Contributions to Tracking and Artificial Intelligence Based Lidar Signal Processing for Automotive Applications

Point Cloud Processing and Augmentation for AI, Optimized Vehicle and Lane Marker Tracking for Automotive

von der Fakultät für Elektrotechnik, Informationstechnik und Medientechnik
der Bergischen Universität Wuppertal genehmigte

Dissertation

---

zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften

von

M. Sc. Martin Alsfasser

aus

Remscheid

Wuppertal 2022

Tag der Prüfung	12.11.2021
Hauptreferent	Prof. Dr.-Ing. Anton Kummert
Korreferent	Prof. Dr.-Ing. Bernd Tibken

The PhD thesis can be quoted as follows:

urn:nbn:de:hbz:468-20220613-103406-4
[http://nbn-resolving.de/urn/resolver.pl?urn=urn%3Anbn%3Ade%3Ahbz%3A468-20220613-103406-4]

DOI: 10.25926/0dvn-9160
[https://doi.org/10.25926/0dvn-9160]

Abstract

The work presented here covers a variety of different algorithms and methods for advanced driver assistance systems and for the training of neural networks for automated driving. Firstly, a motivation and publication background is presented in chapter 1, before chapter 2 presents fundamental algorithms and information on techniques used in this work. Chapter 3 starts by presenting a novel method for tracking of vehicle head and tail lights. Methods are introduced to improve and stabilize the results provided by the implemented Gaussian-mixture probability hypothesis density (GM-PHD) filter. Most notably optical tracking as additional measurement confirmation, section 3.3.1, and group tracking, section 3.5. Chapter 4 continues by introducing an advanced grid structure to allow for real time object detection in lidar point clouds. This is supported by also introducing occupancy grid features, section 4.3.2, and advanced object injection augmentation, section 4.4.1. Nonetheless, training neural networks requires huge amounts of annotated training data. Chapter 5 introduces a novel method of semi-automatic data annotation, to increase the amount of available data without increasing the cost massively. The aforementioned injection augmentation is further improved in section 5.3.3, to increase the quality of the augmentation. Additionally temporal fusion by use of recurrent neural network layers is presented in section 5.3.1. Finally, chapter 6 presents an algorithm for detection, tracking and modeling of lane markers in lidar point clouds and presents possible extensions to full color point clouds created from fusing images and point clouds. Chapter 7 presents a conclusion and outlook into possible future developments.

Zusammenfassung

Die hier vorgestellte Arbeit behandelt eine Vielzahl an Algorithmen und Methoden für fortgeschrittene Fahrerassistenzsysteme und für das Training von Neuronalen Netzen zum automatisierten Fahren. Im ersten Abschnitt, Kapitel 1, wird eine Übersicht über die Motivation und vorhergegangene Publikationen als Basis dieser Arbeit gegeben, bevor Kapitel 2 eine Einführung in genutzte Grundlagen und Sensoren gibt. In Kapitel 3 wird eine neuartige Methode vorgestellt um Lichquellen von Fahrzeugen zu verfolgen. Es werden Methoden vorgestellt den verwendeten „Gaussian-mixture probability hypothesis density filter (GM-PHD)“ zu stabilisieren. Erst mit Hilfe eines optischen Suchalgorithmuses in Sektion 3.3.1, dann durch gruppenbasierte Verfolgsungsalgorithmen in Sektion 3.5. Darauffolgend wird in Kapitel 4 eine neue Gitterstruktur vorgestellt um Lidar-Punktwolken in Echtzeit verarbeiten zu können. Unterstützt wird dies von einer Verdeckungskarte in Sektion 4.3.2 und einer forgeschrittenen Objekteinsetzungsaugmentation in Sektion 4.4.1. Nichtsdestotrotz benötigen Neuronale Netze riesige Mengen annotierter Trainingsdaten. Kapitel 5 stellt eine neue Methode zur halbautomatischen Datenannotation vor um die vorhandene Menge an Trainingsdaten deutlich zu erhöhen ohne den Annotationsaufwand, und damit die Kosten, in gleichem Maße zu steigern. Die vorab vorgestellte Objekteinsetzungsaugmentation wird in Sektion 5.3.3 weiter verbessert. Außerdem wird zeitbasierte Datenfusion für die vorgestellte Anwendung in Sektion 5.3.1 evaluiert. Final wird in Kapitel 6 ein Algorithmus zur Erkennung, Verfolgung und Modellierung von Fahrbahnmarkierungen in Lidar-Punktwolken präsentiert. Eine Erweiterung auf vollfarbige Punktwolken als Kombination von Farbbildern und Punktwolken wird vorgestellt. Kapitel 7 stellt final eine Zusammenfassung und einen Ausblick vor.

Danksagung

Ich möchte all denen danken, die es mir über die letzten Jahre ermöglicht haben diese Arbeit zu verfassen.

Als erstes selbstverständlich Prof. Anton Kummert für die Betreuung dieser Arbeit über viele Jahre. Für Anleitung, Kritik, Zuspruch, für die Anregung von Ideen und jegliche sonstige Unterstützung wie auch der Finanzierung dieser Arbeit.

Genauso Prof. Bernd Tibken, für die Begutachtung und Kritik in den finalen Schritten der Arbeit.

Als nächstes Christian Nunn, Mirko Meuter, Dennis Müller und mit Ihnen Aptiv Services Deutschland GmbH für eine umfangreiche Unterstützung in industrieller Forschung. Sowohl durch anteilige Finanzierung dieser Arbeit als auch Zugang zu spannenden Projekten, modernen Architekturen und umfangreichen Datensets.

Auch ohne Lukas, Frederik, Pascal, Ido, Antonia, Lutz, Urs, Patrick, Maurice, Matthias, Maik, Philip und viele andere Freunde und Kollegen wäre all dies nicht möglich gewesen. Sie waren tägliche Unterstützung, Motivation und ein stetiger Quell neuer und spannender Ideen.

Zuguterletzt möchte ich natürlich meinen Eltern – Burghard und Katharina – und meiner Schwester Nicole für ihre bedingungslose Unterstüzung in allen Lebenslagen danken. Sie haben mir ermöglicht diesen Weg zu gehen und die Steine auf ebendiesem nach allen Möglichkeiten beiseite geräumt, sodass ich mich voll auf meine Ausbildung und Arbeit fokussieren konnte.

Contents

1 Introduction

1.1 Motivation

Modern advanced driver assistance systems, or even fully autonomous cars, require more and more complex and advanced algorithms to detect their surroundings. They need to cover object detection in a variety of different circumstances, tracking discovered objects over time and processing the information generated from these algorithms.

This work shows both, a look at vehicle light detection and tracking for Advanced Driver Assistance Systems (ADAS) but also technologies for 3D object detection for fully or partially autonomous vehicles, including lane markings and other road users.

First to talk about vehicle light detection and tracking. This is important for a system called Advanced Headlight Control (AHC). Modern Light Emitting Diode (LED) headlights are often capable to dim the high beam only in certain angular ranges, without compromising the safety of the driver by completely switching the high beam off, limiting road illumination and leading to a sudden change in brightness. Detection of other vehicles needs to be performed at night and at long distances, therefore one way to do this is by detecting and tracking their head and tail lights. These detections have to be extremely stable, otherwise the high beam would constantly flicker on and off, which would be irritating to both the driver of the ego vehicle and also the drivers of other vehicles. In this work multiple techniques will be shown to ensure stable tracking of vehicle lights, often in challenging situations, by combining optical tracking, detection based tracking and group tracking in a novel way.

Continuing with partially of fully autonomous vehicles, more than AHC is required. Autonomous vehicles need to be aware of their full surroundings, including precise detections of objects on or near the road. These objects might be cars, trucks, pedestrians, bikes or a multitude of different objects which might be relevant to autonomous driving. Nonetheless limited computational power is available. Another large part of this work is therefore dedicated to speeding up state of the art

object detection algorithms by breaking up conventional structures and introducing innovative design.

For training neural networks a large amount of annotated training data is required. This data can range from black and white camera images, over color images up to large point clouds recorded by lidar sensors. Annotating data can be a very time consuming process and therefore be very expensive, which is why another part of this work is focused on creating an algorithm to support human annotators in creating training data. This leads to shorter and cheaper data generation, therefore more data can be annotated, leading to better training results for the algorithms that are supposed to run in an autonomous vehicle.

Finally an algorithm is presented with which lane markings can be detected in lidar point clouds. This is a useful extension to classical, camera based lane marker detection. In camera images problems can occur if the lane marking is occluded by bloom, which can easily happen on sunny days.

1.2 Structure

This work is divided into 6 parts. Chapter 2 will provide a detailed insight into the fundamentals of technologies and methods used in this work going from fundamentals of tracking algorithms in section 2.1 over object detection algorithms in section 2.2 up to sensor details in section 2.3. These should provide the reader with the basics that will be omitted from later chapters in favour of pacing for the experienced reader.

The first major chapter of this work is chapter 3, talking about improvements and adjustments made to the probability hypothesis density filter (PHD) filter, to work well for vehicle light tracking. This consists of an introduction, section 3.1, of challenges and customer requirements for this task. Section 3.2 will first present the Gaussian-mixture probability hypothesis density filter (GM-PHD) introduced by Vo and Ma, 2006 before going into the modifications made for the specific light source tracking in section 3.2.2. One major problem with pulse width modulation (PWM) used for powering LED lights, for vehicle light tracking, is introduced and solved in section 3.3, before section 3.4.1 combines all these ideas for a finalized tracker pipeline. Finally section 3.5 introduces the option of group tracking to further optimize the tracker for most relevant situations.

The next chapter, chapter 4, shows possible modifications to massively reduce memory and runtime requirements for object detection algorithms on lidar point clouds. A short summary of recent state of the art algorithms is presented in section 4.1, followed by a discussion on data grid structures, their advantages and weaknesses, in section 4.2. 4.3 will introduce additional, hand crafted features to support the network in achieving state of the art performance, at a fraction of the hardware requirements. Another very important aspect of training a neural network is training data augmentation. A novel method of injecting objects into point clouds is presented in section 4.4. While others use a similar method, the method shown here ensures validity of the point cloud and provides higher quality augmented data.

Still, even with data augmentation, the amount of training data required is expensive and time consuming to produce. To support this effort, a tool, or specialised algorithm, is presented in chapter 5. The network presented in section 5.2 is a combination of other published network and used as a basis for multiple novelties. Mainly the use of data from multiple time steps at the same time, section 5.3.1, and patch wise data processing, section 5.3.2. While publications like Luo et al., 2018 exist and already use data from multiple time steps they cannot use data from the future, as will be here. Since the annotation tool is processing recorded sequences offline, it is possible to integrate data from both, past and future. Finally, the data augmentation shown previously in section 4.4 is further improved in section 5.3.3 to achieve even more accurate augmented point clouds. A large amount of result evaluation is provided in section 5.4, where multiple new implementations are evaluated against baselines to prove major benefits.

Finally, in chapter 6 a novel system for lane marker detection in lidar point clouds is presented and possible extensions to it are discussed. Section 6.1 goes into plenty detail of the different stages, data pre-processing, lane marker detection, lane marker clustering and tracking, including the use of the auction algorithm for assigning new detections to old tracks. An extension of the algorithm towards the use of RGB point clouds is discussed in section 6.2.

Please note, that due to the very large number of different concepts, methods and algorithms in this thesis, formulations, variable naming and others can not be transferred across chapters, unless specifically referenced (for example to a part of the fundamentals).

As for notation, vectors are usually printed **bold**, unless part of a general definition, where they might be vectors but could also be scalar values. Matrices are printed as capital letters.

1.3 Publication Basis for This Work

This work is based on both published work and previously unpublished work. The tracking algorithm presented in chapter 3 is previously published by Alsfasser et al., 2019 and presented at the Intelligent Vehicles Symposium 2019. The content in this dissertation is much extended with more detail shared on all systems of the tracker. More input on the theory of the GM-PHD is given, the inclusion of the Thikunov regularization is further explained and optical tracker components like the pyramidial approach and forward-backward checking are extended. Furthermore detail is extended on the choice of clustering approach and group tracking method.

The lidar object detector from chapter 4 is previously published under Alsfasser et al., 2020, at the International Conference on Machine Vision 2019. In this chapter most components of the publication are extended, much more focus given to the advantages of using spherical grids and the reasoning behind these. Several components are explained in much more detail, especially in terms of data augmentation and training, which was just a short paragraph in the original publication. A patent application for this methodology was filed at the german patent offices in 2019 but not yet granted at time of writing.

Chapter 5 is not yet published at time of writing, but will be afterwards, likely in two separate publications. The first focused on the overall system of semi automatic data annotation. Data augmentation with structure aware point cloud augmentation (SAPCA) will be published as part of Hasecke et al., 2022.

The work shown in the final chapter, chapter 6 was not previously published at large scale, but is comprised of the masters thesis Alsfasser, 2017. By nature this is less about extending the previous publication, but making it more widely available and going into a bit more detail on possible future work on the system.

2 Fundamentals

While it is assumed that the reader is experienced in signal processing, the topics of this dissertation are so widespread that a short introduction is still given on the topics which will later be extended in more detail. The first part of this chapter will talk about the different kinds of tracking filters, starting on single target tracking in section 2.1.1, before evolving into a multi target tracking method in section 2.1.2.

Next, an introduction into object detection algorithms, both classical (section 2.2.1) and neural network based (section 2.2.2), will be provided before finally talking about sensor modalities and functionality of cameras and lidar sensors in section 2.3.

2.1 Basics of Tracking Algorithms

Tracking algorithms are filter algorithms, used for processing possibly noisy input data into target states with known uncertainties.

2.1.1 Single Target Tracking

The first type of tracking algorithms covered are single target trackers. These are trackers used to track a single object at each time, either taking one measurement per object or multiple. If more than one object is being tracked at the same time, multiple instances of these algorithms have to be used with no or limited interaction between them.

Kalman Filter

One of the most well known single target tracking methods is the Kalman filter, published by R.E. Kalman in 1960 (Kalman, 1960). The Kalman filter is a linear estimation algorithm that can be used to extract a system state approximation from noisy measurement series's, assuming the noise can be assumed to be zero mean

Gaussian. It consists of a two step process where first the state evolution is predicted before a measurement is used to refine the prediction and create the new state vector.

The standard Kalman filter is calculated at discretized time steps k, with the state estimate $\hat{x}_{k|k}$, meaning the state estimate at time k under the inclusion of all state estimates and updates up to time k. Additionally the estimated state variance is given by the covariance matrix $P_{k|k}$.

The following description and the given equations are modified from Kim and Bang, 2018 to follow the notation later used in chapter 6. In the first part of the prediction step the a priori, meaning before measurement update, state prediction is given by

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k, \tag{2.1}$$

with F_k being the state prediction matrix, or the expected state change from $k-1$ to k without external input, $\hat{x}_{k-1|k-1}$ being the a posteriori, or after measurement update, state at the last timestep, B_k is the control input model and u_k the control input. The a priori covariance $P_{k|k-1}$ is calculated very similarly as

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k, \tag{2.2}$$

where Q_k is the covariance of the process noise, or the uncertainty that builds up with every prediction. The second part of the Kalman filter is the update step, which uses noisy measurements z_k to calculate updates to the a priori predictions $\hat{x}_{k|k-1}$ and $P_{k|k-1}$. Both, measurement innovation $\tilde{y}_k$ and innovation covariance S_k are calculated similarly as

$$\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1} \tag{2.3}$$

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{2.4}$$

using the observation model H_k to translate measurements into the desired state domain. The observation model, state covariance and innovation covariance are now used to calculate the optimal Kalman gain

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \tag{2.5}$$

which weights the influence of the measurement innovation against the a priori state prediction. Finally both the a posteriori state prediction and the a posteriori state covariance are calculated as

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \tag{2.6}$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}, \tag{2.7}$$

resulting in the final update for time step k. The Kalman filter allows for great flexibility and adjustment to levels of measurement noise, process noise and required state adaptability. Furthermore using more than one update step/pass allows for the use of multiple observations for each predicted state. Nonetheless the Kalman filter requires explicitly assigning measurement to state predictions to work, which can often be difficult or imprecise. The Kalman filter, as shown, only allows for prediction of linear systems. Extensions are made with both, the extended Kalman filter (EKF) and also the unscented Kalman filter (UKF), which both are specifically designed, by linearizing (EKF) or sampling (UKF), for nonlinear systems. As neither of those extensions are used in this work, not more detail shall be given here.

Particle Filter

Monte Carlo Algorithms are algorithms in which randomly sampled values from distributions are sampled and processed by an algorithm to evaluate or simulate how the algorithm performs on inputs in the probable input range. They can be used for simulation, filtering or other methods of data evaluation. Because of the random method of input sampling, a large number of passes can be required to achieve the desired variety and confidence in the simulation or filter. This can make processing very computationally intensive.

Particle Filters are a type of sequential Monte Carlo algorithms. In general the idea is similar to the Kalman filter in a way such as it is a multi stage approach including state prediction and state correction or update. By sampling a number of particles from a known proposal distribution g and weighting them according to their state difference to the target distribution f, this target distribution can be approximated iteratively – this is called importance sampling (Thrun et al., 2005, p. 80-82). Other than the default Kalman filter, this can be used to approximate nonlinear and non Gaussian distributions. If not noted differently, equations in this paragraph are taken from Thrun et al., 2005, p. 77-82 or adapted from equations from them.

As initialization a set of N particles is sampled from the proposal distribution

$$\chi_k \coloneqq [x_k^{(1)}, x_k^{(2)}, x_k^{(3)}, ..., x_k^{(N)}] \tag{2.8}$$

$$x_k^{(i)} \sim g, \quad i \in N, \tag{2.9}$$

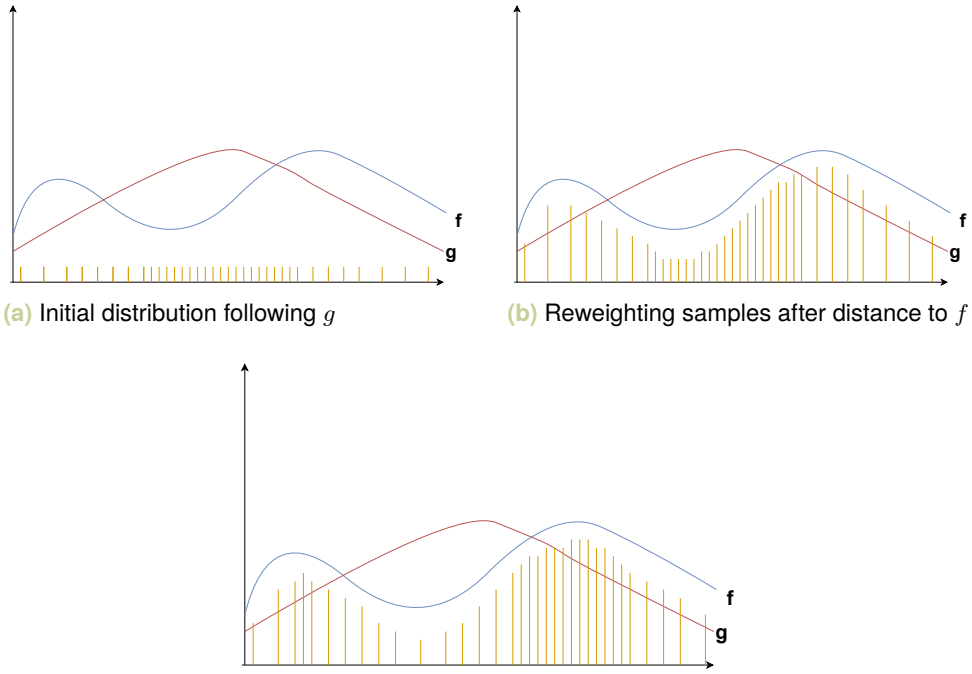
with the weight $w_k^{(i)}$ for each sample $x_k^{(i)}$ being estimated from the probability of a measurement under the assumption $x_k^{(i)}$ belonging to the target distribution

$$w_k^{(i)} = p(z_k, x_k^{(i)}), \quad i \in N, \tag{2.10}$$

with z_k being the measurement at time k and p a function to estimate this probability. Same as the Kalman filter, the state estimates, or particles, of a particle filter can be predicted into the next time step $k+1$ without measurements. Thrun et al., 2005, p. 79 propose to perform this prediction by sampling, alternatively the old particles can be moved as in the Kalman filter, by using a state transition model. The 'update' step is then performed by equation 2.10, weighting the samples with their distance to the measured state z_k. One issue that could arise if this is performed iteratively is the constant reduction of all weights. Unless a sample fits the target distribution perfectly at each new time step, its weight would degrade over time. Additionally more and more processing time would be attributed to particles of low importance, as they move away from the target distribution. To solve this issue, Thrun et al., 2005, p. 79-83 and others suggest resampling of weight, meaning after a number of processing iterations, or even every iteration, instead of reusing particles, new particles are sampled from the distribution represented by χ_k and the particle weights, refocusing the particles around areas of high importance or weight. Over time this way the importance weighted distribution of particles converges to the target distribution f

$$\left[\sum_{i=1}^{N} w^i\right]^{-1} \sum_{i=1}^{N} I(x^{(i)} \in A) w^{(i)} \rightarrow \int_A f(x)dx, \tag{2.11}$$

with A being the distribution spanned by χ. An example of the overall progress of a particle filter is shown in figure 2.1, displaying the initial particle sampling from g, the weighing according to the weights calculated in equation 2.10, up to the resampling, after each filter step. For the application presented in chapter 3 the particle filter could be a decent fit, although possibly large amounts of independent targets and target distributions would require a large amount of particles for each target to achieve good precision. Furthermore the particle filter is a single target filter/tracker, requiring separate processing for each target, loosing out on the possibility of each tracker utilizing global optimization over multiple tracks.



(a) Initial distribution following g

(b) Reweighting samples after distance to f

(c) Resampling following the distribution build from weighted samples

Figure 2.1.: Example progress of a particle filter, from initial sampling, over reweighting up to resampling. Samples in orange, length = likelihood/closeness to target, density = estimated density of target distribution.

Optical Tracking

A different kind of tracker, compared to Kalman and particle filters, is the optical tracker. One of, if not the, most well known optical tracker is the Lucas-Kanade tracker (Lucas and Kanade, 1981). The general principle of the Lucas-Kanade tracker is taking a cut out from one frame k, later called a template, and iteratively search for the affine image transformation to transfer that template T into the next frame, image I. As shown by Baker and Matthews, 2004, there are multiple ways to calculate this transformation, all with their separate benefits and drawbacks. The following calculations are, if not noted otherwise, taken from Baker and Matthews, 2004.

The basic Lucas-Kanade tracker works by minimizing the error between the warped template and the next image frame

$$\sum_{\mathbf{x}}[I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x})]^2] \tag{2.12}$$

with **x** as positions in the image, W the warp function and **p** the parameters of the warp. The assumption is, that **p** can be calculated iteratively from a baseline

$$\sum_{\mathbf{x}}[I(W(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}))-T(\mathbf{x})]^2]. \tag{2.13}$$

To minimize equation 2.13, it needs to be linearized as in

$$\sum_{\mathbf{x}}\left[I(W(\mathbf{x};\mathbf{p}))+\nabla I\frac{\partial W}{\partial \mathbf{p}}\Delta\mathbf{p}-T(\mathbf{x})\right], \tag{2.14}$$

leading to $\Delta\mathbf{p}$ as

$$\Delta\mathbf{p}=H^{-1}\sum_{\mathbf{x}}\left[\nabla I\frac{\partial W}{\partial \mathbf{p}}\right]^T[T(\mathbf{x})-I(W(\mathbf{x};\mathbf{p}))], \tag{2.15}$$

with

$$H=\sum_{\mathbf{x}}\left[\nabla I\frac{\partial W}{\partial \mathbf{p}}\right]^T\left[\nabla I\frac{\partial W}{\partial \mathbf{p}}\right]. \tag{2.16}$$

As can be seen in equation 2.15, each iterative step of the basic Lucas-Kanade algorithm requires the calculation of a Hessian matrix, and, as I is warped with **p** and the Hessian equation 2.16 is dependant on I, the 2-dimensional image gradient ∇I and, in some cases, the Jacobian $\frac{\partial W}{\partial \mathbf{p}}$ for every iteration step, this can be very time consuming. Baker and Matthews, 2004 provide the computational cost of one iteration at $O(n^2N+n^3)$. To solve this, they describe the "Inverse Compositional" Lucas-Kande, in which they switch image I and template T computationally, allowing to pre-compute the expensive Hessian.

The target now becomes minimizing

$$\sum_{\mathbf{x}}[T(W(\mathbf{x},\Delta\mathbf{p}))-I(W(\mathbf{x},\mathbf{p}))]^2, \tag{2.17}$$

in which each iterative $\Delta\mathbf{p}$ estimates the required warp compared to the image being warped with the **p** accumulated over the last iterations. Linearized this leads to

$$\sum_{\mathbf{x}}\left[T(W(\mathbf{x},0))+\nabla T\frac{\partial W}{\partial \mathbf{p}}\Delta p-I(W(\mathbf{x};\mathbf{p}))\right]^2, \tag{2.18}$$

leading to the iteration step

$$\Delta\mathbf{p}=H^{-1}\sum_{\mathbf{x}}\left[\nabla T\frac{\partial W}{\partial \mathbf{p}}\right]^T[I(W(\mathbf{x};\mathbf{p}))-T(\mathbf{x})], \tag{2.19}$$

with the Hessian

$$H = \sum_{\mathbf{x}} \left[\nabla T \frac{\partial W}{\partial \mathbf{p}} \right]^T \left[\nabla T \frac{\partial W}{\partial \mathbf{p}} \right]. \tag{2.20}$$

The advantage here is, that the Hessian in equation 2.20 is dependant on ∇T, but as T is only warped with $W(\mathbf{x}; 0)$ and does not change with each iteration, this can be pre-calculated once for each performed match. Baker and Matthews, 2004 proof that the final result is equal to the basic Lucas-Kanade, but reduces the complexity to $O(n^2 N)$ for pre-computation and afterwards only $O(nN + n^2)$ per iteration step, substantially reducing each steps complexity. Baker and Matthews, 2004 provide more details on all of this and also include other alternatives for solving the Lucas-Kanade, but as this is the one later used in chapter 3, this shall be the only one mentioned here.

2.1.2 Multi Target Tracking

All of the trackers described in section 2.1.1 are single target trackers. Meaning they track one object at a time, with 1 measurement or observation as input. The tracker shown here is a multi target multi measurement tracker. Here the PHD filter, which is what is used in chapter 3, will be presented. The PHD filter is a true, multi target tracking filter, meaning that multiple tracks updated with, possibly, multiple measurements per track are predicted in a combined volume of tracks.

PHD Filter

The probability hypothesis density (PHD) filter is an approximate solution of a multi-target Bayes filter recursion. The following explanation is taken from Vo and Ma, 2006, who also provide more detail than could be fit here. It evolves from the assumption, that for a single target tracking problem the assumed state x_{k-1}, which is part of the state space $\mathbb{R}^{n_x}$, transitions from time $k-1$ to k with probability density

$$f_{k|k-1}(x_k|x_{k-1}), \tag{2.21}$$

if $f_{k|k-1}$ describes the transition density. Assuming a measurement z_k, they call the probability density of x_k receiving the observation z_k

$$g_k(z_k|x_k). \tag{2.22}$$

A very important assumption is made as

$$p_k(x_k|z_{1:k}), \tag{2.23}$$

the posterior density, describing the probability density of state x_k under all previous observations. Given an initial density p_0, this posterior density can be calculated following the Bayes recursion

$$p_{k|k-1}(x_k|z_{1:k-1}) = \int f_{k|k-1}(x_k|x)p_{k-1}(x|z_{1:k-1})dx \tag{2.24}$$

$$p_k(x_k|z_{1:k}) = \frac{g_k(z_k|x_k)p_{k|k-1}(x_k|z_{1:k-1})}{\int g_k(z_k|x)p_{k|k-1}(x|z_{1:k-1})dx}. \tag{2.25}$$

For a multi-target situation, it is assumed $M(k)$ is the number of states at the given time step, and $N(k)$ the amount of observations. It is not known which observation belongs to which state and if it belongs to any state at all. They define the random finite sets (RFS)

$$X_k = x_{k,1}, ..., x_{k,M(k)} \in \mathcal{F}(\mathcal{X}) \tag{2.26}$$

$$Z_k = z_{k,1}, ..., z_{k,N(k)} \in \mathcal{F}(\mathcal{Z}), \tag{2.27}$$

with $\mathcal{F}(\mathcal{X})$ and $\mathcal{F}(\mathcal{Z})$ describing all possible subsets of $\mathcal{X}$ and $\mathcal{Z}$. They treat X_k and Z_k not as sets, but as multi-target state respective observation. As details can be taken from the aforementioned publication, not every assumption shall be explained here, but it is explained that

$$X_k = \left[\bigcup_{\zeta \in X_{k-1}} S_{k|k-1}(\zeta)\right] \cup \left[\bigcup_{\zeta \in X_{k-1}} B_{k|k-1}(\zeta)\right] \cup \Gamma_k, \tag{2.28}$$

with

$$S_{k|k-1}(\zeta) = \text{RFS of states transitioned from previous states } \zeta \tag{2.29}$$

$$B_{k|k-1}(\zeta) = \text{RFS of states spawned from previous states } \zeta \tag{2.30}$$

$$\Gamma_k = \text{RFS of new births.} \tag{2.31}$$

After defining X_k, Vo and Ma, 2006 focus on Z_k, the RFS of observations. They assume that each tracking target x_k is detected with probability $p_{D,k}(x_k)$. Furthermore the same assumption about the probability density of $g_k(z_k|x_k)$ is valid, leading to each target x_k having its own RFS

$$\Theta_k(x_k), \tag{2.32}$$

which is z_k if detected or $\emptyset$ if not. Given a set K_k of false measurements, the RFS Z_k defines as

$$Z_k = K_k \cup \left[\bigcup_{x \in X_k} \Theta_k(x) \right]. \tag{2.33}$$

f and g now being defined as multi-target transition density respectively likelihood leads to the multi-target posterior as

$$p_{k|k-1}(X_k, Z_{1:k-1}) = \int f_{k|k-1}(X_k|X) p_{k-1}(X|Z_{1:k-1}) \mu_s(dX) \tag{2.34}$$

$$p_k(X_k|Z_{1:k}) = \frac{g_k(Z_k|X_k) p_{k|k-1}(X_k|Z_{1:k-1})}{\int g_k(Z_k|X) p_{k|k-1}(X|Z_{1:k-1}) \mu_s(dX)}. \tag{2.35}$$

Vo and Ma, 2006 describe this as being a solution, but one that is computationally intractable. Monte Carlo solutions are possible, but extremely expensive. Therefore they present the PHD approximation as a solution under some assumptions, by not utilizing the posterior density but rather the posterior intensity. They describe that the intensity, or first order moment, ν describes a non-negative function on $\mathcal{X}$ for each region $S \subseteq \mathcal{X}$

$$\int |X \cap S| P(dX) = \int_S \nu(x) dx, \tag{2.36}$$

meaning that the integral of ν provides the expected number of elements of X in S. This way local maxima of ν can be used to describe elements of X. Following several assumptions, this results in the transformation of the multi target posterior density recursion shown in equation 2.35 to the PHD recursion

$$\nu_{k|k-1} = \int p_{S,k}(\zeta) f_{k|k-1}(x|\zeta) \nu_{k-1}(\zeta) d\zeta + \int \beta_{k|k-1}(x|\zeta) \nu_{k-1}(\zeta) d\zeta + \gamma_k(x) \tag{2.37}$$

$$\nu_k(x) = [1 - p_{D,k}(x)] \nu_{k|k-1}(x) + \sum_{z \in Z_k} \frac{p_{D,k}(x) g_k(z|x) \nu_{k|k-1}(x)}{\kappa_k(z) + \int p_{D,k}(\xi) g_k(z|\xi) \nu_{k|k-1}(\xi) d\xi}, \tag{2.38}$$

with

$$\gamma_k = \text{intensity of birth RFS } \Gamma_s \text{ at time k} \tag{2.39}$$

$$\beta_{k|k-1} = \text{intensity of the spawn RFS } B_{k|k-1} \tag{2.40}$$

$$p_{S,k(\zeta)} = \text{probability of target } \zeta \text{ still existing at time k} \tag{2.41}$$

$$p_{D,k}(x) = \text{probability of detection} \tag{2.42}$$

$$\kappa_k = \text{intensity of the clutter RFS } K_k. \tag{2.43}$$

Again, all of this is taken from Vo and Ma, 2006, who propose a closed form solution to this in the form of the Gaussian-mixture probability hypothesis density (GM-PHD) filter, which will be explained in more detail, and used, in section 3.2.1.

2.2 Object Detection Algorithms

Object detection algorithms describe methods to detect the occurrence, position, size and rotation of different, usually pre-defined, objects inside a sensor readout. Classically these sensor readouts are mostly camera images or radar scans, but over the last few years lidar sensors like the Velodyne-HDL64 (Velodyne Lidar Inc., 2021a) got more common, especially in research for autonomous vehicles. This section is about presenting both, classical, non neural network based, approaches in section 2.2.1 and also neural network algorithms in section 2.2.2, mostly focused on images, but especially looking at neural network algorithms, the general network structure is often similar between different types of sensors.

2.2.1 Classic Approaches

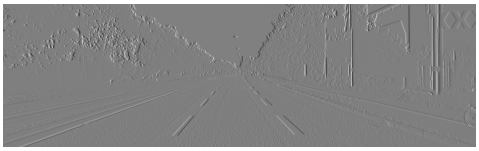
Before the onslaught of neural network based machine learning over the last decade or two there was already a plethora of classical approaches, either based on specific models or early forms of machine learning. This section should give a short overview, although the field is too wide to provide a complete overview. Therefore the general setup of a lot of these algorithms and two examples shall be mentioned.

Gradient Feature Extraction

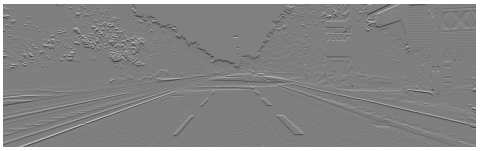
The most fundamental features often used are image gradients. In the most basic way they can be calculated separately in x- and y-direction by multiplying the image with simple vectors $K_x = [1\ 0\ -1]$ and $K_y = [1\ 0\ -1]^T$, generating gradient images highlighting gradients in each direction separately. Extending these is possible with a filter kernel similar to

$$K_x = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} \quad K_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}, \tag{2.44}$$

(a) Example image from Geiger et al., 2013



(b) Gradients in x-direction, kernel K_x



(c) Gradients in y-direction, kernel K_y

Figure 2.2.: Gradient images as calculated with filter kernels K_x and K_y

which calculate gradients over a larger area in the specified direction. A modification of this approach is later used in chapter 6, where the kernel size is further increased to 7x9 – with a similar pattern – to smooth gradients further. This allows to reduce variance, as the resulting gradient value is collected over a larger search window, smoothing out noise. A further variation of this is is the Sobel operator (Sobel and Feldman, 1973)

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \tag{2.45}$$

which introduces weighting into the gradient calculation. The gradient images can then be used in a multitude of ways to detect objects. Examples of gradient images calculated from base image figure 2.2a can be seen in figure 2.2b/figure 2.2c. The base image is taken from the raw Kitti data (Geiger et al., 2013). It can be clearly seen, how gradient images like these can be utilized to extract information from images, as they make object boundaries, and by extend type, more clear. These gradient images can then be further processed to extract this information.

Hough Transformation for Geometric Object Detection

The Hough transformation (Hough, 1962) is a method for extracting geometric objects like lines or circles from black and white images. The method is well

described by Duda and Hart, 1972, explaining how for each data point in the image a large number of possible lines

$$\rho = x \cos\theta + y \sin\theta, \quad \theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \tag{2.46}$$

is sampled and plotted in a feature plane of ρ and θ. Lines that pass through multiple data points will create highlights in the $\rho - \theta$ feature plane, shown in figure 2.3c, which can then be extracted. An example of possible results is shown in figure 2.3b, design and implementation taken from The MathWorks Inc., 2021c. The same principle can be, as shown in Duda and Hart, 1972, used for other geometric shapes like circles. This is a non-learning approach which can unfortunately be very time consuming if the sampling resolution or the amount of data points is high.

Histogram of oriented gradients

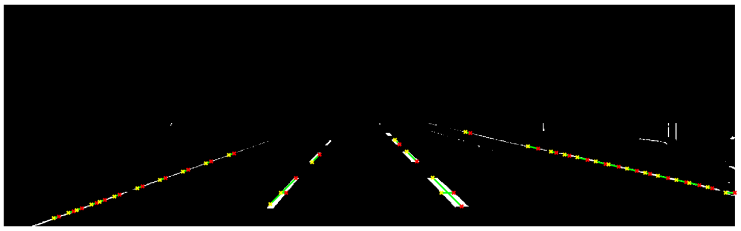
Compared to the Hough transformation, the Histogram of Oriented Gradients (HOG) is a more advanced object detection or rather classification approach first applied to human detection by Dalal and Triggs, 2005. For using the HOG, first gradient images in both, x- and y-direction need to be calculated as shown in section 2.2.1. From these gradient images both the magnitude and the angle of the gradient are calculated as

$$g = \sqrt{g_x^2 + g_y^2} \tag{2.47}$$

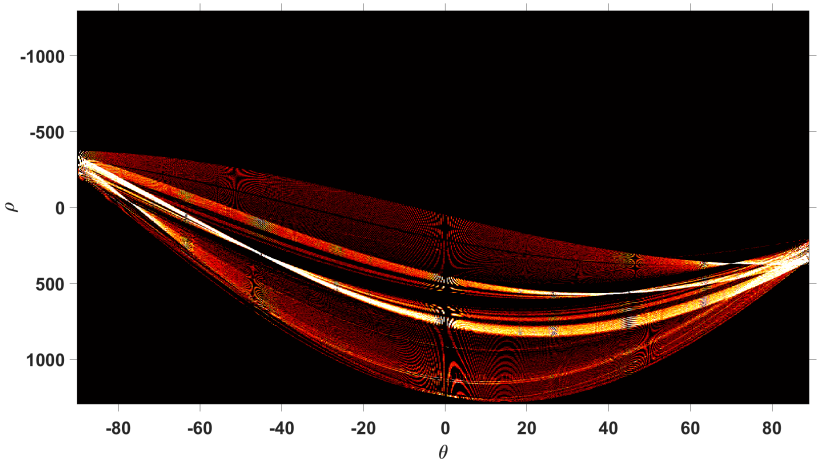
$$\theta = \arctan\left(\frac{g_y}{g_x}\right). \tag{2.48}$$

The image is subdivided into smaller regions, for example 8x8 pixels, and inside each region histograms of gradient orientations θ are build. Usually 9 bins of 20° each are used, as the signs of the orientations are ignored and therefore only 180° have to be covered. Inside each bin the gradient values g are summed up, and in case gradients lie between two orientation bins the values are split and a part is added to each bin. To make the detector more invariant to the absolute brightness of the source image, the histograms are normalized. The normalization can be performed inside each 8x8 cell, but most publications propose normalization regions that contain multiple, for example 2x2, of these cells to smooth out the normalization. These normalization regions can overlap and for each 8x8 cell, the normalized feature vector, for a 2x2 normalization area this would be 36x1 – 4 histograms of 9 bins –, is saved. Finally, all of the feature vectors can be concatenated to a large, global feature vector to be processed further, for example with a Support Vector Machine

(a) Black and white image, thresholded from figure 2.2a



(b) Lines extracted with Hough transformation, as by The MathWorks Inc., 2021c



(c) Hough space created as by The MathWorks Inc., 2021b

Figure 2.3.: Base image, Hough feature space and final result for a Hough transformation on a black and white image.

(SVM) as shown in the next paragraph. A visualization of the intermittent features before block normalization is shown in figure 2.4. Each small set of lines describes the gradients and directions at each pixel, the larger, thicker lines the gradients and directions in an 8x8 pixel bin.

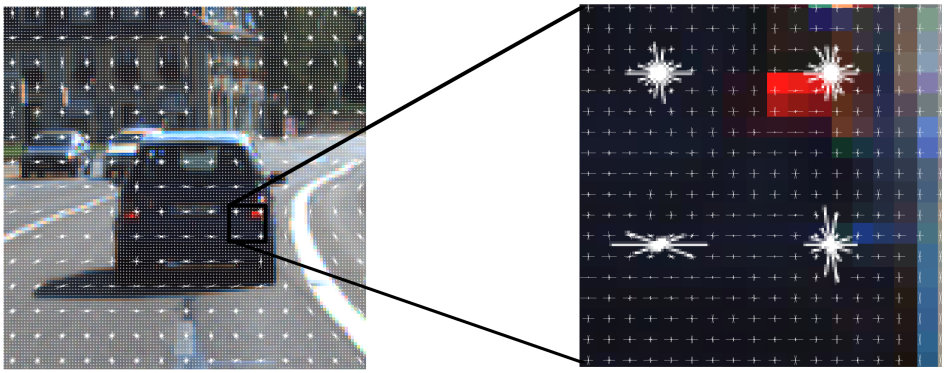


Figure 2.4.: Visualization of HOG features, taken from The MathWorks Inc., 2021a, each set of lines displaying intensity and orientations of each pixel and each bin in a 8x8 area. Base image from Geiger et al., 2013

Support Vector Machines for Feature Classification

A Support Vector Machine is a method to separate data with an (optimal) hyperplane according to their class, either 1 or -1. Among others it is presented by Cortes and Vapnik, 1995. This hyperplane can be very high dimensional, but for this example it shall be two dimensional, with two dimensional data $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$. In this case it can be defined as

$$\mathbf{w}_0 * \mathbf{x}_i + b_0 = 0, \tag{2.49}$$

leading to

$$\begin{cases} \mathbf{w}_0 * \mathbf{x}_i + b_0 \leq 0; & y_i = -1 \\ \mathbf{w}_0 * \mathbf{x}_i + b_0 \geq 0; & y_i = 1 \end{cases} \tag{2.50}$$

or

$$y_i(\mathbf{w}_0 * \mathbf{x}_i + b_0) \geq 1; \quad 1 \leq i \leq n. \tag{2.51}$$

To determine the optimal hyperplane, Cortes and Vapnik, 1995 suggest to maximize the margin between all of the available data points and the hyperplane

$$\rho(\mathbf{w}, b) = \min_{\mathbf{x}:y=1} \left(\frac{\mathbf{x} * \mathbf{w}}{|\mathbf{w}|} \right) - \max_{\mathbf{x}:y=-1} \left(\frac{\mathbf{x} * \mathbf{w}}{|\mathbf{w}|} \right), \tag{2.52}$$

leading to the optimal hyperplane

$$\rho(\mathbf{w}_0, b_0) = \frac{2}{|\mathbf{w}_0|} = \frac{2}{\sqrt{\mathbf{w}_0 \mathbf{w}_0}}. \tag{2.53}$$

To solve this they determine

$$\mathbf{w}_0 = \sum_{i=1}^{n} y_i \alpha_i^0 \mathbf{x}_i \tag{2.54}$$

with

$$\mathbf{\Lambda}^T = (\alpha_1^0, ..., \alpha_n^0) \tag{2.55}$$

to solve

$$W(\mathbf{\Lambda}) = \mathbf{\Lambda}^T \mathbf{1} - \frac{1}{2} \mathbf{\Lambda}^T D \mathbf{\Lambda} \tag{2.56}$$

$$\mathbf{\Lambda} \geq \mathbf{0} \tag{2.57}$$

$$\mathbf{\Lambda}^T \mathbf{Y} = 0 \tag{2.58}$$

$$\mathbf{1}^T = [1, ..., 1] \text{with length of } n \tag{2.59}$$

$$\mathbf{Y} = [y_1, ..., y_n] \tag{2.60}$$

and finally D being a symmetric matrix of size n x n with elements

$$D_{ij} = y_i y_j \mathbf{x}_i \mathbf{x}_j, \quad 1 \geq i, j < n. \tag{2.61}$$

Solving this is done by providing a large number of training parameters, which leads to the hyperplane being able to separate unknown data of the same kind, if the training was performed well enough. This is an early variant of machine learning, only learning a few parameters with a limited data set, but generally not too dissimilar from modern day neural network training. The presented implementation can only separate data well, if it is able to be perfectly separated by a hyperplane of the selected dimension. A huge amount of extensions to this basic principle is published, for example soft margin classification (Cortes and Vapnik, 1995), which allows some wrong classifications without compromising the whole training.

2.2.2 Neural Network Object Detection

Over the last few years, at time of writing, neural networks have become one of the most if not the most dominant algorithmic solution for object detection and classification tasks. These use a combination of artificial neurons combined to larger networks to take input data, process it and provide a desired output, like a classification result. They come in a very large number of different variants for different applications from classification over detection up to generation or data

modification. This makes it impossible to mention and explain all different kinds here, therefore only the ones relevant for this work shall be talked about here.

Basic Neural Network Architecture

The basis for each artificial neural network is a neuron, designed after neurons in a human or animal brain (Krenker et al., 2011). It has one or more data inputs from which a weighted sum is built, added to a bias and transformed through an activation function:

$$y = F((\sum_{i=0}^{N} w_i x_i) + b). \tag{2.62}$$

The activation function F is hereby very important to the functionality of the network and needs to be adapted to the desired use. For a binary classification output it might be desired that y would directly and discretely point to either one of the two output classes. In this case a step function, a function outputting either 0 or 1, might be the correct choice. In a different situation it might be desired that the certainty of the network could be expressed. For that a sigmoid function like (for example, others exist)

$$F(x) = \frac{1}{1 + e^{-x}} \tag{2.63}$$

might be more appropriate. It still scales the output between 0 and 1 (or -1 and 1 for other definitions), but does not discretely output 0 or 1 but rather values in between. A very large number of activation functions is available today, but naming and explaining them would not be feasible here. An example of this structure is shown in figure 2.5, where a very basic neural network with 3 inputs, 5 outputs, 1 layer and 5 neurons is displayed.

