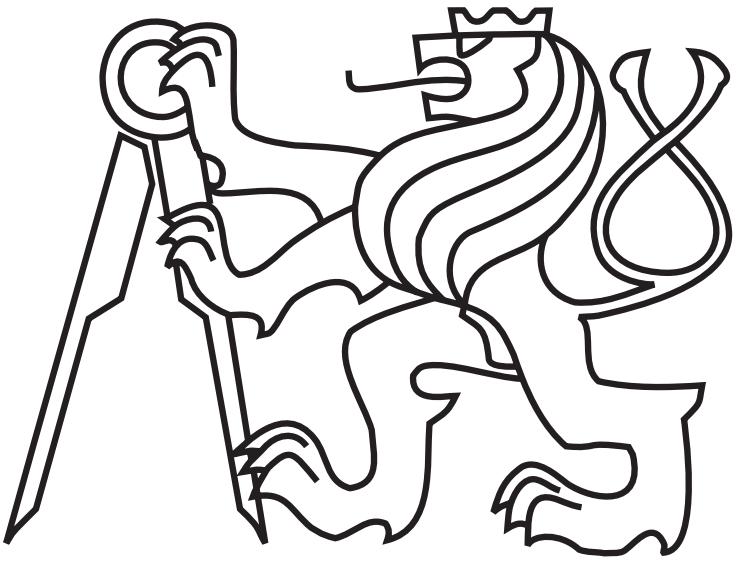
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



Jiří Janota

Car Detection Methods from 2D LIDAR Data Collected with a Mobile Robot

Department of Cybernetics

Thesis supervisor: **Ing. Filip Majer**

May, 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Janota Jiří** Personal ID number: **483485**

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

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Car Detection Methods from 2D LIDAR Data Collected with a Mobile Robot

Bachelor's thesis title in Czech:

Metody detekce automobilu z 2D LIDAR dat nasbíraných mobilním robotem

Guidelines:

1) Research methods used for car detection in 2D LIDAR data.
2) Select a set of methods with the potential to work in real-world scenarios and adverse weather.
3) Select a set of criteria to evaluate the performance of the selected methods.
4) Design and implement a tool capable to automatically evaluate the performance of the individual methods.
5) Perform the experimental evaluation and discuss the results.

Bibliography / sources:

[1] G. Chen et al.: Pseudo-Image and Sparse Points: Vehicle Detection With 2D LiDAR Revisited by Deep Learning-Based Methods, IEEE Transactions on Intelligent Transportation Systems, 2020
[2] L. Beyer, A. Hermans and B. Leibe: DROW: Real-Time Deep Learning-Based Wheelchair Detection in 2-D Range Data, IEEE Robotics and Automation Letters, 2017
[3] R. Q. Charles, H. Su, M. Kaichun and L. J. Guibas: PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017
[4] F. Majer, Z. Yan, G. Broughton, Y. Ruichek and T. Krajník: Learning to see through haze: Radar-based Human Detection for Adverse Weather Conditions, European Conference on Mobile Robots (ECMR), 2019

Name and workplace of bachelor's thesis supervisor:

Ing. Filip Majer, Department of Computer Science, FEE

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

Date of bachelor's thesis assignment: **10.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

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

III. Assignment receipt

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

Date of assignment receipt	Student's signature

Declaration of authorship

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

In Prague on May 21, 2021 Jiří Janota

Acknowledgements

I would like to thank my supervisor Filip Majer for his guidance and support at every stage of writing the bachelor thesis. Also, I express my sincere gratitude to people from Chronorobotics Laboratory on FEE CTU in Prague who provided me with the collected data. Especially, I would like to acknowledge Tomáš Krajník for the great opportunity of working on such a project and George Broughton for insightful comments and advice.

I want to thank my parents Zuzana and Jiří, and my two sisters Ludmila and Eliška, for their patience, love, and endless support, which allowed me to study. I would like to thank our three dogs, whose jiggling tails always brighten my day.

I would like to extend my thanks to all my friends who always cheer me up. Particularly, I appreciate the help not only regarding science from my dear friend Jan Blaha.

Abstrakt

Tato práce se zabývá různými přístupy k detekci automobilů parkovacím robotem v reálném světě. Za tímto účelem byly implementovány tři metody strojového učení založené na segmentaci množin bodů (PointNet), segmentaci obrázků (U-Net) a klasifikaci vektorů atributů (SVM) a jedna geometrická metoda pro lokalizaci kol. Porovnání metod je uskutečněno na anotovaném datasetu 2-dimenzionálních měření ze tří LIDAR senzorů instalovaných na mobilním parkovacím robotu. Použití souboru dat nasbíraného při provozu v reálných podmínkách zaručuje kvalitativní vyhodnocení metod s ohledem na robustnost. Experimenty ukázaly velký potenciál sítě U-Net a algoritmu SVM pro úlohu detekce aut. Navržený systém pro evaluaci v reálném čase se skládá z příjmu dat z LIDAR senzorů, klasifikace jednotlivých bodů metodami strojového učení a lokalizace kol detekovaných automobilů.

Abstract

This thesis focuses on the research of various approaches to real-world car detection by a parking robot. For this purpose, three machine learning methods based on the point cloud segmentation (the PointNet), image segmentation (the U-Net), and feature vector classification (the SVM) were implemented, and one geometrical-based method for wheel localization. Methods comparison is held on an annotated dataset of 2D measurements from three LIDAR sensors installed on a mobile parking robot. The use of the dataset collected during operating in real-world scenarios ensures the authoritative evaluation of methods with respect to robustness. Experiments indicated a great potential of the U-Net network and the SVM for the car detection task. The proposed system for real-time evaluation consists of subscription to LIDAR sensors, point-wise classification with machine learning methods, and wheel localization of detected cars.

Contents

List of Figures

1 Introduction

The modern birth of automobiles dates back to 1885 when Carl Benz built the world's first automobile. In the early 20th century, the mass production of cars began. Since then, the manufacturing processes have changed by integrating new technologies and robots, and the number of produced automobiles a year has prominently risen.

A large number of manufactured cars require vast parking lots, on which cars are systematically stored before their sale and shipping. At present, new cars are parked in the parking lots by the company's employees. But with that comes the problem of inefficiency, as employees have to walk long distances every time a car needs to be moved/relocated. For this reason, it makes sense to look for alternative solutions to this problem.

Technologies associated with autonomous driving recorded widespread interest in recent years. Many of the cars on the market boast features like adaptive cruise control or lane-centering steering, and some car manufacturers are even successfully developing a self-parking system. However, all such systems still require constant supervision from a driver. Therefore, they cannot operate autonomously in parking lots yet, not to mention sensors and hardware that increase costs.

What seems like the future of parking are autonomous parking robots. The German car manufacturer Audi has been using the Ray parking robot (see Fig. 1a), developed in partnership with Serva Transport Systems, for several years now. Audi even won the Logistics Award 2017 from the German Association of the Automotive Industry for its driverless vehicle transport system. Another example of a parking robot's successful development is the robot Stan (see Fig. 1b) from Stanley Robotics, which has been put into operation, for example, at the Lyon-Saint Exupéry Airport.

(a) Ray parking robot [1] **(b)** Stan parking robot [2]

Figure 1: Parking robots that have been already successfully put into operation.

At the time of writing this thesis, Chronorobotics Laboratory on FEE CTU in Prague is working on developing software for a similar parking robot in cooperation with Skoda Auto. This mobile robot should be able to operate safely on huge parking lots and to park cars autonomously. For this reason, it is necessary to have not only a robust car detection

system, but also a system of accurate localization of the car wheels, which will lead to a successful pickup of the car by the robot.

Figure 2: Prototype of the parking robot for which the Chronorobotics Laboratory develops the software and with which the data for experiments were collected.

There does not exist a universal approach that would guarantee robust car detection in noisy real-world conditions. Moreover, most of the research regards vehicle detection in data obtained with 3D LIDAR sensors. Hence, this thesis aims to explore and design methods used for detection in 2D LIDAR data that could be applied for the task of car detection with the mobile parking robot in the real world.

First, in section 2, I cover the most applied sensors and various approaches to car detection, and building on that, I select a set of four methods: wheels fitting, PointNet neural network, U-Net neural network, and SVM that I implement to compare them and to choose the best one. In the following section 3, I present the theory behind the selected methods. Further, in section 4, I describe the implementation of all methods and system architecture used for automatic performance evaluation and real-time classification.

To correctly evaluate the performance of the methods, I annotate my own training and testing datasets that contain 2D LIDAR data collected by the parking robot on a parking lot. The format of the data and the process of annotation is reported in section 5. Finally, in section 6, I mention how I optimized the methods and suggest a set of criteria for evaluation. Based on these criteria, I evaluate the implemented methods.

2 State of the art

The car detection can be formulated as a binary classification problem, which is a classification task of classifying all input elements into one of two categories 0 (non-vehicle) and 1 (vehicle). It is common to solve this problem by creating a model that predicts the Bernoulli probability distribution for each element.[1]

In this section, I first mention commonly used sensors (subsection 2.1). Then, I introduce state-of-the-art methods for detection in 2D range data, and for completeness, I also examine methods related to detection in 3D data. First, I cover approaches based on classifying hand-crafted features (subsection 2.2), and after that, I present Deep learning methods (subsections 2.3 2.4). Eventually, in subsection 2.5 I specify a set of methods with the potential to work in real-world scenarios which I implement.

2.1 Sensors

Autonomous cars and parking robots are often equipped with many different sensors to sense environmental information. The most commonly used sensors include a camera, radar, ultrasonic radar, and LIDAR. Each sensor provides specific information, that is why the related detection methods also differ.

With increased computational power, cameras and classifiers based on deep convolutional networks (see subsection 2.3) have found their wide application. For example, Tesla uses cameras in its Autopilot as the primary source of information on the road. Cameras bring advantages in the form of high resolutions, color determination, and low cost. On the other hand, their performance deteriorates in bad weather and low light conditions.

Recently, radar sensors have been adopted to detect obstacles or pedestrians. They produce 2-dimensional range data that is often inaccurate and sparse. However, the advantage, which can improve the robustness of the detection ([3]), is that radar scanning works relatively consistently in any weather.

LIDAR is an acronym for "light detection and ranging". It works in a similar way as radar but uses light waves from laser instead. The light wave is emitted into the environment, and then, based on the time it takes for the beam to return to the sensor, the distance is calculated. Besides, the energy of the returned beam provides intensity (reflectance) information about the surface of the captured object.

The most common is the scanning LIDAR, which can scan horizontally, vertically, or both. The 3D LIDAR rotates during scanning and provides a 360-degree view, but the produced 3D point clouds are large, making them difficult to process. Moreover, in parking scenarios, the robot needs to have information about a whole surrounding area, which means that 3D LIDAR alone is not suitable for this task. It would not cover some blind

[1] Bernoulli probability distribution is the discrete probability distribution of a random variable that belongs to category 0 with probability p and category 1 with probability 1-p.

spots, and additional sensors would be needed. A less expensive option in cost and computational power is horizontally mounted 2D LIDARs, which showed promising results in on-road detection.

LIDAR measurements are considered the most accurate, with a fast response time and resistance to most lighting conditions. Their most significant disadvantages are poor performance in bad weather, limited distance ranges, and high cost, which, however, comes down continuously.

For this thesis's purposes, the 2D LIDAR data collected by the parking robot will be used. Specifically, the robot has two lower located *SICK MICS3-CBUZ40IZ1P01* sensors on the sides and one *SICK LMS111-10100* sensor, which mainly sees the contours of the cars and walls. The sensors setup is pictured in Figure 3. Details about the data will be discussed in section 5.

