to be considered valid for the i-th car. If this threshold is not met, it suggests that we may not have sufficient information for effective scale detection, so we do not update the dimensions at all. The output of the fitting process is shown in Figure 6.3.

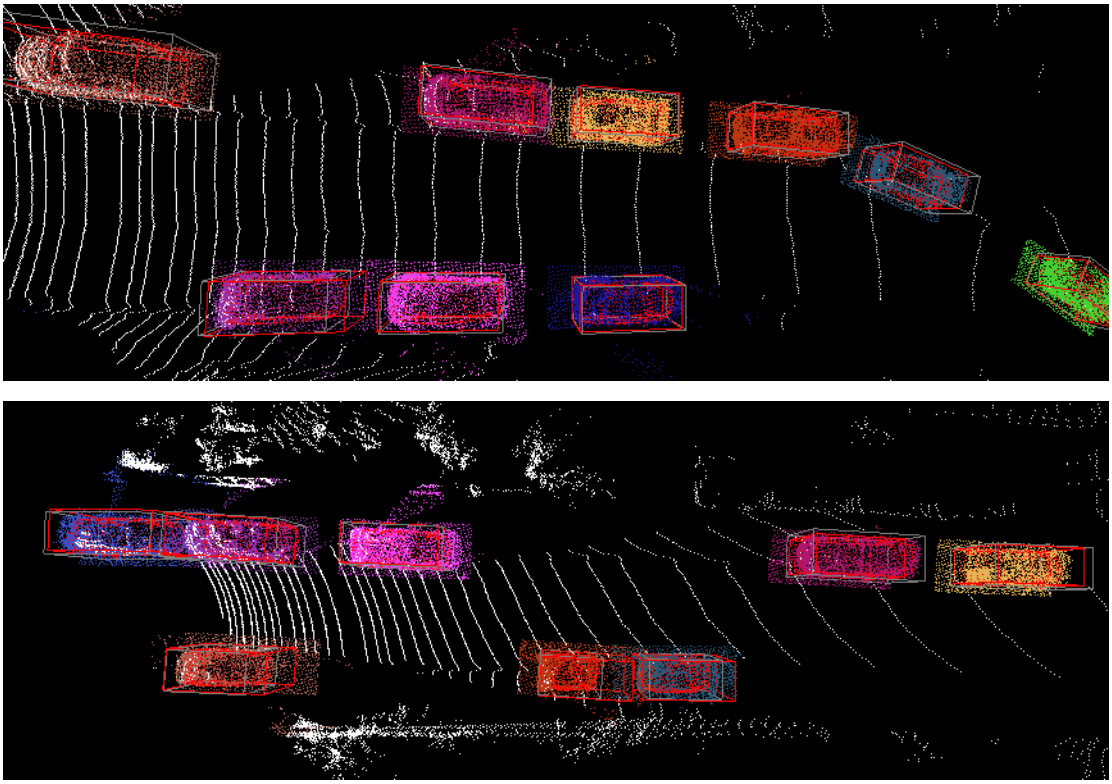


Figure 6.3. Examples of two scenes with the Scale Detector. Colourful points correspond to segmented aggregated point clouds $\hat{S}$. The red 3D bounding box represents the output of the fitting phase with average KITTI car dimensions and the grey 3D bounding box is provided by the scale detection.

6.3 Bounding box reducer

In the fitting process of the Scale Detector process described in Section 6.2, we often encounter a situation where the resulting 3D bounding boxes are larger than the KITTI ground truth labels. This probably arises from a bias in human labelling, where it appears that labellers might be relying only on the reference frame LiDAR scan, providing a bounding box that only describes the car's appearance in that specific scan.

To tackle this issue, we implement a function known as the Bounding Box Reducer. This method begins by taking the bounding box output from the Scale Detector fitting stage and slightly increasing its width. Additionally, the bounding box is shifted upward by 0.4 meters along the y-axis. This shift is done to exclude any points on the ground. Following this adjustment, the method extracts all points from the reference LiDAR scan that fall within the adjusted 3D bounding box.

The final phase of the Bounding Box Reducer involves finding the smallest possible 3D bounding box that can include all the extracted points. We have encountered a problem with this approach due to the sparsity of some segmented point clouds, which sometimes led to unrealistically small bounding boxes. To avoid this, we apply a rule: if the reduction in the bounding box's length exceeds 25%, we keep the original dimensions. In the other case, we adjust the length of the detected car, while keeping the height and width unchanged. This provides the best results. The Scale Detector with Bounding Box Reducer applied to a real scene is shown in Figure 6.4.

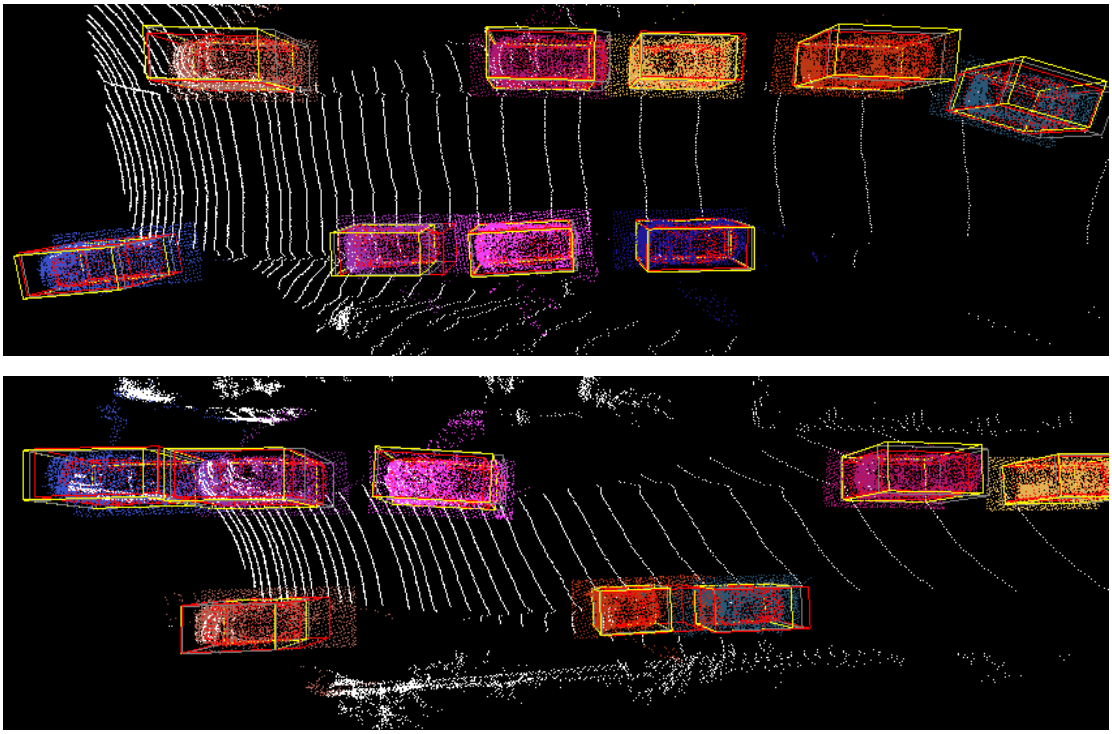


Figure 6.4. Example of two scenes with the Scale Detector and Bounding Box Reducer. Colourful points correspond to segmented aggregated point clouds $\hat{S}$, red points correspond to templates P_h or P_s, the red 3D bounding box represents the output of the fitting phase with average KITTI [3] car dimensions, the grey 3D bounding box is provided by the Scale Detector, and the yellow 3D bounding box is the reduced one.

Chapter 7
Voxel-RCNN adaptation

In this chapter, the adaptation of Voxel-RCNN for weakly-supervised training is explored. The detector's pre-training on pseudo ground truth is discussed in Section 7.1, while Section 7.2 focuses on describing the network's outputs and associated losses. Section 7.3 denotes the Differentiable Template Fitting Loss, and Section 7.4 describes the Mask Appearance loss. Techniques for data augmentation are outlined in Section 7.5, with the entire training process detailed in Section 7.6.

It is worth noting that the OpenPCDet [47] framework performs all operations in the LiDAR coordinate frame (z-axis is upwards instead of y-axis), while the rest of this thesis operates in the Camera coordinate frame. The coordinate frames are illustrated in Figure 2.1.

7.1 Voxel-RCNN with pseudo ground truth labels

We have generated pseudo ground truth labels for all training set scenes using our method mentioned in the previous chapters. This method integrates the foreground segmentation process from Chapter 3, the exploitation of temporal consistency from Chapter 4, the fitting process outlined in Chapter 5, and the Scale Detector approach from Chapter 6. These pseudo ground truth labels were then used to replace the KITTI dataset labels [3], allowing us to pretrain the Voxel-RCNN in a similar manner as with the original KITTI labels.

Interestingly, despite our labels not showing exceptionally high accuracy on their own, their use in the training of a 3D detector has resulted in a substantial performance boost. This approach aligns with common practices in other weakly supervised methods, which often incorporate a 3D detector at the end of their processing pipeline.

Unique to our approach, however, is the modification of the Loss functions in the Voxel-RCNN. These modifications enable the utilization of additional information extracted by our method, such as segmented aggregated point clouds and binary masks. This is further discussed in this chapter.

For implementing Voxel-RCNN [9], we chose the OpenPCDet framework [47], which is a framework that has multiple 3D detectors implemented, including the Voxel-RCNN.

7.2 Voxel-RCNN outputs

While Voxel-RCNN [9] was initially introduced in Section 2.2.5, this section is going to mention specific details about its outputs and loss computation.