Combining multiple neurons into one network, by building layers and stacking them, creates an artificial neural network which can be used to perform even very complex tasks. Inside these networks a very large number of weights w and biases b exist, which are ultimately what is learned by machine learning. After initialization, either random, 0 or some other variant in between, all these weights and biases have values. After pushing the first set of data through the network an output is created. To improve the network, the quality of this output needs to be quantified and evaluated. For this a cost or loss function is required. A good collection is, among others, provided by Janocha and Czarnecki, 2017. One of the most basic loss functions would be the mean square error (MSE)

$$L(o) = ||y - o||_2^2 \tag{2.64}$$

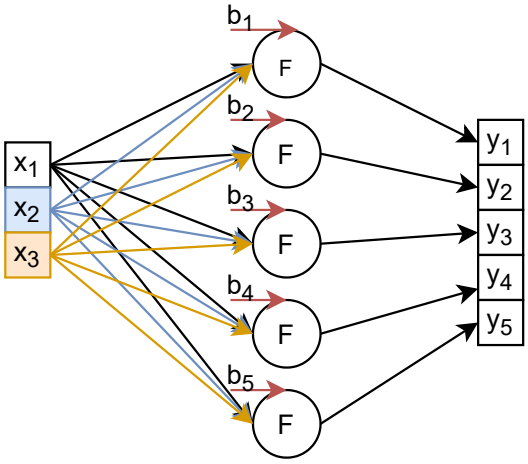


Figure 2.5.: Basic Example of Neural Network with 1 Layer consisting of 5 neurons, 3 Inputs and 5 Outputs; weights w between inputs and nodes omitted for readability, but there is 1 weight for each connection between input and neuron

in which the L2 norm between the network output o and the desired output y is calculated. A more advanced example would be the cross entropy loss

$$L(o) = -y\ log(\sigma(o)), \tag{2.65}$$

in which $\sigma(o)$ describes extracting the certainty from o. After quantifying how right or wrong the network was with its prediction this information can be used to improve the network by adjusting the weights. One popular method for this is stochastic gradient descent (SGD). For this it is assumed, that the loss $L(o)$ is a function of all the different weights w_t at time t in the network: $L(w_t, o)$. As usually more than 1 data point is processed at once, these can be collected, usually averaged, into (following Bottou, 1991)

$$C(w_t) = \frac{1}{N}\sum_{i=1}^{N} L(w_t, o_i). \tag{2.66}$$

To update w_t for each weight for the next iteration, the gradient of the loss function at each weight, $\nabla_w C(w_t)$ is calculated, in practice by backpropagation, as described by Goodfellow et al., 2016 in chapter 6.5, and used as a weighting factor in the final weight update

$$w_{t+1} = w_t - \epsilon \nabla_w C(w_t), \tag{2.67}$$

with ϵ describing the learning rate, or more practically speaking, the speed at which the network adapts to its results. If only a part of the training set is used in each step this is called stochastic gradient descent.

Convolutional Neural Networks

In a basic neural network as presented in section 2.2.2 the input is represented as a one dimensional feature vector. Each value of the input is connected with each neuron by a given weight and processed with this weight. In modern applications this is called a fully connected layer, often used to reduce feature maps to desired final outputs. For multidimensional inputs it is often not feasible to process them this way. As the input needs to be a vector and not a matrix/multidimensional data structure, it would need to be flattened, possibly creating a vector with millions of entries for larger images, resulting in millions of weights and connections. Furthermore it is, especially in images, often beneficial to not process every pixel by itself, but take the area around the pixel in consideration, when calculating a feature vector. For this convolutional neural networks can be used. As explained by Goodfellow et al., 2016, chapter 9, a convolutional neural network (CNN) is a network, or rather network component, in which a multidimensional kernel K, for example 3 x 3, is convolved with the input data I. This convolution

$$S(i,j) = K * I(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n), \tag{2.68}$$

with m and n addressing the different extensions of the filter kernel K and $0 \leq i < x$, $0 \leq j < y$ (x and y being the extend of the input I). This way each position in the image is a part of several calculated features, the number depending on the size of the kernel. Variations are possible in which only every other or even fewer of the positions in the image are processed, resulting in subsampling with learned weights instead of a fixed method. Another possible method of subsampling would be pooling, in which an operation like max pooling over an area only selects the feature with the maximum value to process further. An example of such a convolutional block is shown in figure 2.6. A 3 x 3 filter kernel K is convolved with 8 x 8 input data, zero-padded to 10 x 10 so the convolution can cover all data points similarly, followed by a max pooling block to reduce the final output dimension further.

A typical network consisting of convolutional layers stacks several of them, followed either by a subsampling convolutional layer or max pooling. This results in a processing block which collects features in different layers before scaling down the result. Several of these blocks can be placed one after another including connections between different blocks or layers that are not directly adjacent, for example by Mao et al., 2016.

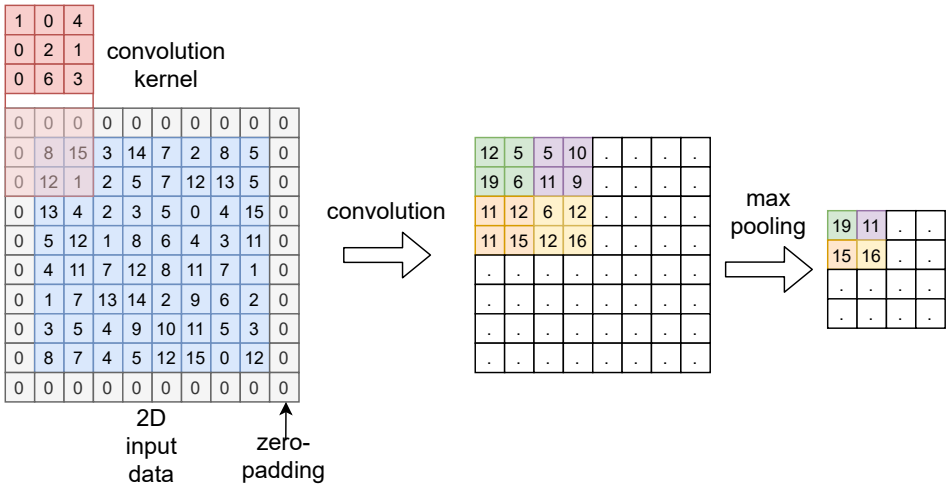


Figure 2.6.: Simple display of convolution of 3 x 3 filter kernel(red), with 8 x 8 input data(blue), followed by max pooling for subsampling; only parts shown

Recurrent Neural Networks

Another different type of network, or network module, is the recurrent neural network (RNN). An RNN can be used to process sequences of data by using a network that shares weights and information across multiple timesteps, as presented in Goodfellow et al., 2016, chapter 10. A major role in this comes to the so called hidden state, often called h, which is the transfer state from one sequence step to the next. In a very simple system

$$h_k = f(h_{k-1}, x_k; \theta) \tag{2.69}$$

would describe the recurrent nature of the hidden state. Breaking this further down, Chung et al., 2014 simplify to

$$h_k = \begin{cases} 0, & k = 0 \\ \Phi(h_{k-1}, x_k), & \text{other} \end{cases}, \tag{2.70}$$

with x_k being the input state and Φ the activation function. A gated recurrent unit (GRU) is a modified version of an RNN, with the following definition being taken from Chung et al., 2014 as well. In a GRU the hidden state is not calculated immediately from the input, but as interpolation between the latest hidden state and the current one

$$h_k = (1 - z_k)h_{k-1} + z_k\hat{h}_k. \tag{2.71}$$

z_k is an update gate, influencing how much the hidden state is updated from the candidate $\tilde{h}_k$. It calculates as

$$z_k = \sigma(W_z x_k + U_z h_{k-1}). \tag{2.72}$$

The candidate is defined as

$$\hat{h}_k = tanh(W x_k + U(r_k \odot h_{k-1}), \tag{2.73}$$

with the reset component

$$r_k = \sigma(W_r x_k + U_r h_{k-1}) \tag{2.74}$$

influencing how much the previous hidden state is represented in the current candidate. Through these definitions, W describes the weights of the training in the current time step k, while U describes the weights in the last time step $k-1$. The activation function σ, also previously called F (equation 2.62) or Φ (equation 2.70), can again be varied, one possible choice would be a hyperbolic tangent function. The naming is adapted from F to keep consistency with the cited source. A second modification of an RNN is given by the "long short term memory (LSTM)" (among others Greff et al., 2016). Both have different methods on how exactly the hidden state is transferred between training steps, and an evaluation is provided by Chung et al., 2014. An overview of a GRU node is shown in figure 2.7.

2.3 Sensor Fundamentals

In this work a multitude of sensor inputs is used. This section is meant to provide a short introduction into the most common sensor types. Cameras (section 2.3.1) are divided into monochromatic (section 2.3.2) and red, green and blue (RGB)-color (section 2.3.2) with some other types, like red pixel cameras, in between. The second major sensor type discussed here, and used in this work, is the Lidar, which is introduced in section 2.3.3. The third important automotive sensor of a Radar is not described here, as it is not used throughout this work.

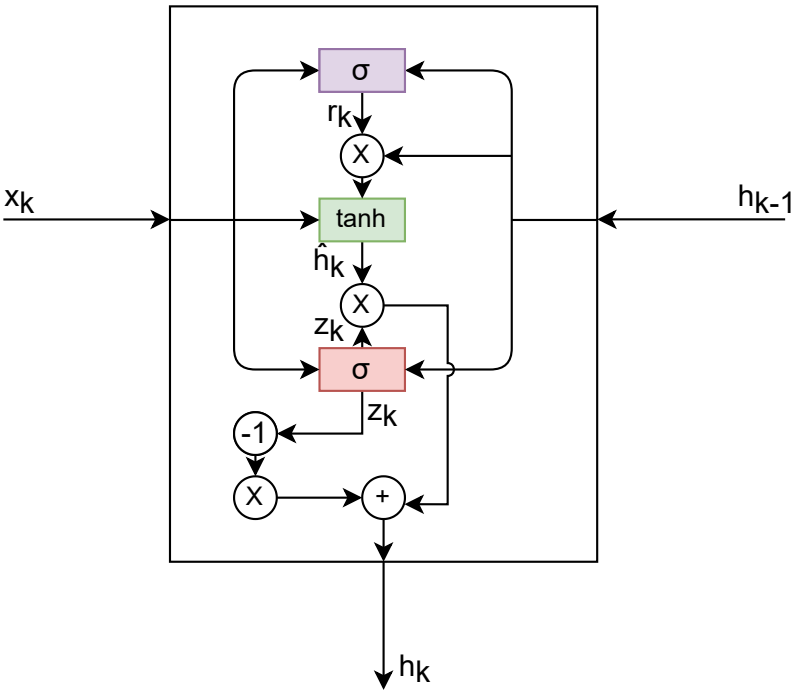


Figure 2.7.: Schematic overview of GRU node. x_k being the new input, h_{k-1} the hidden state from the last frame and h_k the new hidden state, which is also used as output.

2.3.1 Camera Basics

Cameras are one of the most common sensor types in ADAS as they are relatively cheap, provide a clear sensor output and their outputs are easy to understand for a human. Cameras exist in a multitude of different implementations. There are analog cameras that record to photosensitive material and there are digital cameras using some kind of sensor array to capture the scene. In the circumstance of ADAS only digital cameras are relevant, so these will be the focus of this section.

2.3.2 CMOS-Sensors

Over the last few years, "complementary metal oxide semiconductor (CMOS)" type sensors have become the primary sensor for digital camera systems. They work by using a photo sensitive amplification circuit at each pixel position. The brightness value of each pixel is then determined by the voltage level at each pixel. The reading of pixels is usually performed one row at a time. This way the number of detector and processing lanes can be reduced significantly, leading to reductions in required

cost and sensor size. Unfortunately, by reading the sensor output on a row by row basis, potential for error is introduced. If something is moving very fast it might shift in position between reading row r and $r + 1$, leading to a slightly off alignment. This is called rolling shutter effect, as the resulting image looks as it would with a small slit like shutter used on an analog camera exposing the photosensitive film over a given time span and not all at once.

Monochromatic Cameras

A monochromatic camera is a camera that only records 1 brightness value at each pixel position and no color information. A camera image of 1280 x $720 = 921600$ would therefore require 921600 photosensitive pixels and read the 720 rows in sequence. Especially in lower cost systems, monochromatic cameras are easily usable for a large amount of different applications in which color information is not required for accurate classification or detection. One example might be the vehicle light detection and tracking system presented in chapter 3.

Single/Multicolor Cameras

CMOS, and also other sensors like "charge-coupled device (CCD)" only measure the amount of light that hits the sensor and can't deduct a color from that. The most common way to solve this is having separate photosensitive pixels for each color. One pixel in the final image is recorded by 4 color pixels. To limit each pixel to record information for 1 color only, a filter layer is used in front of the sensor, designed to remove everything but 1 color from each sensor pixel. Usually the Bayer pattern (Bayer, 1976) is used, using one red, one blue and two green pixels, but variations are possible. Because of this, full color cameras require 4 times as many pixels as monochromatic cameras to achieve the same output resolution, making them significantly more expensive and the recorded images larger.

A middle ground may be found in red-pixel cameras. These are mostly monochromatic but use a red filter at the top left corner of every 2 x 2 pixel block. The resulting image provides low resolution red information and higher resolution brightness information. This can be useful if limited color information is of use for classification, for example of traffic signs, head light compared to tail light and others.

Much more detail on camera sensor basics is available, among others by Gouveia and Choubey, 2016, providing details on sensor design, the rolling shutter effect and other basics.

Besides the sensor, a camera also needs to be calibrated, both intrinsically and extrinsically. The first to model the internal camera parameters – focal length, lense distortions, distances between sensor and lense and others – while the second describes where the camera sits in the experimental setup. Intrinsic camera calibration is, among many others, described by Zhang, 2000. As this is a complex topic, and one does not need to understand details to read this work, not more information shall be provided here.

2.3.3 Lidar Basics

Lidar, or “light detection and ranging", is a type of active sensor, in which a number of laser light pulses is sent out from the sensor and the time it takes for reflected light to get back to the sensor is measured (Velodyne Lidar Inc., 2021b). Lidars exist in multiple configurations. There are solid state lidars (eg. Blickfeld GmbH, 2021) with a limited number of scanlines and limited field of view and also rotating lidars like the Velodyne HDL-64 (Velodyne Lidar Inc., 2021a) which has a similar sensor setup but is generally a lot more expensive and rotates quickly to allow 360° capture of the sensor surrounding. Baseline for all of the developments in chapter 4 to chapter 6 were rotating lidars. Mostly the aforementioned Velodyne HDL64, but also the Hesai Technology Pandora (Hesai Technology, 2021). These use 64 (HDL-64) respectively 40 (Pandora) scanlines and produce high resolution 360° scans with a decent range. These point clouds are generally around 120.000 data points for the HDL-64 and at max 72000 for the Pandora. The Pandora can increase this by double remission, where each point is measured twice, but generally there is a large amount of overlap.

One example of a raw, unfiltered lidar point cloud is provided in figure 2.8. It is easy to see the different scan lines, oriented circular around the ego vehicle in the middle of the dead zone in the middle of the point cloud. This dead zone appears, as the lidar is mounted on top of the vehicle and only scans a few degrees down from its mounting point.

Figure 2.8.: Unfiltered Pointcloud from Kitti (Geiger et al., 2013); Reflection intensity encoded in brightness of points

3 Advancements in Small Object Tracking in Camera Images

In modern ADAS, precision and reliability are getting increasingly important with the driver putting more trust into the systems. In this chapter the reader will be presented with advancements in tracking algorithms specifically designed for tracking small objects or targets for Advanced Headlight Control (AHC). These targets will be vehicle head and tail lights, which need to be detected and tracked at long distances and need to be robust against occlusion and or visual changes. An overview of where the tracker developed here is sitting in the overall system is presented in figure 3.1. The tracking system itself is detailed in figure 3.2. The optical tracker mentioned will be detailed in section 3.3 and the OPTICS (Ankerst et al., 1999) clustering in section 3.5. The main focus though will be the GM-PHD in section 3.2.

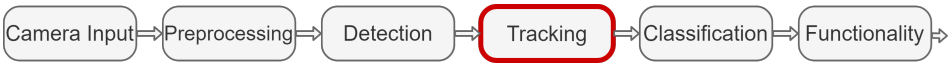


Figure 3.1.: System overview of where the tracker is positioned in the overall ADAS system.

3.1 Challenges of Automatic Headlight Control

For AHC systems a multitude of requirements need to be fulfilled by algorithms. These systems require reliable and stable detections of oncoming or preceding vehicles for the high beam to be switched off at exactly the correct angle without flickering and without missing an object, which could lead to a driver potentially being blinded by a flash of light. At night other traffic participants can, at range, not easily be detected by their full vehicle, as there is not enough light for cameras to work and both radar and lidar sensors don't generally have the required range capabilities, at least not if good resolution is required. Therefore a good way of detecting other vehicles is by their head and tail lights. These are visible at high ranges, will mostly be present and have at least partially defined dimensions. Still, it is not easy to detect vehicles reliably like this.

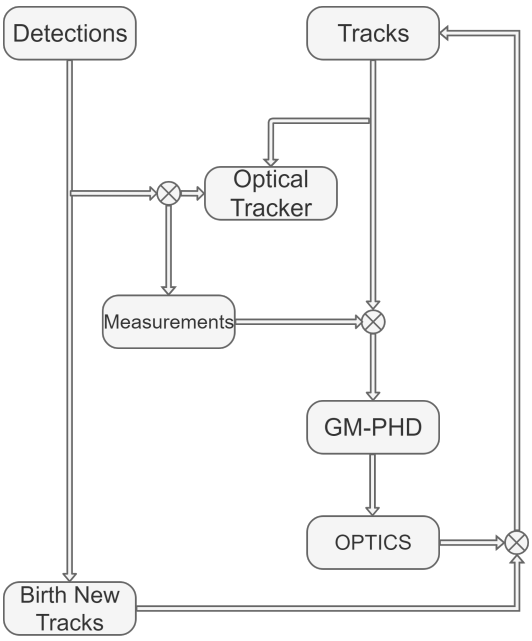
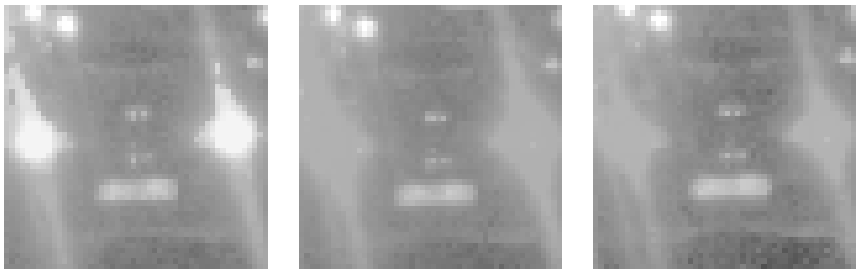
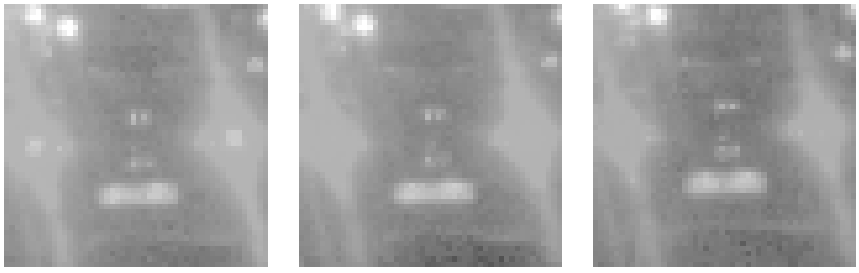


Figure 3.2.: Structure of the overall tracking system, including the optical tracker, the GM-PHD and the OTPICS clustering. Figure from Alsfasser et al., 2019

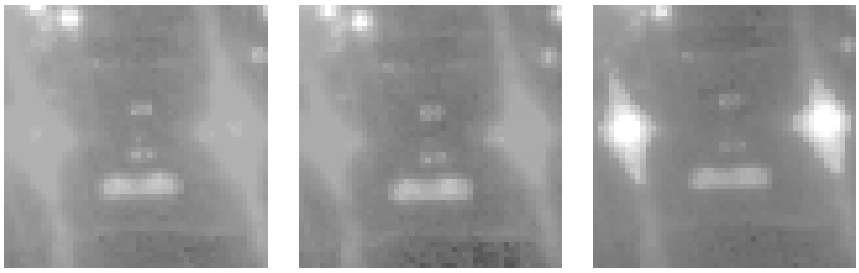
Modern head and tail lights often use LEDs as their light source, which, unlike traditional halogen or xenon lights, often do not output continuous light but are rather switched on and off at a high frequency to reduce power consumption, heat output and also to regulate the perceived brightness of the light. This is called pulse width modulation (PWM). For the human driver the light seems continuous as the eye is slow to react to changes in light intensity, as shown and explained, among others, by Lehman and Wilkins, 2014. For a rolling shutter based camera sensor, as may be used in ADAS this is different. The sensors collects photons for a given amount of time before the shutter is closed and voltage levels at the pixels are measured. A rolling shutter, as described in section 2.3.1, means that not all pixels collect light at exactly the same time, therefore, if the frequencies of the shutter and the PWM controlled LED are not synced perfectly, some pixels will collect more light than others at a given time step or frame in the extracted video. This shows up as partially or even fully switched off lights in the camera images, although the light seems to be emitting continuously to the human driver. This behaviour can make it very difficult for the detector stage of the algorithm to reliably detect all light sources, as the differences can be very large and the appearance of a light might vary a lot between frames. The tracker is therefore required to compensate for missed or partial detections without dropping the track or suddenly shrink and extend it as this would skew other parts of the algorithm that might rely on track size, for example for estimating the distance of the oncoming or preceding vehicle. The variation this



(a) frame = 39 (b) frame = 40 (c) frame = 41



(d) frame = 42 (e) frame = 43 (f) frame = 44



(g) frame = 45 (h) frame = 46 (i) frame = 47

Figure 3.3.: Sequence of images showing variance in tail light intensity because of PWM

creates is shown in the sequence of figures, figure 3.3. It can be seen how the tail light appears switched off for a large percentage of the time, with some partial light emission in between and large blooming recordings at some more rare points in time in frames 39 and 47. This is a very large challenge for the detector which feeds detections to the tracker. Therefore, if the detector fails, as it might on frames 40, 41, 43, 44 and 46, the tracker needs to be able to keep the track stable throughout.

The next problematic circumstance with similar results is the weather. A wet wind shield might break the light in unpredictable ways or the wind shield wipers might, same as the PWM LED's, occlude parts of the lights or vehicles. One example of rain drops in the image is shown in figure 3.4. As can be seen, the raindrops produce highlights in the image, which the tracker needs to be able to suppress.

Furthermore as light sources need to be tracked over a very large range of distances and therefore sizes, the tracker needs to be able to adapt enough to dimensional

Figure 3.4.: Raindrops might cause drops on the windshield, which break the light in possibly unpredictable ways

variations, without being too noisy and susceptible to flashes caused by the light breaking in an unpredictable way. An example of blooming is shown in figure 3.5, where the camera adjusts to the surrounding brightness, leading to some areas of the image producing heavy blooming.

Finally, light is reflected by the ground, especially when wet, but those reflected lights have to be filtered out and cannot be confused with real light sources.

3.1.1 Performance Requirements for Safe on-Road Usage

As ADAS are safety relevant features in potentially high speed vehicles, they need to fulfill a large amount of requirements provided by customers, vehicle manufacturers and regulatory public authorities.

In the case of the AHC system developed here, vehicles need to be reliably detected both, at large distances up to 800 m but also close to the ego vehicle. This requires a very flexible tracker that can deal with a large variance in detection size, frame to frame track movement and track growth or shrinkage.

This tracker is required to run in real time at the same frequency as the camera outputs frames, 15 Hz. The actual processing time varies with the number of detections, a fixed performance can therefore only be achieved if the number of

Figure 3.5.: Different light intensities can lead to more or less blooming in the image

output tracks is artificially limited. This limit cannot be too low and needs to be at least 50, value derived experimentally on a private dataset, for reliable handling of all occurring situations.

3.2 Gaussian-Mixture Probability Hypothesis Density Filter for Multiple Target-Multiple Detection Tracking

The Gaussian-mixture probability hypothesis density (GM-PHD) filter is a flavour, or solution, of a PHD filter implementation that has a closed form solution presented by Vo and Ma, 2006. It is a variant of the PHD filter, which is explained in section 2.1.2 initially. Other than the original PHD filter the GM-PHD is limited to work with targets that follow "linear Gaussian dynamics" (Vo and Ma, 2006).

3.2.1 Definition of a Gaussian-Mixture Probability Hypothesis Density Filter

As presented in section 2.1.2 the PHD recursion, taken from Vo and Ma, 2006, is given as

$$\nu_{k|k-1} = \int p_{S,k}(\zeta) f_{k|k-1}(x|\zeta)\nu_{k-1}(\zeta)d\zeta + \int \beta_{k|k-1}(x|\zeta)\nu_{k-1}(\zeta)d\zeta + \gamma_k(x) \quad (3.1)$$

$$\nu_k(x) = [1 - p_{D,k}(x)]\nu_{k|k-1}(x) + \sum_{z\in Z_k} \frac{p_{D,k}(x)g_k(z|x)\nu_{k|k-1}(x)}{\kappa_k(z) + \int p_{D,k}(\xi)g_k(z|\xi)\nu_{k|k-1}(\xi)d\xi}. \quad (3.2)$$

As by Vo and Ma, 2006 this has no closed form solution and numerical integration is limited by the 'curse of dimensionality'. They explain that for linear Gaussian multi-target models this admits the required closed form solution. For this they set several new assumptions. The first being linear Gaussian behaviour of both, the model $f_{k|k-1}(\mathbf{x}|\zeta) = \mathcal{N}(\mathbf{x}; F_{k-1}\zeta, Q_{k-1})$ and the sensor $g_k(\mathbf{z}|k) = \mathcal{N}(\mathbf{z}; H_k\mathbf{x}, R_k)$, with $\mathcal{N}$ being a Gaussian probability density distribution. Survival and detection probabilities need to be state independent, therefore

$$p_{S,k}(\mathbf{x}) = p_{S,k} \quad (3.3)$$

$$p_{D,k}(\mathbf{x}) = p_{D,k}. \quad (3.4)$$

Vo and Ma, 2006 lastly set the assumption of birth and spawn RFSs being Gaussian mixtures

$$\gamma_k(\mathbf{x}) = \sum_{i=1}^{J_{\gamma,k}} w_{\gamma,k}^{(i)} \mathcal{N}(\mathbf{x}; \mathbf{m}_{\gamma,k}^{(i)}, P_{\gamma,k}^{(i)}) \quad (3.5)$$

$$\beta_{k|k-1}(\mathbf{x}|\zeta) = \sum_{j=1}^{J_{\beta,k}} w_{\beta,k}^{(i)} \mathcal{N}(\mathbf{x}, F_{\beta,k-1}^{(j)}\zeta + \mathbf{d}_{\beta,k-1}^{(i)}, Q_{\beta,k-1}^{(i)}), \quad (3.6)$$

with several parameters for modeling birth and spawn behaviour of new tracks. In case these assumptions hold, the GM-PHD recursion can be solved in a closed form algorithm. The algorithm again is taken from Vo and Ma, 2006, with custom changes and modifications shown in the following section section 3.2.2. Assuming the a posteriori intensity at frame or time $k-1$ is given as

$$\nu_{k-1}(\mathbf{x}) = \sum_{i=1}^{J_{k-1}} w_{k-1}^{(i)} \mathcal{N}(\mathbf{x}; \mathbf{m}_{k-1}^{(i)}, P_{k-1}^{(i)}), \quad (3.7)$$

the predicted, a priori, intensity for frame k is calculated as

$$\nu_{k|k-1}(\mathbf{x}) = \nu_{S,k|k-1}(\mathbf{x}) + \nu_{\beta,k|k-1}(\mathbf{x}) + \gamma_k(\mathbf{x}). \tag{3.8}$$

$\gamma_k(\mathbf{x})$ is the birth intensity of new tracks, as calculated in equation 3.5. Further components of the a priori intensity are the predicted intensities of old tracks, $\nu_{S,k|k-1}(\mathbf{x})$ and predicted intensities of new track spawns $\nu_{\beta,k|k-1}(\mathbf{x})$. Predicted intensity $\nu_{S,k|k-1}(\mathbf{x})$ is set as follows:

$$\nu_{S,k|k-1}(\mathbf{x}) = p_{S,k} \sum_{j=1}^{J_{k-1}} w_{k-1}^{(j)} \mathcal{N}(\mathbf{x}; \mathbf{m}_{S,k|k-1}^{(j)}, P_{S,k|k-1}^{(j)}) \tag{3.9}$$

$$\mathbf{m}_{S,k|k-1}^{(j)} = F_{k-1} \mathbf{m}_{k-1}^{(j)} \tag{3.10}$$

$$P_{S,k|k-1}^{(j)} = Q_{k-1} + F_{k-1} P_{k-1}^{(j)} F_{k-1}^T, \tag{3.11}$$

with $m_{S,k|k-1}^{(j)}$ the mean of a Gaussian component of the mixture, and $P_{S,k|k-1}^{(j)}$ the covariance of a Gaussian component. For the Gaussian mixture of newly spawned tracks, the predicted Gaussian probability densities are weighted as in

$$\nu_{\beta,k|k-1}(\mathbf{x}) = \sum_{j=1}^{J_{k-1}} \sum_{l=1}^{J_{\beta,k}} w_{k-1}^{(j)} w_{\beta,k}^{(l)} \mathcal{N}(\mathbf{x}; \mathbf{m}_{\beta,k|k-1}^{(j,l)}, P_{\beta,k|k-1}^{(j,l)}) \tag{3.12}$$

$$\mathbf{m}_{\beta,k|k-1}^{(j,l)} = F_{\beta,k-1}^{(l)} \mathbf{m}_{k-1}^{(j)} + \mathbf{d}_{\beta,k-1}^{(l)} \tag{3.13}$$

$$P_{\beta,k|k-1}^{(j,l)} = Q_{\beta,k-1}^{(l)} + F_{\beta,k-1}^{(l)} P_{\beta,k-1}^{(j)} (F_{\beta,k-1}^{(l)})^T, \tag{3.14}$$

to produce the final a priori intensity, $d_{\beta,k-1}^{(l)}$ being a distance from the track mean on which the spawn is based. For updating tracks with measurements, this a priori intensity can be described as

$$\nu_k(\mathbf{x}) = (1 - p_{D,k}) \nu_{k|k-1}(\mathbf{x}) + \sum_{\mathbf{z} \in Z_k} \nu_{D,k}(\mathbf{x}; \mathbf{z}), \tag{3.15}$$

which is a combination of predicted tracks which don't get an update, $(1-p_{D,k})\nu_{k|k-1}(\mathbf{x})$ and updated tracks $\sum_{\mathbf{z} \in Z_k} \nu_{D,k}(\mathbf{x}; \mathbf{z})$. This way tracks don't necessarily die out without updates for short times. While this helps, an extension to this property is described in section 3.3. The post update component of the a posteriori intensity is again calculated as

$$\nu_{D,k}(\mathbf{x}; \mathbf{z}) = \sum_{j=1}^{J_{k|k-1}} w_k^{(j)}(\mathbf{z}) \mathcal{N}(\mathbf{x}; \mathbf{m}_{k|k}^{(j)}(\mathbf{z}); P_{k|k}^{(j)}), \tag{3.16}$$

a weighted sum of probabilities of state x belonging to each probability distribution of which the intensity is comprised. The weight for each step calculates as

$$w_k^{(j)}(\mathbf{z}) = \frac{p_{D,k} w_{k|k-1}^{(j)} q_k^{(j)}(\mathbf{z})}{\kappa_k(\mathbf{z}) + p_{D,k} \sum_{l=1}^{J_{k|k-1}} w_{k|k-1}^{(j)} q_k^{(j)}(\mathbf{z})}, \tag{3.17}$$

with

$$q_k^{(j)}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; H_k \mathbf{m}_{k|k-1}^{(j)}, R_k + H_k P_{k|k-1}^{(j)} H_k^T), \tag{3.18}$$

where the single components basically follow the Kalman filter (Kalman, 1960) update step, as described in section 2.1.1:

$$\mathbf{m}_{k|k}^{(j)}(\mathbf{z}) = \mathbf{m}_{k|k-1}^{(j)} + K_k^{(j)}(\mathbf{z} - H_k \mathbf{m}_{k|k-1}^{(j)}) \tag{3.19}$$

$$P_{k|k}^{(j)} = [I - K_k^{(j)} H_k] P_{k|k-1}^{(j)} \tag{3.20}$$

$$K_k^{(j)} = P_{k|k-1}^{(j)} H_k^T (H_k P_{k|k-1}^{(j)} H_k^T + R_k)^{-1}. \tag{3.21}$$

The a posteriori intensity is finally a mixture of several, in this implementation thousands, of Gaussian probability distributions. The overall progress of distribution means over the last frames output $\mathbf{m}_{k-1}$, over the a priori means $\mathbf{m}_{S,k|k-1}$ and finally to the a posteriori means $\mathbf{m}_{k|k}$ is shown in figure 3.6 - figure 3.9. These images only ever show circles around the center position of each distribution mean, they are not properly scaled and don't show any information about other components of the state means. The a posteriori means are displayed slightly differently, as they are roughly scaled to the a posteriori weight $w_k^{(j)}(z)$, but not fully to scale. Bad fits are even more weighted down than shown here, but would be invisible shown at scale. As can be seen, the update step of the GM-PHD produces a very large number of weighted Gaussian distributions, which can be an issue with processing speed. Additionally the desired output needs to be extracted from these, shown later in section 3.2.2 as pruning.

3.2.2 Optimizing the GM-PHD for Vehicle Light Tracking

The implementation of the GM-PHD used here is modified in several places for vehicle light tracking.

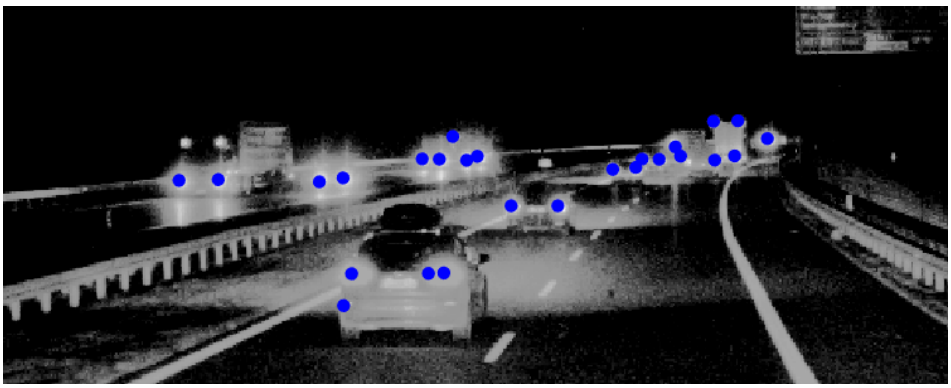


Figure 3.6.: Track outputs $\tilde{\mathbf{m}}_{k-1}$ Frame 167

Figure 3.7.: Track predictions, a priori distribution means, $\mathbf{m}_{S,k|k-1}$, frame 168

Parameters and State definition

When designing or adapting a tracking algorithm, one of the first choices has to be the state and the motion model that will be estimated. In this case the choice fell on a constant acceleration model, meaning the state contains position, velocity and acceleration, therefore track movement is estimated from three parameters in each movement direction. Furthermore the track size s is estimated. This results in the final state being chosen as

$$\mathbf{m}_k^{(i)} = \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \\ s \\ a_x \\ a_y \end{bmatrix} . \tag{3.22}$$

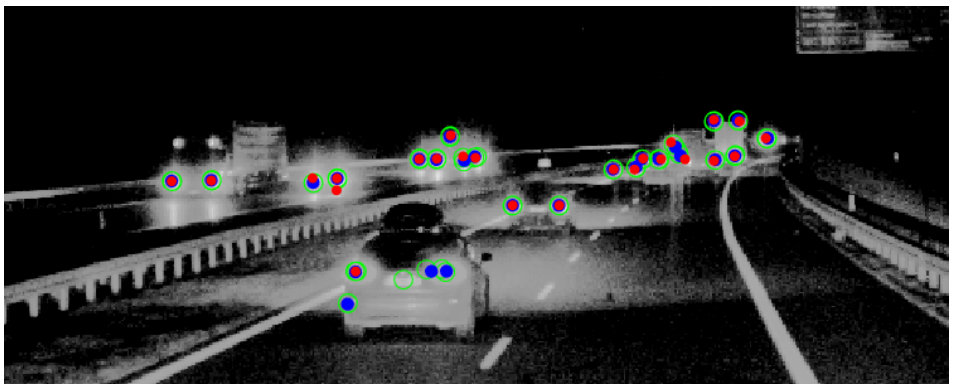


Figure 3.8.: Track predictions, a priori distribution means, $\mathbf{m}_{S,k|k-1}$, frame 168 + new measurements (red = detector, blue = optical tracker)

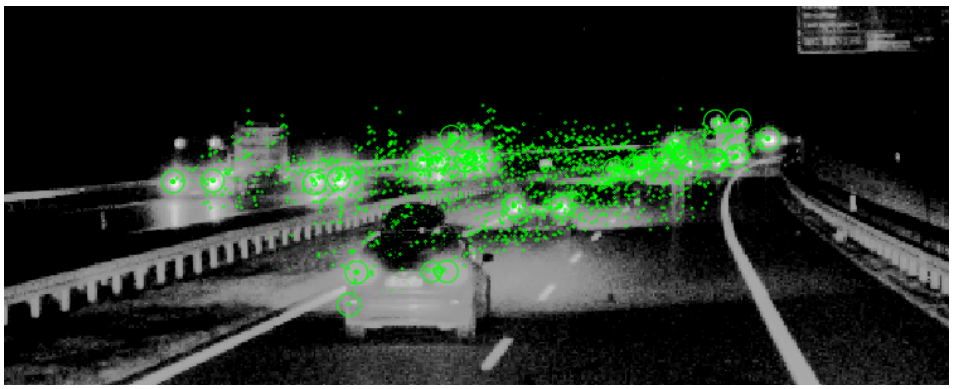


Figure 3.9.: Track updates, a posteriori distribution means $\mathbf{m}_{k|k}(z)$, of each a priori mean $\mathbf{m}_{S,k|k-1}$ combined with each possible measurement $\mathbf{z}_k$. Circle size propotional to weight, but not to scale

In this case the state transition matrix F_{k-1} is chosen to predict constant movement between frames as $\mathbf{m}^{(i)}_{S,k|k-1} = F_{k-1}\mathbf{m}^{(i)}_{k-1}$

$$F_k = \begin{pmatrix} \Delta t & 0 & \Delta t & 0 & 0 & \frac{\Delta t}{2} & 0 \\ 0 & \Delta t & 0 & \Delta t & 0 & 0 & \frac{\Delta t}{2} \\ 0 & 0 & \Delta t & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & \Delta t & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \Delta t \end{pmatrix}. \tag{3.23}$$

Further important choices are both, the process noise Q

$$Q = \begin{pmatrix} \frac{q_x^2}{4} & 0 & \frac{q_x^2}{2} & 0 & 0 & \frac{q_x^2}{2} & 0 \\ 0 & \frac{q_y^2}{4} & 0 & \frac{q_y^2}{2} & 0 & 0 & 0\frac{q_y^2}{2} \\ \frac{q_x^2}{2} & 0 & q_x^2 & 0 & 0 & q_x^2 & 0 \\ 0 & \frac{q_y^2}{2} & 0 & q_y^2 & 0 & 0 & q_y^2 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\ \frac{q_x^2}{2} & 0 & q_x^2 & 0 & 0 & q_x^2 & 0 \\ 0 & \frac{q_y^2}{2} & 0 & q_y^2 & 0 & 0 & q_y^2 \end{pmatrix}, \tag{3.24}$$

with

$$q_x = \text{Estimated process noise in x-direction} \tag{3.25}$$

$$q_y = \text{Estimated process noise in y-direction} \tag{3.26}$$

and measurement, or observation, noise R:

$$R = \begin{pmatrix} r_x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & r_y & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & r_{v_x} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & r_{v_y} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & r_s & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & r_{a_x} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & r_{a_y} \end{pmatrix}. \tag{3.27}$$

$\mathbf{r} = [r_x, r_y, r_{v_x}, r_{v_y}, r_s, r_{a_x}, r_{a_y}]^T$ can either be estimated, or, in case a large enough database of labeled data is available, calculated as mean square error (MSE) between observations obs and desired output $label$:

$$\mathbf{r} = \frac{1}{N} \sum_{n=1}^{N} (obs - label)^2. \tag{3.28}$$

Calculating the observation error can be more precise, as is the precise observation error of the measurement source, but if not enough data is available to encompass a large variety of situations, ideally examples of all possible situations, an educated estimation can be the better choice to avoid bias. Still, the calculated observation noise can be a guideline for this estimation.

Pruning

When using a PHD or GM-PHD filter for tracking, the output of the filter is not a list of tracks, but rather a very large number of Gaussian probability distributions, defined by a weight, mean state and state covariance. An overwhelming majority of these distributions has weights of zero or very close to zero, because they describe tracks updated with measurements that don't match the track details at all. These can be filtered by thresholding by a threshold T_{prune} on the weight, but to get a final output of a limited number of tracks, multiple Gaussian distributions have to be merged together. This pruning was also introduced by Vo and Ma, 2006. Those distributions, which lie above the weight threshold, are then sorted in descending order. From this sorted list of points the merges, and therefore final output tracks/distributions are calculated. Closeness, or mergeability, between tracks is calculated with the Mahalanobis distance (Mahalanobis, 1936) between distributions meaning:

$$d_k^{(i,j)} = (\mathbf{m}_k^{(i)} - \mathbf{m}_k^{(j)})^T (P_k^{(i)})^{-1} (\mathbf{m}_k^{(i)} - \mathbf{m}_k^{(j)}). \tag{3.29}$$

Once a collection of close distributions, $d_k^{(i,j)} < T_{merge}$ is found, weighted averaging is used to calculate the final output state plus covariance. For the following equations, l describes the index of the current output collection/merge, while L describes the current collection of distributions

$$\tilde{w}_k^{(l)} = \sum_{i=0}^{L} w_k^{(i)} \tag{3.30}$$

$$\tilde{\mathbf{m}}_k^{(l)} = \frac{1}{\tilde{w}_k^{(l)}} \sum_{i=0}^{L} w_k^{(i)} \mathbf{m}_k^{(i)} \tag{3.31}$$

$$\tilde{P}_k^{(l)} = \frac{1}{\tilde{w}_k^{(l)}} \sum_{i=0}^{L} w_k^{(i)} (P_k^{(i)} + (\tilde{\mathbf{m}}_k^{(l)} - \mathbf{m}_k^{(i)})(\tilde{\mathbf{m}}_k^{(l)} - \mathbf{m}_k^{(i)})^T). \tag{3.32}$$

l and all elements in L are then removed from the overall collection of thresholded distributions. This is repeated until all Gaussian distributions above the given threshold are processed and pruned. Figure 3.10 shows a combination of the a posteriori states and the pruned outputs (blue). As can be seen a very large number of a posteriori distributions exist, which need to be reduced down to a more manageable number, which is what pruning does by thresholding and calculating weighted sums from multiple distributions.

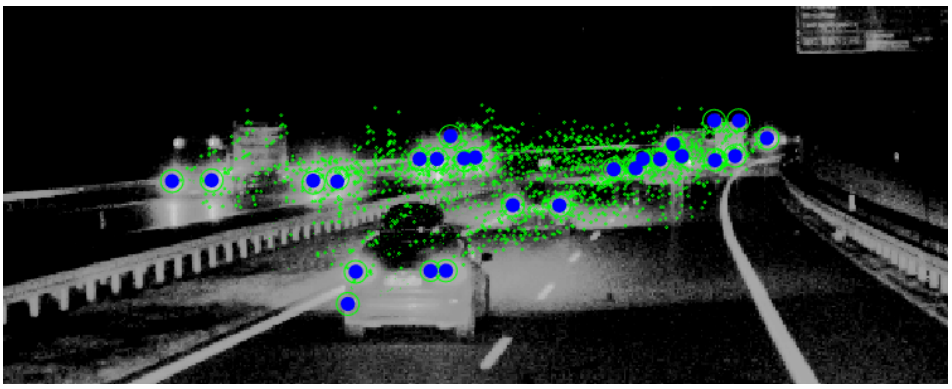


Figure 3.10.: A posteriori means $\mathbf{m}_{k|k}(\mathbf{z})$, shown with pruning results $\tilde{\mathbf{m}}_k$(blue)

Track Health

One important aspect of tracker tuning lies in the judgement of when to let tracks die and when to keep them alive. While the GM-PHD already has some method of doing this, by dimishing weight over time, until the track is no longer above the pruning threshold, a more flexible solution was integrated here. This means, that every track, or rather every Gaussian distribution, contains, besides its mean, covariance and weight, a health value $h_k^{(i)}$. This health value is automatically reduced by 1 during the prediction, or a priori, step of the algorithm, which would lead to the health value approaching 0, unless it is somehow increased or refreshed. This refresh is performed during the update step. Every time a distribution is updated with an actual track detection, its health value is reset to its original value. Predictions from the optical tracker don't refresh the track health as otherwise tracks would almost never die, as total failure of the optical tracker is exceedingly rare. During pruning the health of the final output distribution is taken as the maximum health of any component of that distribution. If the health of a distribution is still 0 or lower than 0 $\tilde{h}_k^{(l)} \leq 0$ after pruning, the track is not continued and will not be projected further. Additionally this health value can be used to filter outputs in a way so that a separate display threshold, higher than the dying threshold, can be used, so tracks that get increasingly uncertain start not being output anymore, while still having a chance to stay alive.

Birth and Spawning

Detector detections are allowed to birth new targets after each processed frame. Meaning that all measurements that were used in a frame and were created by the detector, not the optical tracker, generate one new track, or rather Gaussian

```
for i in range(max_runs):
    get_random_parameters()

    for video in videos:
        results = perform_tracking(video)

        mota, motp, tp = calculate_metrics(results)

        record_results(mota, motp, tp)
        if tp < tp_cutoff:
            break
```

Listing 3.1: Random parameter search for GM-PHD

distribution, each. Equation 3.5 shows the resulting intensity. The birth state $\mathbf{m}_{\gamma,k}^{(i)}$ is simply chosen as the measurement translated into state space, birth covariance $P_{\gamma,k}^{(i)}$ is chosen as the process noise Q, as shown in equation 3.24, but linearly scaled for the size of the newly birthed track. Birth intensity/weight $w_{\gamma,k}^{(i)}$ is assuming linear probability all across the detection region of interest (ROI), and results in $w_{\gamma,k}^{(i)} = \frac{1}{roi_x * roi_y}$. Newly birthed tracks start with a health value of 1, meaning they require an actual measurement update in the next time step or they will die again. They will therefore also not being output before they receive at least one update.