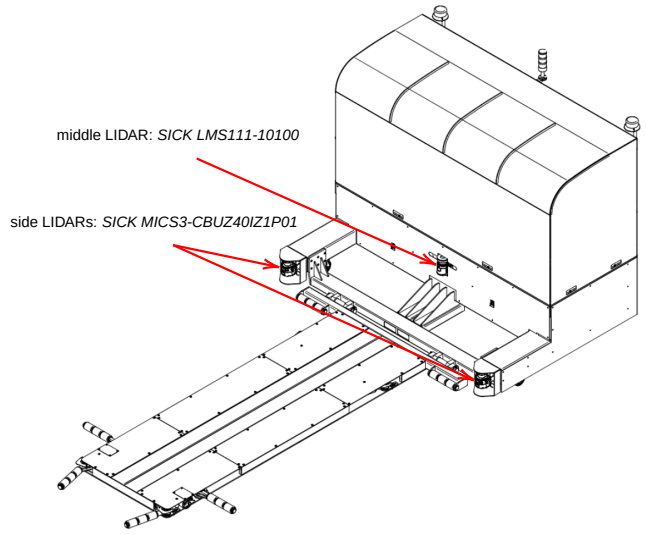


Figure 3: Illustration showing the parking robot with installed LIDAR sensors. Side LIDAR sensors *SICK MICS3-CBUZ40IZ1P01* are lower located and are capable of capturing wheels. The higher positioned middle LIDAR sensor *SICK LMS111-10100* mainly sees cars' contours and walls.

2.2 Shape fitting and hand-crafted feature classifiers

LIDAR and radar sensors are often positioned to capture cars' contours - front/rear of the car and one of the sides. A series of previous studies has indicated that these geometric attributes can be detected by pre-designed heuristics.

In [4] was showed the use of an algorithm based on template matching with Generalised Hough Transform (GHT) to estimate the orientation and position of a car in radar data. Other authors focus rather on the task of L-shape fitting. ([5–8]) They usually formulate

it as an optimization problem which can be solved by using different rectangular fitting methods ([7, 8]) or, for example, by some optimization methods for fitting two orthogonal lines.([6]) Although these methods show good results, in real-world scenarios, the contours of an object may not be fully observed by 2D range sensors, making them harder to capture by these pre-designed fitting methods.

Many existing studies in the broader literature have examined, for the task of the vehicle or pedestrian detection, the use of machine-learning techniques on a wide range of hand-crafted extracted features. The proposed systems usually consist of several parts. First, the data is segmented into clusters containing points belonging to a single object. Furthermore, multi-dimensional feature vectors are calculated from the clusters, which are used to train an appropriate classifier such as SVM ([9–15, 3]) or AdaBoost ([16, 17, 15]).

The literature reported that the effective detection of pedestrians/legs ([9, 18, 16]) and vehicles ([19, 17]) in 2D LIDAR data could be achieved with features based merely on geometric characteristics of clusters. This has also been discovered in the context of detection in 3D range data.

3D data is more difficult to process, which is why it is often projected on a 2D plane to cluster or capture potentially interesting objects.([10, 11]) On the other hand, information about the object's shape is richer and more complex, allowing computing useful features like height along the object's length or y-z plane histogram. Multiple articles also suggest taking benefit from the information about captured objects' reflectivity in the form of mean, standard deviation, or histogram.([12–15, 3])

2.3 Image-based detection with Convolutional Neural Networks

Recently, the classification of hand-crafted features has been superseded by approaches based on Deep learning. Deep learning models use multiple layers to extract features from the input data and discover a representation needed to classify them. Especially since the success of AlexNet [20], the task of image object classification and segmentation has been dramatically improved by the use of Convolutional Neural Networks (CNNs). Since LIDAR sensors generate point clouds, it is intuitive to form a structured representation from the points to apply similar convolutional architectures.

In the context of the car detection in 3D data, it is common to encode the point cloud with a 3D voxel grid. 3D voxel grid is a regularly spaced 3D grid where each voxel cell can contain a scalar value representing occupancy or hand-crafted statistics. 3D CNN then extracts features of voxels and generates a bounding box.[21–24]

Inasmuch as the 3D grids are sparse and contain a large amount of redundant data, applying 3D convolutions is both memory-wise and computationally inefficient. An alternative approach is to project the point cloud onto a 2D plane, discretize it into a 2D grid with hand-crafted features values (occupancy, intensity, density) and use standard more efficient 2D CNNs. Some works dealing with car detection project the points to a bird's

eye view (BEV) ([25–27]), [28–30] discretize the input into a 2D front-view point map instead, [31] combines both. Moreover, [29–31] use a fusion of a 3D LIDAR with a camera to improve the performance.

Similar methods are used when working with 2D data. However, there is no need to reduce the input dimension when projecting onto a ground plane since the data is already two-dimensional.

In [32, 33], the author designed a system for wheelchair and pedestrian detection from 2D radar data, which cut out a window of real-world fixed size around each radar point. These proposals were then classified with CNN, which was based on VGGNet in the first version. Similarly, in [34] was introduced a method of car detection in 2D LIDAR data, where Regions of Interest were generated using DBSCAN clustering and then fed to a CNN based on ResNet for classification.

Other studies focused on the detection using various types of CNNs on whole image grids instead. The author in [35] used Recurrent Neural Networks for predicting future states and a semantic segmentation where each state was represented by discretized fixed-size grid built around a robot. Another study ([36]) showed a successful use of an architecture based on U-Net ([37]), originally designed for biomedical image segmentation, to track people in a cluttered environment. Also, the architecture from [38] based on YOLOv3 indicated promising results. The YOLOv3 was adapted there for the task of car detection in 2D radar data, so it produced non-square bounding box proposals with orientation estimation. Worth mentioning is as well Cascade Pyramid RCNN which was proposed in [34] for vehicle detection in 2D LIDAR data.

2.4 Point Cloud-based detection with Neural Networks

The input of the neural networks, mentioned in subsection 2.3, must follow a regular structure. As the input data needs to be processed to grids using projection and quantization, useful information is naturally lost. Hence, it would be desirable to use the data in its point cloud format.

In [39] was introduced a novel type of network architecture for 3D object classification and semantic segmentation, PointNet, which can learn point features directly from an unordered raw point cloud. Extension to it is PointNet++ [40] which applies the PointNet recursively, allowing to learn local structures at different scales.

Another state-of-the-art method consuming raw point clouds as the input is VoxelNet [41], which addresses the problem of processing huge point clouds. VoxelNet divides the input point set into voxels and applies a feature learning network, 3D convolutions, and Region Proposal Network based on Faster-RCNN. PointPillars [42] explores pillar shapes rather than voxels to extract point-wise features.

Frustum PointNets [43] was the first to solve an effective proposal of 3D object locations in 3D space. A 3D frustum point cloud with an object is extracted from a 2D bounding

box, obtained by an image-based detector. The PointNet architectures are then utilized for a 3D instance segmentation and generating a 3D bounding box. The results against 2D CNNs with bird's eye view LIDAR image (see 2.3) indicated a great potential for the task of 3D object detection (cars, pedestrians, and cyclists).

As the 2D image-based proposal might fail in some challenging cases that can be fully observed only from 3D space, PointRCNN [44] presents an approach using PointNet++, which generates 3D box proposals directly from raw point clouds.

Based on Frustum PointNets, some articles ([45, 38]) dealing with car detection from 2D radar data suggest reducing the search space by establishing fixed patches around every radar point. Furthermore, they use the proposed PointNet method from [43] to classify the patches, segmentation, and 2D bounding box estimation with excellent results.

2.5 Set of chosen methods

The car detection usable on the parking robot is expected to be robust and work in real-world scenarios, including adverse weather. Therefore, to ensure it, suitable methods must be designed, learned, and evaluated on real-world data (see section 5).

Already indicated in subsection 2.1, LIDAR sensors are affected by weather conditions like heavy rain or fog. These wet conditions lead to absorption of the beams, scatter, and reduction in the reflectance of surfaces, caused by water changing the refractive index of objects. Moreover, the intensity values also depend on an incidence angle ([46]). Due to the instability of the reflectance values, using them could deteriorate the performance of learned classifiers. Hence, in this thesis, I design techniques that treat the LIDAR data as a point cloud without the intensity information.

Proper car detection is essential for picking up a car by the robot. Cars are positioned from the front during that, so LIDARs do not scan them as an L-shape, making the L-shape fitting approaches (see subsection 2.2) inapplicable. On the other hand, the lower LIDARs capture the wheels of vehicles. Thus, I will build the first method on a geometrical fitting of these wheels as corners of a fixed rectangle (more in subsection 4.2). Its main advantage is the absence of a learning process and with that accompanied need for annotated data.

Because it is possible to trace some regularities in the way the robot sees its surrounding. I try to engineer appropriate features that could intercept this and train the SVM classifier (details in subsection 4.5, which has shown significant results in people and car detection (see subsection 2.2).

Motivated by the success of deep learning methods, I also choose two different approaches with neural networks. One of them is the PointNet architecture, which achieved state-of-the-art performance in scene segmentation from point clouds (subsection 2.4). Secondly, the U-Net, already mentioned in the context of people detection (subsection 2.3), is trained on quantized point clouds projected to the ground plane. Both methods are further described in section 4.

3 Theory

In this section, I lay the theoretical foundations of the methods presented in subsection 2.5. First, in subsection 3.1, I cover the principle of the DBSCAN algorithm. Then, I introduce the deep neural networks, the PointNet architecture (subsection 3.2), and the convolutional neural network architecture U-Net (subsection 3.3). In subsection 3.4 I present the process of learning the deep learning networks, and finally, I describe the idea behind the SVM machine learning algorithm (subsection 3.5).

3.1 Density-Based Spatial Clustering of Applications with Noise

Clustering is the task of dividing unlabeled data into groups, called clusters, such that points within cluster share similarities and are dissimilar to data in other clusters. It is important in data analysis and data mining applications.

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN), presented in [47], is a clustering algorithm that can discover clusters of arbitrary shapes, find high-density regions, and filter outlier points.

The algorithm has two parameters, *Eps*, the distance threshold that determines the *Eps-neighbourhood* of a point p in a set D defined as 1, and *MinPts*, whose value represents a minimum number of points.[47] These parameters are globally used for finding clusters, thus need to be tuned.

$$N_{Eps}(p) = \{q \in D \mid dist(p, q) \leq Eps\} \tag{1}$$

The key idea is that within each cluster, the density is typically higher than outside. That means that the Eps-neighbourhood of each point in a cluster has to contain at least a fixed number of points. Based on that, if the core point condition: $|N_{Eps}| \geq MinPts$ is fulfilled, the point is labeled as a *core point*. Otherwise, it is classified either as a *border point*, if included in the Eps-neighbourhood of some core point, or as an *outlier* otherwise (see Figure 4).[47]

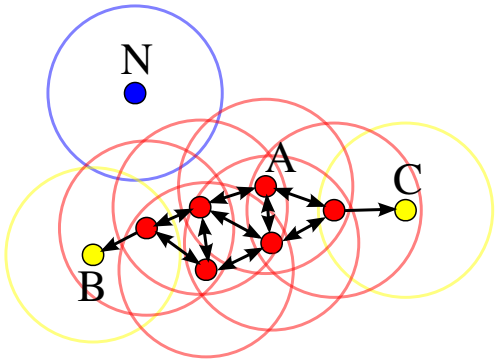


Figure 4: Illustration of core (red), border (yellow) and outlier (blue) points [48]

The basic DBSCAN algorithm, as introduced in [47], loops through all unclassified points. For every point, it calculates the Eps-neighbourhood and determines whether it satisfies the core point condition and forms a cluster or not. If yes, every point in its Eps-neighbourhood is added to it. Moreover, if the added points are also core points, all points within the Eps-neighbourhood become part of the cluster as well. This process repeats until every element from an input set has either its cluster-id or is labeled an outlier.

