Automated Data Labeling of Driving Recordings for the Use in supervised-learning Applications for Sensor Fusion

*A **thesis** presented for the degree of*
Master of Science **in Computer Science**.

Eike Taegener, Freie Universität zu Berlin, Germany
Matriculation number: 5184853
Contact: eike.taegener@fu-berlin.de
Date of submission: **February 21, 2024**
Version: **Final Version**

Supervisor:
Claas-Norman Ritter [1], Freie Universität zu Berlin, Germany

Reviewers:
Prof. Dr. Daniel Göhring [2], Freie Universität zu Berlin, Germany
Prof. Dr. Tim Landgraf [3], Freie Universität zu Berlin, Germany

Short Abstract:
Gathering to train different machine learning models was not as easy as today. Analyzing and labeling these data is time- and human-consuming, particularly if data from several sensors are gathered. Furthermore, the labeling process gets more complex if the data representation is complex.
In this Master thesis, an *Automated Data Labeling Pipeline* is proposed, implemented, and evaluated. The pipeline takes driving recordings of the *Dahlem Center for Machine Learning and Robotics* as input. A *ROS* bag reader extracts the most relevant data. Two stages process the stored camera images (*Image Processing Stage*) and the LiDAR point cloud data (*LiDAR Processing Stage*. The last stage consists of a *Sensor Fusion*, where the image and LiDAR data processing results are fused to form a sensor fusion dataset.
Each pipeline stage is evaluated on its own. Furthermore, several publicly available datasets and a newly created one are used to train different object detection models.
By completing this Master thesis, an *Automated Data Labeling Pipeline* to support a human labeler is realized.

Citation:
Eike Taegener, *Automated Data Labeling of Driving Recordings for the Use in supervised-learning Applications for Sensor Fusion*, Freie Universität zu Berlin, Master Thesis, 2024. (unpublished)

[1] Dept. of Computer Science and Mathematics, Dahlem Center for Machine Learning and Robotics
[2] Dept. of Computer Science and Mathematics, Dahlem Center for Machine Learning and Robotics
[3] Dept. of Computer Science and Mathematics, Dahlem Center for Machine Learning and Robotics

Abstract

Gathering data to train different machine learning models was not as easy as today. In the context of autonomous vehicles, one can buy a *GoPro* action camera and put it into a car. While the car moves, the camera records the environment and stores the footage as a video file. The footage contains explicit information on objects, such as other cars, pedestrians, and traffic signs.

Making the gathered data applicable to AI models, the video footage is converted to images in the first place. A human labeler analyzes the images, focusing on several objects of interest. He annotates each object instance and stores the results on the host system. This process can take a long time. If the driving recording consists of thousands of images and multiple objects of interest exist, the labeling process can take days. The time effort increases rapidly when other sensor data, such as LiDAR, is gathered. LiDAR data is more complex than images because LiDAR creates point cloud data in the 3D space. An object consists of a point cloud of hundreds or thousands of 3D points. Furthermore, LiDAR does not distinguish between ground and non-ground points. The multi-dimensional representation of objects, as well as the non-discrimination of ground and non-ground points makes the labeling process much more complex and time-consuming.

In this master thesis, an *Automated Data Labeling Pipeline* to support a human labeler in his work and mitigate the time-consuming labeling process is proposed. The pipeline takes driving recordings of the *Dahlem Center for Machine Learning and Robotics* stored as *ROS* bag files as input. A *ROS* bag reader extracts the images of the front, left, right, and rear cameras and the LiDAR data. A segmentation model based on *DINOv2* [1] features calculates the segmentation map of each image. Furthermore, *YOLOv8* models detect different objects in each image. The LiDAR data is filtered by ground/obstacle segmentation using *GroundGrid* [2]. A *DBSCAN* [3] algorithm clusters the remaining non-ground LiDAR data. The objects found in the LiDAR data and the visual objects are fused at the end of the pipeline to generate a sensor fusion dataset.

The evaluation takes several datasets and metrics into account. For training and validating the *YOLOv8* models, the *nuImages* [4], *GTSDB* [5] datasets, and a traffic light dataset are used. The segmentation model is evaluated based on the CityScapes dataset. Furthermore, the sensor fusion is weakly evaluated on the *KITTI* [6] dataset. Several data used for the evaluation are taken from a *ROS* bag file, showing its application to the data of the *Dahlem Center for Machine Learning and Robotics*.

By completing this master thesis, an *Automated Data Labeling Pipeline* to support a human labeler is realized.

Zusammenfassung

Daten zu sammeln, um verschiedene Machine Learning Modelle zu trainieren, war noch nie so einfach wie heutzutage. Im Kontext von autonomen Fahrzeugen, jemand kann sich eine *GoPro* Actionkamera kaufen und in sein Auto installieren. Die Kamera nimmt die Umgebung auf, während das Auto bewegt wird und speichert die Aufnahmen als Video-Datei. Die Bilddaten enthalten explizite Informationen zu den verschiedensten Objekten, wie Autos, Fußgänger und Verkehrszeichen.

Um die Daten für AI Modelle verwendbar zu machen, müssen die Videodateien zu einer Bildserie umgewandelt werden. Ein menschlicher Labeler analysiert dann die Bilder bezüglich bestimmten Objekten. Die verschiedenen Objektinstanzen werden annotiert und auf dem System gespeichert. Dieser Prozess kann sehr viel Zeit in Anspruch nehmen, insbesondere wenn das Videomaterial sehr lang ist, aus mehreren tausend Bildern besteht und meherere Objekte annotiert werden sollen. Der zeitliche Aufwand vergrößert sich enorm, sobald Daten von anderen Sensorsystemen wie LiDAR gelabelt werden sollen. LiDAR Daten sind um ein Vielfaches komplexer als Bilder, weil die Punktwolken im 3D-Raum erstellt und gespeichert werden. Außerdem besteht ein einzelnes Objekt aus hunderten oder tausenden 3D Punkten. Die mehrdimensionelle-Darstellung von Objekten, als auch die nicht Unterscheidung zwischen Boden und nicht-Boden Punkten tragen dazu bei, dass der Labelprozess komplexer wird und mehr Zeit braucht.

In dieser Masterarbeit, wird eine automatische Datenlabelungspipeline um einen menschlichen Labeler zu unterstützen und den Zeitaufwand zu verringern, vorgestellt. Die Pipeline nimmt Aufnahmen von Fahrten vom *Dahlen Center for Machine Learning and Robotics* als Input. Ein *ROS* bag reader extrahiert die Bilder von den vier installierten Kameras (Front, Links, Rects und Heck), als auch die LiDAR Daten. Ein Segmentierungsmodel, basierend auf *DINOv2* [1] Features berechnet eine Segmentierungskarte für jedes Bild. Desweiteren werden *YOLOv8* Modelle detektiert verschiedene Objekte in den Bildern. Die LiDAR Daten werden durch ein GroundGrid [2] in Boden/Hindernisse unterschieden. Anschließend wird ein *DBSCAN* [3] Algorithmus verwendet, um Objekte zu finden. Am Ende der Pipeline werden die gefundenen Bildobjekte mit den LiDAR-Objekten fusioniert und ein Sensor Fusion Datensatz wird erstellt.

Die Evaluierung wird auf den Datensätzen *nuImages* [4], *GTSDB* [5] und einem Ampel Datensatz ausgeführt. Das Segmentierungsmodel wird durch den *CityScapes* [29] Datensatz evaluiert. Das *Sensor Fusion Modul* wird mit Hilfe des *KITTI* [6] evaluiert. Am Ende wird die gesamte Pipeline anhand einer Fahrtaufzeichnung vom *Dahlem Center for Machine Learning and Robotics* evaluiert.

Durch die Abarbeitung der verschiedenen Pipelinemodule wird eine automatische Datenlabelungspipeline realisiert, die einen menschlichen Labeler unterstützen kann.

Table of Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

In the chapter *Introduction*, the motivation for this work is described.

Related and relevant work to this work is named and explained in 1.2. They are categorized into *Object Classification and Detection*, *Extracting Visual Features and Segmentation*, and *Data Labeling and Dataset Generation*. Furthermore, different works on the same topic are presented and compared to each other. Additional information is given too.

In *Outline of Contribution*, the outline of this work is given. The goal that should be achieved at the end is named and explained. Furthermore, the approach to the work is described as well.

The chapter ends with the *Structure of the Thesis* in 1.4.

1.1 Motivation

In the last few years, artificial intelligence has become more and more important for different economic sectors and our daily lives. Different AI models are specialized for different purposes, for instance, object recognition and classification. Furthermore, recent developments in machine learning allow the real-time application of different models in autonomous vehicles and similar real-time areas. Due to this development, car manufacturers push towards autonomous driving [7].

Guaranteeing safety needs an AI model, that is trained on hundreds of thousands of data. Achieving reliability, accuracy, and generalizability is the main goal while training AI models for different tasks. This is only possible if large datasets of real driving recordings exist and are pre-labeled.

Different researchers demonstrated that the performance and generalization ability of machine learning models are strongly related to the used training data and the model's architecture. With more diverse data, a model can better generalize to new, unseen data. Furthermore, using more different samples from the same objects leads to a better detection performance for the corresponding objects. All in all, more labeled training data leads to an overall performance boost in accuracy and generalization ability. But to achieve this, a large pre-labeled dataset is needed if models are used that are trained by supervision.

Labeling data is a time and human-resources-consuming task. This is due to the labeling process. For instance, a human labeler must analyze an image and find each relevant object within it. Next, he has to mark every instance of the found object and save the results on the system. With an increasing number of samples and an increasing number of objects within one image, the task takes more time.

The time problem increases rapidly if the data isn't as simple to analyze as an image is. For instance, LiDAR data are more complex than images due to the spatial distribution of information across the 3D space. Furthermore, an object can consist of hundreds of 3D points, where all points have to be marked as the object in question. This is not an easy task if visual context is missing and only LiDAR data is collected.

In this thesis, an automatic data labeling pipeline is proposed to mitigate the time and human-resource problem of manual data labeling. In particular, different approaches for different tasks are examined and evaluated. The question of how to design and implement an automatic data labeling pipeline to support a human labeler is answered. Furthermore, the final pipeline is evaluated in great detail regarding accuracy, run-time, memory consumption, and applicability. The source code is available at *GitLab*[1].

1.2 Related Work

In *Related Work*, various works related to object classification, detection, and localization are presented. Different architectures for different purposes are illustrated. Furthermore, *Related Work* on automatic dataset generation, extracting visual features, and image and video segmentation are presented too.

Object Classification and Detection In [8], the authors proposed one of the first and largest convolutional neural networks, called *AlexNet*, by the time. The proposed network consists of five convolutional layers and three fully-connected layers for the final classification. It has around 60 million parameters and consists of 650.000 neurons, for the time a huge network. The authors executed several experiments and found that their results could be improved by simply using faster GPUs and larger training datasets. Furthermore, their proposed architecture is limited by the amount of available memory and the tolerated training time, they were willing to spend to achieve better classification results. This means, that their *AlexNet* architecture could be further improved by training the model longer and providing more memory to store the model. At the *ImageNet LSVRC-2010* contest [9], where 1.2 million high-resolution images should be classified, *AlexNet* achieved a top-1 error rate of 37.5%. The top-1 error rate is the proportion of the time the classifier doesn't provide the highest score to the correct class.

The authors of [10] followed the idea of deeper networks to achieve better performance and proposed the *VGG*-model family. Their models have a depth

[1] Automated Data Labeling Pipeline: https://git.imp.fu-berlin.de/taegenee98/thesis.git

between 16 and 19 weight layers, resulting in 134 million and 144 million parameters respectively. The architecture consists of several convolutional layers with filter size 3, spatial pooling layers, mainly max-pooling layers with size 2 and stride 2, and three fully-connected layers for the final classification. Between each pair of convolutional layers, a *ReLU* activation function is applied. The convolutional layers are chosen in such a way, that they reduce the parameters needed. For instance, a 5x5 kernel can be represented as two convolutional layers with a 3x3 kernel each, resulting in the same output size, such as the 5x5 kernel, but with fewer parameters needed. Another advantage is that two convolutional layers provide one additional non-linear rectification layer in between them. The *VGG-19* achieves 24.8% top-1 validation error, outperforming *AlexNet* by 12.7%. The corresponding top-5 validation error is 7.5%. The training of *CNNs* is a time-consuming task. Training a *VGG-19* with 19 weight layers needs several days but its performance is promising. The authors pointed out, that an increased depth is beneficial for classification accuracy.

The authors of [11] leveraged the idea of increased depth. They proposed residual learning and experimented with *Residual Convolutional Neural Networks* (*R-CNN*). With residual learning, they developed a 152-layer *ResNet* (Residual Network) that achieves an error rate of 3.57% on the *ImageNet* test set. Despite its high depth, it has a lower complexity than compared to the *VGG-19*. The *VGG-19* model has 19.6 billion *FLOPs* (Floating Point Operations Per Second), while the *ResNet-152* has only 12 billion *FLOPs*, regarding its much bigger depth. To achieve depth and lower complexity, the authors proposed special building blocks for residual learning. One such block is illustrated in Figure 1. A central finding is that depth is not everything. With increasing depth, optimization becomes more and more important because of exploding and vanishing gradients. They use batch normalization and other regularization techniques to deal with these problems and subsequently increase the performance of their architecture.

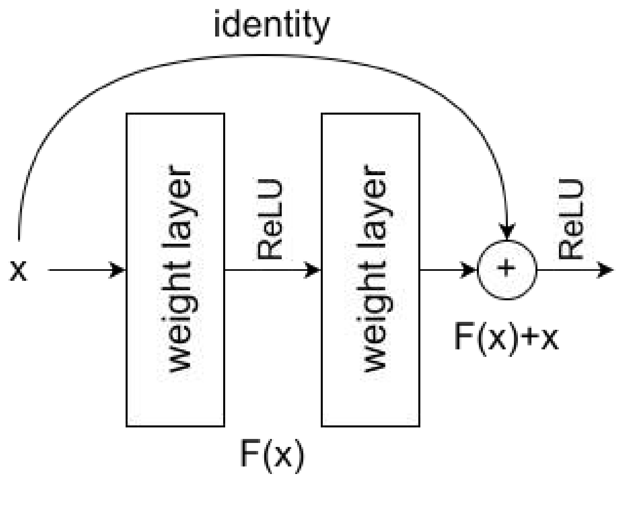


Figure 1: Residual building block of the *ResNet* architecture. [11]

To show the applicability to object detection, they adopted a *FASTER R-CNN* model and replaced the *VGG-16* feature extractor with a *ResNet-101*. With this modification, they achieved an increase of 6% in COCO's standard metric, which corresponds to a 28% relative improvement over the prior model, demonstrating the superiority of deeper models.

The aforementioned approaches are quite good for object classification, but they struggle with object detection. Object detection is more complicated because it doesn't classify objects within an image only, but also provides object localization with bounding boxes, that encapsulate the classified objects. Furthermore, a confidence value is computed as well.

In [12], *R-CNN* is introduced. It is the first deep convolutional neural network for object detection and semantic segmentation. *R-CNN* is a two-stage detector, meaning it first generates region proposals where objects might lay. Then it extracts fixed-length feature vectors via a *CNN* and finally classifies each region by a specialized linear *Support Vector Machine* (SVM). The newly proposed method achieved 53.3% mAP on the *PASCAL VOC-2012* dataset [13], outperforming the state-of-the-art by the time. They pre-trained the network on a large auxiliary dataset, removed the last 1000-way classification layer, and trained a (N+1)-way classification layer on a small, domain-specific dataset. As a feature extractor, they used a *VGG-16* model. The final classification is done by an *SVM*, specially trained on one class each.

In 2015, Redmond et al. introduced *You Only Look Once* (*YOLO*) [14], a real-time object detection approach that achieves a processing speed of 45 frames per second (fps) in the base model. They also proposed a *Fast YOLO* model, processing 155 fps. The authors framed object detection as a "*regression problem to spatially separated bounding boxes and associated class probabilities*". This means that a single neural network predicts bounding boxes and class probabilities directly from full images in one evaluation. In contrast, *R-CNN* is a region proposal-based two-stage detector with a complex pipeline. The first version of *YOLO* has several limitations, such as strong spatial constraints, which can lead to problems with small objects that appear in groups. Furthermore, *YOLO* struggles to generalize to objects in new or unusual aspect ratios or configurations, and their used loss function treats all errors the same even though errors in small bounding boxes have a much greater effect than ones in bigger bounding boxes.

To mitigate the limitations of the first *YOLO* version, the same authors released two consecutive improvements of *YOLO*, called *YOLOv2* [15] and *YOLOv3* [16]. The main aim was to make *YOLO* better and stronger while staying fast. They used more diverse images with new objects to increase the vocabulary and robustness of the *YOLO* model. Both improved methods use a new feature extractor network, named *Darknet-19* and *Darknet-53* respectively. Furthermore, an error analysis was carried out. Based on the results of this analysis, the author focused on improving recall and localization because a high number of errors were due to localization errors. Additionally, they simplified the architecture while increasing the mAP.

Several researchers tried to improve the *YOLO* architecture in the last couple of years. The main changes are made to the training strategy and the architecture to make it better, faster, and stronger. The latest version is *YOLOv8*[2], developed and maintained by *Ultralytics*. *YOLOv8* is based on the original *YOLO* architecture while being fast and efficient. It applies to several computer vision tasks, such as object detection and tracking, instance segmentation, and image classification. It is much faster than the original *YOLO* and provides a much better performance.

Extracting Visual Features and Segmentation Reliable and consistent visual features are essential for different computer vision tasks, such as classification and segmentation. The main aim is to generate features from an input image that are very distinct from each other if they represent different objects.

In the past, convolutional neural networks were used to extract meaningful and consistent features of an image. These features were then fed to a classification head, which performed the object classification. In 2021, Dosovitskiy et al. proposed the transformer architecture for images, called *Vision Transformers* (*ViT*) [17]. The authors found that when pre-trained on a large amount of data, *Vision Transformer* attains excellent results compared to state-of-the-art *CNNs*. Furthermore, they require substantially fewer computational resources to train. Dosovitskiy et. al. experimented with the transformer structure and different-sized datasets and found that a model has to be trained on a dataset consisting of 14 million to 300 million images to generate good, reliable features. Otherwise, its performance is not great. Another finding is that models with smaller patch sizes are computationally more expensive. An image is split into patches of fixed size. If the size of the patches decreases, more patches are needed to split an image completely. This increases the required computational resources.

Based on the findings of Dosovitskiy et al. and the emergence of *ViTs*, Caron et al. proposed self-**di**stillation with **no** labels (*DINO*) [18]. They found out that self-supervised *ViT* features contain explicit information about the semantic segmentation of an image. These findings do not emerge with *CNNs*. The *ViT*-generated features lead to a 78.3% top-1 accuracy performance on *ImageNet* with a k-NN classifier on top of the *ViT*. Another finding is that the self-supervised *ViT* models lead to better results for semantic segmentation compared to fully supervised ones. Following the success of self-supervised *ViTs*, they distilled several publicly available pre-trained *DINO* models.

Following the success of *DINO*, the same authors proposed a modified version of its approach, namely *DINOv2* [1]. They revisited the architecture and combined different techniques to scale pre-training in terms of data and model size. To train

[2] Ultralytics YOLOv8: https://github.com/ultralytics/ultralytics (Online, Accessed: February 21, 2024)

their new model they generated their own curated dataset with over 142 million images, called *LVD-142M*. The resulting *ViT* processes image patches with size 16x16 pixels to generate visual features for image classification (image-level) and segmentation (pixel-level). Following their first approach, they distilled several smaller models from their big one, namely *ViT-s/14* for the small model, *ViT-b/14* for the base model, *ViT-l/14* for the large model, and *ViT-g/14* for the giant model. These models are all publicly available. Furthermore, they performed downstream tasks, such as semantic segmentation and image classification. They used a simple linear segmentation head on top of their *ViT* base model, archiving 71.3% mIoU, while their boosted version achieved 81.0% mIoU on the CityScapes validation data split, showing its good performance with quite simple segmentation heads.

In [19], the authors presented a new architecture addressing any image segmentation task, named *Mask2Former*. *Mask2Former* achieved a new state-of-the-art performance and is built upon a simple meta-architecture. The first part is a backbone feature extractor, that generates meaningful and reliable features from an image. The second one is a pixel decoder, followed by a transformer decoder. The authors proposed several improvements over the meta-architecture. They use masked attention instead of cross-attention. Furthermore, to help the detection and segmentation of small objects, multi-scale high-resolution features are used. Further optimization improvements are proposed as well. The last improvement, regarding the training duration, is the use of a few randomly sampled points to calculate the mask loss. This requires less time than computing the mask loss on the complete output, securing the possibility of more epochs while needing the same amount of time.

Data Labeling and Dataset Generation Data labeling and subsequent dataset generation is a labor-intensive process in which humans are mainly involved. To create a dataset for object classification, a human has to analyze each image separately and annotate a label for each identified object. This is time-consuming. If the goal is a large dataset, the data labeling is human-consuming because several people are working on the labeling task. Furthermore, the labeling process gets more time-consuming if the data becomes more complex to analyze. For example, a dataset consisting of images and corresponding LiDAR data should be created.

Some works are trying to mitigate this problem by using neural networks to help the human labeler or label unlabeled datasets fully automatically. In [20], the authors developed an approach complementary to cooperative learning. The main idea is that a model labels different samples. Difficult samples are also labeled by a human, functioning as a reasoning tool for the model. The human-annotated sample is then used to enhance the model's accuracy. This process is carried out for a long time. If the human labeler is encouraged by the model's performance, he can delegate the labeling task to the machine entirely. Additionally, the authors took several inter-human-machine problems into account and described how to mitigate these.

The authors of [21] proposed a fully automated data labeling pipeline to annotate radar data and camera images simultaneously. For their pipeline, they used a *YOLOv3* model that detects objects within images. Corresponding radar point cloud data is clustered by *DBSCAN* [22]. Afterward, the cluster centroids are projected onto the image. Next, the centroids are associated with the bounding box centroids of the image objects, using the *Hungarian Algorithm* [23]. Even if the *YOLOv3* model fails to detect objects within frames due to bad illumination or similar, the pipeline can label the radar point clouds. The idea to realize this is that an object is continuously tracked across several radar scans. If the tracked object is identified by the *YOLOv3* in one image, the point cloud data can be labeled across all consecutive frames. The tracking is realized by another use of the *Hungarian Algorithm* to match the objects across consecutive radar scans.

1.3 Outline of Contribution

This thesis conceptualizes, develops, implements, and evaluates a fully automated data labeling pipeline for driving recordings saved as *ROS* bag files. Different models for different tasks are analyzed and evaluated. Different pipeline stages group the functionalities and requirements. Besides the requirements, the problem definition, different approaches, and the output of the pipeline stages are described. Furthermore, the possible usage of the different outputs is explained as well.

Different algorithms and approaches are discussed and compared regarding their run-time, memory consumption, and complexity. The data labeling pipeline should support a human labeler but not need to perform in real-time. It has to be reliable and have high accuracy, such that the human labeler does not have to correct many miss-classifications in the images and miss-matchings in the sensor fusion. Furthermore, functionalities are designed and implemented such that the labeler can correct miss-classifications.

1.4 Structure of the Thesis

The structure of this thesis is as follows.

In the second chapter **Basics** general information about autonomous vehicles is given. The sensor systems LiDAR and camera are described. They are compared to each other regarding their advantages, limitations, and applicability in the context of autonomous vehicles. The topic of sensor fusion is described in great detail. Different computer vision tasks that are relevant to this work are explained. The used machine learning architectures are explained and their functionality is described. Additionally, the different types of learning are explained as well. Finally, the technologies used in this work are presented.

The chapter **Data Labeling Pipeline** starts with the proposal of an *automatic data labeling pipeline* for driving recordings stored as ROS bag files. The proposal is followed by a detailed description of the different stages of the pipeline. Design choices and different approaches are explained and compared. Implementation details are provided for each stage as well as the training and validation routines of object detection and classification models. Furthermore, the architecture of the models used is illustrated and described. At last, the question of how the data is processed and the output is used is answered.

In the fourth chapter **Evaluation**, the proposed pipeline is evaluated regarding different aspects. First, all stages are evaluated individually. Their performance, regarding different metrics, is measured. Furthermore, their run-time is evaluated and illustrated. For general evaluation of the different detection and classification models, several publicly available datasets, such as *CityScapes* [29] and *nuImages* [4] are used. A final test is carried out on an original driving recording of the *Dahlem Center for Machine Learning and Robotics*. To do this, segmentation maps, objects, and LiDAR mappings are determined by hand to generate a validation set on real, unseen data. Furthermore, its applicability is demonstrated.

In the final chapter, **Conclusion**, the results of this thesis are described and discussed. Whether the goal of proposing a fully automatic data labeling pipeline for driving recordings has been achieved is discussed. *Limitations* to this work are described and their impact is explained. Finally, possible *Future Work* is described. Additionally, strategies to enhance the proposed pipeline are presented as well.

2 Basics

The basics of different sensor systems, sensor fusion, machine learning models, and learning paradigms will be explained in this chapter.

In the sections 2.1 and 2.2, the sensor systems *Camera* and *LiDAR* are introduced. Their advantages and limitations are explained. Furthermore, their application areas in cars are named. This is followed by the introduction of *Sensor Fusion* and the explanation of inter-sensor calibration as well as the different *Fusion Levels* in section 2.3.

Section 2.4 deals with the *Computer Vision Tasks*. The tasks relevant to this work are described and their differences are pointed out.

Different *Types of Learning* are described in section 2.5.

For the evaluation, several metrics are used. These are described in section 2.6.

The chapter ends with the introduction and explanation of the most relevant machine learning *Architectures* in section 2.7 and the used *Technologies* in 2.8.

2.1 Camera

A camera is an image-producing hardware. It consists of a sensor, a lens, and corresponding circuits. Cameras can take gray-scaled or colorful images and are usable in several different environments. In a cell phone to take pictures and record videos, in a notebook as a webcam, within buildings as security features, and many more.

Areas of Application in Cars Cameras are used for several assistance systems: Parking, lane change, lane keeping, and reversing assistance. Additionally, they are used to recognize and classify traffic signs and determine the traffic light phase on the current lane. Furthermore, they are applicable as security features to track and record who comes close to the car (e.g., *Tesla's* Guard mode[1]).
Emerging regulations propose the monitoring of the driver to detect if the driver is responsible, awake, and concentrated[2]. If not, a signal tone is emitted, such that

[1] Teals Wächter-Modus: https://www.tesla.com/ownersmanual/model3/de_lu/GUID-56703182-8191-4DAE-AF07-2FDC0EB64663.html (Online, German, Accessed: February 21, 2024).

[2] BMDV - Bundesamt für Digitales und Verkehr: URL: https://bmdv.bund.de/SharedDocs/EN/Articles/StV/Roadtraffic/new-vehicle-safety-systems.html (Online, Accessed: February 21, 2024).

the driver is woken up and reminded that he should be concentrated and responsible while the car is moving.
Another application is object recognition and classification within images. Camera images deliver structural information of an object in the corresponding field of view. This information is usable for object recognition and classification of pedestrians, cars, traffic lights, and more.

Advantages In recent years, camera sensors have gotten smaller while taking better pictures with each iteration. Especially nanotechnology plays a huge part in this development. With it, cameras can have the size of a grain of sand and take meaningful pictures. For instance, such cameras are usable in surgery or similar.
With the introduction of mass production, the price of a camera gets cheaper and cheaper. Today, car manufacturers can fit cameras everywhere in the car due to the small form factor and the low costs. This is obvious because cars are already equipped with cameras to detect traffic signs without compromising the view of the driver. They are mounted at the top center of the windshield.
A camera image contains rich data, such as coloring and object shape. These data are important for object classification and recognition. Especially coloring is used to determine the traffic light phase. This is not possible with LiDAR sensors due to their working mode, for instance.

Limitations Taking a good enough image for computer vision tasks relies on different factors. In severe weather conditions, such as heavy rain, fog, or snow, the camera can struggle to take meaningful images because of the bad visibility of different objects. Bad illumination, especially in the dark, is a huge challenge. Because of the missing light, the camera might take only black images, meaningless for object detection. Furthermore, a camera can be blinded by the sun's light like a human if the lens and shutter do not adjust in time. For instance, this can be the case at the end of a tunnel.
Another limitation is that image processing is more expensive than LiDAR data processing. Each image consists of three channels (RGB) unless the image is gray-scaled from the beginning. Furthermore, an image contains several hundreds of information, leading to a high consumption of memory storage. Besides these two factors, algorithms working with camera images are mostly computationally expensive.
To cope with the aforementioned limitations, car manufacturers build specific chipsets only for image processing. They are more energy and computationally-efficient than regular central processing units.
Another limitation is the calibration of cameras. Most cameras used in vehicles are equipped with a fisheye lens, such that the taken image is distorted but contains a larger field of view. Calibration regarding the distortion of the images is needed to rectify the image and make it usable for other tasks. This calibration is done whenever the car is in a service center. Otherwise, algorithms based on rectified camera images might deliver wrong results, leading to possible misbehavior of the car.

2.2 LiDAR

LiDAR stands for *Light Detection and Ranging* and is based on the *Time-of-Flight* (ToF) principle like radar sensors. LiDAR is also referred to as ToF sensor, laser scanner, and laser radar. Furthermore, LiDAR is a 3-D sensor that produces data derived in time and 3D space.

Areas of Application in Cars LiDAR is a quite new and expensive technology compared to radar and cameras. Hence, it is not much used in vehicles yet. However, some manufacturers already implemented them in the front of their cars. For instance, *Mercedes* implements a solid-state LiDAR sensor in the front of their cars to detect objects, replacing prior radar sensors [24].
Furthermore, LiDAR enables 360-degree scans while the sensor is packed into a small form factor. They are usable to detect different objects around the car. These scanners are mainly mounted on the roof of a car, leading to the possibility that objects behind other objects are recognized. For instance, a child behind a car can be detected, providing more security and safety for the driver and their surrounding.

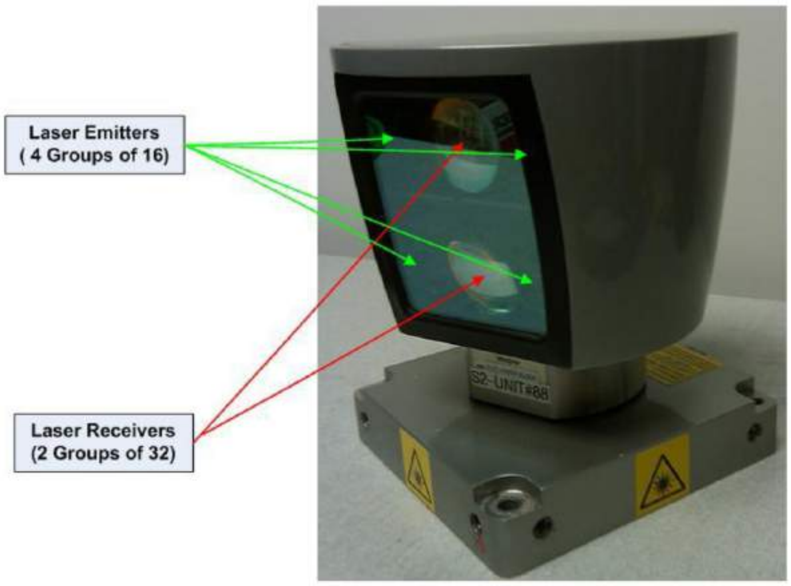


Figure 2: *Velodyne* HDL-64E LiDAR sensor. It consists of 64 laser emitters, split into four groups of 16 emitters and two groups of 32 laser receivers.

LiDAR Basics A LiDAR system consists of a laser source that emits laser pulses, a scanner, and a detector. It is an active sensing method that is based on the ToF principle. Figure 2 shows a *Velodyne* HDL-64E[3] LiDAR sensor. It has 64 laser emitters and 64 receivers.
A typical LiDAR sensor emits pulsed light waves into the surrounding environment. These pulses bounce off surrounding objects and return to the sensor. The

[3] Image from MDPI: https://www.mdpi.com/2072-4292/2/6/1610# (Online; Accessed: February 21, 2024)

sensor uses the time it took for each pulse to travel from the emitter to the detector to calculate the distance it finally traveled by

$$d = \frac{c \cdot \Delta t}{2} \tag{2.1}$$

with the speed of light c and the time difference between sending and receiving the emitted laser pulse Δt.

Types of LiDAR Sensors LiDAR sensors differ in their technique to enlighten the environment and their applicability. The used techniques can be categorized as scanning LiDAR, where the distance measurements are gathered successively by scanning the surrounding bit for bit and non-scanning LiDAR, where all distance measurements are gathered simultaneously.
Scanning LiDAR can be divided further into mechanic and solid-state LiDAR. The mechanic LiDAR works mostly with a motorized mirror that is rotated to reflect the laser pulses. Disadvantages are the mechanical wear and the size of the complete module. This limits its applicability. On the other hand, solid-state LiDAR can use a mirror moved by electromagnetism or an antenna array like radar. Advantageous is the elimination of all mechanical parts as well as enabling the possibility to mount a LiDAR sensor directly onto a chipset. Furthermore, this leads to a possible smaller form factor for autonomous vehicles. The main disadvantage is that re-calibration is needed if the temperatures are changing rapidly. With rapid temperature changes, the mirrors can drift out of alignment. They may not maintain calibration [25] and produce wrong results if not re-calibrated.
The development of a solid-state LiDAR with antenna arrays is inspired by radar. The main advantage is a small form factor, using electromagnetic waves with a wavelength of 905 nanometers (nm) or 1550nm, depending on the used sensor. The main limitation is that small failures inside the antenna arrays can lead to big problems, such as an object appearing further to the right than it is due to a signal being wrongly detected by the wrong antenna. Furthermore, other calculation errors can occur if the array is damaged.
Non-scanning LiDAR is also referred to as flash LiDAR. Its working principle can be described as a camera flash. The LiDAR's flash enlightens the complete scenery at once. The main problem is to choose between a strong flash intensity to measure the environment and receive strong enough signals and a sensitive detector to be easy on the eyes. If not carefully tested, a safety issue can occur for pedestrians and other traffic participants.

Advantages Depending on the sensor type, LiDAR can detect objects at distances ranging between a few meters to more than 240 meters. The capability to provide a field of view of 360-degree depends on the sensor type as well. For instance, solid-state LiDAR sensors can only emit laser pulses in one direction, whereas mechanical sensors can scan 360 degrees due to their mechanical parts. Compared to radar, only one LiDAR sensor is needed to produce 360-degree scans. To achieve this with radar multiple sensors would be needed with corresponding inter-sensor calibration, making it an inapplicable procedure.