3.2.3 Finding an optimized parameter set

A GM-PHD filter has a large number of parameters that can be used for tuning. Estimated process noise Q, measurement/observation noise R, survival probability $p_{S,k}$, detection probability $p_{D,k}$, thresholds for pruning T_{prune} and merging T_{merge} and also birth health and birth parameters for P. For this implementation a simple random parameter selection between human picked boundaries and step sizes was chosen on the mentioned parameter set. Random search usually provides quicker results than a grid search, shown by Bergstra and Bengio, 2012, as a larger parameter space can be covered in the same amount of time. While this might result in similar parameter sets being evaluated multiple times, this is outweighed by much faster coverage of the parameter space. Listing 3.1 shows, very abbreviated, how this random search is structured. Basically a maximum number of tests is run on a given list of videos. After each processed video, the result metrics "multi object tracking accuracy (MOTA)", "multi object tracking precision (MOTP)" and "true positive (TP)-rate" are calculated and recorded. The TP-rate is used for a cutoff, in case a parameter set does not reach a fixed performance the run is canceled early

to save time. MOTA and MOTP are taken from Bernardin and Stiefelhagen, 2008 and are used in plenty of tracker comparisons. They are calculated as

$$\text{MOTP} = \frac{\sum_{k=1}^{N} \sum_{i=1}^{c_k} d_k^{(i)}}{\sum_{k=1}^{N} c_k} \tag{3.33}$$

with

$$N = \text{Number of frames in a video} \tag{3.34}$$
$$d_k^{(i)} = \text{Distance between track i and label i in frame k} \tag{3.35}$$
$$c_k = \text{Number of matches in frame k,} \tag{3.36}$$

and

$$\text{MOTA} = 1 - \frac{\sum_{k=1}^{N} (FN_k + FP_k + MMT_k)}{GT} \tag{3.37}$$

with

$$FN_k = \text{false negative in frame k} \tag{3.38}$$
$$FP_k = \text{false positive in frame k} \tag{3.39}$$
$$MMT_k = \text{mismatched tracks in frame k} \tag{3.40}$$
$$GT = \text{ground truth,} \tag{3.41}$$

and, finally, the TP-rate, the number of correctly tracked samples divided by the number of overall labels, GT as

$$\text{TP-rate} = \frac{\sum_{k=1}^{N} TP_k}{GT} \tag{3.42}$$

.

3.3 Solving Asynchronism Between PWM Light Sources and CMOS Image Sensors

As previously explained, the fact that LED light sources are not constantly on, but regulated by PWM, leads to them being usually asynchronous with the camera CMOS sensors readout. This leads, or rather can lead, to LED lighs looking partially or fully turned off, when in fact they are on and should be detected.

Adjusting the tracker to track these reliably can be a challenge, depending on the detection algorithm used. Therefore a useful upgrade to the tracker would be the capabilities to properly compensate for missed or partial detections.

As the PHD filter works by predicting tracks and allows for tracks to continue without confirmation by new detections this can be done and supported in multiple ways.

First, as shown in equation 3.15 and the subsequent paragraph, one copy of track predictions is processed without applying an update to them. These no-update predictions are weighted down by a constant factor. If this factor is sufficiently large, so the weights stay decently high for some time, this can already bridge a limited number of missing frames, without loosing a large amount of accuracy.

Unfortunately these 'kept-alive' tracks are, if unchecked, prone to slowly move off of their target, grow or shrink and otherwise quickly get less accurate. To support this, an extra detection for each track is provided by a supplemental Lucas-Kanade optical tracker. This tracker tries to match a part of an image in one frame with the same part in the next frame and, although partially off looking vehicle lights have a different illumination and outer shape, their rough shape and the orientation of the image gradients in the patch are similar and a match can usually be found.

3.3.1 Lucas-Kanade Optical Tracker

As previously described in section 2.1.1, there are multiple ways to calculate the Lucas-Kanade tracking. For faster computation the inverse compositional algorithm (Baker and Matthews, 2004) is adapted, resulting in an initial error estimation

$$E = \sum_{\mathbf{c}} [I_{k-1}(A\mathbf{c} + \mathbf{b}) - I_k(\mathbf{c})]^2], \tag{3.43}$$

with E being the error of the match, $\mathbf{c} = [x\ y\ 1]^T$ a pixel position in homogeneous coordinates, $\mathbf{b}$ a translation vector and A a transformation matrix. With the combination of translation and a transformation, the shift between frame $k-1$ and k can be estimated. In this case not many transformations are relevant or even possible, therefore A is reduced to a scaling factor s, resulting in

$$E = \sum_{\mathbf{c}} [I_{k-1}(s(\mathbf{c} - \mathbf{c}_c) + \mathbf{c}_c + \mathbf{b}) - I_k(\mathbf{c})]^2], \tag{3.44}$$

with $\mathbf{c}_c$ being the center position of the current patch, as the scaling and translation is calculated in relation to this center position. Substituting the shift by

$$\mathbf{x}(\mathbf{c}, \mathbf{p}_0) = s(\mathbf{c} - \mathbf{c}_c) + \mathbf{c}_c + \mathbf{b}, \tag{3.45}$$

with

$$\mathbf{p}_0 = \begin{bmatrix} s \\ b_x \\ b_y \end{bmatrix}, \tag{3.46}$$

and then iteratively calculating $\mathbf{p}_0$ as $\mathbf{p}_0 = \mathbf{p}_0 + \Delta\mathbf{p}$ allows for freely choosing the desired accuracy with the number of performed iterations. Solving this as described by Lucas and Kanade, 1981 or Baker and Matthews, 2004 results in one iteration of the update as

$$\Delta\mathbf{p} \approx \left[\sum_{\mathbf{c}} [I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{c}, \mathbf{p}_0))] \left(\frac{\partial I_{k-1}}{\partial \mathbf{c}}\right)\right] \left[\sum_{\mathbf{c}} \left(\frac{\partial I_{k-1}}{\partial \mathbf{c}}\right)^T \left(\frac{\partial I_{k-1}}{\partial \mathbf{c}}\right)\right]^{-1}. \tag{3.47}$$

Several of these iterations, the exact number being manually tuned, finally produce a good translation/scale estimation between both frames.

3.3.2 Regularization Methods to Stabilize KLT Tracking in Noisy Environments

The optical tracker presented in section 3.3.1 is usually able to find the correct match between a track in $k-1$ and k. As an iterative matching process it can sometimes be lead too far away from the correct target by one bad match in the first iteration, making it very time consuming or impossible to ever find the correct match for the target.

Therefore it can be advantageous to limit the amount of movement per iteration, without making large movements impossible with strong enough evidence. For this regularization can be used, as an added penalty term to the minimization target term, preventing large movements most of the time, without strict limits.

Thikonov regularization, for example presented by Kaipio and Somersalo, 2006, was chosen. Adding the thikonov regularization results in an L_2 regularization as

$$E = \sum_{\mathbf{c}} ||(I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{c}, \mathbf{p}_0))) + \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}} \Delta\mathbf{p}||_2^2 + ||\Gamma(\mathbf{p}_0 + \Delta\mathbf{p} - \mathbf{p}_s)||_2^2, \tag{3.48}$$

with $\mathbf{p}_s$ being the transformation vector at the start of the iteration, 0 in the first pyramid step or later the output of the previous pyramid stage. Γ describes a regularization factor, regulating the amount of regularization, or how much influence the regularization term has on the overall error measure E. The transformation proposed by the last iteration is given as $\mathbf{p}_0$. Equation 3.48 shows thikonov regularization with the euclidean norm $||...||_2$, but instead for this a weighted norm was chosen, resulting in

$$E = \sum_{\mathbf{c}} ||(I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{c}, \mathbf{p}_0))) + \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}} \Delta\mathbf{p}||_P^2 + ||\mathbf{p}_0 + \Delta\mathbf{p} - \mathbf{p}_s||_Q^2, \quad (3.49)$$

with $Q = \Gamma^T\Gamma$ as a weighting matrix chosen in a way to leave enough flexibility to the optical tracker. For minimization purposes the regularization can be approximated by a Mahalanobis distance, resulting in

$$||\mathbf{p}_0 + \Delta\mathbf{p} - \mathbf{p}_s||_Q^2 = (\mathbf{p}_0 + \Delta\mathbf{p} - \mathbf{p}_s)^T Q (\mathbf{p}_0 + \Delta\mathbf{p} - \mathbf{p}_s). \quad (3.50)$$

Before minimization, some substitutions are performed for easier reading,

$$\Delta\mathbf{x} = \Delta\mathbf{p} \quad (3.51)$$

$$\mathbf{x}_0 = \mathbf{p}_0 \quad (3.52)$$

$$\mathbf{x}_s = \mathbf{p}_s \quad (3.53)$$

$$\mathbf{x} = \mathbf{p}_0 - \mathbf{p}_s \quad (3.54)$$

$$A = \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}} \quad (3.55)$$

$$b = I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{c}, \mathbf{p}_0)), \quad (3.56)$$

leading to a pre-minimization equation, including approximating the regularization by mahalanobis distance, as

$$E = \sum_c \left[(b + A\Delta\mathbf{x})^T P (b + A\Delta\mathbf{x}) + (\mathbf{x} + \Delta\mathbf{x})^T Q (\mathbf{x} + \Delta\mathbf{x})\right]. \quad (3.57)$$

Minimizing for $\Delta\mathbf{x}$

$$0 = \frac{\partial}{\partial \Delta\mathbf{x}} \left[(b + A\Delta\mathbf{x})^T P (b + A\Delta\mathbf{x}) + (\mathbf{x} + \Delta\mathbf{x})^T Q (\mathbf{x} + \Delta\mathbf{x})\right], \quad (3.58)$$

which can be separated into

$$0 = \frac{\partial}{\partial \Delta\mathbf{x}} \left[(b + A\Delta\mathbf{x})^T P (b + A\Delta\mathbf{x})\right] + \frac{\partial}{\partial \Delta\mathbf{x}} \left[(\mathbf{x} + \Delta\mathbf{x})^T Q (\mathbf{x} + \Delta\mathbf{x})\right], \quad (3.59)$$

to be solved assuming the chain-rule of derivation $(uvw)' = u'vw + uv'w + uvw'$ for

$$0 = \left[A^T P(b + A\Delta\mathbf{x}) + 0 + (b + A\Delta\mathbf{x})^T PA\right] + \left[Q(\mathbf{x} + \Delta\mathbf{x}) + 0 + (\mathbf{x} + \Delta\mathbf{x})^T Q\right]. \tag{3.60}$$

Further evaluation leads to

$$0 = [A^T Pb + bPA + Q\mathbf{x} + \mathbf{x}^T Q] + [A^T PA + Q + Q^T P^T A + Q^T]\Delta\mathbf{x}, \tag{3.61}$$

solving for $\Delta\mathbf{x}$ leads to

$$-[A^T PA + Q + Q^T P^T A + Q^T]\Delta\mathbf{x} = [A^T Pb + bPA + Q\mathbf{x} + \mathbf{x}^T Q] \tag{3.62}$$

$$\Delta\mathbf{x} = -[A^T PA + Q + Q^T P^T A + Q^T]^{-1}[A^T Pb + bPA + Q\mathbf{x} + \mathbf{x}^T Q] \tag{3.63}$$

$$\Delta\mathbf{x} = -[A^T PA + Q + Q^T P^T A + Q^T]^{-1}[2A^T Pb + Q\mathbf{x} + \mathbf{x}^T Q]. \tag{3.64}$$

Resubstituting equation 3.51 to equation 3.56 leads to the final, approximate, per iteration of the shift vector

$$\Delta p = -\left[\frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}}^T P \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}} + \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}}^T P^T \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}} + Q + Q^{-1}\right]^{-1} \tag{3.65}$$

$$* \left[\frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}}^T P(I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{p}_0))) + (I_k(\mathbf{c}) - I_{k-1}(\mathbf{x}(\mathbf{p}_0)))P\frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}}\right. \tag{3.66}$$

$$\left. + Q(\mathbf{p}_0 - \mathbf{p}_s)(\mathbf{p}_0 - \mathbf{p}_s)^T Q \frac{\partial I_{k-1_\mathbf{c}}}{\partial \mathbf{p}}\right]. \tag{3.67}$$

When calculating this pixel-wise, one additional optimization can be performed, by weighting each pixels input differently between pixels close to the center of the search region or far away from it.

3.3.3 Achieving Higher Precision Optical Tracking due to Pyramidal Evaluation

The shifts that need to be estimated by the optical tracker in this scenario can vary massively, from 0 pixel movement and no size change for detections far away from the ego vehicle up to 50 or more pixels and noticeable size change from objects close to the sensor. Having a single stage optical tracker to perform such a task makes it very difficult to get precise results at both extremes of the spectrum. Therefore it can be beneficial to stack several optical trackers in a pyramidal way, as suggested by Bouguet et al., 2001. The processing simply works by starting

at a small pyramid level, or a very downscaled image, and iteratively increase the resolution, using the final shift vector $\Delta\mathbf{p}$ from one stage as the initial shift $\mathbf{p}_s$ for the next, as shown in listing 3.2. The resulting estimation can be accurate all scales of possible movement or scale change. As an added benefit, multiple scales can be beneficial for increasing robustness, as sometimes the target object is difficult to locate. The larger scale pyramidal levels can usually still match the area around the target object. While this is not perfectly accurate for the object it is still better than no match at all. For the implementation used here this is slightly extended. At each scale the template part, or the part that is to be tracked, is scaled to a fixed size image patch. This has advantages mainly in the implementation side, as data structures can be heavily optimized. As a result of this, the amount of the image represented in each pyramid stage changes drastically, and the features on which each scale focuses for tracking are changed, as structures become very rough. An example is shown in figure 3.12, with five different scales shown. Each scale compresses so many original pixels, so the result is a 25 x 25 image. This is not the same number nor the exact resolution used in the final implementation, but the process is the same. These patches clearly show how smaller scales blur more and more details, therefore the algorithm can focus on aligning larger shapes, before refining the warp more and more. Resizing into a fixed size has implementation benefits, as fixed size arrays and pre-allocation can be utilized, instead of less optimized, dynamic allocations. Resizing is performed with a nearest neighbour approach. A fixed number of look up positions, 25 for a 25 x 25, is calculated for a given input and scale and at each position a weighted, by distance of new pixel position and original pixels, mean is calculated as final interpolation output

$$x_c = \lfloor x \rfloor, \quad x \in X \tag{3.68}$$
$$x_r = x - x_c, \quad x \in X \tag{3.69}$$
$$y_c = \lfloor y \rfloor, \quad y \in Y \tag{3.70}$$
$$y_r = y - y_c, \quad y \in Y \tag{3.71}$$
$$I_{inter}^{(x_l,y_l)} = (1 - x_r) * I^{(x_c,y_c)} + x_r * I^{(x_c+1,y_c)} \tag{3.72}$$
$$I_{inter}^{(x_l,y_l+1)} = (1 - x_r) * I^{(x_c,y_c+1)} + x_r * I^{(x_c+1,y_c+1)} \tag{3.73}$$
$$I_{inter}^{(x,y)} = (1 - y_r) * I_{inter}^{(x_l,y_l)} + y_r * I_{inter}^{(x_l,y_l+1)}, \tag{3.74}$$

with X as set of new, interpolated pixel positions in x-direction, Y the set of interpolated pixel positions in y-direction, I the original image and $I_{inter}^{(x,y)}$ the resulting, interpolated and possibly scaled image. The relations are shown in figure 3.11.

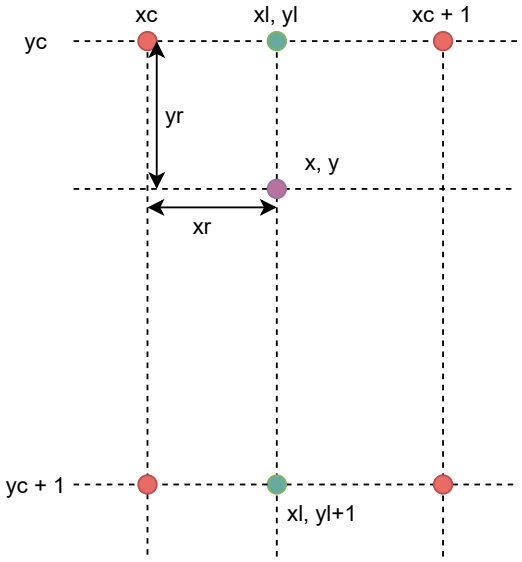
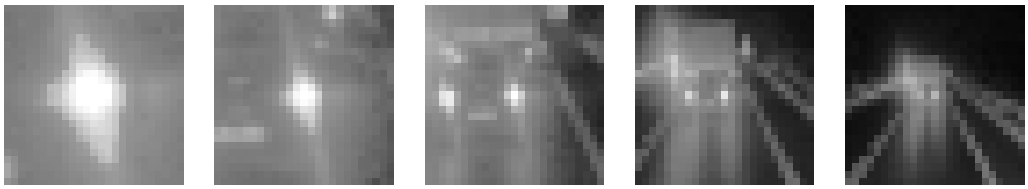


Figure 3.11.: Pixel positions and relations for interpolation. Designed after Press et al., 1992, page. 124, and others



(a) scale = $\frac{1}{1}$ (b) scale = $\frac{1}{2}$ (c) scale = $\frac{1}{4}$ (d) scale = $\frac{1}{8}$ (e) scale = $\frac{1}{16}$

Figure 3.12.: Several pyramid scales, of the same image patch to be tracked by the optical tracker, all scaled to the same 25x25 pixel size

3.3.4 Forward-Backward Verification of Tracking Results

One final thing to improve, or rather verify, results of the optical tracker is forward-backward verification. This can support good quality results, as it prevents the optical tracker from guessing a match on noise, if no good match can be made. If that happens it is extremely unlikely, that the reverse tracker, from I_k to I_{k-1}, will result in the same, or rather exactly inverse, shift as the tracker from I_{k-1} to I_k. After both shift vectors $\mathbf{p}$ and $\mathbf{p}_i$ are calculated similarity between both can be calculated in multiple ways, in this case again the Mahalanobis distance (Mahalanobis, 1936) is used:

$$E = \sqrt{(\mathbf{p} - (-1)\mathbf{p}_i)^T S^{-1} (\mathbf{p} - (-1)\mathbf{p}_i)}, \tag{3.75}$$

with S being a freely chosen covariance matrix. Error E is then thresholded, and only predictions with an E smaller than that threshold are used for updating the

```
pyramid_levels = X
iterations = Y
pyramid_templates = generate_pyramid_image(old_image,
                                           pyramid_levels)
pyramid_images = generate_pyramid_image(new_image,
                                        pyramid_levels)
p_s = [1, 0, 0]
p = [1, 0, 0]

#step through pyramid levels from max to min
for l in pyramid_levels:-1:1:
    p_s = p
    delta_p = [0, 0, 0]
    for i in 1:1:iterations:
        delta_p = perform_klt_iteration(pyramid_templates[l],
                                        pyramid_images[l], p_s, p,
                                        delta_p)
        p += delta_p
    # rescale p according to size difference
    # don't rescale scale component, only translation
    p[1:2] *= pyramid_templates[l+1].size/pyramid_templates[l].size
```

Listing 3.2: Pseudocode for pyramidal processing of optical tracker

GM-PHD. An example of this process is shown in figure 3.13. The green sample is the template from frame k, which is warped into the blue square at time $k+1$. For verification this blue square is warped back into k, the red square. As can be seen, green and red overlap quite well, therefore the example shown would be accepted as prediction. Note that it isn't perfect, but 1 pixel of per direction is 'good enough'.

3.4 Combinatorial use of Optical Tracking and Track Prediction for Stable Object Tracks in Adverse Conditions

With both, the detector and also the optical tracker creating detection inputs for the PHD filter, they can be combined to create more stable and reliable tracks. This way the tracker does not rely on predictions without confirmations for longer periods of time. In cases where real detections are not available the results, or rather detection predictions, from the optical tracker help to guide the tracks from moving too much into erroneous directions.

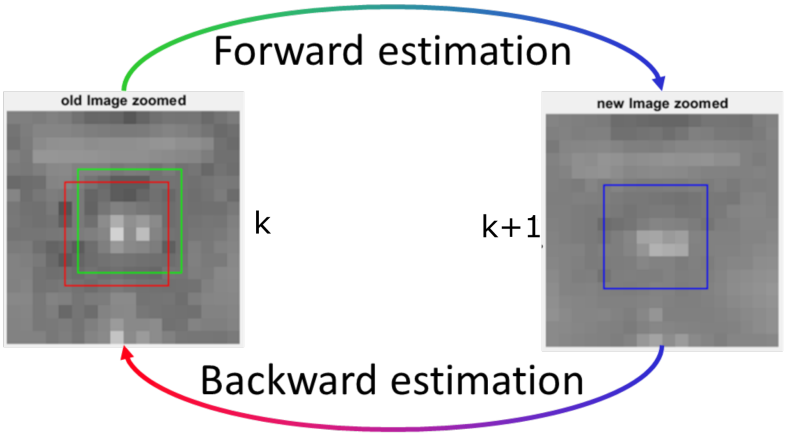


Figure 3.13.: Foward-Backward verification of optical tracking

3.4.1 Organizing and Processing Multiple Types of Detection Inputs

The PHD filter is inherently able to process multiple detections for each track as, finally, each track is just a high probability state in a mixture of potentially thousands of Gaussian probability distributions. Nonetheless, for optimal results, different types of detection or measurement inputs have to be treated differently. In this case this means that detections from the detector are weighted substantially higher than optical tracker predictions. Meaning optical tracker predictions can keep a track alive and stable for some time, but is not meant to substantially alter its state, meaning size, position, velocity. Additionally optical tracker predictions can neither keep a track alive forever nor create new tracks.

Each track has a lifespan value that is increased by 8 for every update with a real, detector detection and not at all for each update with a optical tracker prediction. During each prediction step the lifespan value is reduced by 1. With a lifespan below 3 the track is not included in the output anymore, but will still be used for new predictions, until the lifespan reaches 0 and the track is deleted. This ensures that tracks with low confidence don't pollute the output but can still be reestablished if a new detection is provided, without assigning a new track ID and resetting all track properties like group pairing and covariances. Track losses can be reduced, without adding much noise.

Finally there needs to be a way to start new tracks from detections. Using optical predictions would be too noisy for this, therefore these are excluded and only real

detections can spawn new tracks which may, if they are confirmed in multiple frames or time steps, become fully fledged tracks.

To do this the observation model H_k is used to transfer each detection z_k into a possible new track state x_k. The state covariance P_k is initialized Q, from equation 3.24. The main difference is, that the scale covariance at $Q(5,5) = 0.25$ is set to 50, as for a birthed track the scale confidence is very low, as it is only based on a single detector measurement. Furthermore the full birth covariance is scaled

$$P_k = P_k * s \tag{3.76}$$

with

$$s = 0.5 + 0.5 * \frac{size}{25}. \tag{3.77}$$

25 is the base size to which P_k is initialized but this way a larger birthed object will have a higher covariance, as it is, on average, closer to the ego vehicle leading to larger variances from frame to frame. A birthed track is simply predicted into the next frame by keeping its introductory weight and updating state and covariance as previously shown in equation 3.10 and equation 3.11. New tracks only get a health value and ID after receiving their first update and the first pruning cycle.

3.5 Modelling Car Column Movement by Swarm Movement

While tracking each vehicle light separately is possible, greater consistency can be achieved with group tracking multiple light sources as a group. This can be the pair of head or tail lights on a single vehicle or also spread across multiple vehicles. The main reason to do it is preventing single outlier detections from dragging a track into a direction it should not go, as described in section 3.5.1.

To do this, tracks of multiple different sizes, exact positions and movements need to be clustered into different groups, depicting parts of the scene, like a group of cars in oncoming traffic on a highway. Several clustering algorithms are available for this, most well known likely k-means (MacQueen et al., 1967) and DBSCAN. K-means clustering cannot be used, as the number of cluster centers is no prior knowledge and changes on a scene by scene basis. DBSCAN, developed by Ester et al., 1996, might work, but is not, in its native implementation, able to handle clusters of different densities. One extension of DBSCAN, that works with variable densities

across clusters, would be OPTICS, published by Ankerst et al., 1999. Details on OPTICS are presented in section 3.5.2.

3.5.1 Advantages of Group Tracking for Vehicle Light Tracking

When looking at vehicle light sources, usually at least two are present that can be paired easily, left and right lights of the same vehicle. Different types of vehicles, like trucks and bikes, might have more or less light sources, therefore this algorithm needs to be able to work with a flexible number of light sources.

These will move very similar if not in the same way, just slightly offset from one another, therefore approximating their movement together can smooth the tracks by averaging over the detection errors for each of them.

Looking at multi-lane highways this does not have to be limited to one vehicle at a time. Both oncoming traffic and preceding vehicles will, except for a few outliers at higher/lower speeds, move in a very similar and group like manner. As explained previously this can help further eliminating outlier behaviour.

3.5.2 OPTICS for Clustering Light Sources

"Ordering Points To Identify the Clustering Structure (OPTICS)" by Ankerst et al., 1999 is an extension of the popular "Density Based Spatial Clustering of Applications with Noise (DBSCAN)", Ester et al., 1996, developed to be able to cluster data with irregular densities inside the clusters.

DBSCAN

DBSCAN is a clustering method developed by Ester et al., 1996, able to cluster noisy data into clusters of similar density without requiring a pre-selected number of cluster centers. This works by iterating through all points of the data set, collecting points inside a neighbourhood range and extending clusters iteratively.

Listing 3.3 shows an overview of the DBSCAN algorithm as it could be implemented in python, following pseudocode provided in the original paper by Ester et al., 1996. DBSCAN iterates over all points that don't yet belong to a cluster and then checks if it can be a core point, meaning possible cluster center, by testing if enough points are inside of range *eps* around it, as defined by *min_pts*. Each point, or *seed*, around

```
def region_query(all_points, point, eps):
    # Returns all points in all_points which are closer than
    # distance threshold eps compared to point.
    # distance() can be any distance measure, which is why
    # it is not detailed here
    close_index = [distance(point, p) <= eps for p in all_points]
    return all_points[close_index]

def expand_cluster(all_points, point, cluster_id, eps, min_pts):
    seeds = region_query(all_points, point, eps)

    if len(seeds) < min_pts:
        return False % No core point = no cluster center
    else:
        # enough points are close to point so it is a core point
        for p in seeds:
            # points in range of point are added to cluster
            p.cluster_id = cluster_id
        seeds.remove(point)

        while not empty(seeds):
            # While there are seeds left, check if
            # points are close enough to them to belong to
            # the cluster
            cur_seed = seeds[0]
            res = region_query(all_points, cur_seed, eps)

            if len(res) >= min_pts:
                for r in res:
                    if r.cluster_id in ['UNCLASSIFIED', 'NOISE']:
                        # if r is not in a cluster yet or noise
                        if r.cluster_id == 'UNCLASSIFIED':
                            seeds.append(r)
                        r.cluster_id = cluster_id
            seeds.remove(cur_seed)
        return True

def dbscan(all_points, eps, min_pts):
    for point in all_points:
        point.cluster_id = 'UNCLASSIFIED'
    cluster_id = 1
    for point in all_points:
        if point.cluster_id == 'UNCLASSIFIED':
            if expand_cluster(all_points, point, cluster_id,
                              eps, min_pts):
                cluster_id += 1
```

Listing 3.3: DBSCAN as described in pseudocode by Ester et al., 1996 transferred to python code

this core point is then checked to be a possible core point as well, for extending the cluster. Once all *seeds* are tested, the cluster is extended to its possible maximum, and the remainder of unclustered points is further processed, until all points were tested for being core points. This is very easy to implement and very flexible to cluster sizes, data types and distance measures. The major problem, and reason why it can't be used here, is that *eps* is a fixed value and the same for all possible clusters. For vehicle lights this is an issue, as they are not always distributed equally across the whole image and may vary in density. Lights that are far away will have a very high density and only need a small *eps* to cluster properly, while lights that are a lot closer to the ego vehicle need a much larger *eps* to cluster. Using this large *eps* over the whole scene would lead to errors far away. Besides the singular *eps* value the second tuning parameter for DBSCAN is *min_pts* or the minimum number of points per cluster. This is why the DBSCAN extension OPTICS as described in the following section 3.5.2 is used, as it works properly with clusters of varying densities.

OPTICS

OPTICS is, roughly said, an extension of the DBSCAN algorithm to work with clusters of varying density. This works by not evaluating single distances but rather record core distances, the distance, if any, below *eps* at which a point has *min_pts* points in its neighbourhood and reachability distances, the shortest distance at which a point is directly reachable from a core point. These values are calculated as shown in listing 3.4. The resulting reachability plot looks, for example, as shown in figure 3.14. Now one simply needs to extract clusters at desired densities from the reachability plot. As can be seen in the shown figure, clear valleys and spikes can be extracted and, depending on the desired density reachability, a number of clusters can be built from that information. To extract clusters from the reachability plot clustering algorithms like DBSCAN, or even simpler methods of falling/rising edges, can be utilized.

3.5.3 Virtual Leader-Follower vs. Cucker-Smale Flocking Model

Clark and Godsill, 2007 first published group tracking in combination with a PHD filter. In their paper they explain two different methods of group predictions, one being the virtual leader-follower model, the other being the Cucker-Smale flocking model. The main difference between both is how the individual object motion is

```
def region_query(all_objects, obj, eps):
    # Returns all objects in all_objects which are closer than
    # distance threshold eps compared to object.
    # distance() can be any distance measure, which is why
    # it is not detailed here
    point_dist = [distance(obj, p) for p in all_objects]
    close_index = [point_dist[i] <= eps for
                                    i in range(len(all_objects))]
    return all_objects[close_index], point_dist[close_index]

def get_core_dist(dists, eps, min_pts):
    # returns minimum distance for which min_pts would
    # be below distance threshold
    # returns 'UNDEFINED' if less than min_pts are close
    return core_dist

def update(seed_list, neighbours, center_object):
    core_dist = center_object.core_dist
    for obj in neighbours:
        if not obj.processed:
            new_reach_dist = max([core_dist,
                                  distance(center_obj, obj)]
        if obj.reach_dist == 'UNDEFINED':
            # obj not yet reachable
            obj.reach_dist = new_reach_dist
            seed_list.insert(obj, new_reach_dist)
        else:
            # obj is already reachable
            if new_reach_dist < obj.reach_dist:
                # new reachability dist is lower
                # move up in seed_list
                obj.reach_dist = new_reach_dist
                seed_list.update_position(obj, new_reach_dist)
    return seed_list

def expand_cluster_order(all_objects, obj, eps,
                         min_pts, output_list):
    neighbours, dists = region_query(all_objects, obj, eps)
    obj.processed = True
    obj.reach_dist = 'UNDEFINED'
    obj.core_dist = get_core_dist(dists, eps, min_pts)
    output_list.insert(obj)
    seed_list = []
    if obj.core_dist ~= 'UNDEFINED':
        seed_list = update(seed_list, neighbours, obj)

        while not empty(seed_list):
            # get + remove first seed in seed_list
            seed = seed_list.pop(0)
            neighbours, dists = region_query(all_objects, seed, eps)
            seed.processed = True
            seed.core_dist = get_core_dist(dists, eps, min_pts)
            output_list.insert(seed)
            if seed.core_dist ~= 'UNDEFINED':
                seed_list = update(seed_list, neighbours, seed)

def optics(all_objects, eps, min_pts, output_list):
    ordered_list = []
    for obj in all_objects:
        if not obj.processed:
            expand_cluster_order(all_objects, obj, eps,
                                 min_pts, ordered_list)
```

Listing 3.4: OPTICS, code in python, following pseudo code from Ankerst et al., 1999

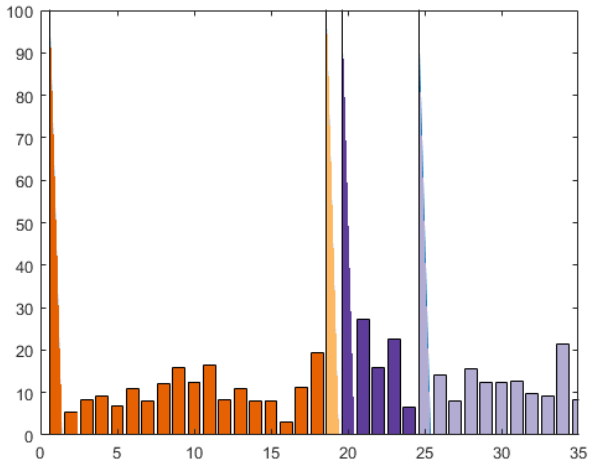


Figure 3.14.: Example of a reachability plot as created by OPTICS

predicted. For the virtual leader-follower model it is assumed, that the parts of a group all move the same as the virtual leader which is averaged over all group member states $\mathbf{x}_k^{(i)}$

$$\bar{\mathbf{x}}_k := \frac{1}{\hat{N}_k} \sum_{i=1}^{\hat{N}_k} \mathbf{x}_k^{(i)}, \tag{3.78}$$

with $\hat{N}_k$ as number of members in a group. Leading to each individuals movement in a constant velocity model being predicted as the averaged virtual leader movement, resulting in a position prediction as

$$p_{k|k-1}^{(i)} = p_{k-1}^{(i)} + \Delta t \bar{v}_{k-1}, \tag{3.79}$$

with $\bar{v}_{k-1}$ being the leader velocity as extracted from $\bar{x}_{k-1}$, Δt being the time difference between k and $k-1$. An example of such a group and virtual leader is shown in figure 3.15. It can easily be seen, how the virtual leader, red, averages over all the individuals and averages the noise to produce a consistent motion.

The Cucker-Smale flocking model works different as only more or less strong influence in the group is assumed and not equal motion. Cucker and Smale developed this model for predicting bird flock movement (Cucker and Smale, 2007).

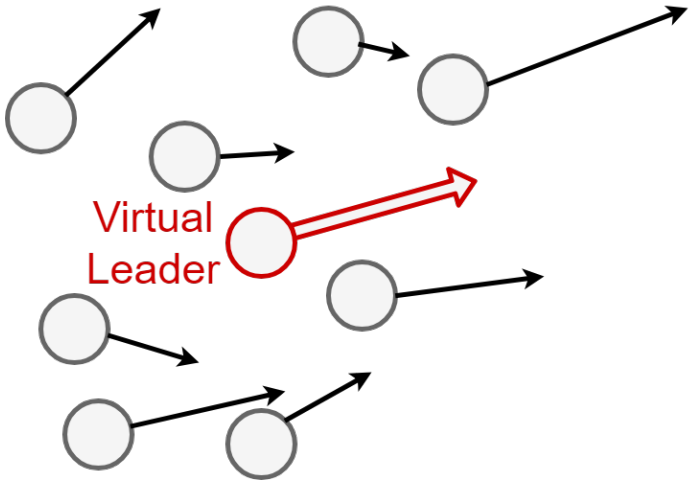


Figure 3.15.: Theory behind virtual leader model

The main difference here is the prediction of the new velocity, resulting in

$$v_{k|k-1}^{(i)} = v_{k-1}^{(i)} + \sum_{j=1}^{\hat{N}_{k-1}} a_{k|k-1}^{(i,j)} (v_{k-1}^{(j)} - v_{k-1}^{(i)}) \tag{3.80}$$

$$a_{k|k-1}^{(i,j)} = \frac{H}{(1 + ||p_{k-1}^{(i)} - p_{k-1}^{(j)}||^2)^{\beta}}, \tag{3.81}$$

with $H > 0$ and $\beta \geq 0$, leading to a slow alignment of velocities inside of the group.

For this work, the virtual leader-follower model was chosen because of the better ability to instantly integrate new tracks into groups. With the Cucker-Smale model, new tracks slowly, depending on H and β, align to the group, which is not desired here.

3.5.4 Predicting Group Movement

For group tracking in this application, the virtual leader-follower model was chosen. This results in an easy implementation and models the desired behaviour of groups better than the Cucker-Smale model.

For this all of the track outputs of the PHD filter are clustered together using the OPTICS algorithm, as described in section 3.5.2. The min_pts value chosen is 1, because of easier handling, every track shall be part of a cluster, even if it is the only one in it. Maximum reachability distance eps is chosen manually at a fixed value,

and default, or as written in listing 3.4, 'UNDEFINED' reachability distance and core distance are selected as 10^{99}, or for all extends and purposes, infinity.

As distance measure for the OPTICS clustering, the Mahalanobis distance, published by Mahalanobis, 1936, between the candidate tracks is used. It is a distance measure between either a value and a distribution or two values of the same distribution

$$d(\tilde{\mathbf{m}}_{k-1}^{(i)}, \tilde{\mathbf{m}}_{k-1}^{(j)}) = \sqrt{(\tilde{\mathbf{m}}_{k-1}^{(i)} - \tilde{\mathbf{m}}_{k-1}^{(j)})^T (P_{k-1}^{(i,j)})^{-1} (\tilde{\mathbf{m}}_{k-1}^{(i)} - \tilde{\mathbf{m}}_{k-1}^{(j)})}. \tag{3.82}$$

Between two tracks the covariance $P_{k-1}^{(i,j)} = \frac{P_{k-1}^{i} + P_{k-1}^{j}}{2}$ is averaged to approximate the covariance of the distribution containing both tracks.

An exemplary result of clustering is shown in figure 3.14. As can be seen there are four clusters, which are extracted by assigning all tracks between two reachability spikes to one cluster.

For contingency between frames, new clusters are matched with old clusters, by calculating the percentage overlap between track IDs. This is done to ensure the same cluster ID between frames. As mentioned before, even tracks that don't belong to any cluster are treated as clusters here. For all clusters the average state $\bar{\mathbf{m}}_{k}^{(i)}$ and covariance $\bar{P}_{k-1}^{(i)}$ is calculated

$$\bar{\mathbf{m}}_{k-1}^{(i)} = \frac{1}{N^{(i)}} \sum_{n=1}^{N^{(i)}} \tilde{\mathbf{m}}_{k-1}^{(n)} \tag{3.83}$$

$$\bar{P}_{k-1}^{(i)} = \frac{1}{N^{(i)}} \sum_{n=1}^{N^{(i)}} P_{k-1}^{(n)} \tag{3.84}$$

as average over all states $\tilde{\mathbf{m}}_k$ and covariances P_{k-1}. In the prediction step of the next PHD iteration, velocity and acceleration of individual states are replaced with their cluster or group velocity and accelerations.

3.6 Evaluation of Advanced Light Source Tracking Components

Evaluation on the advantages of the different components presented in this chapter is mainly performed on a small, private data set from Aptiv. Few, if any, public data sets are available unfortunately. While BDD100K (Yu et al., 2018) exists and includes night time recordings, the detector used is, as it is designed for one specific product, not flexible enough to work with this kind of data. They also don't provide detection outputs for vehicle lights. The MOTChallenge (Milan et al., 2016 and Leal-Taixé et al., 2015) exists, but is focused on very different tracking problems. The light sources tracked here are usually very small and don't move erratically. They also are of a very specific visual profile and are simply less noisy than their data. Adapting this tracker to work on the MOTChallenge would be very time consuming and would likely still not achieve great results. To still provide limited comparability with external trackers a small ablation study between this tracker and a baseline GM-PHD is provided.

General Evaluation on internal data

As data was annotated manually for this evaluation, only a limited number of videos were available. The results are shown in table 3.1. Results are provided in terms of the 'multi object tracking accuracy (MOTA)' and 'multi object tracking precision (MOTP)' measures by Bernardin and Stiefelhagen, 2008 and also the TP-rate. The MOTP is provided in euclidean pixel distance, therefore lower is better. As can be seen, the MOTA is generally over 0.8, with only one video dropping slightly bellow, while the MOTP is only at slightly above 1.5 pixels on average. The very important TP-rate always stays above 0.96, resulting in very few missed objects. This is very important in a system as shown here, as a missed vehicle in oncoming traffic is much worse than being slightly offset or having a few too many false positive detections. A missed vehicle might immediately lead to a driver being blinded with high risk of accidents occurring.

Ablation Studies

As comparisons on public data sets were not easily performable, a small ablation study is performed to quantify the advantages of the novelties introduced here. The

Video	Frames	Objects	MOTA↑	MOTP↓	TP-Rate↑
1	190	268	0,854	1,546	0,97
2	267	435	0,839	1,512	0,968
3	287	1124	0,762	1,283	0,961
4	140	235	0,83	1,89	0,97
5	226	698	0,815	1,447	0,961
overall	1110	2760	0,802	1,535	0,964

Table 3.1.: Results of the full tracker on selected Aptiv videos

results are shown in table 3.2. As can be seen the tracker, utilizing all modifications, performs best or close to best in multiple measurements. While it performs noticeably worse in terms of FP, it also performs very significantly better in terms of the TP-rate, which is much more important in this application. The increase in FP is easily explained by the large bridging capabilities of the tracker, keeping tracks alive up to 8 frames after no detections are provided. Worst case a single track can produce a FP for 7 consecutive frames. Both the optical tracker and also the group tracking modification don't show particularly large benefits by their own. Adding the group tracker leads to less false positives compared to a baseline GM-PHD, while the optical components reduces the number of false negatives and therefore increases the TP-rate. The combined tracker performs by far the best in almost all categories. Further studies on the effects of single components and the tuning of single components might be beneficial, but were not performed here, beyond the usual tuning of the tracker presented in section 3.2.3. One example of the tracker output is shown in figure 3.16. It can clearly be seen, how tracks moving in a similar way and of close proximity are grouped together for more stable tracking. It can also be seen, how all relevant objects in the scene are tracked. The closest vehicle in front of the camera shows how it is not always possible to suppress all erroneous detections and tracks, as the lighting of the number plate holder is picked up and also tracked. This is no major issue though, as it does not change the blocked area for the matrix light source supported by this system.

Tracker	TP ↑	FP ↓	FN↓	MOTA ↑	MOTP↓	TP-Rate ↑
Full Tracker	**2660**	446	**100**	0.802	**1.535**	**0.964**
GM-PHD	2431	117	329	**0.838**	1.548	0.872
GM-PHD + Group	2383	**99**	377	0.827	1.67	0.857
GM-PHD + Optical	2463	249	297	0.802	1.64	0.895

Table 3.2.: Ablation study on novelties

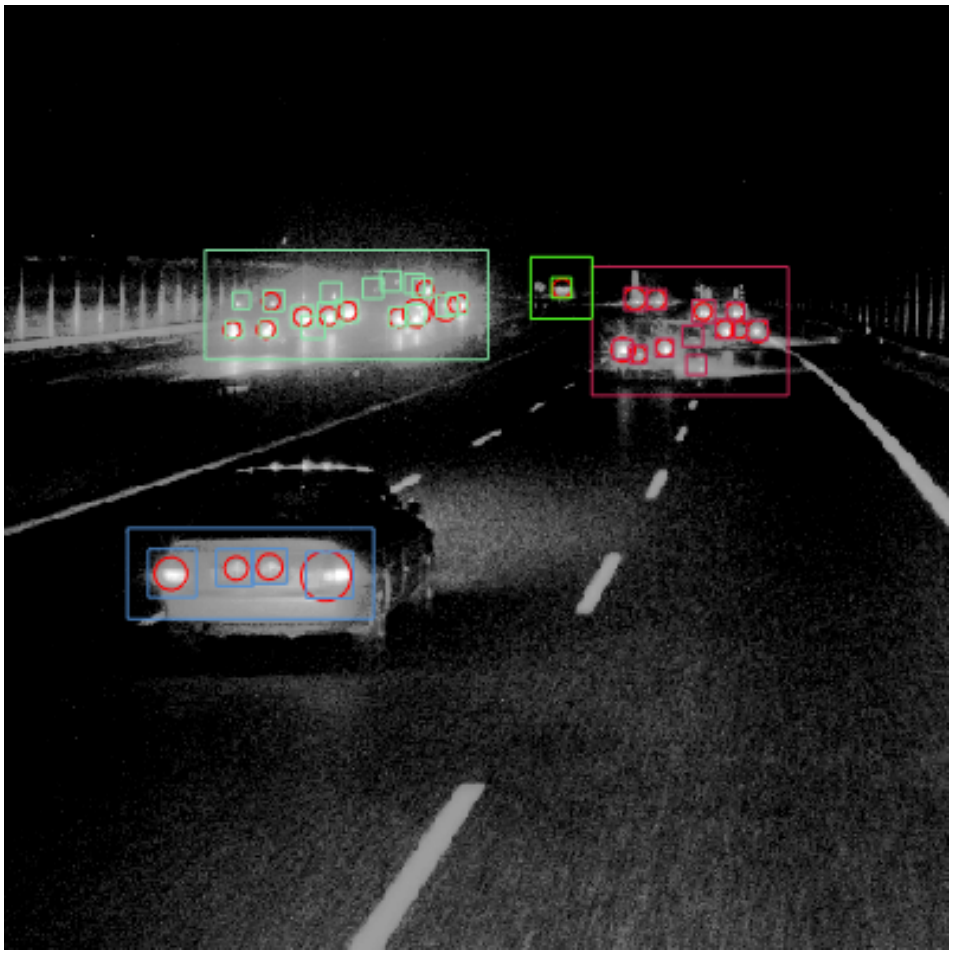


Figure 3.16.: Example result of tracking algorithm. Red circles: detection outputs, colored boxes(small): tracks, colored boxes(large): groups. Image from Alsfasser et al., 2019

4 Improving Lidar Object Detection Algorithms

In this chapter advancements on lidar object detection will be presented in multiple areas. Advancements in data structure, feature generation and training data augmentation will be presented. First, in section 4.1, an overview over different processing methods for point clouds is given. Section 4.2 focuses on structure based algorithms, as these are where most advancements presented here occur. Improvements in feature generation will be shown in section 4.3. Data augmentation is a very important step during training, to improve the networks ability to generalize results towards data not actually present in the training set. New approaches for this are presented in section 4.4.

4.1 A Summary of State of the Art Lidar Object Detection Algorithms

Over the last few years a huge amount of different kinds of object detection algorithms for lidar point clouds were published. These can generally be divided into three different categories, which will be summarized in this section. Point clouds can either be processed in a structured way, like a grid as shown in section 4.1.1. A, usually, more flexible, way to process point clouds would be point wise, as described in section 4.1.2. The final type of algorithms would be fusion based algorithms, in which camera images and lidar point clouds are processed jointly, either by using the image for pre-selection of areas or as additional network input. These will be introduced in section 4.1.3.

4.1.1 Structure-Based Algorithms

Structure based algorithms are algorithms like the original VoxelNet (Zhou and Tuzel, 2018) or its evolutions SECOND (Yan et al., 2018) and PointPillars (Lang et al.,

2019). These structure based algorithms are taking the point clouds and process them in the same, or a very similar, way as one would process a an image, examples shown in section 2.2.2. Most focus in this section will be put on VoxelNet (Zhou and Tuzel, 2018), as it is one of the first and most well known networks. Their main contribution, which is also used in a large amount of other publications, is voxel feature encoding (VFE). This describes the extraction of features from voxelized point clouds, which is also used in the optimized algorithm, described starting in section 4.2.2. An alternative method, focused on ease of use and processing speed is presented by Hahn et al., 2020, where they project features into 2D images to process them with a relatively small and very fast network. They are more focused on segmentation unfortunately and also show, that generalization to some classes does not work as well, as the main focus classes.