3.2 PointNet

3.2.1 Perceptron

An artificial neuron, the perceptron, is a fundamental unit of neural networks. It is a mathematical function $\mathbb{R}^n \to \mathbb{R}$ modeled on the working of biological neurons.

The scheme of a single neuron is showed in Figure 5. The function itself is commonly defined as a weighted sum (dot product) of an input vector $\mathbf{x} = [1, x_1, x_2, ..., x_n]^T$ and a vector of learnable weights $\mathbf{w} = [w_0, w_1, ..., w_n]^T$, followed by a non-linear activation function $\sigma(v)$. The vector of weights, where w_0 is called *bias*, basically determines the influence of the individual input components on the output value $\hat{y}$.

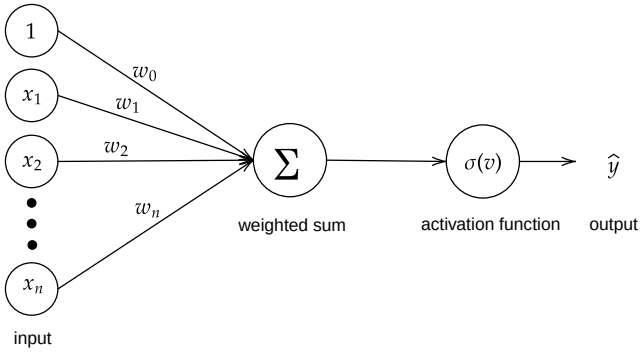


Figure 5: A single perceptron node

3.2.2 Multi-Layer Perceptron

A single perceptron can solve linearly separable problems. However, in most real-world cases, we deal with non-linearly separable problems. This limitation can be overcome by stacking perceptrons into a feedforward neural network called Multi-Layer Perceptron (MLP).

The MLP architecture consists of at least three layers. The initial layer is an *input layer*, which directly consumes the provided input. The last layer producing output is an *output*

layer. The layers between are called *hidden layers* because they are not straightly exposed to the input, and their states are unobserved. In Figure 6, there is an example of the MLP with two hidden layers where each node represents the perceptron shown in Figure 5.

All perceptrons within the same layer have no impact on each other. However, each perceptron in one layer is connected to every perceptron on the next layer. Thus, the information is fed in a forwarding direction without any feedback loop, which explains the term feedforward neural network.[49]

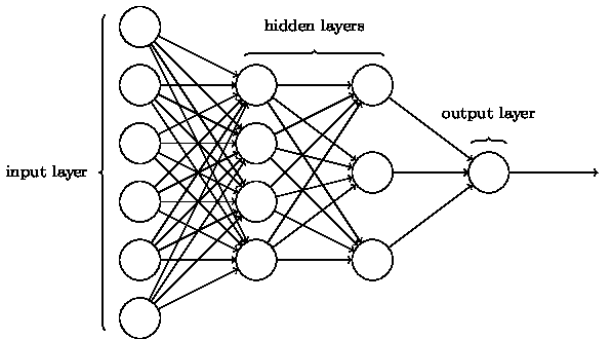


Figure 6: Scheme of Multi-layer Perceptron with two hidden layers [50]

3.2.3 Rectified Linear Unit

Inspired by biology, activation functions define the output level of nodes. The most used are non-linear activation functions that introduce non-linearities in the networks and, thus, allow solving nontrivial problems.

A popular choice in deep learning networks is Rectified Linear Unit (ReLU) function, mathematically described as an identity for positive values and zero for negative values. Compared to other non-linear activation functions like Sigmoid or Tanh, it does not suffer from vanishing gradients and is computationally efficient. A variation of it that allows a small, positive gradient for negative values is Leaky ReLU.

3.2.4 PointNet Architecture

As mentioned in subsection 2.4, the PointNet [39] is a seminal work of object classification and part/scene segmentation that takes point clouds as input. It outputs either scores for all the k candidate classes (classification task) or $n \times k$ scores for each of the n points and k candidate classes (segmentation task).

A point cloud is an unordered set of 3D points in its basic form where each point is represented with three (x, y, z) coordinates. It has unique properties that the network has to handle. The points are not isolated and form local structures, which have to be captured.

Moreover, a point cloud is invariant to rigid transformations. Thus rotation or translation should not affect its global class category nor the segmentation.[39]

The overall network architecture is pictured in Fig. 7. A foundation stone is a shared multi-layer perceptron (MLP). That means that the same MLP is used for each of the n points. The MLP maps the input space into higher dimensions, up to dimension 1024. A max-pooling, a symmetric function invariant to input permutation, is then used to aggregate information from all the points. The classification network predicts from global information, using MLP, output scores for each candidate class. The task of point segmentation requires a combination of local and global knowledge. Therefore, the global feature is after computing concatenated with per-point features and fed to the segmentation network, which extracts new per-point features, lowers the dimensionality with multiple MLPs, and predicts output scores that rely both on local and global geometry structure.[39]

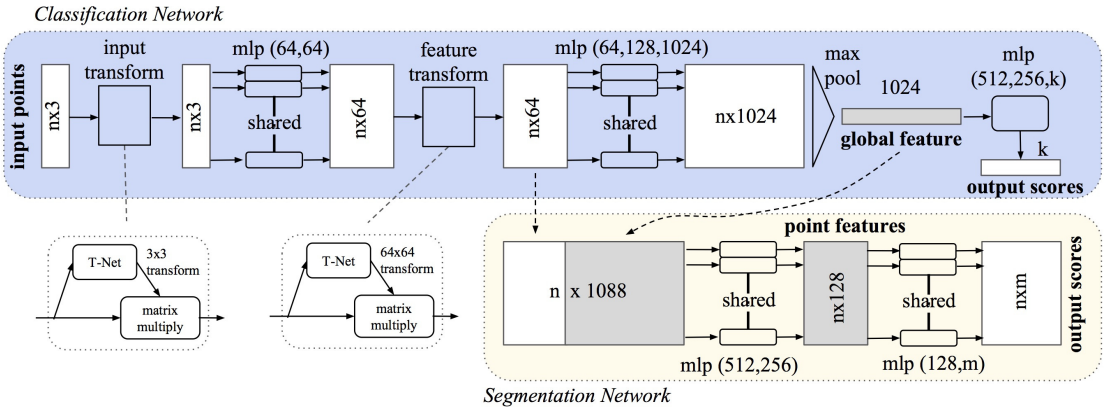


Figure 7: PointNet architecture [39]

To solve the transformation invariance, the authors adopted spatial transformer networks T-Net. The first T-Net attempts to canonicalize the input before feeding it to the network by predicting an affine transformation matrix and applying it to the input points. This transformation network is visualized in Figure 8. It consists of a shared MLP(64, 128, 1024) network (with output sizes 64, 128, and 1024), two fully connected layers[2] (FC) with output sizes 512 and 256. Finally, the matrix multiplication of output and trainable weights is computed, and a bias is added, resulting in a matrix with a size of 3×3.[39]

A similar T-Net network is used for the alignment of high-dimensional feature space as well. It has the same architecture as the first network except that the input is 64-dimensional and the output is a matrix of a size 64×64. Let A be the feature transformation matrix. The dimensionality of the spatial transform matrix in feature space is higher and can lead to overfitting. Hence, a new regularization term premising the matrix A to be orthogonal, is added.[39]

[2]A fully connected (linear) layer is one layer of artificial neurons

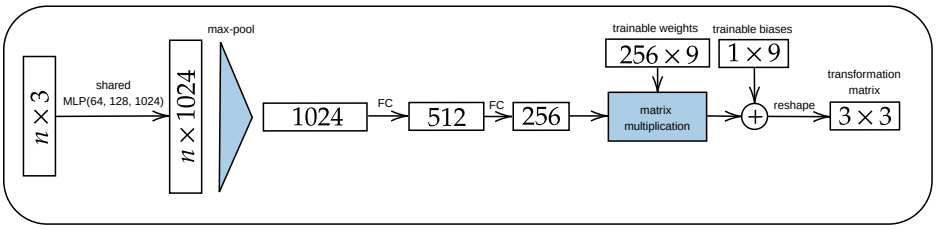


Figure 8: T-Net architecture

3.3 U-Net

3.3.1 Convolution

An image is digitally represented by a 2-dimensional matrix, where each element of the matrix represents a pixel. A convolution is a linear mathematical operation that uses sets of weights called *kernels*, organized to matrices, to extract local features in the image. It takes a 2D input and re-estimates the value of every element as the weighted sum of adjacent pixels it floats over (Figure 9). The output of a convolution is called a *feature map*.

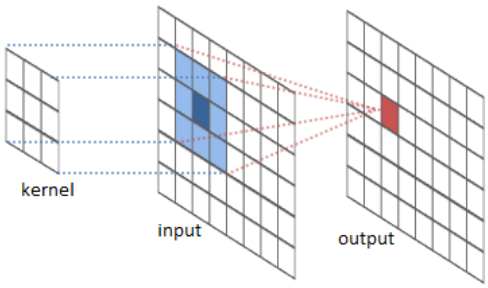


Figure 9: Convolution operation [51]

The values of kernel weights are based upon the operation to be performed. Its size depends on the size of parts that are being focused on. It is typically much smaller than the size of a picture, allowing it to slide over the whole input image and discover specific features anywhere in the image. Thus, the operation is translationally invariant.

Apart from the size, the kernel has a few parameters that modify the convolution's behavior. A *stride* defines the distance by which the sliding window moves each time in the picture. As the size of an image reduces every time a convolution operation is performed, and information on borders of the image is partially lost, additional layers can be added to the image's border, whose size defines a parameter *padding*.

Input images usually have three channels, each representing one color channel R, G, and B. Generally, when applying convolution with a kernel of size $W \times H$ on input with

C channels, for each 2D channel is used unique 2D kernel, and the outputs are summed together. Therefore, the true dimension of a kernel is $C \times W \times H$.

3.3.2 Transposed Convolution

A transposed convolution (or up-convolution) is an operation used to decompress abstract feature maps, widely utilized in scene segmentation tasks. It takes an input matrix and, likewise to standard convolution, applies a kernel. Every element in the input matrix is multiplied by the kernel, forming an output matrix, which is combined with corresponding output matrices from other input elements. The overlapping elements are then summed together. When dealing with multi-channel inputs, the kernel is, again, analogously multi-dimensional (see 3.3.1).

3.3.3 Convolutional Neural Networks

Convolutional neural network (CNN) is a deep neural network used primarily in image recognition. The CNN takes an image as input and repeatedly applies convolution to extract key features. Hence, it is composed of stacked layers of multi-channel convolution filters, which allow a hierarchical decomposition of the image, followed by non-linear activation functions and pooling layers.

The main idea is that the network is forced to learn the optimal weights of the filters during the training. Thus, it itself discovers what types of features to extract from the input based on the particular task. With increasing depth of a model, the filters learn to detect more complex higher-level features such as whole faces or objects.