Because LiDAR's wavelength is a multiple smaller than the ones of comparable radar sensors, its resolution is much higher. Higher resolution enables the possibility of detecting smaller objects, leading to better classification and object recognition performance. Figure 3 illustrates the difference in resolution. The coloring is based on the distance to the sensor and not object-type specific. On the left side, an image produced by a LiDAR scanner is shown. On the right side, the same scene is shown but scanned with a high-resolution radar. The LiDAR sensor produces scans with a much better resolution and fidelity to detail. For instance, the human's shape is sharper in the LiDAR scan compared to the radar one. Furthermore, the one human in the radar scan is almost not distinguishable from the object to the left due to its representation as a block-like structure.
Without the coloring, one may not be able to distinguish the humans from the car in the right image.
Especially at night, a LiDAR sensor outperforms a camera sensor by a huge margin. In case no external light source is available, cameras may not take meaningful images. They are mostly black and do not contain any useful and usable information. LiDAR has its own light source and can be used without problems at night and without the need for an external light source.

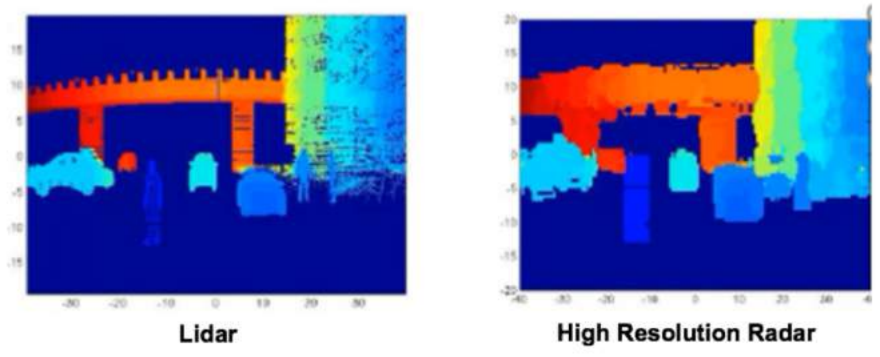


Figure 3: Resolution difference between a LiDAR scanner (on the right) and a high-resolution radar (on the left) [26].

Limitations LiDAR is a sensing method that is unable to distinguish between different colors because of its mode of operation.
LiDAR is a quite new technology and is at the moment more expensive than a comparable radar sensor. For instance, a *Velodyne VLP-16* LiDAR sensor costs approximately 4.600$[4], while a radar sensor costs approximately 200$[5].
In severe weather conditions, a LiDAR scanner might struggle with damped signals due to heavy rain and similar. Additionally, such as cameras, LiDAR sensors are prone to blinding by the sun because of the supersaturation of the photoelectric sensors.

[4] Rockwell Automation: Puck by *Velodyne*, URL: https://store.clearpathrobotics.com/products/puck (Online; Accessed: February 21, 2024)

[5] carparts onlineshop: Radarsensor Mercedes Bosch, URL: https://www.carparts-onlineshop.com/de/radarsensor-a2139058613-mercedes-0203304134-bosch-0203304134.html (Online; Accessed: February 21, 2024)

It does not always require to be bad weather such that LiDAR produces wrong detections. A LiDAR scanner can produce a point cloud behind the ego-vehicle due to the exhaust gases. This happens when the light pulses are reflected by the small gas particles.
The maximum scanning distance of LiDAR is smaller than radar. LiDAR only provides distances up to 240 meters (*Velodyne VLS-128* or *Velodyne Alpha Prime*), while radar achieves up to 300 meters.
The last issue is the energy consumption of LiDAR scanners. E.g., a *Velodyne VLP-16* scanner consumes 8W while needing 9V-18V. A comparable front radar sensor consumes less than 4W. The energy consumption increases with more lines that can be scanned. A Velodyne VLP-16 scans up to 16 different lines, while the newer *Velodyne Alpha Prime* can scan up to 128 lines but consumes 22W [6].

2.3 Sensor Fusion

Sensor Fusion describes the information and data fusion between different sensors, such as LiDAR and camera or radar and camera. The goal is to enrich the data of different sensors and make classifications, predictions, and detections more reliable and robust. Furthermore, sensor fusion can save computational resources by connecting and analyzing data of one sensor and applying these results to other ones.

The sensor fusion process consists of three separate steps. First, different sensors are compared. Following the comparison, an appropriate sensor configuration for a certain use case is selected. Next, the different sensors are calibrated regarding a global origin. This is mostly somewhere in the ego-vehicle. At the end, the sensor fusion is performed.

2.3.1 Sensor Calibration

Sensor Calibration, in the context of LiDAR-camera fusion, consists of the following steps: point cloud filtering, coordinate calibration and error calibration. The different calibration steps are executed in advance because the sensors need to be calibrated before their information can be fused. Otherwise, wrong data may be fused, resulting in false fusion data.

Following, each sensor calibration step is described individually.

Point Cloud Filtering Radar and LiDAR sensors create 3D data points but they are not distinguished between relevant and irrelevant points. For instance, a LiDAR scanner does not distinguish between points lying on the ground or being

[6] *Velodyne Alpha Prime* datasheet: https://www.mapix.com/wp-content/uploads/2019/11/VelodyneLidar_AlphaPrime_Datasheet.pdf (Online; Accessed: February 21, 2024)

reflected by a car's side. To remove such points, point cloud filtering is applied to filter noise and useless detection results, such as ground points or similar. Furthermore, an object detection algorithm can be applied to only gather the objects within the LiDAR scan. Subsequently, the filtering reduces the number of relevant data points for the analysis.
Different works propose methods for noise filtering and target extraction.

Coordinate Calibration The data produced by different sensors differ in their representation and origin. For instance, radar and LiDAR sensors create 3D data points with the sensor's position as the origin. In comparison, the camera produces 2D data with the camera's position as the origin. Appropriate sensor fusion is only possible if the different sensor coordinate systems are calibrated regarding one global origin. Hence, different transformation matrices are calculated to transform LiDAR data from the LiDAR coordinate system to the car's coordinate system. Then from the car's coordinate system to the camera's coordinate system. Furthermore, the projection from the 3D space to the 2D space is calculated as well. A projection of point cloud data onto the corresponding image is illustrated in Figure 4.

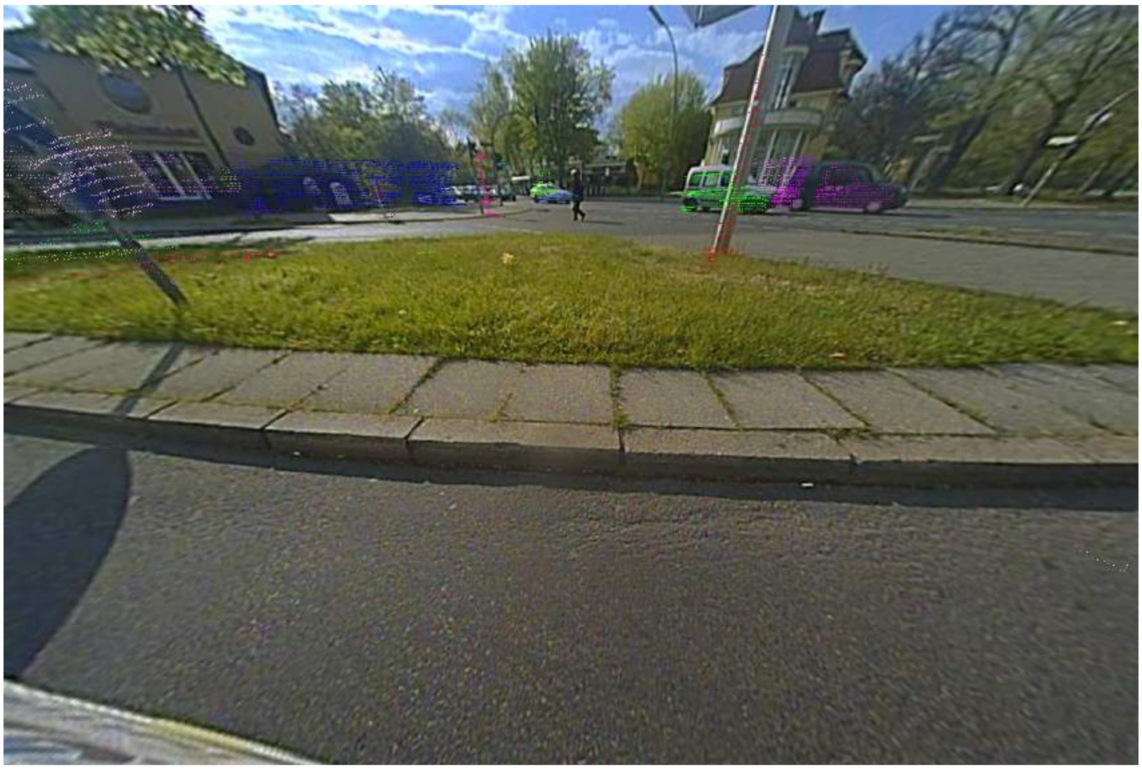


Figure 4: Example for the projection of LiDAR data onto a camera image. The different colors refer to different objects found by a clustering algorithm.

Error Calibration Sensor data may be poor and contain errors. Furthermore, mathematical calculations can contain errors if the results are rounded or similar. Especially, the projection from 3D data onto a 2D plane can produce projection

errors due to unfinished or bad calibration. This could be the case as well if the camera images are rectified from a distorted one.
To cope with these errors, error calibration is carried out.

2.3.2 Fusion Levels

3D data and vision information are fusible in different ways. Each of them has advantages. For instance, data level fusion achieves the most reliable data and uses most of the original data but depends on the availability of enough data points. Furthermore, a hidden security danger exists while using data-level fusion. For decision and feature-level fusion, the point cloud data are preprocessed.

Data Level *Data Level Fusion* is imaginable as the determination of *Regions of Interest* (ROI). After the filter and error calibration, a clustering algorithm determines objects of interest in the point cloud data. The cluster centroids are calculated. Next, the cluster centroids are projected via the calculated transformation matrices from 3D onto a 2D plane. The projected cluster centroids are mapped onto the corresponding camera image. A bounding box, representing the approximation of the possible object, is assigned to each cluster. Each bounding box represents an ROI which is fed into an object detection model to classify the object inside the image part. Figure 5 illustrates this process for a simplified image and radar output.
The determination of ROIs reduces the analysis of irrelevant parts of the corresponding image significantly. Only the image portions, which lay inside an ROI, are considered for the classification. Nevertheless, the determination of ROIs leads to a hidden security danger, especially if the point cloud data aren't accurate. The problem of possibly not detecting objects is a security danger emerging with the use of Data Level Fusion only.

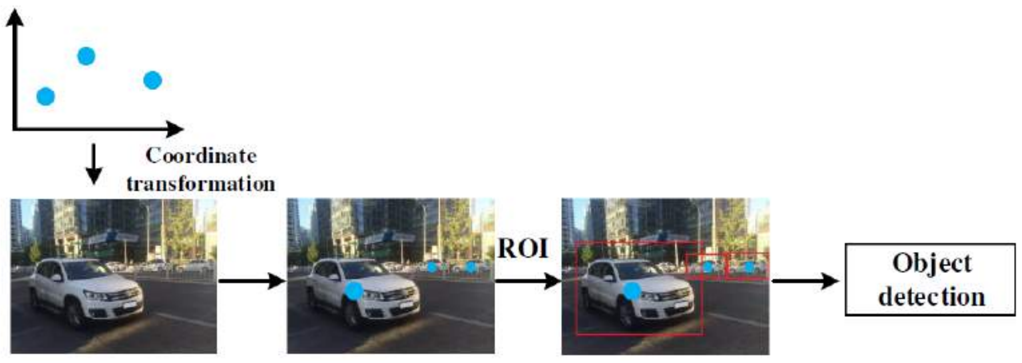


Figure 5: Illustration of the working principle of *Data Level Fusion* [27].

Decision Level The fusion of the individual classification and detection results of the point cloud data and the camera images is called *Decision Level Fusion*. The point cloud data and camera images are processed individually by object detection algorithms. The output of each processing is a list of possible objects with assigned information. Both pieces of information are fused into one final result. The process of Decision Level Fusion is illustrated in Figure 6. Problematic is the computational effort that has to go into the different detection systems. It is much higher than the effort for the *Data Level Fusion* because of the missing ROIs, which shrinks the image into computational smaller areas. An advantage of this approach is that the hidden security danger is mitigated by applying a sliding window to the camera image and analyzing the complete one.

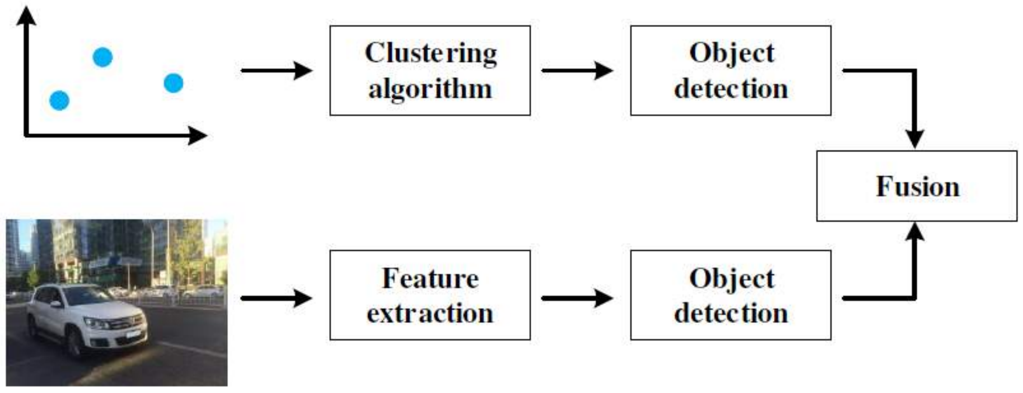


Figure 6: Illustration of the working principle of *Decision Level Fusion* [27].

Feature Level In *Feature Level Fusion*, the features of the point clouds (e.g., position, azimuth, velocity, etc.) and camera (size, object class, etc.) are extracted. For instance, the background class of the image is removed, while other features such as the object classes are extracted. Figure 7 illustrates the *Feature Level Fusion* process. After the extraction, the point cloud data and extracted information are transformed into an image-wise form. Next, the point cloud image and the camera image are fused. An object detection algorithm analyzes the fused features and outputs different detected objects.

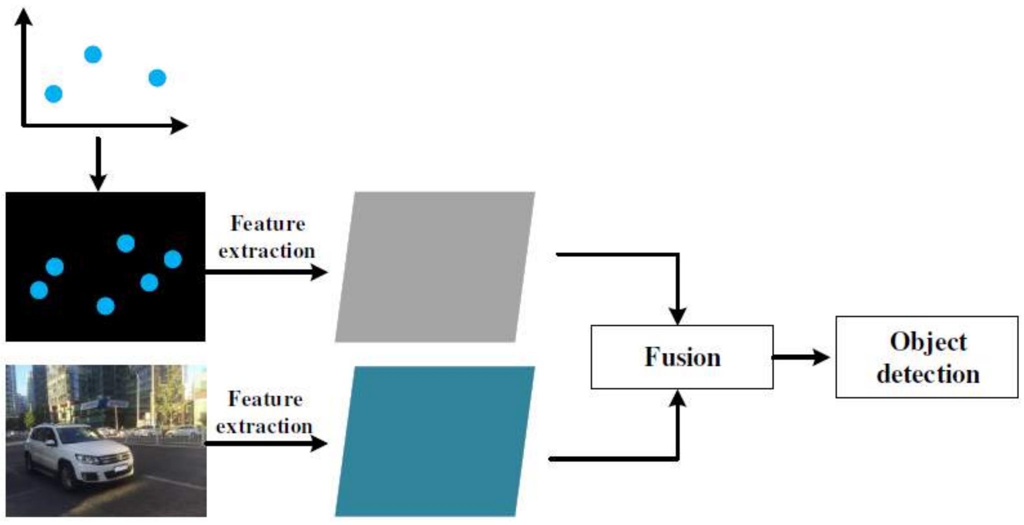


Figure 7: Illustration of the working principle of Feature Level Fusion [27].

Difficulties While using sensor fusion, different difficulties may occur. The most important one is the development of efficient algorithms that process point cloud data and images in real-time to make them applicable to autonomous vehicles. Otherwise, they are a security danger in autonomous vehicles and are not usable in real-life applications because a car may react too late. Another difficulty is the network and point cloud pruning. In network pruning, we are interested in removing unnecessary parts of the neuronal network to increase its processing speed while being accurate and fast. Point cloud pruning is more difficult to carry out. The goal is to create efficient algorithms that clean and filter unnecessary point cloud data while the hidden security danger does not increase. The important point cloud data should remain in the end.

2.4 Computer Vision Tasks

Computer Vision is a subfield of machine learning. It deals with anything that humans see and perceive. If we look at an image, we first identify objects in it. We try to find relations between the objects and the scenery or try to recognize the place in the image. Sometimes, we look at an incomplete or damaged image and use our knowledge and experience to determine what is missing from it.

All of the aforementioned situations are *Computer Vision Tasks*. It includes different methods for acquiring, processing, analyzing, and understanding images.

Following, I describe the relevant *Computer Vision Tasks* for this work.

Object Classification Object classification is a fundamental task in vision recognition that aims to automatically assign a label or class to an unknown or unlabeled example, mainly an image. The model is given an image with one portrayed object as input and computes to which class the object may belong. It automatically assigns the predicted label to the image. Unlike object detection, image classification typically pertains to single-object images.
Typical classification tasks are face expression classification and plant species classification. For each of these tasks, an object classification model is trained on a big preprocessed dataset consisting of hundreds or thousands of samples with corresponding annotations.
E.g., *Google Lens*[7] is a highly known and well-trained classification model.

Object Detection Object detection is a more advanced task than object classification because it consists of two steps. First, an object detection model tries to find all positions of the known objects within an image, namely the localization step. Second, the model classifies the detected objects and assigns a separate label to each object. In short: Object detection is the task where objects are localized and classified.

[7] Google Lens: https://lens.google/intl/de/ (Online; Accessed: February 21, 2024)

Compared to object classification, detection can detect several objects within one image, while classification is mainly restricted to one object per image.

Image Segmentation Image segmentation describes the partitioning of an image into multiple parts or regions. It is a commonly used technique in digital image processing and analysis. The different regions or parts of an image are determined by the characteristics of the pixels. One can split image segmentation into three distinct tasks. These are semantic segmentation, instance segmentation, and panoptic segmentation.
In **Semantic Segmentation** each pixel of an input image is associated with one class or category, such as car, pedestrian, sky, or others. The target is to produce a pixel-wise segmentation map of an image. One is not interested in how many objects of the same class are in the image. For instance, there is no interest in how many cars are in the image. The only relevant information is that there are cars and where they are.
Instance Segmentation is a special form of semantic segmentation that deals with detecting and delineating each distinct instance of an object in an image. It detects all instances of a class while decomposing the separate instances of any segmented class. For instance, instance segmentation is capable of detecting several cars, while semantic segmentation returns only one mask for all cars in the image.
Panoptic Segmentation fuses instance and semantic segmentation and assigns a semantic label based on the class definition and an instance ID from the instance segmentation to each pixel of the original image.

2.5 Types of Learning

Training a machine learning model can be done in one of three different learning paradigms. These paradigms are supervised, unsupervised, and semi-supervised learning. Each paradigm has its own applicability, advantages, and limitations.

Following, each learning type and in which context they are used is described. Furthermore, the structure of the underlying datasets and examples of their applicability are given.

2.5.1 Supervised Training

Supervised training is characterized by the availability of a dataset consisting of a collection of labeled examples. A labeled example (x, y) is an element x, called a feature vector, and the corresponding label y. For instance, a dataset consisting of images of fruits with corresponding annotations for object classification can be used for supervised training. The annotation can be something such as the name of the fruit or its growth state.

The supervised training paradigm is used to predict or classify data accurately. The final layer of a supervised training model is a classification layer with n neurons representing the different labels. For training, the model predicts a label based on the input data. The prediction is compared with the ground truth label and a loss is calculated. Subsequently, the weights of the model are adjusted based on the loss. This procedure helps the model to predict the data appropriately if the process is carried out multiple times. At the training end, a model should be produced that takes a feature vector x as input and returns a label y. Compared to the ground truth, the model should not make many mistakes on unknown samples.

2.5.2 Unsupervised Training

Unsupervised training uses only unlabeled data to make predictions. The dataset is a large collection of unlabeled samples x. For instance, the dataset could be a collection of images without annotations.

The main goal of unsupervised learning is to find structures within the data and cluster them. Furthermore, typical applications are data clustering to find similar objects within the data and outlier detection. To achieve this, a model has to learn features and structures from the data. For instance, the model is fed all data where each sample x is a feature vector of dimension n. A model may try to find a separation of the data inside the n-dimensional space, called clusters. This process is called model fitting. If the fitting is finished, one can input unseen data into the model and get a prediction for the input's class based on the underlying dataset.

2.5.3 Semi-Supervised Training

In semi-supervised training, a dataset consisting of labeled and unlabeled examples is used, while the quantity of unlabeled examples is much higher than the labeled ones. The core idea is to treat a sample differently based on whether it has a label or not. The labeled sample is processed using traditional supervision to update the model's weights. For unlabeled data, the underlying algorithm tries to minimize the difference in prediction between other similar training samples.

The main issue of having much less labeled training samples is the danger of bad generalization onto the unlabeled samples. The model may predict the labeled samples well but can struggle with the unlabeled ones.

The labeled samples function as sanity checks and ground truth data. They add structure to the learning problem by establishing how many classes are there and which clusters correspond to which classes. The unlabeled samples provide contextual information with which the shape and distribution of the whole dataset can be estimated.

2.6 Metrics

For the evaluation of different machine learning models, one can rely on several metrics.

The relevant metrics for this work are *Precision*, *Recall*, F_1, *IoU*, *Accuracy*, *mIoU*, and *mAP*.

The value calculation is based on the prediction results of the different models. In the case of a binary classification problem, the following definition is made. A *True Positive* (TP) is a prediction that is the same as the ground truth but is the relevant class. A *False Positive* (FP) is a prediction that has the same class as the relevant class, but the ground truth is the irrelevant class. A *True Negative* (TN) is a prediction that is the same as the ground truth but is the irrelevant class. A *False Negative* (FN) is a prediction that has the same class as the irrelevant class, but the ground truth is the relevant class.

For a multi-classification problem, the values are taken for each class individually and then calculated into one value.

The *Precision* describes the fraction of relevant retrieved instances among all retrieved instances. It is also referred to as a positive prediction value. A high *Precision* shows that the model does almost no false positive classifications.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

The *Recall* describes the possibility of a model to predict the relevant instances from a given dataset correctly. It is also referred to as sensitivity. A high *Recall* shows that the model does almost no false negative classifications.

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

The F_1-Score is the harmonic mean of *Precision* and *Recall*. The score varies between zero and one. If the score tends to one, it means that the *Precision* and *Recall* are high. If the score tends to zero, the opposite is the case.

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{2.4}$$

The *IoU* describes the *Intersection over Union* of the relevant retrieved instances regarding all retrieved instances plus the wrongly classified instances that are relevant. It describes the proportion of the correctly predicted relevant instances among all relevant retrieved instances.

$$IoU = \frac{TP}{TP + FP + FN} \tag{2.5}$$

The *Accuracy* describes the fraction of correctly classified instances among all instances of a given dataset.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.6}$$

The *mIoU* describes the *mean Intersection over Union* and is a common evaluation metric in semantic image segmentation. To gather the *mIoU* the *IoU* of each class are summoned and divided by the number of classes.

The *mAP* describes the *mean Average Precision* and is a standard metric to evaluate the performance of object detection algorithms such as *YOLO* or *R-CNN*. It is the average of *Average Precision* of each class. The *mAP* incorporates the trade-off between *Precision* and *Recall* and considers both FP and FN.

2.7 Architectures

A variety of models for different machine learning tasks exist. Some are more appropriate for certain tasks than others. For instance, a *Linear Regression Model* (LR) is more suited to predict future incomes than a *Deep Neural Network* (DNN). The predictions of a DNN might be good as well, but its architecture may be way bigger than the architecture of a LR model. Furthermore, the computational cost of an LR model is a fraction of the costs of a DNN because of its reduced complexity.

Next, I will describe the most important architectures for this work: *The Convolutional Neural Network* (CNN) and *Vision Transformer* (ViT).

2.7.1 Convolutional Neural Network (CNN)

A CNN is a *Neural Network* consisting of three main layers: *Convolutional Layer*, *Pooling Layer*, and *Fully-Connected Layer*. Further calculations, such as normalization or similar, are integrated as separate CNN parts.

Convolutional Layer The *Convolutional Layer* (conv layer) is the core building block of a CNN. A conv layer processes mainly 2D data but can be extended to 3D data. It takes a tensor of shape n$\times$ h$\times$w as input, where n is the number of inputs, h is the height, and w is the width. The output of a conv layer is an abstracted feature map from the image, also referred to as an activation map. The height and width of the activation map may be smaller than the shape of the input data. The conv layer consists of the input tensor, the output activation map, and a set of filters whose parameters have to be learned. The filter, often referred to as the kernel, has a height and width, which are smaller than those of the input image. The filters run over the input tensor in the fashion of a sliding window. The selected image part is multiplied by the different filter values. The results of this calculation are stored as an entry of the different activation maps, one for each filter. This procedure is done for the complete image to obtain the complete activation maps, one map for each filter. The filter's function can be modified by

additional parameters. In default settings, the windows slide over each column and row of the input tensor one after another. With increasing stride, one can skip over several columns and rows. This shrinks the activation map more rapidly compared to the default settings.
Figure 8 illustrates how the different components of a conv layer work together and how the activation map is calculated.

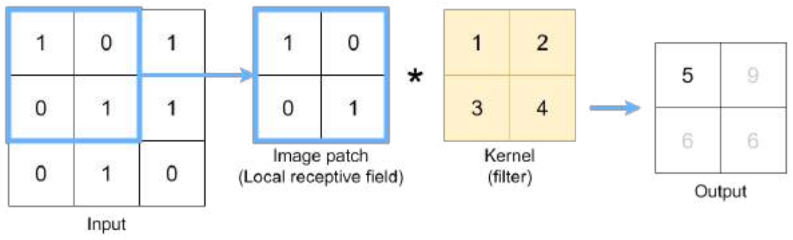


Figure 8: Convolution layer architecture with its different components. The stride of the kernel is set to one, as well as the padding. The result of applying a kernel of size 2×2 to a tensor of size 3×3 is an activation map of size 2×2.

Pooling Layer The *Pooling Layer* (pool layer) lies between two conv layers. Its main task is to reduce the dimensionality of feature maps produced by a conv layer while obtaining the most important features of it. It clusters the output of neurons at one layer into a single neuron in the next layer. The pool layer has a height and a width, defining the sliding window's size.
Different types of pooling methods are used in machine learning. The most popular ones are max and average pooling. Figure 9 illustrates how max-pooling and average-pooling works: A kernel with size 2x2 slides across a feature map of size 4x4. Each time the kernel slides further, the maximum or average value within the kernel's field of view is calculated. The calculated value is stored as output for this operation. This process is repeated until the kernel reaches the bottom right corner of the input feature map. The output of the pooling layer can then be used for further operations. For example as input to the next conv layer.

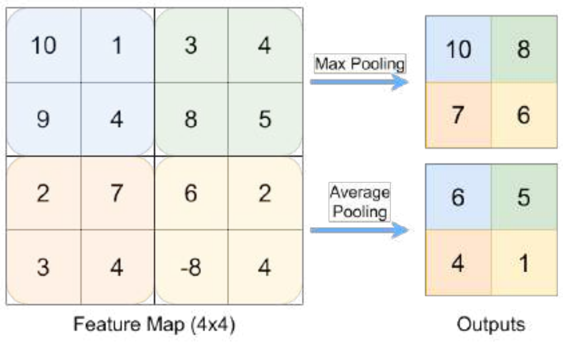


Figure 9: Illustration of how a Pooling layer works based on average and maximum pooling.

Fully-Connected Layer *Fully-Connected Layers* (FC layers) are neural networks where each neuron of a layer is connected with each neuron of the prior and the next layer. It is the same as a *Multilayer Perceptron* (MLP). Before the output of the last pool layer or conv layer is fed into the FC layers, the output tensor is flattened to map a value to each neuron. Then, the flattened matrix goes through a fully connected layer to classify the image or the input data of the CNN.

Figure 10 illustrates the architecture of a simple CNN (*AlexNet*). More complex CNNs are available. The *VGG-16* (*Very Deep Convolutional Neural Network*) is well known. It has a depth of 16 convolutional layers and consumes roughly 533MB of storage, making implementation and training of it a time-consuming task.

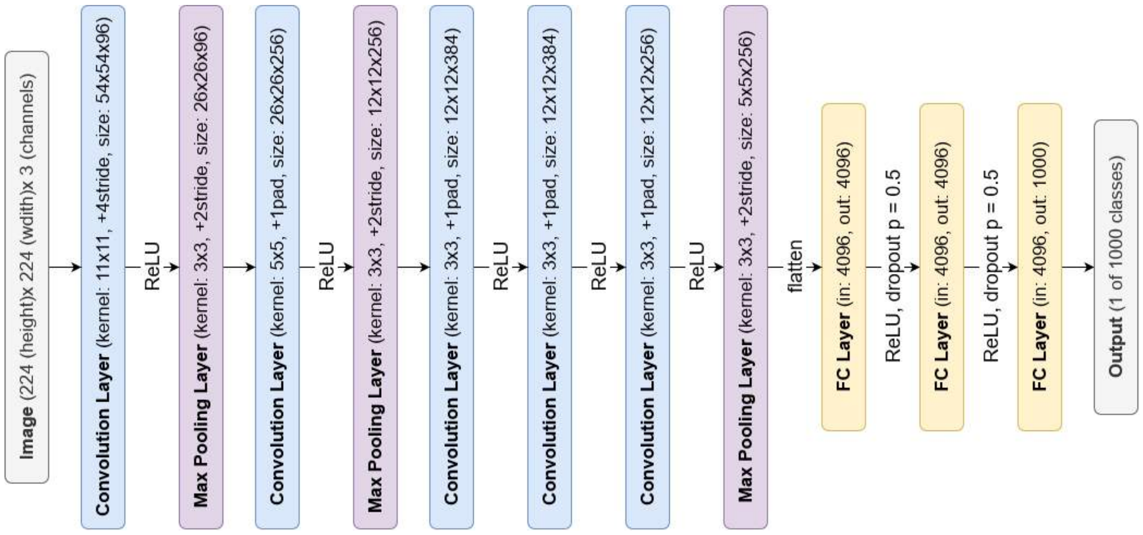


Image (224 (height)x 224 (width)x 3 (channels))
Convolution Layer (kernel: 11x11, +4stride, size: 54x54x96)
ReLU
Max Pooling Layer (kernel: 3x3, +2stride, size: 26x26x96)
Convolution Layer (kernel: 5x5, +1pad, size: 26x26x256)
ReLU
Max Pooling Layer (kernel: 3x3, +2stride, size: 12x12x256)
Convolution Layer (kernel: 3x3, +1pad, size: 12x12x384)
ReLU
Convolution Layer (kernel: 3x3, +1pad, size: 12x12x384)
ReLU
Convolution Layer (kernel: 3x3, +1pad, size: 12x12x256)
ReLU
Max Pooling Layer (kernel: 3x3, +2stride, size: 5x5x256)
flatten
FC Layer (in: 4096, out: 4096)
ReLU, dropout p = 0.5
FC Layer (in: 4096, out: 4096)
ReLU, dropout p = 0.5
FC Layer (in: 4096, out: 1000)
Output (1 of 1000 classes)

Figure 10: Architecture of the *AlexNet Convolutional Neural Network* [8]. It consists of five conv layers, three max-pool layers with kernel size 3×3, and three FC layers. After each conv layer, a *ReLU* activation function is applied.

2.7.2 Vision Transformer (ViT)

The arising of *Vision Transformers* (ViT) goes back to the work of Dosovitskiy et al. [17]. They tested the applicability of well-known transformers on computer vision tasks.

Transformers are well-studied in the area of *Natural Language Processing* (NLP). They are already used in *ChatGPT*[8], for instance. Dosovitskiy et al. found out that a well-trained *Transformer Encoder* can extract good visual features for computer vision tasks.

Figure 11 illustrates the architecture of a *Vision Transformer* with a *Multilayer Perceptron* classification head. The classification head is replaceable by other

[8] ChatGPT: https://chat.openai.com (Online; Accessed February 21, 2024)

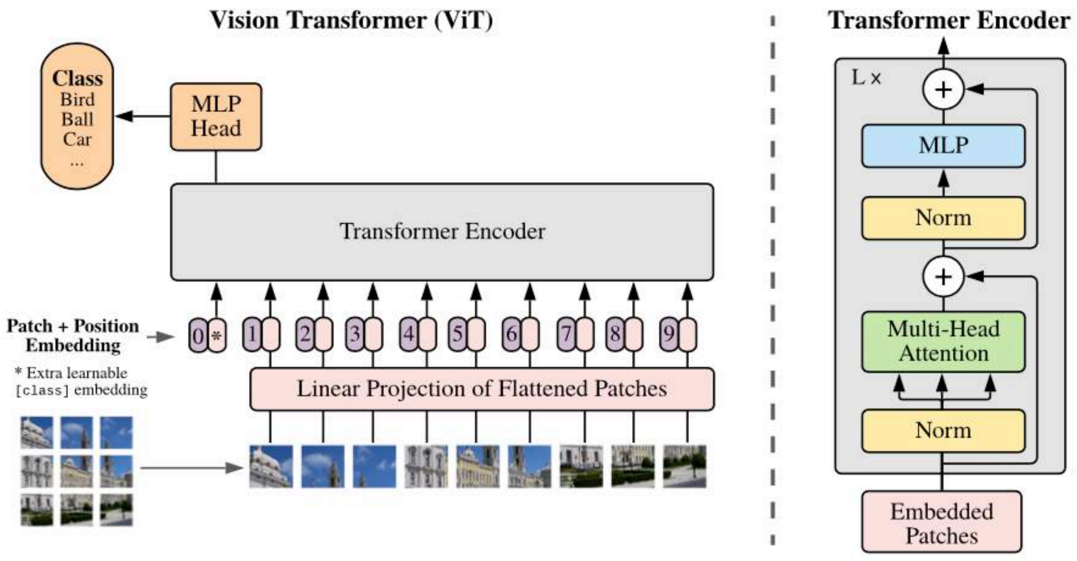


Figure 11: On the right: Architecture of a *Vision Transformer* on an object classification downstream task. On the left: Architecture of a *Transformer Encoder Building Block* [17].

processing heads for other vision tasks with ease. For instance, the classification head can be replaced by an image segmentation head. A ViT works as follows: First, the input image is divided into several patches of fixed patch size, which are subsequently flattened. The flattening has to be performed to transform the 2D or 3D image patches into the interpretable input for a *Transformer Encoder*. Next, linear projection is performed to produce lower-dimensional linear embeddings from the patches. The last step is the addition of positional embeddings to each flattened patch to create a relationship between the different patches. These positional embeddings for flattened image patches can be interpreted as the positional embeddings of words in a sentence in the context of NLP. Afterward, the sequences are fed as input to a standard *Transformer Encoder*, which generates vision feature vectors.