The architecture and outputs of the Voxel-RCNN are shown in Figure 7.1. Focusing first on the RPN Head (highlighted in green), it produces 70,400 anchors. Each anchor is associated with a 7-DOF 3D bounding box and a classification score prediction. Based on the IoU overlap with ground truth labels, the classification score is evaluated using Binary Cross Entropy Loss (BCE). For the 3D bounding box parameters, a smooth-L1

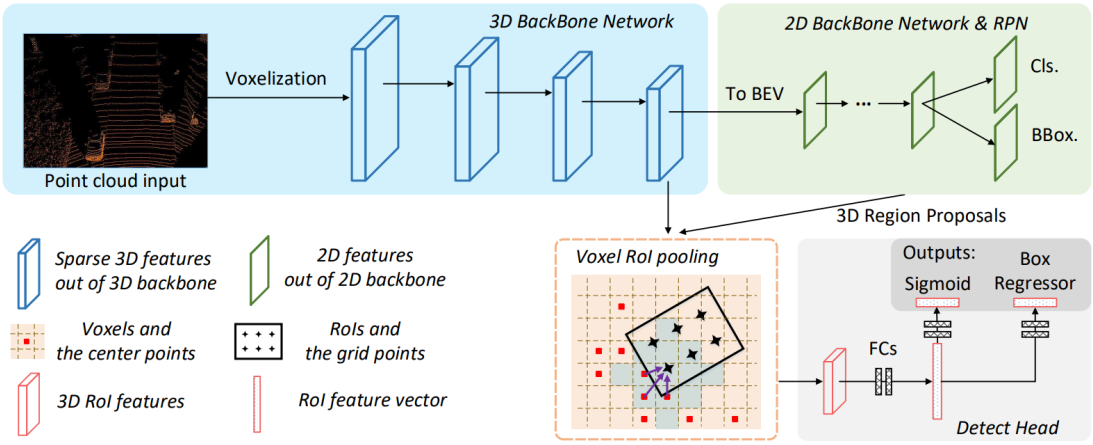


Figure 7.1. Pipeline of Voxel-RCNN [9] described in 2.2.5. The figure is taken from [9].

Regression Loss is applied, but only for anchors exceeding a certain IoU threshold with any ground truth label.

An additional component in the RPN Head, not detailed in the original paper, is the direction classification loss. Anchors are grouped in pairs, with one anchor in each pair rotated by 90 degrees. This direction classification predicts which anchor in each pair should be selected, employing BCE loss.

In the next stage, the Detect Head (highlighted in orange and grey), only a subset of RPN anchors, those with high IoU with ground truth labels, are considered. Similar to the RPN Head, the outputs here are 7-DOF bounding boxes with corresponding scores. However, these scores represent the predicted IoU with the ground truth label rather than the probability of the prediction being a car. The same BCE loss is used for the IOU scores, and the smooth-L1 Regression Loss is applied to the bounding boxes.

Furthermore, the Detect Head includes additional loss not mentioned in the original paper. The loss is the Corner Regression Loss, where two points in space represent the 3D bounding boxes, and the smooth-L1 Regression Loss is computed on the difference of points corresponding to prediction and ground truth.

In our adaptation of the Voxel-RCNN, we have decided not to modify the loss functions in the RPN Head. We found that using our pseudo ground truth labels was efficient enough to generate reliable detections at this stage. Furthermore, modifying only the loss functions in the RPN Head wouldn't significantly impact overall performance. This is because the final evaluation relies on the outputs from the Detect Head, which converges to similar solutions regardless of the specific weights used in the RPN Head.

On the other hand, we identified an opportunity to enhance performance by adjusting the losses in the Detect Head. While keeping the IoU-based classification BCE loss and the smooth-L1 Regression Loss, we introduced two additional losses specifically for the 3D bounding box parameters: the Template Fitting Loss and the Mask Appearance Loss. These new losses, which we will discuss in more detail later in this chapter, are designed to further refine and improve the accuracy of the 3D bounding box predictions

7.3 Differentiable Template Fitting Loss

Each proposal generated by the RPN Head is selected based on its IoU with the 3D pseudo ground truth bounding box. Thus, for each proposal, we have an associated

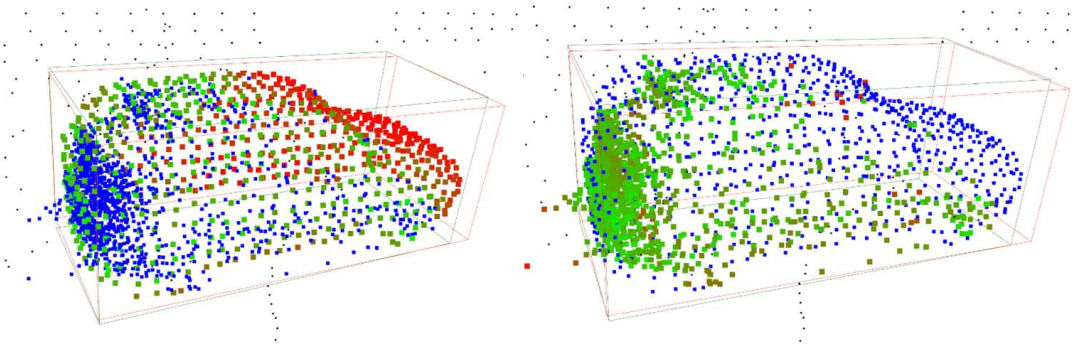
segmented (aggregated) point cloud, either F_i or $\hat{F}_i$. We can create a template that represents the 7-DOF 3D bounding box, similar to the process detailed in Chapter 5. Consequently, we can apply the Template Fitting Loss, as introduced in Subsection 5.6.3, which has demonstrated superior performance among various losses.

However, a significant challenge arises with the Template Fitting Loss: it is inherently non-differentiable and therefore unsuitable for backpropagation. The primary issue is the saturation function within the loss calculation. To address this, we've adapted the loss function by substituting the non-differentiable saturation function with a differentiable sigmoid function. This modification allows the Template Fitting Loss to approximate its original functionality while becoming differentiable and suitable for backpropagation. The modified Template Fitting Loss formula, utilizing the sigmoid function, is presented as follows:

$$L(F_i, P_h) = \Big(\sum_{x \in F_i} g(min_{y \in P_h} \|x - y\|_2^2) \Big) / |F_i| + \Big(\sum_{y \in P_h} g(min_{x \in F_i} \|x - y\|_2^2) \Big) / |P_h| \quad (1)$$

$$g(d) = \sigma(k \cdot d) \quad (2)$$

Where k denotes the steepness parameter. We have empirically proved k to be optimal when set to 10. The differentiable Template Fitting Loss is shown in Figure 7.2.



a) b)

Figure 7.2. Example of the Differentiable Template Fitting Loss. (a) represents the points from the template P_h, blue points are from F_i, red points are outliers (value close to 1) and green points are inliers (value close to 0.5) from P_h. (b) represents the points from the F_i, blue points are from P_h, red points are outliers and green points are inliers from F_i.

Adding this loss function did not pose a significant performance improvement. This loss function increases the performance slightly in the BEV, however, it does not handle well the car's dimensions that are crucial for the 3D 0.7 IoU threshold.

7.4 Mask Appearence Loss

In Chapter 5, we highlighted our approach to not focus on the z-axis spatial location and the car's height. We relied on the Robust Spatial Location Estimator for the z-axis and simply calculated the height by measuring the difference between the car's lowest and highest points. We suspect that this approach might contribute to our method's

strong performance in achieving 0.7 BEV IoU and 0.5 3D IoU, but not quite reaching the performance at 0.7 3D IoU.

Optimizing the z-axis spatial location or car height using the Template Fitting Loss alone may not result in significant improvements. The loss is vulnerable to the variations in the upper parts of cars, which differ significantly across real cars in the wild.

To enhance our method, we want to utilize the binary masks provided by the Detectron2 framework [35]. By rendering a 3D template to generate a binary mask and aligning these masks, we can achieve a more precise match in both 2D and hopefully in 3D. This concept aligns with the techniques used by Zakharov et al. [24], as described in Section 2.3.2.

Our process begins with each predicted 7-DOF 3D bounding box, for which we prepare the hatchback template mesh T_h and the sedan template mesh T_s. These meshes are translated, rotated, and scaled to represent the predicted 3D bounding box perfectly. Each mesh is duplicated with a 180-degree rotation, resulting in four hypotheses per bounding box. The mesh creation and transformation are done within the Pytorch3D [48] Python library, as it keeps the gradients along the transformations.

These meshes are rendered using the PyTorch3D library, aiming to create a 2D binary mask. The Differentiable Soft Silhouette Shader [49] is ideal for this task, as it outputs the probability of mesh presence per pixel.

We utilize the KITTI dataset's camera calibration matrix [3] in our rendering process. However, probabilities of hard ones and zeros would not be suitable for backpropagation, so we need to add blur both to the rasterizer and shader employed within the PyTorch3D library. This approach slightly enlarges the binary mask, creating a smoother probability transition between one and zero.

The final stage involves calculating the loss for each of the four hypotheses, selecting the one with the lowest loss for backpropagation. We use per-pixel Binary Cross Entropy (BCE) Loss, comparing the rendered mask with the Detectron2 mask. Notably, backpropagation is only employed on the z-axis spatial location and the car's height. The entire process and its pipeline are illustrated in Figure 7.3.

One might wonder why this method isn't utilized to backpropagate the width and length of a car, in addition to the z-axis location and height. The primary challenge here is the inherent ambiguity in projecting 3D space into 2D space through rendering. As illustrated in Figure 7.3, the alignment of the masks is not perfect. Intuitively, one might consider adjusting the z-axis location, height, width, and length to better align the masks. However, this approach can lead to ambiguity.

As it can be seen in Figure 7.3, increasing all the dimensions slightly might seem like a solution in the 2D rendered mask, a slight decrease in length combined with an increase in width could also produce a matching 2D mask. Yet, the second proposal would result in an unrealistic representation in 3D space. This ambiguity is why we've chosen to restrict backpropagation to only the z-axis spatial location and the height of the car.