The pooling layers also called downsampling layers, reduce the dimensionality of input and help prevent overfitting. Commonly used is a max-pooling layer, which, similarly to convolution, slides a kernel over an image. Instead of weighted sums, it takes a maximum value over the kernel. Thus, it has no parameters to learn. Upsampling layers, such as trainable transpose convolution or un-pooling layer, do a pseudo-reverse operation. There are many kinds of un-pooling layers. For example, max-unpooling takes indices from the previous max-pooling operation and computes partial inverse to it where non-maximal elements are set to zero.

3.3.4 U-Net Architecture

The U-Net [37], widely recognized for semantic segmentation, was first introduced for biomedical image segmentation, which requires having a label assigned for each pixel. Specifically, the output of the network has to have the same size as the input.

Authors, motivated by the success, modify and extend the fully convolutional network. The network was named after its symmetric U-shaped architecture (Figure 10), which

consists of left - contracting path and right - expansive path. The main innovation is supplementing the contracting network by successive layers with many feature channels and upsampling operators. Interestingly, the architecture doesn't contain a single fully connected layer.[37]

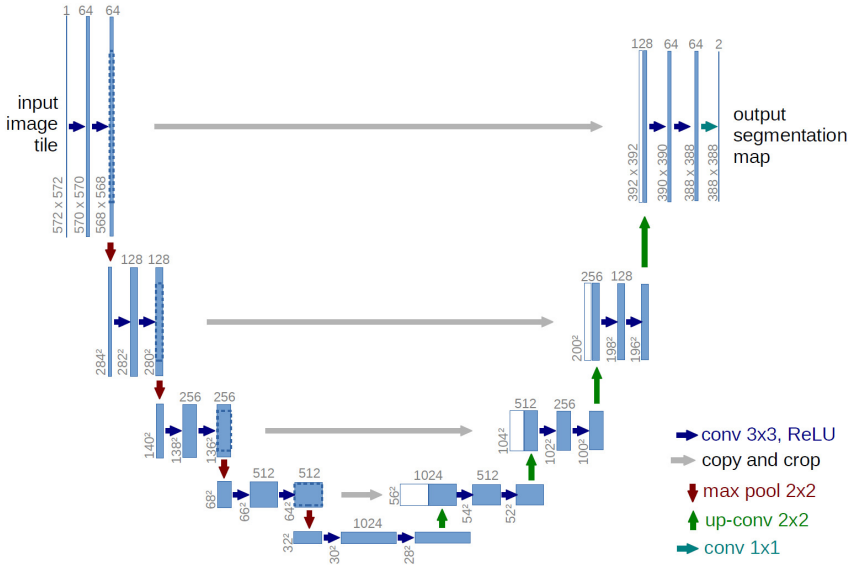


Figure 10: Original U-Net architecture [37]

The contracting path consists of two convolutions on each level with kernels of size 3×3, followed by the ReLU activation function and max-pooling layer. After each down-sampling layer, the number of feature channels is doubled, starting with 64 on top of the architecture.[37]

In each level of the expansive path, the feature map is first upsampled with 2×2 up-convolution, which halves the number of feature channels, followed by concatenation with a correspondingly cropped feature map from the contracting path, two 3×3 convolution layers, and ReLU function. At the final layer, a single 1×1 convolution reduces the number of feature channels to a number of candidate classes.[37]

3.4 Supervised Learning

In order to make the deep learning model perform well, it has to learn the extraction of appropriate features. Given a training dataset $X = \{(\mathbf{x_1}, \mathbf{y_1}), ..., (\mathbf{x_n}, \mathbf{y_n})\}$, where $\mathbf{x_i}$ is a single input data vector and $\mathbf{y_i}$ its ground-truth label vector, the model discovers how to predict the desired labels. This task is called supervised learning.

The training process is held over a training dataset. However, a deep learning network should accurately describe the test dataset as well. Thus, the model should be prevented

from overfitting, a situation when the model is too complex and does not generalize unseen data well. This is achieved by early stopping the learning process or adding regularization.

The performance is measured with loss functions, which take classifications and the ground truth as input and calculate loss between them. The goal is to find optimal weights in all layers that minimize the loss value.

3.4.1 Gradient Optimization Methods

To minimize the loss function $\mathcal{L}$, the Gradient Descent, an iterative algorithm for finding a local minimum of a differentiable function, can be used. The key idea is to compute a gradient of the loss function with respect to the weights $\frac{\partial \mathcal{L}(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}}$. Since the gradient points to the steepest ascent direction, the algorithm moves in each iteration in the opposite direction.

The Gradient Descent (GD) calculates the gradient over the entire dataset, which can be computationally intensive. Stochastic Gradient Descent (SGD) estimates the gradient over one example of data at a time instead. Hence, to reduce the complexity and increase training speed, the gradient is in each iteration estimated over mini-batches of data. The gradient itself is calculated using the backpropagation algorithm with the Chain rule.

The optimizing algorithm typically runs for several epochs until convergence. In the beginning, the weights of the model are initialized. Then, for every epoch, it iterates through batches of data, computes predicted labels, corresponding loss and gradient of the loss function, and updates the weights according to an updating rule. A tunable hyperparameter *learning rate* α defines how large the step in the direction of the gradient will be.

3.5 Support Vector Machines

A Support Vector Machines (SVM) is a supervised machine learning algorithm introduced in [52], commonly employed for binary classification problems. It takes a set of m n-dimensional vectors $\mathbf{x_i}$ and their labels y_i as input and looks for an optimal hyperplane separating the two classes.

The situation in 2D space is illusrated in Figure 11. A hyperplane (Eq. 2) is a flat subspace of dimension n–1, where $\boldsymbol{\omega}$ is an orthogonal vector to the separating hyperplane and b is an offset from the origin. A minimum distance of the hyperplane to the input observations is called a *margin*. The closest vectors to the hyperplane are called *support vectors* (on the dashed boundaries in Fig. 11).[52]

$$H : \boldsymbol{\omega}^T \mathbf{x_i} + b = 0 \tag{2}$$

The optimal hyperplane is considered to be the one with the largest distance to the input examples. Hence, to find it, the SVM algorithm maximizes the minimum distance to the training data. It was shown that this task could be formulated as a quadratic optimization

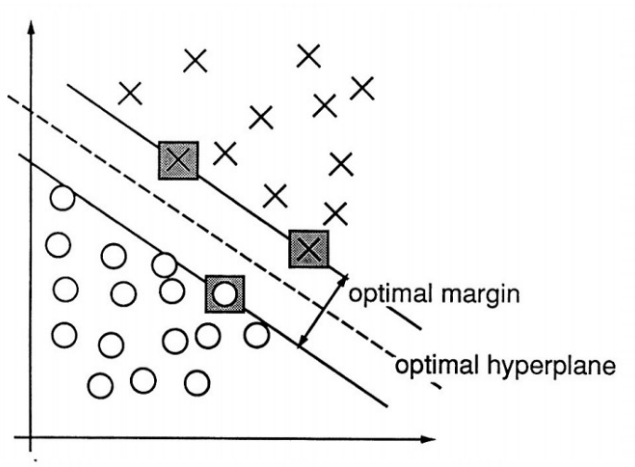


Figure 11: Illustration of the optimal separating hyperplane in 2D space [52]

problem, which can be iteratively solved, for example, with the SGD (see subsection 3.4). However, it is convenient to solve its dual form, which has the same optimal solution and depends merely on pairwise dot products of data examples.[52]

3.5.1 Non-linear Separation and Kernel Trick

Problems in the real world are usually non-linearly separable, and thus, the SVM would not find an optimal linear hyperplane. There are two solutions. The first method, called soft margin SVM, is to introduce an additional penalty for wrong guesses. The largeness of the penalty is determined with a tunable parameter C.[52]

The main idea of the second approach is to map the input vectors to a higher dimensions, where the data are more likely to be linearly separable. Let $\phi(\mathbf{x_i})$ be a feature mapping function. Then, the dot product in the dual form would be replaced by $\phi(\mathbf{x_i})^T\phi(\mathbf{x_j})$. It can be showed that mapping to higher dimensions and applying the dot product is equivalent to computing the corresponding kernel function $K(\mathbf{x_i}, \mathbf{x_j}) = \phi(\mathbf{x_i})^T\phi(\mathbf{x_j})$ instead, which is often more efficient. This approach is called a kernel trick.[52]

Commonly used kernels are, for instance, a Polynomial function and a Radial Basis function (RBF). The RBF on two examples $\mathbf{x_i}$ and $\mathbf{x_j}$ is defined as Eq. 3, where γ is a trainable parameter.[52]

$$K_{RBF}(\mathbf{x_i}, \mathbf{x_j}) = exp\left(-\gamma\|\mathbf{x_i} - \mathbf{x_j}\|^2\right) \tag{3}$$

4 System Description

This section covers the overall system architecture used both for real-time classification with methods proposed in subsection 2.5. The system pipeline is indicated in Figure 12. Measurements from the LIDAR sensors (see subsection 2.1) are obtained with a ROS part (subsection 4.1). The data are appropriately preprocessed and passed to a classifier, which gives output in the form of point-wise class predictions or, in the case of the wheels fitting method (subsection 4.2), directly in the form of wheel positions. The point-wise classifications then go through a wheel extraction (subsection 4.3.3) that localizes wheels. Furthermore, I describe preprocessing and implementation of the segmentation with the PointNet (subsection 4.3), the U-Net (4.4), and the classification process with the SVM algorithm (subsection 4.5). The same system architecture, only without the ROS part, is utilized for automatic evaluation, which I introduce later in subsection 6.2.2.

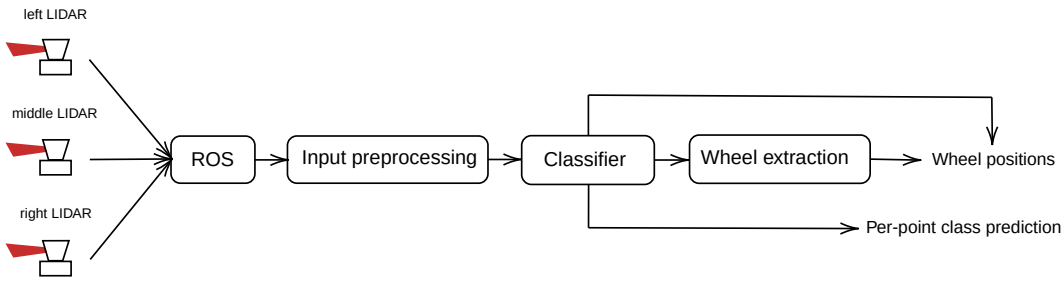


Figure 12: LIDAR data processing pipeline. The ROS part subscribes to LIDAR sensors. The obtained measurements are appropriately preprocessed and passed to a classifier that output point-wise classifications or, in the case of the wheels fitting method, the wheel positions. The point-wise classifications then go through the wheel extraction stage to localize the wheels.

4.1 Robot Operating System

The parking robot operates in Robot Operating System (ROS) [53], an open-source framework for robot software. The ROS is designed as a peer-to-peer network of processes called *nodes*, organized into ROS packages. The nodes communicate with each other via *topics* and can be distributed across machines. Every node can loosely publish and subscribe *messages*, a special ROS data type, to any ROS topics.