Although the *Transformer Encoder* is a standard one, it has to be pre-trained on a large dataset to generate meaningful and reliable visual features across different appearances of the same object in different images. The training happens in a fully supervised manner a priori to the actual downstream task. For instance, the *DINOv2* ViT models [1] are all distilled from a model that was trained on the *LVD-142M*, which consists of 142 million images.

Based on the transformer's visual features, an MLP is fine-tuned on a much smaller downstream dataset for image classification or similar. This MLP takes the visual features as input and predicts the label of the original image. To achieve high performance, the visual features have to be very distinct from each other for different objects but have to be very similar for the same objects.

The ViT's *Transformer Encoder* consists of blocks of *Norm Layers*, *Multi-Head Attention Networks*, and *Multilayer Perceptrons*. Furthermore, residual connections before each norm layer, after each multi-head attention layer, and multilayer perceptron are added. Several of these blocks are put together to create a complete *Transformer Encoder*.

"Attention functions can be described as mapping a query and set of a key-value pair to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key." [28].

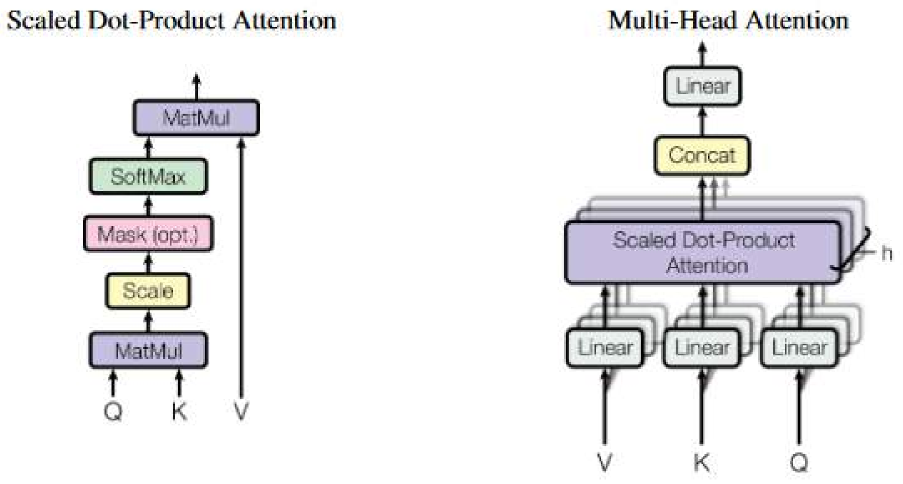


Figure 12: (left) *Scaled Dot-Product Attention*. (right) *Multi-Head Attention* consists of several *Scaled Dot-Product Attention Blocks* running in parallel [28].

The multi-head attention mechanism is illustrated in Figure 12. The multi-head mechanism consists of multiple scaled dot-product attentions, running all in parallel. The several output values of the multiple scaled dot-product attentions are concatenated and once again projected, resulting in the final values.

A distinct feature of *Vision Transformers* compared to standard *Transformers* is that a ViT does not need a decoder. The only thing that matters is the output of the encoder network. All downstream computer vision tasks are performed on the output of the *Transformer Encoder*.

2.8 Technologies

Different technologies are used to design, implement, and evaluate the automated data labeling pipeline. Each used technology is described briefly.

Python *Python* is a universal programming language with the main focus on readability and shortcode. It comes with dozens of useful packages and is a per-

fect fit for prototyping. In this work, several scientific and graphical packages are used, such as *pandas*, *PyTorch*, *Torchvision*, *Ultralytics*, and *Tkinter*.
Tkinter is a package that provides functionalities to build a graphical user interface. *Ultralytics* provides different machine learning models, such as *YOLO*, which are used in this work. The functionality to train a pre-trained model with a new classification head is provided too.
Furthermore, *Python* comes with an integrated *ROS* package, namely *rospy*. This package allows the direct integration of *ROS* into the *Python* environment without any trouble. Additionally, the package *rosbag* comes with the functionality to read driving recordings, which are saved as *ROS* bag files.

ROS *ROS* stands for *Robot Operating System* and is used to control the autonomous vehicle of the *Dahlem Center for Machine Learning and Robotics*. It is a framework for developing different kinds of robots, such as football robots or autonomous vehicles. It is universally applicable and easy to integrate into the *Python* environment. All driving recordings are saved as *ROS* bag files.

LabelImg *LabelImg* is a special *Python* package with an easy-to-use user interface. *LabelImg* provides functionality to store the labeled images in either *YOLO* or *COCO* format. Rectangles can be drawn around obstacles and one or more labels can be associated with bounding boxes. With *LabelImg*, its own training and validation dataset is created to evaluate the performance of several object detection models and to train the models on more data, if needed.

Draw.io *Draw.io* is a free software to create different kinds of diagrams, ranging from *UML* to general ones. It comes with support for several cloud storage, such as *OneDrive* by *Microsoft* or *Google Drive*. Besides its online presence, it is also a desktop application downloadable.

Training computer All models are trained on the same computer, in the same configuration. The system is powered by two *Nvidia* 1080 Ti GPUs with a combined 24GB RAM.

3 Automated Data Labeling Pipeline

In this chapter, the automated data labeling pipeline is proposed. The question of how the pipeline works, what data it takes as input, and what data it produces is answered.

In section 3.1, a special folder structure is introduced for storing the different kinds of data.

Figure 13 illustrates the proposed *Automated Data Labeling Pipeline*. The pipeline consists of several processing stages. The input data is processed by the *Data Extractor* which extracts the relevant data for the pipeline and stores them in a newly defined Folder Structure. The details of the *Data Extractor* are given in section 3.2. The defined folder structure is explained in section 3.1.

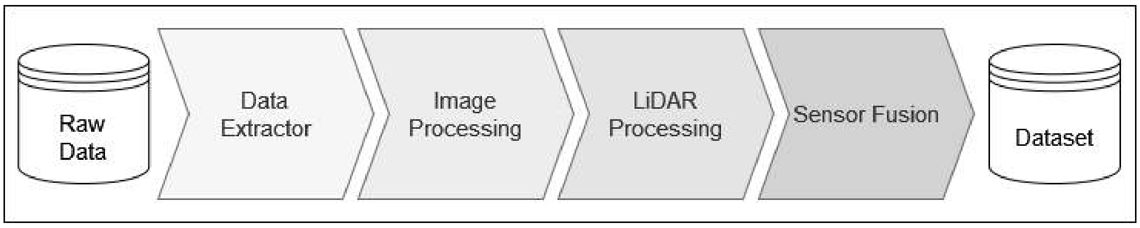


Figure 13: Illustration of the *Automated Data Labeling Pipeline*.

Following the *Data Extractor*, the extracted images are processed by the *Image Processing* stage, described in section 3.3. The generation of segmentation maps and execution of the object detection is performed in this stage.

Next, the extracted LiDAR data are processed in the *LiDAR Processing* stage, described in section 3.4. The LiDAR data are cleaned and objects are found in there.

The last stage is the *Sensor Fusion*, described in section 3.5. The results of the *Image Processing* and *LiDAR Processing* stages are fused. The output of the sensor fusion stage is then stored as a sensor fusion dataset usable in supervised sensor fusion applications.

The *Requirements* of the user interface and the implementation details of it, as well as a usage instruction, are described in section 3.6.

The complete overview, of how the stages are interconnected is described and visualized in section 3.7. Furthermore, it brings the chapter to a close.

3.1 Folder Structure

A special folder structure is introduced to save the different outputs of the automated data labeling pipeline. The folder structure is split into several subcategories, defined by the different stages of the pipeline. The data held by a *ROS* bag file are stored in a special folder for each processed *ROS* bag, determined by the bag's name. The camera images are further divided by the orientation of the camera. This means that all front and rear camera images are stored in separate subfolders of the image main folder.

The LiDAR scans are stored in the sub-folder *lidar*.

The labels of the object detection models are stored in the subfolder *labels*.

The different segmentation maps are also stored in the subfolder *segmentation*.

The folder structure with an example *ROS* bag file is shown in Figure 14.

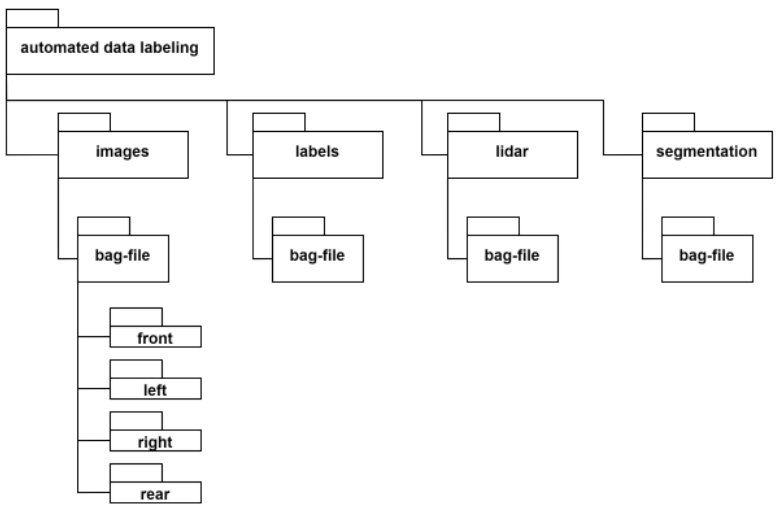


Figure 14: Proposed folder structure with an example *ROS* bag file.

The data of each *ROS* bag are stored in separate folders, determined by the bag's name. This is done due to the needed mapping between LiDAR scans and the corresponding camera images. Each LiDAR scan is describable by four camera images. One for each perspective.

Another issue is that the LiDAR and the cameras have different refreshing rates. LiDAR operates on 10Hz, while each camera operates on 30Hz. This means that the LiDAR scans every 0.1 seconds the environment, while a camera takes every 0.033 seconds an image. Furthermore, the cameras do not take the pictures at the same time. Hence, a LiDAR scan has to be mapped to the closest image of each camera regarding the timestamp of the LiDAR scan.

Furthermore, a separate folder makes the processing easier because the data of different bag files do not have to be renamed. For instance, an image of the front camera has the name *broadrreachcam_front_1461942440681981837*. It's much simpler to access the images of one *ROS* bag through the bag's name instead of saving all images to one folder without structure.

In the end, each *ROS* bag is interpreted as a sequence of the later dataset.

3.2 Data Extractor

The *Data Extractor* is the first stage of the automatic data labeling pipeline. It consists of a *ROS* bag file reader and a data extractor.

The *ROS* bag reader takes a bag file containing a driving recording as input and extracts the compressed images from the front, right, left, and rear cameras as well as the LiDAR data. These data are passed to and processed by the data extractor. The data extractor rectifies the compressed images and extracts the LiDAR data to make them usable at later stages.

Implementation Details Each camera is equipped with a fisheye lens and produces compressed images. The fisheye lens distorts every captured image. To use the image properly, the rectification of each one is needed. The rectification works as follows. First, the rectification matrix, camera matrix, and distortion coefficients corresponding to the given camera orientation are loaded from the *fub_mig*-repository[1], a sub-project of the *Autonomos Labs GitLab*. Next, the image is loaded from the *ROS* bag file as a compressed image message. The compressed image message is converted to *cv2*. Afterwards, the rectification matrix, camera matrix, and distortion coefficients are applied to the image, resulting in the rectified image. At last, the processed image is stored on the host system. The rectification of an example image is shown in Figure 15.
The extraction of the LiDAR data requires the use of an additional python package, named *velodyne_decoder*[2]. The decoder works on different *Velodyne* models and applies the corresponding configuration file to the point cloud data directly. It takes the *velodyne_packets* messages as input. The decoded message is transformed into a pandas DataFrame and subsequently saved as csv-file on the host system. Besides the raw data, a summary file for each folder is generated, containing the timestamp of each file and the corresponding file name.

[1] fub_mig git-repository: https://git.imp.fu-berlin.de/autoauto/fub_mig (Online, Accessed: February 21, 2024)

[2] velodyne_decoder: https://pypi.org/project/velodyne-decoder/1.0.1/ (Online, Accessed: February 21, 2024)

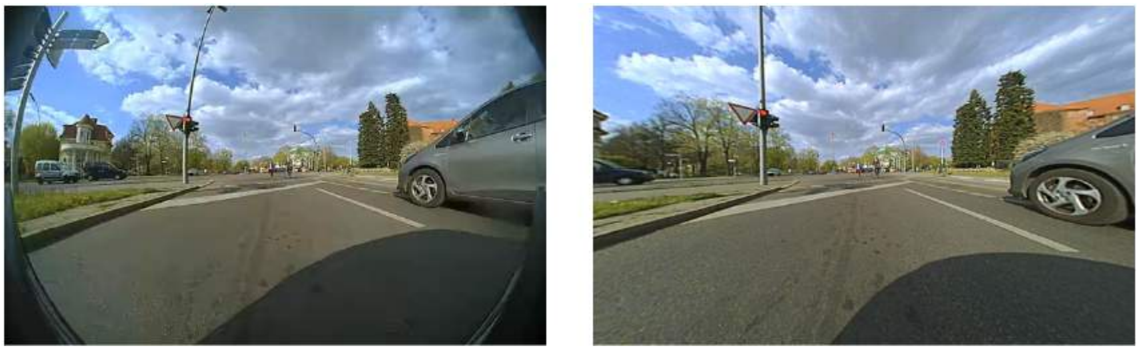


Figure 15: Rectification results based on an example image. Note the imperfection of the rectification process: (left) Distorted image taken with a camera, equipped with a fisheye lens. (right) Same image, but after rectification.

I have to note that the rectification of an image is not perfect. This can be seen in Figure 15. In the left image, the traffic sign is distorted and the blue car on the left-hand side is squashed. Compared to the right image, the traffic sign is straightened and the blue car on the left-hand side is rectified. But the car's wheel on the right side of the image is distorted. This is not the case in the distorted image on the left side.

Another issue is that the field of view of the right image is much smaller compared to the left one. This is obvious because the beige house on the left side is missing in the right image.

3.3 Image Processing

The *Image Processing* consists of two individual tasks. A segmentation task to generate segmentation maps and an object detection task to detect different objects within an image. Each task can be processed on its own and computed on the before-processed and rectified images.

Following, I will describe each task individually. Furthermore, I explain how the different outputs can be used for different stages in the pipeline and how they are used at the end.

3.3.1 Semantic Segmentation

The *Semantic Segmentation* is performed at first in the image processing stage. It takes an image as input and returns a granular segmentation map based on the following semantic categories. *Unlabeled*, *Road*, *Sidewalk*, *Building*, *Wall*, *Fence*, *Pole*, *Traffic Light*, *Traffic Sign*, *Vegetation*, *Terrain*, *Sky*, *Person*, *Rider*, *Car*, *Truck*, *Bus*, *Train*, *Motorcycle*, and *Bicycle*, following the class definition of the *CityScapes* dataset [29].

The segmentation model consists of a feature extractor and a segmentation head. The feature extractor's goal is the creation of reliable and meaningful features of several image patches. The features should be distinguishable from other ones, such that a categorical connection between different patches can be found. For instance, the features of a patch containing a car should be distinguishable from a patch of the sky. This should apply to patches of different orientations, sharpness, and illumination.

On top of the feature extractor, a segmentation head is applied. It takes the generated features of the feature extractor as input and predicts the category of each patch. Afterward, the predictions are assembled such that a segmentation map is gathered.

Following, I describe several implementation details, the model's architecture, the training and validation routine, and the detailed processing of an image.

Furthermore, I give two approaches to how the segmentation map can be used.

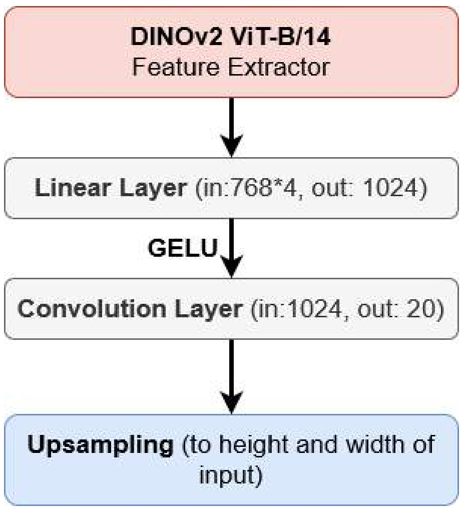
Implementation Details I've implemented semantic segmentation on top of a *DINOv2* model due to its robust and reliable visual features. Different *DINOv2* models exist and are trained on a large dataset (*LVDS-142M*) consisting of 142 million different images. They produce reliable, meaningful, and distinct features for semantic segmentation or classification problems.
DINOv2 is originally applied to a *Vision Transformer* (ViT) with a patch size of 16x16 pixels, while publicly available pre-trained models are ViTs with a patch size of 14x14 pixels. The publicly available models are distilled from the model with the larger patch size.
As the vision feature extractor the *DINOv2* ViT base model with patch size 14, referred to as *DINOv2 ViT-B/14*, is used. The selection of this model is a trade-off between memory consumption and relative performance gain compared to the other models. Table 1 shows the memory consumption and the performance on a semantic segmentation task on the *CityScapes* dataset for each model respectively. The ViT-B/14 model consumes around 330MB storage with its 86 million parameters and achieves 80.0% mIoU on the *CityScapes* dataset in the multi-scale setting. The much larger ViT-L/14 model achieves a relative performance gain of +0.9% compared to the base model while consuming 1.13GB of storage. Besides the model's size, the patch size is an important factor for the performance. The shown performance on the semantic segmentation task is achieved with a ViT with a patch size of 14x14. Unfortunately, the authors did not release the ViT with a patch size of 16x16, so I'm bound to use ViTs with a smaller patch size. However, with a smaller patch size, a performance boost is possible regarding the mIoU. This is because the mIoU is calculated by comparing the ground truth data with the predicted segmentation map on the pixel level. A finer segmentation map can be generated if the image size is fixed but the patch size decreases. This is because the image is split into more patches than previously. But this comes with

the cost of the need for more computational resources because more patches have to be computed.
On top of the *DINOv2* base model, a segmentation head consisting of one linear layer and one conv layer with kernel size one and stride one. Between the two different layers, a *GeLU* activation function is applied [30]. The conv layer functions as a classification layer, classifying the different patches into one of 20 distinct categories.

			mIoU	
	Architecture	Memory	lin.	+ms
DINOv2	ViT-S/14	21M Parameters	66.6	77.1
	ViT-B/14	86M Parameters	69.4	80.0
	ViT-L/14	0.3B Parameters	70.3	80.9
	ViT-g/14	1.1B Parameters	71.3	81.0

Table 1: Memory Consumption and Performance of the different *DINOv2* models on semantic segmentation task on *CityScapes* validation data split with a linear classifier (lin.) and with multiscale (+ms) [1].

Model Architecture The model's architecture is presented in Figure 16.
The *DINOv2* backbone takes an image as input and outputs the patch tokens of the last four ViT layers, following the analysis of the authors of [1]. Afterward, the gathered patch tokens are concatenated to serve as input to the segmentation head.
The architecture of the segmentation head is experimentally determined. It consists of one linear layer and one conv layer that determines the corresponding class. The input size of the linear layer is determined by the embedding dimension of the *DINOv2* base model. I'm using the output of the last four layers, following the idea of [1], resulting in four times 768 neurons as input size. The embedding size is multiplied by four because each *DINOv2* layer outputs a feature vector of size 768. The corresponding output size of the first layer is 1024.
The linear layer's output is reshaped to an image-like form of height tokens times width tokens such that the output of the final conv layer can be resized to the input image's size afterward. After the reshaping, a conv layer with an input size of 1024 and an output size of 20 is applied to classify each patch token. The conv layer takes a single patch at once and outputs the corresponding category. The complete segmentation is gathered by running all patches through the conv layer. The last step of the segmentation head consists of upsampling the output of the conv layer to the size of the input image to retrieve the corresponding segmentation map.
The model is implemented in such a way that images of different aspect ratios can be processed.



DINOv2 ViT-B/14
Feature Extractor
Linear Layer (in:768*4, out: 1024)
GELU
Convolution Layer (in:1024, out: 20)
Upsampling (to height and width of input)

Figure 16: Final segmentation head to generate the segmentation maps.

Training To train the implemented segmentation head on top of the *DINOv2 ViT-B/14* model, the *CityScapes* dataset is used. The training split of the dataset contains 2975 images with corresponding segmentation masks as ground truth data.
As a penalty criterion, *CrossEntropyLoss* with initialized weights is used. The weights of the *CrossEntropyLoss* are initialized due to the unbalance of the number of finely annotated pixels per class in the training split. Figure 17 shows this unbalance. The optimizer is *Stochastic Gradient Descent* (SGD) with a static learning rate of 0.001 and momentum of 0.9. Each parameter was experimentally determined. The model is finally trained for 100 epochs to get the best model possible.
Dealing with the insufficient number of training samples requires several data augmentation techniques. First, the sample is randomly resized by a factor between 1.5 and 4.5 of the original image size. The random resizing follows the advice of [1], where multi-scale training is applied to achieve a performance boost. Following, random mirroring and random Gaussian blur are applied. The last fix augmentation is a random crop, selecting randomly a portion of the resized image of size 672x672 pixels corresponding to 48x48 patches.
After the standard augmentations, two more augmentations, randomly selected from an augmentation pool, are applied. The idea of the augmentation pool goes back to the work of Cubuk et al. [31]. The authors experimented with several augmentations and measured if a performance gain was achieved.
The augmentation pool consists of the following augmentations. The identity augmentation returns the original image patch. The sharpness augmentation changes the sharpness of an input image by a factor between -2 (less sharp) and $+5$ (sharper). The next selectable augmentation regards the brightness value of an image. Its factor lies between -2 (less bright) and $+2$ (brighter). The last color value regarding augmentation is the gray scaling of an image. The last possible augmentation is a rotation by a random angle between 1 and 45 degrees.
For each training image, two additional augmentations are applied to artificially

generated unseen and new images. The augmented image is the model's input. The layer's weights are updated by the mean loss with backpropagation. After applying the backpropagation, the optimizer is applied.
The model is trained for 100 epochs to gather the best possible model. Furthermore, the individual checkpoints are saved such that one can start from the latest stage if one wants to retrain the model with other or more data.

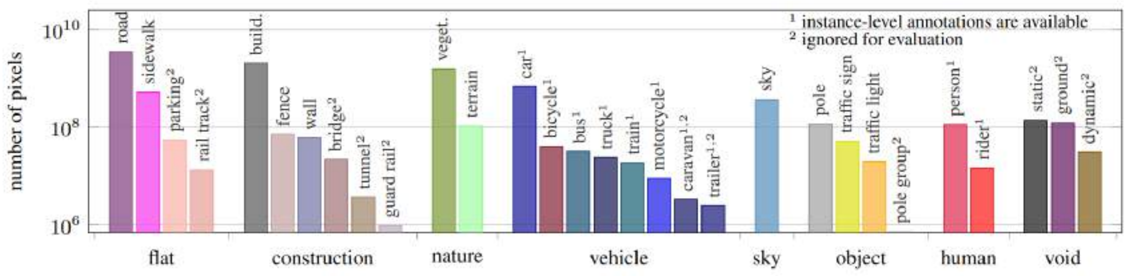


Figure 17: Number of finely annotated pixels (y-axis) per class and their associated categories (x-axis) [29].

Validation The validation is carried out after each training epoch. The corresponding dataset split of the *CityScapes* dataset consists of 500 different images with additional segmentation masks.
For the validation process, each image is randomly resized by a factor between 1.5 and 4.5 of the original image size. Afterward, a random crop with a crop size of 672x672 is applied.
The performance of the model is tracked during each validation run to find the best model over the whole training routine. The performance for each category is tracked, as well as for the whole predicted segmentation map. For both evaluations, the *mean Intersection over Union* (mIoU) of the predicted segmentation map and the ground truth data is calculated. Furthermore, the *Intersection over Union* (IoU) of each class is calculated separately.
Each evolution is tracked as a checkpoint and the corresponding evaluation data is stored in a txt-file. The model with the best performance is defined by the highest mIoU. The mIoU is tracked over the whole training process. If the current model has a higher mIoU than the best model, the current model is considered to be the best. The model weights are stored separately from the checkpoints.

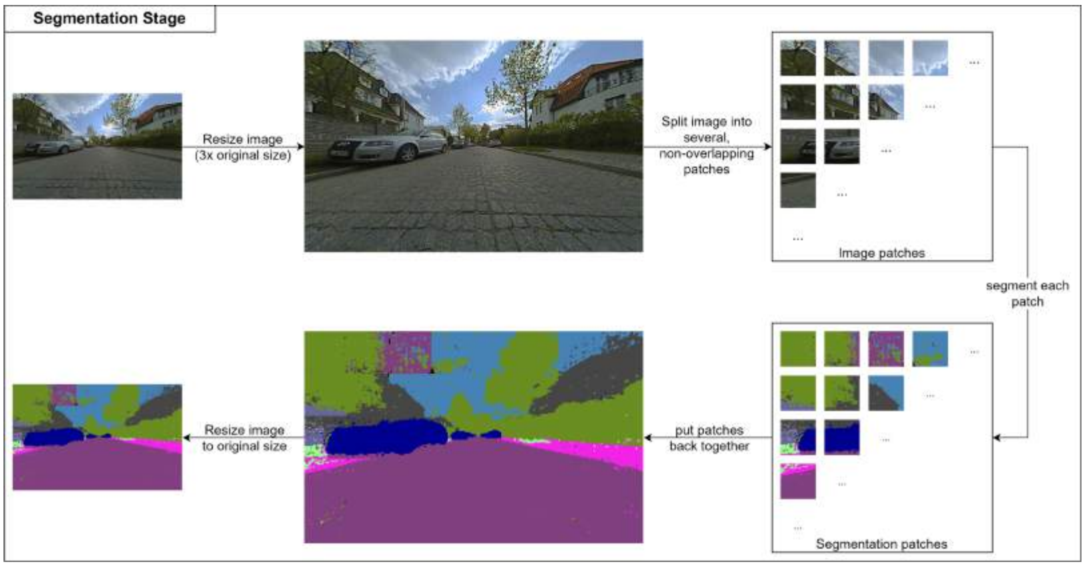

Segmentation Stage
Resize image (3x original size)
Split image into several, non-overlapping patches
Image patches
segment each patch
Segmentation patches
put patches back together
Resize image to original size

Figure 18: Workflow of the segmentation stage.

Segmentation Process After training and validation, the semantic segmentation model can be applied to images of different *ROS* bag files.
The segmentation is illustrated in Figure 18 and works as follows. First, the input image is scaled so that it has the same aspect ratio as the training data (two-to-one). Otherwise, the image would be overdrawn with lines. Figure 19 shows this issue using the original image size. Next, the image is resized with its new height and width by the formula

$$w = round(old\_width/14) \cdot 14 \tag{3.1}$$
$$h = round(old\_height/14) \cdot 14 \tag{3.2}$$

such that the height and width are a multiple of 14 and have the same shape as the original training data. The old height and width are divided by 14, rounded, and multiplied by 14. This has to be done due to the required patch size of the *DINOv2* models. Otherwise, the ViT cannot proceed with the image because the image patches cannot be generated. This leads to the possibility that the processed image may be a bit smaller or larger than the original one.
Next, the input image is interpolated by a factor of three to artificially decrease the patch size of the model. An interpolation by a factor of three leads to a factual patch size of 4x4 pixels per patch instead of 14x14 pixels. This can be done because the model was trained with multi-scale training ranging between a factor of 1.5 and 4.5. If the original image size were used, the segmentation map would become very pixelated (big chunks of one category) and the edges of the objects would not be as clear. The resizing leads to a much finer segmentation map.
The Figures 20, 21, 22, and 23 illustrate the difference in the input image's size and the resulting segmentation maps. It is clear to see that the segmentation map without interpolation more boxy is than the other ones. The edges are better distinguishable from the other segmentation categories.
Next, the input image is split into multiple chunks of size 672x672 pixels, corresponding to the random crop size of the training and validation routines. This is

mostly done for resource reasons. If the complete interpolated image is fed into the model, the RAM of the CPU and GPU overflows.
Each interpolated image crop is processed by the segmentation model individually. The obtained segmentation maps of each piece are put together into one segmentation map, representing the segmentation map of the complete input image. The final segmentation map is stored in the segmentation subfolder of the pipeline's folder structure.

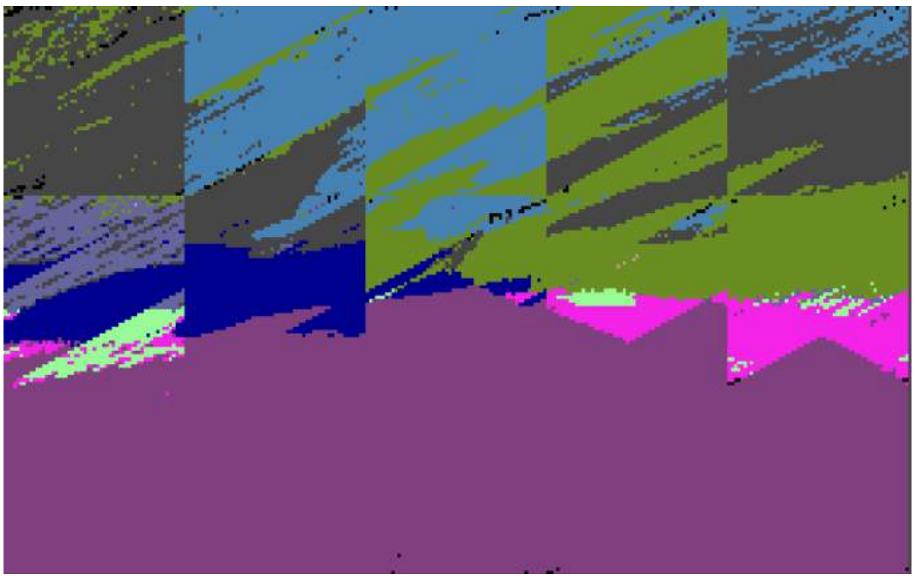


Figure 19: Segmentation map issues if the aspect ratio of the image is not the same as of the training data. A reason for this could be that the crop of the image is square, benefiting images with an aspect ratio in which a square can perfectly fit.

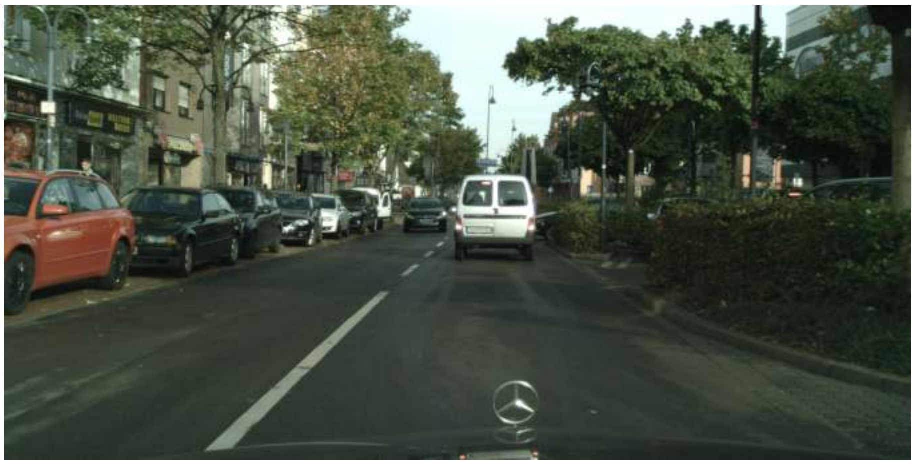


Figure 20: Example image on which the segmentation based on different interpolation factors is performed. The image is taken from the *CityScapes* dataset.

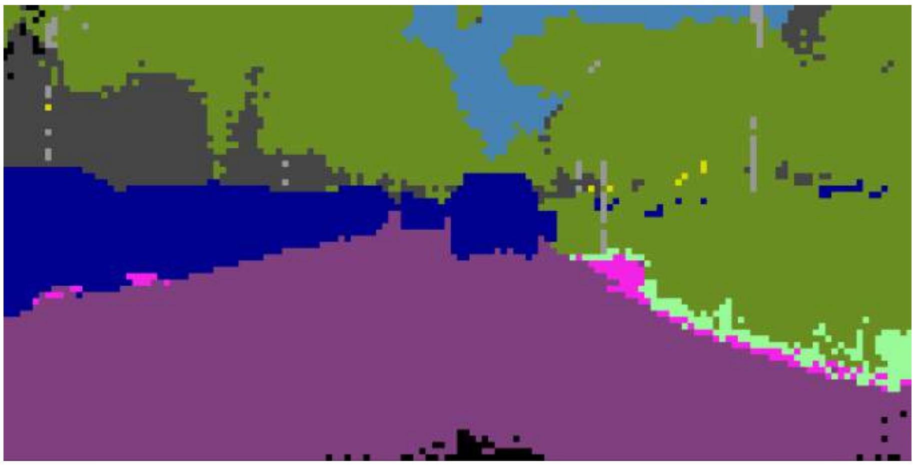


Figure 21: Segmentation map of the image without interpolation.

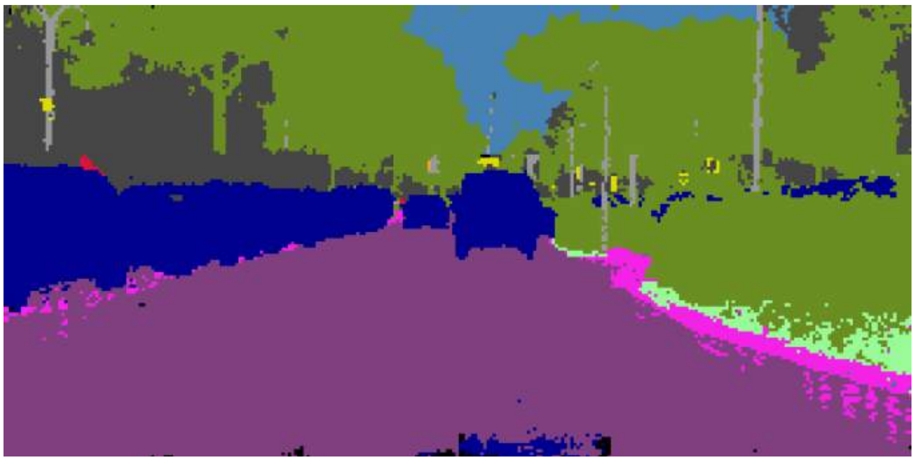


Figure 22: Segmentation map of the image with interpolation by a factor of 2.