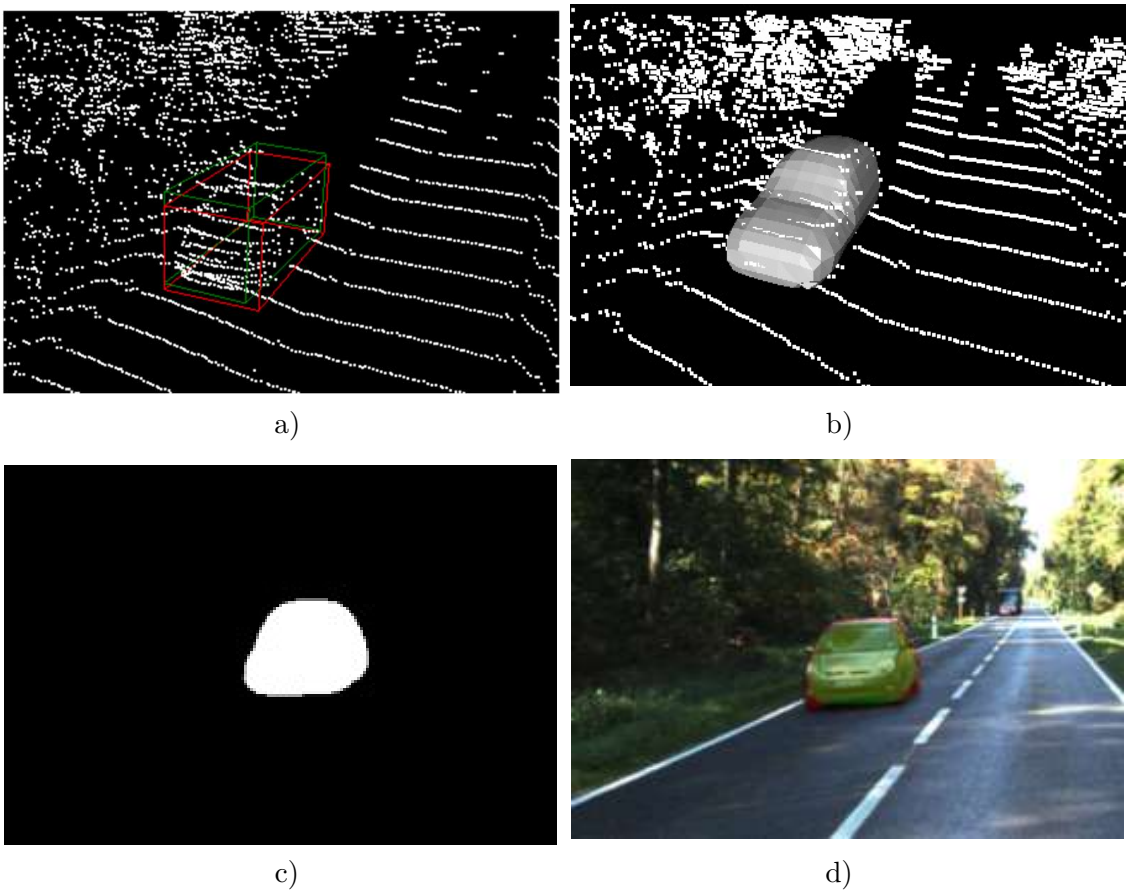


Figure 7.3. Example of the Mask Appearance Loss process. (a) 7-DOF 3D bounding box detection (red), 7-DOF 3D bounding box ground truth (green). (b) Template mesh representing the 3D bounding box. (c) Rendered 2D binary mask. (d) Alignment of the rendered binary mask (green) and the binary mask from Detectron2 [35] framework (red).

7.5 Data Augmentation

Data augmentation is an essential component in the training process that needs special attention. Initially, for easier debugging, we had chosen not to use data augmentation. This approach, however, led to Voxel-RCNN predicting zero yaw for all cars. It was caused by the lack of data augmentation.

The OpenPCDet [47] uses three types of augmentation. The entire scene is rotated around the z-axis, within a range of ± 45 degrees. Additionally, the scene is scaled by $\pm 10\%$ and the whole scene can be mirrored around the x-axis.

For the Template Fitting Loss, it is necessary to adjust the segmented (aggregated) point clouds $\hat{F}$ or F, due to the data augmentation. Fortunately, rotating, scaling, and mirroring a point cloud is a simple task.

However, integrating these augmentations with the Mask Appearance Loss is more complex, especially when dealing with binary masks provided by Detectron2 [35] framework, as it works only with the images, so the data augmentation would need to be first converted to image transformation. To address this, we use inverse Data Augmentation. Each 3D bounding box proposal is inversely rotated, scaled, and mirrored before applying the Mask Appearance Loss.

Implementing data augmentation, along with these modifications to adapt our losses, has significantly enhanced performance compared to training without data augmentation.

7.6 Training Process

First, we want to illustrate how is the fully-supervised version of the Voxel-RCNN [9] trained in Figure 7.4.

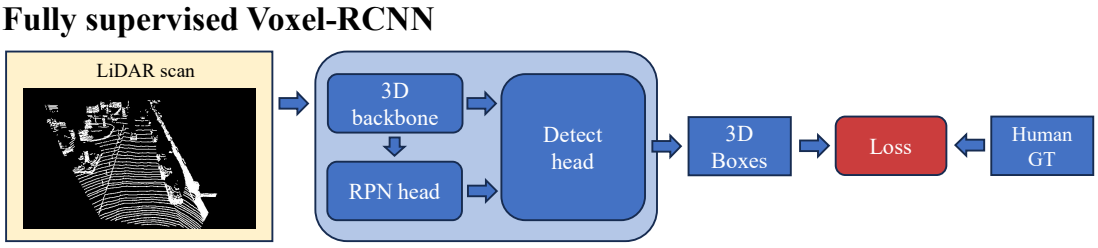


Figure 7.4. Visualization of the overall pipeline with the fully-supervised training.

The training process for our Voxel-RCNN implementation is structured into two phases. In the first phase, we train the Voxel-RCNN using standard losses and utilizing pseudo-ground truth labels. A key advantage of this initial phase is its speed and memory efficiency, allowing us to train the model with a relatively large batch size. The pipeline of the pretraining is shown in Figure 7.5.

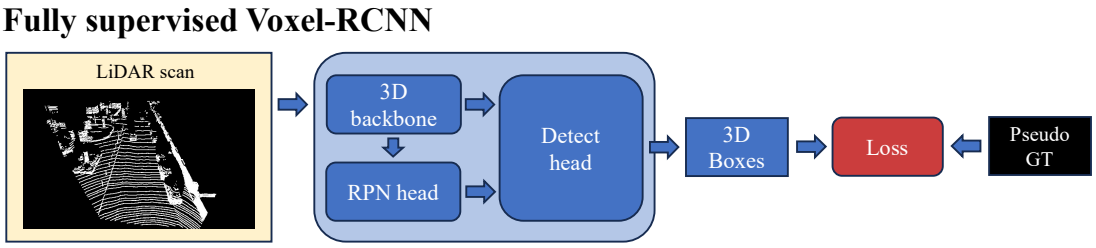


Figure 7.5. Visualization of the overall pipeline for the weakly-supervised pretraining.

The second phase is where we introduce the new losses while using the pre-trained weights from the first phase. As we utilize the Mask Appearance Loss during the second phase, which requires more memory resources, we need to reduce the batch size significantly. Additionally, due to the complexity of the new losses, the training time per epoch is significantly longer. Therefore, we also reduce the total number of epochs in this second stage of training. The whole weakly-supervised pipeline is shown in Figure 7.6.

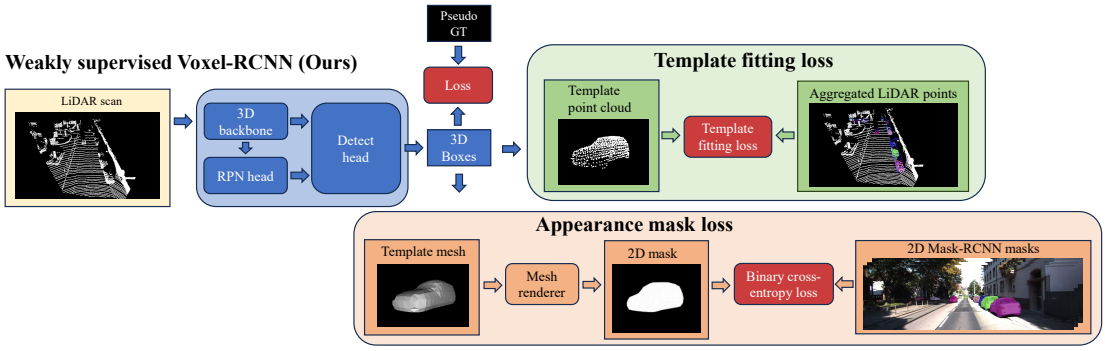


Figure 7.6. Visualization of the overall pipeline for the weakly-supervised training.

Chapter 8
Experiments

In this chapter, we present a range of experiments carried out to evaluate and refine our method. We start with a discussion on various evaluation methods used for parameter tuning, detailed in Section 8.1. Next, Section 8.2 outlines the implementation of our complete pipeline and the hardware employed.

In Section 8.3, we explore the impact of different 2D detection backbones on our method's performance. The selection process for the fitting threshold is then discussed in Section 8.4. Our attention shifts to the Histogram Yaw Estimation in Section 8.5 and the Occlusion Loss in Section 8.6. The effect of the fine fitting process is evaluated in Section 8.7, followed by an analysis of various downsampling methods in Section 8.8. The influence of the Scale Detector is discussed in Section 8.9.

We continue with an evaluation of using different numbers of adjacent frames in temporal consistency exploitation in Section 8.10, and the impact of the ICP algorithm in Section 8.11. We then discuss various Losses used in the fitting process in Section 8.12 and investigate the optimal steepness parameter value for the Differentiable Template Fitting Loss in Section 8.13.

Further, Section 8.14 focuses on the effect of integrating our proposed Losses into the training loop of Voxel-RCNN [9].

It is important to note that we conducted numerous experiments, yet many did not yield significant improvements or interesting findings, so they are not detailed here. Initially, we focus on evaluations that contributed to the development of our method. Towards the end, we shift our attention to the final pipeline, analyzing how certain parameter adjustments impact its overall performance.

8.1 Evaluation methods

Evaluating our models required a fast evaluation method. Initially, we considered the official evaluation script provided by the KITTI object detection dataset [3]. However, this script, written in C++, was not ideal as it sends evaluation results via email, so there is no immediate feedback.

To address this need for quicker evaluation, we have decided to use an unofficial Python adaptation of the KITTI evaluation script, sourced from a GitHub repository [50]. This Python script allows us to focus specifically on the car class, which is our research focus. It computes the average precision across three difficulty levels (Easy, Moderate, Hard) as defined in Subsection 2.1.1, covering metrics such as 2D, BEV (Bird's Eye View), 3D, and AOS (Angle of Orientation Similarity). Our primary focus, however, lies in the BEV and 3D evaluations. These metrics are calculated for two IoU thresholds: 0.7 and 0.5.

A notable challenge in our approach was the generation of pseudo ground truth labels without focus on output scores, which are essential for average precision evaluation. To resolve this, we utilized scores from the Detectron2 framework [35] outputs, corresponding to car instance detections in the reference frame. However, these scores did

not always accurately reflect the complexity of the fitting process for each car instance, leading to inconsistencies like an occluded distant car having the same score as a clearly visible one.

To enhance the evaluation process and align it more closely with our research goals, we modified the script to include recall calculations for each difficulty category. This adaptation provided a deeper understanding of our method's effectiveness, especially since the recall was a key focus in generating our pseudo ground truth labels.