Data from laser scanners are published through their own topics. Messages are of *LaserScan* data type, which includes points effectively stored in polar coordinates and intensities. Cartesian coordinates of an ith point can be computed with Equation 4, where *angle_min*, *angle_increment* and *ranges*, an array of radial distances, are parameters involved in the message.

$$\begin{aligned} x &= ranges[i] \cdot cos(angle\_min + (i \cdot angle\_increment)) \\ y &= ranges[i] \cdot sin(angle\_min + (i \cdot angle\_increment)) \end{aligned} \tag{4}$$

4.1.1 Implementation

Every implemented method is realized in a new node. The nodes share the same structure in the ROS language. They subscribe to messages from all three LIDARs and process them to obtain a point cloud format, which is fed to the particular detection algorithm.. The following example shows a possible realization:

```
import rospy
import message_filters
import laser_geometry.laser_geometry as lg
from tf2_sensor_msgs.tf2_sensor_msgs import do_transform_cloud
from sensor_msgs import LaserScan, PointCloud2, PointCloud

def callback(msg_right, msg_left, msg_middle):
    lp = lg.LaserProjection()
    pc2_msg_right = lp.projectLaser(msg_right)
    cloud_right = do_transform_cloud(pc2_msg_right, lidarTransform_right)
    msg = PointCloud()
    ...     # placeholder for the car detection method
    pub.publish(msg)

if __name__ == '__main__':
    rospy.init_node('detection_node', anonymous=True)

    sub_right = message_filters.Subscriber('/back_right/sick_safetyscanners/scan', LaserScan)
    sub_left = message_filters.Subscriber('/back_left/sick_safetyscanners/scan', LaserScan)
    sub_middle = message_filters.Subscriber('/back_middle/scan', LaserScan)
    sub = message_filters.ApproximateTimeSynchronizer([sub_right, sub_left, sub_middle],
     queue_size=2, slop=0.2)
    sub.registerCallback(callback)
    pub = rospy.Publisher('/wheels_new', PointCloud, queue_size=5)

    rospy.spin()
```

As each LIDAR has information about another part of a scene, it is desirable to synchronize and use them together. The synchronization is held by a function *ApproximateTimeSynchronizer()* from a *message_filters* package, which synchronizes the incoming message filters by their timestamps. After receiving new data, a callback function registered by a function *registerCallback()* is invoked, responsible for the following data processing and triggering the detection methods.

Apart from time synchronization, the data need to be transformed into the same coordinate system (a global frame *map*). That is accomplished with a package *laser_geometry* and its function *projectLaser()*.

Finally, the messages are converted from LaserScan to *PointCloud2* data type, allowing

to directly run the detection methods, with a function *do_transform_cloud()* from a package *tf2_sensor_msgs*, directly allowing running the vehicle detection.

The wheel positions estimated with the methods are published in a point cloud format through a new topic *wheels_new*.

4.2 Wheels Fitting

As already pointed earlier, the lower positioned LIDARs on the parking robot can sense the wheels of the car. Concerning that, the wheels fitting method tries to capture them using simple geometry. For this task, the third LIDAR, which sees the contours of cars and walls, isn't used. The system consists of the part in the ROS, a clustering part, and a geometrical fitting.

4.2.1 Clustering

For filtering out background noise and capturing continuous groups of points, the DBSCAN is utilized. Considering the effectiveness, I use the DBSCAN from a Python library *scikit-learn* [54] over my own implementation. The parameters were experimentally determined to values $minPts = 3$ and $Eps = 0.35$.

Each cluster is estimated by its center. In some cases, the robot senses the front bumper of the car, so additional processing is required. More accurately, if a particular cluster containing at least 25 points fulfills that maximal distance between all inner points falls into the interval $[1; 3.1]$, it is split into fifths according to the distance from the first point. Ideally, at least two of them should approximate the positions of the front wheels. I chose all parameters experimentally.

4.2.2 Fitting Algorithm

The fitting process builds on the geometrical properties of a car and tries to find all four wheels of it. Its performance is illustrated with two examples in RVIZ [55] in Figure 13. Experiments showed that it is crucial to require all four wheels to prevent false-positive classifications.

The algorithm iterates through all given clusters and looks for clusters A, B in the distance close to the width of the car from each other. If it finds any such clusters, the directional vector $\mathbf{u} = \mathbf{AB}$ and associated normalized normal vector $\mathbf{n}$ are calculated. Afterward, approximate positions C, D of the two remaining wheels on both sides are estimated (see Eq. 5), and it searches for clusters near them. If those clusters are found, it determines the best ones, concerning the deviation from ideal distances, which then create a potential car.

$$\begin{aligned} C &= A \pm LENGTH \cdot \mathbf{n} \\ D &= B \pm LENGTH \cdot \mathbf{n} \end{aligned} \tag{5}$$

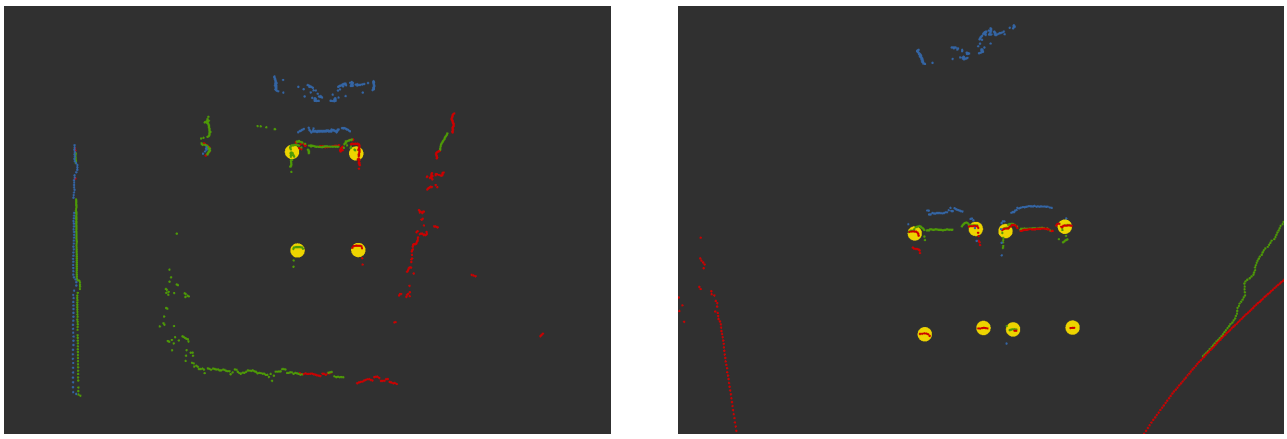


Figure 13: Images from RVIZ showing the wheels fitting method applied to two examples from the collected data. The red/green/blue points stand for the measurements captured with the right/left/middle LIDAR sensor. The yellow points represent the correctly estimated wheel positions by the method.

Respecting that each cluster can be part of only one car, the new potential car is compared with already accepted potential cars, containing any of the present clusters. In case that the new potential car has better distance properties, the other potential cars are discarded before accepting this new one.

4.3 PointNet Classifier

Although, as noted in subsection 2.4, many articles suggested proposing regions of interest followed by classification and segmentation, this procedure can be computationally demanding. Hence, I approach car detection as a scene segmentation, for which I utilize the PointNet segmentation network. The system consists of the part in ROS language, an input preprocessing, the PointNet model, and extraction of wheels.

4.3.1 Input Representation

PointNet was originally intended for three and more dimensional data. It could be adapted for two-dimensional data. Nonetheless, it is generally desirable to give the classifier as much relevant information as possible and let it decide whether to use them. Since a pertinence to a particular LIDAR offer potentially valuable information, it is added to the two-dimensional coordinates to form three-dimensional data. The 3rd element is set to 0.4 if it comes from the right LIDAR, 0.7 if from the left LIDAR, and 1 otherwise.

It was already pointed out earlier that the PointNet takes a point cloud as input. However, it expects a fixed number of n points in it, which I set to 1200. LIDARs can produce up to 1526 finite points, ordinarily, it is less. Hence, the most distant points from the

robot are discarded until the size of the point cloud matches 1200. Similarly, the points are randomly duplicated if there are fewer of them.

In order to speed up the learning process and make the model converge faster, it is practical to normalize the data before feeding them to the neural network. Standard techniques are, for instance, setting the mean to zero, scaling the input ranges to $[0; 1]$ interval, or methods like min-max normalization. However, no rule guarantees better results. Hence, I will try three approaches. The first only involves setting the mean to zero. Since the wide range of coordinates can make larger values dominant, the other methods, apart from the zero-mean, map the coordinates to the unit circle to distribute the importance of each input equally. One of them calculates the maximal euclidean distance from the origin and divides all coordinates by it (dynamic scaling). However, this method loses the information about the scale. Hence, the other method (static scaling) divides the coordinates by the fixed maximal range that the lasers can capture, which is 40 meters.

4.3.2 PointNet Model Implementation

I slightly modified the implementation [56] of the PointNet model in Python library Pytorch. The architecture is almost identical to the network introduced in section 3.2.4, except, as suggested in the original implementation, it does not involve the smaller T-Net network that estimates the 3D rotation matrix. That is specific to the scene segmentation, for there could be more than one object in the scene, which would make the finding of an appropriate rotation challenging.

The MLP with shared weights can be implemented as a 1-dimensional convolution with a kernel of size 1. To allow using such convolutions, the input is first transposed to the shape 3×1200. Already outlined, the implementation then strictly follows the architecture in Figure 7 without the input transformation. The MLPs are represented by 1D convolution, followed by batch normalization layer and the ReLU activation function. The T-Net network for transforming the features is implemented with the 1D convolution as well, followed by three fully connected linear layers. To map the output to log probabilities, the last layer is concluded with the logarithmic Softmax function. Finally, the output of the network is transposed back to the shape $1200 \times k$, where the $k = 2$ is a number of candidate classes.

4.3.3 Wheel Extraction

The output of the PointNet network is in the format of point-wise class prediction. However, the robot needs the positions of wheels, and thus, it is necessary to extract these positions from points predicted as a car class. The extraction is almost the same as the wheels fitting method presented in subsection 4.2, with the difference that if the method does not find all four wheels, it outputs only three or two of them. The only condition is that there is a bumper detected from the middle LIDAR sensor nearby.

4.4 U-Net Classifier

Although there are various kinds of possible approaches to car detection with CNNs, discussed in section 2.3, I address it as the scene segmentation task for the same reason as with the PointNet. Motivated by the success, I utilize the U-Net architecture presented in section 3.3. The overall system includes the part in ROS language, input preprocessing, U-Net classifier, and wheels extraction, which is exactly the same as the one presented in subsection 4.3.3.

4.4.1 Input Preprocessing

The U-Net expects an image as input. Nevertheless, the produced LIDAR data are in a point cloud format. Hence, it is necessary to convert them to the image. As the point cloud is 2-dimensional, there is no need to project the data to the ground plane. Therefore the point cloud is only discretized into a grid of fixed size $H \times W$ and fixed resolution. To cover most of the observed scene, I experimentally chose the image size 256×256 with a resolution of 7.5 cm.

The grid is initialized to zero values. First, the centroid of the point cloud is calculated, on which is the grid centered. Then, for every point in the point cloud, its column and row index is computed with Eq. 6. It translates the point to the centroid, discretizes the coordinates in the x and y-axis, applies the ceiling function, and adds up these coordinates with the center of the grid. Finally, the value in the grid on thereby calculated coordinates is set to one.

$$\begin{aligned} column &= ceil((W//2 - 1) + (point_x - centroid_x)/resolution) \\ row &= ceil((H//2 - 1) - (point_y - centroid_y)/resolution) \end{aligned} \tag{6}$$

In order to leverage the information about the pertinence to the particular LIDAR, the data from each sensor are gathered to a unique grid as a new image channel. Since there are three LIDAR sensors, the image has three channels altogether, which can be interpreted as RGB channels. An example of a so processed point cloud is visualized in Figure 14a. The red channel represents the measurements from the right LIDAR, the green from the left, and the blue channel from the middle LIDAR. Corresponding pixels that belong to the car class are highlighted in Figure 14b. I will mention more on the data annotation in the following section 5.