Voxel Feature Encoding

The voxel feature encoding introduced by Zhou and Tuzel, 2018 is the feature encoding foundation of many popular lidar object detection algorithms, such as SECOND (Yan et al., 2018) or PointPillars (Lang et al., 2019). The idea of the original VFE is to generate one feature vector for each voxel or grid cell. First, a fixed number of data points is sampled per cell, usually 35 or 50. If less than the selected amount are available the selection is padded with zero values. The setup is shown in figure 4.1. In the first step a fully connected network layer is used on each data point, estimating a feature vector, resulting in, for example, 35 feature vectors for the data cell. These are point-wise features, containing no input of the cell neighbourhood. To improve this, element-wise max pooling is utilized on the 35 feature vectors, resulting in a new, cell-wise feature vector. This cell feature vector is concatenated with each of the point-wise feature vectors, resulting in a feature vector containing point-wise and local features. The result of each VFE layer is therefore one feature vector for each data point inside a given cell. Multiple of these layers can be stacked, each again increasing the size of the feature vector and using both point-wise and localized features. Depending on the desired output of the feature extraction, a final max pooling might be added to consolidate all point-wise feature vectors into one final cell-wise vector.

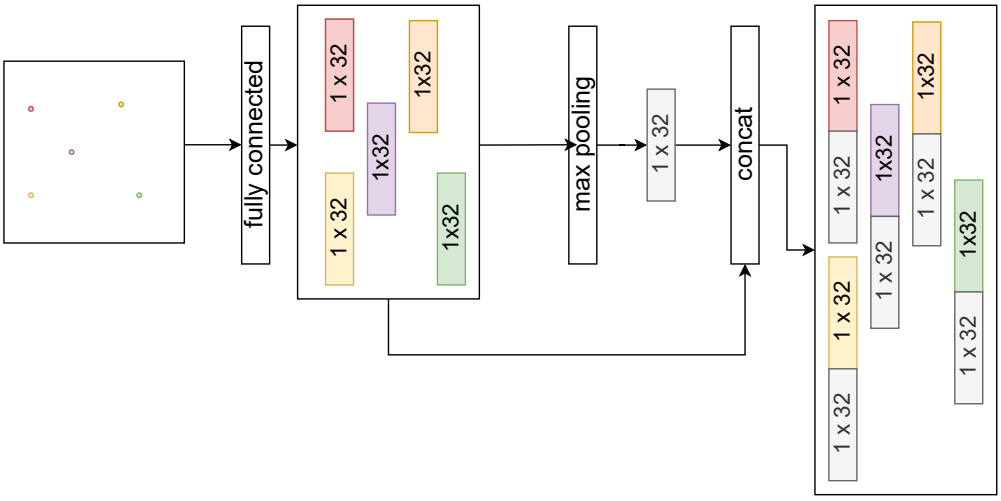


Figure 4.1.: Example of a VFE Layer. This might be stacked for multiple layers, if not it would be followed by an additional max pooling to reduce the cell to 1 1 x 64 vector

4.1.2 Point-Wise Algorithms

Same as structure based algorithms, point-wise algorithms are the basis of a large amount of advanced networks. The baseline architectures of PointNet and PointNet++ were published by (Qi et al., 2017a) and (Qi et al., 2017b) in 2017 and 2018 respectively, with PointNet++ being a strict improvement on PointNet, by stacking several PointNet processing layers at different scales to capture a combination of local and global features, while the original PointNet is fixed to one scale and can only capture features at that scale.

4.1.3 Fusion Algorithms

A multitude of different fusion based algorithms are available. Most are combining one or more camera images with the lidar point cloud. One of the earlier examples of this is published by Qi et al., 2018 in "Frustum pointnets for 3D object detection from rgb-d data". These are, in part, the authors that developed the PointNet and PointNet++ architecture and now utilize it for a powerful object detection algorithm. They use an object detection algorithm on images to predict positions of relevant objects in the image. These images, and regions, are used to cut 3D frustums from the point cloud. These are projections of an angle range found by projecting the image into 3D, with the distance as a free variable. The result is a trapezoid shaped cutout from the point cloud. The data points inside of this cutout are processed

by a PointNet network to find and refine 3D bounding boxes for relevant objects. The authors show impressive results, but at extremely high runtime cost. A second, similar approach is presented in RoarNet (Shin et al., 2019), which uses a similar toolchain, but trust their 3D estimation from image detections more, to cut more precise regions from the point cloud. They project the image bounding box position, dilate the area and add some additional areas in front and behind the object. Inside of these regions they again refine bounding box predictions.

4.2 Exploring Structured Approaches to Point Cloud Processing

When looking at developing a new object detection algorithm, it is important to choose the right basis for the network to be developed. As described previously, there are roughly three categories of algorithms for object detection in point clouds, each having their own strengths and weaknesses. As single sensor use, without having a camera available, was desired for this network, and pure PointNet implementations did not work well enough, the choice was made for a grid based algorithm. As shown in section 4.1.1, several good quality algorithms of this type were published over the last few years, leading to a large amount of published knowledge and experiments which could be utilized. For first experiments, a standard network, as proposed by VoxelNet, PointPillars and SECOND was chosen. Looking at the results of this, good results could be achieved, but at too much processing time and too much memory usage on the graphics processing unit used to run the network, see section 4.2.1. After changing to a spherical grid a large improvement in runtime and memory usage was achieved but at the cost of detection performance. Details are provided in section 4.2.2. To support the network and regain the lost performance additional, hand crafted, features are integrated with the network, as will be presented in section 4.3.

4.2.1 Issues and Options With the Classical Square Grid Based Approach

When processing lidar point clouds in a large, uniform 3D grid, a very large number of grid cells will be empty, because lidar point clouds are not uniformly distributed. When analyzing a point cloud statistically, a large percentage of points is closer than

20 m. This is grounded in the nature of how a lidar records data. A typical sensor like the Velodyne HDL64 has an horizontal angular resolution of 0.08° (Velodyne Lidar Inc., 2021a), therefore at a distance of 50 m, a single horizontal angle step already covers a distance of ≈ 7 cm, while at 5 m it only covers ≈ 0.7 cm. Additionally a typical point cloud created by a Velodyne HDL64 contains roughly 120,000 data points. When looking at a full, 3D grid with 20 cm edge length for each cell, 100 m side length in x- and y-direction, and 4 m height, the resulting grid contains $N = \frac{100}{0.2} * \frac{100}{0.2} * \frac{4}{0.2} = 5,000,000$ cells. At 120,000 data points, even if every cell would only contain 1 data point, the sparsity index of the grid would still be at $1 - \frac{120,000}{5,000,000} = 0.976$, therefore 97.6% of cells would be empty. More realistically only 25,000 cells are occupied, leading to a sparsity of $1 - \frac{25,000}{5,000,000} = 0.995$, or 99.5% of cells being empty. Naively processing this with 3D convolutions, as VoxelNet (Zhou and Tuzel, 2018) does, wastes a large amount of processing time. This estimation comes from the voxelization process, in which data points are discretized with the final grid size and resolution, as

$$x_v = \left| \frac{x - g_{x,min}}{r_x} \right|, \tag{4.1}$$

with x the original point position, $g_{x,min}$ the minimum x-position the grid will cover, and r_x the grid resolution in x-direction. SECOND (Yan et al., 2018) reduces this waste by implementing sparse convolutions, meaning convolutions that are only processed at occupied grid cells. This works well, but requires a more complex processing setup and additional libraries that can reduce interchangeability of the generated code. Another way to reduce the data amount is explored in PointPillars (Lang et al., 2019), where they remove the height component of the grid, reducing the amount of grid cells in the aforementioned grid to only $N = \frac{100}{0.2} * \frac{100}{0.2} = 250,000$. The number of occupied cells also goes down, but still, the sparsity is heavily reduced to realistically 90% for 25.000 occupied cells and at minimum 52% for 120.000 occupied cells.

4.2.2 Advantages & Disadvantages of Sphere Based Grids

Spherical grids, which are designed to closer follow the structure of the data generated by the lidar sensors, can be a good substitute for square grids. This is based on the general functionality of lidar sensors. A lidar sensor measures a new data point at fixed angular intervals, both in height and in azimuth. The resulting point cloud therefore has, theoretically, constant angular density and variable euclidean density. A cartesian grid as used in section 4.1.1 models a constant euclidean density and therefore suffers of the issues described in section 4.2.1. The spherical model follows the constant angular distribution more closely, with cells spanning 2°

both in azimuth, as $\Delta\varphi$, and height, $\Delta\vartheta$. For the size in range/radius direction there are multiple choices. If a constant cell size is chosen, the cells aspect ration would constantly change, which increases the variation between cells even further, and subsequently likely decreases the generalization abilities of the network. A better way is scaling the size in range direction according to the distance from the sensor and the angular size of the cells. Doing this leads to close to square cells at each range. To limit errors at each end of the cells, the length of the cell is aligned with the width at the middle of the cell. As first step, the ratio between width and distance of the cell boundary from the origin is calculated as

$$s = 2 * \tan\left(\frac{\Delta\varphi}{2}\right) \tag{4.2}$$

with

$$\Delta\varphi = 2°. \tag{4.3}$$

Now, as explained, the cells are scaled so the width at the middle equals the length of the cell. This is given by

$$r_{i+1} - r_i = s\frac{r_{i+1} + r_i}{2} \tag{4.4}$$

with

$$r_{i+1} - r_i = \text{Length of the given cell} \tag{4.5}$$

$$\frac{r_{i+1} + r_i}{2} = \text{Distance of cell center from origin.} \tag{4.6}$$

Solving for r_{i+1}, or the furthest extend of the cell, results in

$$r_{i+1} - r_i = s\frac{r_{i+1} + r_i}{2} \tag{4.7}$$

$$\Leftrightarrow r_{i+1} - r_i = \frac{sr_{i+1}}{2} + \frac{sr_i}{2} \tag{4.8}$$

$$\Leftrightarrow r_{i+1} - \frac{sr_{i+1}}{2} = \frac{sr_i}{2} - r_i \tag{4.9}$$

$$\Leftrightarrow r_{i+1}(2 - s) = r_i(2 + s) \tag{4.10}$$

$$\Rightarrow r_{i+1} = r_i\frac{(2 + s)}{(2 - s)}. \tag{4.11}$$

The result is a recursive scaling between r_{i+1} and the previous cell. This can be exploited to get the furthest extend of a given cell as

$$r_i = \left(\frac{(2 + s)}{(2 - s)}\right)^i r_0, \tag{4.12}$$

using the index i as power of the scaling between cells which are i indices apart and $r_0 = 1.5$ m as start of the closest cell. A sketch of these relations is given by figure 4.2.

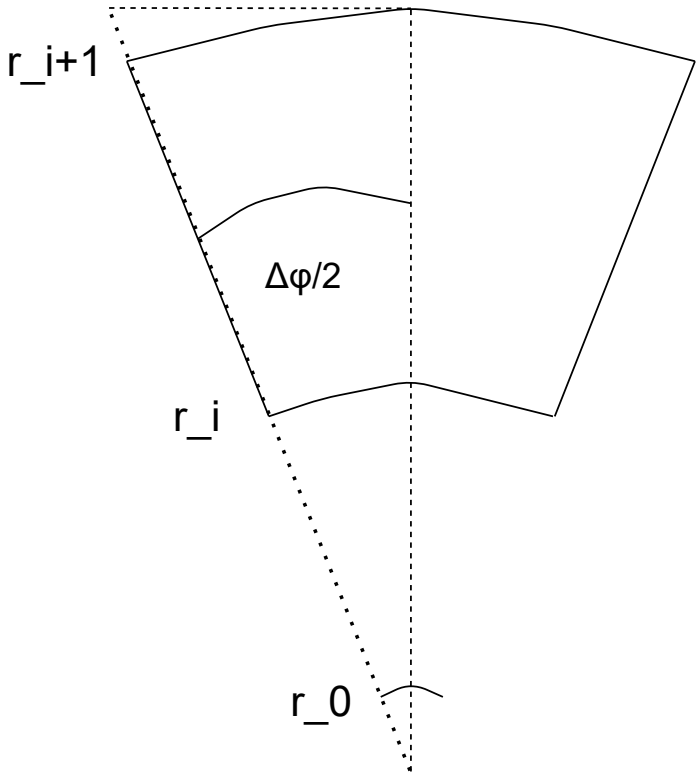


Figure 4.2.: Relations of how cells are scaled

The resulting grid would cover a chosen angular range in front of the sensor, up to a maximum distance determined by the number of cells in range direction, r_N. With the given values and $r_N = 112$ this grid would cover up to a range of 75 m. A cell like this is shown, schematically, in figure 4.3. Scaling the grid like this leads to every cell covering a similar area of the sensor detection field. This can roughly normalize the number of points per cell, although especially at far ranges cells might still only contain few points, despite being 8 or more cubic meters of volume. The causes of this are twofold. First, at large distances even small errors might lead to large irregularities in the measurement, and, secondly, occlusions closer to the sensor block the laser rays from reaching large distances. Grid plots from top and side perspective are shown in figure 4.4 and figure 4.5. It can be seen how grid cells are kept relatively square by scaling their extension in r direction. The result of using the spherical grid in a basic convolutional neural network is shown in figure 4.6. The left part of the figure shows the projection of a spherical grid into a cartesian representation. It is not easy to see, but the middle part of the image, vertically,

shows the road surface, which is displayed much wider on the left, close to the sensor, while it heavily narrows on the right, further away from the vehicle. This is the result from each cell being shown as one pixel. In close range each cell covers a much smaller area, therefore each pixel covers a smaller area, and a 10 m wide road consists of many more pixels than on the right. The right part of the figure shows a similar scene in the same projection, but from a cartesian grid. Again each grid cell is represented by one pixel. Comparing both, it can be seen how in this projection, the road (highlighted area in the vertical middle of the image) is of a consistent size. This is much easier for the neural network to understand. The same is shown later, when looking at occupancy grids of both grid types. It can also be seen, how the leftmost 10-15% of the point cloud, for the spherical grid, don't contain any information, as this area mainly covers the dead zone around the sensor. Furthermore the resolution of the grid is lower and while it could easily be increased, for example only covering $\Delta\varphi = 1°$, this would also heavily increase the amount of 'wasted' grid cells in the dead zone and provide little actual benefit, as tested in multiple experiments. As this could often be recognized early in training, no special evaluation was performed. As one last difference, the spherical grid is only employed in front of the vehicle, as this is the most interesting area for a live object detection system, while the cartesian grid covers the full 360° area around the sensor. In figure 4.6 the vehicle is therefore located at the very left (outside the drawn area) of the spherical grid and in the middle (dark blue, circular deadzone) of the cartesian grid.

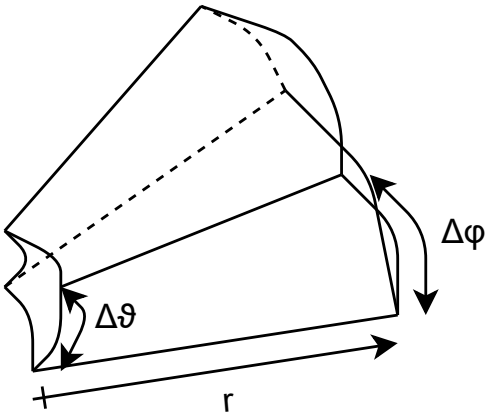


Figure 4.3.: Schematic view of cell

One disadvantage of a grid like this is, that the cell size scaling at the extremes of the grid is very different, from a typical grid. For the given values, 75 of 112 rows of cells, distance-wise, are closer than 20 m, 55 even closer than 10 m. This can cut even small objects into a large number of cells, meaning the network needs

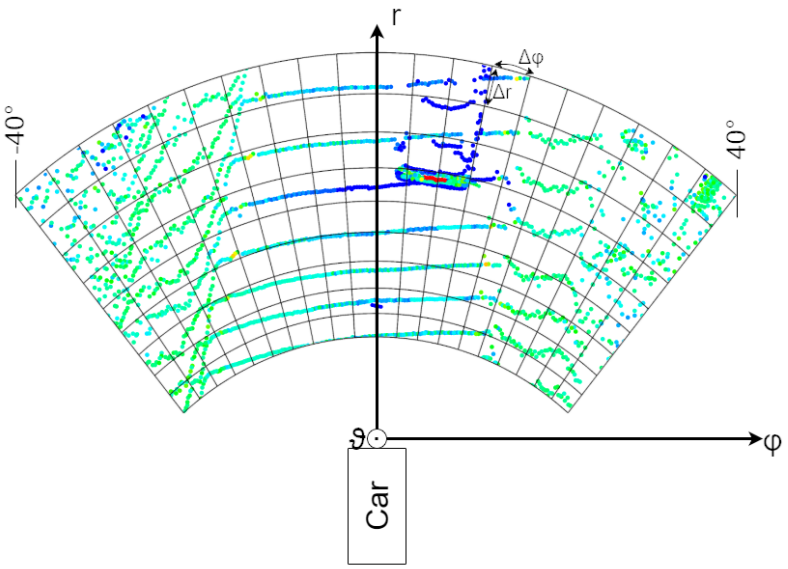


Figure 4.4.: Schematic top view of spherical grid and point cloud. Grid resolution not to scale with point cloud

to be designed to merge results from multiple cells into one bounding box. If the nature of the cells, being almost square, should be kept, this is difficult to change. One has to adjust the angular resolution $\Delta\varphi$, reducing the resolution in multiple directions. It can be difficult for the network to learn at such different scales, as the standard VFE layers are used for feature extraction, resulting in a 2 m x 2 m x 2 m cell having the same number of features as a 0.05 m x 0.05 m x 0.05 m cell. Furthermore the problem of grid approaches of even small increases in resolution massively increasing the computational cost is not solved, only alleviated as the grid cells are more focused on the areas where high resolution is required.

4.3 Improving Detection Results by Adding Hand-Crafted Features

While spherical grids can heavily reduce the number of required grid cells and are structured closer to the point cloud structure, they still loose some detection performance compared to cartesian grids, because the network can struggle to counteract the differences in cell scale. Therefore some additional features can be included into the network, to reach the default cartesian grid cell detection performance at much lower processing requirements. The first of these features

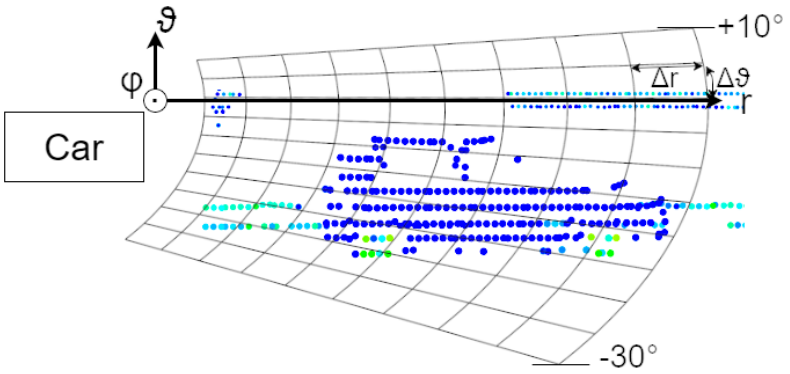
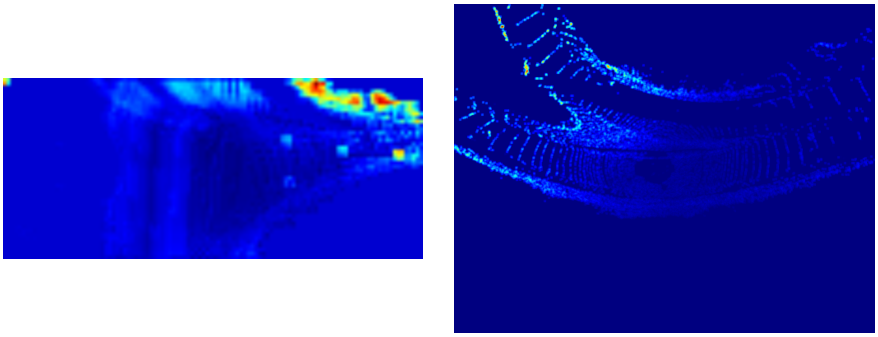


Figure 4.5.: Schematic sideview of spherical grid and point cloud. Grid resolution not to scale with point cloud



(a) spherical grid (b) cartesian grid

Figure 4.6.: Feature maps of a spherical grid and a cartesian grid of the same point cloud, early in training. These show, how the data looks, when being processed by the network. In (a), an early feature map of a spherical grid training is shown, on the right a feature map of the same scene in a cartesian network

would be a separate height map as described in section 4.3.1, the second would be an occupancy grid map as shown in section 4.3.2.

4.3.1 Encoding Height Information in 2D Feature Map

One additional feature that can easily be provided to the network is a height map. This involves encoding specific height information inside of a structure that can be fed into the networks middle layers together with the voxel features from the VFE layers. This height map is of the same dimensions as the original voxel grid, spherical for spherical grids, cartesian for cartesian grids, and encodes 1 value per

cell. This one value is calculated as the mean height $\bar{h}_v$ of all points inside of that voxel, or grid cell,

$$\bar{h}_v = \frac{1}{N_v} \sum_{i=1}^{N_v} h_v^{(i)}, \tag{4.13}$$

with N_v describing the number of data points per voxel, and $h_v^{(i)}$ as the height value of data point i in voxel v.

4.3.2 Occupancy Grid Maps as Additional Feature Layer

A second interesting feature can be occupancy grid maps. These are maps that describe which cells of the grid structure are filled with data points, which are hidden and occluded by other objects, and which are free. This can help the network 'understand' the overall structure, and adds information in areas which are otherwise free of data points and therefore don't provide any information. The main advantage lies in giving the network some pre-computed information on where there should be free space (free), where there might be an object (hit) and which parts might not provide much information (unknown/occluded). An occupancy grid map can be a single time step represented as a map, or an accumulation of information across several time steps and point clouds, both focused on different things. The single time step occupancy grid, as used here, supports the network in frame by frame detections, where context between several time steps might not be important or evaluated, or where only one time step of data is available at each time. An accumulated point cloud is more focused on aggregating free space information at the cost of diminished unknown or shadow data. The computational effort of calculating an occupancy grid is varying heavily depending on the structure of the grid. While it is very easy and quick to calculate an occupancy grid for a spherical grid structure, as shown in listing 4.1, cartesian grids require a lot more effort, as for every single point in the point cloud a raytracing calculation has to be performed to calculate which cells the ray passed through. This can be extremely time consuming, and take several seconds per point cloud, making this feature only really feasible for spherical coordinate systems, where cells are defined by angular ranges, and the precise angle of a data point is known at all times. Therefore it is easy to count the number of rays that pass through a cell, the number of rays that end in a cell and the number of rays that suggest cells behind their stopping point are occluded. Finally the total number of ray collisions with a cell is used to normalize, for calculating probabilities of a cell having the given state. The raytracing process is, simplified, shown in figure 4.7. The red points in the sketch represent lidar

detection points. These points are counted as hits in the occupancy grid, as the ray is, unless there is noise, reflected from an object. They also provide the information that the space the ray went through, marked in green, is free as otherwise the ray could not have passed through. Furthermore the ray is assumed to continue behind the lidar point, but at that point marked as unknown, as there is no information present if there is free space or occupied space. The rays are then projected into the grid cells they pass through, and because as previously explained in listing 4.1, if multiple rays pass through the same cell, this information will be aggregated. Examples of an actual occupancy grid are shown in figure 4.8. Figures 4.8a-4.8c show occupancy information of a spherical grid, as used in this work, figures 4.8d-4.8f show a cartesian grid for comparison. It is easy to see, how the cartesian grid contains more detailed information, but the calculation is, as mentioned previously, extremely time consuming. These images are created from the same point cloud. As seen, and described previously, the spherical grid heavily warps the point cloud if mapped to a cartesian image space, as a standard convolutional network does. Unfortunately the occupancy grid cells had to be averaged over the z-direction, height, to be shown here, losing information and clarity, especially in the spherical grid representation.

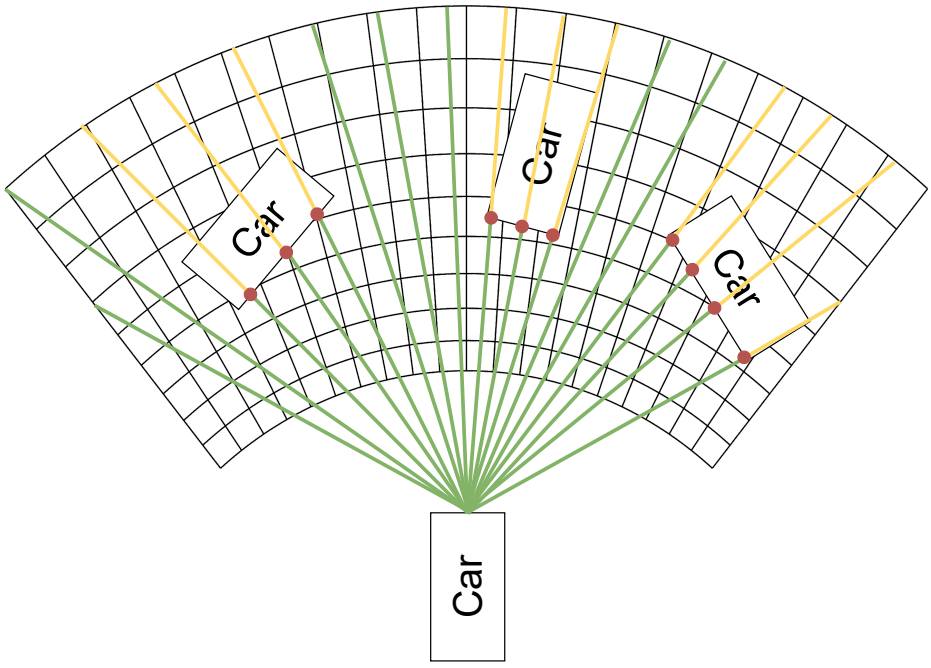
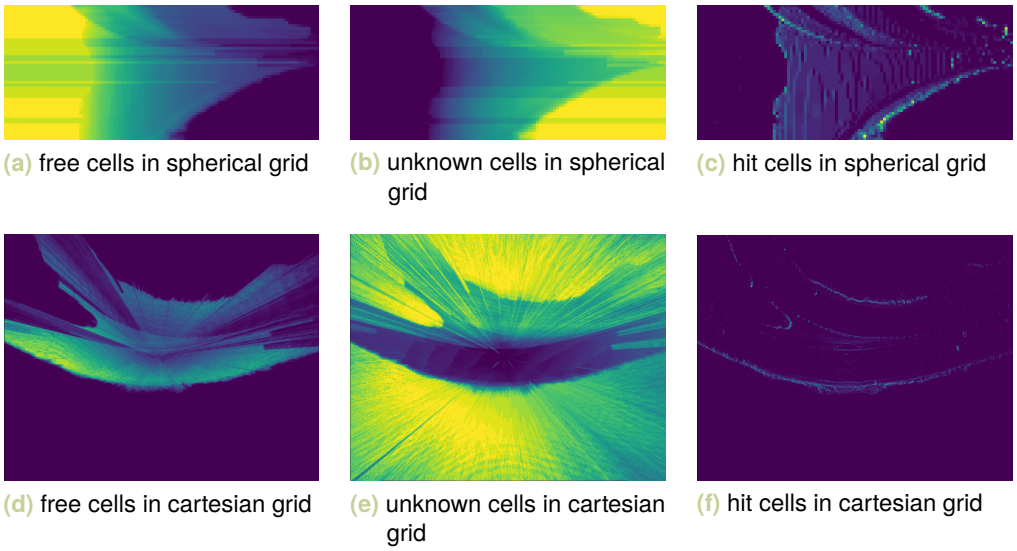


Figure 4.7.: Sketch on how the occupancy grids are calculated. Red dots describe lidar points, which are in the occupancy grid marked as hits. The green rays are raytraced through the lidar points and mark free space, while the yellow rays are continuations of the green rays after hitting objects. They are marked as unknown, as no information is present behind a given point.

```
def get_cell_position(point):
    y_n = floor((phi_point - phi_min) / phi_res)
    p_n = floor((theta_point- theta_min) / theta_res)
    d_n = range_position(point) # from look up table, see eq. 4.12

def calculate_occupancy_grid(point_cloud, og):
    for point in point_cloud:
        y_{n}, p_n, d_n = get_cell_position(point)
        og[0:d_n-1, y_n. p_n, 0] += 1 #free
        og[d_n, y_n, p_n, 2] += 1 #hit
        og[d_n+1:end, y_n, p_n, 1] += 1 #occluded
        og[0:end, y_n, p_n, 3] += 1 #count
    #cellwise normalization by count
    og[:, :, :, 0] /= og[:, :, :, 3]
    og[:, :, :, 1] /= og[:, :, :, 3]
    og[:, :, :, 2] /= og[:, :, :, 3]
```

Listing 4.1: Occupancy Grid calculating for spherical grid



(a) free cells in spherical grid

(b) unknown cells in spherical grid

(c) hit cells in spherical grid

(d) free cells in cartesian grid

(e) unknown cells in cartesian grid

(f) hit cells in cartesian grid

Figure 4.8.: Occupancy information for spherical ((a)-(c)) and cartesian((d)-(f)) grid; dark = low probability of state, bright = high probability of state (free/unknown/hit)

4.4 Novel Methods of Input Data Augmentation for Improved Network Generalization

When training neural networks, a very large amount of varied, annotated training data is required. Recording and manually annotation data containing every possible scenario and enough variety is not realistic and additionally showing the network the same exact data multiple times can lead to overfitting, meaning the network, given enough capacity, does not generalize well, but rather memorizes the data. Data augmentation can be used to alleviate this issue by modifying the training data at every step, so one point cloud never looks the same to the network, even after multiple, or many, training iterations. There are a multitude of ways to achieve this. The easiest augmentations are rotational and mirroring. Rotational augmentation meaning, that the point cloud and all the corresponding annotations are rotated by angles φ, ϑ, ρ along the x-,y-,z-axes. The most common approach is rotating along the z-axis, up-down in the coordinate system used here, to slightly alter the orientation of objects in relation to the ego vehicle. For mirroring the x- and/or y-axis can be mirrored by simply multiplying the positional value by -1 for the given axis. Left-right mirroring requires no special handling of annotation orientation, objects simply need to have their sign flipped as

$$\varphi_{flipped} = -1 * \varphi. \tag{4.14}$$

Flipping an object from the front to the back of the vehicle requires slightly more advanced augmentation, as the required rotation varies significantly, as objects that are at a 90° angle from the x-axis don't need any orientation change, while objects that are at a 0° or 180° angle require a 180° rotation

$$\varphi_{flipped} = \begin{cases} \varphi - 2(\varphi + \frac{\pi}{2}) \,\%\, (-\pi) & -2\pi \leq \varphi < 0 \\ \varphi - 2(\varphi - \frac{\pi}{2}) \,\%\, \pi & 0 \leq \varphi < 2\pi \end{cases}. \tag{4.15}$$

Small scale and translation changes are also possible and viable, although for translational augmentation special consideration has to be given to ensure the point cloud stays valid, concerning occlusion and the sensor model.

A more advanced augmentation would be injection augmentation, first published by Yan et al., 2018. This is focused on building a database from all relevant objects in the dataset and, during training, randomly selecting a number of objects from this database and injecting them into the original point cloud. This can happen in a variety

of quality levels. The most basic approach would be completely randomly injecting the objects without regard for plausibility in terms of occlusion and orientation towards the sensor. This is prone to a larger number of errors, and while neural networks would likely be able to handle this type of data, it could result in worse training results or longer training times. A more advanced approach would be to only inject objects at the position they were originally cut out at. This would fix the orientation to be the same as it was originally, resulting in a more realistic point cloud. This might also include basic occlusion validation, checking if the bounding box of the injected object would overlap with any of the objects in the original point cloud and, in case there is overlap, disregard the injection. A point cloud like this is shown in figure 4.9. The quality of these injections is already a lot higher than the most basic approach, but simply checking for overlap in 3D does not ensure a valid point cloud, as objects can be injected in front of, or behind, other objects, leading to situations where the sensor could not see the object. This takes one important feature away from the network: occlusion shadows. Objects in lidar point clouds can occlude each other, resulting in shadows of very specific shapes, which the network can consider when processing the point cloud. Examples of this can be seen in figure 4.9. Several of the colored objects sit at random positions in the point cloud, not occluding anything behind them, from the perspective of the sensor.

4.4.1 Exploiting Object Shadows With Injection Augmentation

As mentioned before, object shadows, with or without injection, are an interesting feature the network might consider, when processing point clouds. For objects that are included in the original point cloud these shadows are created naturally, as the sensor ray is blocked and the area behind an object is in unknown state, or rather: a shadow. For injection objects this is not evaluated or even handled in the implementation from Yan et al., 2018. This section will introduce major improvements to create these shadows and handle occluded areas created by objects during positioning of injected objects.

In a first step this is done by calculating the occluded angle range for every object in the point cloud. This was done as it is easy and fast to calculate these ranges for objects in a spherical coordinate system, as presented earlier in this chapter. For injecting new objects this is already enough, as objects cannot be injected if they overlap at all in angle. In figure 4.10 these blocked angular ranges are represented by thick, red lines. Objects declined for injection are shown in red, an object accepted for injection in yellow. After performing all injections, every annotated object in the

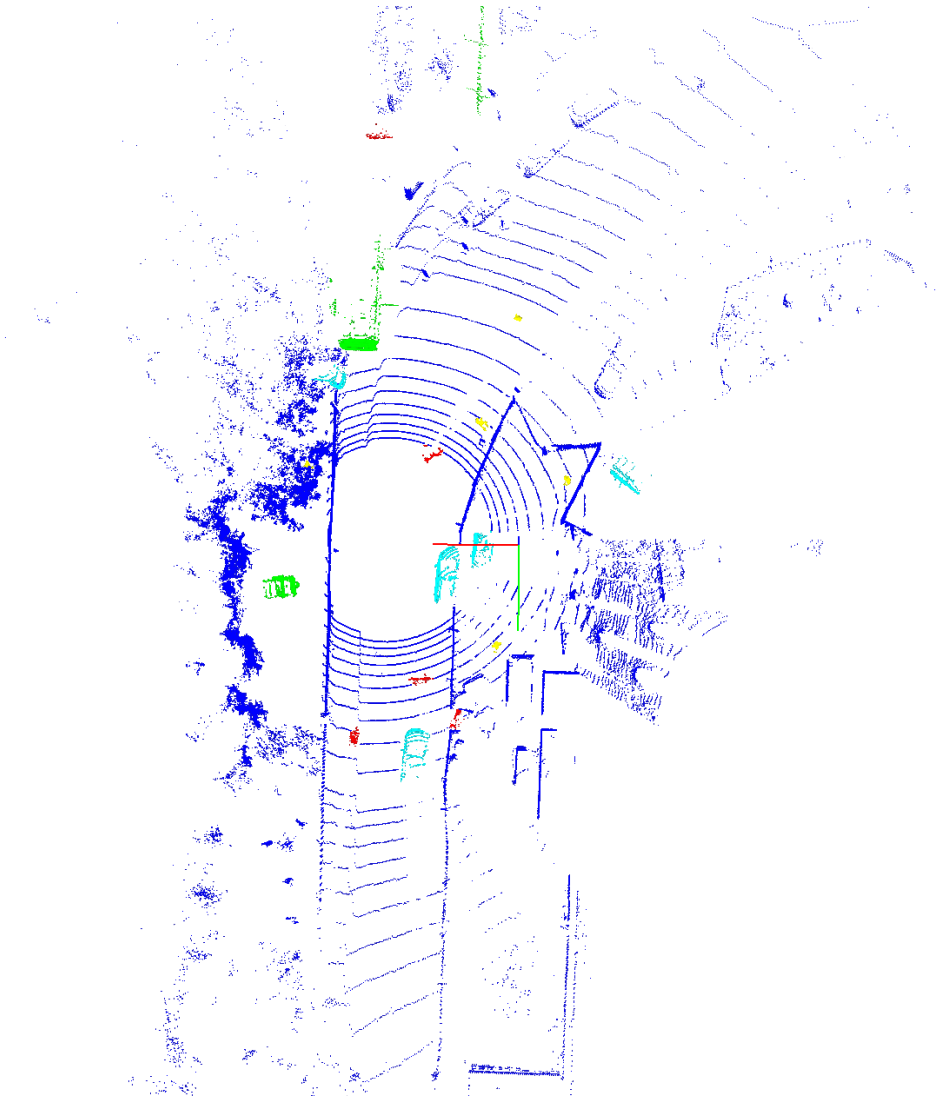


Figure 4.9.: Naively injecting objects at random, only testing against bounding box occlusion. Colors highlight object classes (blue = background, turquoise = car, green = truck, red = pedestrian, yellow = bike). The result are objects inside of walls or objects which would in reality be occluded by other objects.

point cloud is split into a number of sub objects, by angle, and for each of these sub objects the distance to the sensor is calculated. Combining the results from each object allows a mapping of object distance for every angle. The resolution for this can be chosen freely, higher resolution leads to more exact results but also requires more performance. Now every point further away than the maximum object distance at each angle is removed from the point cloud. Angular ranges not containing any objects are kept. The sketch figure 4.10 represents this in blue lines. These lines show how objects are divided in the aforementioned angular bins, and how the cut-off distance for each bin is selected. As can be seen, small

areas close to the objects are not removed. One possible fix is a higher resolution – although the actual resolution is much finer than in this sketch. Nonetheless, choosing the minimum distance instead of maximum distance in each bin would potentially remove points belonging to the actual object, which is why removing less is better than remove more at this point. The method is also useable for cartesian grids, but much more time consuming, as each point has to be converted into polar coordinates first. An example result is displayed in figure 4.11. It can be seen, how injected objects occlude full angular ranges behind them, leading to large cut outs and large blocked areas in the point cloud. Nonetheless, the point cloud is much cleaner than after performing naive injection without occlusion checks or shadow clean up, as previously shown in figure 4.9.

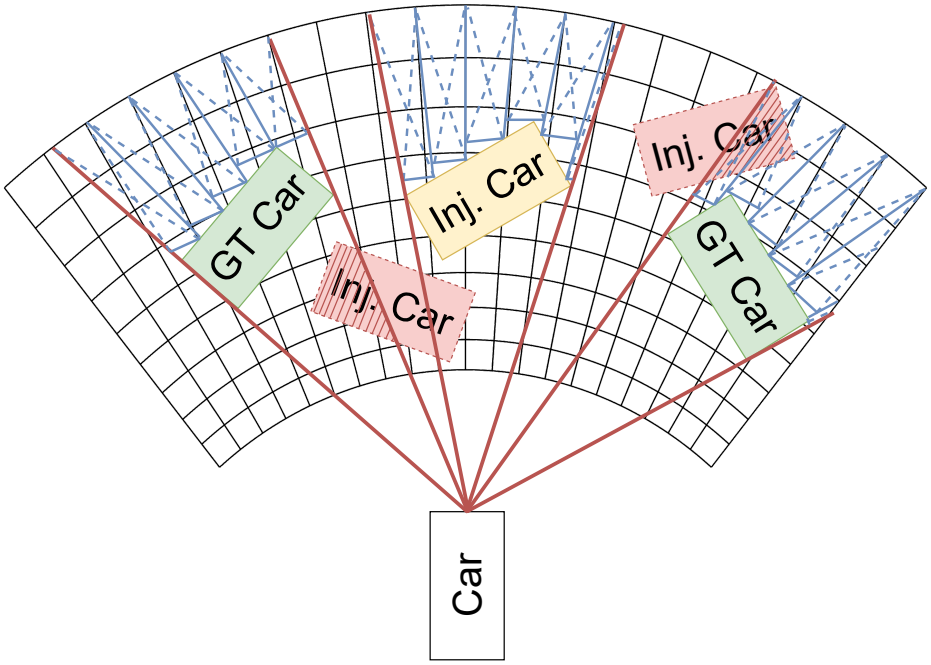


Figure 4.10.: Sketch of angle range/wedge based object injection. 'GT Car' represent objects present in the original point cloud. Red 'Inj. Car' declined object injections and the yellow 'Inj. Car' an accepted injection. Red lines represent the angular ranges that are blocked for injection by other objects. The blue lines represent the shadows which are removed after injection.

4.5 Training, Network Structure and Challenges

The overall network structure chosen is very similar to what Zhou and Tuzel, 2018 and Lang et al., 2019 use and describe. In this case resulting in 1 VFE layer being followed by concatenating these features with both the height map presented in

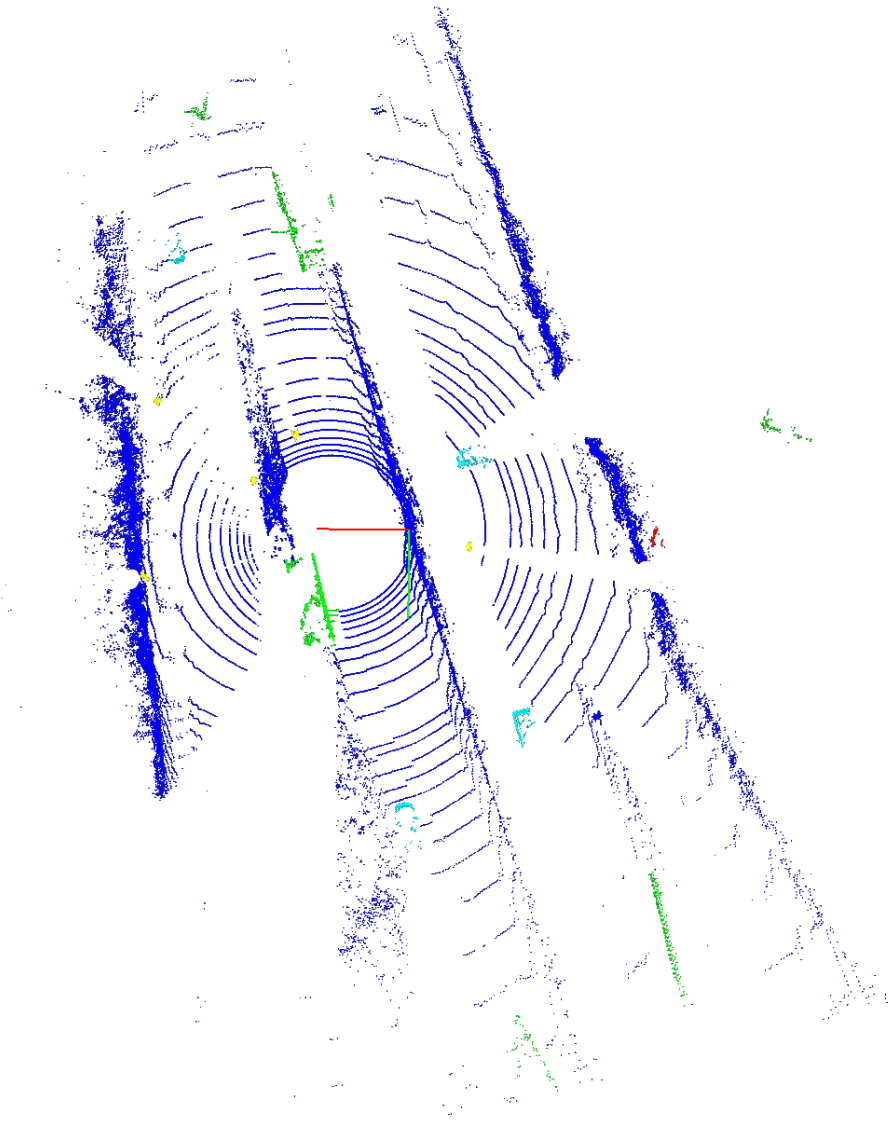


Figure 4.11.: Injecting objects with regard to angular occlusion, including shadow clean up after all injections are performed, resulting in a much more realistic point cloud. Colors highlight object classes (blue = background, turquoise = car, green = truck, red = pedestrian, yellow = bike).

section 4.3.1 and the occupancy grid presented in section 4.3.2. As described by Zhou and Tuzel, 2018 this is followed by several subsampling convolution blocks, with the same number of features extracted at each scale. In the final step these features are again concatenated before being fed into a few final layers to estimate the anchor probability and regression maps. An anchor approach was chosen, as this is what was previously evaluated by others and showed good and reliable results. For training, a large amount of different configurations were tested, but as a general

setup was not a large difference to other state of the art approaches. The chosen optimizer was the, by now, basically standard choice of the Adam Optimizer (Kingma and Ba, 2014). The loss is composed of two different components, a classification loss and a regression loss. This setup follows VoxelNet (Zhou and Tuzel, 2018), as they proved it effective and multiple experiments away from this did not improve performance without cost at other places. This leads to

$$L = L_{cls} + 0.5(L_{reg} + L_{reg;\theta}), \tag{4.16}$$

with the classification loss being a binary cross entropy (BCE) in a 2 class problem – the desired class and the background/none class –

$$L_{cls} = \alpha \frac{1}{N_{pos}} \sum_i BCE(p_i^{pos}, 1) + \beta \frac{1}{N_{neg}} \sum_j BCE(p_j^{neg}, 0), \tag{4.17}$$

α and β begin scaling values, p_i^{pos} and p_j^{neg} the softmax outputs of positive and negative anchors and finally N_{pos} and N_{neg} the amounts of positive or negative anchors. As a regression loss, SmoothL1

$$l_{SmoothL1}(\Delta b) = \begin{cases} \frac{0.5(\Delta b)^2}{\beta}, & \text{if} |\Delta b| < \beta \\ |\Delta b| - 0.5\beta, & \text{otherwise} \end{cases} \tag{4.18}$$

is used, β as free parameter,

$$L_{reg} = \sum_{b \in (x,y,z,w,l,h,\theta)} l_{SmoothL1}(\Delta b), \tag{4.19}$$

Δb as difference between target and network output, and

$$L_{reg,\theta} = \sum_{\theta} l_{SmoothL1}(\sin(\theta^{'} - \theta)), \tag{4.20}$$

following the anchor definitions of Lang et al., 2019 and Yan et al., 2018. For use with multiple classes, the classification loss, and also the probability and anchor predictions, are performed separately for each class. In this work the network was only evaluated for performance on the 'car' class. The network for this is shown in figure 4.12.

One of the biggest challenges of training this network was overcoming overfitting. Usually this is no major issue during 3D object detection in point clouds, when augmentation as previously presented, is used. Unfortunately reducing the grid size so far, while keeping the network relatively big can easily result in overfitting. This is

shown by a rising loss on a validation set, while the loss on the training data still reduces. One might suggest reducing network size, but this showed worse overall results in some areas. While classical approaches like dropout – setting a number of weights to 0 during training – can alleviate this significantly, in this case it did not eliminate the issue. Another big part of the issue comes from the data itself. As the augmentation presented in section 4.4 is good, but heavily limits the area of point clouds in which objects can be injected, to not occlude or be occluded, the overall shape of point clouds stays similar. Additionally a very large amount of cars being oriented in the same or a very similar way in the original data also leads to a large amount of objects in the injection data base looking and being oriented similar. The result are point clouds with a large bias towards vehicles oriented in parallel to the ego vehicle. Without utilizing more and different data, of which not much is available in the Kitti 3D benchmark by Geiger et al., 2012, this is very difficult to solve and could not fully be fixed at the end of the connected project.