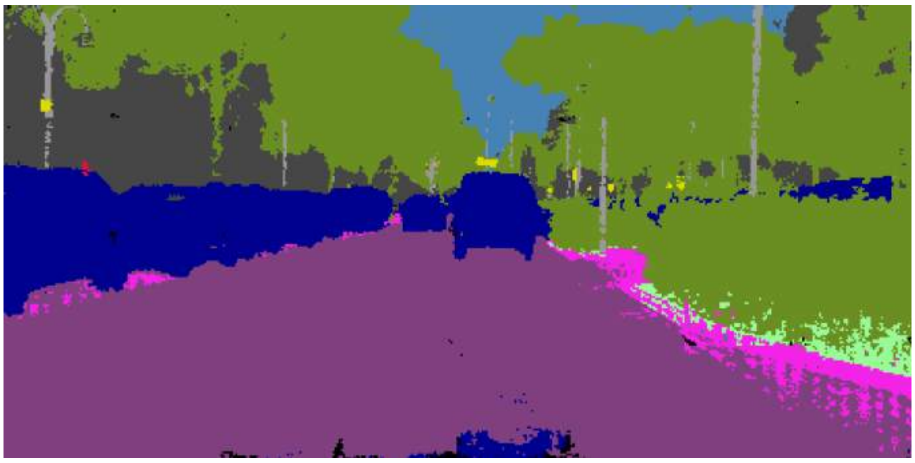


Figure 23: Segmentation map of the image with interpolation by a factor of 3.

Applicability I developed two application ideas for the segmentation map of an image. The first idea is to use the segmentation map to retrieve only the rele-

vant parts of an image. For instance, only extracting all parts that lay on the car category of the segmentation map. The extracted parts are then the input of the object detection stage, where only specialized detection models are applied. For instance, a specialized model, only trained on car data, is applied to the aforementioned parts of the image.
The second idea regards a further filtering stage for point cloud data based on the segmentation map. LiDAR and radar data contain several points that are not relevant for object matching in the sensor fusion stage. For instance, points that lay on buildings or fences are not relevant for the sensor fusion but are relevant to be found as objects in the LiDAR or radar data. These points should be removed before the object matching between camera and LiDAR or camera and radar objects is carried out.

3.3.2 Object Detection

The main task of the object detection stage is to find different objects in the camera images. The objects are categorized into vehicles, pedestrians, traffic signs, and traffic lights. For each category, a specialized model is trained on a corresponding dataset.

Datasets

Different Datasets are used to train the object detection models. Following, each used dataset is described briefly. Furthermore, advantages and disadvantages are presented, and if needed, extensions to the dataset are explained.

German Traffic Sign Detection Benchmark The *GTSDB* [5] was introduced on the IEEE in 2013. It is a single image detection dataset for use in computer vision, pattern recognition, and image-based driver assistance. It consists of 900 images, divided into 600 training and 300 evaluation samples. The images are taken from an ego-vehicle's point of view. The authors defined 42 different classes that they annotated across the dataset.
It is the only publicly available dataset that contains only German traffic signs. Compared to other datasets for vision tasks, it is very small, containing only 900 images in total. Furthermore, Germany has approximately over 200 different traffic signs, while the *GTSDB* only represents a fraction of it. The selection of the annotated traffic signs seems to be random because some signs exist in different directions, but the dataset contains only one of them. Additionally, the names of the traffic signs are not right either.
To mitigate these issues, the *GTSDB* is extended and the class definitions are redefined based on the official German traffic sign catalog. Furthermore, more different types of traffic signs are added. The newly created dataset is available at *https://git.imp.fu-berlin.de/taegenee98/gtsd*.

nuImages *nuImages* [4] is a special split of the original *nuScenes* dataset by providing 93.000 2D annotated images from a much larger pool of data. Several camera images are provided, resulting in a total of 1.200.000 camera images

at the *nuScenes* dataset. The *nuImages* dataset is available, like the *nuScenes* dataset, as free to use strictly for non-commercial purposes.
The data annotation for the 93.000 images with 2D bounding boxes results in 800.000 foreground objects and 100.000 semantic segmentation masks. *nuImages* provides several main classes with many sub-classes. For instance, the class pedestrian is split into an adult, child, construction worker, personal mobility, police officer, stroller, and wheelchair. The split into several sub-classes is a good idea because an autonomous vehicle should react differently if different pedestrians walk onto the road or different pedestrians are recognized. For instance, a child does not always act rationally.
Based on the *nuImages* dataset, several specialized object detection models are trained. For instance, an individual pedestrian and vehicle model are trained. The classes of each model correspond to the sub-classes of the dataset.

Traffic Lights The traffic light dataset[3] is a publicly available dataset for traffic light recognition. It represents several different traffic light types straight, only right, only left, and more. Furthermore, the authors of this dataset defined the traffic light phases.
The dataset consists of 3.000 images, where 2.600 belong to the training split and 400 to the validation split.
The central issue of this dataset is that it does not consist of only German traffic lights and is mixed with other traffic lights from other countries. Furthermore, the only labels that are added are the traffic light phase and not the pictogram of the traffic light. The pictograms are also interesting because some traffic lights only indicate that one is allowed to go straight or left.

Implementation Details

To realize the object detection, I use the *YOLOv8* model from *Ultralytics*. The *YOLOv8* model is based on the original *YOLO* paper and is much faster and more precise than all other models.

I've trained four models. One for pedestrian detection, one for vehicle detection, one for traffic light detection, and one for traffic sign detection.

The models are trained on the aforementioned datasets. The number of epochs varies from model to model because the datasets have different sizes.

To detect all objects in an image, the different models are processed individually and their results are put together at the end. This process is illustrated in Figure 24.

[3] Traffic Light Detection Dataset: URL: https://www.kaggle.com/datasets/wjybuqi/traffic-light-detection-dataset (Online; Accessed: February 21, 2024)

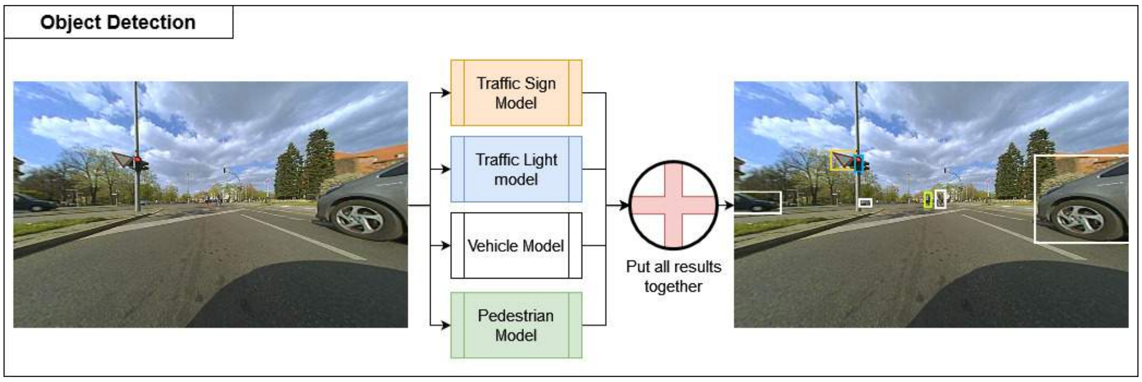


Figure 24: Object Detection Stage. The different colors of the bounding boxes indicate the results of the different models, put together.

The model's output is saved in *YOLO*-Format in the *label* folder.

The *YOLO*-Format is a definition of how the results of the object detection are stored. An object is saved as one line of a text file. The values, except the first value, are all relative to the width and height of the image. The first entry is the label, represented by an integer. The second and third values describe the center point of the object in the image. The value varies between zero and one. The fourth and fifth values describe the width and height of the object. For instance, the line *'0 0.5 0.5 0.9 0.9'* describes an object with label 0. Its center is in the center of the image, described by *'0.5 0.5'*. Its width and height are 90% of the image's width and height, described by *'0.9 0.9'*. All in all, the object in question fills almost the entire image.

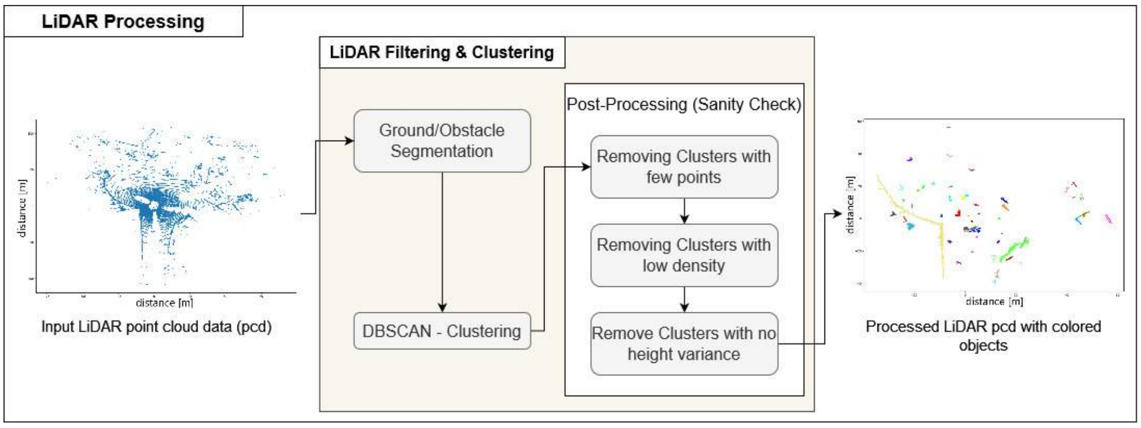
3.4 LiDAR Processing

The LiDAR Processing consists of two successive tasks. The first one can be summoned under LiDAR Filtering. The second one is summoned as clustering and object detection within the LiDAR data.

At last, the clustered LiDAR data are put into object tracking to track an object across consecutive frames.

Following, I'll describe every task in great detail. Furthermore, I give insights into the implementation and compare different approaches for LiDAR Filtering.

The complete LiDAR processing is illustrated in Figure 25.



LiDAR Processing
LiDAR Filtering & Clustering
Post-Processing (Sanity Check)
Ground/Obstacle Segmentation
DBSCAN - Clustering
Removing Clusters with few points
Removing Clusters with low density
Remove Clusters with no height variance
distance [m]
distance [m]
Input LiDAR point cloud data (pcd)
distance [m]
distance [m]
Processed LiDAR pcd with colored objects

Figure 25: LiDAR Processing overview.

3.4.1 LiDAR Filtering

The filtering of LiDAR points is necessary because the sensor does not distinguish between relevant and irrelevant detections. Irrelevant detections can be points that lay on the ground or noise generated by exhaust gases for example. These ground and noise points are useless for the detection of objects within the LiDAR data. They would make the object finding much harder.

Two approaches to segmenting ground points from obstacle points are evaluated. The approaches are distinct from each other and use different data and techniques to achieve ground segmentation. Following, both approaches are described in great detail. Furthermore, the selected approach for the automatic data labeling pipeline is named based on different criteria.

Following, I describe each filtering step in great detail and give insights into the implementation.

Ground Segmentation using Markov Random Fields

The first approach is based on the *Ground Segmentation Algorithm for Sloped Terrain and Sparse LiDAR Point Cloud* paper [32] and consists of two subsequent steps. A channel-based initial classification is followed by a ground map and final classification, resulting in an obstacle-ground segmentation.

The first stage is a channel-based initial classification. Channels are created by dividing the 360-degree scan into fixed portions of fixed angular size. The authors used a 2-degree split value. This means that the 360-degree scan is split into 180 channels of 2-degree horizontal representation each. After the channel creation, the points are sorted by their relative distance to the LiDAR sensor in ascending order. Next, several geometric features are evaluated on each channel, searching for ground and obstacle evidence. The algorithm, shown in Algorithm 1, decides if a point is a ground or an obstacle point based on the following heuristic rules.

Algorithm 1 Channel Ground Segmentation [32].

```
1: Input: Channel's points
2: Output: Obstacle-ground labels
3: $l_0 \leftarrow Ground$
4: for all $p \in C$
5:     if $CheckNoise(p)$
6:         continue
7:     end if
8:     switch $l_{(i-1)}$ do
9:         case $Ground$
10:            $l_i \leftarrow CheckObstacle(p_i, p_{(i-1)}, p_g)$
11:        end case
12:        case $Obstacle$
13:            $l_i \leftarrow CheckGround(p_i, p_{(i-1)}, p_g)$
14:        end case
15:        case $Doubt$
16:            $l_i \leftarrow CheckBoth(p_i, p_{(i-1)}, p_g)$
17:            if $l_i \neq Doubt$
18:                $CorrectDoubtPoints(l_i)$
19:            end if
20:        end case
21:    end switch
22: end for
```

Heuristic Rules The following two heuristic rules are defined to find obstacle evidence. Only one of them has to be met such that a point is considered an obstacle.
The first one considers the maximum allowed slope. The gradient between the current point and the previous one is calculated. If the gradient exceeds a threshold α_Θ, the current point is considered to belong to an obstacle.
The second considers the distance of the current point compared to the previous one. If the current point is closer to the sensor than the previous one, the current point is considered to belong to an obstacle.
Both rules can be triggered by ground imperfections or irrelevant, small objects, such as bumps, grass, or curbs. To avoid false classification, found obstacle evidences have to be confirmed by a relevant height difference. This height difference is calculated between the tentative obstacle point and the last ground point. If the difference exceeds a threshold α_h the point is considered as an obstacle, otherwise it is classified as doubt.
Likewise, three heuristic rules for ground evidence are defined. But contrary to the obstacle rules, the ground rules have to be simultaneously met to consider a point as ground.
First, the expected distance is checked. The assumption is that the current point is farther away from the sensor than the last ground point.
The second rule considers a height reduction. This means that the height of the current point has to be lower than the height of the previous point. This may mean

that the obstacle is over and the ground is hit again.
The third rule checks if the current point has a similar height to the last classified ground point.
If doubt points are not solved at the end or before a certain distance, the obstacles evidences are assumed not to be strong enough and, therefore all doubt points are corrected to ground.

The second and final step is the re-classification of the point cloud data. The re-classification is performed based on the solution of a *Loopy Belief Propagation* algorithm for a *Markov Random Field*.

The first step of the re-classification is the creation of a polar grid map, which is interpreted as an undirected graph, corresponding to a *Markov Random Field* with four neighbors: Forward, backward, clockwise, and counterclockwise. Each cell defines a portion of the surrounding area in terms of angle and radial distance ($\Delta\Theta$, Δr). The cell holds the information on the points' height and initial classification from the channel-based initial classification.

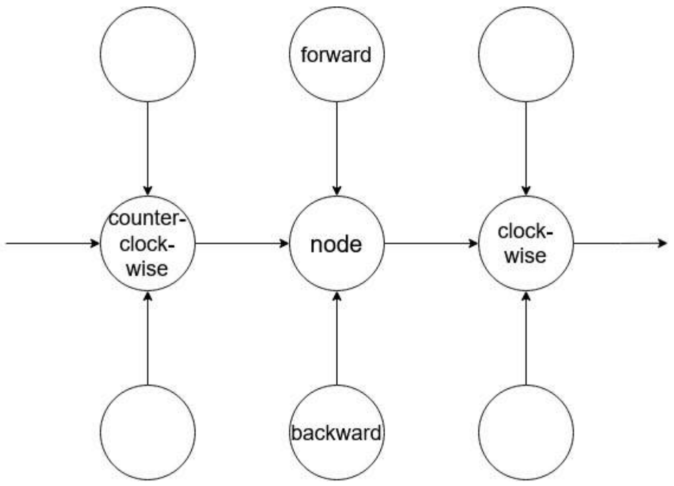


Figure 26: *Loopy Belief Propagation*: Illustration of the message sent from the node *node* to the node *clockwise*. The node *node* has to wait until it receives the messages from the nodes *counterclockwise*, *forward*, and *backward*.

Next, the *Loopy Belief Propagation* (LBP) algorithm is applied to solve the MRF. The LBP sends messages through the graph, but only if the current node has received all messages from its neighbors, except from the one it wants to send the message. Figure 26 illustrates the message sending. The node wants to send its message to its neighbor clockwise. It has to wait until it receives all messages from its neighbors forward, backward, and counterclockwise. If this is the case, the node sends its message to clockwise. To meet the assumption that a node has to wait for all messages to arrive before the node can send, all messages of the MRF are initialized beforehand. This guarantees the successful execution of the LBP.

After a certain number of iterations, the LBP stops and the belief vector for each node is calculated. The label with maximum belief (minimum cost) is selected as an optimal label. The label represents the height value of a certain cell.

In the final step, the LiDAR points are re-classified by comparing their height with the optimal label from the MRF solution.

Implementation Details The channel-based initial classification is implemented with a for loop, iterating over all channels. The points are sorted ascending by their distance to the sensor. For the classification, the algorithm, presented in Algorithm 1, is applied to each point of each channel. The result is stored as an additional *pandas* column to the DataFrame, containing all LiDAR points.
The parameters and threshold values for the initial classification are taken from the paper [32].
The polar grid map for the final re-classification is represented as a graph. For this, a dictionary is used for storing each node with its corresponding values and neighbors. After inserting all points, the LBP algorithm is applied to the polar grid. The LBP sends messages around the defined graph in the order *forward*, *clockwise*, *counterclockwise*, and *backward*. Following the authors' advice, all messages of each graph's node are initialized with zero.
After five iterations of the LBP, the belief vector of each graph is computed. The belief vector shows values for all possible height values. We are interested in the height with the lowest value. Based on the belief vector's result, the LiDAR points of each node are re-classified regarding the new ground height.

Ground Segmentation using Height Variance

The second approach to achieve ground segmentation uses a *GroundGrid* [2] and different values computed by the points' height. Compared to the first approach, it is a much simpler one and much easier to implement. Although an open-source implementation exists, it can't be directly used because of the special use case in the pipeline. The original *GroundGrid* is implemented for the segmentation o sequential data, using the odometry data of the car, where I proceed only one point cloud at once, without positional information. This means I had to implement the *GroundGrid* on my own and adjust it to the pipeline's use case.

The space around the autonomous vehicle is interpreted as a grid where each cell has a size of length times width. Furthermore, several layers are defined on top of the grid map. These layers have the same structure as the original grid map but hold different information. The layers are min ground height, max ground height, average ground height, point count for each cell, and height variance.

The aforementioned information is used to generate a terrain elevation map that is used to segment the point cloud data into ground and non-ground points. To do this, the ground height of each cell of the grid map is attached to the terrain elevation map.

Implementation Details The space around the sensor is split into fix-sized cells of size 0.33 meters times 0.33 meters, creating the ground grid. The width and length of each square are experimentally determined by the authors. It is a compromise between accuracy and computational performance.
The maximum covered distance for the *Velodyne* models HDL-64E and VLS-128 are used to limit the size of the ground grid. The maximum detection distance of an HDL-64E sensor is 120 meters, which is equivalent to a ground grid size of 726x726 cells. In contrast, the VLS-128 sensor covers 245 meters, corresponding to a ground grid size of 1.484x1.484 cells.
In advance of the rasterization of the point cloud data, an outlier detection is performed, where points are removed that lie under the already known ground. These points arise if a light pulse is reflected to the ground by a car. Because of the ToF principle, the LiDAR sensor computes the point much lower than it truly is. These miss-computations are the most harmful points because they can corrupt the height estimation at the end.
Following the outlier detection, the point cloud is rasterized and the aforementioned layers are updated, such as min and max ground height, the number of points in a grid cell, the average height, and the height variance inside the cell.
At last, the ground height is estimated for each cell, based on the direct neighborhood. The neighborhood is either a 3x3 square cell or a 5x5 square with the current cell as the center cell, depending on the distance to the sensor. Based on the ground estimation, the point cloud is classified and a ground label is assigned.

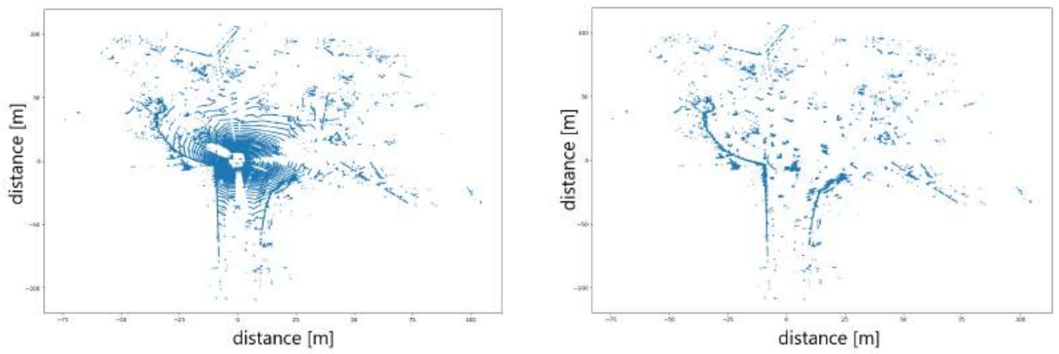


Figure 27: Ground/Obstacle segmentation based on the *GroundGrid*. left: Input point cloud data with ground points. right: point cloud data without ground points. The x-axis and y-axis represent the distance [m] from the ego-vehicle.

Figure 27 illustrates the results of the ground segmentation approach based on the *GroundGrid*.

To filter the LiDAR point cloud data of the *Dahlem Center for Machine Learning and Robotics*, the *GroundGrid* implementation is used.

3.4.2 LiDAR Clustering

The second step of the LiDAR processing is the clustering of the remaining points with an adaptive *Density-Based Spatial Clustering of Applications with Noise* (DB-

SCAN) algorithm for LiDAR points. This version of *DBSCAN* is based on [3] and estimates the parameters eps and minPts dynamically.

The estimation of feps is expressed in Equation 3.3. The first term considers the increase of the eps parameter based on the distance d of the selected LiDAR point to the LiDAR sensor as origin. One can assume that the LiDAR point cloud of an object gets sparser the further away the object is and gets denser the nearer the object is. The distance d is the Euclidean distance from the point to the data origin. The second term takes the density per square meter into account with density describing the density value of the corresponding cell of the grid map used in the filtering step. Both terms are inspired by the adaptive *DBSCAN* paper [3], where the authors found that eps is proportional to the distance and proportional to density^{-1}. The growth factor $g = \text{ln}(1.077938182482261)$ is determined by analyzing the distance between the ground point rings of the *Velodyne* sensor. The underlying and used data can be found in Table 15 in the Appendix. I'm using the experimentally determined growth factor because it's determined by using the measurements and not theoretically determined by considering the optimal value of the datasheet.

$$eps = (0.15 \cdot e^{\text{ln}(g) \cdot \frac{d}{2}}) \cdot 0.6 + (8 \cdot \frac{1}{\text{density} + 9} + 0.1) \cdot 0.7 \tag{3.3}$$

The growth function is multiplied by 0.15 to guarantee at a very small distance an eps value of at least 0.15. Furthermore, it regularizes the e function in such a way that the function grows very slowly in the first 50 meters. Furthermore, the growth value is multiplied by the half distance such that the e function increases with the distance to the LiDAR sensor. The distance is halved to decrease the impact of the distance. Figure 28 illustrates the eps estimation based only on the first term.

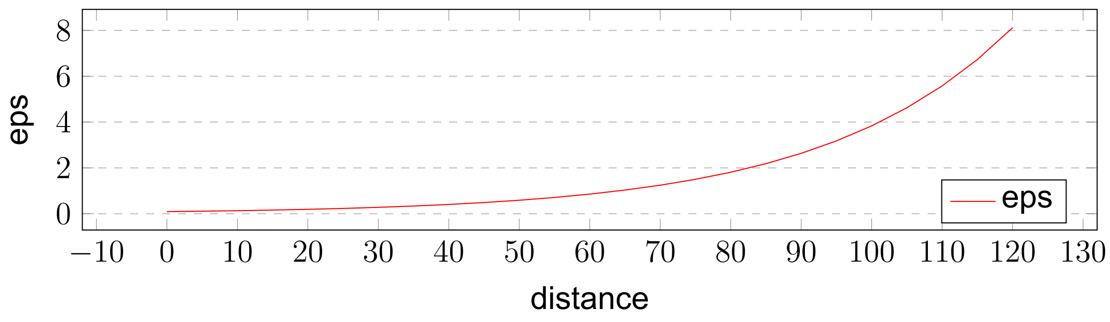


Figure 28: Visualized eps estimation.

The density term is multiplied by eight to guarantee a larger eps if the density is very small around one point. Furthermore, the 0.1 is added to the density term to guarantee an addable value to the eps if the density is very high. If the density is high, the term goes against zero. Otherwise, the term goes against positive infinity.

Additionally, the two terms are weighted differently. The first term is multiplied by 0.6 and the second one by 0.7. Both values are determined by experiments on different example data. The weighting makes the impact of the two terms differently on the final eps estimation.

minPts is estimated by multiplying a start value by eps and the distance divided by the density, following the observation of [3]. The start value is selected based on an eps of 1, corresponding to a search radius of 1 meter. The formula is expressed in Equation 3.4.

$$minPts = 75 \cdot eps \cdot \frac{d}{\text{density}} \tag{3.4}$$

After performing the *DBSCAN* algorithm, several clusters are obtained. These clusters are analyzed regarding their density in combination with their convex hull and the number of points building the cluster. If the cluster contains less than 15 points, it is considered to be a false cluster and is removed from the list. The underlying structure of such a cluster could be a bump in the road, a passing bird, or similar. Furthermore, if the cluster has a small density, calculated by the area covered by the convex hull divided by the number of cluster points, it is also considered a false cluster and removed. For instance, not filtered ground points could form such a cluster. This cluster is useless in the automated data labeling pipeline and is removed from the cluster object list. The thresholds that a cluster has to exceed to be considered a false cluster are experimentally determined.

3.4.3 Object Tracking

The clustering produces several object proposals. The object proposals are identifiable by the corresponding cluster ID, provided by the *DBSCAN* algorithm. However, the same object across different scans can have different IDs. The ID of an object depends mostly on the time when the first point of the cluster is proceeded by the *DBSCAN* algorithm.

To track the same object across consecutive frames, an object ID is assigned. This object ID is propagated through successive LiDAR scans to track an object.

The tracking algorithm consists of a tracking logic and the *Hungarian Algorithm* [23] to match the object IDs across consecutive LiDAR scans.

Hungarian Algorithm The *Hungarian Algorithm* is a matching method to find the best possible assignment across different objects. The algorithm works as follows:
Assume a matrix is given where the rows are toys and the columns correspond to children. Each entry indicates the popularity value of each child regarding each

toy.
First, the row minimum of each row is determined and subtracted from each element in each row. The column's minimum of each column is determined and subtracted from each element in each column as well. Next, an algorithm determines the number of lines needed to cover all zeros inside the matrix. If the number of lines needed is smaller than the number of the smaller dimension of the matrix, further zeros should be created as follows. Find the smallest uncovered number and subtract it from all uncovered numbers. Furthermore, add the smallest uncovered number to all elements that are covered by two lines.
If the number of lines needed to cover all zeros is equal to the smaller dimension of the matrix, the optimal assignment is found. In the assumed scenario, the optimal assignment corresponds to which children which toy gets.
The *Hungarian Algorithm*'s pseudocode is displayed in Figure 2 and an example of its application is given in Appendix Table 16.

Algorithm 2 Pseudocode of the *Hungarian Algorithm* [23].

```
1: Input List of objects of the Image Processing stage $O_{image}$, n size of $O_{image}$,
   List of object proposals of the LiDAR Processing stage $O_{lidar}$, m size of $O_{lidar}$
2: Output Mapping of the object proposals of LiDAR with the vision objects
3: Create matrix of size $n \times m$
4: Insert distance between each object and object proposal according to their
   matrix position
5: for $i <= n$
6:     Find row minimum $rmin_i$
7:     Subtract $rmin_i$ from each entry in row $i$
8: end for
9: for $j <= m$
10:     Find column minimum $cmin_j$
11:     Subtract $cmin_j$ from each entry in column $j$
12: end for
13: Calculate $min_{lines}$: number of minimum lines needed to cover all zeros inside
    the matrix
14: while $min_{lines} < n$ do
15:     Determine smallest entry $s$ not covered by any line
16:     Subtract $s$ from all uncovered entries
17:     Add $s$ to all entries that are covered by two lines
18: end while
19: Determine the best mapping based on the afore-created matrix
20: Return mapping
```

Implementation Details I've implemented the *Hungarian Algorithm* based on two different data representations. Once with *NumPy* array and once with *pandas* DataFrame. The *NumPy* and *pandas* packages are both implemented in C, a hardware nearer programming language than *Python*. This offers a performance

boost regarding their processing speed compared to *Python*'s standard lists. The implementation details are the same for both data structures and the final implementation of both approaches is available in the code base.
Following an analysis of their processing speed, I've chosen the *NumPy* implementation of the *Hungarian Algorithm* because it is faster by a factor of ten compared to the *pandas*' implementation. The corresponding analysis is carried out in chapter *Evaluation*, section 4.2.
The implementation can be split into several steps. Step 1 consists of finding and subtracting the internal minimum of every row and then of every column.
Step 2 targets the minimum number of rows needed to cover all zeros inside the matrix. To realize this, the matrix is converted into a boolean matrix, where each zero is mapped to True and False otherwise. Next, the row with the minimum number of True values is determined (Step 2-1). In Step 2-1, the entries of the row and column, in which the minimum number of True values was determined, are all set to False and the found zeros are stored in the marked_zero list.
Next, the matrix from steps 2-1 is checked and we mark the matrix according to certain rules. First, we mark rows that do not contain marked zero elements and store the corresponding row indexes in the non_marked_rows list. Next, we search the non-marked rows' element to find unmarked zero elements in the corresponding column. The column indexes are stored if an unmarked zero is found and the column index is not stored elsewhere. Step 2-2-4 compares the column indexes of the marked_zero list and the before-gathered column indexes. If a matching column index exists, the corresponding row index is saved to the non_marked_rows list. At the end, we determine the indexes that are not in non_marked_rows, store them in a marked_rows list, and return the result of the matrix marking process.
Checking if an optimal assignment is achieved is performed in Step 3. The smaller dimension of the input matrix is checked against the number of lines needed to cover all zeros. If the number of lines needed is smaller than the smaller dimension, Step 4 is performed. Otherwise, the optimal assignment is found and the corresponding matrix is returned.
Step 4 is called the adjustment step. The matrix gathered from Step 2 is adjusted to create additional zeros. The function can be separated into three steps. Step 4-1 finds the minimum element that is covered by no line. In Step 4-2, the minimum value is subtracted from each uncovered entry, whereas in Step 4-3, the minimum value is added to each entry which is covered by a row and a column line. The output of the whole of Step 4 is again checked by Step 2.
An example of the algorithm is given in Appendix Table 16 with a 4x4 matrix for the assignment problem.

For illustration purposes, the number of different LiDAR object IDs produced by the object tracking is tracked. At the end of the LiDAR processing, a color is assigned to each object ID such that the coloring of one object is consistent across several scans if they are visualized.

3.5 Sensor Fusion

The sensor fusion is performed to match the objects found by the vision object detection with the objects found in the LiDAR data. The fusion consists of several steps, illustrated in Figure 29.

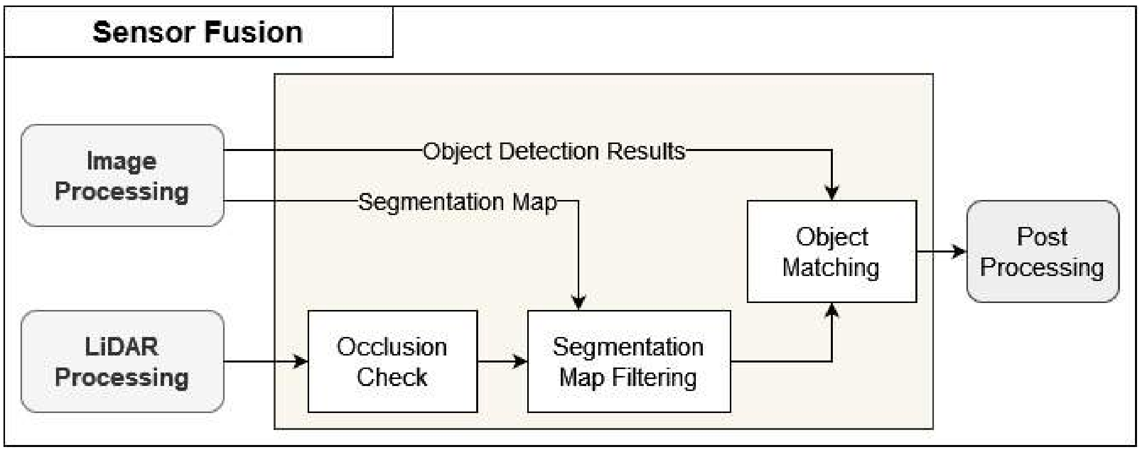


Figure 29: Overview of the *Sensor Fusion Stage*.

The field of view of the cameras can be split into four directions: front, left, right, and rear. Based on this field of view discrimination, the LiDAR 3D space is split into four fields. The camera front view can be described in a LiDAR sense, as all 3D points with $x > 0.98$. Similarly, the rear view can be described as all 3D points with $x < -3.739$. The Left is describable as all 3D points with $y > 1.04453$ and the right view as all 3D points with $y < -1.04453$.

First, depending on the selected camera's point of view, the corresponding LiDAR data are projected onto the corresponding 2D plane. Furthermore, only points that lie inside the image plane are considered for the next steps.

Second, an occlusion check for all remaining, projectable clusters is performed to exclude such clusters, that can not be mapped to a corresponding object due to occlusion by another object in the 2D plane.

Last, a matching algorithm is used to match the remaining LiDAR objects to the corresponding objects found in the images. Furthermore, if an occluded object cannot be classified in several frames, but is classified in one image, it is classified in the prior scans too. This is done at the end as some kind of post-processing stage.

3.5.1 Occlusion Check

The LiDAR sensor is mounted on top of the vehicle. Consequently, the LiDAR sensor can detect objects behind other ones where the second object can not be seen in the camera images. Especially, this is obvious if the LiDAR point cloud data are projected onto a 2D plane. Clusters, which lay behind each other in 3D, may lay directly on top of each other in the 2D space.

In the object matching stage, clusters that lay on top of each other may be wrongly matched to an object. For instance, a child behind a car may be detected by the LiDAR sensor but is not visible in the camera image. The child's and car's clusters lay on top of each other when the point cloud is projected onto a 2D plane. The matching algorithm uses the distance of the cluster's centroid and the bounding boxes' centroid to match the objects. In the worst case, the matching stage could classify the child as a car because its cluster centroid is closer to the bounding box centroid of the image object.

To mitigate this problem, an occlusion check is carried out. This check computes the occlusion of different objects in the 2D plane. If an object is occluded in the 2D plane, it is not visible in the corresponding camera image.