4.4.2 U-Net Model Implementation

I implemented the U-Net model in the Python library Pytorch. The architecture follows the structure introduced in subsection 3.3.4, the size of kernels and stride are identical for all layers. However, I reduced its size. The input is smaller than in the original paper. Thus, the contracting path starts with mapping to 32 channels and ends with 512 channels. In each

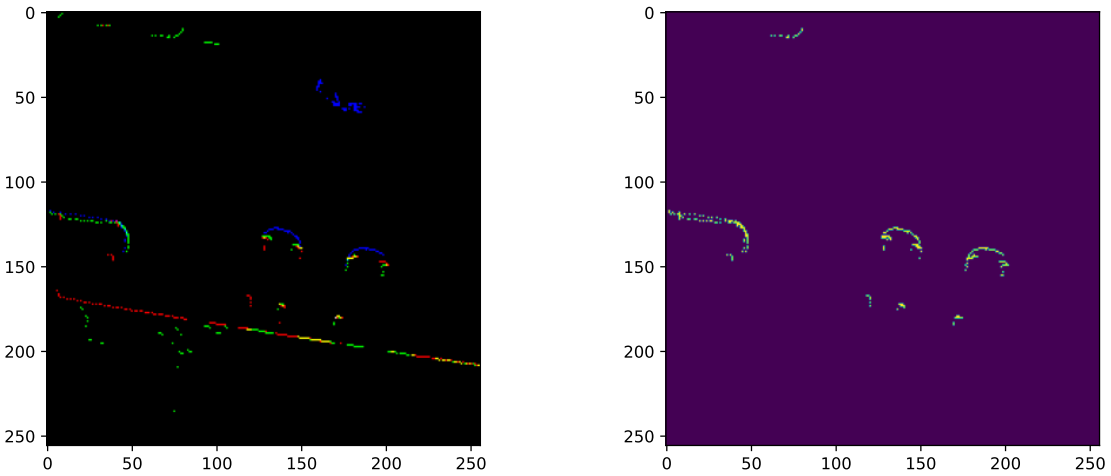


Figure 14: LIDAR measurements processed into images that the U-Net network takes as input. The x-axis and y-axis represent pixel coordinates. Every pixel cell occupies 7.5×7.5 cm

level of the expansive path, I apply transposed convolution to upsample the image. Then, I concatenate the feature map with the corresponding feature map from the contracting path, followed by two convolutional layers, which reduce the number of channels to half. Every convolutional layer is concluded by the LeakyReLU activation function. The output, which has two channels, each representing the candidate class, is obtained with a convolutional layer with a 1×1 kernel.

4.5 SVM Classifier

As indicated in section 2.2, the SVM algorithm proved to be useful in people and car detection. However, its performance depends on proposed features, which have to be carefully hand-engineered for the specific task. The overall system is carried out as follows: the ROS part subscribes to the LIDAR data from all three sensors, and the clustering groups the points belonging to the same object. Clusters with a greater inner maximum distance than the threshold of 4.2 meters are discarded to filter out irrelevant walls. Then, for each cluster, the devised features are extracted. Hereby obtained feature vectors are classified with the SVM as a car or background, and finally, the wheels extraction localizes the wheels of identified cars. All points in a scene are labeled according to a predicted label of a cluster they belong to.

The clustering is realized with the DBSCAN algorithm from the Python library *scikit-learn* [54] with the same parameters as in subsection 4.2.1. I used the implementation of the SVM classifier from the the same Python library *scikit-learn*. The wheels extraction is almost identical to the one presented in subsection 4.3.3 with the difference that the

clustering is already done in the data preprocessing part.

4.5.1 Feature Extraction

Like many articles (see subsection 2.2) suggested, I design features based merely on geometrics. Since legs captured with a 2D LIDAR look similar to a wheel, I will adopt some of the features originally proposed for people detection. Moreover, the features must be invariant to rotations, as the SVM is not feasible for large datasets, and thus, it is not possible to cover them in the training dataset. Altogether, I constructed the following eight features that form 10-dimensional feature vectors:

1. **number of LIDAR mesurements** (dim=3): The number of measurements differs with the size and the type of the observed object. For instance, the front of the car typically contains a great number of measurements from all three LIDARs, yet the wheel and noise clusters often consist only of measurements from the side LIDAR sensors (see Fig. 13).([3, 12–14, 16–19])

2. **Euclidean distance to the object from the robot** (dim=1): The number of measurements decreases with increasing distance to the object. Hence, this feature can be thought of as a correction of the previous feature.([3, 12–14])

3. **number of adjacent clusters in a 2-meter radius** (dim=1): This feature is based on the fact that noise clusters, in opposition to car clusters, have more other noise clusters near it typically.

4. **diameter of a cluster** (dim=1): The diameter of a cluster corresponds to the maximum distance in it.([9, 16, 19])

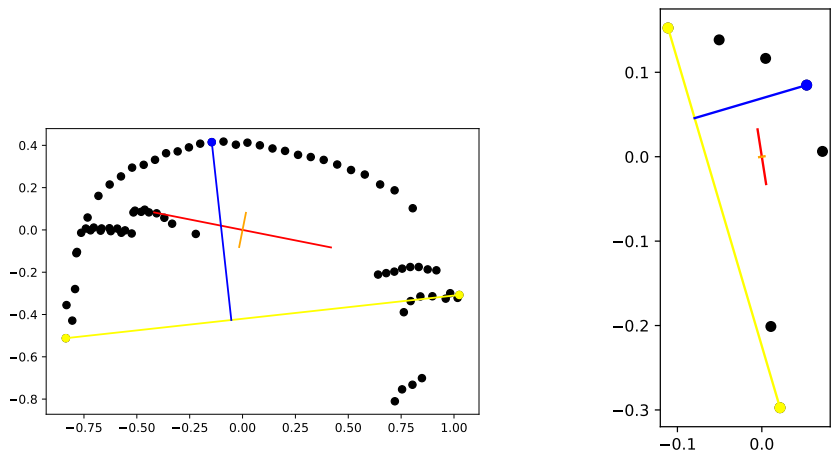
5. **width of a cluster** (dim=1): The width of a cluster is the maximum distance of a point from a line formed by the two most distant points in a cluster. Together with the previous feature, it gives information about the cluster shape.([9])

6. **standard deviation from a cluster centroid** (dim=1): The standard deviation (Eq. 7) measures the variation of point distances from the centroid of a cluster.([16–19])

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i}\|\mathbf{x_i} - \hat{\mathbf{x}}\|} \tag{7}$$

7. **ratio of eigenvalues** (dim=1): Eigenvectors (principal components) of a covariance matrix represent directions in which the variation of data values is maximal. The corresponding eigenvalues to eigenvectors explain the axes' magnitude. The ratio between the two eigenvalues describes how much linear a cluster of points is. The eigenvalues are computed with the *PCA*[3] from the Python library *scikit-learn*.

[3]PCA = Principal Component Analysis



(a) **(b)**

Figure 15: Visualized geometrical features on car clusters. The yellow points represent the most distant points inside a cluster. The distance of the blue point from a line they form defines the width of a cluster. The red and orange lines in the origin demonstrate the direction of the eigenvectors scaled with their corresponding eigenvalues.

8. **linearity** (dim=1): The smaller eigenvalue represents an error of fitting the cluster with a line and gives information about the linearity of a cluster together with the previous feature.

Some of the proposed features are illustrated in Figure 15. It shows the difference between a number of measurements in a large cluster containing the front of a car (Fig. 15a) and a small cluster of a single wheel (Fig. 15b).

Inasmuch as the features are in different numeric ranges and the ones with the greater numeric ranges could dominate the smaller ones, similar to neural networks, it is necessary to scale the features before feeding them to the SVM classifier. I chose to use the standard score (z-score) for standardization, computed with a *StandardScaler()* from the Python library *scikit-learn*. From each feature, the mean is subtracted, and the feature is scaled with the standard deviation.

5 Dataset

Supervised learning (see subsection 3.4) requires a training dataset with annotated data, from which it can learn the appropriate input representation. The laboratory of chronorobotics on FEE CTU in Prague provided me with data collected by the parking robot in real-world conditions. The data contain situations where the robot is driving along a car line, picking up a car, or driving around a car. For gathering the data, a rosbag is used, a file format in the ROS framework for storing ROS messages.

To incorporate most of the possible situations that can occur, I annotated 15 rosbags. The recorded data take about 2300 seconds in total. It is crucial to evaluate the performance of the trained models on unseen data, which allows seeing how well the model generalizes. Therefore, I split up the annotated data into training and testing datasets with a ratio of $11 : 4$. It is necessary to state that none of the rosbags in the testing dataset was collected on the same day as any rosbag from training data, ensuring the uniqueness of the testing dataset and legit evaluation. After data augmentation, the training dataset contains 7968 samples, and the testing dataset 2403 samples.

5.1 Annotation

Identically to the ROS part introduced in subsection 4.1, I subscribe to the LIDARs and convert the messages to a point cloud. For each point, it is determined whether it belongs to a car. According to that, it is labeled as 0 (non-vehicle) or 1 (vehicle). For this task, I designed a semi-automatic method based on using approximated coordinates of the four wheels of every car in the scene.

In the global frame *map*, the parked cars are primarily static. I take advantage of that. For every parked car, I estimate by hand the coordinates of its four wheels. Since the vehicles are only moving if the parking robot picks them up and drives with them, I use for the annotation of moving cars the wheels fitting method proposed in subsection 4.2, which is capable of estimating the coordinates of four wheels of these cars.

Figure 16 shows an annotated scene. The points that belong to the car class are highlighted according to their pertinence to the right/left/middle LIDAR with red/green/blue color. It should be noted that the ground truth involves only point-wise class labels, and the rectangles in Fig. 16 are there only for a better understanding.

For every car are first estimated positions of its wheels (corners of orange rectangles in Fig. 16). Then, implicit equations of lines forming the sides of a rectangle are computed. For example, an implicit equation of a line l with endpoints (x_p, y_p) and (x_q, y_q) is calculated as Eq. 8. A point (x, y) is then labeled as the car class if it satisfies four inequations that limit the maximal distance (Eq. 8) of the point from each of the lines.

$$\begin{aligned} l(x,y) : ax + by + c &= (y_p - y_q)x + (x_q - x_p)y + (x_p y_q - x_q y_p) = 0 \\ dist_l(x,y) &= \frac{a \cdot x + b \cdot y + c}{\sqrt{a^2 + b^2}} \end{aligned} \tag{8}$$

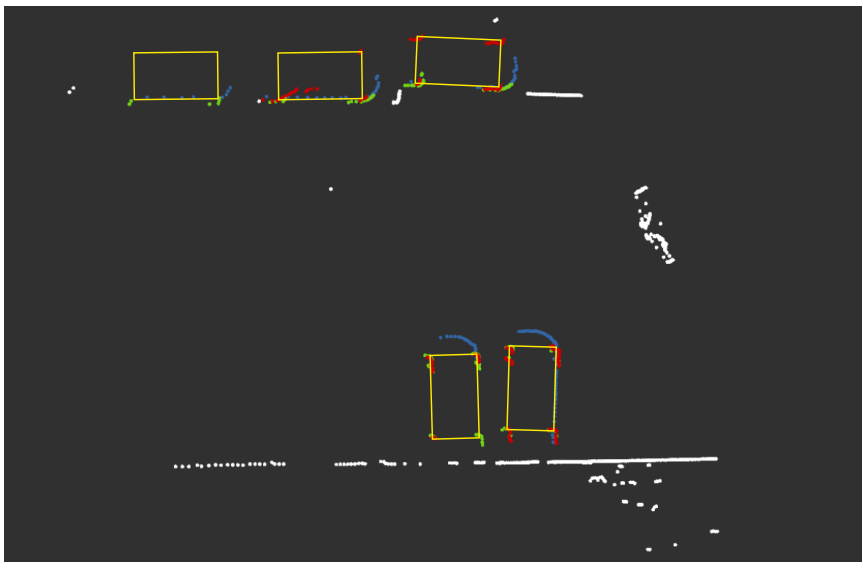


Figure 16: Illustration of the annotation process in RVIZ. Corners of the rectangles represent manually estimated car wheels, from which the inequations used for a class assignment are formed.