In the early stages of our method's development, we also added precision calculations into the script. Precision is crucial, as a high rate of false positives can significantly decrease the overall average precision of the trained 3D detector. We conduct these recall and precision evaluations using the 0.7 BEV IoU metric, which is less challenging than the 0.7 3D IoU. This approach remained relevant until we developed the Scale Detector, described in Chapter 6, which enabled us to estimate car dimensions essential for achieving high scores in the 0.7 3D IoU metric.

It's important to note that evaluations of our pseudo ground truth labels are conducted on the entire training set. In contrast, for comparison with other weakly-supervised methods, evaluations of the 3D detector are performed on a 50/50 train/validation split of the training set.

8.2 Implementation and hardware used

The preprocessing pipeline of our project is structured into three key steps, with most computations being single-core loads. To optimize the processing speed, we divided the training dataset into smaller subsets (up to 200, as that is the maximum number of jobs). This division allowed us to process the subsets in parallel, leveraging the cluster's extensive core capacity, far surpassing that of a standard desktop PC. Without access to the RCI cluster, our research would take significantly more time and we couldn't achieve those results.

The first step involves calculations needed for the Iterative Closest Points algorithm [8]. This process is CPU-intensive and, when utilizing 100 cores, takes approximately 8 hours to compute frame-to-frame transformations for ± 120 frames for the whole training set, which contains 7481 frames. Computing the frame-to-frame transformations for ± 30 frames with 100 cores takes approximately 2 hours.

The second step in our pipeline makes use of the Detectron2 framework [35], requiring GPU capabilities, to get the segmented (aggregated) point clouds and trajectories exploiting the temporal consistency. The NVidia A100, available on the cluster, is employed for this purpose. Most of the computation time during this phase is dedicated to GPU tasks, ensuring efficient use of the resource. This phase, when processing ± 30 frames, takes around 3 hours with a single GPU employed. Of course, this process can be parallelized in the same manner as the first step to employ multiple GPUs.

The third phase of our preprocessing pipeline involves the fitting process. Similar to the first step, this phase is primarily a CPU load and is processed in parallel, as described earlier. Utilizing 100 CPU cores, this fitting process is completed in about 2 hours, however, it varies with the loss employed.

Once we've generated the pseudo ground truth labels, we proceed to pretrain the Voxel-RCNN [9]. This training is executed on 2x NVidia A100 GPUs, with a batch size set to 50 and 50 epochs. For this phase, we set a learning rate of 0.01 and a weight decay of 0.01. The training employs the Adam optimizer [51] together with the Cosine

Annealing learning rate scheduler, which includes a warm-up period of one epoch. The entire pretraining process is completed in approximately 2 hours.

The next step involves integrating our new loss functions into the training loop of the pre-trained model. Adding these losses significantly raises both the computational and memory demands, resulting in a needed reduction in the batch size to 8 and limiting the training to 10 epochs. We maintain a learning rate of 0.001 with the same weight decay of 0.01, using the Adam optimizer [51] and the Cosine Annealing learning rate scheduler with a one-epoch warmup. This training phase requires around 10 hours to complete.

8.3 Various 2D detection backbones

This section describes the influence of different 2D detection backbones on car detection performance. In the initial stages of our research, we conducted tests with multiple backbones and selected the *RegNetY* [36] backbone. This choice was based on its superior performance and efficiency on the 2D object detection KITTI dataset [3].

However, as our development progressed, we wanted to explore more models, mostly the larger models. To investigate this further, we experimented with the *ViT-H* [38] model, a significantly larger and more computationally demanding model than *RegNetY*. The *ViT-H* model, which is a Cascade Mask-RCNN model, achieves a 51.0% Mask AP on the MS-COCO dataset [7], in contrast to the 43.3% Mask AP achieved by *RegNetY*. This significant difference in model performance motivated us to explore the models further.

Additionally, we explored the Segment Anything (SAM) model [39], as detailed in Section 3.1. Our experiments with SAM aimed to provide tighter instance masks. We have generated pseudo ground truth labels using various 2D detection backbones. The results of these experiments, comparing the performance of various backbones, are shown in Table 8.1.

Backbone	SAM	BEV Recall			BEV Precision		
		Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
RegNetY	✗	43.88	40.30	35.29	49.86	60.92	53.90
ViT-H	✗	42.64	37.30	35.04	48.95	58.54	51.30
RegNetY	✓	43.98	39.35	35.21	50.25	60.12	54.31

Table 8.1. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different 2D Mask-RCNN [23] backbone or SAM.

The outcomes of our experiments indicate that using the Segment Anything (SAM) model does not yield a significant performance improvement. We noticed that the masks produced with SAM were more precise and tighter, however, the removal of some outliers by these improved masks did not enhance the overall performance of our method.

Surprisingly, our results also revealed that the *ViT-H* backbone, despite being a larger and more complex model, actually achieved lower precision and recall compared to *RegNetY*. This result was unexpected, given the higher Mask AP achieved on MS-COCO than *RegNetY*.

Based on these results, we have concluded that the *RegNetY* backbone is the most suitable choice for our needs. Its ability to provide better precision and recall, without the additional computational demands of larger models like ViT-H or the marginal improvements offered by SAM, makes it the optimal backbone.

8.4 Fitting threshold

In this section, we explore the impact of the fitting threshold on the performance of our method. or each segmented aggregated point cloud $\hat{F}_i$, which is a standing car, we require a minimum number of points, otherwise, we do not perform this fitting. We have generated pseudo ground truth labels for three different values of the threshold. The results are shown in Table 8.2.

Threshold	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
200	87.69	78.08	57.30	56.42	69.22	54.28
500	87.61	77.19	56.81	59.27	70.75	56.44
1000	87.64	75.52	54.23	63.52	72.53	58.49

Table 8.2. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different fitting thresholds.

From the results, it is evident that a lower threshold correlates with higher recall, while a higher threshold enhances precision. This trend aligns with our expectations. However, to find the optimal threshold, we need to consider the performance of a 3D detector trained with these thresholds. The outcomes of training the 3D detector using these varied thresholds are detailed in Table 8.3.

Number of frames	BEV			3D		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
200	87.97	84.97	82.94	57.18	48.24	48.55
500	87.66	84.6	82.38	58.26	48.38	48.47
1000	87.69	83.66	82.44	57.38	48.22	48.31

Table 8.3. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] trained with different fitting threshold.

The analysis of the fitting thresholds reveals that variations in the threshold do not lead to significant improvements in performance. Given these findings, we have decided to keep a fitting threshold of 1000 for our method.

8.5 Histogram Yaw Estimation

This section presents the outcomes of our Histogram Yaw Estimation method, as detailed in section 5.7. Here, we also discuss the impact of various loss functions used to measure histogram similarity. We have generated pseudo ground truth labels employing

Histogram Loss	BEV Recall		
	Easy @0.7	Moderate @0.7	Hard @0.7
✗	70.69	41.40	16.89
L1	55.00	38.43	14.48
L2	56.69	37.37	14.29
Bhattacharyya [46]	56.71	37.33	14.18

Table 8.4. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different fitting thresholds.

the Histogram Yaw Estimation. The results of these experiments are shown in Table 8.4.

Please note, that this evaluation comes from the early stages of development, so the results are quite poor and only the recall is computed.

The results indicate that employing the Histogram Yaw Estimation in our method leads to a decrease in overall performance, regardless of the loss function used for computing histogram similarities. Consequently, we have decided to avoid the Histogram Yaw Estimation as a research direction.

8.6 Occlusion loss

In this section, we discuss the impact of integrating Occlusion Loss with two distinct fitting losses: the Median Chamfer Distance Loss and the Template Fitting Loss, detailed respectively in Sections 5.6.2 and 5.6.3.

A challenge we encountered, as discussed in Section 5.6.4, comes from using a generic car shape for fitting. This generic template does not always align perfectly with the point cloud data, occasionally leading to overlaps and consequently, an increased Occlusion Loss. To resolve this issue, we experimented with calculating the Occlusion Loss only for the lower half of the car template. This approach should reduce the penalty from misalignments in the upper part of the template, which is more varied among different car models.

We have generated the pseudo ground truth labels and evaluated them in Table 8.5.

Loss	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
TFL no occlusion	43.81	40.34	35.37	49.76	60.97	54.02
TFL	43.48	40.13	31.78	49.89	60.77	48.22
TFL half	44.11	40.76	34.10	50.25	61.71	51.95
Median Chamfer	40.68	33.69	24.59	46.95	50.88	37.40
Median Chamfer half	43.26	37.57	28.25	49.70	56.93	42.90

Table 8.5. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different fitting losses employed together with occlusion loss.

The evaluation results indicate that Occlusion Loss does not significantly improve recall or precision in our method. While it shows some improvement in the Easy and

Moderate categories when combined with the Template Fitting Loss, it falls short in the Hard category. Therefore, we have opted to exclude Occlusion Loss from our approach.

8.7 Fine fitting

In this section, we examine the improvement in performance by the fine fitting process, which is employed after coarse fitting. The fine fitting is described in Section 5.3. We have generated pseudo ground truth labels with or without the fine fitting employed. To showcase the impact of this step, we present the results in Table 8.6.

Fitting	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
Coarse	87.00	73.03	49.62	71.59	72.35	60.12
Fine	87.54	73.29	49.74	72.04	72.64	60.27
Coarse 40*	87.96	74.79	50.74	72.32	74.12	61.5
Fine 40*	88.23	74.78	50.98	72.58	74.12	61.76

Table 8.6. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with or without fine fitting employed. * 40 denotes, that the number of steps for each parameter during fitting is increased from 20 to 40

Analyzing the results, we observe that the fine fitting process, especially when the number of steps for each parameter is increased from 20 to 40, does not result in a significant improvement. This outcome can be explained by the usage of the Scale Detector, which uses fine fitting as a prior estimate, so it does not significantly influence the final output. However, considering that the addition of the fine fitting process does not significantly increase computation time, we have chosen to keep it in our method.

8.8 Downsampling

This section focuses on the impact of various downsampling methods on our method's performance. The specifics of these downsampling methods are detailed in Section 4.4. To evaluate their effects, we generated pseudo ground truth labels using different downsampling methods. The comparative results are tabulated in Table 8.7.

Downsampling	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
Random	87.74	74.69	51.00	72.15	74.00	61.82
Voxel	88.18	74.89	50.24	72.54	74.24	60.85
Both	88.23	74.78	50.98	72.58	74.12	61.76

Table 8.7. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different downsampling methods employed.