Implementation Details The inputs of the occlusion check are the clusters determined by the LiDAR clustering step, the projection matrix of the corresponding camera's point of view, and the image's size.
Each LiDAR point cloud data is projected onto the 2D plane, defined by the provided camera's projection matrix. Only projections that lay inside the image plane, defined by the width and height of the corresponding image, are considered for the next steps.
For each cluster inside the projected points, an object consisting of an ID, the centroid, the distance to the LiDAR sensor, and the vertices of the corresponding convex hull of the object, is created. The centroid is calculated by the projected pixel values of the processed cluster. Additionally, the occlusion state is added beforehand and initialized with -1 (neither occluded nor not occluded). The objects are stored in a list to use in the next steps.
Next, the gathered object list is sorted ascending by the distance to the LiDAR sensor. This is done based on the assumption that a near cluster c_1 is likely to occlude a cluster c_2 that is further away if c_2's centroid lays inside the convex hull of c_1.
The final occlusion check works as follows: Initialize an occlusion state list to track already checked clusters. Take a cluster c_1 of the sorted object list. If the picked cluster has already an entry in the occlusion state list, continue with the next one. Otherwise, check for all remaining clusters, whether their cluster's centroid lays inside the convex hull of c_1. If the centroid of cluster c_2 does not lay inside the convex hull of c_1, c_2 is considered as not occluded by c_1. Otherwise, the intersection area of the convex hulls of c_1 and c_2 are calculated. If the intersection area of both convex hulls exceeds a given threshold th_{inter}, the cluster c_2 is considered as occluded by c_1, and the occluded state of c_2 is set to 1.
At the end, all remaining clusters, that are not saved to the occlusion state list are marked as not occluded and their occluded state is appropriate.
The output consists of all clusters, where the occluded state is set to zero. All

other clusters are not visible in the camera image and would lead to false results.

3.5.2 Segmentation Map Filtering

In *Segmentation Map Filtering*, the corresponding segmentation map to the underlying camera image is used to gather only the most relevant clusters from the non-occluded clusters. For instance, clusters lying on the categories *Road*, *Sidewalk*, *Building*, *Wall*, *Fence*, *Vegetation*, *Terrain*, and *Sky*, are of no interest at the moment. These clusters are filtered and only clusters that lay on the remaining categories are used for the next steps.

Implementation Details First, the relevant categories of the segmentation map are extracted. The result of this extraction is a list of indices, representing certain categories of the segmentation map.
After extracting the relevant pixels of the segmentation map, the not-occluded 2D objects are projected onto the 2D plane of the segmentation map. Each point that does not lay on a relevant category is ignored for further processing steps. Furthermore, if the centroid of a LiDAR object does not lay on the extracted pixel coordinates, the cluster is also ignored for further processing.

3.5.3 Object Matching

The outputs of the image processing stage and the LiDAR processing stage differ in their number of objects and information, as well as their classification quality. Image processing produces several objects with corresponding, beforehand defined, labels and bounding boxes, while LiDAR processing produces clusters corresponding to possible objects without labels. These objects need to be matched to create a sensor fusion dataset.

The matching of the object proposals by the LiDAR processing with the object produced by the image processing is realized by the use of the *Hungarian Algorithm*. The previous occlusion check assigns the attribute occluded to each LiDAR cluster and returns only LiDAR objects, that are not occluded in the camera image. The *Hungarian Algorithm* finds the best possible assignment of the object proposals and the image objects.

Unfortunately, the occlusion check leads to a kind of security problem because occluded clusters can not be matched to camera objects on the fly. For instance, a child behind a car can be scanned by a top-mounted LiDAR sensor but may be occluded from a camera's point of view. The child may run onto the street and the vehicle may recognize the child too late.

To mitigate this issue, the LiDAR processing stage performs object tracking across consecutive frames. This object tracking is relevant here because it gives the possibility to mitigate the aforementioned security problem of occluded objects.

If an occluded object is once detected in a camera image due to the object tracking, it can be classified across all scans in which the object appeared. This gives the possibility to classify objects in hindsight at a later time and increases safety because algorithms learning on LiDAR data can recognize and classify an object, although it is occluded in the corresponding camera image.

Hungarian Algorithm The implementation of the *Hungarian Algorithm* is almost the same as in the LiDAR processing. It differs in the calculation of the assignment problem. In the LiDAR processing stage, the assignment problem lies inside the 3D space because the objects in consecutive LiDAR scans should be tracked. In the Sensor Fusion, the assignment problem is a 2D-related problem because the centroids of the not-occluded LiDAR object proposals are projected onto the 2D plane of the corresponding camera image. These centroids should be assigned to the bounding boxes of the image objects.
The assignment is done by calculating the cluster centroids' position regarding the 2D projection. At the same time, the centroids of the bounding boxes are calculated too. The distances between each cluster centroid and each bounding box centroid are calculated and stored as a matrix. This matrix functions as input to the *Hungarian Algorithm* to find the best possible assignment.

Implementation Details The input to the final object matching stage are the LiDAR object proposals that are not occluded and the objects found in the corresponding camera image. As mentioned before, the *Hungarian Algorithm* is performed on the object proposals and the visual objects. The result is a matrix, where the best possible assignment of a visual object (row) to a LiDAR object proposal (column) is marked with the distance between the corresponding centroids.
After performing the algorithm, a matching list is retrieved.
A dictionary is implemented to save the matching results for later use. The dictionary keys are the IDs of the LiDAR objects, while the corresponding values are the labels of the visual objects.

3.5.4 Post Processing

The output of the Sensor Fusion or the object matching step is an object list, with objects found in the LiDAR data and a corresponding label, based on the results of the *Hungarian Algorithm*. This mapping of LiDAR objects to labels is used to mitigate the possible safety problem created by occluded objects.

The aforementioned mapping contains all detected and classified objects by the sensor fusion. To label objects that are occluded in one but are classified in another LiDAR scan, all LiDAR data, included by the *ROS* bag, are post-processed.

In the post-processing, each object in the different scans is given the corresponding label, regardless of whether the object is occluded in a certain scan or not. This is done by iterating over all LiDAR scans available and checking if the different objects of the scan have a label assigned. If not, the mapping from the sensor fusion is checked if such an object ID to label mapping exists. If yes, the label is assigned to the corresponding object. Otherwise, the object is classified as not classifiable.

3.6 User Interface

The automated data labeling pipeline can work without a graphical user interface. However, due to the data it produces and the need to be precise and allow to mitigate wrong classifications, a user interface is needed.

A user interface allows the easy visualization of all results of the data labeling pipeline and a human labeler can go through a labeled *ROS* bag file and take a look at the results it produced.

Following, I describe the requirements of the user interface that should be implemented at the end. Furthermore, I give insights into the implementation details of the user interface and which packages I used. The available functionalities are described as well.

Requirements I've defined several requirements that the user interface should fulfill. Some regarding the training and execution of the data labeling pipeline, and some regarding the visualization of the pipeline results.
Further requirements are defined based on the possible need for a human labeler. For instance, functionalities to correct misclassifications.
Additional requirements to work with *ROS* bag files and different *Velodyne* sensors are defined.
Following, a list of all defined requirements is presented.

R-1 The user interface allows the selection of a recording saved as a *ROS* bag file from the host system.

R-2 The user interface mitigates the change of the *Velodyne* sensor and allows a selection between the older and the current model.

R-3 The function to extract all relevant data from the *ROS* bag file is available as a separate button.

R-4 The segmentation stage can be started on its own.

R-5 The object detection stage can be started on its own.

R-6 The LiDAR processing stage can be started on its own.

R-7 The Sensor Fusion stage can be started on its own, but only if the segmentation, object detection, and LiDAR processing results are available.

R-8 The automated data labeling pipeline can be started by a single button push.

R-9 The output of each pipeline stage can be visualized.

R-10 A human labeler can go through the generated and processed data and can take a look at the results of the different pipeline stages.

R-11 A human labeler can mitigate false object classification in the image processing stage.

R-12 A human labeler can approve the results of the automated data labeling pipeline.

R-13 If the pipeline results are approved, a sensor fusion dataset consisting of images and LiDAR scans are generated.

R-14 A training interface is available, where the different object detection models can be trained on further data.

Implementation Details I've used the *Python* package *tkinter*[4] for the implementation of the user interface. Many useful tools and widgets are available. For instance, the notebook widget enables a simple visualization tool for the different pipeline outputs.
The UI functions as a visualizer and accesses the different pipeline stages through predefined functions only.
From the home UI, one can access the execution or training UI.

3.7 Overview

Figure 30 shows the structure of the different pipeline stages and the connection between them.

Each pipeline stage can be executed on its own if the needed data are given. Furthermore, the different pipeline stages can be executed simultaneously.

After the *Data Extractor*, the *Image Processing* and *LiDAR Processing* stages are executed. The sub-stages of *Image Processing* can be executed at the same time because they do not rely on each other. Contrary, the LiDAR stage has to be executed consecutively because the *Object Tracking* can only be performed if the LiDAR point cloud data is clustered.

The output of both pipeline stages is fused in the Sensor Fusion stage, from which the final sensor fusion dataset is generated.

[4] tkinter — Python interface to Tcl/Tk: https://docs.python.org/3/library/tkinter.html (Online, Accessed February 21, 2024)

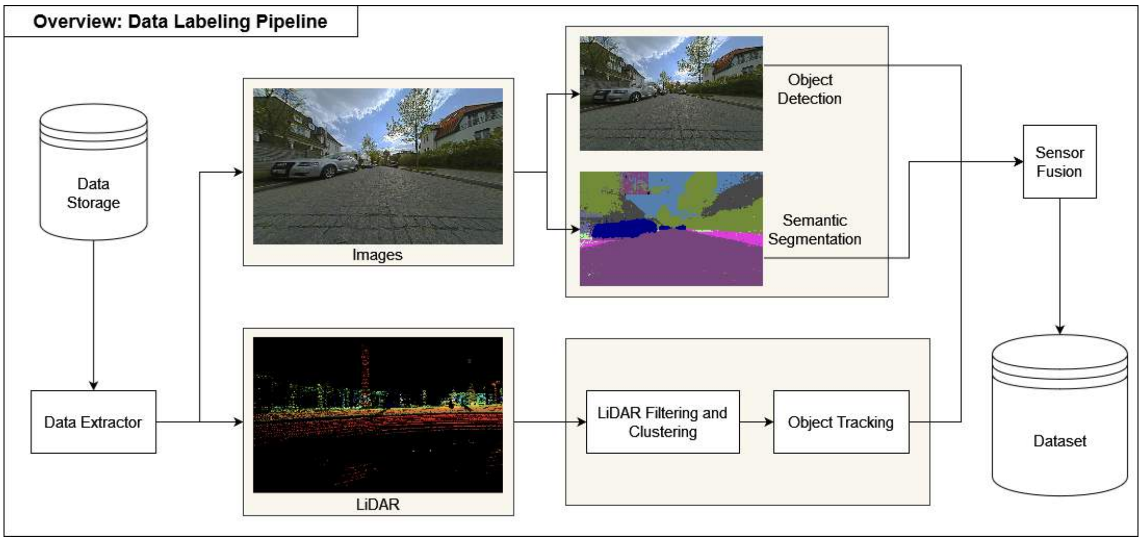


Figure 30: Overview of the automated data labeling pipeline and its stages. The stages are completely separate from each other and can be executed simultaneously.

The requirements are implemented in the user interface, illustrated in the Figures 46, 47, 48, 49, 50, and 51 in the Appendix.

Table 2 shows all requirements of the user interface and their corresponding status.

Req.-ID	Short Description	Fulfillment	Status
R-1	The user can select a *ROS* bag file from the system	Yes, implemented in the default execution user interface	Done
R-2	Several *Velodyne* models are supported	Yes, implemented in the control functions of the execution user interface. VLS-128 and HDL-64E are selectable	Done
R-3	The relevant data of a *ROS* bag file can be extracted	Yes, implemented in the execution user interface and can be executed by a button click	Done
R-4	The segmentation stage can be started	Yes, implemented in the function frame of the execution user interface and can be executed by a button click	Done
R-5	The object detection stage can be started	Yes, implemented in the function frame of the execution user interface and can be executed by a button click	Done
R-6	The LiDAR processing stage can be started	Yes, implemented in the function frame of the execution user interface and can be executed by a button click	Done
R-7	The Sensor Fusion can be started, only if segmentation, object detection, and LiDAR processing results are available	Yes and No. The sensor fusion can be started by a button click but it does not check if the corresponding data are available	Unfinished
R-8	The automated data labeling pipeline can be started	Yes, implemented in the function frame of the execution user interface and can be executed by a button click	Done
R-9	The output of each pipeline stage can be visualized	Yes, implemented in the different notebooks on the right side of the execution interface.	Done
R-10	The user can go through all data and all results	Yes, implemented in the notebooks on the right side of the execution interface.	Done
R-11	The user can mitigate false object classifications	Yes, implemented in the *Image Processing* notebook in the subcategory *Object Detection Results*	Done
R-12	The results of the Pipeline can be approved	No	Unfinished
R-13	From the approved results, a sensor fusion dataset consisting of images and LiDAR scans is generated	No	Unfinished
R-14	A training interface is available to retrain or newly train the object detection models	Yes, implemented in an extra training user interface, accessible from the start user interface	Done

Table 2: Status of the requirements of the User Interface. A short description of the requirements and how and where they are fulfilled is given.

4 Evaluation

In this chapter, *Evaluation*, the evaluation regarding the accuracy, performance, and run-time of each pipeline stage is carried out. Besides testing each stage separately, and evaluating methods on different implementations, the whole pipeline as one is evaluated.

4.1 Image Processing

Following, each image processing stage, consisting of semantic segmentation and object detection, is evaluated individually. Different metrics are used and explained in advance.

4.1.1 Semantic Segmentation

The model used for the semantic segmentation is evaluated regarding the mean intersection over union (mIoU) on the validation set of the *CityScapes* dataset. Furthermore, different segmentation heads on top of the *DINOv2* feature extractor are tested to find the best-performing one.

The tested segmentation heads differ in the use of an activation function, either *ReLU* or *GeLU* and in the number of linear layers before the final conv layer. At the end of each model, an upsampling is performed to match the segmentation map's size with the input image.

The following configurations were tested: i) Based on the original *DINOv2* paper, a lonely convolutional layer is used on top of the feature extractor. ii) One linear layer is placed between the feature extractor and the convolutional layer, but no activation function is between them. iii) Based on the second configuration, a *GeLU* activation function is applied between the linear layer and the convolutional layer. iv) The feature extractor is followed by two linear layers and a final convolutional layer. Between each additional layer, a *GeLU* activation function is applied. v) Four linear layers are applied before the final convolutional classification layer. In between the layers, a *GeLU* activation function is applied. vi) The best-performing *GeLU* architecture is modified. Instead of using *GeLU* activation, *ReLU* is used.

The mIoU of the different segmentation head configurations can be found in Table 3. The bold row is the model used for the automated data labeling pipeline.

Regarding the segmentation maps, the interpolation achieves a finer segmentation map compared to no interpolation. Furthermore, most traffic signs and the

Configuration	mIoU
i)	88.12
ii)	88.31
iii)	**88.82**
iv)	88.62
v)	88.44
vi)	88.80

Table 3: mIoU of the different segmentation head configurations, evaluated on the validation split of the *CityScapes* dataset. The mIoU is calculated pixel-wise. All models are trained on the same amount of epochs (100), the same learning rate of 0.003, and the same optimizer with momentum of 0.9.

person on the left-hand side are missing in the segmentation map without interpolation in Figure 21. Both are better illustrated in the segmentation maps presented in Figure 22 and 23, where interpolation was applied. The segmentation map with interpolation by a factor of two detects the person better but struggles with the traffic signs on the right-hand side of the image compared to the segmentation map with an interpolation factor of three.

Interestingly, the human in the segmentation map in Figure 23 is worse segmented than in Figure 22. This can happen if the image splits split a small object into several pieces. To mitigate this, one could combine the segmentation maps of different interpolation factors. Another idea is the analysis of the segmented patch around the patch in question. A remapping could be carried out to re-segment certain patches. Furthermore, the confidence of the model would be interesting for such patches. Maybe several categories have equal confidence as the selected category.

In the current state, this problem is not huge. If the relevant categories are extracted from the segmentation map, a buffer zone with a radius of five pixels is introduced to cope with these segmentation errors.

Another issue with the creation of the segmentation map is that the image has to be rescaled to an aspect ratio of two-to-one due to the training routine. The random crop used is a square, benefiting images of an aspect ratio where squares perfectly fit. The model could perform better on the data of the *Dahlem Center for Machine Learning and Robotics* if a model is trained with a crop of the same aspect ratio as the *ROS* bag images.

4.1.2 Object Detection

Different specialized models are trained to detect objects within images. The models are split into traffic signs, vehicles, pedestrians, and traffic lights. The final result consists of the concatenation of all prior results of the aforementioned models.

The main advantage of using *Ultralytics*' *YOLO* models is that they come with an automated evaluation sheet. The provided evaluation is split into training, validation, and general evaluation metrics. The different metrics are explained in the subsection Metrics.

During the training, the mAP50, mAP50-95, F1, precision, recall, box loss, class loss, and distribution focal loss (dfl) are tracked for each epoch.

Following, I show and explain the results of the object detection models for pedestrians, vehicles, traffic signs, and traffic lights. I describe the used validation datasets briefly and give insights into the dataset composition.

4.1.3 Datasets

To train the different object detection models, I've used several publicly available and own-created datasets.

Following, the used datasets are presented in great detail. The modifications I've done are explained and justified.

Traffic Light Dataset

The traffic light dataset I've used is publicly accessible at *Kaggle*[1]. The dataset consists of 3.000 images, where I created a training split of 2.400 and a validation split of 600 images.

The following nine traffic light categories are included: Motor vehicle signal light, Non-motor vehicle signal light, Left turn non-motor vehicle signal light, Crosswalk signal light, Lane light, Direction indicator light, Flashing warning light, Crossing signal light, and U-turn signal light. Unfortunately, neither of these categories is annotated. The only annotations are for the traffic light phase, including *green*, *yellow*, and *red*.

class	green	yellow	red
#annotations val split	720	39	939
#annotations train split	2809	125	3894
#annotations dataset	3529	164	4833

Table 4: Number of annotations per traffic light phase in the validation and training split, as well as the overall sum in the whole dataset.

[1] Kaggle: https://www.kaggle.com/ (Online; Accessed: February 21, 2024)

Table 4 shows the number of annotations per class. Yellow has the fewest annotations. This is obvious because compared to the traffic phases red and green, yellow appears only a fraction of the time. In Germany for instance, yellow appears for three seconds while a speed restriction of 50km/h[2].

The dataset is okay for learning the traffic light phases but struggles with the different pictograms in German traffic lights.

nuImages

The *nuImages* dataset consists of 93.000 images with 2D bounding boxes for 800.000 foreground objects. The objects are categorized into 24 distinct categories. The main categories are pedestrians, movable objects, and vehicles. The main categories are further defined into several subcategories. For instance, the pedestrian category is split into adult, child, construction worker, personal mobility, police officer, stroller, and wheelchair. The same applies to the other main categories.

The most relevant classes to this thesis are presented in Table 5. Furthermore, the number of annotations in the training and validation split is shown.

class	number of instances (training)	number of instances (validation)
human.pedestrian.adult	121200	28721
human.pedestrian.child	1683	251
human.pedestrian.construction_worker	10465	3117
human.pedestrian.personal_mobility	1828	453
human.pedestrian.police_officer	368	96
human.pedestrian.stroller	293	70
human.pedestrian.wheelchair	33	2
vehicle.bicycle	13708	3352
vehicle.bus.bendy	203	62
vehicle.bus.rigid	6538	1823
vehicle.car	202809	47279
vehicle.construction	4768	1301
vehicle.emergency.ambulance	34	8
vehicle.emergency.police	104	35
vehicle.motorcycle	13682	3097
vehicle.trailer	3286	486
vehicle.truck	29456	6857

Table 5: Number of annotations for each relevant class of the *nuImages* dataset for the training (center column) and the validation (right column) split.

I'm using the full 93.000 images in *nuImages* v1.0. The *nuImages* include the files train, val, test, and mini splits. The images are loaded from the train and val split. Subsequently, the training split consists of 60.668 and the validation split of

[2] ADAC: Ampelphasen: Diese Ampelfarben gibt es, URL: https://www.adac.de/verkehr/recht/verkehrsvorschriften-deutschland/ampel/ (Online, Accessed: February 21, 2024)

14.884 images, all extracted from the val and train split of the *nuImages* data.

The dataset is highly unbalanced. This is obvious because of the class vehicle.car appears approximately 250.000 times compared to the class vehicle.emergency.ambulance appearing only 42 times.

All in all, the dataset is good because of its pure size and the number of annotations.

German Traffic Sign Dataset (GTSD)

The used traffic sign dataset is based on the *GTSDB* of the *Ruhr-University Bochum*. The dataset is heavily modified with more distinct object classes. Furthermore, more images are added.

Instead of 42, the used dataset has 93 different traffic sign classes. The labels are taken from the traffic sign catalog from last year (2023). The annotated traffic signs are illustrated with their official IDs in Appendix Figure 45.

The newly created dataset has a strong unbalance of the number of appearances of different traffic signs. Figure 31 shows the unbalance and the number of annotations per class for the training and validation dataset. For instance, the traffic sign 1002-23 appears only once, while the traffic sign 205 appears 161 times in the training split. The same applies to the validation split. The traffic sign 1002-23 appears also only once, while the traffic sign 274-70 appears 70 times.

The dataset is neither large enough nor has all traffic signs in it. Germany has over 200 distinct traffic signs. For instance, for speed restrictions exist 14 different traffic signs plus 14 for the cancellation of a speed restriction. All in all, the used dataset is way better than the original *GTSDB*, because of its composition and the definition of the traffic sign classes, which were poor in the *GTSDB*. The authors of the original dataset used the same label definition for two distinct traffic signs. For instance, a speed of 30 km/h-zone was defined as a 30km/h speed restriction sign, which is wrong.

To overcome the weaknesses of the *GTSDB* and extend the vocabulary, I've added several hundreds of images. These images are taken by a camera from different perspectives but with a focus on a car's point of view. The newly created images are cropped to a three-to-two format. Furthermore, the images are added once in their original resolution and once added as near as at the same resolution as the original *GTSDB* images, while keeping the aspect ratio to three-to-two.

The final dataset consists of 1126 images in total. 916 images are preserved for the training split and 210 for the validation split. A test split could not be gathered due to the time-consuming task of manually labeling each image on its own.

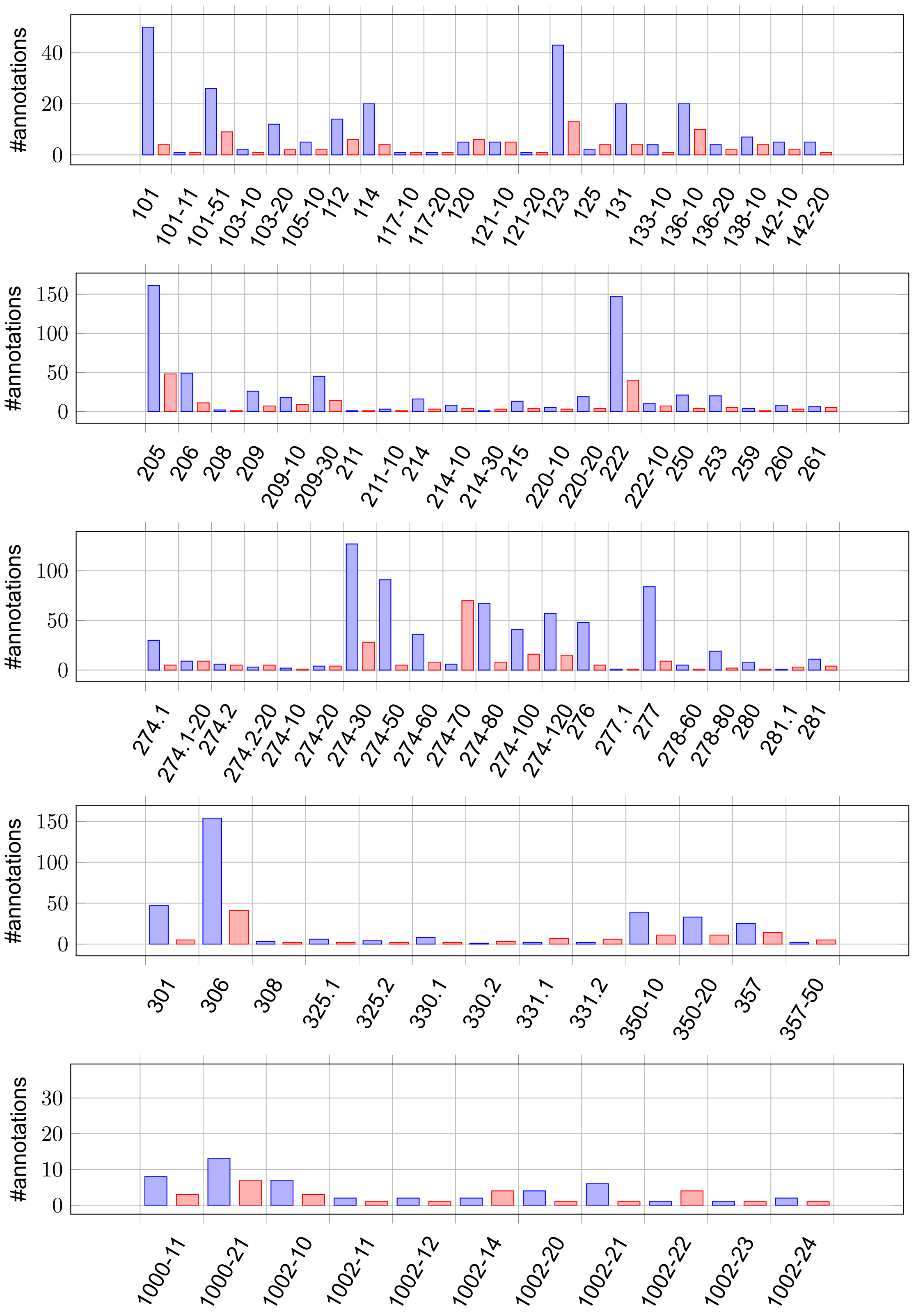


Figure 31: Number of annotations for each traffic sign across the training dataset (blue bars) and the validation dataset (red bars). The y-axis represents the number of annotations for each traffic sign. The x-axis shows the IDs of the corresponding traffic signs determined by the traffic sign catalog.

4.1.4 Metrics

Ultralytics' *YOLO* model computes different validation metrics on its own. Furthermore, an automated evaluation sheet is provided, illustrating some metrics, such as box, class, and distribution focal loss for training, as well as for validation. Furthermore, the *precision*, *recall*, *mAP50*, and *mAP50-95* are illustrated too.

The box loss shows how well a model predicts the bounding boxes of the objects. If the loss is low, the model predicts the boxes of the validation and training set are good. Otherwise, the predicted boxes are bad.

The class loss illustrates how good the predictions are made for the objects inside the bounding boxes. If the model finds all objects and classifies them correctly, the loss is low. Otherwise, the loss is high.

Distribution focal loss (dfl) is a loss function and is used to improve the model's performance. *YOLOv8* has the dfl implemented and directly optimizes distribution of bounding box boundaries. DFL is used in bounding box regression, and since the detection task of *YOLO* is formulated as a regression problem, it is used. The general idea is to predict distributions of the box offsets instead of predicting the values directly. It has nothing to do with the object class classification.

mAP50 and *mAP50-95* are both metrics that measure the mean average precision across a certain *Intersection over Union* (IoU) threshold. The first one takes all results into account, where the predicted bounding box has an IoU with a ground truth of at least 0.5. The *mAP50-95* is calculated at varying IoU thresholds from 0.5 to 0.95, in steps of 0.05.

The F_1-Score is the mean of precision and recall and varies between one and zero. If the score is near one, it corresponds to good recall and good precision, while a score near zero corresponds to the opposite case.

4.1.5 Traffic Sign Model

The detection model is trained for 200 epochs on the aforementioned newly created traffic sign dataset, based on the publicly available *GTSDB* dataset of the *Ruhr-University Bochum*.

Results

I've experimented with the *YOLOv8* default settings of the parameter *imgsz* to find the best possible model. This parameter describes how the input images are reshaped. The default value is 640, resulting in the resizing of each input image to the shape 640x640 pixels.

The *YOLOv8* model achieves the best performance if trained on rectangular images. However, the model can predict objects in images of different shapes.

I also considered training and validation images of arbitrary size and tested whether the performance is better with images of arbitrary size or with a common size across the whole dataset. Arbitrary size refers to the case that the images of the dataset can have different sizes. Common size refers to the case that the images have all the same width size and nearly the same height. The common width is 1360 pixels, while the height varies between 800 pixels and 906 pixels.

Following, the tested models are presented in Table 6. Several other YOLOv8 models are publicly available. These are *YOLOv8n* (nano), *YOLOv8s* (small), *YOLOv8m* (medium), *YOLOv8l* (large), and *YOLOv8x* (extreme). As the base model, I've chosen a pre-trained *YOLOv8m*. This is based on a compromise between training time and memory consumption. The larger the model, the more time it needs to be trained and perform object detection.

I've chosen 1280 as imgsz because the images of the *ROS* bag driving recordings have the shape 1280x800 pixels if rectified.

ID	*YOLOv8* model	pre-trained	image size	imgsz
M1	yolov8m.pt	yes	arbitrary	640
M2	yolov8m.pt	yes	arbitrary	1280
M3	yolov8m.pt	yes	common	832
M4	yolov8m.pt	yes	common	1280

Table 6: *YOLOv8* models that are tested. The image size refers to whether the validation and training images have arbitrary or common sizes. The imgsz is the *YOLOv8* parameter that determines the reshaping of the input image while training and validation.

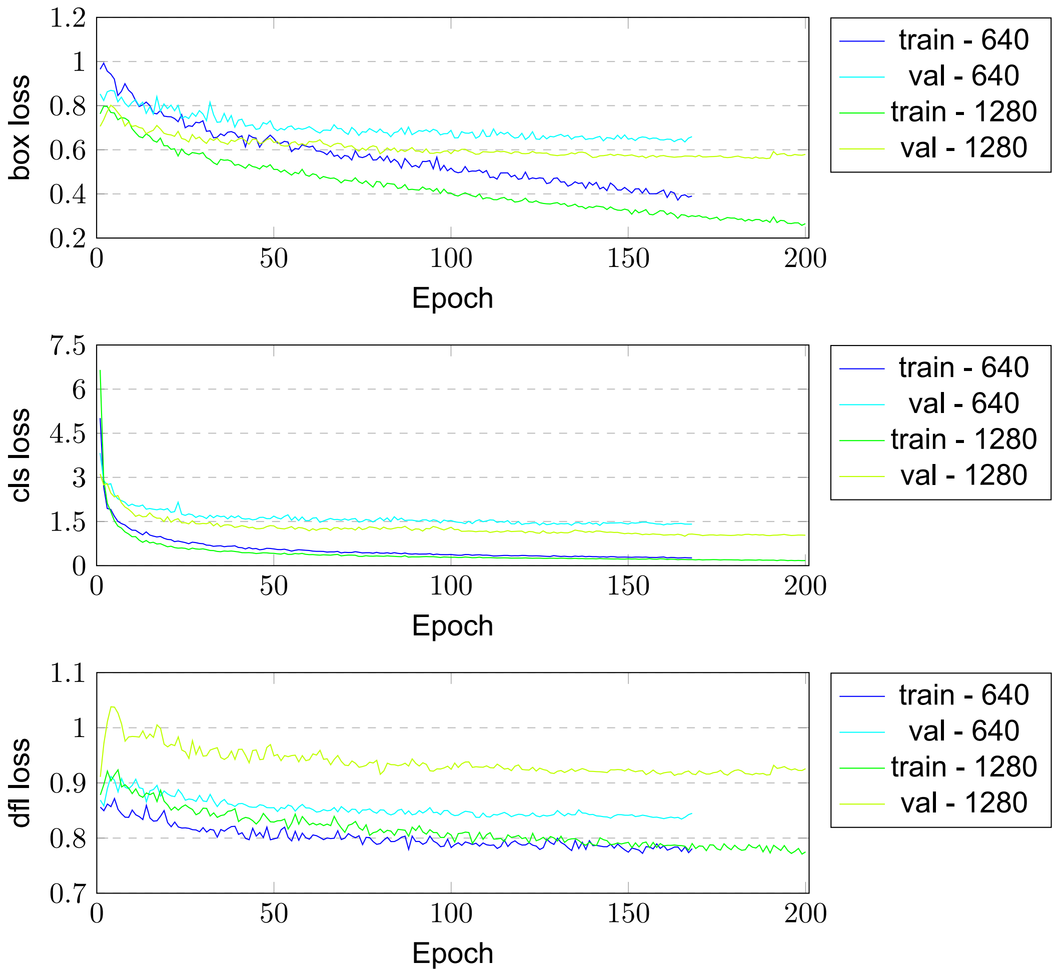


Figure 32: Box, cls, and dfl loss for different trained traffic sign models with different *YOLOv8* parameters and arbitrary training and validation image size. The greenish lines correspond to the train and validation loss of the *YOLOv8* model by setting the imgsz-parameter to 640. The blueish lines correspond to the model by setting the imgsz-parameter to 1280.

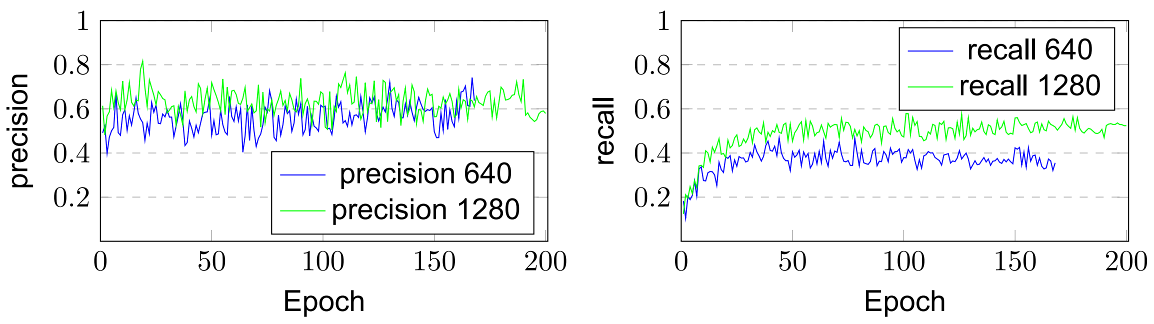


Figure 33: Recall and Precision of different trained traffic sign models with different *YOLOv8* parameters and arbitrary training and validation image size. The green line corresponds to the *YOLOv8* model by setting the imgsz-parameter to 640. The blue line corresponds to the model by setting the imgsz-parameter to 1280.