5.2 Data Preprocessing and Augmentation

The preprocessing for the PointNet architecture involves reducing the size of a point cloud to 1200 points and adding the 3rd dimension as stated in subsection 4.3.1.

Similarly, for the U-Net, it is necessary, apart from the RGB image generating (see subsection 4.4.1), to create a grid with labels for each pixel (see Figure 14b). This grid is first initialized to zero values. If any point mapped to a specific pixel belongs to the car class, the particular position is set to value one.

For the SVM, the preprocessing consists of reducing the size of the whole non-rotated training and testing dataset due to its optimizing method. It is followed by clustering, and computing the feature vectors that consists of features proposed in subsection 4.5.1. The training dataset contains 6670 labeled feature vectors, the size of the testing dataset is 2312.

Data augmentation is used for increasing the size of the training dataset by adding slightly modified duplicates of already existing samples. There are many various techniques such as scaling, cropping, translation, rotation, etc. I suggest only rotation, as the other mentioned techniques could potentially discard valuable information.

The rotation is essential for the robustness of both the PointNet and U-Net networks. Not only it allows the PointNet to learn the transforming T-Net network properly, though it helps the networks to understand all the possible and unpredictable rotations which will most likely occur in real-world scenarios. The process contains rotating each sample about the z-axis by a random angle. Moreover, some original non-rotated samples were discarded due to the statics of the scene, which could lead to overfitting.

6 Experiments

It is desirable to find the best method for real-time car detection. Hence, in this section, I first describe the optimizing process of the proposed models, and I mention the parameters that led to finding the finest models (subsection 6.1). Then, in subsection 6.2, I state the evaluation criteria and details on realizing the requisite experiments. Finally, I present the results that the particular models achieved.

6.1 Training of Classifiers

6.1.1 Training of PointNet

Since the PointNet is concluded with logarithmic Softmax function (see subsection 4.3.2), I use the NLL (Negative Log-Likelihood) loss function for measuring the performance. For the optimization, I chose the batch size of 128 samples and the Adam optimizer with a learning rate of 10^{-4} for all models.

To determine the optimal model, I visualize the training process with learning curves. Figure 17 shows, for each of the normalizations (see subsection 4.3.1): no-scaling (Fig. 17a), dynamic scaling (Fig. 17b), and static scaling (Fig. 17c), the average loss function values in every iteration on both the training and testing dataset. The performance on the testing dataset gives information on how well the model generalizes and thus, helps to specify when the model starts to overfit on the training dataset. This happens when the training loss continues to decrease, but the testing loss begins to increase. The optimal trained models that I chose are demonstrated in Figure 17 with the orange points.

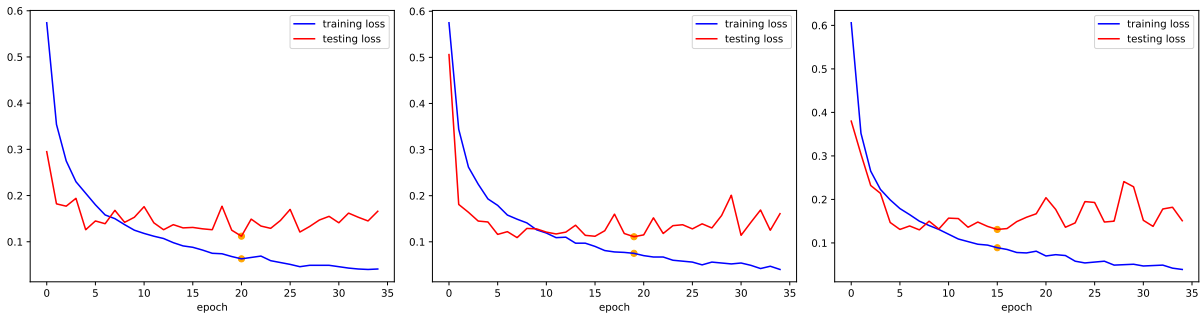


Figure 17: Training and testing loss curves of PointNet models

6.1.2 Training of U-Net

I trained the U-Net network with the Adam optimizer with a learning rate of 0.001 and a batch size of 32 samples. The performance was evaluated with the Cross-entropy loss

function.

Since the classified images are sparse, the value of the loss function on both training and testing data was almost zero. Hence, it is useless to visualize the learning curve nor the pixel accuracy curve, which was almost perfect in each iteration. The optimal trained model was determined according to the classifications on both the training and testing dataset.

6.1.3 Training of SVM

The SVM has few parameters that need to be tuned (see subsection 3.5.1): a parameter C, γ, and a kernel type. It is common to find the optimal hyperparameters with a method called grid search, which tries all of the possible combinations of given values for the hyperparameters. I used an implementation of the grid search in Python library *scikit-learn* [54]. Together with the grid search, I used 3-fold cross-validation, which solves the problem of the missing validation data by splitting the training data.

The performance of the SVM models with promising values of hyperparameters was measured on the testing dataset. The best accuracy 92.43% was obtained with the RBF kernel and values $C = 100$, $\gamma = 0.1$. Same accuracy also had a model with the RBF kernel and values: $C = 10$, $\gamma = 0.1$. However, I chose the first one, which needs fewer support vectors for the classification

6.2 Evaluation

6.2.1 Set of Evaluation Criteria

One of the generally crucial aspects is the correctness of the classifications of the selected models. However, the implemented approaches have different outputs. To unitize them and make them comparable with each other, I measure the performance on point-wise classifications. More specifically, I measure the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Based on that, I calculate accuracy, precision, and recall (see Eq. 9). Accuracy describes the overall correctness of classifications regardless of the classes. Precision expresses how trustable the model is with classifications of positives (car class). The recall represents the proportion of correctly classified positives.

$$\begin{aligned} accuracy &= \frac{TP + TN}{TP + FP + TN + FN} \\ precision &= \frac{TP}{TP + FP} \\ recall &= \frac{TP}{TP + FN} \end{aligned} \tag{9}$$

Since the parking robot works in real-time, it is necessary for the methods to be able to give a new classification for a scene as soon as possible after its observation. Moreover,

each approach carries out variously compute-intensive operations and requires different data preprocessing. Hence, another criterion is the computational time measured for each stage of the methods.

As already mentioned, an essential output of the methods is the positions of car wheels. Inasmuch as the process of extracting the wheels is the same for all the approaches (see section 4), and the actual accurate wheel positions are not unambiguously determined, I do not measure the precision of the wheel detection. For comparing the ability of methods to extract wheels, I suggest using the frequency and number of correct wheel position estimates, which directly depends on how well the methods can detect cars in a scene correctly. More specifically, for each scene sample, the methods predict wheel positions, and it is evaluated how many times do the methods manage to find a car and how many wheels the found cars consisted of.

6.2.2 Evaluation Details

The point-wise evaluation is realized on the testing dataset of size 2403 (see section 5), which allows measuring how well the model generalizes. Experiments for the wheel estimation are held on one specific rosbag of size 112 from the testing dataset. The rosbag contains a typical situation for picking up a car - scenes with the parking robot and one car with fully observable four wheels in front of it at different distances. The evaluation is performed on Asus ROG G551J with a graphic card Nvidia GTX 850M 4GB, used for classification with the neural networks.

Implemented evaluation scripts used for method comparison strictly follow the system pipeline presented in section 4. Nevertheless, the input is an annotated dataset of point clouds. Thus, the evaluation process starts with data preprocessing, and the ROS part is omitted. It should be noted that for the neural networks, the classified class is considered to be the one with the maximum value of the output function.

The PointNet processes 1200 points of all 1526 possible incoming measurements from LIDARs (see subsection 4.3.1). Similarly, the U-Net is not able to processing a whole scene with only one input image due to its size (see subsection 4.4.1). Therefore, unprocessed points are default classified as a background (non-vehicle).

Inasmuch as the wheels fitting method (see subsection 4.2) is not capable of point-wise class classification, it is not included in the point-wise evaluation criterion moreover nor in the computational time criterion, as it forms the wheel extraction stage of all the other proposed methods.

6.3 Results

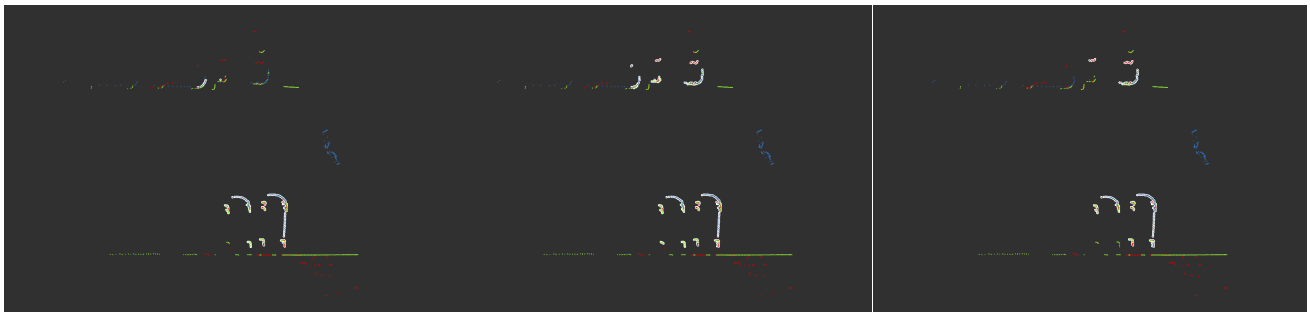
6.3.1 Point-wise Classifications

The point-wise classification of the car class was evaluated for the PointNet model with three different types of normalization (see subsection 4.3.1), for the U-Net model, and the SVM model. Results are enrolled in Table 1.

Clearly, the PointNet with static scaling and without scaling had similar performance. Opposing them, the PointNet with the dynamic scaling had expectedly worse results, as it discards valuable information about fixed scene scale. The SVM had similar accuracy to the PointNet and better precision. On the other hand, the PointNet models accomplished better recall. Both methods were outperformed by the image-based U-Net network, which achieved the best results. To better understand them, the point-wise classifications are illustrated for the three best models in Figure 18 on a scene from the testing dataset, whose ground-truth labels are demonstrated in Figure 16 (subsection 5.1). It should be noted that the misclassification of the distant points from the robot by the U-Net is probably due to the limited scene size that the network can process.