Analysis of the results reveals that employing Random downsampling alone provides good performance, especially for Hard examples. On the other hand, Voxel downsampling, when used exclusively, shows enhanced performance for Easy targets. The most notable finding is that the combination of both Random and Voxel downsampling methods delivers the best overall recall and precision.

8.9 Scale Detector

This section evaluates the impact of the Scale Detector, as described in Chapter 6, on our method's performance. We first evaluate the pseudo ground truth labels generated with different configurations of the Scale Detector. The results are displayed in Table 8.8.

Scale Det.	Width	Reducer	Points	BEV Recall Easy @0.7	BEV Recall Moderate @0.7	BEV Recall Hard @0.7	BEV Precision Easy @0.7	BEV Precision Moderate @0.7	BEV Precision Hard @0.7
No	No	No	No	89.05	76.81	52.89	73.27	76.18	64.06
Yes	Dep	Yes	Agg	87.87	74.49	48.59	72.24	73.83	58.79
Yes	Dep	No	Agg	82.48	69.62	46.41	67.85	68.98	56.18
Yes	Dep	Yes, limit	Ref	88.29	75.78	51.81	72.65	75.12	62.77
Yes	Indep	Yes, limit	Agg	87.76	74.42	47.90	72.21	73.48	58.01
Yes	Dep	Yes, limit	Agg	88.23	74.78	50.98	72.58	74.12	61.76

Table 8.8. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with various settings of the Scale Detector.

Surprisingly, the results reveal that the Scale Detector does not enhance our method's recall or precision on the 0.7 IoU BEV. Despite trying various configurations, the performance remains the same as using average KITTI car dimensions for each car.

To examine this situation more deeply, we trained the Voxel-RCNN [9] with these pseudo ground truth labels to determine if the Scale Detector might boost the 3D detector's performance. We think that the Scale Detector adds more information into the training loop, as the 3D detector would otherwise not learn to estimate car dimensions due to the lack of variation in spatial dimensions without the Scale Detector.

The trained Voxel-RCNN's performance is shown in Table 8.9.

Scale Det.	Width	Reducer	Points	BEV Easy @0.7	BEV Moderate @0.7	BEV Hard @0.7	3D Easy @0.7	3D Moderate @0.7	3D Hard @0.7
No	No	No	No	88.99	86.37	83.97	61.38	53.47	52.82
Yes	Dep	Yes	Agg	89.74	87.18	84.40	78.61	65.12	62.51
Yes	Dep	No	Agg	88.90	85.73	78.59	63.16	54.66	52.91
Yes	Dep	Yes, limit	Ref	90.00	87.47	84.53	79.45	65.25	62.95
Yes	Indep	Yes, limit	Agg	89.76	86.95	84.17	79.69	65.86	62.19
Yes	Dep	Yes, limit	Agg	89.81	87.13	84.31	78.37	65.06	63.71

Table 8.9. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] with various settings of the Scale Detector.

Examining the BEV results, it's apparent that the Scale Detector's settings minimally affect the BEV average precision. However, one specific setting, where the Bounding Box Reducer is not applied, results in significantly lower BEV average precision, as the bounding boxes tend to be large.

In contrast, the 3D average precision shows more promising results. The implementation of the Scale Detector significantly improves the 3D average precision. The poorest 3D average precision occurs again when the Bounding Box Reducer is not used. The optimal configuration, which employs the Scale Detector with dependent width and length, Bounding Box Reducer with limited reduction, and aggregated points for reduction, achieves the highest average precision.

Figure 8.1 shows the usage of the Scale Detector for the pseudo ground truth label generation and also for the Voxel-RCNN [9] training. This figure also shows how the training can recover from wrong pseudo ground truth labels.

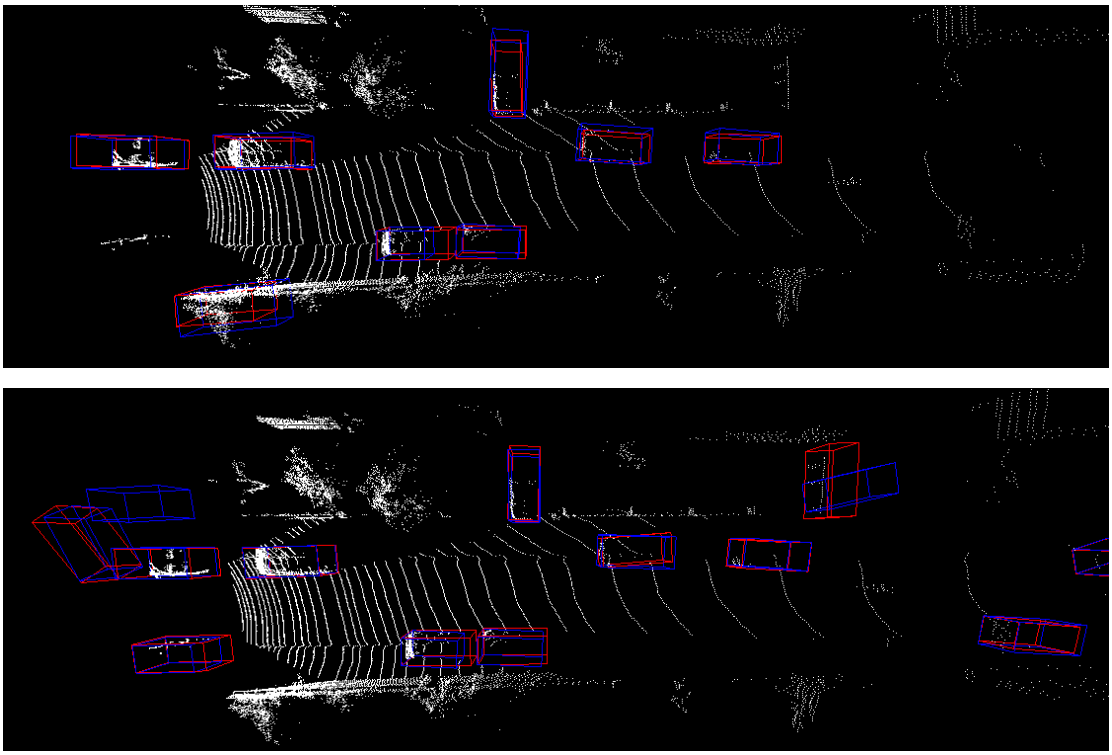


Figure 8.1. Example of the usage of the Scale Detector. The upper scene shows the comparison of using the Scale Detector for the pseudo ground truth label generation. Blue bounding boxes are with the Scale Detector, Red without. The lower scene shows the comparison of using the pseudo ground truth labels with Scale Detector to train Voxel-RCNN [9]. Blue bounding boxes are with the Scale Detector, Red without.

8.10 Number of frames

This section investigates how the number of frames utilized in temporal consistency exploitation influences the recall, precision, and average precision of our method. We experimented with four different numbers of frames. The temporal consistency exploitation is described in Chapter 4.

First, ± 0 frames which means there is no tracking employed. Consequently, all cars are treated as standing, and there is no point aggregation or yaw estimation based on trajectory.

Second, ±10 frames, correspond to ±1 second before and after the reference frames. The tracking is employed, as well as the classification of standing or moving cars, which allows point aggregation or yaw estimation.

The third and fourth settings are ±30 and ±120 frames, which translates into ±3, respectively ±12 seconds. These settings differ from the ±10 frames settings only by the time span.

An extended time span requires processing more images with the Detectron2 [35] framework, making a very long time span computationally challenging and inefficient.

Let us examine the pseudo ground truth labels based on the 0.7 IoU BEV metric. The results are presented in Table 8.10.

Number of frames	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
±0	88.46	69.08	34.84	76.60	70.06	49.47
±10	86.33	75.94	49.43	74.10	74.50	62.31
±30	87.66	74.84	51.05	72.11	74.20	61.85
±120	86.47	70.34	47.43	68.84	74.68	59.63

Table 8.10. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with different numbers of frames for temporal consistency exploitation.

The results demonstrate that exploiting temporal consistency improves recall and precision, particularly for Moderate and Hard examples. However, extending to ±120 frames results in reduced recall and precision. Consequently, we have decided to use the ±30 frames setting, which offers the highest recall for Hard examples and, along with ±10 frames, delivers the best overall recall and precision.

A slight decline in recall and precision for Easy examples is observed with an increased number of frames. This is probably caused by that when we can see more time in the future, we can detect cars at bigger distances. However, these distant cars are often not labeled in the dataset, leading to a decrease in recall and precision.

Next, we utilized the pseudo ground truth labels, along with segmented (aggregated) point clouds, trajectories, and binary masks, into our complete pipeline for weakly-supervised training. The performance of this approach was evaluated by training the Voxel-RCNN [9] and evaluating it in both Bird's Eye View (BEV) and 3D metrics. The results of this evaluation, based on different numbers of frames used for temporal consistency exploitation, are presented in Table 8.11.

Number of frames	BEV			3D		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
±0	88.83	84.7	78.11	53.70	45.17	41.26
±10	89.70	87.71	86.75	83.77	68.05	67.13
±30	90.09	88.25	86.95	85.92	75.33	73.74

Table 8.11. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] with different number of frames for temporal consistency exploitation.

The results demonstrate that exploiting temporal consistency is an essential component of our method. It significantly boosts performance, particularly in 3D object detection. Additionally, there is a noticeable improvement in performance when increasing the frame number from ±10 to ±30. These findings show the importance of temporal consistency in enhancing the average precision of 3D object detection in our weakly-supervised training framework.

8.11 Iterative Closest Points

In this section, we explore the impact of the Iterative Closest Points (ICP) algorithm [8] on our method's performance. As outlined in Chapter 4, the provided IMU data does not always ensure perfect alignment in frame-to-frame transformations. We implement the ICP algorithm for fine-tuning these transformations.

Our initial evaluation focuses on the recall and precision of our pseudo ground truth labels, specifically using the 0.7 IoU BEV metric and considering ±30 frames. The results of this evaluation are summarized in Table 8.12.

ICP	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
✗	80.96	63.48	43.71	66.52	62.67	52.62
✓	87.66	74.84	51.05	72.11	74.20	61.85

Table 8.12. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with or without the Iterative Closest Point [8] algorithm employed.

The results clearly show that utilizing the ICP algorithm in the temporal consistency exploitation significantly enhances our method's performance. Throughout our experiments, we observed that employing ICP enables the use of a larger number of frames, which is essential for our method. For instance, without the use of ICP, choosing a larger frame window, such as ±30 frames instead of ±10 frames, results in a significant drop in both recall and precision. The utilization of ICP addresses this issue, showing its importance in the overall method.