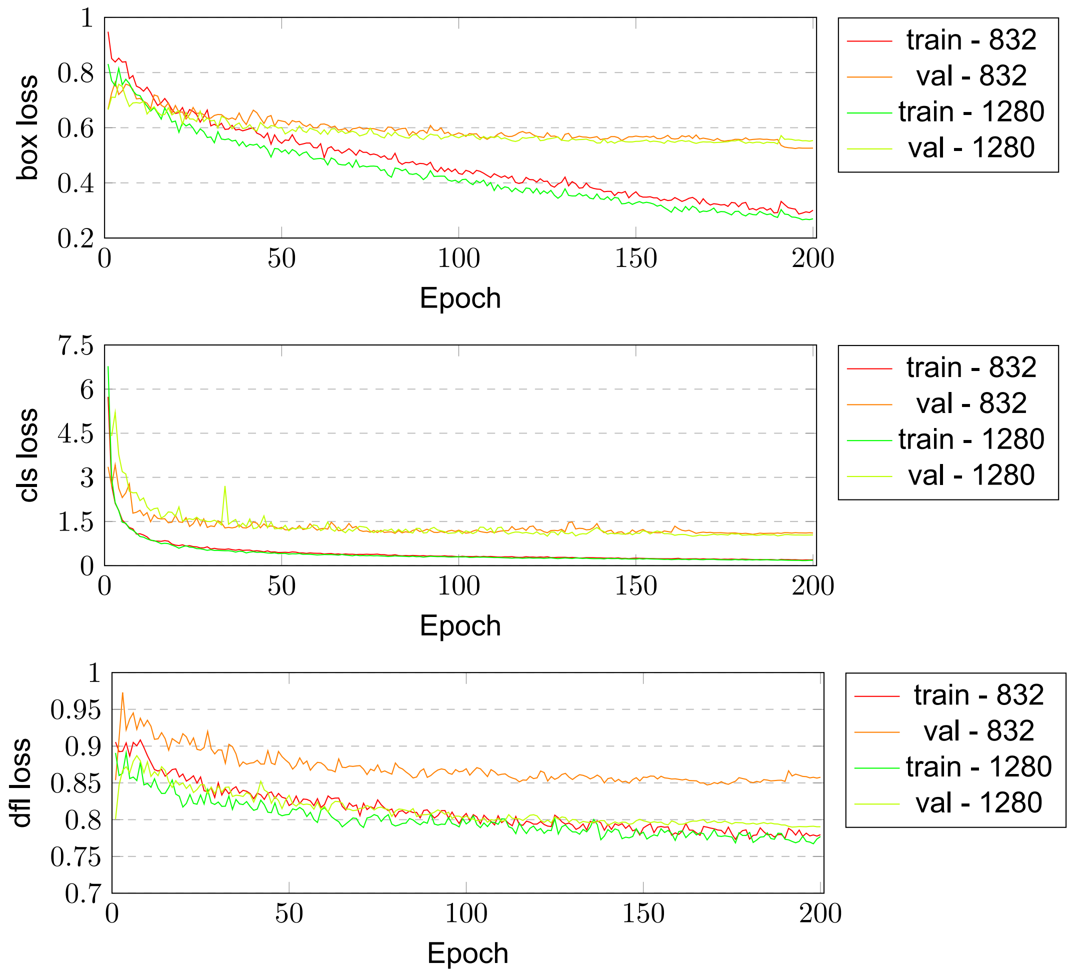


Figure 34: Box, cls, and dfl loss for different trained traffic sign models with different *YOLOv8* parameters and a common training and validation image size. The reddish lines correspond to the train and validation loss of the *YOLOv8* model by setting the imgsz-parameter to 832. The reddish lines correspond to the model by setting the imgsz-parameter to 1280.

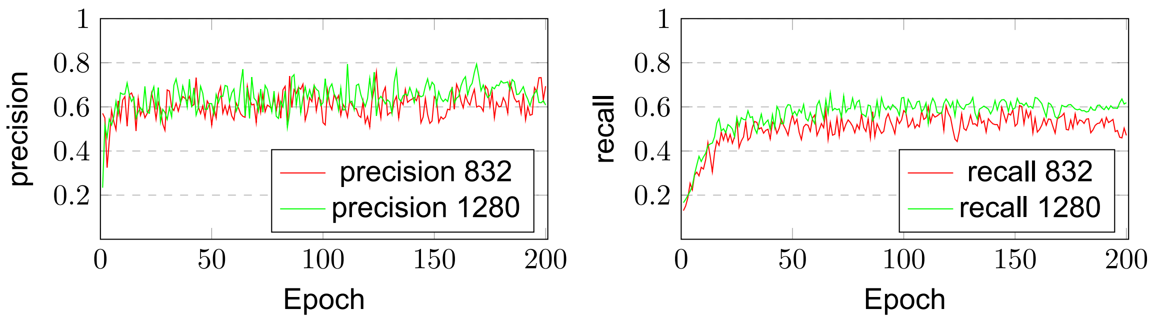


Figure 35: Recall and Precision of different trained traffic sign models with different *YOLOv8* parameters and a common training and validation image size. The red line corresponds to the *YOLOv8* model by setting the imgsz-parameter to 832. The blue line corresponds to the model by setting the imgsz-parameter to 1280.

One may notice that the blue and cyan lines in Figure 32 as well as the blue lines in Figure 33 do not extend to the 200th epoch. This happens if the model does not achieve a better performance after 50 consecutive epochs. The threshold of 50 epochs is a default setting of *YOLOv8* and can be changed if needed.

In Figure 32, the box and cls loss plots indicate that setting the imgsz-parameter to 1280 is beneficial for a training and validation dataset consisting of images with arbitrary sizes. The dfl loss plot shows the opposite for the prediction performance of bounding boxes. The validation's dfl loss is lower using imgsz of 640 (cyan line) instead of 1280 (lime line). It is approximately 0.8 points lower (0.84486 compared to 0.92576 at the 200th epoch).

The precision and recall plots in Figure 33 indicate that a higher imgsz-parameter benefits training and validation images with arbitrary size. The precision is almost the same, while the recall differs a lot. This is mainly because the validation images are at least twice as big as the training images.

In the cls loss plot in Figure 34, the validation loss for the first epoch of the model with the larger imgsz-parameter was removed. This is because the cls loss of the first validation epoch is at 579.76 and would eliminate any visualization of the remaining progress. I do not indicate why this happened. I guess the model predicted every class wrong. The box loss of the first epoch substantiates this guess because it is comparable to the box loss of the smaller imgsz.

The cls and box loss do not show a significant difference while using different imgsz-parameters. The loss of the model with the larger imgsz is mainly a bit lower than the loss of the other model. Only the dfl loss is different by a visible margin. The validation loss of the model with the smaller imgsz (orange line) is approximately 0.5 points worse than the model with the larger imgsz (lime lines). This correlates to the model's ability to predict the bounding boxes better if the training images are larger. This is attributed to some bounding boxes being only a few pixels big. If the image size decreases, the bounding box might be only one pixel big in the end, leading to difficulties in the prediction.

Interestingly, Figure 35 does not show a difference between both imgsz settings regarding the precision. For both evaluated *YOLOv8* parameters, the precision is almost the same. The recall differs from this observation. The model with an imgsz setting of 1280 (red line) achieves a higher recall (0.61897) than the model with an imgsz setting of 832 (green line) (0.47088), indicating that fewer mistakes (false negatives) are made if the image size increases. This coincides with the assumption that small objects are better detectable if the image size is bigger.

Comparing the best model of Figure 32 and Figure 34 leads to the observation that a common image size for all training and validation images is beneficial to train an object detection model. Table 7 compares the models M2 and M4 directly. One can see that the cls and box loss of the models M2 and M4 are comparable for the training. One can say that the margin between both models is so insignificant that they perform equally. The same observation can be made in the validation columns of both models. They perform equally, but M2 struggles with the dfl loss.

The precision and recall tell a different story. The precision of both models is quite close together, but the recall differs a lot. M4 outperforms M2 regarding recall and precision. The main cause of this is that the images, especially of the validation dataset, do not have to be resized that much when using a common image size across validation and training datasets. After the resizing, I went through all the images and reassigned the bounding boxes if any of them did not match anymore. Another issue is that by using and annotating images with higher resolution, more objects can be found because they are clearer to see for a human. If an already small bounding box of a small object is resized, the bounding box may shrink to only one pixel, making it almost impossible for the model to detect the underlying object.

The aforementioned explanation is the reason for the selection of model M4 as a traffic sign detection model because it makes fewer mistakes (fewer false negatives).

Model	Image size	imgsz	training			validation			precision	recall
			cls loss	box loss	dfl loss	cls loss	box loss	dfl loss		
M2	arbitrary	1280	0.17295	0.26515	0.77499	1.0339	0.57914	0.92576	0.57957	0.52375
M4	common	1280	0.18011	0.26979	0.65548	1.0435	0.55356	0.85766	0.60849	0.61897

Table 7: Comparison of the best models for arbitrary and common training and validation image size.

4.1.6 Traffic Light Model

I've experimented with the traffic light model if training the model iterative, every time only for 50 epochs is beneficial compared to a model that is trained for 150 epochs directly.

All models are trained on the same training and validation split of the traffic light dataset, described in the section 4.1.3. They predict the traffic light phases and the position of the traffic light.

Results

I've experimented with the number of training epochs and fixed the imgsz-parameter of the *YOLOv8* model at 832 for all test runs.

Interestingly, the model trained straight up for 150 epochs achieves much better results compared to the iterative trained model. The iterative trained model seems to struggle with the restart of the training procedure at most.

For the evaluation, I concatenated the results from the iterative models. Hence, the results of the different metrics can be displayed in one plot and the difference to the ad-hoc model can be better seen.

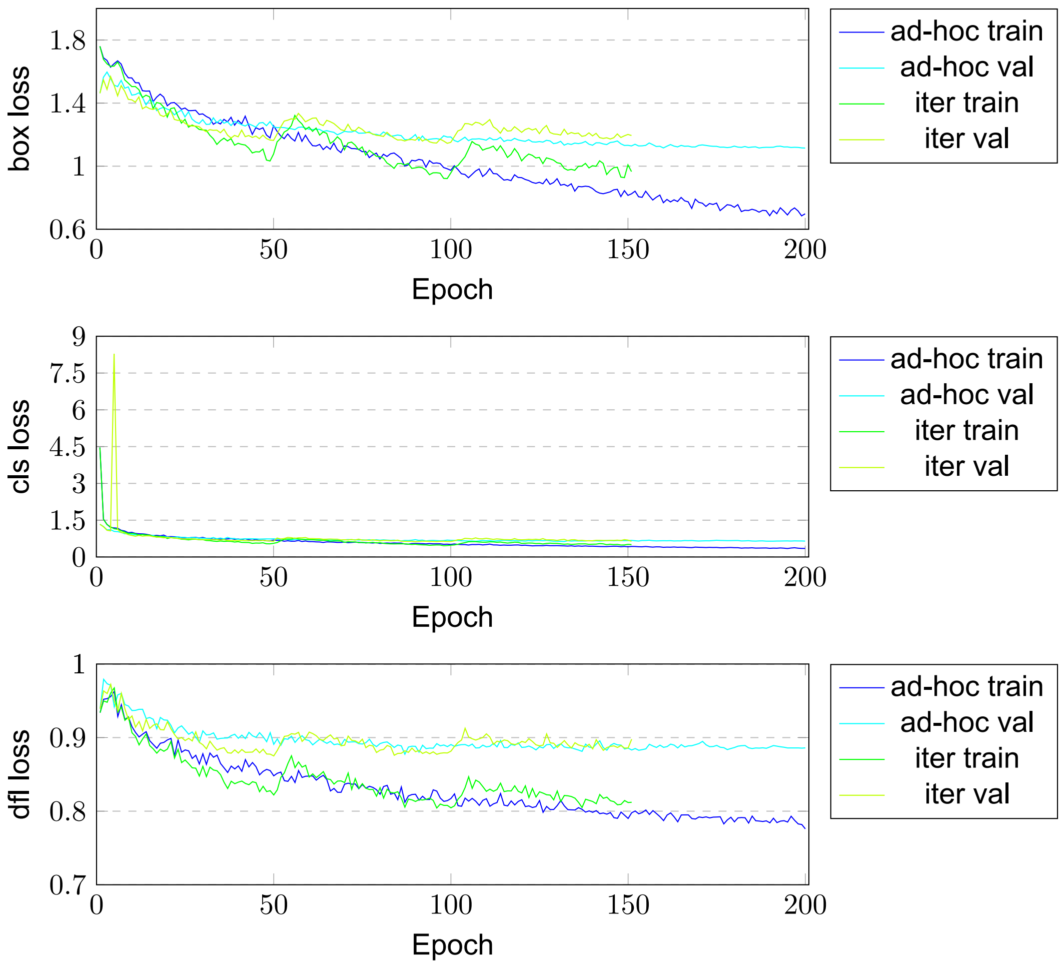


Figure 36: Box, cls, and dfl loss for the ad-hoc traffic light model (blueish lines) and the iterative trained model (greenish lines). All models are trained with a fixed imgsz-parameter of 832.

All plots in Figure 36 show the trend that the ad-hoc trained model tends to be a bit better. The validation results for both models seem to prove this because the results are very close together.

The most interesting aspect is the epoch when a new model is trained based on a previous iteration. For the first iteration, the ad-hoc model and the iterative model perform equally. Only the cls loss has one outlier. But coming close to the 50th epoch, the iterative model seems to outperform the ad-hoc model by some margin. This is mainly due to *YOLOv8*'s dataloader. The mosaic dataloader is configured to shut itself off if the last ten epochs start. The mosaic dataloader applies the mosaic augmentation to the input images. Several images are put together to form a mosaic. The bounding boxes of each image are adjusted to match the newly created image. Next, a random crop of the mosaic is taken. The bounding boxes lying in the crop are also reshaped. The crop is then given as training input to the model. Without the mosaic dataloader, the model learns for the last ten epochs on the full images of the training dataset.

The mosaic augmentation was introduced in *YOLOv4* and achieved a performance boost regarding generalizability and accuracy.

With the 51st epoch, the ad-hoc model performs much better because the iterative model does not generalize well to the mosaic-augmented images.

Like in the first iteration, the iterative model performs at the end better than the ad-hoc model but the margin is not as big as it was in the first iteration.

The difference can be seen at the end of the third iteration. The iterative model does not come close to the performance of the ad-hoc model regarding the box loss and the dfl loss.

Model	Precision	Recall	mAP50	mAP50-95
ad-hoc	0.76567	0.60209	0.67059	0.4252
ad-hoc (after 150 epochs)	0.72618	0.60225	0.68004	0.42084
iterative	0.78987	0.59341	0.65961	0.39427

Table 8: Precision, Recall, mAP50, and mAP50-95 at the end of the training for the iterative and ad-hoc trained model.

Table 8 shows the results for the iterative and ad-hoc trained model after each training. The results indicate that training a model iterative benefits the precision. This can be because, at the end of the training, the precision is calculated on input images without the mosaic dataloader. Since the iterative model was trained already twice before the training ended, the model seems to perform better. The recall is also comparable between both models. The difference is not large at all.

The main differences are regarding the mAP50 and the mAP50-95 where the ad-hoc trained model seems to outperform the iterative trained model. This is also the case for the ad-hoc model after the 150th epoch.

The reason behind this could be that the ad-hoc model learns better to generalize and predict the bounding boxes better. This is substantiated by the box loss plot in Figure 36. The box predictions of the iterative model are not as precise as those of the ad-hoc trained model.

For the pipeline, I'm using the ad-hoc trained model as the object detection model for traffic lights.

4.1.7 Vehicle Model

The vehicle model is trained for ten epochs on the *nuImages* dataset using only the categories of vehicles.bicycle, vehicles.bus.bendy, vehicle.bus.rigid, vehicle.car, vehicle.construction, vehicle.emergency.ambulance, vehicle.emergency.police, vehicle.motorcycle, vehicle.trailer, and vehicle.truck.

For the training, I've used a fixed imgsz-parameter of 640. With a larger imgsz-parameter, the model would need much more time to be trained for the same amount of epochs.

The labels are taken from the original *nuImages* data annotation definitions. Furthermore, the dataset's structure is described in section 4.1.3. For the validation, the validation split is used.

Results

The different metrics were tracked during the training and validation. The progression of the box and class loss, as well as the dfl for training and validation, can be seen in Figure 37. After each epoch, the validation is carried out on the afore-created data split.

All plots show that the loss for training as well as for validation continually went down. The validation loss is generally below the training loss. The validation loss comes below 0.9, while the training loss is above 0.9. The cls loss decreases also for training and validation, reaching less than 0.7. The dfl loss went down below 1.0. I have to note that all loss curves show a falling trend, indicating that the model is not trained well or long enough. The results can be improved by running the method for a longer period of time. And for more epochs.

The same applies to the validation metrics.

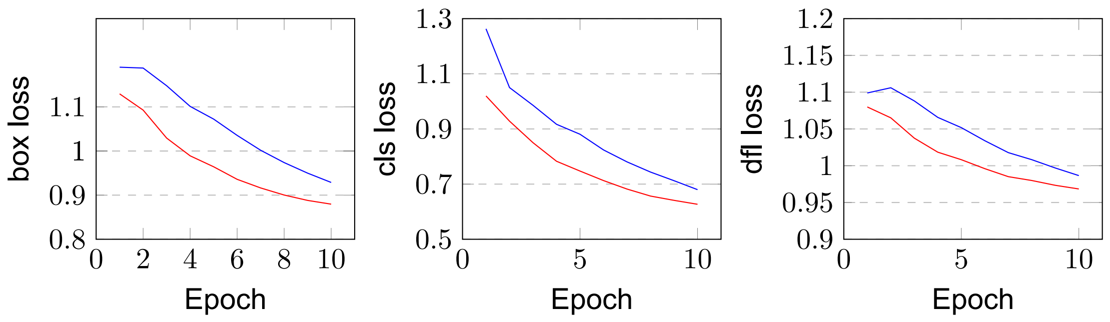


Figure 37: Training (blue) and validation (red) evaluation of the vehicle model for each epoch. From top to bottom: performance of predicting the bounding boxes (box loss), performance of predicting the class of the object correctly (cls loss), performance of distribution of bounding box boundaries (dfl loss).

precision	recall	mAP50	mAP50-95
0.7893	0.42561	0.47024	0.32332

Table 9: Precision, Recall, mAP50, and mAP50-95 values after ten epochs at the end of the training

Table 9 shows the final results regarding the precision, recall, mAP50, and mAP50-95 after the training. The values seem not to be good but looking at the tracked data, one can see that the recall, mAP50, and the mAP50-95 have a rising trend. The recall rose from 0.275 in the first epoch to 0.425 after the 10th epoch, indicating that the model can perform significantly better if trained for longer.

Table 10 shows the individual results for each class divided into precision, recall, mAP50, and mAP50-95. The results are okay, mostly above 0.6 regarding the precision and above 0.4 regarding the recall. To note is the fact, that the performance of these metrics is taken as the average across all classes. This means that the precision value for each class is summoned and divided by the number of classes. This adds a pinch of salt to the credibility of the evaluation. For instance the vehicle.emergency.ambulance class has eight annotations across the whole validation split. The recall is zero and pushes the overall recall down. If we remove the vehicle.emergency.ambulance and vehicle.emergency.police class from the performance summary, we would get a recall of 0.53, which would be 0.11 points higher than the overall recall. This indicates, that the results can be easily improved if more samples of emergency vehicles are added, which subsequently would increase the recall if the annotations are given.

Class	Precision	Recall	mAP50	mAP50-95
all	0.79	0.426	0.47	0.323
vehicle.bicycle	0.698	0.681	0.714	0.488
vehicle.bus.bendy	1	0.069	0.137	0.0799
vehicle.bus.rigid	0.739	0.58	0.647	0.5
vehicle.car	0.798	0.794	0.85	0.619
vehicle.construction	0.663	0.483	0.551	0.305
vehicle.emergency.ambulance	1	0	0.00168	0.00151
vehicle.emergency.police	1	0	0.00582	0.00506
vehicle.motorcycle	0.754	0.758	0.796	0.536
vehicle.trailer	0.552	0.284	0.333	0.207
vehicle.truck	0.694	0.608	0.668	0.492

Table 10: Summary of the model's performance.

The raw data are available in the *GitLab* repository of the *Automated Data Labeling Pipeline*.

4.1.8 Pedestrian Model

The model for pedestrian detection is also trained on the *nuImages* dataset and for 20 epochs. The used categories are human.pedestrian.adult, human.pedestrian.child, human.pedestrian.construction_worker, human.pedestrian.personal _mobility, human.pedestrian.police_officer, human.pedestrian.stroller, and human.pedestrian.wheelchair.

While training the model, I experimented with the number of epochs and the imgsz-parameter. Once I used an imgsz of 640 and one of 1280. With a larger imgsz-parameter, the training took 24 hours for ten epochs. The model with the smaller imgsz-parameter was trained for 20 epochs, taking approximately 14 hours.

Like the vehicle dataset, the labels are taken from the original *nuImages* data annotation definitions. Furthermore, the dataset split is the same as well, described in section 4.1.3.

Results

The different metrics were tracked during the training and validation. The progression of the box and class loss, as well as the dfl for training and validation, can be seen in Figure 38. After each epoch, the validation is carried out on the validation data split.

The cls and box loss for the model trained for only ten epochs but with larger imgsz is better compared to the model trained for 20 epochs with an imgsz-parameter of 640. The main reason behind this is mostly the missing of the mosaic dataloader. The mosaic dataloader is not applied for the last ten epochs of the training routine.

Another finding is that all curves went down and had a falling trend at the end of the training. This indicates that the models can be improved by training for longer.

The dfl loss is much worse for the model with the larger imgsz-parameter than the smaller imgsz-parameter.

In Table 11, the precision, recall, mAP50, and mAP50-95 are illustrated for both tested models. The results confirm that training a model with a larger imgsz-parameter is beneficial. Especially the recall benefits from this change.

Another interesting observation is that all metrics went down for the model trained with an imgsz-parameter of 640. The reason behind this could be irrelevant measurement errors or the fact that the mosaic dataloader is removed for the last ten epochs. It seems that the model can not generalize well enough. An idea to mitigate this could be to train the model for more epochs.

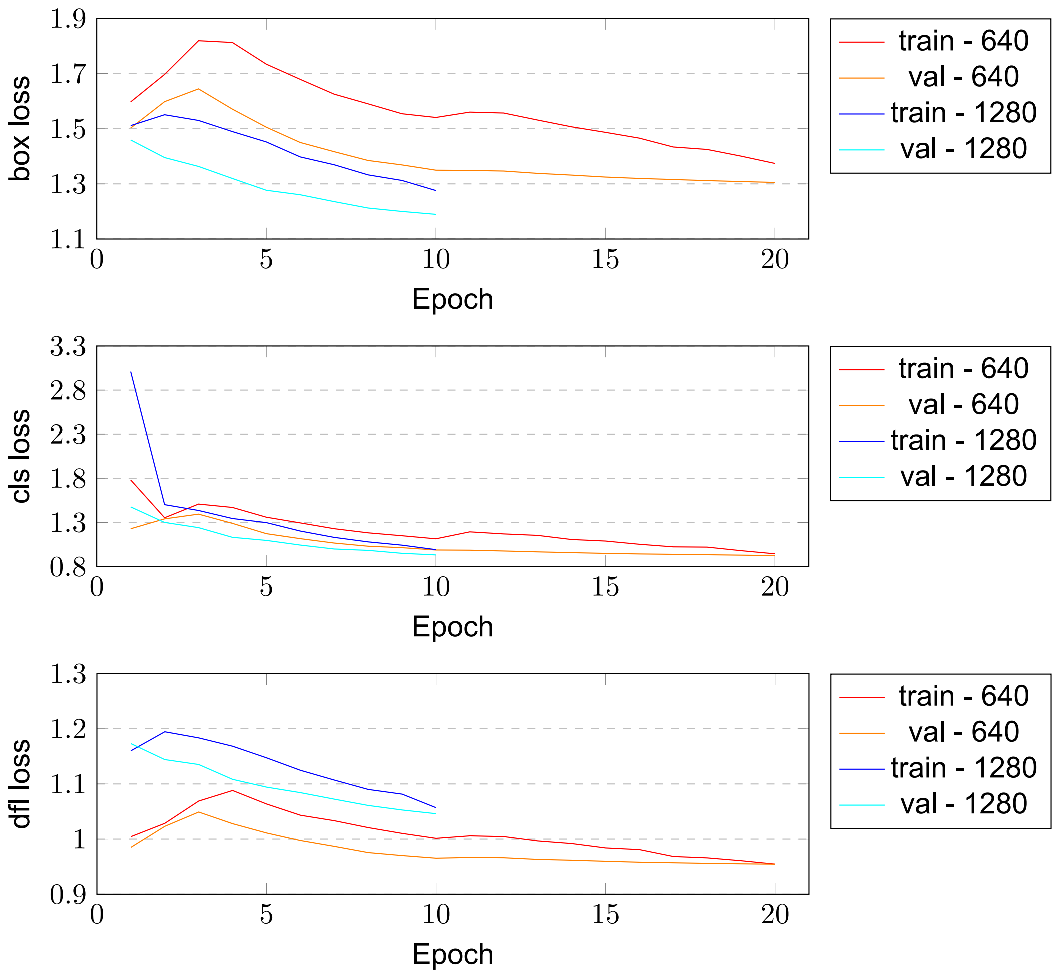


Figure 38: Training and validation evaluation of the tested pedestrian models. The blueish lines show the model trained with an imgsz of 1280. The reddish lines show the model trained with an imgsz of 640. From top to bottom: performance of predicting the bounding boxes (box loss), performance of predicting the class of the object correctly (cls loss), performance of distribution of bounding box boundaries (dfl loss).

Model	Precision	Recall	mAP50	mAP50-95
imgsz 640	0.60588	0.31044	0.32011	0.17556
imgsz 640 (after 10 epochs)	0.65661	0.27471	0.28991	0.15438
imgsz 1280	0.5851	0.36569	0.3566	0.21505

Table 11: Precision, Recall, mAP50, and mAP50-95 at the end of the training for the iterative and ad-hoc trained model. The results after ten epochs of the smaller imgsz model are shown too.

For the final pipeline, I use the model with the larger imgsz-parameter.

4.1.9 Run Time

I've tested each model individually regarding its run time. They need approximately three milliseconds per image, depending on how many objects are in there and how difficult are they to find. The cause of this is that I'm using *YOLOv8* as an object detector for all models. Its detection performance enables real-time image processing and its speed is in the milliseconds.

Furthermore, I've tested how long all models together need to process 100 images. They take approximately 14 seconds. This is because I measure the time how long the models need to detect the objects and then save the results in the corresponding label file. Breaking down the number to the time to process one image, the models take 0.14 seconds or 7.14 Hertz (Hz).

4.2 Hungarian Algorithm

I've tested two different *Python* packages for the implementation of the *Hungarian Algorithm* regarding their run-time. One with *NumPy* and one with pandas. Furthermore, I've experimented with several implementation approaches with *NumPy*.

Following *NumPy* implementation variants are tested: (1) All computation steps are implemented with for loops and without the use of *NumPy* built-in functions. (2) The Step 1 implementation is changed from a for loop to *NumPy*'s built-in function. To calculate the row minimum, the *NumPy* min function is used. To subtract it from each row, a for loop is used to iterate through the rows of the matrix. The minimum of each column is also determined by the *NumPy* min function, but the subtraction is performed with the *NumPy* subtract function instead of an iteration through the columns. (3) The implementation of Step 2-1 is replaced with a *NumPy* based one, instead of using a for loop. (4) The nested for loops of Steps 4-1, and 4-2 are replaced with NumPy functions to calculate the minimum non-marked number and subtract it from each non-marked matrix entry. (5) The implementation of Steps 4-3 is changed, in such a way, that the minimum non-covered number is added to each entry that is covered twice. Instead of using a nested for loop list comprehension is used. (6) The final variation uses an alternative implementation of steps 4-1, 4-2, and 4-3. Instead of using multiple nested for loops, *NumPy* functions are used.

The run-time of the different variants is determined with the *timeit* python package. *timeit* takes the function, which one wants to be tested, and the number of how many runs tested. I've used 100000 runs and performed the evaluation five times to get an average time and have a more meaningful result.

The run-time results of the *NumPy* implementation variants are shown in Table 12. All methods are tested on the same benchmark and with the same amount of runs. Additionally, all variants are tested on a computer with the following configuration: Intel i7-9700KF CPU @ 3.60GHz and 32GB RAM.

The different variants are all run on the CPU without GPU support.

Variants	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Run 1 [ms]	2.906	2.819	3.432	3.135	2.994	2.991	6.181
Run 2 [ms]	2.900	2.799	3.444	3.079	2.963	3.029	6.091
Run 3 [ms]	2.923	2.811	3.433	3.081	2.962	3.025	6.032
Run 4 [ms]	2.934	2.808	3.461	3.086	2.964	3.025	5.925
Run 5 [ms]	2.935	2.814	3.488	3.086	2.966	3.016	5.860
Average [ms]	2.920	2.810	3.452	3.094	2.970	3.017	6.018

Table 12: Results of evaluating different implementation variants, using the *NumPy* and *pandas* package. The variants (1), (2), (3), (4), (5), and (6) are all using the *NumPy* package. The variant (7) is implemented with the *pandas* package. All shown times are in milliseconds per .1000 runs.

Compared to the standard variant with multiple loops, variant (2) is faster by approximately 0.1 ms over 1.000 runs. This is mainly due to the more efficient implementation of *NumPy*'s built-in functions. The min function of *NumPy* is faster than a for loop because it is implemented in c. The 0.1 ms improvement is worth the expense because the algorithm is just tested on a matrix of shape 10x15, resulting in 150 matrix entries. This is mainly due to the application in the sensor fusion stage, where the LiDAR data are projected onto the corresponding image plane and not all found LiDAR objects are considered. Many LiDAR scans may produce many more object proposals that have to be tracked in the 3D space. This would lead to an increased gap between the different variants due to the bigger matrix size.

The other variants are mostly slower compared to (1), while the variant (5) is only 0.05ms slower.

All in all, the most alternative implementations make the Hungarian Algorithm slower. This is mostly due to the amount of data. If the *NumPy* matrix gets bigger, a for loop is more costly, compared to a *NumPy* function because of its implementation details. Hence, the variant (2) is used as implementation.

The used *NumPy* implementation can be found in Source Code 6.2. The alternative implementations that are not used are added as comments.

4.3 LiDAR Processing

The LiDAR processing stage consists of several algorithms that compute different things on the input data. The following components are evaluated: Point cloud filtering by a ground grid approach, based on [2], point cloud filtering by an initial and final classification with a *Markov-Random Field*, based on [32], and point cloud clustering with a variant of the *DBSCAN* algorithm.

The performance of the *Hungarian Algorithm* is not evaluated again. The overall performance does not change. The distance between the centroids is now calculated based on 3D data and not on 2D data. The run-time does not change either because the implementation is not changed.

4.3.1 Evaluation Procedure

The evaluation of both approaches is oriented on the evaluation metrics of the GroundGrid paper. A classification is considered as *True Positive* (TP) if the point is predicted as ground and is by ground truth also classified as ground. A classification is *False Positive* (FP) if the point is predicted as ground, but in ground truth, the point is classified as non-ground. *True Negative* (TN) occurs when a point is predicted to be non-ground, while the ground truth is classified as non-ground as well. A point classified as non-ground, while the ground truth is ground is referred to as *False Negative* (FN).

4.3.2 Point Cloud Filtering with Markov Random Fields

I've evaluated the channel-based initial classification only because the run-time of the LBP is so bad, that a single iteration takes almost 1 hour. The iteration is performed five times. Furthermore, the evaluation should be carried out on the same data as the *GroundGrid*. Some of the evaluated sequences have over 4.000 LiDAR scans. This would result in a high time consumption.

Sequences	Precision	Recall	F_1	IoU	Accuracy
00	44.23	55.81	49.35	32.75	49.65
01	76.94	88.34	82.25	69.85	70.97
02	72.61	81.10	76.62	62.10	65.19
03	38.28	50.47	43.54	27.83	53.73
04	64.43	67.00	65.69	48.91	55.33
05	55.67	58.18	56.90	39.76	49.81
06	63.39	64.30	63.84	46.89	54.91
07	48.30	56.50	52.08	35.20	54.21
08	68.26	75.27	71.60	55.76	60.78
09	50.46	63.64	56.29	39.17	49.47
10	70.66	70.35	70.51	54.45	57.27

Table 13: Performance of the channel-based initial classification regarding the first 40 LiDAR scans of the *SemanticKITTI* dataset.

Table 13 shows the precision, recall, F_1, IoU, and Accuracy of the channel-based initial classification of the first point cloud filtering approach. Compared to the *GroundGrid*, the results are very bad. The channel-based initial classification is not meant to work on its own in point cloud filtering. It is the input to the re-classification based on the solution of the LBP algorithm.

If the LBP algorithm were implemented more efficiently, one could carry out the same evaluation but for this approach.

4.3.3 Point Cloud Filtering with GroundGrid

The evaluation of the ground segmentation with *GroundGrid* is carried out following the procedure of the original paper. Hence, I have used the *SemanticKITTI* dataset [33].

The *SemanticKITTI* dataset is based on the *KITTI Vision Benchmark*, providing additional semantic annotations for all sequences of the *Odometry Benchmark*. The authors *"labeled each scan resulting in a sequence of labeled point clouds, which were recorded at a rate of 10Hz"*. Furthermore, they annotated moving and non-moving traffic participants. The classes of cars, trucks, motorcycles, pedestrians, and bicyclists are introduced to distinct the participants.

Table 14 shows the qualitative results of the evaluation. From top to bottom, I compare my *GroundGrid* implementation with the original one. Furthermore, the performance of further ground segmentation methods, such as *Patchwork++*, *JPC*, and *GndNet* is shown as well. The data are taken from the evaluation carried out in the *GroundGrid* paper.

The evaluation is carried out on the same sequences as the paper does.

For the evaluation, I've used multiprocessing and created four processes that run the evaluation of four different sequences in parallel. In the end, they write the results in the same text file for further inspection and evaluation.

The overall results are comparable to the original paper and the other listed methods but mainly worse. Especially sequence 03 is worse. The original *GroundGrid* achieved a precision of 97.96 (green), while my implementation achieved only 79.28 (red). This means it classifies many non-ground points as ground. Interestingly, the recall of sequence 03 is almost the same as the original *GroundGrid* ones (95.68 (red) to 97.95 (green)). This indicates that mine implementation makes also a few mistakes while classifying ground points as non-ground points.

Furthermore, mine implementation outperforms the original *GroundGrid* regarding the recall. For instance, the sequence 04 achieves a recall of 98.04 compared to 97.85. But this is no overall trend it's an outlier. Only sequence 02 achieves the same result in the context of recall.

Interestingly, the recall is for all sequences quite high, indicating that the *GroundGrid*, without odometry data, classifies most of the ground points as ground points.

Nevertheless, the precision is over all sequences worse.

I assume, that the bad performance of the precision lies in the classification of the ground points. Mine implementation classifies many points as ground, generating more false positive results than the original one. This is based on the analysis of the precision values. For instance, the sequence has a precision of 79.28, while the recall is 95.68. This indicates that the implementation gets the most ground points right, but struggles with false positives.