Splitting the network for close and far range to improve efficiency

As previously mentioned, one issue of angular grids for processing in a neural network is the large variance in grid cell size, and content, between the grid close to the vehicle and further away. One idea to solve this was splitting the network and using a separate network for close and far range. For this the grid was split at the closest full cell after 35 m distance to the sensor. The split occurs even before feature extraction, resulting in a fully separate path for both zones. Unfortunately this does not noticeably improve results at neither range, but eats a large part of computational benefits, even if the separate networks are smaller than the combined network, as more data needs to be kept in memory. As these results surfaced quickly while performing the experiments, no major evaluation was performed here.

4.6 Quantifying Runtime and Memory Advantages and Evaluating Detection Results

All comparisons are performed between the network described here and a similar implementation following a mixture of the PointPillars structure (Lang et al., 2019) and VoxelNet by Zhou and Tuzel, 2018, meaning that no sparse convolutions were used. This will increase the runtime compared to the evaluations by Lang et al., 2019, but should not influence the memory requirements much, as the same

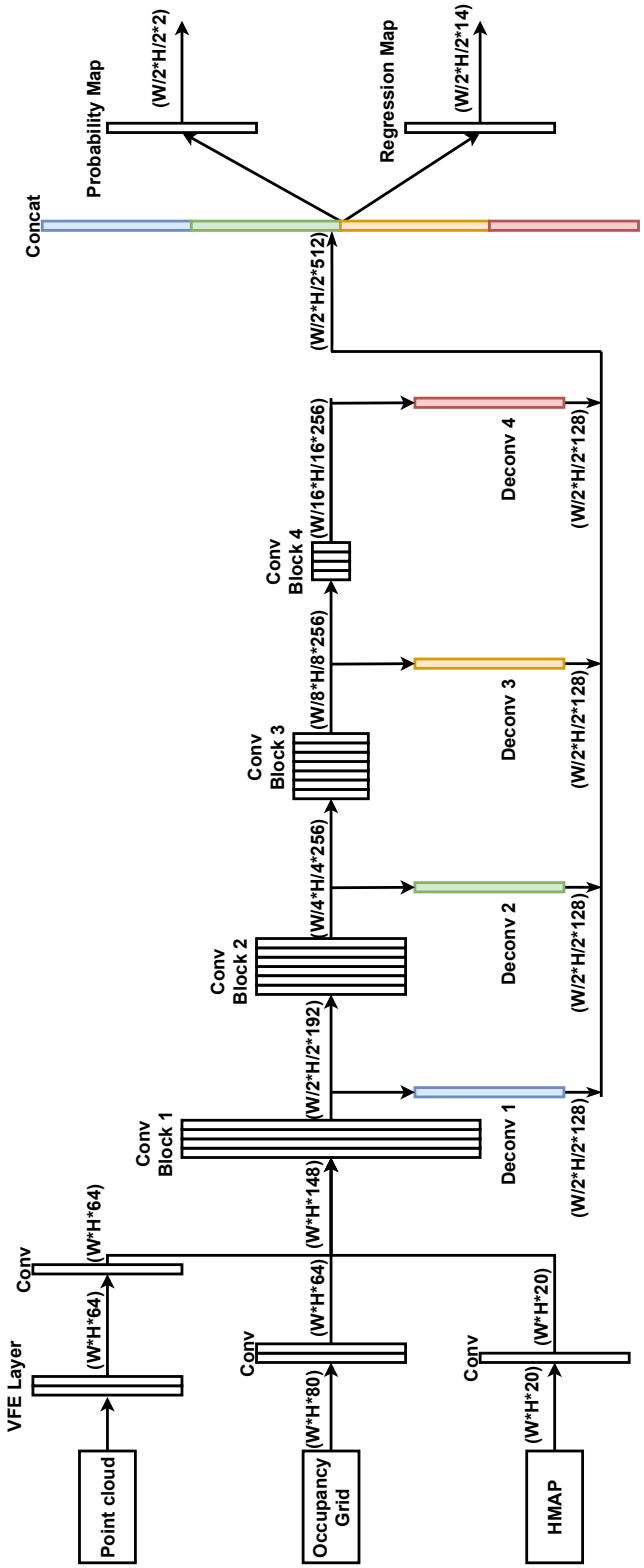


Figure 4.12.: Final network implementation for spherical grid. Image from Alsfasser et al., 2020

number of convolution kernels is kept in memory. All evaluations were performed on an GeForce RTX 2080Ti (with 11 gigabyte (GB) video memory) from Nvidia, utilizing Cuda 10.2, Cudnn 7.6.5 and Tensorflow 1.14 GPU. The dataset was the 3D Object Detection data set from Geiger et al., 2012. Table 4.1 displays these comparisons with a few, key measurements. The first of these are the so called 'FLOPs', or 'Floating Point Operations'. As the name says, these are operations, like additions, multiplications and others, performed on floating point numbers. Floating point numbers are most commonly used in neural network processing, as simplification to integer or fixed point numbers are not always possible. Reducing the grid to a spherical, or polar, grid can, as shown, reduce the required amount of FLOPs by almost 90%. The main reason is given by the dimensional differences between both grids. The spherical grid contains a lot fewer grid cells, leading to a lot fewer convolutions being performed. The second major advantage is the memory requirement. A high memory requirement makes the network more difficult to run during inference. In an ideal scenario the network would be able to work on an embedded system, on which generally memory is low and expensive. The reduction in memory requirement is based on the same as the reduction in FLOPs. The grid is a lot smaller, therefore a much smaller structure has to be kept in memory. All if this also leads to much reduced runtime, with this network only requiring 12ms during inference. These 12 ms were calculated on the same system as the 25 ms for the cartesian network and should be a good indication of benefits in actual applications.

Performance Results

While the main goal of this work was reducing the computational effort behind object detection in lidar point clouds, keeping a good detection performance was also implicit. Table 4.2 displays how this goal was achieved for 3D object detection of cars in the 3D object detection benchmark by Geiger et al., 2012. It is shown how this network only performs slightly worse than the state of the art, at time of development. While these are not perfect results, and at time of writing even better networks like PV-RCNN (Shi et al., 2020) exist, it is still a good result, allowing for real time use even on hardware a lot weaker than the Nvidia GeForce 2080Ti used for evaluation. The slight decline in performance can easily be explained by the much increased complexity of the task asked from the network. Instead of regularly spaced and sized grid cells, each grid cell only contains one feature vector anyways, the network now has to work with vastly different grid cells. The ones closest to the sensor are around 5 cm on all edges, while the ones furthest away are around 3 m

Method	FLOPS↓	Memory↓	Runtime per batch↓
Cartesian Grid	100.14 billion	6 GB	25 ms
Polar Grid	**11.81** billion	**2.9** GB	**12** ms

Table 4.1.: Computational requirements compared to state of the art cartesian network similar to PointPillars (Lang et al., 2019).

Method	Easy↑	Medium↑	Hard↑	Runtime↓
VoxelNet(Zhou and Tuzel, 2018)	76.37%	63.99%	56.55%	500 ms
SECOND(Yan et al., 2018)	**83.13**%	73.66%	66.20%	38 ms
PointPillars(Lang et al., 2019)	79.05%	**74.99**%	**68.30**%	16 ms
This	80.04%	69.53%	64.09%	**12** ms

Table 4.2.: The spherical network compared to state of the art approaches on the Kitti 3D Object Detection benchmark (Geiger et al., 2012). Easy, medium and hard denote difficulty of detection as denoted by the benchmark, values describe mean average precision. It is a function of occlusion, distance, number of points per object and so on.

on each edge. This of course massively changes what kind of content or feature the vfe extracts from each cell and later feeds to the actual detection heads. This might also explain the problems with generalization which occured during training, as much effort was required to reduce and prohibit overfitting produced during training. Figure 4.13 shows an example of this issue. On the left, a probability heatmap from a training with spherical grid is shown, on the right a similar heatmap from a cartesian grid training. Both cover roughly the same area, sizes defined in section 4.2. As can clearly be seen, the cartesian grid shows all objects in a relatively similar size and shape, the network could search for the same shape at all different ranges and therefore could easily detect most of them. Overall the resolution on the cartesian grid is also much much higher than with the spherical grid. On the spherical grid one can see, how the high probability areas are much larger close to the sensor, while they grow increasingly small, and closer together, further away. This represents perfectly how the actual spherical data is perceived by the network and displays the biggest issue this setup has. While the network managed to detect the objects in this scene, it is much more difficult, at long range, to separate objects from one another, which influences the accuracy of the bounding boxes and detections overall.

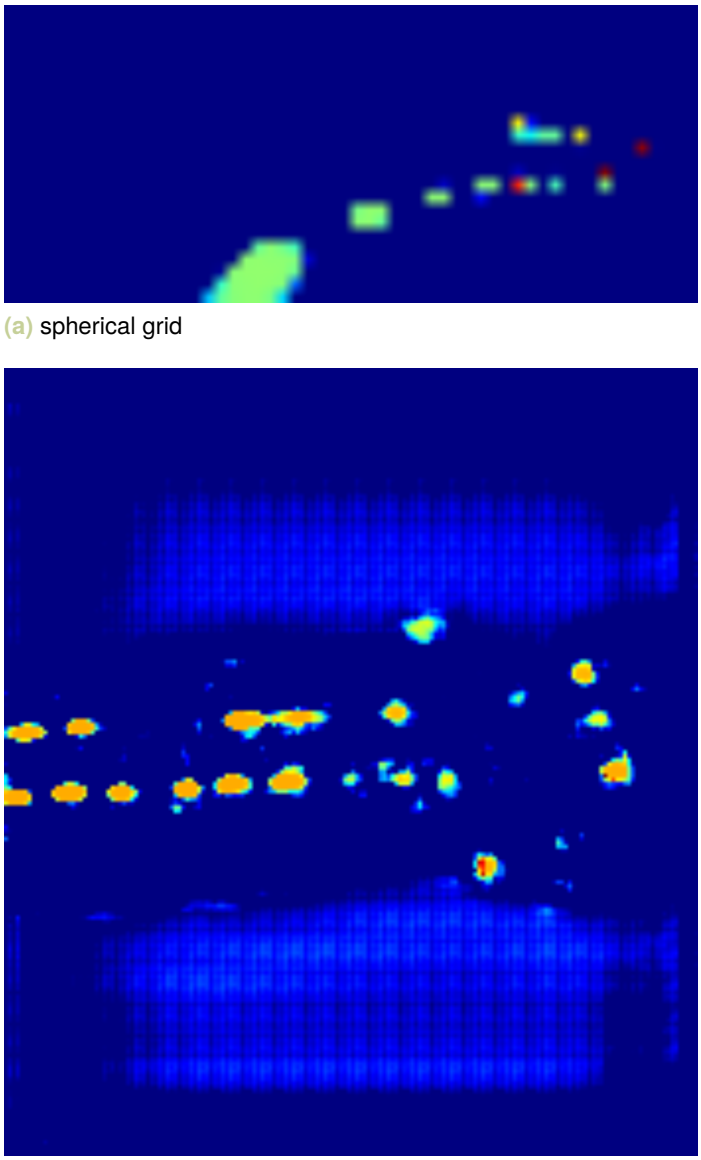
Ablation Studies

As previously explained, the decline in performance by switching from a cartesian grid to a spherical grid was minimized by adding or improving additional features like an occupancy grid or improving injection augmentation methods. The impact of each of these steps is evaluated in this section. For this the same network was trained in multiple different stages of improvement. These trainings were not performed until ultimate convergence, therefore the results are different than what is shown in the previous section. As a baseline, a network was trained utilizing a cartesian grid of 352 x 400 x $20 = 2.816.000$ cells. The same baseline was trained with a spherical grid of 112 x 48 x $20 = 107.520$ cells. These are savings of over 96% of cells. As previously explained in section 4.2.2, increasing the resolution of the spherical grid heavily increases the number of grid cells. Doubling the resolution increases the number of cells to 244 x 96 x $40 = 860.160$, or 8 times more cells, heavily reducing the savings. The trade-off was also evaluated here. The results of this are presented in table 4.3 and were collected from a custom split of the Kitti 3D Object Detection benchmark (Geiger et al., 2012), on the class of cars. As can be seen, the cartesian grid performs best, but also has by far the largest requirements to hardware, as previously shown, and inference time. As for spherical grids, it can be seen, how each added feature improves the detection rate of objects slightly. The worst detection rate is given by the spherical grid without occupancy grid, at low resolution and without injection augmentation. Each of the aforementioned features increases performance slightly. Increased grid resolution does the same, but only slightly improves detection quality over the much smaller network utilizing the occupancy grid, and not to the extend to justify an 8 times increase in grid cells, memory requirements and, if optimized, longer training time. The final result comes, as previously shown, quite close to the performance of a baseline cartesian grid, with the cartesian grid only utilizing injection augmentation and no occupancy grid.

Method	det. Car↑	1000Iterations↓	FLOPs ↓	GPU	CPU(8 threads each)
Cartesian Baseline	**77.3**%	8min:30s	110 billion	RTX 2080Ti	R9 3900X
Spherical Baseline	67.15%	7min:45	**10** billion	RTX 2080	i9 9900k
Spherical No Injection	65.33%	**1min:55s**	**10** billion	RTX 2080Ti	R9 3900X
Spherical Double Res	73.11%	8min:35s	25 billion	RTX 2080	i9 9900k
Spherical Occupancy Grid	70%	8min:35s	11 billion	RTX 2080	i9 9900k

Table 4.3.: Ablation study of spherical grid and the developed features versus cartesian baseline. Result given in average precision, as by Geiger et al., 2012.

As for training times, the numbers vary heavily, as they were collected on a cluster utilizing shared ressources and slightly different hardware between different



(a) spherical grid

(b) cartesian grid

Figure 4.13.: Comparison of detection heatmaps between spherical network and cartesian network (different scene). Pixel color encodes object probability. Blue = low probability for object, red = very high probability for object. Bother cover a similar area, the cartesian grid shows much higher resolution and much more consistency, explaining the slight decline in performance when switching to the spherical grid.

cluster nodes. Therefore the hardware is noted down besides training times. Still, differences are possible. Ressources like central processing unit (CPU) cores and GPUs were not shared, but used through virtualization. The actual training times might therefore still vary depending on the secondary loads on the machine (mainly memory, harddrives, networking). It is therefore important to take these training

times as hints and not as absolutes. FLOPs required for the forward passes of each network are more representative of computational requirements. As can be seen, training times are not even close to scale linearly with the required FLOPs. This is caused by multiple things. First, as explained, differences in hardware. Secondly differences in efficiency. While the spherical grid could run with a much larger batch size compared to the cartesian grid, both were trained with batch size of one here. This introduces much more overhead than required, as data has to be copied from system memory to GPU memory and back. The large differences between the spherical grids also show, how all of these trainings are heavily CPU bound, especially with calculating the occupancy grid or processing injection augmentation. These values are also not comparable to what was previously reported in table 4.1, as they were collected from less optimized and less converged trainings. Table 4.1 results are also more representative of final network performance and hardware requirements, as they were collected on local machines without cluster overhead.

5 Innovative Semiautomatic Data Annotation Methods for Enhanced Annotation Efficiency

While reshaping the processing grid and reducing the processing resolution of point clouds can be used to reduce computational performance requirements as shown in chapter 4, simply using finer processing structures does not linearly translate to higher result performance. When using neural networks to achieve automatic or semi automatic data annotation the highest possible performance is required with less regard for computational effort, as real time is not a requirement. But still, computation time is expensive and a finite resource in most use cases, therefore a combination has to be found which achieves the highest possible performance, while keeping computation feasible. An approach to achieve this balance between high performance and moderate computation is presented in this chapter.

5.1 Improving Training Data Generation With Neural Network Support

One interesting application for high quality lidar object detections is given by automatic or semi automatic object annotation. As previously explained, object detection networks require vast amounts of annotated training data. Generating these annotations by hand is extremely time consuming and therefore expensive, as even with an optimized toolset a single point cloud with an average amount of objects can easily take more than 30 minutes. For large amounts of data this quickly becomes unfeasible for smaller research groups or companies. An automatic, or semi automatic, annotation solution could therefore be hugely beneficial, saving large amounts of money and allowing for much larger annotated databases, resulting in better final products, even if the annotations are of slightly lower quality than fully manual annotations.

Fully autonomous data annotations can lose a lot of precision unfortunately. If they don't have any human supervision they can produce large amounts of false positives, false negatives or annotations with significantly wrong dimensions, positions or orientations. If human supervision is possible this can easily reduce the effectivity, as a human would need to clean up all the false positives and negatives, which is very time consuming as it is not often easy to distinguish between these, if they are not created by the human that is supervising. It would therefore be beneficial to integrate the human into the data annotation, without requiring fully manual annotation. This is where the patch processing from section 5.3.2 can be hugely beneficial, as shown in section 5.4. The resulting application would result in the human annotator selecting an area of a point cloud where they see, or assume, an object. This area of the point cloud is cut out and then, as described in section 5.3.2, fed into a neural network, to extract very precise bounding boxes and point-wise classifications from this selection. An example network for this application is shown in figure 5.3 and figure 5.4, which is a combination of the feature extraction from PV-RCNN (Shi et al., 2020) and the bounding box regression and classification from Part-A^2 (Shi et al., 2021). PV-RCNN is a large, complex network, trained on multiple GPUs at the same time. The feature extraction they propose is a combination of sparsely processed voxel grid based point clouds at multiple scales, and PointNet (Qi et al., 2017a) feature extraction at each of these scales. This results in highly processed features at each scale, collected at a freely chosen number of keypoints, which can be processed further for bounding box regression and classification. The basic structure of this will be shown later in figure 5.3, although Shi et al., 2020 introduce even more features, which were cut here. The output layers and structure was taken from the authors earlier publication, Part-A^2 (Shi et al., 2021), where they propose anchorless bounding box regression as a combination of multiple outputs. A center position estimated by a rough binning refined by an offset, a size regression and an orientation regression. This is implemented here, as anchorless approaches generally require less memory on the graphics card, which was important here, as multiple time steps of data are to be processed at the same time, all being kept in memory.

The small input patch of data allows very fast processing, enabling the user to seamlessly work with the tool, without having to wait for the network to process the point cloud, as this is done in few milliseconds per patch. Further improvements can be achieved by processing data over multiple time steps as shown in section 5.3.1. As this annotation work is done in an offline system, the whole data sequence is available at each time step. To achieve greater stability, this can be exploited, by

taking data from multiple time steps into account. Here 5 time steps are processed at the same time and combined in a bi-directional GRU.

5.2 Baseline Network Structure for Offline Annotation Algorithm

The network chosen for this task is a combination from the well known publications PV-RCNN (Shi et al., 2020) and Part-A^2 (Shi et al., 2021). This combines highly advanced feature extraction provided by the combined implementation of voxel grids and PointNet (Qi et al., 2017a) layers with low cost and fast anchorless bounding box regression.

5.2.1 Data Preparation and Pre-Processing

Input data for this algorithm consists of 5 consecutive, ego motion compensated point clouds. This requires more complex pre-processing especially during training, as ground truth annotations, augmentation and other modifications have to be synchronized. Pre-processing 5 time steps of data for each processed frame of data is very time consuming and has to be limited in scope for this reason. The first step of pre-processing is data augmentation. This is done first as the actual point cloud is modified by object injection, as shown in section 5.3.3. These added and removed points have to be incorporated into several pre-processing steps like the ego motion compensation, clustering and annotation synchronization.

As described in section 5.3.1, multiple time steps are processed in the same frame, therefore the 5 point clouds have to be aligned. For the relatively simple alignment performed here, yaw-rate and vehicle movement are required. These can be collected by on vehicle sensors or alternatively "simultaneous localization and mapping (SLAM)" based methods as LOAM (Zhang and Singh, 2014). The ego-motion compensation in this case is done by a combination of rotation around the z-axis

$$R = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{5.1}$$

with rotation angle φ estimated from a recorded yaw rate and the time between point cloud recordings. Point clouds also need to be translated

$$S = \begin{bmatrix} s \sin \varphi \\ s \cos \varphi \\ 0 \end{bmatrix}, \tag{5.2}$$

with s as the recorded forward movement distance recorded by the ego vehicle. The final ego motion compensation is finally defined as

$$p_{\text{comp}} = (R * p^T)^T + S, \tag{5.3}$$

with p as the original point cloud and p_{comp} as the ego motion compensated point cloud. Annotations don't have to be ego motion compensated at this stage, as only the results at time step T are evaluated. Annotations and point clouds are therefore shifted to match this time step. Afterwards the point clouds are voxelized. As only small parts of the point clouds are processed at each time, see section 5.3.2, this voxelization is performed at a high resolution to ensure enough detail is being captured, resulting in a grid of 128 x 128 cells for each 10 m x 10 m patch. This is supported by the choice of only using a single data point per voxel cell – instead of several as in a standard VFE as shown in section 4.1.1. This choice is made to ensure the ability to process all 5 point clouds in a single iteration, even on consumer grade graphics cards like the GeForce RTX 2080 with only 8 GB of Video Memory. An overlay of the 5 ego motion compensated point clouds is shown in figure 5.1. As this is a simple form of motion compensation based on recorded ego vehicle motion, it does not work perfectly and the compensation accuracy decreases with increased distance, as angular errors get more and more pronounced. Additionally dynamic objects can not easily be compensated, as they move in relation to the ego vehicle. To compensate for this, a much more complex approach would be required, offering dynamic compensation for each lidar point. A complex point flow estimation, for example proposed by Baur et al., 2019, would be required to estimate this point-wise shift.

5.2.2 Feature Extraction Structure

Following Shi et al., 2020, feature extraction for the algorithm presented here, is performed in a multi-stage and multi-scale approach. After pre-processing, each batch consists of 5 point cloud cut outs including positional grid coordinates. Following PV-RCNN the feature extraction per patch is a combination of sparse convolutions,

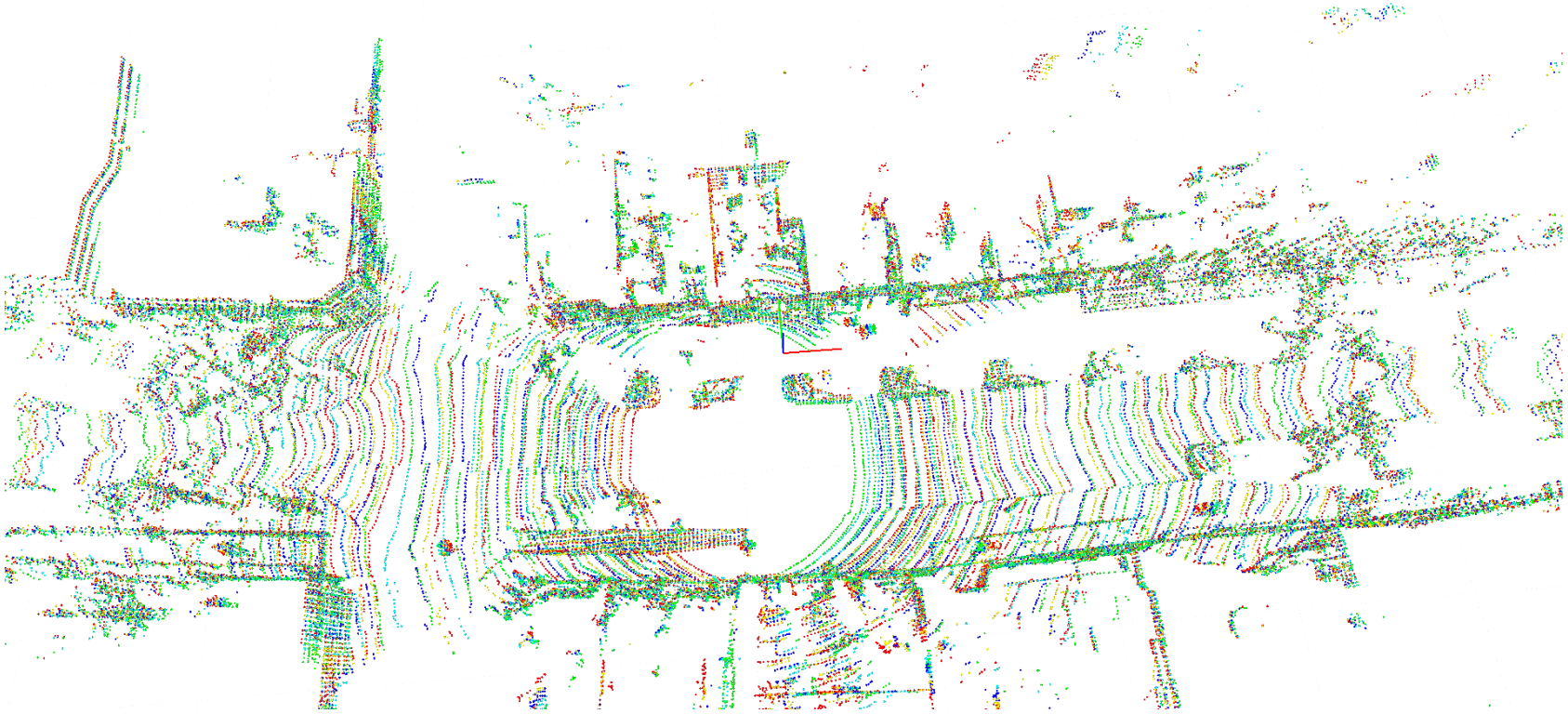
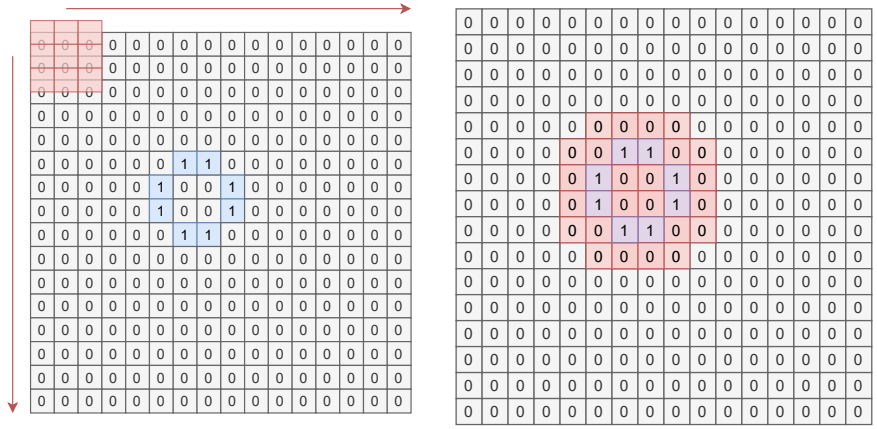


Figure 5.1.: Ego motion compensated point clouds overlayed (blue = T-2, light blue = T-1, green = T, yellow = T+1, red = T+2). This shows how static object like parked cars and buildings are almost perfectly matched, but also how moving objects can simply not be properly matched by a simple egomotion compensation. For this point cloud flow would need to be obtained and applied.

as shown in figure 5.2, on the voxelized point clouds and PointNet layers at the different voxelized scales. While standard 2D convolutions simply convolve the filter kernel with every data point of the input, even if the data point contains no information, sparse convolutions only work on cells actually containing relevant data. This reduces the number of FLOPs on a 16 x 16 grid (as shown) with only 8 cells containing data with a 3 x 3 kernel from 16 x 16 x 3 x $3 = 2304$ to only 8 x 3 x $3 = 72$, in cases where only the calculations centered on filled input cells are performed. Figure 5.3 displays the developed network, where the upper part of the graphic shows the voxelized data processing at different scales, with the lower part handling the final feature output at each scale, extracting features from the voxelized feature maps. The result at each scale is a feature vector of variable size. These are concatenated as seen by the colored feature vectors in figure 5.3. This figure shows both submanifold and standard sparse convolutions. The difference is mainly, if outputs are only kept at the exact position at which input information was present, or if the dilation created by the convolution kernels is kept. Much more detail on that is given by Graham and Maaten, 2017. A submanifold convolution only keeps output information for, as the authors call them, active cells, while a sparse convolution keeps them for all 9 cells, the kernel can be centered on, while touching the active cell. In this case a combination is used. In the provided example in figure 5.2, the difference between 8 cell outputs (submanifold -> shown in purple) and 56 (standard sparse -> shown in red) is shown. The first 2 layers are submanifold, to keep the sparseness high, while the final output of each block is a standard sparse convolution, to pass more information to the next block/scale.

5.2.3 Class and Bounding Box Regression

The input to the class and bounding box regression is a large feature vector for each of the 5 time steps. The structure of this block is visualized in figure 5.4. On the left, the 5 feature extractions, shown in figure 5.3, is shown for each time step in a different color. These are stacked into a sequence, which is fed into the bi-directional GRU shown in the middle of the figure. Only the $T = 0$ result is further processed, the other time steps are additional inputs to the GRU-Block. This still provides a benefit to the training, which can be seen in table 5.1 and table 5.2. The output of the GRU is fed into a combination of regression and classification blocks following the anchorless bounding box regression by Shi et al., 2021. These are 3 major blocks. The first is a simple classification block, consisting of a number of fully connected layers, culminating in probabilities for each of the classes and the none/background class. The number of foreground classes can vary depending on



(a) Standard 2D convolution (b) Sparse convolution

Figure 5.2.: Schematic comparison of standard 2D convolution and sparse convolution. In the standard convolution shown in (a), the filter kernel (red) is convolved with the full input map, even though most of it contains no information, is zero. The sparse convolution in (b) only convolves the filter kernel at the parts of the input map, which contain relevant data. Red cells show on which cells the filter would produce an output for a standard sparse convolution, the purple cells show output positions for a submanifold sparse convolution.

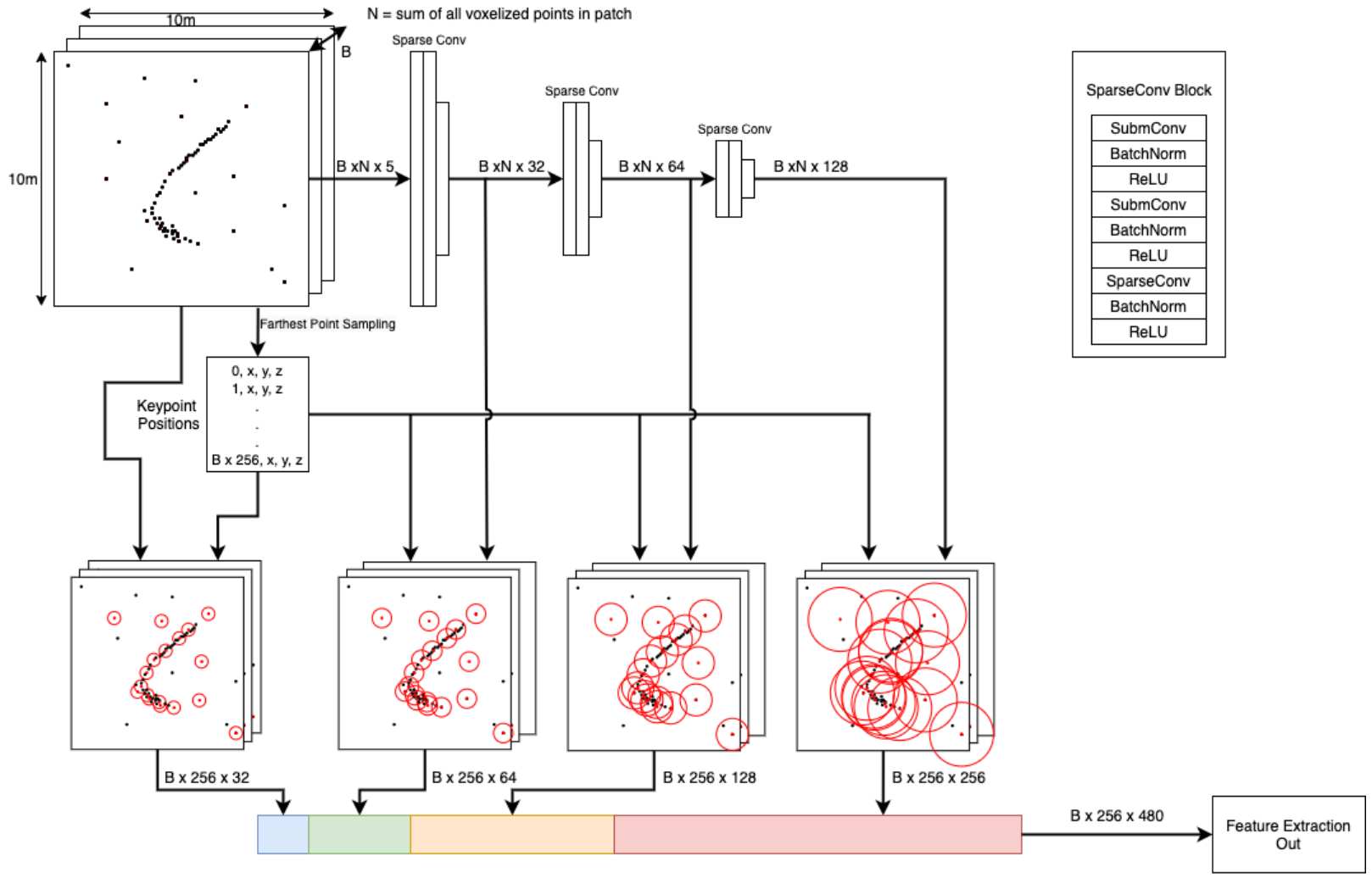


Figure 5.3.: Feature extraction as modification from PVRCCN(Shi et al., 2020). In a first step (top), the voxelized point cloud is processed by sparse convolutions to generate feature maps at 4 different scales. These feature maps and keypoints determined by farthest point sampling at the original stage are fed into PointNet blocks to extract features at each keypoint and at each scale. This results in a combination of localized and globalized features at each keypoint, which are finally concatenated.

the setup, but generally these are cars, trucks, pedestrians or bikes. Evaluations for single class networks compared to a multi class network are shown in table 5.4. The second major block is the center point regression for the bounding box. The first few fully connected layers are shared across the different directions, before the network splits into multiple small blocks for predicting the x-positional bin, y-positional bin and an orientation bin. These bin predictions are supported by an offset regression for x-, y-, z-coordinates and bounding box orientation to produce the final outputs. This combination was determined by Shi et al., 2021 to provide better results than simply regressing the values directly, as directly regressing the values includes too much variance for reliable predictions. Bins can vary in size, and therefore the regression resolution varies. In this case, x- and y-bins were chosen as 0.5 m, the orientation bin was chosen as $30°$. Finally, the bounding box sizes are regressed directly. This was selected, over the more well known and more wide spread anchor based approach, as anchorless algorithms generally, as the authors from Part-A^2 discovered, use less memory, which is again important when keeping 5 point clouds in memory at the same time.

5.3 Advances in Network Structure

While the overall network structure is taken from baseline state of the art approaches – PV-RCNN (Shi et al., 2020) and Part-A^2 (Shi et al., 2021) – , multiple modifications were made and multiple advances will be shown here. First, the introduction of multiple time steps will be detailed in section 5.3.1, followed by clarifications on the patch-wise data processing in section 5.3.2. After introducing a modified object injection augmentation in section 4.4.1, this will be further expanded in section 5.3.3, before finally an additional input feature will be introduced in section 5.3.4. Each of these modifications and improvements is introduced for a specific reason detailed in the corresponding sections.

5.3.1 Time Series Considerations for Improved Result Stability

One method for improving and stabilizing object detection results over a sequence of point clouds is processing multiple time steps at the same time. A problem of object detection in point clouds is the inconsistency created by the strict structure of the sensor measurements. Because of this even small obstructions and errors in the scan can lead to large occlusions from one time step to the next. Additionally,

Figure 5.4.: RNN + bounding box regression, following Part-A^2 (Shi et al., 2021). On the left, each of the different time steps is displayed, each with a different color for their output feature vector. These are stacked as a sequence, before being fed into the bi-directional GRU in the center of the figure. On the right the separate output blocks are shown, color code for visualization

increased confidence can be gained if an object is present and detected in multiple time steps during one processing step. While this can be done in a separate tracker, recurrent neural networks can also consider this directly into the deep learning model. Doing this can go beyond what an external tracker can do, as the network can take information from multiple frames and use these features for better detections, even in unclear situations or non perfect situations. These situations might include a pedestrian that is walking behind a pole or a car which blocks a bike for half a second.

The basics of an RNN were explained in section 2.2.2.

In the final network, 3 layers of these GRU components are utilized, being fed a sequence of 5 feature vectors, once from each direction of the sequence, in a bi-directional setup. As each layer contains 512 GRU nodes, the resulting output in each direction is a 512 feature long feature vector for each input. The relation of input to output data is shown in figure 5.4. As only the output for T is utilized, the other outputs are discarded, and both feature vectors for T concatenated, for a final 1024 long feature vector out of the GRU block. As each block is only ever fed 5 inputs, it is important to not initialize the hidden state with zeros as shown in equation 2.70. Doing this can bias the GRU to overfit towards very small, diminishing outputs. Therefore it was chosen to utilize xavier initialization as by Glorot and Bengio, 2010, simply resulting in initializing weights to values in the range of

$$U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \tag{5.4}$$

with U a uniform distribution and n the size of the input layer. This ensures that the initialization is never the same and results in the network focusing more on disregarding the initialization quickly in favor of the actual input. A schematic overview of such a GRU block is given in section 2.2.2 in the fundamentals overview. The relations between input, hidden state and the state output h_k, which is interpreted as output of the layer, are shown, showing a relatively complex processing structure, explained from equation 2.71 to equation 2.74. This does only show a single direction and a single element of one GRU layer, with the final GRU used being multilayer and bi-directional, as shown in figure 5.4.

5.3.2 Improving Network Performance by Patch-Wise Data Processing

Processing a full point cloud at each time step can limit the resolution at which it can be processed, as GPU memory is a major limiting resource, even when

using the network for offline data processing, as GPUs with very large memory capacities are often prohibitively expensive. This issue can be resolved in multiple ways. The processing resolution, in a grid-based approach this would be the grid resolution, can be reduced, the depth or wideness of the network can be lowered or serialized approaches that only store parts of the network/data in the memory at each time step can be utilized to achieve this. In other words, the network can be made smaller, reducing memory, and computational, requirements. Unfortunately, as shown previously in chapter 4, it is often difficult or impossible, to achieve the same quality of results this way, even when also adapting the processing structure and introducing more focused features. Another way of reducing computational and memory requirements would be removing unnecessary data from the point cloud to only process parts of it which contain relevant data or objects. This can be done in multiple ways, either in a two stage network or with some kind of external input. A two-stage network would use a first, more simple, network to select relevant regions, which are then be processed by the more complex and deep second stage. Examples for this are given by Shin et al., 2019 or Qi et al., 2018, which use camera images in which they detect objects and calculate approximate 3d positions which are later refined by a 3d network. Another example is given by Lehner et al., 2019, which only requires a lidar input and uses its first stage to find probable regions in a birds eye view of the point cloud.

Looking ahead at the application presented in this chapter, the area selection for this approach is done by a human supervisor using some kind of interaction based interface. The result of this selection process is an area of the point cloud, at variable size, which is then processed further. Furthermore, it is tested how the same network performs if the whole point cloud is processed in large patches compared to the single object patches. As for using the patches, it is important to note how positional data is shifted. As each patch is from a different location in the original point cloud but processed without any relation to that original position, it would simply confuse the network to keep the original coordinates for the x- and y-positions. Therefore a transformation to relative coordinates

$$x_p^{(i)} = x - x_c \quad \forall x \in \mathbb{P}_x \tag{5.5}$$

$$y_p^{(i)} = y - y_c \quad \forall y \in \mathbb{P}_y, \tag{5.6}$$

with $\mathbb{P}$ being the set of all lidar points in the patch, x and y the original point coordinates, and x_c, y_c the coordinates of the patch center in the original coordinate

system, is required. The result are coordinate positions relative to the patch center, for a patch size of 10m, this would lead to

$$x \in [-5, 5] \quad \forall x \in \mathbb{P}_x \tag{5.7}$$

$$y \in [-5, 5] \quad \forall y \in \mathbb{P}_y. \tag{5.8}$$

Of course all annotations have to be shifted in the same way. The network can now learn on relative positions, and therefore easily adapt and work on patches from any position in the original point cloud. The size of the patches can, and must, be adapted to fit to the corresponding object. For example, a truck needs a much larger patch to fit than a pedestrian, where a large patch might actually be counter productive, as more objects are included, possibly confusing the network which is only trained to find one object per patch.

5.3.3 Structure Aware Point Cloud Augmentation

The algorithm shown in section 4.4.1 is a good introduction into more precise object injection augmentation, but time consuming to evaluate in cartesian coordinate systems. Additionally, it is more precise than simply injecting objects at random, but as whole angle ranges are cut from point clouds, or occlude and block injections, they still remove too many points from point clouds and inject too little. In a further development, "structure aware point cloud augmentation (SAPCA)", this was heavily improved, both in accuracy and also runtime.

The basics of range images are presented by Hasecke et al., 2021 (and Hahn et al., 2020), as in projecting point clouds into range images to process them, at least partially, in 2D. While they originally use the range images for clustering, these can easily be modified to calculate precise object shadows for injected objects and do pixel perfect occlusion checks when injecting objects. This will also be published in collaboration with Hasecke et al., 2022 at a later date.

To understand the basis of this technique one has to know how a rotating lidar generally records data. A number of separate beams is sent out, and the time to reflection and its intensity is recorded at fixed angular intervals – 0.08° for the Velodyne HDL64 (Velodyne Lidar Inc., 2021a). Both pieces of information are known, or can be calculated, for each resulting point from the point cloud. For the HDL64 this results in $\frac{360°}{0.08°} = 4500$ data points for each scan line or each lidar beam/vertical step. Adding the vertical component, each recorded data point can be mapped to a fixed pixel position in an 4500 x 64 pixel image in case of the 64 scan lines of the

HDL64. Furthermore this also means, that each pixel in the resulting range image maps to exactly one measurement in the point cloud. If for some reason multiple data points are mapped to the same pixel, only one of them could actually have been recorded by the lidar in the given point cloud/scan. This property will later be exploited to estimate precise object shadows for injected objects. An example of this masking is shown in figure 5.5.

For a first step of the injection augmentation, it has to be selected which objects are supposed to be able to occlude other objects. In some cases one might only want moving objects to be relevant, or only have cars occlude injections, other cases might require more detailed inclusions, where everything up to lamp posts or plants are occluding objects. To perform this mapping, a point cloud segmentation needs to be calculated by assigning an object class to every data point in the point cloud. This segmented point cloud is then projected into a range image, as shown in the upper part of figure 5.5. It creates a map of forbidden areas, in which no further objects can be injected. The same is done for each object that is supposed to be injected, one at a time, shown in the middle of the aforementioned figure. The result are two range images, each having some pixels marked as foreground, by having a value other than zero, and some as background with a pixel value of zero. To calculate if the newly injected object would be occluded or would occlude some other object, both range images can be summed over, to get a resulting range image, in which both, objects of the original point cloud and the newly injected objects, are marked, shown in the lower part of figure 5.5. If any of the resulting pixels has a value of 2, this means there is an overlap between the original point cloud and the injection object and therefore the injection object is rejected, and the next object is processed. After all injections for a point cloud are performed, it is exploited, that each pixel in the range image can only be re-projected to 1 point cloud data point, this will represent the first point the lidar ray hits. The complete point cloud is therefore projected into a range image and immediately projected back into 3D, which removes all but the first data point each ray would create. The result is a perfectly cleaned point cloud, where each object has realistic shadows. Of course the shadows are limited by the resolution of the input point cloud, as any gaps in the range image result in mistakes in the shadow calculation. This can be fixed by gap-filling algorithms, to solidify objects in the range image, and remove gaps in otherwise filled areas.

The resulting pointcloud is a perfectly valid point cloud that is still augmented by injecting additional objects, without creating non-plausible scenarios. In further thoughts this technique can also be used to build completely new scenes.

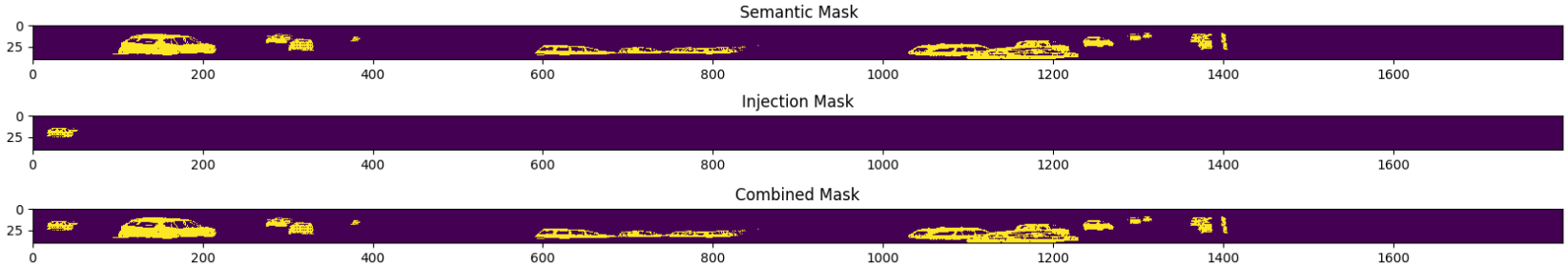


Figure 5.5.: Range image projection used for SAPCA. The top/leftmost image shows the object projection before injecting the new candidate, the middle shows the projection of the candidate and the bottom/rightmost shows the combination. If there were any overlap in the last image the candidate would be rejected. If the injection is performed, the combined mask can be used as object projection mask for the next candidate, reducing the performance overhead massively.

As for comparison with the approach described in section 4.4.1, figure 5.6 is from the same scene as figure 4.11. It can easily be seen how the edges of the cutouts are much more natural, and injected objects don't occlude the full angular range behind them anymore. Detailed comparisons are shown in figure 5.7. In the naive injection method, the newly injected car does not occlude any of the lidar points behind it. As will be shown in the evaluation, this results in the network having trouble deciding between background and foreground, and results in much reduced classification accuracy for each data point. The middle ground implementation, angular injection, shows how the complete area behind the injected car is removed from the point cloud. While this prevents errors as in the naive injection, it still does not describe a realistic point cloud, as the final SAPCA example shows. From the point of view of the sensor, only some parts of the point cloud behind the car will be occluded.