Next, we utilized the pseudo ground truth labels, along with segmented (aggregated) point clouds, trajectories, and binary masks, into our complete pipeline for weakly-supervised training. The performance of this approach was evaluated by training the Voxel-RCNN [9] and evaluating it in both Bird's Eye View (BEV) and 3D metrics. The results are presented in 8.13.

ICP	BEV			3D		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
✗	89.80	87.56	86.34	74.78	70.04	64.61
✓	90.09	88.25	86.95	85.92	75.33	73.74

Table 8.13. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] with or without the Iterative Closest Point [8] algorithm employed.

Our findings show that while the ICP algorithm doesn't lead to significant changes in the BEV performance metric, it notably enhances performance in the 3D metric. This improvement in 3D can be explained by the near-perfect alignment of point clouds achieved through the ICP. It is particularly beneficial in a 3D metric, where achieving the 0.7 IoU threshold is harder than in the BEV.

8.12 Loss Function Comparison

This section focuses on the different losses employed in the fitting process within our method. First, we evaluate various losses for the pseudo ground truth label generation The evaluation of pseudo ground truth labels is shown in Table 8.14.

Loss	Recall			Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
Chamfer	56.26	37.96	22.20	46.43	37.65	26.72
Median Chamfer	82.51	67.68	43.18	67.94	51.40	52.29
Template Fitting	87.54	73.29	49.74	72.04	72.64	60.27
Template Fitting 40*	88.23	74.78	50.98	72.59	74.12	61.76
Differentiable TF	85.95	72.55	46.73	70.68	71.94	56.54

Table 8.14. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with various fitting losses employed. * 40 denotes, that the number of steps for each parameter during fitting is increased from 20 to 40.

The evaluation reveals that the Chamfer Distance Loss, despite its widespread use, delivers suboptimal performance, primarily due to its inadequate handling of outliers as described in Subsection 5.6.1. On the other hand, the Median Chamfer Distance Loss, described in Subsection 5.6.2, significantly enhances both recall and precision. Yet, it's the Template Fitting Loss, described in Subsection 5.6.3, that delivers the best performance, surpassing the Median Chamfer Distance Loss significantly in both metrics.

Notably, the fast implementation of the Template Fitting Loss allows us to increase the number of steps in coarse optimization from 20 to 40, further boosting performance. The comparison between the non-differentiable and differentiable versions of the Template Fitting Loss shows a relatively small performance gap, proving that this approximation is good enough for our method.

As the Median Chamfer Distance Loss and Template Fitting Loss aren't differentiable, we cannot use them in the whole pipeline for weakly-supervised training. So they are omitted in the next table showing the performance difference with various losses employed in the whole weakly-supervised training shown in Table 8.15. The Mask Fitting Loss described in Section 7.4 is employed in both cases.

This analysis reveals that the Differentiable Template Fitting Loss, described in Section 7.3, significantly enhances the performance in both the 0.7 BEV IoU and 0.7 3D IoU metrics. The steepness parameter σ is chosen 10 as described in Section 8.13. On the other hand the Chamfer Distance Loss, despite its wide usage, provides poor performance compared to the Differentiable Template Fitting Loss.

Loss	BEV			3D		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
Chamfer	84.02	73.11	66.39	67.21	53.93	51.37
Differentiable TF	90.09	88.25	86.95	85.92	75.33	73.74

Table 8.15. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] with various losses employed.

8.13 Steepness parameter in Differentiable Template Fitting Loss

This section examines the influence of the steepness parameter σ in the Differentiable Template Fitting Loss, which is an essential setting of the loss. As detailed in Section 8.12, this loss has demonstrated superior results. However, it contains the steepness parameter σ in the sigmoid function that must be tuned.

To determine the optimal steepness parameter, we generated various pseudo ground truth labels with different values of σ. The performance of each setting was evaluated using the 0.7 IoU BEV metric, with the results shown in Table 8.16. For context, these results are compared against the non-differentiable Template Fitting Loss.

σ	BEV Recall			BEV Precision		
	Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
5	86.59	75.37	52.32	62.80	72.33	56.33
10	85.88	74.48	52.44	62.26	71.51	56.47
15	84.73	72.96	50.75	61.41	70.06	54.64
25	83.25	69.36	46.65	60.27	66.59	50.21
TFL	86.43	74.29	53.39	64.65	71.33	57.57

Table 8.16. Evaluation of the pseudo ground truth labels on the whole BEV object detection KITTI dataset [3] with various steepness parameters σ employed.

Two key insights come from this analysis. Firstly, the Differentiable Template Fitting Loss demonstrates a performance comparable to the Template Fitting Loss. Secondly, the optimal value for the steepness parameter σ appears to lie within the range of 5 to 10. Considering these findings, we have decided to use a σ value of 10, which balances the trade-off between sensitivity and performance.

8.14 Losses in the training loop

This section describes the influence of integrating additional losses into the weakly-supervised training pipeline on the performance of a 3D detector. We start by using a pre-trained model on pseudo ground truth labels without any added losses, basically continuing in the pretraining phase. We then incrementally add the Differentiable Template Fitting Loss (DTFL) and Mask Appearance Loss (MAL) to the training

DTFL	MAL	BEV			3D		
		Easy @0.7	Moderate @0.7	Hard @0.7	Easy @0.7	Moderate @0.7	Hard @0.7
✗	✗	89.64	87.55	85.48	78.42	64.77	63.64
✓	✗	90.12	88.15	86.99	76.42	66.01	65.00
✗	✓	90.09	88.05	86.64	85.35	74.54	67.67
✓	✓	90.09	88.25	86.95	85.92	75.33	73.74

Table 8.17. Evaluation of the weakly-supervised Voxel-RCNN [9] on the validation object detection KITTI dataset [3] with various added losses employed.

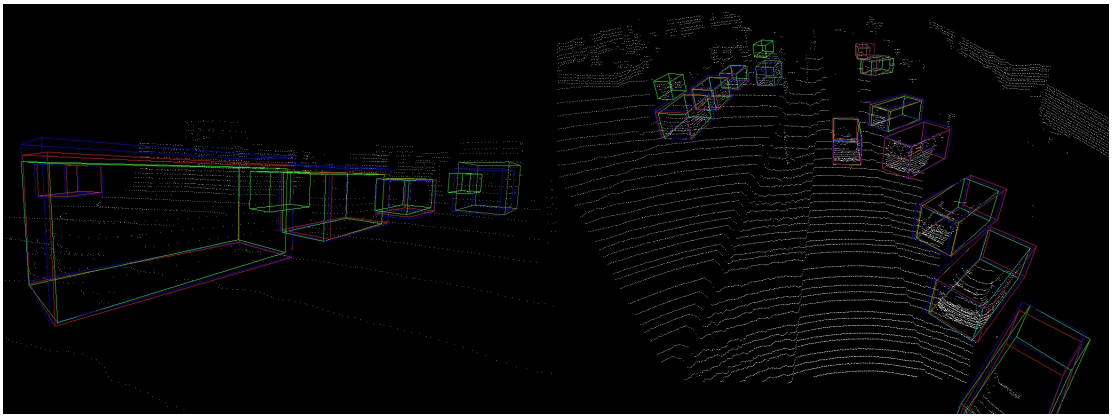
loop, analyzing their individual and combined effects on the performance. The results of this evaluation are shown in Table 8.17.

First, the results show that adding only DTFL, described in Section 7.3, provides a marginal performance increase. Since DTFL was already utilized in generating pseudo ground truth labels, it adds limited additional information to the training process. However, the average precision is still improved.

Second, we can see that the introduction of MAL, described in Section 7.4, significantly improves performance, especially in the evaluation with a 3D 0.7 IoU threshold. As this loss is not used for generating the pseudo ground truth labels, it offers additional information for the training, which is beneficial in terms of improving performance.

Third, the results show that employing both losses together achieves the highest average precision, significantly enhancing 3D detection performance while also slightly improving BEV results.

Further, we want to illustrate the enhancement that the addition of new losses adds to the method. It is shown in Figure 8.2. It can be seen that predictions proposed by Voxel-RCNN, trained with or without added losses, are a little bit different. The main enhancement lies in the correct prediction of the 3D bounding box height and also the z-axis spatial location.



a) b)

Figure 8.2. Examples of 3D object detections on the KITTI dataset. Detections by our method with added losses are in red, without the added losses are in blue, and ground truth is in green. (a) Examples of how is the height prediction improved, (b) the same scene to show the minor differences.

Chapter 9
Evaluation

In this Chapter, we first compare our method's average precision both with the weakly-supervised and fully-supervised in Section 9.1. Further, we provide a few scenes, where we can compare the quality of our methods labels to the human annotations in Section 9.2.

9.1 Comparison with the State-of-the-art

This section presents a comparison of our method against other weakly-supervised 3D detection approaches, as detailed in Section 2.3. We divide the methods into two distinct categories. The ones that do not use any human labels, and where our method falls. And the ones that use some subset of the human labels.

The comparison is based on the performance metrics obtained on the validation subset of the KITTI object detection dataset [3]. The results are shown in Table 9.1. This comparison is essential for understanding the strengths and weaknesses of both categories and how they can overlap.

		KITTI labels		BEV AP						3D BEV					
				Easy		Moderate		Hard		Easy		Moderate		Hard	
Method	Year	2D	3D	@0.5	@0.7	@0.5	@0.7	@0.5	@0.7	@0.5	@0.7	@0.5	@0.7	@0.5	@0.7
		Fully-supervised methods													
PV-RCNN [16]	2020	Yes	Yes	×	×	×	×	×	×	×	89.35	×	83.69	×	78.70
Voxel-RCNN [9]	2021	Yes	Yes	×	×	×	×	×	×	×	89.41	×	84.52	×	78.93
CasA+T [18]	2021	Yes	Yes	×	×	×	×	×	×	×	*90.11*	×	*86.63*	×	*79.49*
		Weakly-supervised methods with partial human labels													
WS3D [52]	2020	No	500	96.33	88.56	89.01	84.99	88.52	84.74	95.85	84.04	89.14	75.10	88.32	73.29
FGR [27]	2021	Yes	No	×	×	×	×	×	×	×	86.11	×	74.86	×	67.53
WS3D v2 [6]	2021	No	500	96.46	88.95	89.35	85.83	88.97	85.03	96.34	85.04	89.44	75.94	88.95	74.38
MAP-Gen [30]	2022	Yes	500	×	×	×	×	×	×	×	87.87	×	77.98	×	76.18
Mtrans [32]	2022	Yes	500	×	×	×	×	×	×	×	88.72	×	78.84	×	77.43
		Weakly-supervised methods with no human labels													
VS3D [19]	2020	No	No	81.60	×	72.43	×	64.31	×	41.83	×	39.22	×	32.73	×
Zakharov [24]	2020	No	No	94.9	81.0	88.5	59.8	×	×	90.7	22.4	71.1	13.3	×	×
McCraith [26]	2022	No	No	90.23	×	85.74	×	76.84	×	×	×	×	×	×	×
Ours		No	No	*98.86*	*90.09*	*89.58*	*88.25*	*89.06*	*86.95*	*98.81*	85.92	*89.54*	75.33	*88.97*	73.74

Table 9.1. Comparison of ours, other weakly-supervised and fully-supervised evaluated on the validation object detection KITTI dataset [3].