To improve these results, one could use the odometry data for the different scans or carry out a fundamental error analysis.

sequences	00	01	02	03	04	05	06	07	08	09	10	average
						Precision						
Patchwork++	94.99	**98.27**	95.96	96.81	98.18	92.65	97.86	93.29	96.97	96.06	92.81	95.80
GndNet	92.40	96.54	93.74	95.60	97.30	89.58	96.15	90.09	95.09	93.81	88.34	93.51
JPC	**96.78**	97.97	**97.50**	**98.09**	99.01	94.03	**97.96**	**95.65**	**97.97**	**97.64**	95.27	**97.08**
GroundGrid	96.05	98.01	97.36	97.96	**99.08**	**95.19**	97.82	95.31	97.50	97.25	**95.38**	96.99
GroundGrid (mine)	92.85	86.28	93.38	79.28	91.08	92.12	95.75	93.76	93.47	90.96	90.03	90.81
						Recall						
Patchwork++	98.67	96.52	97.20	**98.17**	97.21	98.13	97.39	98.42	97.41	96.45	**95.93**	97.41
GndNet	**99.50**	**96.91**	96.94	96.68	**99.06**	**98.69**	**99.00**	**99.44**	**98.74**	96.14	93.60	**97.70**
JPC	97.20	95.46	93.72	94.86	96.91	95.64	96.23	96.53	95.13	92.66	88.47	94.97
GroundGrid	98.70	96.17	**97.71**	97.95	97.85	98.13	98.38	98.72	97.79	**96.91**	95.90	97.96
GroundGrid (mine)	97.39	96.63	95.71	95.68	98.04	96.73	96.95	97.10	94.84	95.11	92.12	96.02
						F1						
Patchwork++	96.80	**97.39**	96.58	97.49	97.69	95.31	97.63	95.79	97.19	96.25	94.35	96.59
GndNet	95.82	96.72	95.31	96.14	98.17	93.91	97.55	94.53	96.88	94.96	90.89	95.53
JPC	96.99	96.70	95.57	96.45	97.95	94.83	97.09	96.09	96.53	95.09	91.74	95.91
GroundGrid	**97.35**	97.08	**97.54**	**97.96**	**98.46**	**96.64**	**98.10**	**96.99**	**97.64**	**97.08**	**95.64**	**97.32**
GroundGrid (mine)	95.06	91.16	94.53	86.71	94.43	94.37	96.35	95.40	94.15	92.99	91.21	93.31
						Accuracy						
Patchwork++	96.64	**95.96**	95.08	96.08	96.39	94.79	96.63	95.88	96.37	94.90	93.75	95.68
GndNet	95.53	94.88	93.20	93.99	97.10	93.10	96.46	94.52	95.91	93.08	89.81	94.33
JPC	96.89	94.93	93.80	94.59	96.81	94.37	95.89	96.26	95.60	93.50	91.35	94.91
GroundGrid	**97.24**	95.50	**96.48**	**96.84**	**97.60**	**96.32**	**97.29**	**97.08**	**96.97**	**96.05**	**95.25**	**96.60**
GroundGrid (mine)	96.11	89.32	94.91	86.68	94.07	95.33	95.46	96.32	94.83	93.29	93.54	93.62
						IoU						
Patchwork++	93.79	**94.90**	93.38	95.09	95.49	91.04	95.36	91.91	94.53	92.78	89.30	93.41
GndNet	91.97	93.65	91.04	92.55	96.41	88.52	95.22	89.63	93.94	90.41	83.30	91.51
JPC	94.15	93.61	91.52	93.14	95.98	90.16	94.34	92.47	93.29	90.64	84.75	92.19
GroundGrid	**94.84**	94.33	**95.19**	**96.00**	**96.97**	**93.49**	**96.27**	**94.15**	**95.40**	**94.33**	**91.64**	**94.78**
GroundGrid (mine)	90.59	83.76	89.63	76.54	89.45	89.34	92.95	91.20	88.95	86.90	83.84	87.56

Table 14: Accuracy evaluation of my implementation of the *GroundGrid*, compared to the original paper and further works [2].

Run-Time

The run-time of my implementation is much slower than the original implementation. This is mainly due to the use of *Python* as the programming language. Unfortunately, multithreading or multiprocessing is in *Python* not as easy as it is in *C++*.

I've tried several methods to speed up the ground detection, but neither worked well. In multiprocessing, allowing all processes to work on a common memory, one has to create a shared memory. Unfortunately, the access time to the shared memory is much higher than computing on a local, process-bound memory. This is the reason why the run-time for one LiDAR scan is approximately 38.36 seconds, corresponding to 0.026Hz, compared to 171Hz of the original implementation. The most time-consuming task is the detection of ground patches. This takes approximately 37 seconds, while the rasterization and classification take only around 1 second.

Multiprocessing in *Python* is only preferable if the computation runs on large data and a common memory is not needed. Otherwise, the access time to the shared memory makes the overall run-time worse.

4.4 Sensor Fusion

The Sensor Fusion evaluation is a bit tricky because several steps have to be considered that may have an impact on the results. For instance, fusing LiDAR data with camera images requires first ground filtering to remove ground points and second clustering to determine the possible object proposals. In the end, the LiDAR data are projected onto the corresponding image plane to match the different objects in the 2D plane.

4.4.1 LiDAR to Camera Projection

I've tested the already-performed LiDAR/camera calibration for the autonomous vehicle, stored in the *fub_mig* repository. Unfortunately, the calibration is poor and impacts the object matching a lot. The LiDAR points are projected with a rotation to the left and a pitch to the top compared to the underlying image. This is no issue regarding one camera. It relates to all of them. Figure 39 shows the calibration regarding only LiDAR objects found by the *DBSCAN* algorithm.

To mitigate this, I've re-calibrated the extrinsic parameters x, y, z, roll, pitch, and yaw such that the LiDAR objects are as close as possible to the corresponding visual objects.

My re-calibration indicates that the main issue is either the pitch and roll are not perfectly calibrated or the *Velodyne* sensor as a unit is installed with a small rotation to the left. Furthermore, the cameras might be orientated downwards. This may reason why the LiDAR projections are higher than the visual objects.

The problems of the LiDAR projection of the left camera are shown in two figures. Figure 39 shows the projection before and Figure 40 after the re-calibration. The new parameters are listed in the Appendix Table 17.

I've experimented with the *Velodyne*'s extrinsic parameters and achieved also changes. But changes regarding one camera made the other camera projection worse. Hence, I assume that the poor calibration comes not only from the calibration of the *Velodyne* model.

The projection onto the four cameras is different. For instance, the projection onto the rear camera is almost perfect compared to the other ones. A small rotation to the side was only needed.

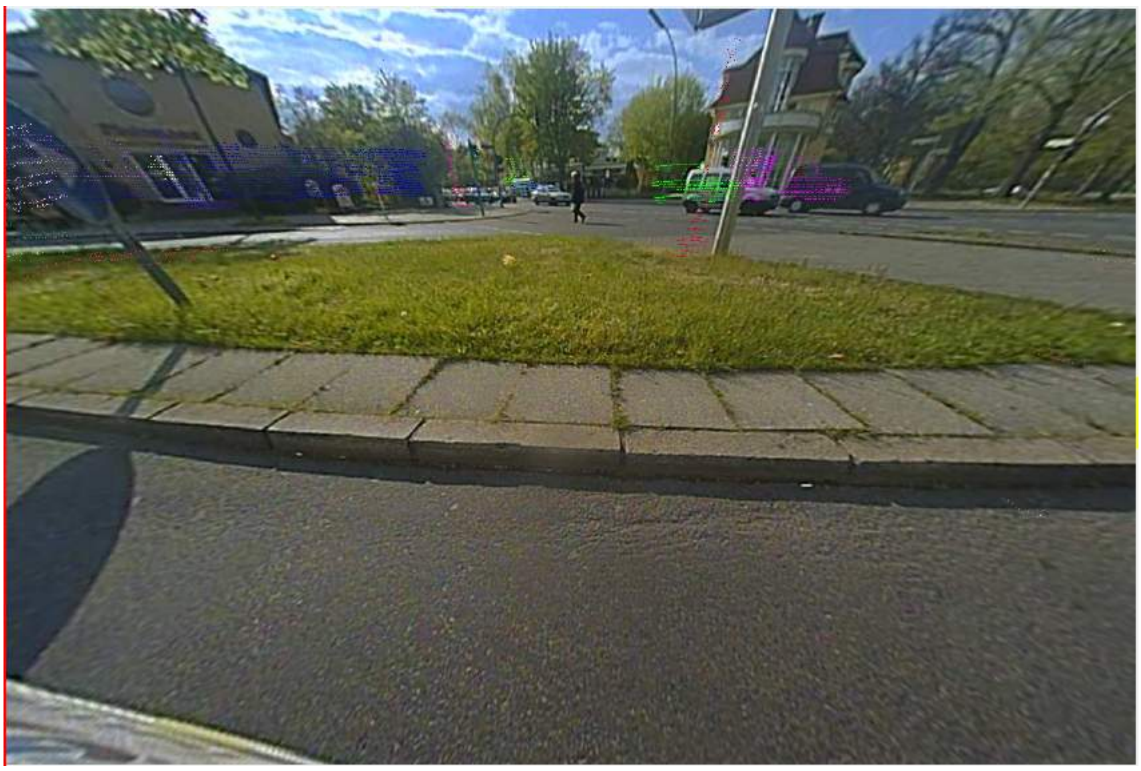


Figure 39: Poor calibration of the LiDAR to the left camera..

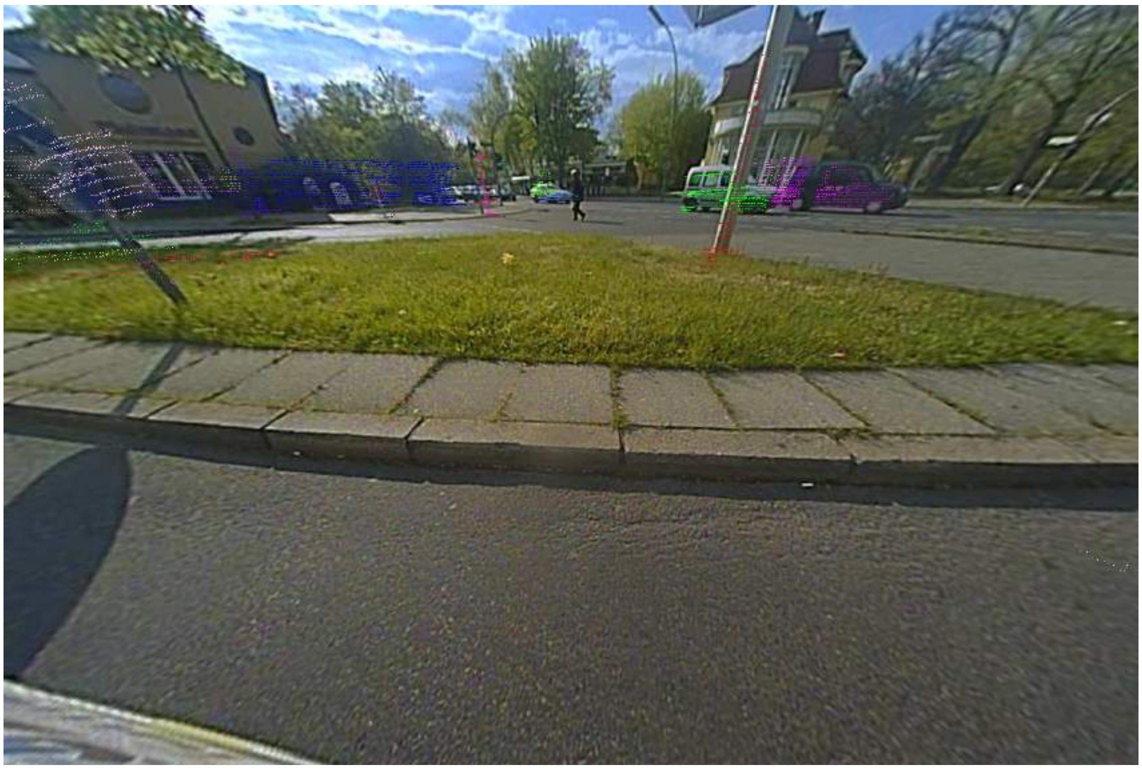


Figure 40: Better calibration of the LiDAR to the left camera.

4.4.2 Object Matching

For the evaluation I use the *KITTI dataset for Object Tracking Evaluation* [6]. More precisely, I use the *Velodyne* point clouds and the left color images of the tracking data set. From the 21 training sequences, I use one because the evaluation takes some time.

The evaluation works as follows: First, the LiDAR scan is loaded as pandas DataFrame. Then, ground segmentation is applied to the point cloud data to gather only non-ground points. Next, the remaining point cloud is clustered by the *adaptive DBSCAN* to find objects. Simultaneously, the different object detection models are applied to the corresponding images.

Before applying the *Hungarian Algorithm* to match visual objects with LiDAR objects, the occlusion check is carried out. This ensures that only visible LiDAR clusters are matched to the corresponding visual objects.

Next, the *Hungarian Algorithm* is executed. For this, the distances between the bounding box centroids of the visual objects and the centroid of the LiDAR objects are computed and given as input to the matching algorithm.

The output of the *Hungarian Algorithm* is checked for sanity. The re-checked output is then used to evaluate the performance of the sensor fusion.

The performance of the sensor fusion is measured by a subjective evaluation of the results of the implemented sensor fusion routine.

The Figures 41, 42, 43, 44 show the tested images with corresponding LiDAR projections and sensor fusion results.

The first thing to notice is that the clustering of the LiDAR data is not perfect, especially if the objects are further away. For instance, in Figure 41 the red cluster stretches from the left to the right side of the image and is mapped to the first car of the same driving lane. This is obviously not right. The same can be also seen in Figure 43.

Another point to notice is that the traffic sign in Figure 42 was detected and a LiDAR point cloud is mapped to it. The point cloud is not 100 percent right but it shows that the sensor fusion also takes these objects into account.

An example of a bad performance is the Figure 44. The *DBSCAN* did not achieve a good enough clustering of the objects such that a big cluster is mapped to the detected car.

All in all, the performance of the sensor fusion stage depends strongly on the performance of the *DBSCAN*. If the *DBSCAN* clusters the objects successfully, a cluster can be mapped to each found object. For instance, in Figure 41, the closest cars have a cluster assigned that represents the cars in the LiDAR data. This indicates that the *DBSCAN* algorithm struggles with objects that are far away.

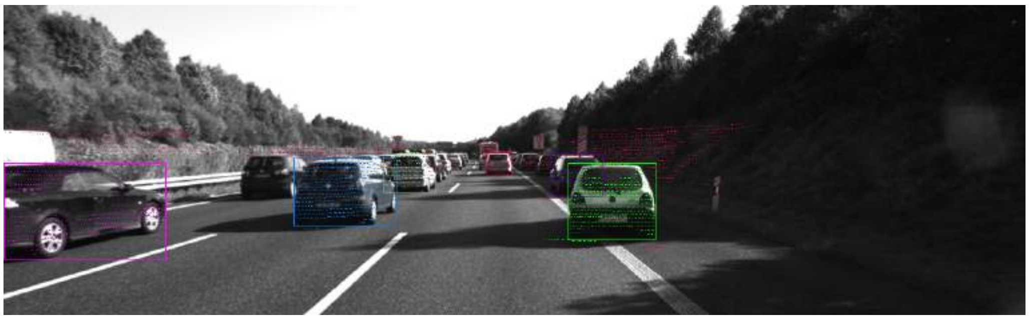


Figure 41: Sensor Fusion evaluation on the image *000000.png* of sequence 20 of the *KITTI* dataset. Only objects that could be mapped to a LiDAR object are shown. The colors are the same for a bounding box and the corresponding LiDAR object.

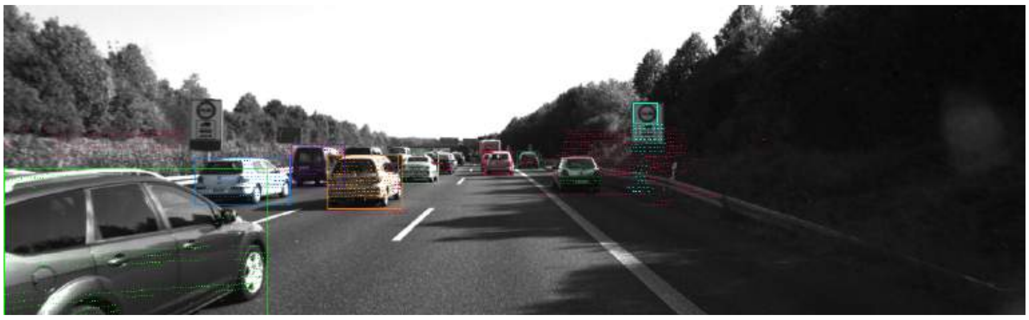


Figure 42: Sensor Fusion evaluation on the image *000111.png* of sequence 20 of the *KITTI* dataset. Only objects that could be mapped to a LiDAR object are shown. The colors are the same for a bounding box and the corresponding LiDAR object.

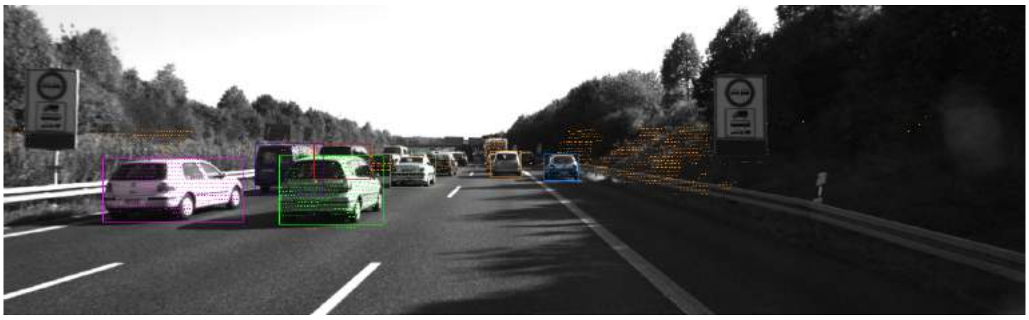


Figure 43: Sensor Fusion evaluation on the image *000129.png* of sequence 20 of the *KITTI* dataset. Only objects that could be mapped to a LiDAR object are shown. The colors are the same for a bounding box and the corresponding LiDAR object.

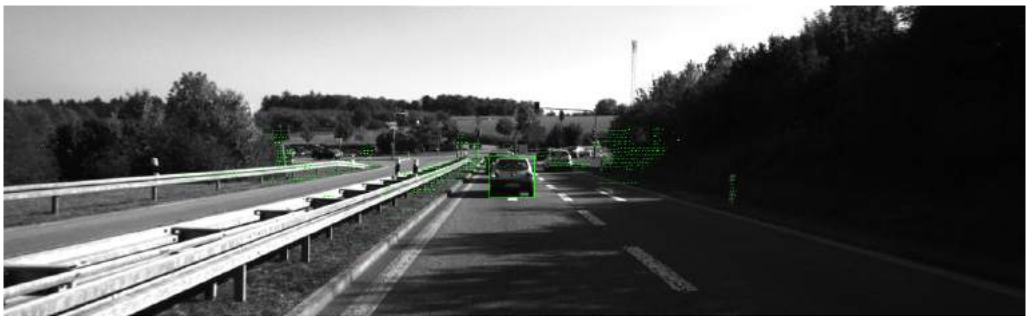


Figure 44: Sensor Fusion evaluation on the image *000753.png* of sequence 20 of the *KITTI* dataset. Only objects that could be mapped to a LiDAR object are shown. The colors are the same for a bounding box and the corresponding LiDAR object.

5 Conclusion

In this chapter, the results from this thesis are described and discussed in detail. Decision processes are justified and presented. Furthermore, *Limitations* and possible *Future Work* are pointed out.

5.1 Discussion of Results

A tool for automatic data labeling of highly complex driving recordings to support a human labeler by his task, was necessary to increase the amount of well-annotated, available, and distinct data. The unlabeled data were gathered, while the autonomous vehicle was in motion. They consist of LiDAR, camera, and radar data.

The image processing stage is implemented in *Python* and consists of the object detection and localization in images of the front, left, right, and rear camera, and the semantic segmentation of these images. The object detection is distributed over several *YOLOv8* models to guarantee a fast and accurate detection and enable the possibility of executing all models simultaneously. The segmentation is implemented based on the visual features of a *DINOv2* ViT feature extractor. The segmentation map is quite good, although it works on a patch-wise level and not a pixel-wise level. Both results are used to enhance the performance of the pipeline. Especially the segmentation map provides an additional filter stage for the Sensor Fusion.

The LiDAR processing stage is implemented in *Python* and consists of LiDAR filtering, clustering, and object tracking. The filtering achieves an accuracy of 93.62 %, evaluated on the *SemanticKITTI* dataset. The implementation of the clustering is based on a variant of *DBSCAN*. It estimates the eps and minPts parameters dynamically, based on the points' distance to the LiDAR sensor and the density within the corresponding grid cell. Unfortunately, some objects are split across two clusters due to a possible connection loss between both clusters. The object tracking works as expected. Objects in consecutive frames are tracked successfully and can be illustrated with the same color for the same object.

At last, the sensor fusion stage is implemented in *Python* and consists of an occlusion check, segmentation map filtering, object matching between visual and LiDAR objects, and the post-processing of the results. The assumption that only near objects can occlude other objects is used for the implementation of the occlusion check. The algorithm detects occluded objects successfully and returns only the non-occluded ones. The object matching implementation uses almost the same underlying algorithm as the object tracking of the LiDAR object tracking. It matches the different LiDAR projections to the corresponding visual objects. The labels are stored for all matched LiDAR objects to execute the post-processing. In the post-processing, all LiDAR scans are processed again. While accessing each scan, the gathered object/label matching is used to classify the most LiDAR objects possible. If a label can not be assigned to a cluster, it is considered unclassifiable but remains in the dataset as an object.

A user interface is implemented to visualize the results of the different pipeline stages. Furthermore, additional functionalities such as the correction of missclassifications in the object detection stage are implemented. These changes can be stored directly on the system.

Unfortunately, the last step to create a final dataset from the processed data could not be achieved. However, the data are available because each stage saves the data on the host system. The only missing part is a function that takes the processed data with the results, strips off the unnecessary information, and stores them in one dataset folder. All other information are persistent on the system.

All in all, I've successfully implemented the first *Automated Data Labeling Pipeline* for driving recordings, saved as a *ROS* bag file of the *Dahlem Center for Machine Learning and Robotics*. The pipeline is usable either with a *Python* console or with the provided user interface. Furthermore, a German traffic sign dataset is created using the original *GTSDB* and extended with several additional traffic signs. Additionally, the labels of the traffic signs are taken from the official traffic sign catalog.

5.2 Limitations

The main goal to conceptualize, develop, and evaluate an automated data labeling pipeline for driving recordings is achieved. LiDAR point cloud data and camera images are fused at the end, such that objects are labeled across different scans, and occluded objects can be handled too.

The training of the different object detection models consumed different amounts of time due to the size of the training dataset. For instance, training a traffic sign model took three hours on 20 epochs, while training a vehicle model on the *nuImages* dataset took two days, also on 20 epochs. The models can be enhanced by allowing a higher training duration. I limited the number of epochs due to time restrictions of the work. Furthermore, with another training computer, the training may be way faster due to more computational resources.

The projection of the LiDAR data onto the corresponding camera images is not very good. This is due to the quite bad calibration between the LiDAR sensor and the camera sensors. The LiDAR point cloud does not match the objects in the camera image quite well. This affects the performance of the pipeline, especially, the sensor fusion step. To mitigate this in the first instance, I've re-calibrated the camera's extrinsic parameters and achieved better results. To achieve better sensor fusion and object-matching performance, a new LiDAR camera calibration is needed. Because of the amount of work and the time restriction, I couldn't do the calibration by myself and had to rely on manual re-calibration.

Another limitation is that the integration of radar data into the pipeline is not implemented. Due to the high workload regarding the LiDAR data and the final sensor fusion, an integration could not be fulfilled.

5.3 Future Work

The proposed data labeling pipeline is restricted by the data I've used to train the different object detection models. Furthermore, the algorithms used are not 100 percent accurate, leading to possible misclassifications and wrong filtering results.

The pipeline is designed to fuse LiDAR data with camera images. An autonomous vehicle may use several additional sensors, such as radar. To fuse radar data, an additional pipeline stage has to be added. Radar data are quite similar to LiDAR data. An advantage of Radar is that it provides several more information, such as the Doppler velocity. The proposed LiDAR stage may function as inspiration for how the radar processing stage may work. For object detection for automotive radar clouds, the paper [34] may be a good starting point.

The algorithms used are a bit optimized regarding their run times, allowing fast, precise, and reliable processing of the different kinds of data. Unfortunately, not all algorithms achieve a fast processing speed. For instance, the adaptive *DBSCAN* algorithm for LiDAR point cloud data clustering is computationally expensive. Its run time varies between one and two hours, depending on the number of non-ground points. One may optimize the implemented *DBSCAN* algorithm or use another algorithm to achieve better run-time performance while being as accurate or even better than the used implementation. Furthermore, the eps and minPts estimation seems not to be as accurate as expected. Objects that are further away are fused with other objects, resulting in one big cluster for several smaller ones. This can be seen in the *Evaluation* Figure 44. This prevents a better performance of the sensor fusion stage.

The first approach of the ground segmentation could be implemented efficiently and evaluated on the sequences of the *SemanticKITTI* dataset. Because of the long execution of the LBP algorithm, the approach did not apply to testable. However, the channel-based initial classification is tested. One could create an efficient implementation of the LBP algorithm to test this approach as well.

I've created my own German traffic sign object detection dataset. This dataset is far from being complete and achieving an appropriate dataset size for training an object detection model. Future researchers may work on creating good traffic signs and traffic light datasets to increase the performance of the corresponding models. The datasets I created can be used as a foundation and inspiration.

Regarding datasets, one could train a traffic light model on the *DriveU Traffic Light Dataset* (*DTLD*) of *Ulm University* [35]. This dataset is superior compared to the one I've used because it distinguishes different kinds of traffic lights. For instance, a traffic light is not only defined by the traffic phase. The attributes relevance, direction, aspects, orientation, state, occlusion, pictogram, and reflection are added to the annotations. The code to read the images of the dataset is given in Appendix Code 6.3.

Another interesting starting point is to replace the LiDAR clustering with an object detection model on LiDAR data. With this, the computationally expensive *DBSCAN* may be replaced by a faster object detection model. This needs validation and one may experiment on this topic.

An important task is the inter-sensor calibration. The LiDAR to camera calibration is not very good, leading to a necessary manual re-calibration. Several calibration methods are proposed and may be implemented to enhance the projection of the LiDAR data onto a corresponding camera image [36], [37], [38]. This can further increase the reliability and fusion performance of the pipeline.

A GUI reformatting and improving can be an interesting task as well. The provided GUI is only functional and is not designed to be as fast as possible. The GUI can be improved regarding visualization of the results of the individual pipeline stages. Furthermore, such as the correction tool for objects, found in the camera images, a tool to do the same in LiDAR data could be implemented to remove the last wrong classified LiDAR point cloud data. For instance, the LiDAR tool could fuse two clusters that represent the same object but are separated by the *DBSCAN* algorithm.

The functionality of the object detection GUI can be further enlarged. For instance, a tool to add further detection results should be added. If an object detection model detects an object, but a corresponding label cannot be added, the GUI should provide a function to add a new label. Furthermore, the human labeler should be allowed to add further object detections to an already processed image. The *labelimg* tool could be used as an inspiration for this.

References

[1] **M. Oquab**, **T.Darcet**, **T. Moutakanni**, **H. Vo**, **M. Szafraniec**, **P. Fernandez**, **D. Haziza**, **F. Massa**, **A. El-Nouby**, **M. Assran**, **N. Ballas**, **W. Galuba**, **R. Howes**, **P. Huang**, **S. Li**, **I. Misra**, **M. Rabbat**, **V. Sharma**, **G. Synnaeve**, **H. Xu**, **H. Jegou**, **J. Mairal**, **P. Labatut**, **A. Joulin**, **P. Bojanowski**, *DINOv2: Learning Robust Visual Features without Supervision*, arXiv, 2023.

[2] **N. Steinke**, **D. Göhring**, and **R. Rojas**, *GroundGrid: LiDAR Point Cloud Ground Segmentation and Terrain Estimation*, IEEE 2024, IEEE Robotics and Automation Letters, vol.9, n. 1, pp. 420-426, 2024.

[3] **M. E. Yabroudi**, **K. Awedat**, **R. C. Chabaan**, **O. Abudayyeh**, and **I. Abdel-Qadar**, *Adaptive DBSCAN LiDAR Point Cloud Clustering For Autonomous Driving Applications*, IEEE eIT 2022, IEEE International Conference on Electro Information Technology, 2022.

[4] **H. Caesar**, **V. Bankiti**, **A. H. Lang**, **S. Vora**, **V. E. Liong**, **Q. Xu**, **A. Krishnan**, **Y. Pan**, **G. Baldan** and **O. Beijbom**, *nuScenes: A multimodal dataset for autonomous driving*, 2020 IEEE, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2020.

[5] **S. Houben**, **J. Stallkamp**, **J. Salmen**, **M. Schlipsing**, and **C. Igel**, *Detection of Traffic Signs in Real-World Images: The **G**erman **T**raffic **S**ign **D**etection **B**enchmark*, IJCNN 2013, International Joint Conference on Neural Networks, 2013.

[6] **A. Geiger**, **P. Lenz**, and **R. Urtasun**, *Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite*, Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

[7] **A. J. Hawkins**, *Mercedes-Benz is the first to bring Level 3 automated driving to the US*, The Verge, URL: https://www.theverge.com/2023/1/27/23572942/mercedes-drive-pilot-level-3-approved-nevada (Online; Accessed: 20.12.2023).

[8] **A. Krizhevsky**, **N. Sutskever**, and **G. E. Hinton**, *ImageNet Classification with Deep Convolutional Neural Networks*, Curran Associates, Inc., Advances in Neural Information Processing Systems, vol. 25, 2012.

[9] **O. Russakovsky**, **J. Deng**, **H. Su**, **J. Krause**, **S. Satheesh**, **S. Ma**, **Z. Huang**, **A. Karpathy**, **A. Khosla**, **M. Bernstein**, **A. C. Berg**, and **L. Fei-Fei**, *ImageNet Large Scale Visual Recognition Challange (LSVRC)*, International Journal of Computer Vision (IJCV), vol. 115, n. 3, pp. 211-252, 2015.

[10] **K. Simonyan** and **A. Zisserman**, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, International Conference on Learning Representations ICLR, 2015.

[11] **K. He**, **X. Zhang**, **S. Ren**, and **J. Sun**, *Deep Residual Learning for Image Recognition*, Corr, vol. abs/1512.03385, December 10th 2015.

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

[13] **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, vol. 88, n. 2, pp. 303-338, 2010.

[14] **J. Redmon**, **S. Divvala**, **R. Girshick**, and **A. Farhadi**, *You Only Look Once: Unified, Real-Time Object Detection*, arXiv, May 9th 2016.

[15] **J. Redmon** and **A. Farhadi**, *YOLO9000: Better, Faster, Stronger*, arXiv, December 25th 2016.

[16] **J. Redmon** and **A. Farhadi**, *YOLOv3: An Incremental Improvement*, arXiv, April 8th 2018.

[17] **A. Dosovitskiy**, **L. Beyer**, **A. Kolesnikov**, **D. Weissenborn**, **X. Zhai**, **T. Unterthiner**, **M. Dehghani**, **M. Minderer**, **G. Heigold**, **S. Gelly**, **J. Uszkoreit**, and **N. Houlsby**, *An Image is Worth 16x16 word: Transformers for Image Recognition at Scale*, ICLR 2021, International Conference on Learning Representations, January 3rd 2021.

[18] **M. Caron**, **H. Touvron**, **I. Misra**, **H. Jegou**, **J. Mairal**, **P. Bojanowski**, and **A. Joulin**, *Emerging Properties in Self-Supervised Vision Transformers*, IEEE 2021, IEEE/CVF International Conference on Computer Vision (ICCV), pp. 9630-9640, 2021.

[19] **B. Cheng**, **I. Misra**, **A. G. Schwing**, **A. Kirillov**, and **R. Girdhar**, *Masked-attention Mask Transformer for Universal Image Segmentation*, IEEE 20222, IEEE/CVF International Conference on Computer Vision (ICCV), pp. 1280-1289, January 15th 2022.

[20] **M. Desmond**, **E. Duesterwald**, **K. Brimijoin**, **M. Brachman**, and **Q. Pan**, *Semi-Automated Data Labeling*, NeurIPS 2020, Journal of Machine Learning Research, 133, pp. 156-169, 2021.

[21] **A. Sengupta**, **A. Yoshizawa**, and **S. Cao**, *Automatic Radar-Camera Dataset Generation for Sensor Fusion Applications*, IEEE 2022, IEEE Robotics and Automation Letters, vol. 7, no. 2, April 2nd 2022.

[22] **M. Ester**, **H. Kriegel**, **J. Sander**, and **X. Xu**, *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with noise*, Knowledge Discovery and Data Mining, 1996.

[23] **H. Kuhn**, *The Hungarian method for the assignment problem*, Naval Research Logistics (NRL), vol. 52, 1955.

[24] **Mercedes Benz**, *Mercedes-Benz Drive Pilot*, URL:https://www.mercedes-benz.de/passengercars/technology/drive-pilot.html (Online; German; Accessed February 21, 2024).

[25] **N. Mokey**, *A self-driving car in every driveway? Solid-state lidar is the key*, digitaltrends, URL: https://www.digitaltrends.com/cars/solid-state-lidar-for-self-driving-cars/, March 15, 2018 (Online; Accessed: February 21, 2024).

[26] **Fierce Electronics**, *LiDAR vs. RADAR | Fierce Electronics*, URL: https://www.fierceelectronics.com/components/lidar-vs-radar, (Online, ;Accessed: February 21, 2024).

[27] **Z. Wei**, **F. Zhang**, **S. Chang**, **Y. Liu**, **H. Wu**, and **Z. Feng**, *MmWave Radar and Vision Fusion for Object Detection in Autonomous Driving: A Review*, Sensors 2022, vol. 22(7), March 25th 2022.

[28] **A. Vaswani**, **N. Shazeer**, **N. Parmar**, **K. Uszokereit**, **L. Jones**, **A. N. Gomez**, **L. Kaiser**, and **I. Polosukhin**, *Attention Is All You Need*, Curran Associates, Inc., Advances in Neural Information Processing Systems, vol.30, 2017.

[29] **M. Cordts**, **M. Omran**, **S. Ramos**, **R. Rehfeld**, **M. Enzweiler**, **R. Benenson**, **U. Franke**, **S. Roth**, and **B. Schiele**, *The Cityscapes Dataset for Semantic Urban Scene Understanding*, IEEE 2016, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[30] **D. Hendrycks** and **K. Gimpel**, *Gaussian Error Linear Units (GELUs)*, arXiv, 2023.

[31] **E. D. Cubuk**, **B. Zoph**, **J. Shlens**, and **Q. V. Le**, *RandAugment: Practical automated data augmentation with a reduced search space*, arXiv, November 14th, 2019.

[32] **V. Jiménez**, **J. Godoy**, **A. Artunedo**, and **J. Villagra**, *Ground Segmentation Algorithm for Sloped Terrain and Sparse LiDAR Point Cloud*, IEEE Access, vol. 9, pp. 132914-132927, 2021.