5.3.4 Clustering Results as Input Feature

As additional input feature to support detection of relevant objects the clustering algorithm presented by Hasecke et al., 2021 is applied. This algorithm uses range images, as described in the previous section to cluster point clouds. For this, the authors utilize direct connectivity in the range image, but also employ 'map connections' connecting pixels not neighbouring each other directly, to estimate cluster membership in a classical, non machine learning way. The result of this algorithm is a cluster ID for every point cloud data point. Data points that belong to no cluster will be assigned the same ID 0, while other data points are assigned consecutive, non persistent IDs. The result is a data point like

$$p = [x, y, z, i, c], \tag{5.9}$$

where i contains the reflection intensity recorded by the lidar and c the cluster ID. While this could be used in a network, it might also confuse the network, as cluster IDs are non consistent and the same object will likely have different IDs across different time steps and also the number of clusters, and therefore the value range will vary. Even when normalized, the value of the cluster ID does not provide any relevant information to the network. Therefore, a choice was made to simply assign a *true* value in case the point belongs to a cluster and a *false* value if not:

$$p = \begin{cases} [x, y, z, i, 0] & \text{point does not belong to a cluster} \\ [x, y, z, i, 1] & \text{point does belong to a cluster} \end{cases} . \tag{5.10}$$

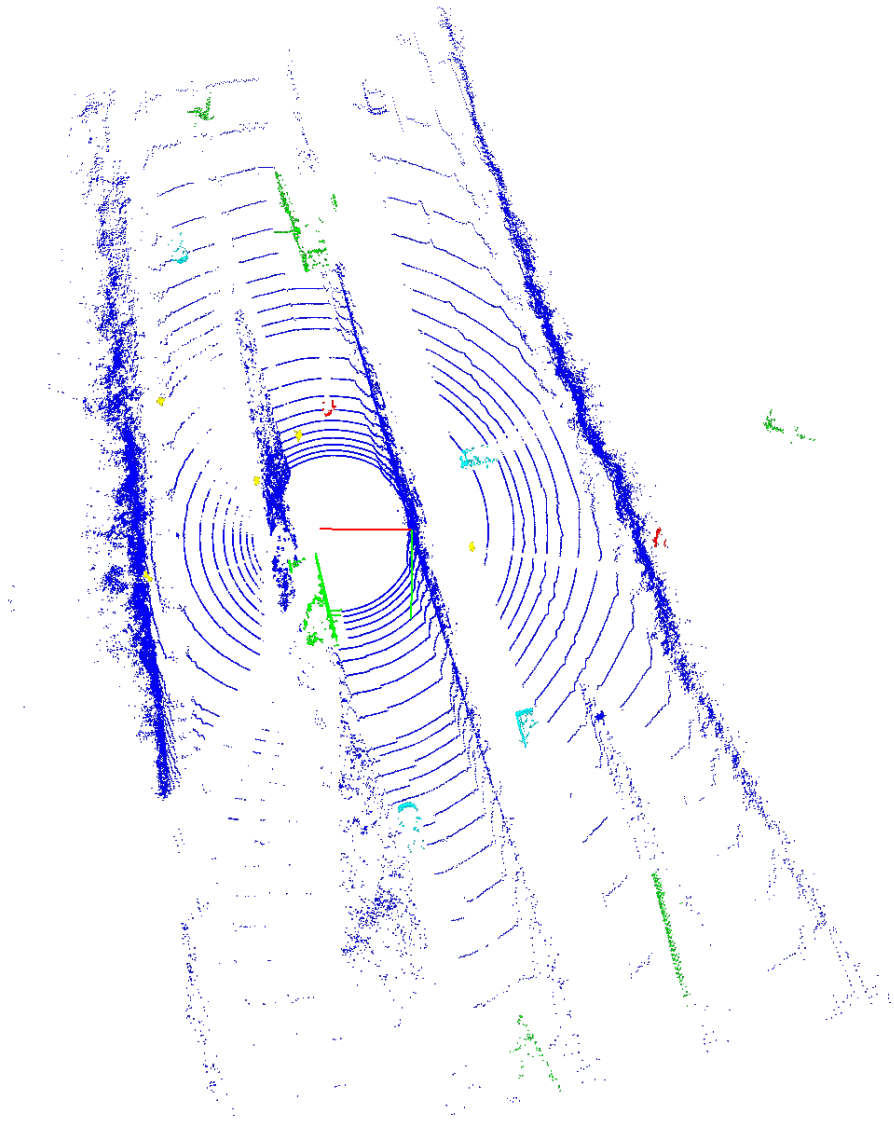
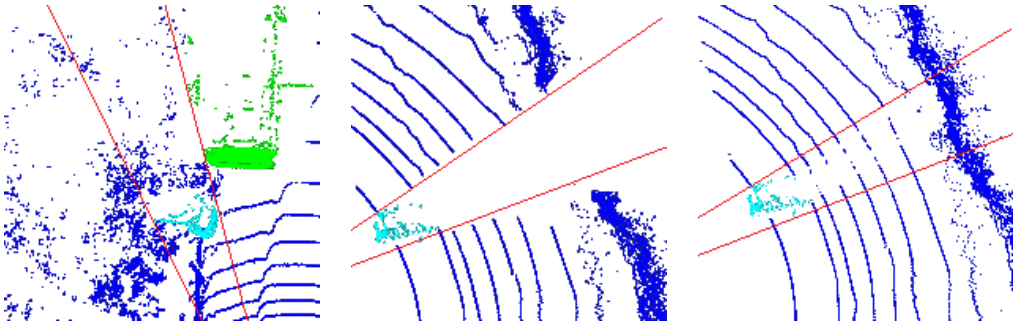


Figure 5.6.: SAPCA based injection. More objects injected than with wedge based injection shown in figure 4.11 and still a cleaner point cloud with cleaner object shadows. Colors highlight object classes (blue = background, turquoise = car, green = truck, red = pedestrian, yellow = bike).

As a result the network could more easily distinguish between background points and foreground points, leading to higher classification accuracy. Figure 5.8 provides an overview of an example point cloud and the information of which point belongs to any given cluster (red). Looking at this figure, it can be seen, how, while the ground plane is well filtered (blue; not in a cluster), especially walls are an issue, as most of them are classified as clusters (red).



(a) Naive injection of car

(b) Angle based injection of car(section 4.4.1)

(c) SAPCA based injection

Figure 5.7.: Comparing the injection of a car in naive injection, angle based injection and SAPCA injection. The red lines show the field of view from the sensor. It can be seen, how naive injection does not create any object shadows, angle or wedge based injection creates a full shadow behind the object and SAPCA creates a realistic shadow behind the car. Colors highlight object classes (blue = background, turquoise = car, green = truck)

As can be seen, there is a lot of noise looking at walls and vegetation, but the singular objects, mostly cars, are often clearly split from any surrounding points belonging to clusters. In reality a large number of different clusters would be present, but as previously mentioned, the employed feature only classifies if a point belongs to any cluster, not to a specific cluster.

5.4 Evaluation of Network Improvements Against Baseline

In the following evaluation, classification accuracy on keypoints of the different networks is compared. This was chosen over bounding box results, as adding bounding box accuracy introduces many more parameters and variables to the evaluation, making results much harder to compare. Classification accuracy is much simpler to evaluate. Results are collected on a validation set not augmented and not used for training. Utilizing accuracy as a metric comes with a possible drawback of bias in the evaluation, if no special care is taken. This can mainly occur in datasets which have an imbalance in data distribution. Accuracy in this format describes how many of the resulting data points were correctly classified. If one class would be significantly more common than another, a resulting accuracy, if not class specific, would hide results from a rarer class. As an example, if 90% of data points belong to the background class, a result in which all data points

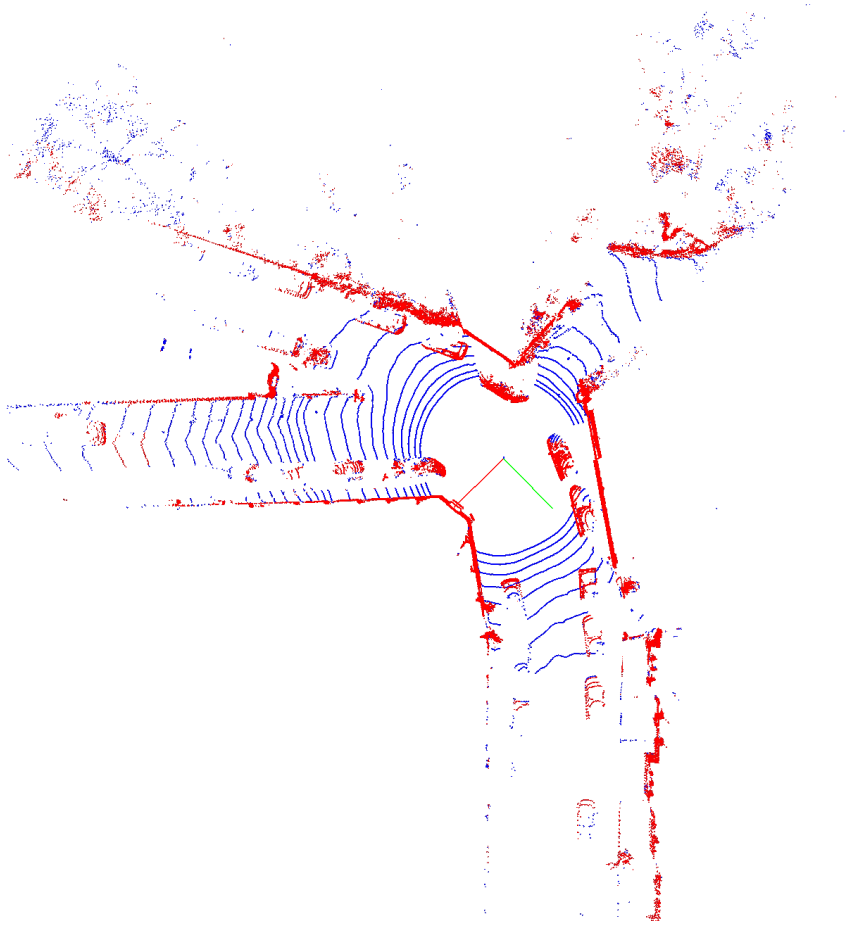


Figure 5.8.: Cluster affiliation overview of example point cloud; blue = no cluster, red = belongs to a cluster

were classified as background would show 90% accuracy, a seemingly good value, ignoring that the classification accuracy on the foreground would be 0%. Therefore care has to be taken to evaluate all classes separately and class agnostic scores are not used without recognizing this. Therefore most evaluations in this and the next section are performed classwise, each class receiving a separate accuracy score, only evaluating the data points of that class. This was accuracy can still be used as a metric and provides valuable insight into the results and highlights specific strengths – as the car class – and weaknesses like the bike class.

The first result is shown in table 5.1, where a network utilizing simply concatenated feature vectors from different time steps is compared against the same network using 3 GRU layers to combine the features from each time step. As can be seen, the GRU network outperforms the simple stacking by a significant margin. Looking at figure 5.4, this replaces the connecting part in the middle of the network, the 3 GRU layers, by 2 fully connected layers. This highlights the benefits of utilizing a

network architecture that intelligently combines features from multiple time steps, above one that processes them naively.

Further along the difference of just using data from a single time step and the combination of 5 time steps via GRU is compared in table 5.2. These results are slightly different from table 5.1, as they were collected on an experiment in which the whole point cloud was processed in large patches, while the results from table 5.1 were from a network in which only singular patches around likely objects were fed into the network. As can be seen, utilizing information from 5 time steps expectedly improves results very significantly. In figure 5.4, this would only leave the orange feature vector. More precise evaluation data is available, as the experiments were performed later and a better toolchain was implemented, allowing for classwise accuracy measurements.

Table 5.3 now displays how much benefit was generated by adding the cluster membership information and also data augmentation. Cluster membership helps classifying across the board, but mostly on the dominant classes of cars and pedestrians. This is different from data augmentation, especially the SAPCA injection, which significantly improves classification results on the classes which are of very limited occurrence in the data set. These evaluations were performed on full point clouds that were processed patch-wise. Comparing them with the results from table 5.2 is possible, but some hyperparameters and definitions changed between experiments, but at least overall class accuracy can be compared, with the 'GRU 5 Frame' result being closest to the 'No Cluster & no Aug'. It can therefore be assumed, that even with full augmentation and cluster data support, a single frame network would still not perform at the same level as the 5 frame network.

The final evaluation performed here is more focused at the application, semi-automatic labeling, described throughout this chapter. A comparison was performed on how much performance could be gained by letting the human annotator perform a pre-selection of the class he or she wants to annotate compared to letting the network not just fit a bounding box but also classify the results. In table 5.4, for the most part, the classwise networks perform significantly better on their respective class, at the cost of increased user interaction and more or less worse background suppression. This is especially prevalent in the truck network. The bike network shows surprisingly good results here compared to the rest. This might be caused by the network not being confused between bikes and other objects, as in multiclass networks, bikes are by far the rarest class.

Overall these evaluations and experiments display more or less large benefits of the improvements presented in this chapter. Especially the augmentation method

Method	Overall Class Acc.↑
Stacked Features	87.8%
GRU Features	**91.4**%

Table 5.1.: Comparing a network utilizing GRU layers with a network simply stacking feature vectors of different time steps

Method	Class Acc.↑	None Acc.↑	Car Acc.↑	Pedestrian Acc.↑
Single Frame	79.91%	80.44%	76.97%	84.38%
GRU 5 Frame	**92.25**%	**93**%	**87.14**%	**92.04**%

Table 5.2.: Comparing a network utilizing GRU layers with a network only utilizing one time step of data

which hugely improves results on underrepresented classes and also the GRU component which increases results across all classes. The impact of the patch-wise processing is difficult to evaluate on its own, as it is heavily impacted by the selection method for the patches which are fed to the network. As the application shown here utilizes manual patch selection, not much effort was put into experimenting with automatic selection. Either augmented annotation positions from the training data were utilized – shifting the patch in a way that the annotated object which should be detected is not in the center of the patch – or simply dividing the whole point cloud into large patches. A further method was shortly experimented with, which utilized the cluster centers detected by the algorithm from Hasecke et al., 2021, but this was disregarded, as results were not satisfactory. Especially background objects were heavily over represented and focusing the network on relevant objects proved difficult.

Method	Class Acc.↑	None Acc.↑	Car Acc.↑	Pedestrian Acc.↑	Truck Acc.↑	Bike Acc.↑
No Cluster & no Aug	93.9%	95.5%	83.6%	70.5%	35.1%	27.8%
Cluster & no Aug	**94.6**%	**95.8**%	88.2%	77%	41.3%	33.1%
Cluster & Aug	93.76%	94.6%	**89.3**%	**83.3**%	**50.9**%	**50.1**%

Table 5.3.: Adding Cluster Information and Data Augmentation

Method	Class Acc.↑	None Acc.↑	Car Acc.↑	Pedestrian Acc.↑	Truck Acc.↑	Bike Acc.↑
Multiclass	89%	89.6%	93.5%	89.2%	47.1%	52.9%
Car	**93.2**%	92.6%	**97**%	-	-	-
Pedestrian	83.9%	83.1%	-	**91**%	-	-
Truck	73.2%	68.9%	-	-	**82.3**%	-
Bike	92.9%	**93.5**%	-	-	-	**80.4**%

Table 5.4.: Compare Single Class Networks and Multi Class Networks

5.5 Training and Performance Evaluation of SAPCA vs Angular vs. Naive Injection

As novel point cloud augmentations were presented both in the previous chapter, in section 4.4.1, and this chapter, section 5.3.3, an extensive comparative evaluation was performed. For this, the otherwise exact same network with exactly the same settings was trained with 4 different injection methods: no injections, naive injection (inject at original position, no overlap check), wedge injection (the angular injection from section 4.4.1) and finally SAPCA. Please note that all the trainings were saved and restarted after epoch 25, to reset learning rate, hidden states and other parameters set during training. As this evaluation will show, SAPCA and wedge based injection generally perform extremely similar on foreground classes with SAPCA generally outperforming wedge based injection in overall accuracy and validation accuracy. Naive injection performs similar on all foreground classes except trucks and even performs best on bikes, but falls off in background classification. No injection at all is best at overall training accuracy and background classification but suffers majorly on rarer classes like trucks and bikes. Please note that by nature, not the same amount of objects was injected for each method. The naive method injected 5 objects per class and point cloud, while the other two had up to 15 attempts per class and point cloud, with most of them usually being rejected for overlaps. Of course it would be possible to force a given number for each point cloud, but in point clouds with a very large amount of native objects, this could increase pre-processing time significantly. All results in this chapter are given as classification accuracy for object keypoints. Overall performance is shown in figure 5.9 and figure 5.10. It is easy to see, how SAPCA outperforms all other augmentation methods and also shows a faster convergence on validation data. This is caused by SAPCA keeping the augmented point clouds very realistic. This makes it very easy for the network to transfer performance from augmented training data to unaugmented validation data. All injection based augmentations experience

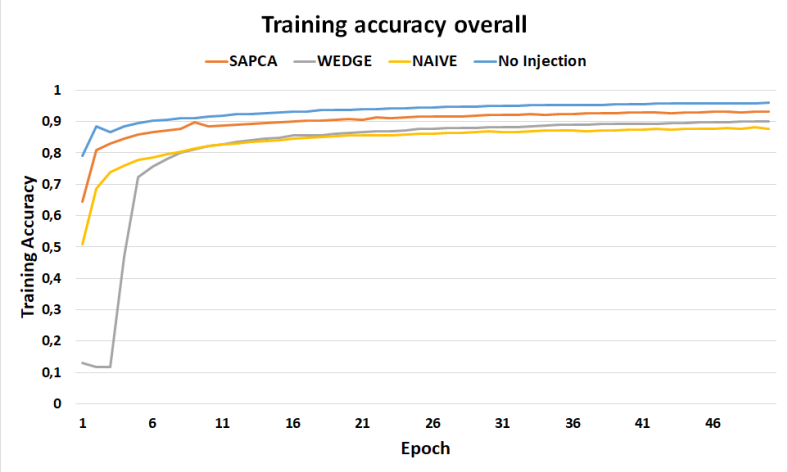


Figure 5.9.: Overall training accuracy (augmented data)

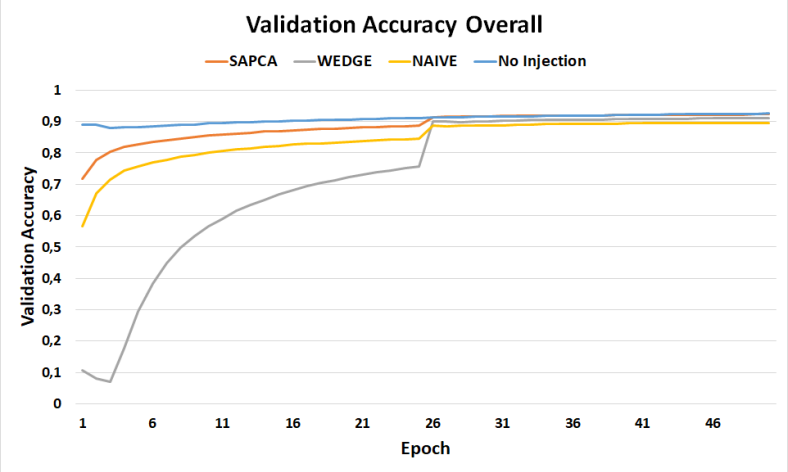


Figure 5.10.: Overall validation accuracy (unaugmented data)

a decent upward spike in performance after learning rate reset, bringing wedge based injection close to SAPCA but staying behind. Naive injection performs worst, confirming how overlaying objects and unrealistic point clouds hurt overall network performance on both, training and validation sets. Noticeably no injection at all results in the best training accuracy, which is caused by the network always having to process the same point clouds, without major change. As can be seen on the validation results, no injection, but also the two advanced injection methods, perform similarly, although wedge based injection is behind, and only catches up after the learning rate reset. The naive injection shows its weaknesses here. This is caused by the performance on the none or background class, shown in figure 5.11. Note, how all performance shown for separate classes is recorded from the unaugmented validation data. While SAPCA and wedge injection show comparable results in the

end, SAPCA learns much quicker. The problem for the naive injection is, how point clouds without proper injection shadows can make it much harder for the network to differentiate between background and foreground objects, as foreground objects can be injected in the middle of background. Of course using no injection at all performs the same, or better, than using advanced injection methods, as injection will push the network to get better at foreground classes, not at the background class. Although one goal of optimizing injection was to reduce this issue. Both advanced injection methods proove this a success, while SAPCA performs slightly better than wedge injection.

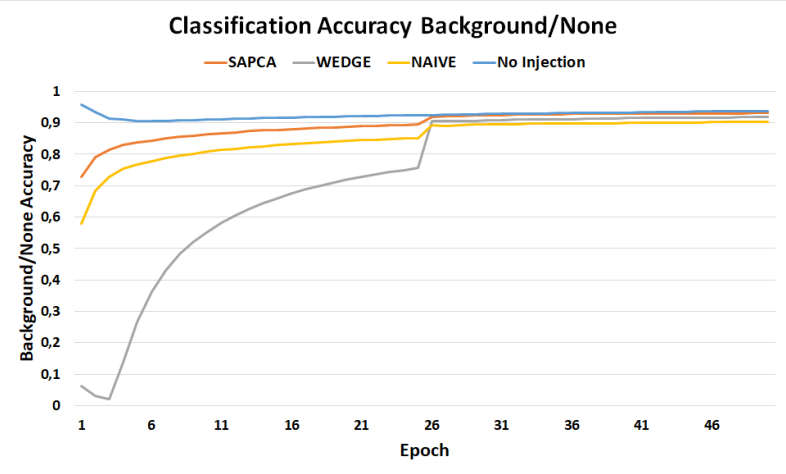


Figure 5.11.: Training accuracy on background/none class

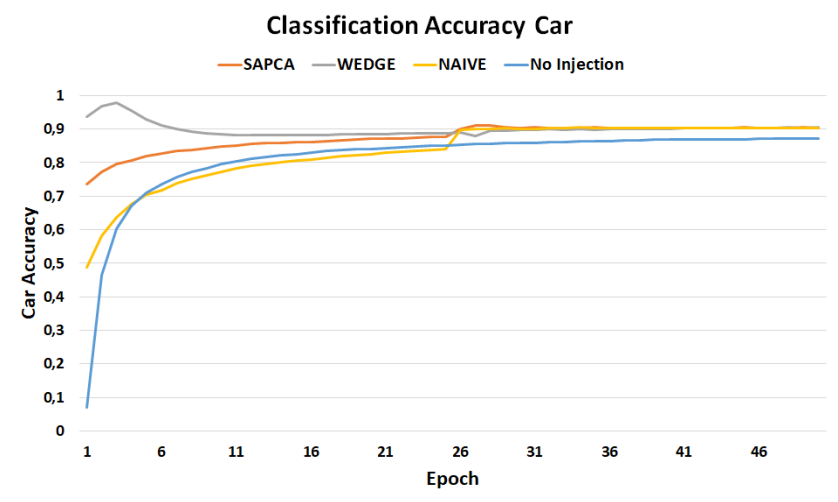


Figure 5.12.: Car accuracy(unaugmented)

Looking at the 4 major foreground classes evaluated, the results look different, but with the same conclusion or final output. Figures 5.12 to 5.15 display these results. First looking at the performance on the car class, figure 5.12. It shows how in the end

SAPCA and the wedge augmentation are performing basically the same, with wedge augmentation performing much better from the start and SAPCA only catching up slowly. The naive injection performs well in the end, but learns much slower than SAPCA or wedge based injection. The early rise and good quality of the wedge injection has two reasons. The first is luck of initialization. All of these networks are initialized with random parameters, and depending on these parameters will first focus on either optimizing the background class or the car class. This is because most relevant objects would be cars and also cars have a much larger amount of points in each object, and therefore have more influence on the average patch than some other classes. The second reason is likely how wedge injection leads to very large shadows for cars, making it easy to find them quickly. How, relatively speaking, easy cars are to detect is also shown by the training without injection, which is very close in performance, only 3% behind in the end.

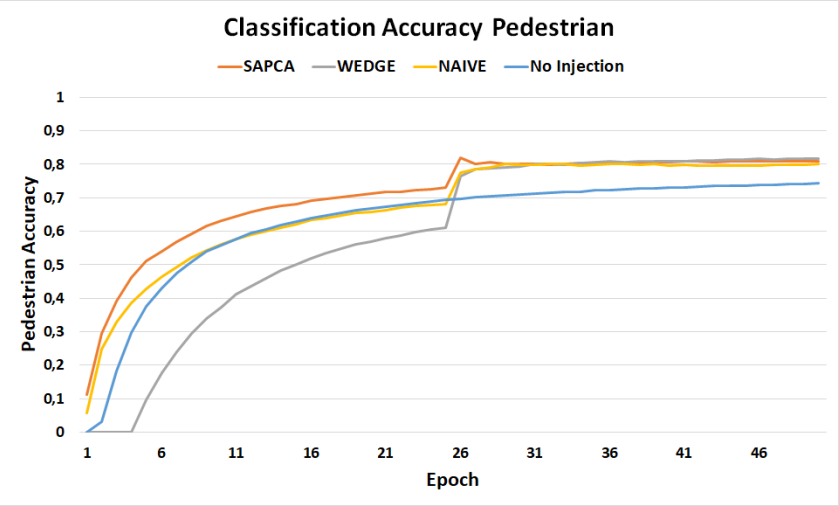


Figure 5.13.: Pedestrian accuracy(unaugmented)

Taking a look at the pedestrian class in figure 5.13, the overall behaviour is much more similar to the overall validation accuracy. In the very beginning, all three methods increase at similar speed, with SAPCA significantly ahead and especially wedge based injection performing noticeably worse. After the learning rate reset, all three injection methods perform similar, with wedge injection beating out sapca slightly. As the huge difference in performance before and after the reset shows, this is a mix of a bad initialization for wedge injection and some bad convergence for SAPCA after the reset. It jumps significantly higher than the other two methods, but due to non ideal training drops down to their level and even slightly below in the end. Nonetheless, the early performance proofs, how reliable SAPCA trains. Naive injection is again quite close which is caused by the relatively good pedestrian performance even without any injection. While the no injection algorithm is behind

the others, the difference is again only single digits. Even the relatively random injections provide a good push on this common class.

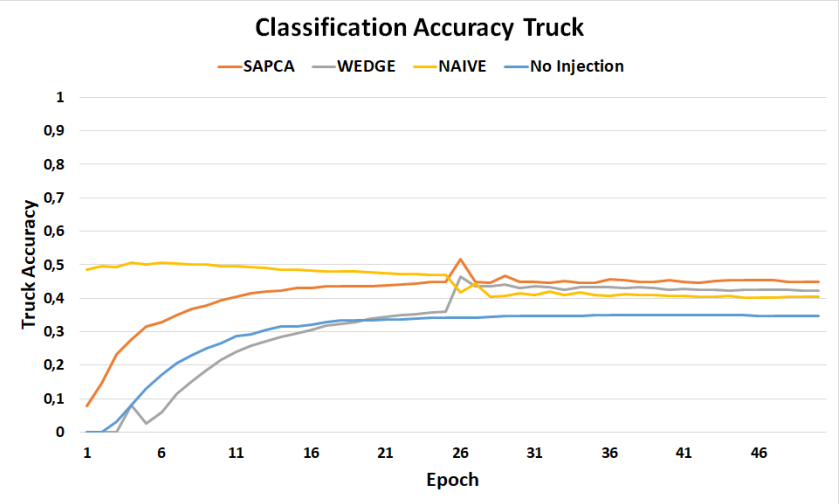


Figure 5.14.: Truck accuracy(unaugmented)

The truck class, figure 5.14, displays more interesting results. Early on the naive injection starts out at a good performance, same as wedge based injection for cars, but continuously drops, even after the learning rate reset. This is caused by the prevalence of trucks in naively augmented point clouds. Trucks are big and potentially contain a large number of data points. They are therefore occluded by many other objects and themselve occlude other objects, if occlusion is not handled in the injection method. With rising performance the results on the truck class decline for naive injection, as the network ignores some trucks to improve its background performance. SAPCA again starts out best of all methods, again very significantly better than wedge based injection, but this time keeps the best performance after the reset and after the decline of naive injection. Second best performance is achieved by wedge based injection again, but this time noticeably behind SAPCA. The network not utilizing any injection performs similarly in comparison as it did on the pedestrian class, around 10% behind the best performance. While this is still a large benefit for the augmentation methods, it is not as good as one would expect with how rare trucks are generally in point clouds. This is mainly caused, for SAPCA and wedge based injection, by trucks occluding large angular ranges, resulting in fewer injections than for other classes, which are easier to fit. The same was already shown in (table 5.3), where trucks had a much lower augmentation benefit compared to bikes.

The last evaluated class are bikes, their results shown in figure 5.15. This is the class with by far the biggest gain in performance for all 3 methods compared to

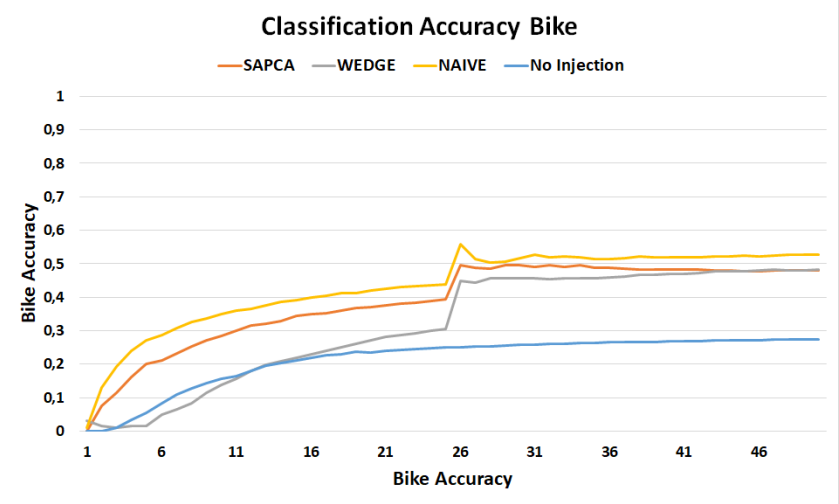


Figure 5.15.: Bike accuracy(unaugmented)

the training without any injection. Bikes are by far the rarest objects, of these 4 classes, in the utilized data set, resulting in the no injection network only seeing few bikes. Compared to trucks they are also much easier to fit inside point clouds without occluding or being occluded, allowing for more injections, resulting in a larger performance gain. From training start, naive injection and SAPCA perform similar, with naive injection performing best. As with all non-car classes, wedge based injection starts out very weak, but with a large performance gain after learning rate reset. In the end, naive injection performs best by around 5%. This is likely caused by some bias in the other two injection methods. In preprocessing bikes were injected last. For naive injection this is no issue, as the injection does not care for occlusion of pre-existing objects. For SAPCA and wedge based injection this is different. Both won't occlude other objects with injections. As a result, all three classes had their objects injected, leading to relatively crowded point clouds. As a result there is less space to inject bikes, leading to fewer bikes injected for both methods.

Overall it can be seen, how the novel SAPCA injection augmentation method outperforms older approaches, sometimes significantly, depending on the classification task. It also creates the most realistic point clouds as shown by the smallest difference in accuracy when comparing accuracy on augmented training data and unaugmented validation data. A final benefit of SAPCA compared to wedge injection is shown in the average training time for each epoch, shown in table 5.5. While these times also depend on the background load of the cluster machine used – machines were not used exclusively – they are still comparable. This shows how much faster SAPCA injections can be performed compared to wedge injections.

-	No Injection	Naive	SAPCA	Wedge
Time per Epoch↓	**2h:30min**	3h:20min	3h:50min	6h:30min

Table 5.5.: Comparing training time of each injection method. Times vary significantly between other loads on the cluster used for evaluation, these are averages, collected on Nvidia GeForce RTX 2080Ti

This is mainly caused by the method for calculating overlap. For SAPCA the overlap masks can be re-used between objects that should be injected. The final overlap check, bottom picture in figure 5.5, can be used as occlusion check, upper picture in the same figure, for the next injection. This reduces computational effort noticeably, as the native occlusion image only has to be calculated once for each point cloud and each injection step only needs to calculate the range image projection for the object that should be injected. For the wedge based injection, this is different. Each object in the point cloud separately has to be compared with the injection candidate. While the list can be reused and only has to be collected once, reducing the effort in computing angles for each object, this comparison cannot be simplified, resulting in each injection candidate taking slightly longer than the one before to check against occlusion. Naive injection without any occlusion checks is the fastest injection method. But with the large performance gain, SAPCA is very much worth the slightly increased training time, especially if the training converges quicker and can likely be stopped earlier. Naturally, using no injection augmentation at all has the shortest training time per epoch, and also the most stable training, as proven by the very steady accuracy gains, but is significantly outperformed by all 3 methods in final accuracy. The full data from which the plots were created is included in A.1

6 Lane Marker Detection with Lidar + RGB camera sensor fusion

Lidar is a viable sensor to detect lane markers on roads in situations in which camera images don't show high quality results because of glare or other reflections. An algorithm to do this is presented, in a shortened version, in section 6.1.

But as it is not computationally expensive to enrich those lidar point clouds with red, green and blue (RGB) information collected by a camera, it can be beneficial to fuse data from both sensors. This can help in some of the situations shown where lidar alone would not provide decent results, as with very rough road surfaces or differentiating between white and yellow lines in construction zones.

6.1 Introduction to Lane Marker Detection in Lidar Point Clouds

Lane markers can be detected in lidar point clouds, as they are usually painted with highly reflective paint which results in a clear differentiation to the surrounding road when looking at lidar intensity measurements. The work presented here does not employ modern machine learning methods for detecting lane markings, but rather classical approaches as presented in section 2.2.1, as it pre-dates the rise of machine learning on lidar point clouds.

6.1.1 Preparing the Point Cloud for Further Processing

A multitude of pre-processing steps has to be performed for accurate lane marker detection. The first would be limiting the point cloud to the relevant area to increase processing speed and reduce computational load. This is done by calculating a ROI and only process the points inside that ROI. The outline of this ROI curves if the

vehicle is performing a turn, allowing to detect lane markings even around corners. This outline is calculated as

$$y_i = \frac{1}{2} C_0 x_i^2 + y_O \tag{6.1}$$

with

$$x_i \in [0, 40] \tag{6.2}$$
$$C_0 = \frac{\alpha}{v_f} \tag{6.3}$$
$$\alpha = \text{yaw rate in } \frac{\text{rad}}{s} \tag{6.4}$$
$$v_f = \text{forward velocity in } \frac{m}{s}, \tag{6.5}$$

y_O being the left/right offset from the center position at which the ROI curvature is calculated. x_i gives the distances at which the offset is calculated, in this case between 0 and 40 m, as further away from the sensor the resolution gets too bad to reliably detect lane markings.

The second part of pre-processing is rasterization of the point cloud. As this is already described in section 4.2.1, no more detail will be given here, other that in case of multiple data points falling into one cell, the mean intensity $\bar{i}_{x,y} = \frac{1}{N} \sum_{i=1}^{N} i_{x,y}^{(n)}$ is used as intensity for the grid cell. As for resolution, $r_x = 0.3$ m and $r_y = 0.02$ m sized cells are used, with 40 m range in x-direction, and 14 m in y-direction, resulting in 705 x 134 cells. While the mean intensity might theoretically smooth gradients and make it harder to detect lane markings, the resolution is high enough, to still reproduce the required detail. Using the maximum value would on the other hand be prone to noise. The z-component is squashed down as pillars, as in PointPillars (Lang et al., 2019) are used. Nt all grid cells will contain data points, therefore both, intensity and height information per cell are interpolated from surrounding data points, as

$$\bar{i}_{x,y} = \frac{\bar{i}_{x-1,y-1} + \bar{i}_{x-1,y} + \bar{i}_{x,y-1}}{6} \tag{6.6}$$
$$\bar{h}_{x,y} = \frac{\bar{h}_{x-1,y-1} + \bar{h}_{x-1,y} + \bar{h}_{x,y-1}}{3}, \tag{6.7}$$

resulting in a constant value decline, for $\bar{i}_{x,y}$ in larger, empty areas. This is achieved by dividing the sum of intensities by 6, even though only 3 values are summarized. For $\bar{h}_{x,y}$ this decline is not desired, therefore the accurate mean is calculated. This gap filling follows a left to right direction to not enrich the influence of empty neighbours on this mean. This way it can always be guaranteed that all 3 source

data cells are filled with information. If a larger, full 3 x 3, kernel was used, empty cells would influence the mean calculation, which is undesired.

Contrast stretching is applied across 3 distinct zones – 0 m - 25 m, 25 m - 35 m and 35 m - 40 m –, defined by x-coordinate, as data density and mean point intensity differs massively. The cutoff fraction parameter $c_p = 0.03$, and overall algorithm, is taken from Fisher et al., 1996. To actually calculate the cutoff value, and therefore lower and upper limits, a discretized, to 8 bit, histogram of the intensities in the point cloud zones is generated. The peak of the histogram, h_M is extracted and with it and c_p the cutoff value $c = c_p * h_M$ is calculated. With this cutoff value, the upper, h_v, and lower, l_v, limits of the histogram are extracted, by taking the highest, and respectively lowest, value that appears more often than c. Upper and lower target values, $t_H = 1$ and $t_l = 0$ are set. With these pre-calculated variables, the final contrast at each grid cell position is calculated as

$$i_{x,y} = (i_{x,y} - l_v)\frac{t_h - t_l}{h_v - l_v + t_l}. \tag{6.8}$$

In the resulting point cloud, the full intensity range between 0 and 1 is used, allowing for stronger gradients. A sketch of this procedure, for one of the range based zones, is shown in figure 6.1. The sketch also displays the large change in intensity distribution from the original, shown in red, to the stretched intensities, green. The result on a real point cloud is shown by figure 6.2.

As a final step in pre-processing, the ground plane has to be detected, and points not belonging to the ground plane need to be filtered, as those can produce noise. Unfortunately it would be too easy an assumption to only estimate a plane ground plane, as the road surface is usually slightly curved down at the sides of the road. As a result a more complex cutout from a cylindrical figure has to be estimated, as usually also the height of the road towards the sensors changes with distance. A RANSAC algorithm, as described by Beneš et al., 2011 is used for this. One problem with RANSAC lies in the fact, that complex structures are very costly to estimate, as the full possible set of equations has to be calculated for a large amount of randomly selected data points. For a cylindrical plane this can be time consuming. Therefore the problem was split in two, and the curvature is estimated separately from the height change. For the height change a simple linear equation is estimated at the $y = 7$ m position in front of the sensor. The curvature is slightly more complex, therefore a second degree polynomial is estimated at a fixed distance, $x = 5$ m in front of the sensor. While this is not perfect, as the curvature might change across the length of the ROI, it works well enough in practice. All points outside 0.15 m of the estimated ground plane are removed from the point cloud, therefore only

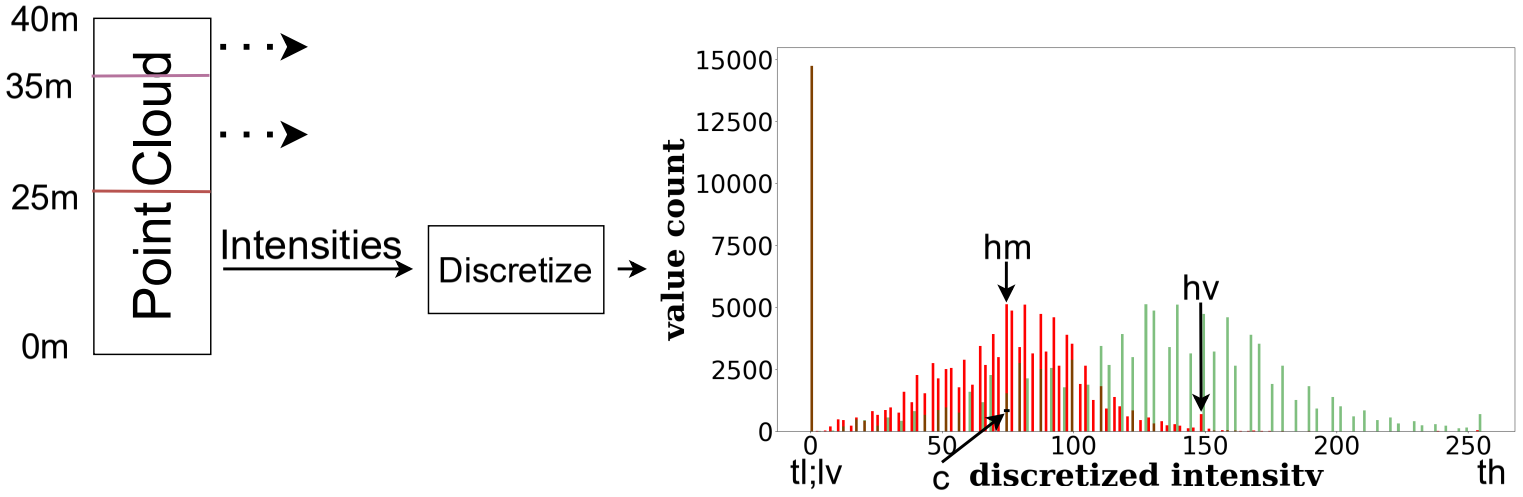


Figure 6.1.: Procedure of point cloud intensity stretching. Red = original intensity distribution, green = distribution after stretching to full 8 bit range from 0 to 255.

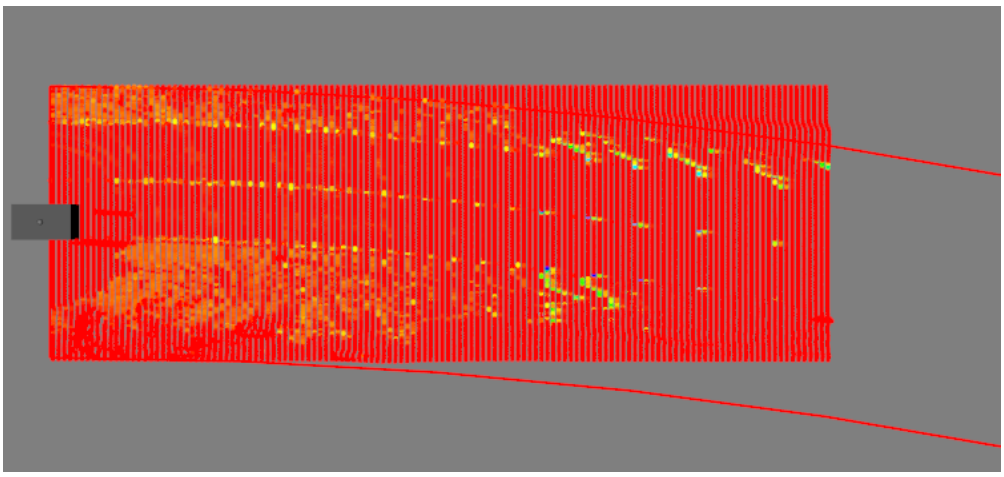


Figure 6.2.: Point cloud after ROI, rasterization and contrast stretching. Brightness encodes point intensity. Image from Alsfasser, 2017

an almost flat cut from the point cloud remains. The concept of this is shown in figure 6.3, with results shown in figure 6.4. It can clearly be seen, how the road surface is curved, and how the ground plane estimation adapts and removes outliers from the resulting point cloud.