First, examining the BEV results, it's evident that our method achieves top-tier performance in both 0.5 and 0.7 BEV IoU metrics on the validation KITTI dataset compared to other weakly-supervised methods. Notably, our method outperforms all other approaches that do not use human labels by a significant margin. It also surpasses those methods that employ a subset of human labels, although some of these methods haven't published their BEV metrics results.

Second, looking at the 3D results, our method demonstrates state-of-the-art performance at the 0.5 3D IoU metrics on the validation KITTI dataset among weakly-supervised methods. While it doesn't lead in the 0.7 3D IoU metrics, the performance gap is below 5%, and those leading methods use a subset of human labels. Our method significantly outperforms other methods that do not use human labels and shows a small performance gap compared to state-of-the-art fully-supervised methods.

Next, we evaluate our method on the test object detection KITTI dataset and compare it with other methods. These results are shown in Table 9.2. We focus on the 0.7 IoU threshold as it is the standard for the test set evaluation.

		KITTI labels		BEV AP			3D BEV		
				Easy	Moderate	Hard	Easy	Moderate	Hard
Method	Year	2D	3D	@0.7	@0.7	@0.7	@0.7	@0.7	@0.7
		Fully-supervised methods							
PV-RCNN [16]	2020	Yes	Yes	94.98	90.65	86.14	90.25	81.43	76.82
Voxel-RCNN [9]	2021	Yes	Yes	94.85	88.83	86.13	90.90	81.62	77.06
CasA+T [18]	2021	Yes	Yes	*94.57*	*91.22*	*88.43*	*90.68*	*84.04*	*79.69*
		Weakly-supervised methods with partial human labels							
WS3D [52]	2020	No	500	90.11	84.02	76.97	80.15	69.64	63.7
FGR [27]	2021	Yes	No	90.64	82.67	75.46	80.26	68.47	61.57
WS3D v2 [6]	2021	No	500	90.96	84.93	77.96	80.99	70.59	64.23
MAP-Gen [30]	2022	Yes	500	90.61	85.91	80.58	81.51	74.14	67.55
Mtrans [32]	2022	Yes	500	91.42	85.96	78.82	83.42	75.07	68.26
		Weakly-supervised methods with no human labels							
Ours		No	No	91.20	85.13	80.15	77.76	65.41	60.90

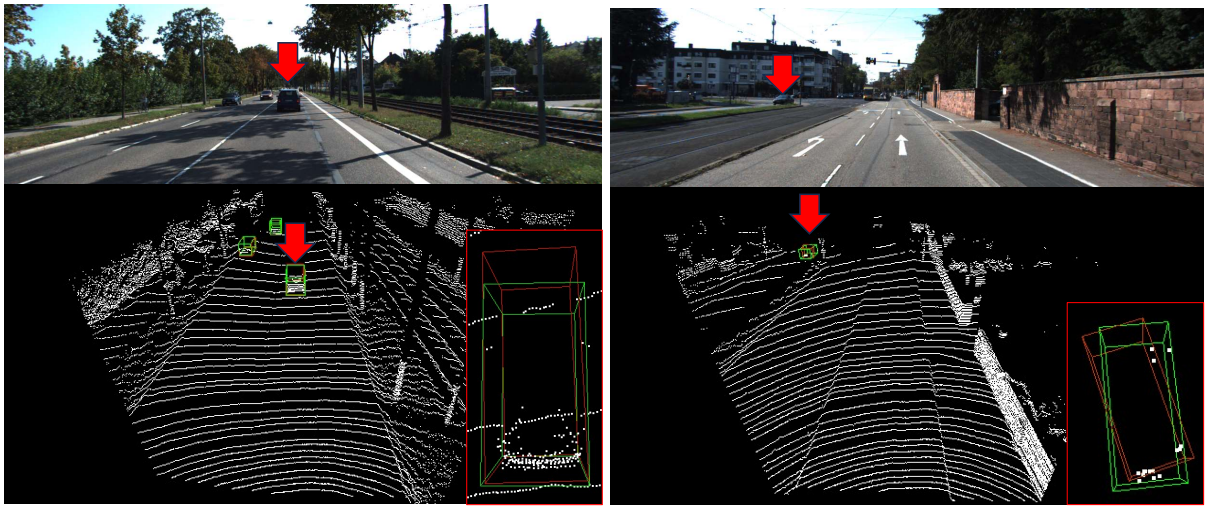
Table 9.2. Comparison of ours, other weakly-supervised and fully-supervised evaluated on the test object detection KITTI dataset [3].

Reviewing the 0.7 BEV IoU metric on the test set, our method is competitive with other state-of-the-art weakly-supervised approaches, especially those using a subset of human labels. Unfortunately, no methods using no human labels have published their results on the test set. Compared to fully-supervised methods, there's a noticeable but small performance difference.

In the 0.7 3D IoU metric, our method closely follows other approaches but doesn't outperform any. Again we can't compare our method with other methods using no human labels. The gap with fully-supervised methods is significant, and it's worth noting that our method's performance drop between the validation and test sets is larger than in other methods.

9.2 Qualitative comparison to human annotations

In this section, we show frames where we employ our method and compare the outputs of our method with the human-labeled ground truth labels. First, in Figure 9.1 we show scenes where our method fails. Second, in Figure 9.2, we show scenes, where our method performs well.



a) b)

Figure 9.1. Typical failure modes on the KITTI dataset. Estimating length of a car which is moving and has the same yaw as the ego-vehicle is extremely difficult as there is no data even in subsequent frames to infer vehicle length (a). A vehicle in the Hard category (car) has very sparse LiDAR point cloud and since it is a moving car in opposite direction, we cannot aggregate enough LiDAR points for this instance (b).

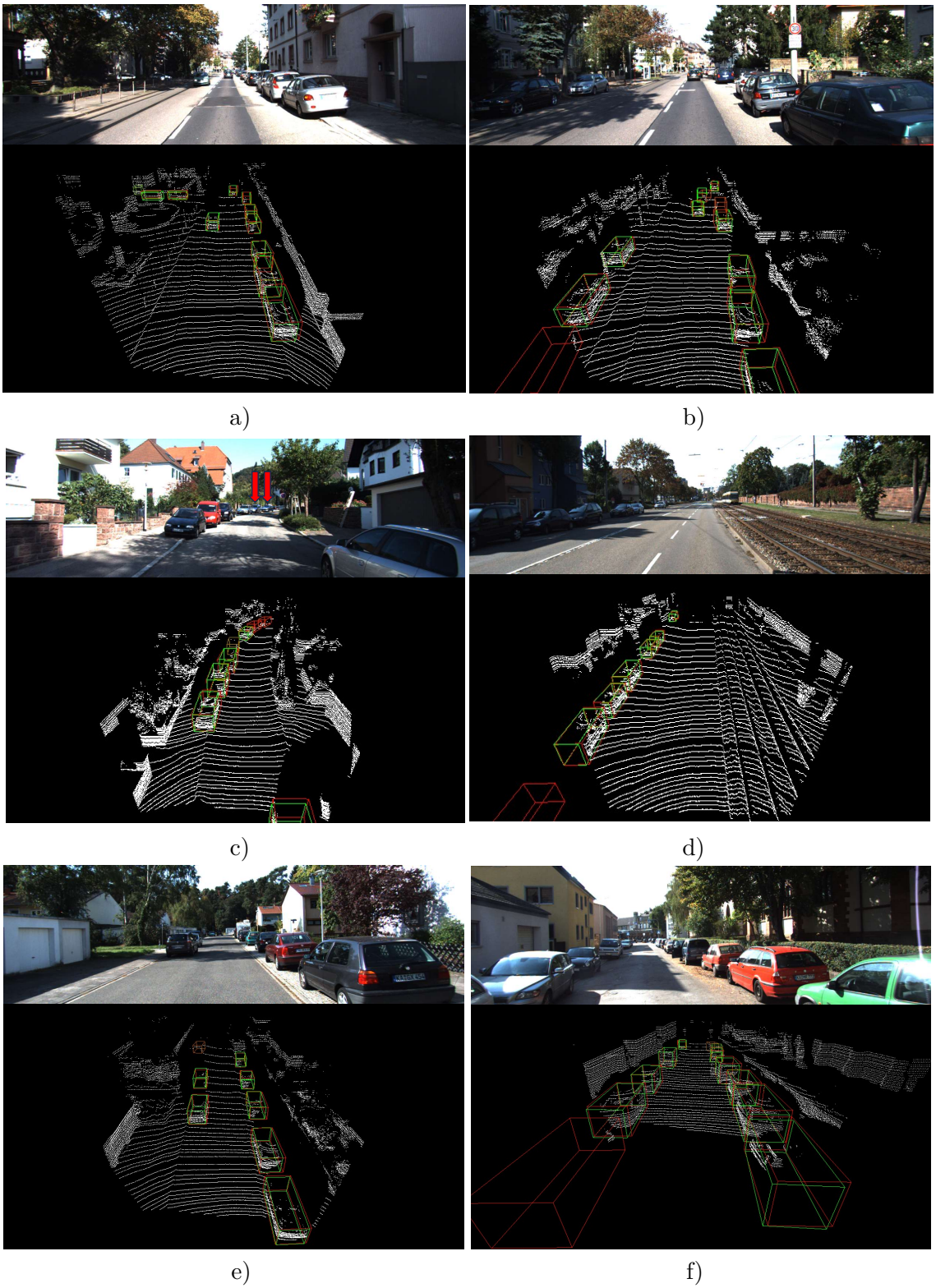


Figure 9.2. Examples of 3D object detections on the KITTI dataset. Detections by our method are in red color, ground truth is in green color. Note the two detections in c) marked with a red arrow which are cars missed by human annotators of KITTI. Best viewed zoomed in.