[33] **J. Behley**, **M. Garbade**, **A. Milioto**, **J. Quenzel**, **S. Behnke**, **C. Stachniss**, and **J. Gall**, *SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences*, arXiv, 2019.

[34] **N. Scheiner**, **F. Kraus**, **N. Appenrodt**, **J. Dickmann**, and **B. Sick**, *Object detection for automotive radar point clouds – a comparison*, AI Perspect 3, Article number 6 (2021).

[35] **A. Fregin**, **J. Muller**, **U. Krebel**, and **K. Dietmayer**, *The DriveU Traffic Light Dataset: Introduction and Comparison with Existing Datasets*, IEEE International Conference on Robotics and Automation (ICRA), pp. 3376-3383, 2018.

[36] **Z. Chai**, **Y. Sun**, and **Z. Xiong**, *A Novel Method for LiDAR Camera Calibration by Plane Fitting*, IEEE 2018, IEEE International Conference on Advanced Intelligent Mechatronics (AIM), pp. 286-291, 2018.

[37] **Y. Lyu**, **L. Bai**, **M. Elhousni**, and **X. Huang**, *An Interactive LiDAR to Camera Calibration*, IEEE 2019, IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-6, 2019.

[38] **H. Cai**, **W. Pang**, **X. Chen**, **Y. Wang**, and **H. Liang**, *A Novel Calibration Board and Experiments for LiDAR and Camera Calibration*, Sensors, vol. 20, n. 4, 2020.

[39] **Eason**, *Hungarian Algorithm Introduction & Python Implementation*, https://python.plainenglish.io/, URL: https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementation-93e7c0890e15, 2021 (Online; Accessed: February 21, 2024).

6 Appendix

The *Appendix* contains figures, tables, and code related to the automated data labeling pipeline.

Analysis of the Velodyne Rings I've taken two approaches to calculate the growth factor of the distance between each ring, produced by the *Velodyne* sensor. The first approach is to analyze a *ROS* bag driving recording LiDAR scan that was captured without obstacles in the way. The second approach is to calculate the theoretical distances covered by a laser where the angle increases for every ring.

Table 15 shows how I computed the growth factor based on a LiDAR scan. The first 19 rings are removed because they are not representative for the evaluation and and would add some sort of noise. The growth factor between each ring increases exponentially. The final parameter is calculated by averaging over all growth factors.

Source Code 6.1: Calculate the average growth factor for the *Velodyne* model HDL-64E, in *Python*.

```python
from math import tan, radians

z = 1.56                    # mounting height of the sensor
a_inc = 0.4                 # angular increase value per channel
channels = 64               # number of channels

d = [z/tan(radians(a_inc*x)) for x in range(1,channels)]

d.sort()

gf = []                     # result list for the growth values
for i in range(1,len(d)):
    gf.append((d[i]-d[i-1])/d[i-1])

a_gf = sum(gf)/len(gf)  # average growth factor
```

The second approach to calculate the growth factor is to take a look at the datasheet of the corresponding *Velodyne sensors*. The datasheet for the *Velodyne* HDL-64E gives us an angular resolution of 0.4 degrees and a vertical field of view of 26.9 degrees. Each channel's angle increases by 0.4 degrees compared to the prior channel.

The distance $d$to the sensor is calculated by

$$\frac{z}{\tan{(a_{inc} * c)}} = d \tag{6.1}$$

where a_{inc} describes the angular increase per channel, c is the channel number, and z is the mounting height of the *Velodyne* sensor. From the *fub_mig* repository, the mounting height is 1.56 meters for the HDL-64E.

The data are generated and processed by the *Python* script in Code 6.1. The average growth factor is 0.07713135500720329 almost identical with the experimental determined one (0.779381824822261).

Ring	Distance	Growth
20.0	6.876576535748794	0.10403096145448196
21.0	7.591953404278072	0.018896291163408845
22.0	7.735413166304343	0.05918292144779239
23.0	8.193217516091952	0.005819914258260863
24.0	8.240901339534888	0.0794994071079342
25.0	8.896048110062893	0.03203775969950384
26.0	9.181057561688313	0.08626801953682255
27.0	9.973089214788732	0.011105617091013913
28.0	10.083846524822697	0.09684694857968854
29.0	11.060436290697671	0.04271585014583911
30.0	11.532892229838714	0.07640958101245256
31.0	12.41411569298246	0.0749468227722087
32.0	13.34451422169811	0.038633766165058196
33.0	13.86006306372549	0.0533255393527552
34.0	14.599158402061851	0.05464753663318328
35.0	15.396966445652172	0.062305194631396645
36.0	16.356277436781614	0.05395677477736545
37.0	17.238809414634144	0.05567084980745513
38.0	18.198508584415585	0.07459643752882321
39.0	19.556052493150695	0.0692693067018623
40.0	20.91068669117647	0.07559659920465682
41.0	22.491463492063488	0.07664007609206948
42.0	24.215210965517237	0.08654935330863064
43.0	26.311021814814815	0.10123753071281007
44.0	28.974684693877546	0.10922130578698758
45.0	32.13933759090909	0.1048128852141212
46.0	35.50795429268294	0.15088156150899437
47.0	40.86544988235294	0.17963294347217082
48.0	48.206230931034476	0.1763175661072812
49.0	56.70583624000001	0.20432910487310346
50.0	68.292489	0.17062298266309575
51.0	79.94475716666666	-
average	-	0.77938182482261

Table 15: The growth factor development regarding the different rings (corresponding to channels) based on a LiDAR scan with almost no obstacles in the way.

Application of the Hungarian Algorithm Following an example of the application of the *Hungarian Algorithm* is given.

	J1	J2	J3	J4	①
W1	82	83	69	92	69
W2	77	37	49	92	37
W3	11	69	5	86	5
W4	8	9	98	23	8

(a) Calculate the minimum of each row and subtract it from each row entry. The minimum is in the column with ①.

	J1	J2	J3	J4
W1	13	14	0	23
W2	40	0	12	55
W3	6	64	0	81
W4	0	1	90	15
②	0	0	0	15

(b) Calculate the minimum of each column and subtract it from each column entry. The minimum is in the row with ②.

	J1	J2	J3	J4
W1	13	14	0	8
W2	40	0	12	40
W3	6	64	0	66
W4	0	1	90	0

(c) Determine the minimum number of lines to cover all zeros of the grid. Three lines are needed.

	J1	J2	J3	J4
W1	13	14	0	8
W2	40	0	12	40
W3	**6**	64	0	66
W4	0	1	90	0

(d) Only three lines needed for four lines means a non-optimal assignment. Find the smallest uncovered number. In this example, it is the **6**.

	J1	J2	J3	J4
W1	7	8	0	2
W2	40	0	18	40
W3	0	58	0	60
W4	0	1	96	0

(e) Subtract the smallest number (6) from each uncovered entry (teal) and add it to each entry that is covered by two lines (purple).

	J1	J2	J3	J4
W1	7	8	**0**	2
W2	40	**0**	18	40
W3	**0**	58	0	60
W4	0	1	90	**0**

(f) Four lines are needed to cover all zeros means an optimal assignment is found (**orange**).

Table 16: Example for applying the *Hungarian Algorithm*. An optimal assignment is (J1, W3), (J2, W2), (J3, W1), and (J4, W4).

Traffic Sign Dataset Following, the traffic signs of the own created traffic sign dataset are illustrated. Images taken from wikipedia[1]

[1] wikipedia: https://de.wikipedia.org/wiki/Wikipedia:Hauptseite (Online; Accessed: February 21, 2024)

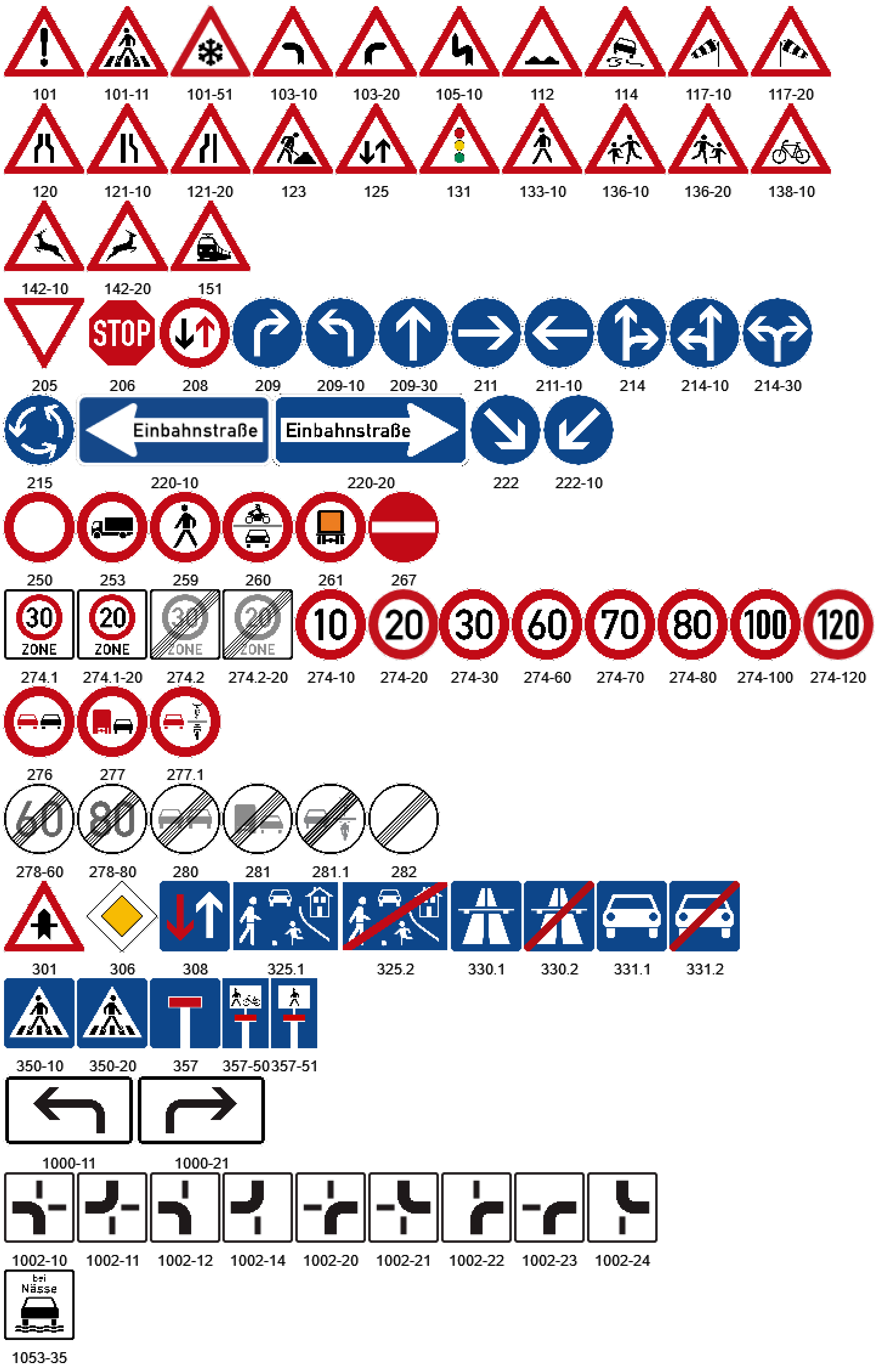


Figure 45: All Traffic Signs that are included in the dataset.

Hungarian Algorithm Implementation of the *Hungarian Algorithm* with the numpy package. The variant (2) is implemented and every not used alternative implementation is added as code to comprehend the implementation. The implementation is taken from [39] and modified by myself.

Source Code 6.2: Implementation of the tested *Hungarian Algorithm* with different, tested variants, in *Python*.

```python
import numpy as np
import math
import random
import timeit

def create_objects(n):
    objects = [(round(random.randint(0,20)),round(random.randint(0,20))) for i in range(n)]
    return objects

def min_zero_row(zero_mat, mark_zero):
    '''The function can be splitted into two steps:
    #1 The function is used to find the row which containing the fewest 0.
    #2 Select the zero number on the row, and then marked the element corresponding row and column as False
    '''

    #Find the row
    min_row = [99999, -1]

    for row_num in range(zero_mat.shape[0]):
        if np.sum(zero_mat[row_num] == True) > 0 and min_row[0] > np.sum(zero_mat[row_num] == True):
            min_row = [np.sum(zero_mat[row_num] == True), row_num]
    #bool_sum = np.sum(zero_mat, axis=1)
    #masked_bool_sum = np.ma.masked_equal(bool_sum, 0., copy=False)
    #min_row = [np.min(masked_bool_sum), np.argmin(masked_bool_sum)]

    # Marked the specific row and column as False
    zero_index = np.where(zero_mat[min_row[1]] == True)[0][0]
    mark_zero.append((min_row[1], zero_index))
    zero_mat[min_row[1], :] = False
    zero_mat[:, zero_index] = False

def mark_matrix(mat):
    '''
    Finding the returning possible solutions for LAP problem.
    '''
    #Transform the matrix to boolean matrix(0 = True, others = False)
    cur_mat = mat
    zero_bool_mat = (cur_mat == 0)
    zero_bool_mat_copy = zero_bool_mat.copy()

    #Recording possible answer positions by marked_zero
    marked_zero = []
    while (True in zero_bool_mat_copy):
        min_zero_row(zero_bool_mat_copy, marked_zero)

    #Recording the row and column positions seperately.
    marked_zero_row = []
    marked_zero_col = []
    for i in range(len(marked_zero)):
        marked_zero_row.append(marked_zero[i][0])
        marked_zero_col.append(marked_zero[i][1])

    #Step 2-2-1
    non_marked_row = list(set(range(cur_mat.shape[0])) - set(marked_zero_row))

    marked_cols = []
    check_switch = True
    while check_switch:
        check_switch = False
        for i in range(len(non_marked_row)):
            row_array = zero_bool_mat[non_marked_row[i], :]
            for j in range(row_array.shape[0]):
                #Step 2-2-2
                if row_array[j] == True and j not in marked_cols:
                    #Step 2-2-3
                    marked_cols.append(j)
                    check_switch = True
        for row_num, col_num in marked_zero:
            #Step 2-2-4
            if row_num not in non_marked_row and col_num in marked_cols:
                #Step 2-2-5
                non_marked_row.append(row_num)
                check_switch = True
    #Step 2-2-6
    marked_rows = list(set(range(mat.shape[0])) - set(non_marked_row))

    return(marked_zero, marked_rows, marked_cols)

def adjust_matrix(mat, cover_rows, cover_cols):
    cur_mat = mat
    non_zero_element = []

    #Step 4-1
    for row in range(len(cur_mat)):
        if row not in cover_rows:
            for i in range(len(cur_mat[row])):
                if i not in cover_cols:
                    non_zero_element.append(cur_mat[row][i])
    min_num = min(non_zero_element)
    #idx = tuple(np.array([[row, col] for row in range(cur_mat.shape[0]) if row not in cover_rows for col in range(cur_mat.shape[1]) if col not in cover_cols]).T)
    #min_num = min(cur_mat[idx])

    #Step 4-2
    #cur_mat[idx] -= min_num
    for row in range(len(cur_mat)):
        if row not in cover_rows:
            for i in range(len(cur_mat[row])):
                if i not in cover_cols:
                   cur_mat[row, i] -= min_num
    #Step 4-3
    #idx_gen = np.array([[row, col] for row in cover_rows for col in cover_cols])
    #cur_mat[tuple(idx_gen.T)] += min_num
    for row in range(len(cover_rows)):
        for col in range(len(cover_cols)):
            cur_mat[cover_rows[row], cover_cols[col]] += min_num
    return cur_mat

def hungarian_algorithm(mat):
    # select the smaller dimension to get an assignment result and don't run into an error
    dim = min(mat.shape[0], mat.shape[1])
    cur_mat = mat

    #Step 1 - Every column and every row subtract its internal minimum
    #for row_num in range(mat.shape[0]):
    #    cur_mat[row_num] = cur_mat[row_num] - np.min(cur_mat[row_num])

    #for col_num in range(mat.shape[1]):
    #    cur_mat[:,col_num] = cur_mat[:,col_num] - np.min(cur_mat[:,col_num])
    r_min = mat.min(axis=1)
    #cur_mat = np.subtract(cur_mat, r_min.T)
    for i in range(len(r_min)):
        cur_mat[i] -= r_min[i]

    c_min = cur_mat.min(axis=0)
    cur_mat = np.subtract(cur_mat, c_min)

    #print('Matrix row and column minimum subtracted:\n',
   cur_mat)

    zero_count = 0
    while zero_count < dim:
        #Step 2 & 3
        ans_pos, marked_rows, marked_cols = mark_matrix(
   cur_mat)
        zero_count = len(marked_rows) + len(marked_cols)

        if zero_count < dim:
            cur_mat = adjust_matrix(cur_mat, marked_rows,
   marked_cols)

    return ans_pos

def ans_calculation(mat, pos):
    total = 0
    ans_mat = np.zeros((mat.shape[0], mat.shape[1]))
    for i in range(len(pos)):
        total += mat[pos[i][0], pos[i][1]]
        ans_mat[pos[i][0], pos[i][1]] = mat[pos[i][0], pos[i
   ][1]]
    return total, ans_mat

def main():
    '''Hungarian Algorithm:
    Finding the minimum value in linear assignment problem.
    Therefore, we can find the minimum value set in net
   matrix
    by using Hungarian Algorithm. In other words, the maximum
    value
    and elements set in cost matrix are available.'''
    objects = create_objects(10)
    object_proposal = create_objects(15)

    # change objects to row and object_proposal to columns
   because objects are fixed and the object proposals should
    # be mapped to the corresponding objects
    cost_matrix = np.zeros(shape=(len(objects), len(
   object_proposal)))

    # insert the distance between object and object_proposal
    into the matrix
    for i in range(len(objects)):
        for j in range(len(object_proposal)):
            ob = objects[i]
            ob_p = object_proposal[j]
            distance = math.sqrt((ob[0]-ob_p[0])**2+(ob[1]-
    ob_p[1])**2)
            cost_matrix[i,j] = distance
    ans_pos = hungarian_algorithm(cost_matrix.copy())#Get the
    element position.
    ans, ans_mat = ans_calculation(cost_matrix, ans_pos)#Get
    the minimum or maximum value and corresponding matrix.

    return ans, ans_mat

if __name__ == '__main__':
    times = []
    for j in range(5):
        total_time = timeit.timeit(main, number=100000)
        print(f'Average time for 1000 runs {total_time/100}')
```

Calibration of LiDAR and camera In Table 17 are the manually re-calibrated parameters of the LiDAR to camera calibration.

Camera	front	left	right	rear
x	0.98	-0.68915	-0.7776	-0.37635
y	0.594	0.97608	-0.88889	0.7368
z	0.52	0.91405	0.89035	0.82795
roll	-1.52	-2.15	-2.4	-1.57
pitch	0.0125	0.0	0.215	0.09
yaw	-1.61	0.15	2.735	1.55

Table 17: Re-calibrated parameters of the camera's extrinsic parameters.

User Interface Following, figures of the implemented user interface are presented.

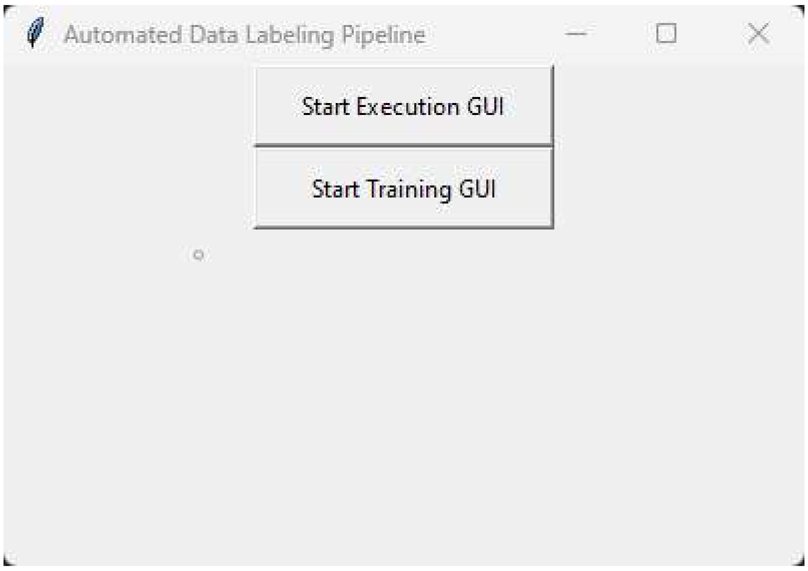


Figure 46: Start User Interface of the Automated Data Labeling Pipeline.

Read images from DTLD Code to read the images of the *DTLD* dataset. The images are stored with bayering and shift to save memory storage.

Source Code 6.3: Load an image of the *DTLD*, in *Python*.

```python
file_path = path_to_dir+'/Bochum/Bochum/Bochum1/2015-04-21_17
    -28-49/DE_BBBR667_2015-04-21_17-28-50-748255_k0.tiff'
img = cv2.imread(file_path, cv2.IMREAD_UNCHANGED)
img = cv2.cvtColor(img, cv2.COLOR_BAYER_GB2BGR)
img = np.right_shift(img, 4)
img = img.astype(np.uint8)
Image.fromarray(img).show()
```

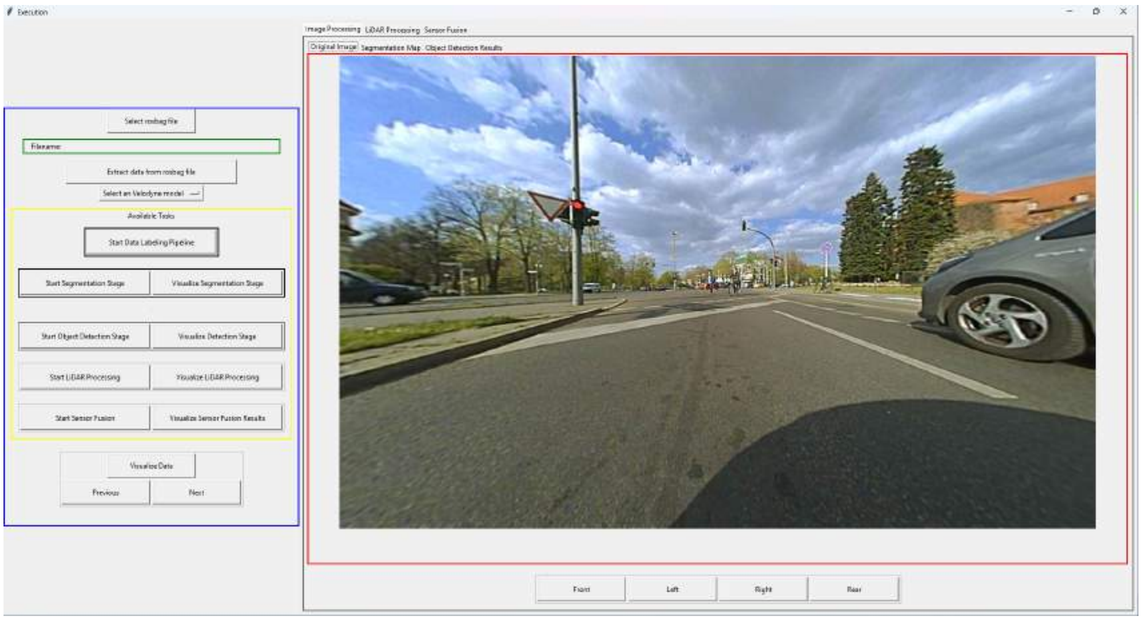
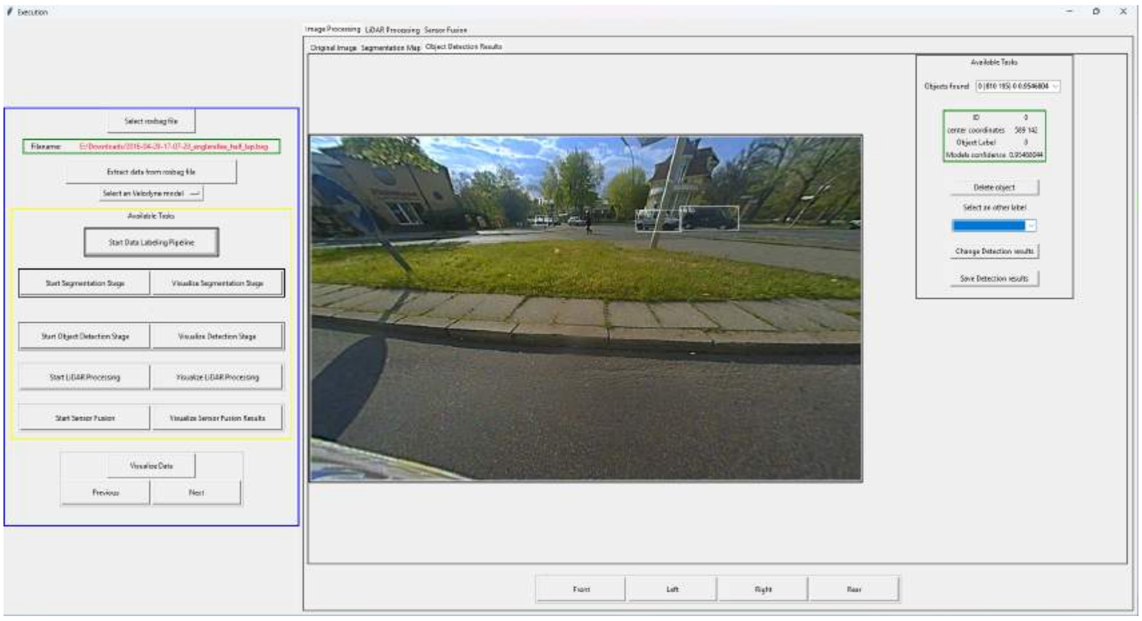


Figure 47: Default Execution User Interface. The requirements **R-1**, **R-2**, **R-3**, **R-4**, **R-5**,**R-6**, **R-7**, **R-8** are all implemented as buttons on the left side.



Filename: E:/Downloads/2016-04-29-17-07-23_englandiee_half_lap.bag

Objects found 0 |810 185| 0 0.9546804

ID	0
center coordinates	569 142
Object Label	0
Models confidence	0.95468044

Figure 48: Execution User Interface of the Object Detection Stage. The requirement **R-11** is implemented with functionality to change or remove an object detection result.

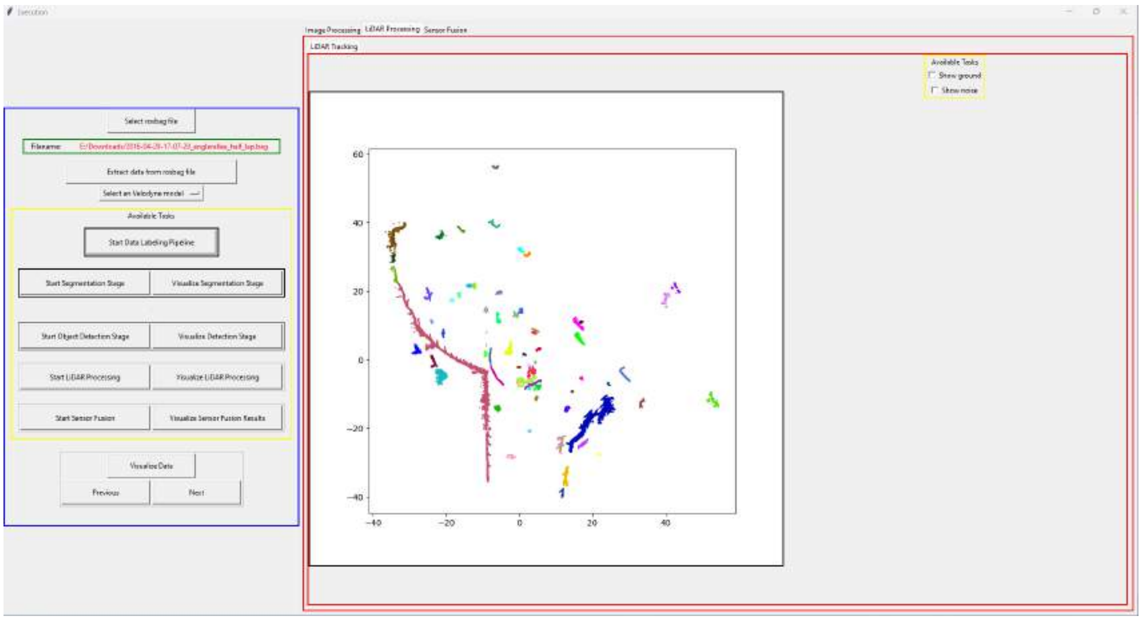


Figure 49: Execution User Interface of the LiDAR Tracking of LiDAR objects. The requirement **R-10** is implemented here.

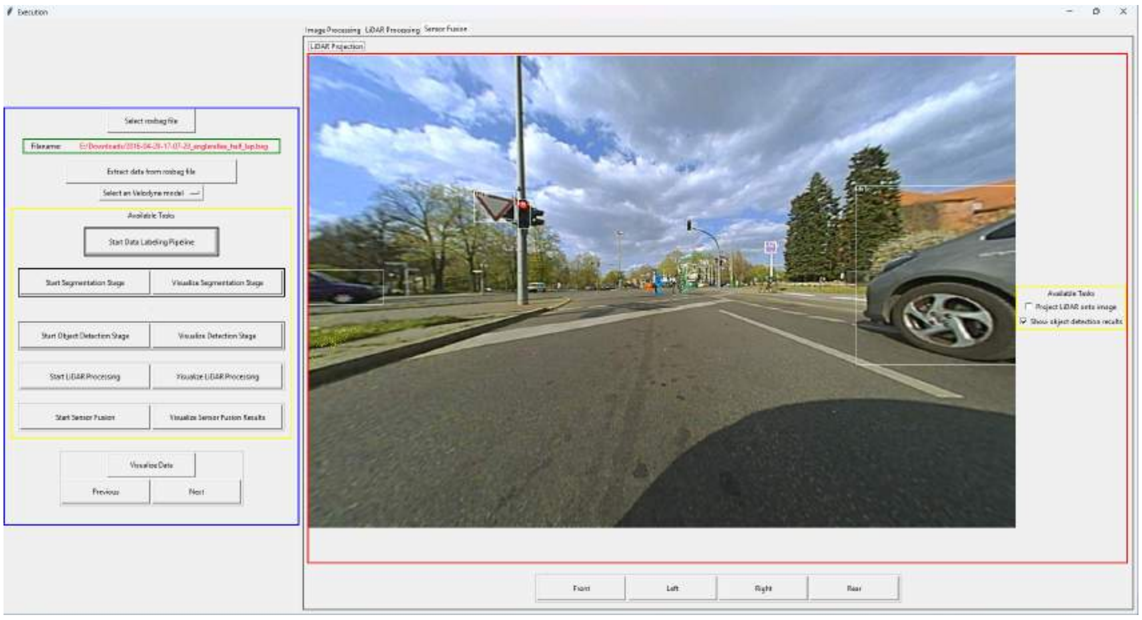


Figure 50: Execution User Interface of the Sensor Fusion visualization. The requirement **R-10** is implemented here.

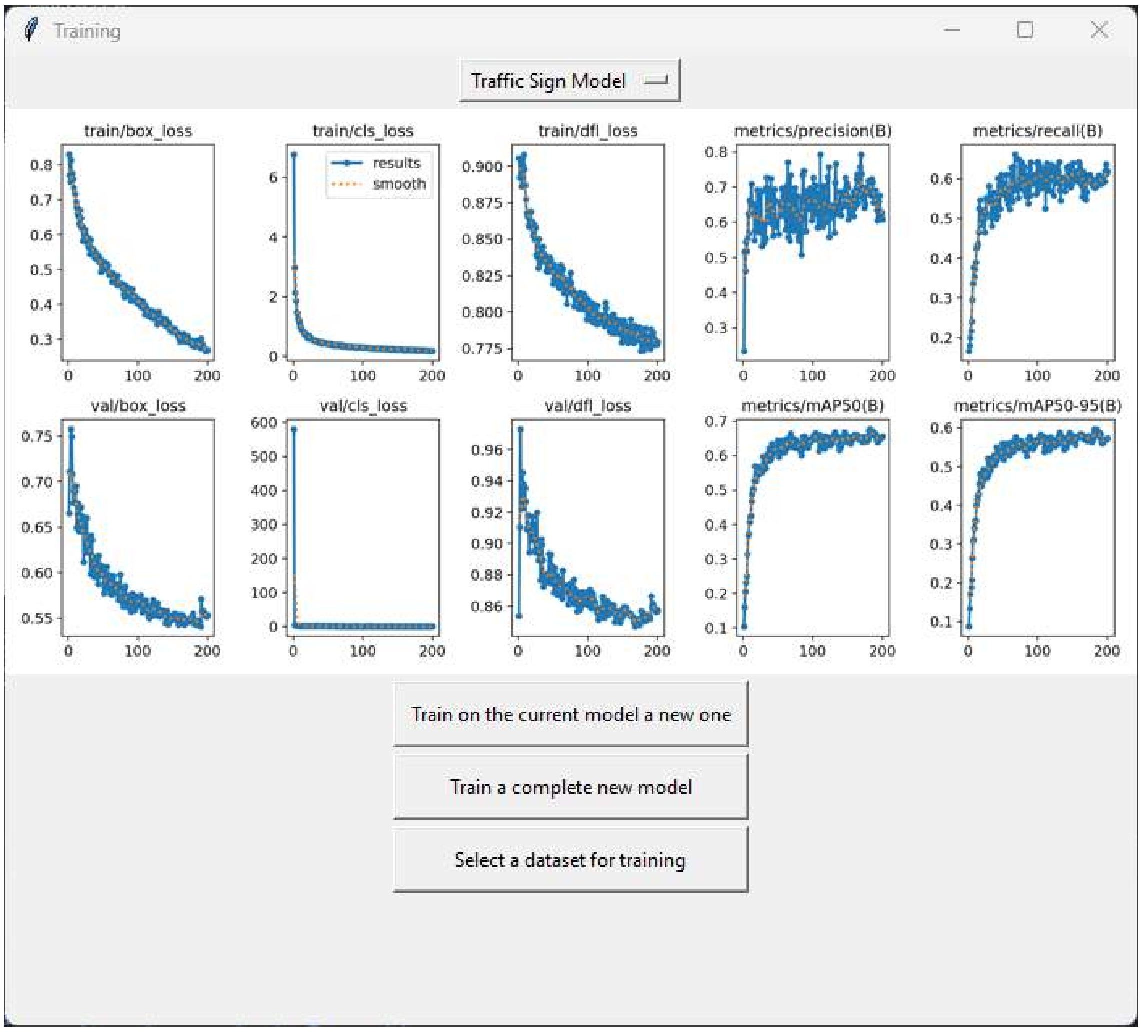


Figure 51: Training User Interface. The requirement **R-14** is implemented. Several pieces of information are shown.