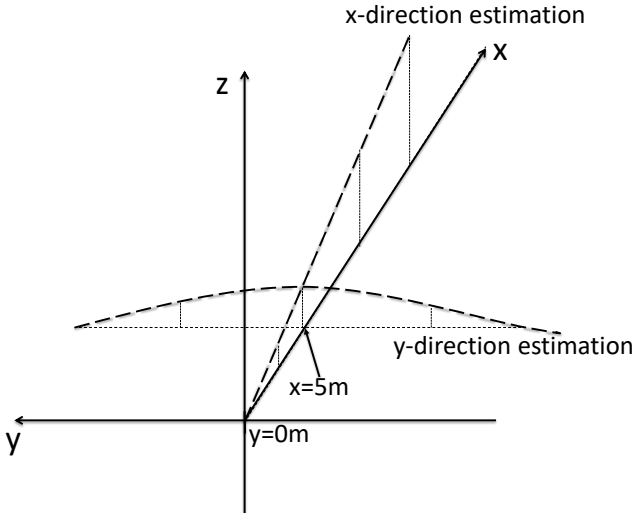


Figure 6.3.: Diagram of RANSAC use for ground plane extraction. Image from Alsfasser, 2017

y-Direction Ground Plane Estimation

In the y-direction of the ground plane, a quadratic curve

$$z = y_{P2}y^2 + y_{P1}y + y_{P0} \tag{6.9}$$

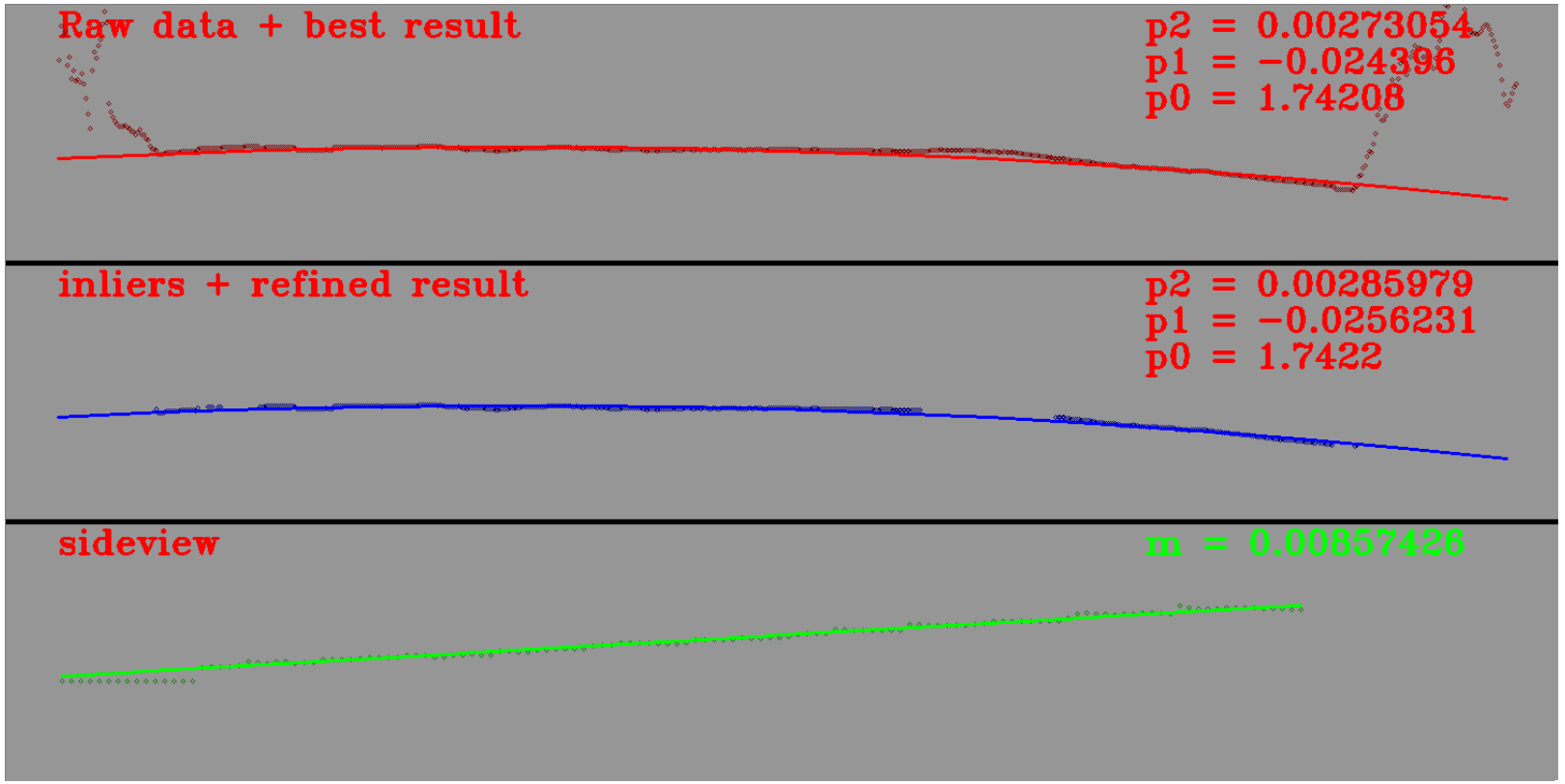


Figure 6.4.: Result of ground plane estimation, including estimation parameters. Top/leftmost/red plot shows the best estimation on the raw data at $x = 5$ m, the middle/blue plot the result on inlier data and the bottom/rightmost/green the linear incline estimation at $y = 7$ m

is estimated at $x = 5$ m in front of the sensor. This distance was chosen by trial, as it is a distance at which the point cloud is extremely dense, but there are no missing parts, like a circle around the ego vehicle where the ground can't be seen. This deadzone would otherwise have to be interpolated, resulting in flat spots. For this a non-linear RANSAC is chosen, as presented by Beneš et al., 2011. 3 data points out of 705 at $x = 5$ m are selected and setup to solve

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix} = \begin{pmatrix} 1 & y_0 & y_0^2 \\ 1 & y_1 & y_1^2 \\ 1 & y_2 & y_2^2 \end{pmatrix} \begin{bmatrix} y_{P0} \\ y_{P1} \\ y_{P2} \end{bmatrix}. \tag{6.10}$$

Solving for y_{P0}, y_{P1} and y_{P2} by

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix} * \begin{pmatrix} 1 & y_0 & y_0^2 \\ 1 & y_1 & y_1^2 \\ 1 & y_2 & y_2^2 \end{pmatrix}^{-1} = \begin{bmatrix} y_{P0} \\ y_{P1} \\ y_{P2} \end{bmatrix}. \tag{6.11}$$

With

$$\begin{pmatrix} 1 & y_0 & y_0^2 \\ 1 & y_1 & y_1^2 \\ 1 & y_2 & y_2^2 \end{pmatrix}, \tag{6.12}$$

being a Vandermonde matrix (Weisstein, 2021b). Details on the Vandermonde matrix, including its inverse, are given, among others, by Knuth, 1997, p. 37-38. This results in the generalized solution of the parameters y_{P0}, y_{P1} and y_{P2} being calculated as

$$y_{P0} = \frac{-(-z_2 y_0^2 y_1 + z_1 y_0^2 y_2 + z_2 y_0 y_1^2 - z_1 y_0 y_2^2 - z_0 y_1^2 y_2 + z_0 y_1 y_2^2)}{(y_0 - y_1)(y_0 - y_2)(y_1 - y_2)} \tag{6.13}$$

$$y_{P1} = \frac{(y_0^2 z_1 - y_1^2 z_0 - y_0^2 z_2 + y_2^2 z_0 + y_1^2 z_2 - y_2^2 z_1)}{(y_0 - y_1)(y_0 - y_2)(y_1 - y_2)} \tag{6.14}$$

$$y_{P2} = \frac{-(y_0 z_1 - y_1 z_0 - y_0 z_2 + y_2 z_0 + y_1 z_2 y_2 z_1)}{(y_0 - y_1)(y_0 - y_2)(y_1 - y_2)}. \tag{6.15}$$

This is done a fixed number of times, with fresh data points. For each iteration, all 705 data points are checked against the produced polynomial, collecting the number of inliers by calculating the distance

$$d = |(y_{P2} y_c^2 + y_{P1} y_c + y_{P0} - z_c)| \tag{6.16}$$

with

$$z_c = \text{z-position of control point} \tag{6.17}$$

$$y_c = \text{y-position of control point} \tag{6.18}$$

$$\tag{6.19}$$

and adding the point to the list of inliers, if $d \leq 0.2$. The best estimate is collected, by choosing the estimation with the lowest sum of d over all control points. To get the final output, the inliers of the best estimate are fed into another set of equations

$$\begin{bmatrix} z_0 \\ \vdots \\ z_{N-1} \end{bmatrix} + \begin{bmatrix} e_0 \\ \vdots \\ e_{N-1} \end{bmatrix} = \begin{pmatrix} y_0^2 & y_0 & 1 \\ \vdots & \vdots & \vdots \\ y_{N-1}^2 & y_{N-1} & 1 \end{pmatrix} \begin{bmatrix} y_{P2} \\ y_{P1} \\ y_{P0} \end{bmatrix} \tag{6.20}$$

to find the best possible parameters for all inliers. Equation 6.20 is solved, as suggested by Beneš et al., 2011, with singular value decomposition (SVD), as, among others, by Henry and Hofrichter, 1992, Chapter 6, Part III. The method itself shall not be explained here.

x-Direction Ground Plane Estimation

The ground plane estimation in x-direction is a simpler version of the estimation in y-direction, as it can be estimated by a linear equation

$$z = x_{P1}x + x_{P0} \tag{6.21}$$

at the $y = 0$ m position of the vehicle or sensor.

$$\begin{bmatrix} z_0 \\ z_1 \end{bmatrix} = \begin{pmatrix} x_0 & 1 \\ x_1 & 1 \end{pmatrix} \begin{bmatrix} x_{P1} \\ x_{P0} \end{bmatrix} \tag{6.22}$$

resolves to

$$x_{P0} = y_0 - \frac{z_1 - z_0}{x_1 - x_0} x_0 \tag{6.23}$$

$$x_{P1} = \frac{z_1 - z_0}{x_1 - x_0}, \tag{6.24}$$

with $x_{P0} = -1.73$ m fixed, as that is the distance between sensor and street, so it is a fixed origin point, ignoring possible slight deviations because of the suspension of

the vehicle compressing or extending for bumps in the road. Inlier distances are calculated with

$$d = |(x_{P1}x_c + x_{P0} - z_c)| \tag{6.25}$$

and

$$z_c = \text{z-position of control point} \tag{6.26}$$
$$x_c = \text{x-position of control point.} \tag{6.27}$$

Again, all N inliers of the best hypothesis are taken and

$$\begin{bmatrix} z_0 \\ \vdots \\ z_{N-1} \end{bmatrix} + \begin{bmatrix} e_0 \\ \vdots \\ e_{N-1} \end{bmatrix} = \begin{pmatrix} x_0 & 1 \\ \vdots & \vdots \\ x_{N-1} & 1 \end{pmatrix} \begin{bmatrix} x_{P1} \\ x_{P0} \end{bmatrix} \tag{6.28}$$

is again solved by SVD for the best possible approximation.

Height Filtering

With both estimates done and parameters for both x-and y-direction of the ground plane being calculated, the final filtering is performed as

$$y_c = y_{x,y} \tag{6.29}$$
$$x_c = x_{x,y} - x_O \tag{6.30}$$
$$z_b = y_{P2}y_c^2 + y_{P1}y_c + y_{P0} \tag{6.31}$$
$$z_g = z_b + x_{P0}x_c \tag{6.32}$$
$$d = |z_g - z_{x,y}| \tag{6.33}$$

with

$$x_{x,y} = \text{x-position of cell at raster position x,y} \tag{6.34}$$
$$y_{x,y} = \text{y-position of cell at raster position x,y} \tag{6.35}$$
$$z_{x,y} = \text{z-position of cell at raster position x,y} \tag{6.36}$$
$$x \in [1, N_x] \tag{6.37}$$
$$y \in [1, N_y] \tag{6.38}$$
$$x_O = \text{Offset, as approximation is calculated } 5 \text{ m in front of the sensor,} \tag{6.39}$$

with all cells where $d \geq 0.15$ are discarded by setting their intensity to 0. With this, the point cloud is finally cleaned, contrasts are enhanced and noise is filtered.

6.1.2 Lane Marker Detection

Same as the pre-processing, the actual marker detection is a multi step process, consisting of different filter steps and further processing. In the first filter step a large 7x9 sized filter kernel

$$K = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix} \tag{6.40}$$

is convoluted with the pre-processed point cloud. The result is a gradient map with, because of the large kernel, relatively wide and smooth gradients at the edges of lane markings, and unfortunately false positive objects. For thresholding at a later time, a mean absolute gradient $g_t = \frac{1}{N} \sum_{x=1}^{N_x} \sum_{y=1}^{N_y} |g_{x,y}|$ is calculated, with $g_{x,y}$ being the gradient at the given raster position.

To reduce the number of false positives an assumption is made. It is assumed, that lane markings are usually only present on flat surfaces. To enforce this, a variance filter is applied on the height information stored in the grid cells. A 10 cell sliding window approach is used to calculate the height variance around each cell in the grid as

$$\bar{h}_{x,y} = \frac{1}{N} \sum_{n=y-5}^{y+5} z_{x,n} \tag{6.41}$$

$$\sigma_{x,y}^2 = \frac{1}{N} \sum_{n=y-5}^{y+5} (z_{x,n} - \bar{h}_{x,y})^2, \tag{6.42}$$

with edge cases being handled by reducing the sliding window. If the variance is too high, $\sigma_{x,y}^2 \geq 0.05$, meaning the cell lies close to or inside of noisy areas like vegetation, the cell gradient is reset to 0, effectively removing the cell from lane

marker consideration. Unfortunately this also removes curbs from consideration, but, while interesting for detecting overdriveable area, these were no focus here. Results of this are shown in figure 6.5, which shows how mostly lane markings show high intensity after applying both filters to the point cloud. Some noise, esp. far away from the ego vehicle is present, but removed in later steps.

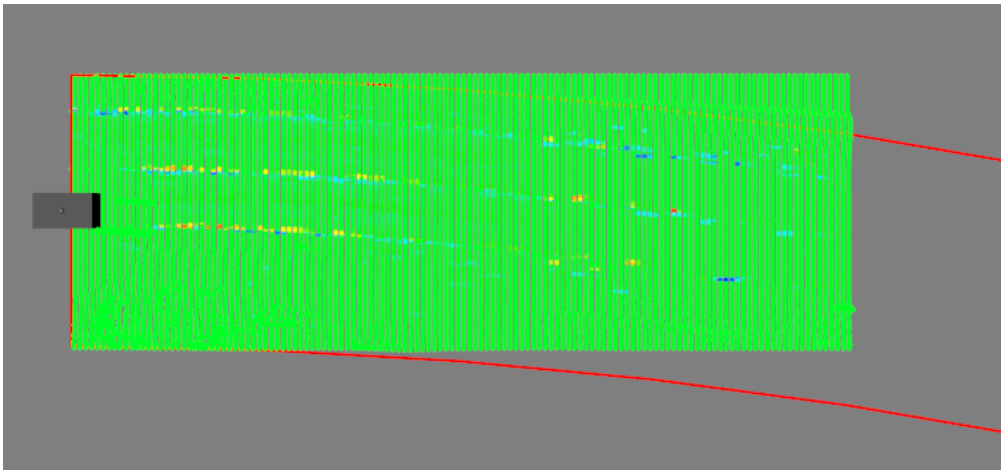


Figure 6.5.: Point cloud after filtering with variance filter and detection kernel. Image from Alsfasser, 2017

As explained previously, the lane markings produce relatively wide zones of high gradients as this helps smoothing out the results. As a first step, only gradients higher than g_t are further processed. This ensures fast enough computation, as the number of processed grid cells is massively reduced. The threshold is dynamically calculated, as road surface and other environmental circumstances influence the overall intensity of all points of the point cloud. For actual detection gradients have to be thinned out to a single gradient value for each change between road and lane marking. Additionally left and right (rising and falling) gradients are paired together, again to increase stability.

The resulting lane marking detections can be spotty and, as they are only positions in a grid, are not really usable in later processing steps. For this, short line segments are fit on these points, again by a RANSAC algorithm. Alternatively the Hough transformation (Duda and Hart, 1972), could be used but would be to computationally expensive, as it needs to calculate every possible line through the data points and record angle θ and length ρ, which can be plotted and maxima in the parameter space selected, as described in section 2.2.1. RANSAC works in a similar way, but the runtime requirement can be controlled by the number of iterations performed, and the randomly selected points don't need to be fully random, therefore the algorithm can be more targeted that Hough transformation brute forcing every possible line

in the region. As a lane marking is not straight, estimating the full lane marking at once would again be very difficult and time consuming, as a high dimensional polynomial would have to be fitted, requiring a lot of control points at each step. To alleviate this issue the rasterized, and gradient processed, data grid is split into 15 separate, overlapping ROIs. These are not always the same size, with ROIs close to the sensor being 3 m large and the ones furthest away being 8 m. They are processed independently, with data points/cells being processed in multiple ROIs at once. For each iteration in a given ROI, 2 mostly random data points, being pairs of left and right gradient, are selected, following a number of limitations:

- **Gradient distance**: The gradient values between both data points should not differ too much, as it is assumed that over a short distance, the lane marking is roughly constant in intensity
- **Euclidean distance**: Points should not be more than 1.25x the ROI length apart. This prohibits matching points that don't belong to the same marking, as the y component of the distance is too large
- **Gradient pair width distance**: Lane marking width, or distance between left and right gradient edge, should be roughly the same, as it won't vary much over a short distance
- **Angle difference between points**: The points should describe a similar angle between both matched points, as otherwise it is suggested that they don't belong to the same lane marking
- **Angle difference to x-axis**: The points should also describe close to the same angle between the data point and the x-axis, as again, otherwise different lane markings might be described

After selecting 2 valid data points a line hypothesis is created, and the validity is tested, by calculating the euclidean and gradient distance between the hypothesis and the other data points in the ROI. The euclidean distance calculates the 3 dimensional distance

$$d_p = \frac{|(\mathbf{p}_c - \mathbf{p}_s)x(\mathbf{p}_c - \mathbf{p}_e)|}{|\mathbf{p}_e - \mathbf{p}_s|}, \tag{6.43}$$

while the gradient distance is calculated trivially as

$$d_g = \left| \frac{(\Delta g_e + \Delta g_s)}{2} - \Delta g_c \right| \tag{6.44}$$

with $\mathbf{p}_{[s|e|c]}$ being the position of the hypothesis start, end and the tested control point and $\Delta g_{[e|s|c]}$ the absolute gradient distance between left and right edge of the end

point, the start point and the control point. All hypothesises with enough inlier control points are collected and added to the total collection of line segments. As final output these hypothesises are clustered and the overlap between processing segments is being removed. As a result 9 ROIs remain. Hypothesises closer than 0.2 m in y-direction, and having similar slope, line direction and so on, are clustered together into a maximum of 3 bins for 3 different directions. A ROI can contain multiple of these cluster collections. All hypothesises inside of each bin are finally averaged to create 1 final output line for each bin. This is preferred over simply selecting the hypothesis with most inliers, as it is more robust against outliers and the slight error introduced by averaging is removed by the tracker described in section 6.1.3. The result is shown in figure 6.6, where the perspective from previous screenshots of the birds eye view point cloud has changed to line segments projected into a camera image. This is mainly done for easier viewing. It can clearly be seen, how the lane markings are very well represented and, in this example, no noise is left.

6.1.3 Refining a Spline Model to Track Lane Markings

The lane marking detections described in section 6.1.2 are tracked over multiple frames to increase stability and reliability. As there is no fixed number of track segments and no fixed lane marker positions, tracks have to be approximated from lane marker segments. Therefore lane markers are represented by Catmull-Rom splines made up from 5 different control points 12 m apart. 4 of these control points are in front of the sensor/vehicle, 1 is behind it. All tracks share x-positions $l_i, \quad i \in [0, 4]$ for their control points, therefore the 5 x-positions are recorded separately, and the positions themselves are updated at each time step to represent the forward movement of the vehicle. As a result they constantly shift closer and closer to the vehicle, which would quickly result in the number of control points in front reducing. Therefore as soon as a second control point drops below $x = 0$ m, the control point furthest behind the vehicle is removed and replaced by a new control point in front of the vehicle. This is shown in figure 6.7. The position is extrapolated from the 4 remaining control points. This model is taken from Zhao et al., 2012, but extended here to 3D, by including the z-incline or decline in the state estimation. It was chosen as it provides enough flexibility to model lane markings, while being defined by a limited number of parameters, simplifying tracking. The Catmull-Rom spline is composed of 3rd degree polynomials

$$f(s) = \frac{C_1 s^3}{6} + \frac{C_0 s^2}{2} + as + b, \tag{6.45}$$

Figure 6.6.: After fitting of line segments and clustering them. Image from Alsfasser, 2017

between sets of 2 control points. Additionally the z-slope and the mean width of each lane marking are tracked, resulting in the final tracker state

$$\mathbf{x} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ w \\ m_z \end{bmatrix}. \tag{6.46}$$

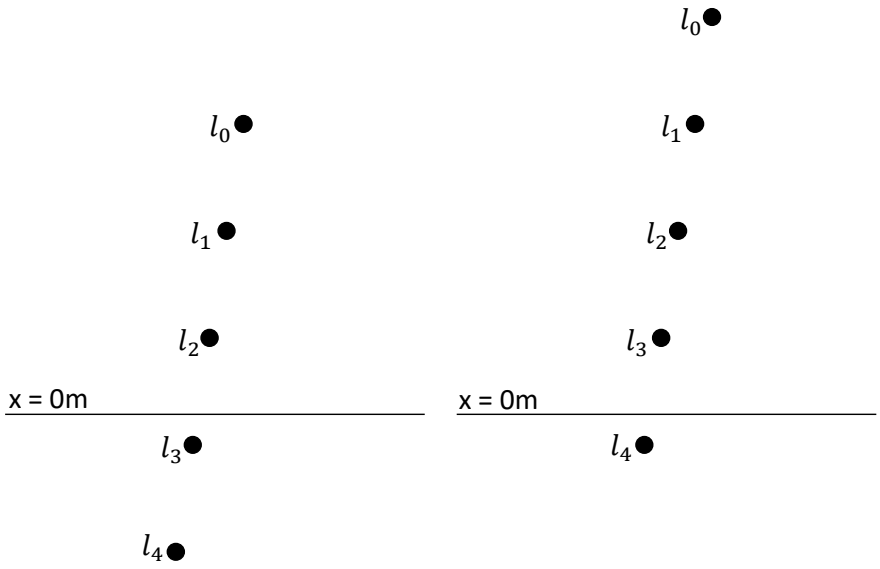
For tracking the well known Kalman filter, as described in section 2.1.1, is used. It has the advantage of being well understood and being easy to implement and to modify, while still being very powerful in linear state estimation, as performed here. While it can be extended (EKF, UKF) for non-linear estimations, this is not required here. The only major disadvantage of the Kalman filter is the fact, that each track needs a specific assigned measurement, as tracks can only be updated with one measurement at each time. One way to solve this would be using a joint probabilistic data association filter (JPDA), as by Bar-Shalom et al., 2009. In this case it would provide no major benefit to the quality of the produced tracks, as each track, or each lane marking, at each control point should only be represented by one measurement anyways. To assign these measurements, the auction algorithm, Bertsekas, 1988, was implemented.

Track Prediction

As previously explained, the first step of prediction is moving the control points closer, and behind, the vehicle, to simulate the forward motion. This is done linearly by calculating the driven distance, in x-direction, $d = v_f \Delta t + \frac{a_f \Delta t^2}{2}$, with v_f the forward velocity in $\frac{m}{s}$ and a_f in $\frac{m}{s^2}$ the forward acceleration at the time of the last measurement, and moving all control point x-positions by simply subtracting d. Predicting the y-positions, and therefore the track state, can best be described by a multi stage approach in this scenario, as

$$\hat{\mathbf{x}}_{tmp} = F_k \hat{\mathbf{x}}_{(k|k)} \tag{6.47}$$

$$\hat{\mathbf{x}}_{(k+1|k)} = U_k \mathbf{g}_k \tag{6.48}$$



(a) k - 1 (b) k

Figure 6.7.: Example of control point being removed behind the vehicle and injected in front of the vehicle. Image from Alsfasser, 2017

with

$$F1_k = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad F2_k = \begin{pmatrix} 3 & -3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \tag{6.49}$$

with $F_k = F1_k$ in case no new control point had to be extrapolated and $F_k = F2_k$ in case a new point had to be interpolated. The main part of the state prediction is the rotation + possible shift described in equation 6.48, with components

$$U_k = \begin{pmatrix} l_0 & \hat{\mathbf{x}}_{tmp}(0) & b_0 & 0 & 0 & \Delta t v_l \\ l_1 & \hat{\mathbf{x}}_{tmp}(1) & b_1 & 0 & 0 & \Delta t v_l \\ l_2 & \hat{\mathbf{x}}_{tmp}(2) & b_2 & 0 & 0 & \Delta t v_l \\ l_3 & \hat{\mathbf{x}}_{tmp}(3) & b_3 & 0 & 0 & \Delta t v_l \\ l_4 & \hat{\mathbf{x}}_{tmp}(4) & b_4 & 0 & 0 & \Delta t v_l \\ 0 & 0 & 0 & \hat{\mathbf{x}}_{tmp}(5) & 0 & 0 \\ 0 & 0 & 0 & 0 & \hat{\mathbf{x}}_{tmp}(6) & 0 \end{pmatrix} \tag{6.50}$$

and

$$\mathbf{g}_k = \begin{bmatrix} \sin(\Delta t\varphi)\cos(\Delta t\vartheta) \\ \cos(\Delta t\varphi) + \sin(\Delta t\varphi)\sin(\Delta t\vartheta)\sin(\Delta t\varrho) \\ \cos(\Delta t\varphi)\sin(\Delta t\varrho) + \sin(\Delta t\varphi)\sin(\Delta t\vartheta)\cos(\Delta t\varrho) \\ 1 \\ 1 \\ -1 \end{bmatrix}, \tag{6.51}$$

which describes the y-component of a 3 way rotation of a vector $[x\ y\ z]^T$ by yaw rate φ, pitch rate ϑ and roll rate ϱ, all 3 in $\frac{\text{rad}}{s}$, for the choosen coordinate system.

For this, $b_i, \quad i \in [0,4]$ is the interpolated z-position at each of the control point distances l_i and v_l is the leftward velocity of the vehicle. Finally the state covariance is predicted as

$$P_{(k+1|k)} = F_k P_{(k|k)} F_k^T + Q_k, \tag{6.52}$$

with Q_k being estimated noise per control point position. In case a new control point has to be extrapolated it is done from the assumption of a 3rd degree polynomial, reduced to a 2 dimensional polynomial, as it is assumed that the curvature does not change from the extrapolation and therefore $C_1 = 0$, resulting in

$$\hat{\mathbf{x}}_{(k|k)}(0) = (\frac{s^2}{2} \quad s \quad 1) \begin{pmatrix} \frac{s_1}{2} & s_1 & 1 \\ \frac{s_2}{2} & s_2 & 1 \\ s_2 & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{pmatrix} \begin{bmatrix} \hat{\mathbf{x}}_{(k|k)}(1) \\ \hat{\mathbf{x}}_{(k|k)}(2) \\ \hat{\mathbf{x}}_{(k|k)}(3) \end{bmatrix}, \tag{6.53}$$

with

$$s_1 = 1, \text{Normalized x-position of furthest away control point} \tag{6.54}$$

$$s_2 = 0, \text{Normalized x-position of second furthest away control point} \tag{6.55}$$

$$s = 2, \text{Normalized x-position of new, extrapolated control point.} \tag{6.56}$$

Measurement Model

Each measurement is a four dimensional vector

$$\mathbf{z}_{k+1} = \begin{bmatrix} p_y \\ \arctan(m_y) \\ w \\ p_z \end{bmatrix}, \tag{6.57}$$

with p_y the y-coordinate at the middle of the detected line segment, m_y the slope in y-direction of the line detection, w the width, as long as the segment is closer than 10 m as otherwise the low horizontal resolution of the lidar becomes a problem, and p_z the z-coordinate at the middle of the line segment. For the Kalman filter update, a predicted measurement $\hat{\mathbf{z}}_{(k+1|k)}$ has to be calculated from the predicted state $\hat{\mathbf{x}}_{(k+1|k)}$, to calculate the innovation, described in equation 2.3. The predicted measurement has to be calculated at the exact x-position of the detected lane marker, meaning in a first step the corresponding spline segment has to be selected. The parameters of the given spline segment can then be calculated as

$$\begin{bmatrix} C_1 \\ C_0 \\ a \\ b \end{bmatrix} = A_i^{-1} B \mathbf{b}_i, \tag{6.58}$$

following Zhao et al., 2012, with the index i describing the spline segment in which the measurement lies. Components A, B and $\mathbf{b}_i$ are defined as

$$A_i = \begin{pmatrix} \frac{s_{n_i}^3}{6} & \frac{s_{n_i}^2}{2} & s_{n_i} & 1 \\ \frac{s_{n_i}^2}{2} & s_{n_i} & 1 & 0 \\ \frac{s_{n_{i-1}}^3}{6} & \frac{s_{n_{i-1}}^2}{2} & s_{n_{i-1}} & 1 \\ \frac{s_{n_{i-1}}^2}{2} & s_{n_{i-1}} & 1 & 0 \end{pmatrix} \tag{6.59}$$

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} \end{pmatrix} \tag{6.60}$$

$$\mathbf{b}_i = \begin{bmatrix} \hat{\mathbf{x}}_{(k+1|k)}(i+1) \\ \hat{\mathbf{x}}_{(k+1|k)}(i) \\ \hat{\mathbf{x}}_{(k+1|k)}(i-1) \\ \hat{\mathbf{x}}_{(k+1|k)}(i-2) \end{bmatrix} \tag{6.61}$$

with

$$s_{n_i} = \frac{s - l_{i-1}}{l_d}, \tag{6.62}$$

with $f(s)$ the y-coordinate of the spline segment at position s, s the x-position for which the spline position is predicted, l_{i-1} the x-coordinate of the control point before s and finally l_d the distance between 2 control points. To prevent possibly negative indices from $i-1$ or $i-2$, measurements can only be assigned directly to few of the

possible bins, as can be seen from figure 6.7. For bins outside of that range, the spline definition from the closest possible segment is extrapolated. Additionally it is assumed that the width and z-position of the lane marking don't change, resulting in a final predicted measurement vector

$$\hat{\mathbf{z}}_{(k+1|k)} = \begin{bmatrix} f(s) \\ \arctan\left(\frac{\partial f(s)}{\partial l}\right) \\ \hat{\mathbf{x}}_{(k+1|k)}(5) \\ s\hat{\mathbf{x}}_{(k+1|k)}(6) + p_{z_0} \end{bmatrix}, \tag{6.63}$$

with $p_{z_0} = -1.73$ m as z-position of the road at the sensor position. Note that for measurements further than 10 m away, the width is not updated at all and the value currently in the track state is re-used.

Kalman update

The track update closely follows the standard Kalman update as described in equation 2.3 to equation 2.7, with only the measurement covariance R_{k+1} and the observation matrix $H_{(k+1)}$ having to be calculated. The observation matrix $H_{(k+1)}$ is defined as Jacobian matrix of the predicted measurement being derived after the state, following Weisstein, 2021a. R_{k+1} is shaped as 4 x 4 diagonal matrix, with the diagonal containing differently scaled covariances for each of the measurements.

Measurement - Track Assignment

As the Kalman filter can only handle 1 measurement per update, it is important to correctly assign a measurement to each track. A first evaluation was performed using nearest neighbour assignments with the Mahalanobis distance between track and measurement as optimizer input, but this resulted in some issues, as it would only be local optimization, leading to issues at situations where multiple valid assignments would be possible. For a globally optimized assignment approach the decision had to be made between the hungarian algorithm (Kuhn, 1955) and the auction algorithm (Bertsekas, 1988). The auction algorithm, specifically an extension published by Bertsekas and Castanon, 1992, was used. This extension has the advantage of working natively with asymmetric problems, more measurements than

tracks. The assignment is done by maximizing the sum of likelihoods $\sum a_{i,j}$, the likelihood of measurement j to track i. $a_{i,j}$ defined as

$$a_{i,j} = ln(f_{t_i}[z_j(k+1)]) \tag{6.64}$$

$$f_{t_i}[z_j(k+1)] = \frac{e^{-\frac{1}{2}(z_l(k+1)-\hat{z}(k+1|k))^T S^{-1}(k+1)(z_l(k+1)-\hat{z}(k+1|k))}}{\sqrt{|(2\pi)^4 S(k+1)|}}, \tag{6.65}$$

resulting from simplifying max likelihood

$$p[Z(k+1)|\theta(k+1), Z^k] = \prod_{l=1}^{m(k+1)} f_{t_{i_l}}[z_{j_l}(k+1)]^{\tau_l} V^{-(1-\tau_l)}, \tag{6.66}$$

which can be simplified by replacing the product by a summation of logarithms to

$$ln(p[Z(k+1)|\theta(k+1), Z^k]) = \sum_{l=1}^{m(k+1)} ln(f_{t_{i_l}}[z_{j_l}(k+1)]^{\tau_l} V^{-(1-\tau_l)}), \tag{6.67}$$

with components

$$k = \text{last fully processed frame} \tag{6.68}$$

$$p = \text{resulting overall likelihood} \tag{6.69}$$

$$Z = \text{set of measurements} \tag{6.70}$$

$$\theta = \text{current association event} \tag{6.71}$$

$$m = \text{number of assignments} \tag{6.72}$$

$$i_l = \text{track index } i \text{ for assignment } l \tag{6.73}$$

$$j_l = \text{measurement index } j \text{ for assignment } l \tag{6.74}$$

$$f_{t_l}[z_l(k+1)] = \text{likelihood of assignment between } t_l \text{ and } z_l \tag{6.75}$$

$$V^{-1} = \text{constant probability for wrong or missing assignments} \tag{6.76}$$

$$\tau_l = \begin{cases} 0 & \text{if wrong/no assignment} \\ 1 & \text{valid assignment} \end{cases}. \tag{6.77}$$

The auction algorithm is an iterative approach, in which first one forward iteration is performed, followed by reverse iterations until no more unassigned data points

have a price higher than λ. The forward iteration is defined as selecting the first unassigned track i and assigning measurement j_i to it

$$j_i = \underset{j \in A(i)}{\operatorname{argmax}} \{a_{ij} - c_j\} \tag{6.78}$$

$$v_i = \max_{j \in A(i)} \{a_{ij} - c_j\} \tag{6.79}$$

$$w_i = \max_{j \in A(i), j \neq j_i} \{a_{ij} - c_j\} \tag{6.80}$$

$$c_{j_i} \coloneqq \max\{\lambda, a_{ij_i} - w_i + \epsilon\} \tag{6.81}$$

$$\pi_i \coloneqq w_i - \epsilon, \tag{6.82}$$

with definitions

$$j_i = \text{index of best fitting data point } j \text{ to track } i \tag{6.83}$$

$$a_{ij} = \text{benefit of assignment of track } i \text{ with data point } j \tag{6.84}$$

$$p_j = \text{price of data point } j \tag{6.85}$$

$$A(i) = \text{Collecting of possible assignments to track } i \tag{6.86}$$

$$v_i = \text{value of benefit - price for best assignment} \tag{6.87}$$

$$w_i = \text{value of benefit - price for second best assignment} \tag{6.88}$$

$$c_j = \text{current price of data point } j \tag{6.89}$$

$$\lambda = \text{lowest possible price/price threshold} \tag{6.90}$$

$$\epsilon = \text{minimum bid} \tag{6.91}$$

$$\pi_i = \text{current profit of track } i. \tag{6.92}$$

As described, this is followed by as many reverse iterations as required – until no more unassigned data points have a price higher than λ –, selecting the first unassigned data point j with $p_j > \lambda$ and doing

$$i_j = \underset{i \in B(j)}{\operatorname{argmax}} \{a_{ij} - \pi_i\} \tag{6.93}$$

$$\beta_j = \max_{i \in B(j)} \{a_{ij} - \pi_i\} \tag{6.94}$$

$$\gamma_j = \max_{i \in B(j), i \neq i_j} \{a_{ij} - \pi_i\}, \tag{6.95}$$

and afterwards checking if $\beta_j \geq \lambda + \epsilon$. If that is true, assign data point j to track i_j

$$c_j \coloneqq \max\{\lambda, \gamma_j - \epsilon\} \tag{6.96}$$

$$\pi_{i_j} \coloneqq a_{i_j j} - \max\{\lambda, \gamma_j - \epsilon\}, \tag{6.97}$$

if not simply

$$c_j \coloneqq \beta_j - \epsilon, \tag{6.98}$$

with

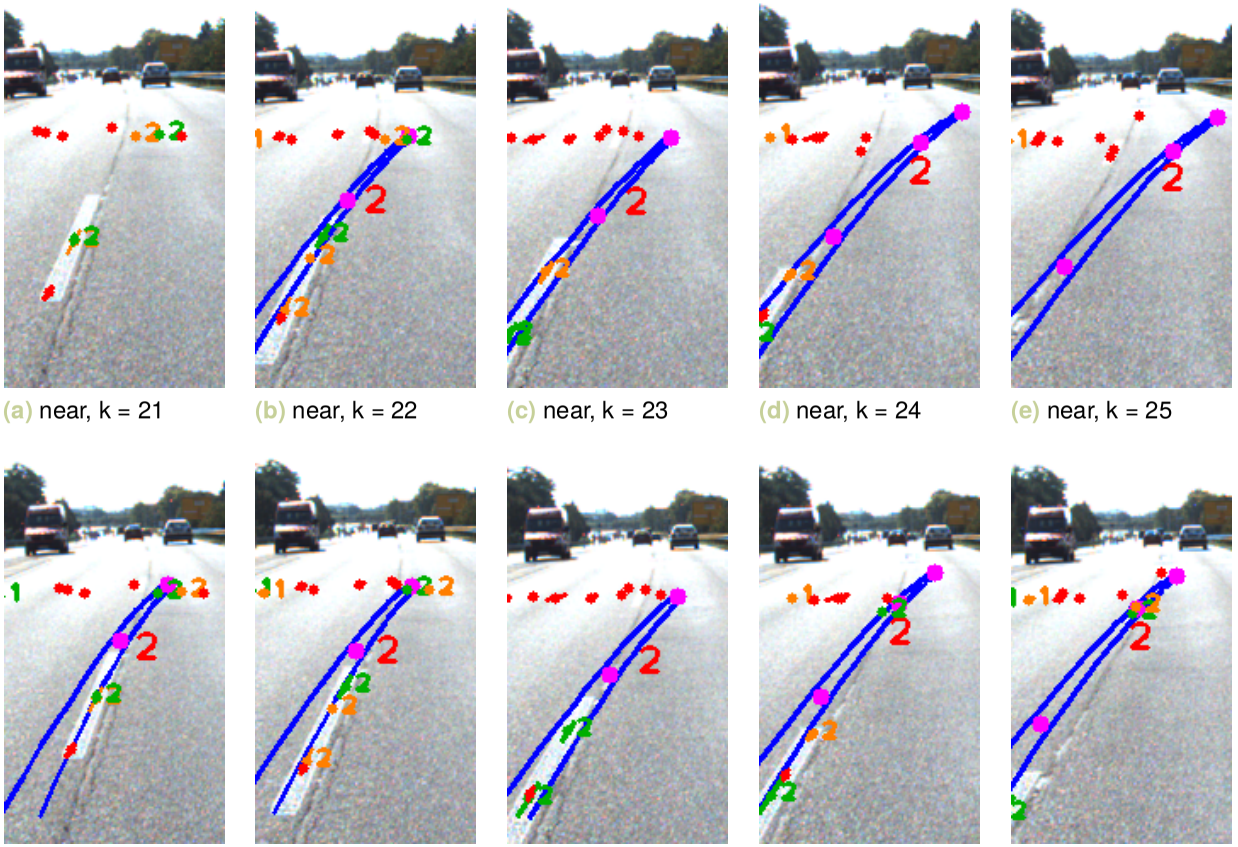
$$i_j = \text{index of best fitting track } i \text{ to data point } j \tag{6.99}$$
$$B(j) = \text{collection of best possible assignments to data point } j \tag{6.100}$$
$$\beta_j = \text{value of benefit - profit for best assignment} \tag{6.101}$$
$$\gamma_j = \text{value of benefit - profit for second best assignment.} \tag{6.102}$$

This is proven by Bertsekas and Castanon, 1992 to result in an optimal assignment, while not requiring dummy tracks to provide possible assignments for all available measurements, even if no real track is available. This assignment is done separately for each ROI, resulting in each track getting a maximum of one update per ROI. Figure 6.8 and figure 6.9 show the advantages of the auction algorithm for assignment, compared to using nearest neighbour assignments. In both of these it can be seen, how the assignments performed with the auction algorithm lead top tracks being closer to their ground truth, if multiple assignments are possible. In figure 6.8, the track is drawn away by noise if using the nearest neighbour assignments, while in figure 6.9, the track on the right side of the hashed area on the road does wander towards the next lane marking on the far right. The auction algorithm does not work perfectly here as well, assigning a wrong lane marking in the near field, but generally holds the lane marking more stable.

Track Initialization

Unassigned lane segments, at most 6.5 m in front of the sensor, are used for initializing new tracks. This way it is ensured that measurements are relatively accurate and reliable. An additional limitation is introduced, that a new track cannot be closer than 1.5 m to another track, as usually lane markings are not that close. As introduced in equation 6.45, each spline segment is defined by a 3rd degree polynomial. $C_1 = 0$ is assumed to reduce complexity, as only 1 measurement is available. $C_0 = \frac{\varphi}{v_f}$ is calculated. For a and b

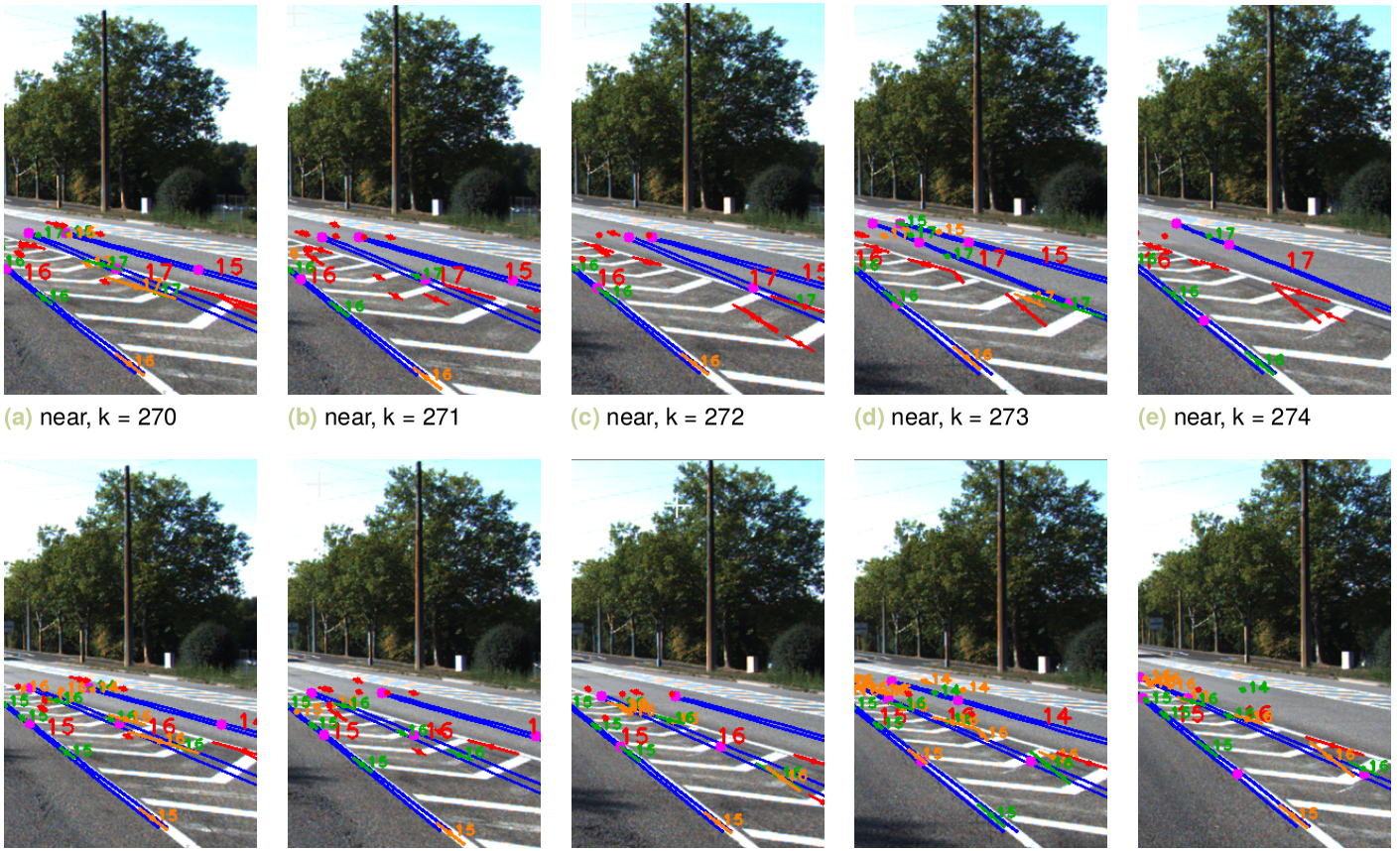
$$a = \arctan(m_y) - C_0 x \tag{6.103}$$
$$b = y - \frac{1}{2} C_0 x^2 - ax \tag{6.104}$$



(a) near, k = 21 (b) near, k = 22 (c) near, k = 23 (d) near, k = 24 (e) near, k = 25

(f) auction, k = 21 (g) auction, k = 22 (h) auction, k = 23 (i) auction, k = 24 (j) auction, k = 25

Figure 6.8.: Comparison of track-measurement assignment between nearest neighbour assignment and the auction algorithm during a lane change. The auction algorithm performs better assignments, keeping the track on the lane marking, while the nearest neighbour assignments result in the track being lost. (Blue = Lane marking, purple = control points (tracked), red number = track id, red = non assigned measurement, orange = close enough for assignment but not assigned, green = assignment). Images from Alsfasser, 2017



(a) near, k = 270 (b) near, k = 271 (c) near, k = 272 (d) near, k = 273 (e) near, k = 274

(f) auction, k = 270 (g) auction, k = 271 (h) auction, k = 272 (i) auction, k = 273 (j) auction, k = 274

Figure 6.9.: Comparison of track-measurement assignment between nearest neighbour assignment and the auction algorithm during the merge of 2 lane markings. Again, the auction algorithm assigns better, resulting in cleaner and more stable tracks. (Blue = Lane marking, purple = control points (tracked), red number = track id, red = non assigned measurement, orange = close enough for assignment but not assigned, green = assignment). Images from Alsfasser, 2017

with

$$x, y = \text{x-/y-position of current measurement} \tag{6.105}$$

$$m_y = \text{slope in y-direction of detection} \tag{6.106}$$

is calculated. As each track spline consists of 5 segments, this is calculated at each of the 5 pre-defined x-positions. Lane width and z-slope are simply initialized with the measurement values.

6.1.4 Evaluation of Lane Marker Results

Only a short excerpt from the evaluation shown in Alsfasser, 2017 is given here. Evaluation and development of the algorithm was performed on a selection of recordings from the Kitti raw data set (Geiger et al., 2013). Lane marker annotations were manually produced and lane marker detection/track evaluation was performed by collecting TP-rate and FP and FN numbers, simply checking for overlap between annotations and predicted lane marker positions. As can be seen, the algorithm usually performs very well, but there are some situations in which results are noticeably worse: Especially rough road surfaces and rails from street cars or trains. Rough road surfaces produce noticeably more noisy reflections and smaller gradients between the road surface and the lane markings. Rails on the other hand are extremely reflective and produce very high gradients. Additionally they are shaped similarly to lane markings and not tall enough to be removed by the variance filter previously mentioned. Both of these problems can be improved by adding additional information as shown in the next section, section 6.2, as image information can help differentiate between road, rail and lane marking. Table 6.1 shows these results on a number of videos from the Kitti raw data set (Geiger et al., 2013). As can be seen, generally the algorithm performs very well on the overall row it can be seen, that over a selection of 12 videos, the true positive rate is at $\frac{1844}{1844+57} = 0.97$. A lane marker spline is detected as true positive if there is an overlap of more than 50% with an annotation. Columns 2-4 of the table show the mean pixel distance of the middle of the spline to the annotation, at 3 different distances from the ego vehicle. This distance is calculated in pixel distance, as the evaluation is performed in camera images, because it was not feasible to annotate 3D point clouds for this work. As can be seen, distances are relatively consistent and it is shown that, as expected, precision stays relatively consistent at the close ranges evaluated here. On visual evaluation it can be seen, that the only major issue of

Video ID	10m↓	20m↓	30m↓	True Positive↑	False Positive↓	False Negative↓
0	6.5	10.98	8.85	14	2	1
1	8.69	5.07	5.87	51	0	4
2	18.07	10.86	7.1	246	7	3
3	16.21	8.19	6.75	106	8	9
4	13.33	7.7	5.78	238	6	3
5	12.46	7.02	5.29	94	7	2
6	13.65	7.08	8.55	23	5	1
7	18.46	8.1	4.22	43	0	5
8	10.65	7.35	6.03	518	14	7
9	11.26	5.59	5.5	70	7	8
10	10.18	7.46	6.27	178	2	7
11	13.23	8.57	6.4	326	43	12
overall	12.69	7.88	6.19	1844	101	57

Table 6.1.: Results of statistical evaluation, distances in pixels

this algorithm is the slow adaption to fast changing lane markings. In the current implementation this is a trade off between staying stable on sudden noise and the ability to quickly adapt to intended changes. An example of a given scene, from Video 2 of this list, is shown in figure 6.10.