Chapter 10
Conclusion

In this thesis, we have analyzed the current state of 3D vehicle detection, focusing on datasets and methods in both fully-supervised and weakly-supervised domains, as detailed in Chapter 2. This analysis has been used to gather insights into the problem of weak supervision.

Further, a new method which exploits a generic off-the-shelf 2D detector and a number of real-world priors to train a 3D object detector was proposed. Exploitation of the temporal consistency, such as car tracking or point aggregation, plays a significant role in the method, together with a novel Template Fitting Loss used in the fitting process. The method has been described in Chapters 3, 4, 5, and 6.

Furthermore, in Chapter 7, we show how a fully-supervised 3D detector pipeline can be adapted to a weakly-supervised one to increase the method's performance by adding both novel Differentiable Template Fitting and Mask Appearance Loss. The addition of novel losses incorporates more information into the training loop in the form of segmented aggregated point clouds, car trajectories, and 2D instance segmentation binary masks.

The experiments performed during the development of the method are discussed and evaluated in Chapter 8 to support our decisions during the development of the method. Numerous experiments were possible thanks to the granted access to the RCI cluster. The results are proposed in Chapter 9, where the comparison both with the weakly and fully supervised methods on the KITTI dataset is shown and discussed together with an illustration of the method's performance on real-world scenes.

One of the most significant advantages of our method is that it can be used to train any 3D detector by only collecting sensor recordings in the real world, which is extremely cheap and allows training using orders of magnitude more data than traditional fully-supervised methods.

Our method *significantly outperforms all previous methods which do not rely on domain-specific human labels*, thus achieving state-of-the-art accuracy in 3D object detection trained without human annotations. In Bird's Eye View (BEV) AP, our method also outperforms methods which rely on partial 2D or 3D KITTI [3] annotations, and in 3D AP it achieves similar results, despite not having access to any 3D human labels.

The main limitation of our method seems to be the inability to account for some annotation bias, which is demonstrated by a smaller gap to the fully-supervised method in the less strict overlap evaluation (IoU threshold 0.5). Also, by improving the spatial dimensions estimation, the method's 0.7 3D IoU average precision would increase significantly, however, the estimation of spatial dimensions for a special subset of cars is challenging, even for the human annotator.

References

[1] Xinyu Huang, Peng Wang, Xinjing Cheng, Dingfu Zhou, Qichuan Geng, and Ruigang Yang. The apolloscape open dataset for autonomous driving and its application. *IEEE transactions on pattern analysis and machine intelligence*. 2019, 42 (10), 2702–2719.

[2] Shuran Song, Samuel P Lichtenberg, and Jianxiong Xiao. *Sun rgb-d: A rgb-d scene understanding benchmark suite*. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015. 567–576.

[3] Andreas Geiger, Philip Lenz, and Raquel Urtasun. *Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite*. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[4] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

[5] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A multimodal dataset for autonomous driving. *arXiv preprint arXiv:1903.11027*. 2019.

[6] Qinghao Meng, Wenguan Wang, Tianfei Zhou, Jianbing Shen, Yunde Jia, and Luc Van Gool. Towards a weakly supervised framework for 3d point cloud object detection and annotation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2021, 44 (8), 4454–4468.

[7] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll'ar, and C Lawrence Zitnick. *Microsoft COCO: Common objects in context*. In: *ECCV*. 2014. 740–755.

[8] Francois Pomerleau, Francis Colas, Roland Siegwart, and others. A review of point cloud registration algorithms for mobile robotics. *Foundations and Trends in Robotics*. 2015, 4 (1), 1–104.

[9] Jiajun Deng, Shaoshuai Shi, Peiwei Li, Wengang Zhou, Yanyong Zhang, and Houqiang Li. *Voxel r-cnn: Towards high performance voxel-based 3d object detection*. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2021. 1201–1209.

[10] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*. 2013.

[11] Scott Ettinger, Shuyang Cheng, Benjamin Caine, Chenxi Liu, Hang Zhao, Sabeek Pradhan, Yuning Chai, Ben Sapp, Charles R. Qi, Yin Zhou, Zoey

Yang, Aur'elien Chouard, Pei Sun, Jiquan Ngiam, Vijay Vasudevan, Alexander McCauley, Jonathon Shlens, and Dragomir Anguelov. *Large Scale Interactive Motion Forecasting for Autonomous Driving: The Waymo Open Motion Dataset*. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021. 9710-9719.

[12] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. *Pointnet: Deep learning on point sets for 3d classification and segmentation*. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017. 652–660.

[13] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*. 2017, 30

[14] Yin Zhou, and Oncel Tuzel. *Voxelnet: End-to-end learning for point cloud based 3d object detection*. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018. 4490–4499.

[15] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. *Pointpillars: Fast encoders for object detection from point clouds*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019. 12697–12705.

[16] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. *Pv-rcnn: Point-voxel feature set abstraction for 3d object detection*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020. 10529–10538.

[17] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, and others. *Scalability in perception for autonomous driving: Waymo open dataset*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020. 2446–2454.

[18] Hai Wu, Jinhao Deng, Chenglu Wen, Xin Li, Cheng Wang, and Jonathan Li. CasA: A cascade attention network for 3-D object detection from LiDAR point clouds. *IEEE Transactions on Geoscience and Remote Sensing*. 2022, 60 1–11.

[19] Zengyi Qin, Jinglu Wang, and Yan Lu. *Weakly supervised 3d object detection from point clouds*. In: *Proceedings of the 28th ACM International Conference on Multimedia*. 2020. 4144–4152.

[20] Marcelo Bertalmio, Guillermo Sapiro, Vincent Caselles, and Coloma Ballester. *Image inpainting*. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000. 417–424.

[21] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*. 2010, 88 (2), 303–338.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. *Imagenet: A large-scale hierarchical image database*. In: *2009 IEEE conference on computer vision and pattern recognition*. 2009. 248–255.

[23] Kaiming He, Georgia Gkioxari, Piotr Dollr, and Ross Girshick. *Mask r-cnn*. In: *Proceedings of the IEEE international conference on computer vision*. 2017. 2961–2969.

[24] Sergey Zakharov, Wadim Kehl, Arjun Bhargava, and Adrien Gaidon. *Autolabeling 3d objects with differentiable rendering of sdf shape priors*. In: *Proceedings of*

the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020. 12224–12233.

[25] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. *Deepsdf: Learning continuous signed distance functions for shape representation*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019. 165–174.

[26] Robert McCraith, Eldar Insafutdinov, Lukas Neumann, and Andrea Vedaldi. *Lifting 2d object locations to 3d by discounting lidar outliers across objects and views*. In: *2022 International Conference on Robotics and Automation (ICRA)*. 2022. 2411–2418.

[27] Yi Wei, Shang Su, Jiwen Lu, and Jie Zhou. *Fgr: Frustum-aware geometric reasoning for weakly supervised 3d vehicle detection*. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021. 4348–4354.

[28] Martin A Fischler, and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*. 1981, 24 (6), 381–395.

[29] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. *Pointrcnn: 3d object proposal generation and detection from point cloud*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019. 770–779.

[30] Chang Liu, Xiaoyan Qian, Xiaojuan Qi, Edmund Y Lam, Siew-Chong Tan, and Ngai Wong. *MAP-Gen: An Automated 3D-Box Annotation Flow with Multimodal Attention Point Generator*. In: *2022 26th International Conference on Pattern Recognition (ICPR)*. 2022. 1148–1155.

[31] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. *Pyramid scene parsing network*. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017. 2881–2890.

[32] Chang Liu, Xiaoyan Qian, Binxiao Huang, Xiaojuan Qi, Edmund Lam, Siew-Chong Tan, and Ngai Wong. *Multimodal Transformer for Automatic 3D Annotation and Object Detection*. In: *European Conference on Computer Vision*. 2022. 657–673.

[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. 2018.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*. 2017, 30

[35] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. *Detectron2*.
`https://github.com/facebookresearch/detectron2`. 2019.

[36] *RegNetY configuration file*.
`https://github.com/facebookresearch/detectron2/blob/main/configs/new_baselines/mask_rcnn_regnety_4gf_dds_FPN_400ep_LSJ.py`. Accessed: 2023-12-15.

[37] Golnaz Ghiasi, Yin Cui, Aravind Srinivas, Rui Qian, Tsung-Yi Lin, Ekin D Cubuk, Quoc V Le, and Barret Zoph. *Simple copy-paste is a strong data augmentation method for instance segmentation*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021. 2918–2928.

[38] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring plain vision transformer backbones for object detection. *arXiv preprint arXiv:2203.16527*. 2022.

[39] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, and others. Segment anything. *arXiv preprint arXiv:2304.02643*. 2023.

[40] utiasSTARS. *pykitti*.
`https://github.com/utiasSTARS/pykitti`. 2019.

[41] *Fiat Uno from Free3D.com*.
`https://free3d.com/3d-model/low-poly-fiat-uno-28162.html`. Accessed: 2023-11-14.

[42] *Volkswagen Passat from sketchfab.com*.
`https://sketchfab.com/3d-models/2004-volkswagen-passat-18t-obj-7831001635b248fa8c5df741e312f753`. Accessed: 2023-12-15.

[43] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A Modern Library for 3D Data Processing. *arXiv:1801.09847*. 2018.

[44] Jeff Johnson, Matthijs Douze, and Herv Jgou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*. 2019, 7 (3), 535–547.

[45] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*. 1965, 4 (1), 25-30. DOI 10.1147/sj.41.0025.

[46] Thomas Kailath. The divergence and Bhattacharyya distance measures in signal selection. *IEEE transactions on communication technology*. 1967, 15 (1), 52–60.

[47] OpenPCDet Development Team. *OpenPCDet: An Open-source Toolbox for 3D Object Detection from Point Clouds*.
`https://github.com/open-mmlab/OpenPCDet`. 2020.

[48] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3D Deep Learning with PyTorch3D. *arXiv:2007.08501*. 2020.

[49] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. *Soft rasterizer: A differentiable renderer for image-based 3d reasoning*. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019. 7708–7717.

[50] traveller59. *Python KITTI evaluation*.
`https://github.com/traveller59/kitti-object-eval-python`. 2019.

[51] Diederik P Kingma, and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 2014.

[52] Qinghao Meng, Wenguan Wang, Tianfei Zhou, Jianbing Shen, Luc Van Gool, and Dengxin Dai. *Weakly supervised 3d object detection from lidar point cloud*. In: *ECCV*. 2020. 515–531.