6.2 Projection and Reprojection for RGB + Lidar Fusion

An easy method of improving detection results further is adding camera image information to the point cloud. This can either be done at some point during the algorithm, or during data preparation. Here the method of adding RGB information from the image to the data points in the point cloud. The easiest way to do this is to project the data points into the image, get the RGB information for each projected point, by nearest neighbour matching, and assign the RGB values to the data points. The resulting point cloud data point is RGB colored, at least in the areas that can be projected to the camera image. Geiger et al., 2013 show how to perform this projection in their original introduction of the Kitti dataset.

$$\mathbf{y} = P_{\text{rect}}^{(i)} R_{\text{rect}}^{(0)} T_{\text{velo}}^{\text{cam}} T_{\text{imu}}^{\text{velo}} \mathbf{x}, \tag{6.107}$$

which is a projection in homogenous coordinates, meaning $\mathbf{x} = [x\ y\ z\ 1]^T$. $T_{\text{imu}}^{\text{velo}}$ being a translation from the vehicle coordinate system to the lidar sensor coordinate system, $T_{\text{velo}}^{\text{cam}}$ the translation from lidar sensor to the appropriate camera, $R_{\text{rect}}^{(0)}$ rectification via a rotation matrix and $P_{\text{rect}}^{(i)}$ the final projection from the rectified data to the image plane, allowing a match of world position to pixel position. Indices

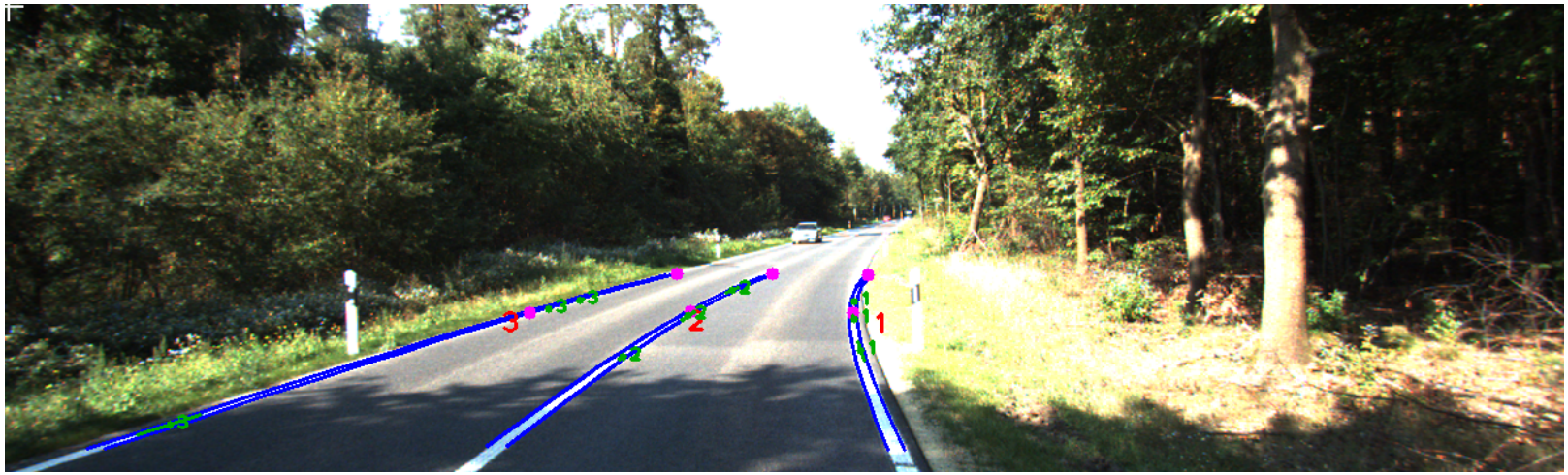


Figure 6.10.: Final result after pre-processing, detection and tracking. Blue = spline outline, purple = control point, red = track ID and green = line segment from detection output. Image from Alsfasser, 2017

(i) or cam describe the selected camera, as the Kitti data presented contains 4 camera point of views. After projection, each Lidar data point, that is visible from the selected camera, can get assigned red, blue and green color information. The pixel position is disregarded as it is not used in later processing. It is important to project lidar to camera and then transfer the information backwards, as 2D camera pixels cannot be projected into a 3D world space, without extra depth information at each position, which is not easily available here. At best a ray of infinite length could be projected, but not exact position on this ray chosen without further information input.

The resulting data, one example shown in figure 6.11, contains full RGB information for lidar data points in the field of view of the selected camera. The main advantage of this should be on differentiating between yellow lane markings and white lane markings. Theoretically it should also help with noise in some situations, where a rough road surface dimishes gradients in the lidar point cloud. Unfortunately no occurences of yellow lane markings are present in the utilized data set and no significant benefits were found on white lanemarkings. Therefore no extensive evaluation was performed here.

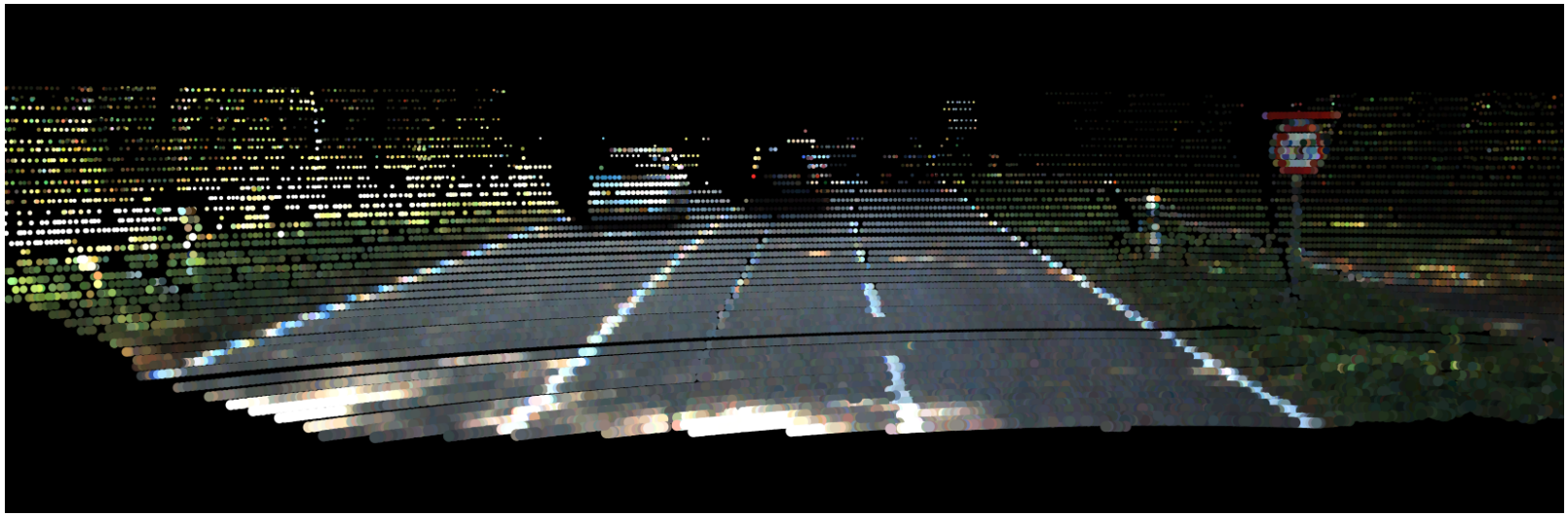


Figure 6.11.: RGB point cloud build, by the described projection method, from point cloud/image from Geiger et al., 2013

7 Conclusion and Outlook

As this is a very varied work, drawing a single conclusion is not easy, therefore this is split into some segments.

7.1 Conclusion and Evolution from Head- and Taillight Tracking

Looking at the results presented in 3.6 it is clear, that a tracker needs to be well adapted to the problem at hand to perform well. In this case this is mainly the trade off between a higher number of false positive tracks to achieve a much higher amount of true positives. False positives are created by the optical tracker, as it, combined with the track health system, can keep dying tracks or false positives alive for several frames before they die. This mostly benefits tracks consisting of spotty detections. They are reliably kept alive through dropped detections and obstructions. Further advancements integrated and evaluated here are group tracking and the use of a dedicated multi target multi measurement tracker, like the Gaussian-mixture probability hypothesis density (GM-PHD) filter. Group tracking allows for higher stability against singular noisy detections, which is beneficial, if wrong predictions might lead to oncoming traffic being blinded for a short while. The GM-PHD makes this all relatively easy to implement, as it does not require hand-crafted track - measurement matching, which can be complicated and unreliable if tracks are close to each other. In the case of the GM-PHD these tracks would actually support each other, as measurements can influence and partially update multiple tracks. Looking at possible further evolutions and improvements on this might include a simple classifier to support mainly the optical tracker at not outputting any noise. The risk is very much reduced because of the forward-backward check but it can still occur if it is a noisy object on a clearly trackable structure.

7.2 Advancing Fast Object Detection in Lidar Point Clouds

Multiple advancements to the training of object detection in lidar point clouds were shown in this work, both towards detection accuracy and runtime. First it was shown, that major computational savings can be made by adjusting the processing structure of the network. It was shown, that reducing the amount of grid cells proportionally reduces the required computations, while staying very close in output performance. This is especially true, if the lost data is replaced by more hand engineered features like occupancy grid maps or extracted height information for all cells. Still, there is a drop in detection performance, very likely caused by the now irregular grid cells. These vary wildly between cells closest to the sensor only being around 5 cm x 5 cm x 5 cm, while the cells far away are up to 3 m x 3 m x 3 m. This is caused by the cells being defined by angular size instead of cartesian. A 2° angle covers a much larger area further away, and the extension of cells in distance direction is scaled by those same 2° and the position of the cell. Resulting in cells that are almost square, despite their angle definition. The intention with almost square cells was keeping the structure close to a standard grid, but experiments showed issues with generalization and overfitting, even though novel and advanced data augmentation was employed. This data augmentation is based on object injection previously published in multiple publications, but improves it at several key areas. Basing injection on a verification against angular overlaps with other objects in the scene improves the realism of the point cloud compared to naive injections only testing against actual bounding box overlap. This realism makes it easier for the network to generalize against real point clouds produced by the lidar in a live system without injections and augmentation. Futhermore this allows to reproduce shadows in the point cloud caused by injected objets. As lidar cannot pass through opaque materials an object like a car blocks the rays from seeing behind it. In naive injection this is not handled. As an outlook, the resulting network could still be improved in some ways, mainly by fixing parts of the issue caused by the large differences in grid cell size. A different feature extraction could help with this, either aggregating features across multiple cells or also using spatial transformer layers to normalize cell content. Combining all of this allows to reduce the computational effort in terms of FLOPs by almost 90% heavily reduces memory usage and also shortens inference times noticeably. Additional optimizations as with utilizing sparse convolutions or replacements for multidimensional convolutions, for example presented in Effnet (Freeman et al., 2018) could be evaluated for

further speed gains, as a large backbone of the network is still comprised of 2d convolutions.

7.3 Applying Complex Neural Networks to Data Annotation

In contrast to the advancements explained in section 7.2, this is more focused on improving detection quality instead of runtime or computational effort. To achieve this, several modifications to well known state of the art networks are performed. The first such modification is the use of GRU layers to combine information and features from multiple different time steps into one network. Evaluation shows a large impact by this, as stability of results is increased and objects are detected much more reliably, across different classes. It is also shown that the effect of simply stacking features from multiple time steps and replacing the GRU by a few fully connected network layers does not achieve the same improvement. The next major change is allowed by the use case for this network. As the network is developed to be used for human supported automatic data annotation, it is possible to focus the attention of the network on smaller parts of the point cloud. This is done by cutting small patches around selected positions, voxelize those patches at high resolution and feed those small patches into the final network. As a result a much larger network can be utilized at the same memory capacity, improving hardware compatibility massively. As a final input feature, cluster information provided by a third party algorithm is utilized, allowing the network to easily detect which data points might belong to the same object. This improves point-wise classification on all tested object classes. Finally the injection augmentation presented in chapter 4 is improved in multiple ways. It is shown, how range images created from point clouds can be used to perform injection overlap checks on a much easier, faster and more precise way, by testing overlap in the range image instead of cartesian or angular point clouds. The result are more precise object shadows for injected objects, as not a full angle range has to be removed. As the range image allows to know exactly which scan areas of the sensor are occluded by the new object, exactly this area can be removed. This allows for even more realism, making it difficult to differentiate real and fake point clouds, translating to a better transfer of training accuracy to accuracy on real data. The final evaluation shows how the combination of all these modifications and improvements allows for very high classification accuracy in an offline scenario. Most tested classes can be detected at much over 80% accuracy, the more common and popular classes cars and pedestrians at over 90%. As for

an outlook, it could be possible to even further extend the time component, by processing a complete sequence of a point cloud recording end to end. This could allow the network to simultaneously perform tracking on objects, which could be utilized to produce suggestions the human annotator just has to confirm to process a point cloud. While this approach is focused on annotations of 3D bounding boxes, there is also considerable interest in semantic/instance segmentation, also requiring a huge amount of training data. Automating this is significantly more difficult, but approaches like MetaBox+(Colling et al., 2021) exist, to provide cost estimations and active learning benefits to reduce the amount of required annotation effort.

7.4 A Classical Approach to Lane Marker Detection in Point Clouds

The final chapter of this work covers the detection and tracking of lane marker segments in lidar point clouds. This can be useful in a multitude of situations, mainly those in which no camera is available or where the camera image is not clear because of weather or other influences. It is presented, that data pre-processing is a very important aspect of such a system. By removing irrelevant points from the point cloud and improving clarity on the relevant data points, even a relatively simple detection method can reliably detect lane marker segments. This is finally supported by a high precision tracking system based on the well known Kalman filter. This filter is used for tracking control points of several spline segments, which are used to describe the full lane marking at a range of up to 40 m. This limit is relatively low, as the resolution of the lidar sensor used to generate these point clouds is not high enough to reliably detect lane markers further away and this already allows for good situational awareness, by covering around 1.5 seconds of travel, even at highway speeds. Furthermore the importance of good measurement-track association is shown. While the tracker used in chapter 3 is able to work without such assignments, it is not a good fit for all problems and a different system, as used here, is more well fitted. Possible extensions to this system are widespread. As previously explained, the detection system is relatively simple by just using thresholding on a gradient filtered image. While this works, more advanced systems might produce better results or the differentiation of white lane markings and yellow lane markings. Additionally the complex system of tracking spline segments and building splines from them, makes it relatively difficult to include new driving scenarios. While marker merging and splitting is supported, the algorithm cannot recognize hashed zones,

only the markings around them, and also cannot easily recognize arrows and other markings on the roads.

Bibliography

Alsfasser, Martin (2017). “Development and Evaluation of a Lidar based Lane Detection System”. MA thesis. Wuppertal, Germany: Department of Electrical, Information and Media Engineering - University of Wuppertal (cit. on pp. 4, 121, 127, 130, 132, 139–141, 143).

Alsfasser, Martin, Mirko Meuter, and Anton Kummert (2019). “Combinatorial use of optical tracker, Gaussian Mixture PHD and group tracking for vehicle light tracking”. In: *Proceedings of the IEEE Intelligent Vehicles Symposium*, pp. 410–416 (cit. on pp. 4, 30, 62).

Alsfasser, Martin, Jan Siegemund, Jittu Kurian, and Anton Kummert (2020). “Exploiting polar grid structure and object shadows for fast object detection in point clouds”. In: *Proceedings of the 12th International Conference on Machine Vision*. Vol. 11433. International Society for Optics and Photonics. SPIE, pp. 111 –118 (cit. on pp. 4, 83).

Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander (1999). “OPTICS: Ordering Points To Identify the Clustering Structure”. In: ACM Press, pp. 49–60 (cit. on pp. 29, 53, 56, 165).

Baker, Simon and Iain Matthews (2004). “Lucas-Kanade 20 years on: A unifying framework”. In: *International journal of computer vision* 56.3, pp. 221–255 (cit. on pp. 9–11, 44, 45).

Bar-Shalom, Yaakov, Fred Daum, and Jim Huang (2009). “The probabilistic data association filter”. In: *IEEE Control Systems Magazine* 29.6, pp. 82–100 (cit. on p. 131).

Baur, Stefan A., Frank Moosmann, Sascha Wirges, and Christoph B. Rist (2019). “Real-time 3D LiDAR flow for autonomous vehicles”. In: *Proceedings of the IEEE Intelligent Vehicles Symposium*, pp. 1288–1295 (cit. on p. 92).

Bayer, Bryce E. (1976). “Color Imaging Array”. U.S. pat. 3971065. Eastman Kodak Co. (cit. on p. 26).

Beneš, Radek, Martin Hasmanda, and Kamil Říha (2011). “Non-linear RANSAC method and its utilization”. In: *elektrorevue* (cit. on pp. 119, 123, 124).

Bergstra, James and Y. Bengio (2012). “Random Search for Hyper-Parameter Optimization”. In: *The Journal of Machine Learning Research* 13, pp. 281–305 (cit. on p. 42).

Bernardin, Keni and Rainer Stiefelhagen (2008). “Evaluating multiple object tracking performance: the CLEAR MOT metrics”. In: *EURASIP Journal on Image and Video Processing*, pp. 1–10 (cit. on pp. 43, 60).

Bertsekas, Dimitri P. (1988). “The auction algorithm: A distributed relaxation method for the assignment problem”. In: *Annals of operations research* 14.1, pp. 105–123 (cit. on pp. 131, 135).

Bertsekas, Dimitri P. and David A. Castanon (1992). “A forward/reverse auction algorithm for asymmetric assignment problems”. In: *Computational Optimization and Applications* 1.3, pp. 277–297 (cit. on pp. 135, 138).

Bottou, Léon (1991). “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8, p. 12 (cit. on p. 21).

Bouguet, Jean-Yves et al. (2001). “Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm”. In: *Intel corporation* 5.1-10, p. 4 (cit. on p. 47).

Chung, Junyoung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio (2014). “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *NIPS Workshop on Deep Learning* (cit. on pp. 23, 24).

Clark, Daniel and Simon Godsill (2007). “Group Target Tracking with the Gaussian Mixture Probability Hypothesis Density Filter”. In: *3rd. International Conference on Intelligent Sensors, Sensor Networks and Information*, pp. 149–154 (cit. on p. 55).

Colling, Pascal, Lutz Roese-Koerner, Hanno Gottschalk, and Matthias Rottmann (2021). “MetaBox+: A New Region based Active Learning Method for Semantic Segmentation using Priority Maps”. In: *Proceedings of the 10th International Conference on Pattern Recognition Applications and Methods*. Vol. 1. INSTICC. SciTePress, pp. 51–62 (cit. on p. 150).

Cortes, Corinna and Vladimir Vapnik (1995). “Support-vector networks”. In: *Machine Learning* 20, pp. 273–297 (cit. on pp. 18, 19).

Cucker, Felipe and Steve Smale (2007). “Emergent behavior in flocks”. In: *IEEE Transactions on automatic control* 52.5, pp. 852–862 (cit. on p. 57).

Dalal, Navneet and Bill Triggs (2005). “Histograms of oriented gradients for human detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 1, pp. 886–893 (cit. on p. 16).

Duda, Richard O. and Peter E. Hart (1972). “Use of the Hough Transformation to Detect Lines and Curves in Pictures”. In: *Communications of the ACM* 15.1, 11–15 (cit. on pp. 16, 127).

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu (1996). “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: AAAI Press, pp. 226–231 (cit. on pp. 52–54, 165).

Fisher, Robert B., Simon Perkins, Ashley Walker, and Erik Wolfart (1996). “Hypermedia Image Processing Reference (HIPR)”. In: *Artificial Intelligence - AI*, pp. 75–76 (cit. on p. 119).

Freeman, Ido, Lutz Roese-Koerner, and Anton Kummert (2018). “Effnet: An Efficient Structure for Convolutional Neural Networks”. In: *Proceedings of the 25th IEEE International Conference on Image Processing*, pp. 6–10 (cit. on p. 148).

Geiger, Andreas, Philip Lenz, Christoph Stiller, and Raquel Urtasun (2013). “Vision meets Robotics: The KITTI Dataset”. In: *International Journal of Robotics Research* 32, pp. 1231–1237 (cit. on pp. 15, 18, 28, 141, 142, 145).

Geiger, Andreas, Philip Lenz, and Raquel Urtasun (2012). “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”. In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361 (cit. on pp. 82, 84–86).

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *13th International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256 (cit. on p. 99).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press (cit. on pp. 21–23).

Gouveia, Luiz Carlos Paiva and Bhaskar Choubey (2016). “Advances on CMOS image sensors”. In: *Sensor review* (cit. on p. 26).

Graham, Benjamin and Laurens van der Maaten (2017). “Submanifold sparse convolutional networks”. In: *arXiv preprint arXiv:1706.01307* (cit. on p. 94).

Greff, Klaus, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber (2016). “LSTM: A search space odyssey”. In: *IEEE transactions on neural networks and learning systems* 28.10, pp. 2222–2232 (cit. on p. 24).

Hahn, Lukas, Frederik Hasecke, and Anton Kummert (2020). “Fast Object Classification and Meaningful Data Representation of Segmented Lidar Instances”. In: *Proceedings of the IEEE 23rd International Conference on Intelligent Transportation Systems*, pp. 1–6 (cit. on pp. 64, 101).

Hasecke, Frederik, Martin Alsfasser, and Anton Kummert (2022). *What Can be Seen is What You Get: Structure Aware Point Cloud Augmentation*. Preprint, not yet published (cit. on pp. 4, 101).

Hasecke, Frederik, Lukas Hahn, and Anton Kummert (2021). “FLIC: Fast Lidar Image Clustering”. In: *Proceedings of the 10th International Conference on Pattern Recognition Applications and Methods*. Vol. 1. INSTICC. SciTePress, pp. 25–35 (cit. on pp. 101, 104, 109).

Henry, Eric and James Hofrichter (1992). “[8] Singular value decomposition: Application to analysis of experimental data”. In: *Methods in Enzymology* 210, pp. 129–192 (cit. on p. 124).

Hough, Paul V.C. (1962). “Method and Means for Recognizing Complex Patterns”. U.S. pat. 3069654. Hough, Paul V.C. (cit. on p. 15).

Janocha, Katarzyna and Wojciech Czarnecki (2017). “On Loss Functions for Deep Neural Networks in Classification”. In: *Schedae Informaticae* 25 (cit. on p. 20).

Kaipio, Jari and Erkki Somersalo (2006). *Statistical and Computational Inverse Problems*. Applied Mathematical Sciences. Springer New York, pp. 16–26 (cit. on p. 45).

Kalman, Rudolph Emil (1960). “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D, pp. 35–45 (cit. on pp. 5, 36).

Kim, Youngjoo and Hyochoong Bang (Nov. 2018). “Introduction to Kalman Filter and Its Applications”. In: (cit. on p. 6).

Kingma, Diederik and Jimmy Ba (2014). “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (cit. on p. 81).

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc. (cit. on p. 123).

Krenker, Andrej, Janez Bešter, and Andrej Kos (2011). “Introduction to the artificial neural networks”. In: *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech*, pp. 1–18 (cit. on p. 20).

Kuhn, Harold W. (1955). “The Hungarian method for the assignment problem”. In: *Naval research logistics quarterly* 2.1-2, pp. 83–97 (cit. on p. 135).

Lang, Alex H., Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom (2019). “PointPillars: Fast Encoders for Object Detection From Point Clouds”. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pp. 12689–12697 (cit. on pp. 63, 64, 67, 79, 81, 82, 85, 118).

Leal-Taixé, Laura, Anton Milan, Ian Reid, Stefan Roth, and Konrad Schindler (2015). “MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking”. In: *arXiv preprint arXiv:1504.01942[cs]* (cit. on p. 60).

Lehman, Brad and Arnold J. Wilkins (2014). “Designing to Mitigate Effects of Flicker in LED Lighting: Reducing risks to health and safety”. In: *IEEE Power Electronics Magazine* 1.3, pp. 18–26 (cit. on p. 30).

Lehner, Johannes, Andreas Mitterecker, Thomas Adler, Markus Hofmarcher, Bernhard Nessler, and Sepp Hochreiter (2019). “Patch Refinement–Localized 3D Object Detection”. In: (cit. on p. 100).

Lucas, Bruce D. and Takeo Kanade (1981). “An Iterative Image Registration Technique with an Application to Stereo Vision”. In: *7th International Joint Conference on Artificial Intelligence*. Vol. 2. Morgan Kaufmann Publishers Inc., 674–679 (cit. on pp. 9, 45).

Luo, Wenjie, Bin Yang, and Raquel Urtasun (2018). “Fast and Furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 3569–3577 (cit. on p. 3).

MacQueen, James et al. (1967). “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA, pp. 281–297 (cit. on p. 52).

Mahalanobis, Prasanta Chandra (1936). “On the generalised distance in statistics”. In: *Proceedings of the National Institute of Sciences of India*, pp. 49–55 (cit. on pp. 40, 49, 59).

Mao, Xiao-Jiao, Chunhua Shen, and Yu-Bin Yang (2016). “Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections”. In: *arXiv preprint arXiv:1603.09056* (cit. on p. 22).

Milan, Anton, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler (2016). “MOT16: A Benchmark for Multi-Object Tracking”. In: *arXiv preprint arXiv:1603.00831[cs]* (cit. on p. 60).

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (1992). *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press (cit. on p. 49).

Qi, Charles R., Wei Liu, Chenxia Wu, Hao Su, and Leonidas J. Guibas (2018). "Frustum pointnets for 3d object detection from rgb-d data". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 918–927 (cit. on pp. 65, 100).

Qi, Charles R., Hao Su, Kaichun Mo, and Leonidas J. Guibas (2017a). "Pointnet: Deep learning on point sets for 3d classification and segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 652–660 (cit. on pp. 65, 90, 91).

Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas (2017b). "Pointnet++: Deep hierarchical feature learning on point sets in a metric space". In: *Advances in neural information processing systems* 30, pp. 5099–5108 (cit. on p. 65).

Shi, Shaoshuai, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li (2020). "PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 10526–10535 (cit. on pp. 84, 90–92, 96, 97).

Shi, Shaoshuai, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li (2021). "From Points to Parts: 3D Object Detection From Point Cloud With Part-Aware and Part-Aggregation Network". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.8, pp. 2647–2664 (cit. on pp. 90, 91, 94, 97, 98).

Shin, Kiwoo, Youngwook Paul Kwon, and Masayoshi Tomizuka (2019). "Roarnet: A robust 3d object detection based on region approximation refinement". In: *Proceedings of the IEEE Intelligent Vehicles Symposium*, pp. 2510–2515 (cit. on pp. 66, 100).

Sobel, Irwin and Gary Feldman (Jan. 1973). "A 3×3 isotropic gradient operator for image processing". In: *Pattern Classification and Scene Analysis*, pp. 271–272 (cit. on p. 15).

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press (cit. on pp. 7, 8).

Vo, Ba-Ngu and Wing-Kin Ma (2006). "The Gaussian Mixture Probability Hypothesis Density Filter". In: *IEEE Transactions on Signal Processing* 54.11, pp. 4091–4104 (cit. on pp. 2, 11–14, 33, 34, 40).

Yan, Yan, Yuxing Mao, and Bo Li (2018). "Second: Sparsely embedded convolutional detection". In: *Sensors* 18.10, p. 3337 (cit. on pp. 63, 64, 67, 76, 77, 81, 85).

Yu, Fisher, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell (2018). "BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling". In: *arXiv preprint arXiv:1805.04687[cs]* (cit. on p. 60).

Zhang, Ji and Sanjiv Singh (2014). "LOAM: Lidar Odometry and Mapping in Real-time." In: *Robotics: Science and Systems*. Vol. 2. 9 (cit. on p. 91).

Zhang, Zhengyou (2000). "A flexible new technique for camera calibration". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.11, pp. 1330–1334 (cit. on p. 27).

Zhao, Kun, Mirko Meuter, Christian Nunn, Dennis Müller, Stefan Müller-Schneiders, and Josef Pauli (2012). “A novel multi-lane detection and tracking system”. In: *Proceedings of the IEEE Intelligent Vehicles Symposium*, pp. 1084–1089 (cit. on pp. 129, 134).

Zhou, Yin and Oncel Tuzel (2018). “VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4490–4499 (cit. on pp. 63, 64, 67, 79–82, 85).

Webpages

Blickfeld GmbH (2021). *What is Solid-State LiDAR?* URL: {https://www.blickfeld.com/what-is-solid-state-lidar/} (visited on Jan. 19, 2021) (cit. on p. 27).

Hesai Technology (2021). *Pandora*. URL: {https://www.hesaitech.com/en/Pandora} (visited on Jan. 19, 2021) (cit. on p. 27).

The MathWorks Inc. (2021a). *extracthogfeatures*. URL: {https://de.mathworks.com/help/vision/ref/extracthogfeatures.html} (visited on Feb. 17, 2021) (cit. on p. 18).

– (2021b). *hough*. URL: {https://de.mathworks.com/help/images/ref/hough.html} (visited on Feb. 17, 2021) (cit. on p. 17).

– (2021c). *houghlines*. URL: {https://de.mathworks.com/help/images/ref/houghlines.html} (visited on Feb. 17, 2021) (cit. on pp. 16, 17).

Velodyne Lidar Inc. (2021a). *HDL-64E*. URL: {https://velodynelidar.com/products/hdl-64e/} (visited on Jan. 19, 2021) (cit. on pp. 14, 27, 67, 101).

– (2021b). *What is Lidar?* URL: {https://velodynelidar.com/what-is-lidar/} (visited on Jan. 19, 2021) (cit. on p. 27).

Weisstein, Eric W (2021a). *Jacobian - From MathWorld–A Wolfram Web Resource*. URL: {https://mathworld.wolfram.com/Jacobian.html} (visited on Feb. 8, 2021) (cit. on p. 135).

– (2021b). *Vandermonde Matrix - From MathWorld–A Wolfram Web Resource*. URL: {https://mathworld.wolfram.com/VandermondeMatrix.html} (visited on June 29, 2021) (cit. on p. 123).

List of Figures

List of Tables

List of Listings

Acronyms

ADAS Advanced Driver Assistance Systems. 1, 25, 29, 30, 32, 159

AHC Advanced Headlight Control. 1, 29, 32

BCE binary cross entropy. 81

CCD charge-coupled device. 26

CMOS complementary metal oxide semiconductor. 25, 26, 43

CNN convolutional neural network. 22

CPU central processing unit. 86–88

DBSCAN Density Based Spatial Clustering of Applications with Noise. 53, 55

EKF extended Kalman filter. 7, 131

FN false negative. 43, 141

FP false positive. 43, 61, 141

GB gigabyte. 84, 85, 92

GM-PHD Gaussian-mixture probability hypothesis density. iii, iv, 2, 4, 14, 29, 30, 33, 34, 36, 40–42, 50, 60, 61, 147, 165

GPU graphics processing unit. 66, 86–88, 90, 99, 100

GRU gated recurrent unit. 23–25, 91, 94, 98, 99, 107–109, 149, 159

GT ground truth. 43

HOG Histogram of Oriented Gradients. 16, 18

JPDA joint probabilistic data association. 131

LED Light Emitting Diode. 1, 2, 30, 31, 43

LSTM long short term memory. 24

MMT mismatched tracks. 43

MOTA multi object tracking accuracy. 42, 43, 60

MOTP multi object tracking precision. 42, 43, 60

MSE mean square error. 20, 39

OPTICS Ordering Points To Identify the Clustering Structure. 53, 55, 57–59, 160

PHD probability hypothesis density. 2, 11, 13, 33, 34, 40, 44, 50, 51, 55, 58, 59

PWM pulse width modulation. 2, 30, 31, 43, 159

RFS random finite set. 12, 13, 34

RGB red, green and blue. 24, 117, 142, 144, 145, 162

RNN recurrent neural network. 23, 24, 99

ROI region of interest. 42, 117–119, 128, 129, 138

SAPCA structure aware point cloud augmentation. 4, 101, 103, 104, 106, 108, 110–116, 161

SGD stochastic gradient descent. 21

SLAM simultaneous localization and mapping. 91

SVD singular value decomposition. 124, 125

SVM Support Vector Machine. 16–18

TP true positive. 42, 43, 60, 61, 141

UKF unscented Kalman filter. 7, 131

VFE voxel feature encoding. 64, 71, 72, 79, 92

Appendix

A

A.1 Full Training Progression for 5.5

(a) Training Progression - Overall training accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,645	0,13	0,508	0,791
2	0,809	0,117	0,687	0,884
3	0,829	0,116	0,738	0,866
4	0,846	0,47	0,759	0,886
5	0,858	0,723	0,777	0,895
6	0,866	0,757	0,786	0,902
7	0,871	0,78	0,797	0,906
8	0,877	0,801	0,804	0,91
9	0,897	0,813	0,814	0,912
10	0,884	0,822	0,821	0,916
11	0,887	0,828	0,827	0,919
12	0,89	0,834	0,831	0,923
13	0,893	0,84	0,834	0,924
14	0,896	0,845	0,838	0,927
15	0,897	0,849	0,84	0,929
16	0,9	0,856	0,846	0,932
17	0,902	0,857	0,849	0,932
18	0,904	0,857	0,85	0,936
19	0,906	0,861	0,853	0,937
20	0,909	0,863	0,855	0,937
21	0,907	0,866	0,857	0,939
22	0,913	0,868	0,857	0,94
23	0,91	0,869	0,855	0,942
24	0,913	0,872	0,858	0,943
25	0,915	0,877	0,862	0,944
26	0,916	0,876	0,862	0,946
27	0,917	0,879	0,865	0,947
28	0,917	0,879	0,863	0,948
29	0,919	0,879	0,866	0,947
30	0,921	0,882	0,868	0,949
31	0,921	0,881	0,867	0,95
32	0,922	0,883	0,867	0,951
33	0,923	0,885	0,869	0,953
34	0,921	0,888	0,871	0,952
35	0,925	0,89	0,873	0,952
36	0,925	0,89	0,872	0,954
37	0,927	0,89	0,869	0,954
38	0,926	0,892	0,872	0,954
39	0,927	0,892	0,873	0,956
40	0,928	0,893	0,875	0,956
41	0,929	0,892	0,875	0,956
42	0,928	0,893	0,877	0,957
43	0,927	0,896	0,875	0,957
44	0,929	0,894	0,877	0,958
45	0,929	0,897	0,877	0,958
46	0,931	0,899	0,876	0,959
47	0,931	0,898	0,879	0,959
48	0,93	0,901	0,877	0,959
49	0,931	0,901	0,881	0,959
50	0,931	0,901	0,877	0,96

(b) Training Progression - Overall validation accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,718	0,106	0,566	0,891
2	0,778	0,08	0,671	0,891
3	0,803	0,069	0,716	0,88
4	0,819	0,176	0,743	0,883
5	0,827	0,296	0,758	0,883
6	0,834	0,384	0,77	0,885
7	0,841	0,449	0,779	0,887
8	0,846	0,498	0,787	0,89
9	0,851	0,535	0,794	0,891
10	0,855	0,566	0,801	0,894
11	0,859	0,591	0,806	0,895
12	0,862	0,615	0,811	0,897
13	0,865	0,634	0,815	0,899
14	0,868	0,651	0,819	0,9
15	0,87	0,667	0,822	0,901
16	0,872	0,681	0,827	0,903
17	0,874	0,693	0,829	0,904
18	0,876	0,704	0,831	0,905
19	0,878	0,713	0,833	0,906
20	0,88	0,723	0,836	0,907
21	0,881	0,731	0,838	0,908
22	0,883	0,738	0,84	0,909
23	0,884	0,745	0,842	0,91
24	0,885	0,751	0,844	0,911
25	0,887	0,757	0,846	0,912
26	0,913	0,9	0,887	0,913
27	0,915	0,9	0,886	0,913
28	0,915	0,899	0,888	0,914
29	0,917	0,901	0,888	0,915
30	0,917	0,901	0,888	0,915
31	0,918	0,903	0,888	0,916
32	0,918	0,904	0,89	0,917
33	0,919	0,905	0,891	0,917
34	0,919	0,905	0,892	0,918
35	0,92	0,905	0,893	0,919
36	0,92	0,906	0,893	0,919
37	0,92	0,906	0,893	0,92
38	0,92	0,907	0,893	0,92
39	0,921	0,908	0,893	0,921
40	0,921	0,908	0,894	0,921
41	0,921	0,908	0,894	0,922
42	0,921	0,909	0,894	0,922
43	0,921	0,909	0,895	0,923
44	0,922	0,909	0,895	0,923
45	0,922	0,91	0,895	0,924
46	0,922	0,911	0,896	0,924
47	0,922	0,911	0,896	0,924
48	0,922	0,911	0,896	0,925
49	0,923	0,911	0,896	0,925
50	0,923	0,912	0,896	0,926

Table A.1.: Training/Validation progression

(a) Training Progression - Background accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,728	0,063	0,579	0,957
2	0,79	0,031	0,683	0,934
3	0,814	0,021	0,727	0,913
4	0,83	0,137	0,754	0,911
5	0,837	0,267	0,768	0,907
6	0,844	0,362	0,779	0,907
7	0,851	0,431	0,787	0,907
8	0,855	0,483	0,795	0,908
9	0,86	0,522	0,802	0,909
10	0,864	0,554	0,809	0,91
11	0,867	0,581	0,814	0,912
12	0,87	0,606	0,818	0,913
13	0,874	0,627	0,822	0,914
14	0,876	0,645	0,826	0,915
15	0,878	0,661	0,829	0,916
16	0,88	0,675	0,832	0,917
17	0,882	0,688	0,835	0,918
18	0,884	0,7	0,837	0,919
19	0,886	0,71	0,84	0,92
20	0,888	0,72	0,842	0,921
21	0,889	0,728	0,845	0,922
22	0,891	0,736	0,846	0,922
23	0,892	0,743	0,848	0,923
24	0,893	0,75	0,85	0,924
25	0,894	0,756	0,852	0,925
26	0,919	0,907	0,893	0,925
27	0,922	0,907	0,891	0,926
28	0,922	0,905	0,893	0,927
29	0,924	0,907	0,894	0,927
30	0,924	0,908	0,894	0,928
31	0,925	0,909	0,894	0,928
32	0,926	0,91	0,896	0,929
33	0,926	0,912	0,897	0,93
34	0,927	0,911	0,897	0,93
35	0,927	0,912	0,898	0,931
36	0,928	0,912	0,899	0,931
37	0,928	0,913	0,898	0,932
38	0,928	0,913	0,899	0,932
39	0,928	0,914	0,899	0,933
40	0,929	0,915	0,9	0,933
41	0,929	0,915	0,9	0,934
42	0,929	0,915	0,9	0,934
43	0,929	0,916	0,901	0,935
44	0,929	0,916	0,901	0,935
45	0,93	0,917	0,901	0,936
46	0,93	0,917	0,902	0,936
47	0,93	0,917	0,902	0,936
48	0,93	0,918	0,902	0,937
49	0,931	0,918	0,902	0,937
50	0,931	0,918	0,902	0,938

(b) Training Progression - Car accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,736	0,938	0,488	0,07
2	0,773	0,969	0,581	0,464
3	0,797	0,979	0,638	0,603
4	0,806	0,955	0,677	0,67
5	0,819	0,928	0,705	0,711
6	0,827	0,911	0,719	0,737
7	0,835	0,9	0,738	0,757
8	0,839	0,892	0,752	0,772
9	0,844	0,888	0,763	0,784
10	0,849	0,884	0,773	0,795
11	0,852	0,883	0,783	0,804
12	0,855	0,882	0,79	0,811
13	0,858	0,883	0,795	0,817
14	0,86	0,883	0,802	0,822
15	0,861	0,883	0,807	0,826
16	0,862	0,883	0,81	0,83
17	0,864	0,883	0,814	0,834
18	0,867	0,884	0,819	0,837
19	0,869	0,885	0,823	0,84
20	0,871	0,886	0,826	0,841
21	0,872	0,886	0,829	0,844
22	0,873	0,887	0,833	0,845
23	0,875	0,887	0,835	0,848
24	0,876	0,887	0,838	0,85
25	0,877	0,888	0,84	0,852
26	0,9	0,89	0,897	0,854
27	0,91	0,879	0,901	0,855
28	0,911	0,896	0,901	0,856
29	0,906	0,895	0,903	0,858
30	0,903	0,898	0,9	0,859
31	0,905	0,899	0,901	0,86
32	0,904	0,9	0,903	0,861
33	0,902	0,899	0,903	0,862
34	0,904	0,9	0,905	0,863
35	0,905	0,899	0,904	0,864
36	0,904	0,9	0,903	0,865
37	0,903	0,901	0,902	0,866
38	0,904	0,9	0,903	0,867
39	0,904	0,901	0,903	0,868
40	0,903	0,901	0,902	0,868
41	0,904	0,902	0,902	0,868
42	0,904	0,902	0,902	0,869
43	0,904	0,903	0,902	0,869
44	0,904	0,904	0,902	0,87
45	0,905	0,904	0,902	0,87
46	0,904	0,904	0,903	0,871
47	0,903	0,904	0,903	0,872
48	0,904	0,905	0,903	0,872
49	0,905	0,904	0,902	0,873
50	0,904	0,905	0,902	0,872

Table A.2.: Background/Car progression

(a) Training Progression - Truck accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,077	0	0,486	0
2	0,146	0	0,496	0
3	0,233	0	0,493	0,03
4	0,277	0,08	0,506	0,081
5	0,316	0,027	0,501	0,129
6	0,33	0,06	0,505	0,172
7	0,35	0,115	0,504	0,207
8	0,367	0,151	0,502	0,23
9	0,379	0,186	0,502	0,25
10	0,394	0,216	0,497	0,267
11	0,405	0,239	0,497	0,286
12	0,416	0,257	0,492	0,292
13	0,421	0,271	0,491	0,305
14	0,422	0,285	0,486	0,315
15	0,43	0,296	0,485	0,316
16	0,431	0,306	0,484	0,321
17	0,435	0,317	0,479	0,329
18	0,437	0,324	0,481	0,333
19	0,437	0,33	0,479	0,334
20	0,436	0,338	0,478	0,335
21	0,439	0,345	0,476	0,337
22	0,441	0,35	0,472	0,337
23	0,443	0,353	0,472	0,339
24	0,448	0,357	0,47	0,342
25	0,448	0,361	0,469	0,341
26	0,517	0,464	0,417	0,343
27	0,448	0,437	0,444	0,343
28	0,446	0,435	0,405	0,345
29	0,468	0,44	0,406	0,346
30	0,448	0,431	0,416	0,346
31	0,448	0,435	0,41	0,347
32	0,446	0,432	0,419	0,348
33	0,452	0,426	0,41	0,348
34	0,445	0,432	0,417	0,348
35	0,445	0,432	0,41	0,35
36	0,456	0,432	0,408	0,349
37	0,453	0,43	0,411	0,35
38	0,449	0,433	0,409	0,35
39	0,45	0,431	0,409	0,35
40	0,453	0,426	0,407	0,349
41	0,448	0,428	0,408	0,349
42	0,446	0,425	0,405	0,35
43	0,452	0,425	0,404	0,349
44	0,454	0,424	0,406	0,349
45	0,453	0,425	0,403	0,349
46	0,454	0,425	0,401	0,348
47	0,454	0,425	0,403	0,347
48	0,45	0,426	0,405	0,346
49	0,448	0,423	0,404	0,347
50	0,45	0,422	0,404	0,346

(b) Training Progression - Pedestrian accuracy

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,111	0	0,057	0
2	0,294	0	0,249	0,03
3	0,392	0	0,33	0,182
4	0,463	0,001	0,387	0,297
5	0,511	0,096	0,428	0,375
6	0,541	0,176	0,465	0,431
7	0,569	0,239	0,494	0,474
8	0,592	0,295	0,523	0,51
9	0,615	0,338	0,544	0,539
10	0,631	0,374	0,56	0,559
11	0,644	0,411	0,576	0,577
12	0,658	0,437	0,589	0,595
13	0,667	0,46	0,6	0,605
14	0,675	0,482	0,611	0,618
15	0,681	0,5	0,622	0,629
16	0,691	0,519	0,633	0,639
17	0,698	0,535	0,64	0,648
18	0,703	0,547	0,647	0,656
19	0,708	0,56	0,654	0,663
20	0,713	0,57	0,659	0,667
21	0,717	0,58	0,663	0,673
22	0,719	0,588	0,67	0,678
23	0,724	0,597	0,676	0,684
24	0,726	0,606	0,679	0,69
25	0,73	0,612	0,682	0,694
26	0,819	0,765	0,774	0,698
27	0,802	0,786	0,785	0,701
28	0,807	0,787	0,791	0,705
29	0,801	0,791	0,802	0,708
30	0,802	0,794	0,798	0,711
31	0,8	0,8	0,799	0,713
32	0,799	0,801	0,8	0,715
33	0,801	0,799	0,8	0,719
34	0,803	0,804	0,797	0,719
35	0,803	0,807	0,798	0,722
36	0,804	0,809	0,8	0,724
37	0,803	0,807	0,802	0,725
38	0,805	0,81	0,799	0,728
39	0,806	0,81	0,8	0,729
40	0,807	0,809	0,797	0,731
41	0,808	0,81	0,798	0,732
42	0,808	0,812	0,797	0,734
43	0,807	0,813	0,796	0,735
44	0,809	0,814	0,796	0,736
45	0,809	0,815	0,795	0,737
46	0,809	0,816	0,797	0,738
47	0,81	0,815	0,798	0,739
48	0,81	0,816	0,799	0,74
49	0,81	0,816	0,799	0,741
50	0,809	0,816	0,801	0,743

Table A.3.: Truck/Pedestrian progression

EPOCH	SAPCA	WEDGE	NAIVE	No Injection
1	0,001	0,031	0,009	0
2	0,075	0,016	0,131	0
3	0,114	0,01	0,193	0,01
4	0,161	0,016	0,241	0,033
5	0,202	0,016	0,27	0,054
6	0,212	0,049	0,286	0,084
7	0,231	0,065	0,308	0,109
8	0,254	0,083	0,325	0,127
9	0,272	0,114	0,336	0,143
10	0,285	0,137	0,349	0,156
11	0,299	0,156	0,361	0,164
12	0,315	0,179	0,365	0,181
13	0,32	0,197	0,377	0,195
14	0,329	0,209	0,386	0,204
15	0,345	0,219	0,392	0,211
16	0,35	0,229	0,398	0,218
17	0,353	0,24	0,404	0,227
18	0,359	0,251	0,411	0,23
19	0,368	0,262	0,413	0,237
20	0,371	0,271	0,419	0,235
21	0,376	0,281	0,425	0,24
22	0,382	0,287	0,43	0,243
23	0,384	0,293	0,433	0,245
24	0,388	0,3	0,435	0,247
25	0,393	0,305	0,439	0,25
26	0,495	0,448	0,558	0,251
27	0,487	0,444	0,513	0,253
28	0,485	0,456	0,504	0,254
29	0,495	0,456	0,507	0,256
30	0,497	0,457	0,518	0,258
31	0,491	0,456	0,526	0,259
32	0,495	0,453	0,52	0,26
33	0,491	0,457	0,523	0,262
34	0,497	0,457	0,519	0,263
35	0,488	0,457	0,515	0,264
36	0,487	0,458	0,514	0,266
37	0,486	0,462	0,516	0,267
38	0,483	0,466	0,522	0,266
39	0,482	0,466	0,52	0,266
40	0,482	0,47	0,52	0,268
41	0,484	0,469	0,519	0,268
42	0,484	0,473	0,52	0,269
43	0,48	0,477	0,522	0,271
44	0,48	0,477	0,521	0,272
45	0,478	0,478	0,525	0,272
46	0,477	0,481	0,523	0,272
47	0,479	0,482	0,524	0,273
48	0,48	0,48	0,526	0,274
49	0,479	0,48	0,526	0,274
50	0,479	0,483	0,528	0,275

Table A.4.: Training Progression - Bike accuracy