Model	Accuracy (%)	Precision (%)	Recall (%)
PN (no-scaling)	95.30	87.94	87.99
PN (dynamic-scaling)	93.16	80.90	85.05
PN (static scaling)	95.13	85.84	89.89
U-Net	99.56	98.91	98.84
SVM	95.55	90.13	86.71

Table 1: Results of point-wise class classifications



(a) PN (no scaling) **(b)** U-Net **(c)** SVM

Figure 18: RVIZ visualization of point-wise class classifications on a sample from the testing dataset. The red/green/blue points stand for the measurements captured with the right/left-/middle LIDAR sensor, and the white points feature the classifications of the car class made by a particular method.

6.3.2 Time Efficiency

The computational time needed for each stage of the classification with selected methods is shown in Table 2. The most time-intensive is the U-Net method, to which the application of the U-Net network took on average 50.12 milliseconds. The SVM classification was the fastest, yet its stumbling block is the preprocessing that consists of clustering and computing the feature vectors. On the other hand, the preprocessing speeds up the wheel extraction stage since it applies the clustering. The PointNet methods, which gave comparable results that differ mainly in the preprocessing stage, achieved the best time efficiency.

Model	Preprocessing (ms)	Classification (ms)	Wheel extraction (ms)	Total (ms)
PN (no-scaling)	4.53	4.33	3.51	12.37
PN (dynamic-scaling)	4.91	4.58	4.01	13.50
PN (static scaling)	4.82	4.40	3.99	13.21
U-Net	1.45	50.12	3.35	54.92
SVM	16.06	1.13	2.39	19.58

Table 2: Time taken for each stage of the car detection process

6.3.3 Wheels Localization Ability

Table 3 demonstrates the proportion of correctly found wheels as the ratio between a number of scenes in which the method was able to localize at least two wheels of a car and a number of all scenes in the tested dataset (see subsection 6.2.2). Furthermore, it shows in percentage how many times the method detected a particular number of wheels.

As the car in the testing dataset had all four wheels fully observable, it is not surprising that the wheels fitting method achieved perfect results. The PointNet methods had the worst outcomes, especially the PointNet with dynamic scaling failed in most cases, which was partly expected since the wheel extraction directly depends on point-wise classifications. In oppose to that, promising results were accomplished with the SVM and the U-Net method, which achieved 100 percent success.

Method	Wheels localized (%)	4 wheels (%)	3 wheels (%)	2 wheels (%)
PN (no-scaling)	70.54	97.47	0.00	2.53
PN (dynamic-scaling)	8.93	80.00	10.00	10.00
PN (static scaling)	68.75	97.40	0.00	2.60
U-Net	100.00	100.00	0.00	0.00
SVM	97.32	99.08	0.00	0.92
Wheels fitting	100.00	100.00	0.00	0.00

Table 3: The proportion of correctly found wheel positions

7 Conclusion

This thesis aimed to research and analyze different approaches to car detection in 2D range data capable of performing well in real-world conditions and adverse weather. I implemented and evaluated the performance of three various machine learning methods: the PointNet for detecting the cars from point clouds, the U-Net for detection in images, and the SVM from feature vectors and one geometrically based wheels fitting method for wheels localizations.

Experimental results showed the necessity of choosing a compromise between performance and time efficiency. The U-Net neural network accomplished the best results both in point-wise classifications and wheel localizations, and thus, it would be the primary choice. On the other side of the coin, promising results regarding the point-wise classifications were also achieved by the SVM and the PointNet, which are considerably less time-demanding. Yet concerning wheel localization, the SVM would be preferred over the PointNet.

However, experiments with different setups of LIDAR sensors on the parking robot indicated that the wheels of cars might not be observable. I consider this the weakness of the proposed wheel extraction method, which would not estimate wheel positions despite the encouraging results of car detection methods.

Future works could address this problem by reformulating the whole task of vehicle detection on the instance segmentation task, which would facilitate wheel estimations. Furthermore, I see potential in the bounding box prediction networks. For instance, fixed-size rectangle proposals for vehicle detection were proved to be suitable in many articles in the context of both CNNs (such as YOLOv3) and PointNet.

References

[1] serva GmbH. The ray parking robot by serva transport systems. Meet Ray - serva. Available at: `https://serva-ts.com/en/parking/meet-ray/`, accessed 2021-20-05.

[2] Stanley Robotics. The stan parking robot by stanley robotics. Technology. Available at: `https://stanley-robotics.com/tech-new/`, accessed 2021-20-05.

[3] F. Majer, Z. Yan, G. Broughton, Y. Ruichek, and T. Krajník. Learning to see through haze: Radar-based human detection for adverse weather conditions. In *2019 European Conference on Mobile Robots (ECMR)*, pages 1–7, 2019.

[4] J. Schlichenmaier, N. Selvaraj, M. Stolz, and C. Waldschmidt. Template matching for radar-based orientation and position estimation in automotive scenarios. In *2017 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*, pages 95–98, 2017.

[5] Anna Petrovskaya and Sebastian Thrun. Model based vehicle detection and tracking for autonomous urban driving. *Autonomous Robots*, 26(2-3):123–139, April 2009.

[6] X. Shen, S. Pendleton, and M. H. Ang. Efficient l-shape fitting of laser scanner data for vehicle pose estimation. In *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pages 173–178, 2015.

[7] X. Zhang, W. Xu, C. Dong, and J. M. Dolan. Efficient l-shape fitting for vehicle detection using laser scanners. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 54–59, 2017.

[8] Sanqing Qu, Guang Chen, Canbo Ye, Fan Lu, Fa Wang, Zhongcong Xu, and Yixin Ge. An efficient l-shape fitting method for vehicle pose detection with 2d lidar, 2018.

[9] Eugenio Aguirre, Miguel Garcia-Silvente, and Javier Plata. Leg detection and tracking for a mobile robot and based on a laser device, supervised learning and particle filtering. *ROBOT2013: First Iberian Robotics Conference*, 252:433–440, January 2014.

[10] Heng Wang and Xiaodong Zhang. Real-time vehicle detection and tracking using 3d lidar. *Asian Journal of Control*, March 2021.

[11] Luis Navarro-Serment, Christoph Mertz, and Martial Hebert. Pedestrian detection and tracking using three-dimensional ladar data. *I. J. Robotic Res.*, 29:1516–1528, October 2010.

[12] J. Cheng, Z. Xiang, T. Cao, and J. Liu. Robust vehicle detection using 3d lidar under complex urban environment. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 691–696, 2014.

[13] K. Kidono, T. Miyasaka, A. Watanabe, T. Naito, and J. Miura. Pedestrian recognition using high-definition lidar. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 405–410, 2011.

[14] Z. Lin, M. Hashimoto, K. Takigawa, and K. Takahashi. Vehicle and pedestrian recognition using multilayer lidar based on support vector machine. In *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 1–6, 2018.

[15] Takuma Oiwane, Priscilla Osa, and Shuichi Enokida. Research on feature descriptors for vehicle detection by lidar. August 2019.

[16] Kai Arras, Oscar Mozos, and Wolfram Burgard. Using boosted features for the detection of people in 2d range data. pages 3402–3407, April 2007.

[17] Pierre Merdrignac, Evangeline Pollard, and Fawzi Nashashibi. 2d laser based road obstacle classification for road safety improvement. In *2015 IEEE International Workshop on Advanced Robotics and its Social Impacts (ARSO 2015)*, Lyon, France, July 2015.

[18] C. Premebida, O. Ludwig, and U. Nunes. Exploiting lidar-based features on pedestrian detection in urban scenarios. In *2009 12th International IEEE Conference on Intelligent Transportation Systems*, pages 1–6, 2009.

[19] Q. Tang, L. Kurnianggoro, and K. Jo. Statistical and geometrical features for lidar-based vehicle detection. In *2016 IEEE/SICE International Symposium on System Integration (SII)*, pages 192–197, 2016.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[21] D. Wang and I. Posner. Voting for voting in online point cloud object detection. In *Robotics: Science and Systems*, 2015.

[22] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks, 2017.

[23] Bo Li. 3d fully convolutional network for vehicle detection in point cloud, 2017.

[24] D. Maturana and S. Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, 2015.

[25] Bin Yang, Wenjie Luo, and Raquel Urtasun. Pixor: Real-time 3d object detection from point clouds, 2019.

[26] Bin Yang, Ming Liang, and Raquel Urtasun. Hdnet: Exploiting hd maps for 3d object detection, 2020.

[27] Sascha Wirges, Tom Fischer, Jesus Balado Frias, and Christoph Stiller. Object detection and classification in occupancy grid maps using deep convolutional networks, 2018.

[28] Bo Li, Tianlei Zhang, and Tian Xia. Vehicle detection from 3d lidar using fully convolutional network, 2016.

[29] Limin Guan, Yi Chen, Guiping Wang, and Xu Lei. Real-time vehicle detection framework based on the fusion of lidar and camera. *Electronics*, 9(3), 2020.

[30] X. Du, M. H. Ang, and D. Rus. Car detection for autonomous vehicle: Lidar and vision fusion approach through deep learning framework. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 749–754, 2017.

[31] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving, 2017.

[32] Lucas Beyer, Alexander Hermans, and Bastian Leibe. Drow: Real-time deep learning based wheelchair detection in 2d range data, 2016.

[33] Lucas Beyer, Alexander Hermans, Timm Linder, Kai O. Arras, and Bastian Leibe. Deep person detection in 2d range data, 2018.

[34] G. Chen, F. Wang, S. Qu, K. Chen, J. Yu, X. Liu, L. Xiong, and A. Knoll. Pseudo-image and sparse points: Vehicle detection with 2d lidar revisited by deep learning-based methods. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–13, 2020.

[35] Peter Ondruska, Julie Dequaire, Dominic Zeng Wang, and Ingmar Posner. End-to-end tracking and semantic segmentation using recurrent neural networks, 2016.

[36] Ángel Manuel Guerrero-Higueras, Claudia Álvarez Aparicio, María Carmen Calvo Olivera, Francisco J. Rodríguez-Lera, Camino Fernández-Llamas, Francisco Martín Rico, and Vicente Matellán. Tracking people in a mobile robot from 2d lidar scans using full convolutional neural networks for security in cluttered environments. *Frontiers in Neurorobotics*, 12:85, 2019.

[37] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[38] M. Dreher, E. Erçelik, T. Bänziger, and A. Knol. Radar-based 2d car detection using deep neural networks. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8, 2020.

[39] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017.

[40] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space, 2017.

[41] Y. Zhou and O. Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

[42] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds, 2019.

[43] Charles R. Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J. Guibas. Frustum pointnets for 3d object detection from rgb-d data, 2018.

[44] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud, 2019.

[45] A. Danzer, T. Griebel, M. Bach, and K. Dietmayer. 2d car detection in radar data with pointnets. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 61–66, 2019.

[46] Christophe Reymann and Simon Lacroix. Improving LiDAR Point Cloud Classification using Intensities and Multiple Echoes. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015)*, page 7p., Hamburg, Germany, September 2015.

[47] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[48] Chire. Dbscan illustration. Wikimedia Commons, October 2011. Available at: `https://commons.wikimedia.org/w/index.php?curid=17045963`, accessed 2021-20-05.

[49] Caroline Clabaugh, Dave Myszewski, and Jimmy Pang. Neural networks. Stanford university. Available at: `https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/index.html`, accessed 2021-20-05.

[50] Erin LeDell. Mlp network. Github, September 2017. Available at: `https://github.com/ledell/sldm4-h2o`, accessed 2021-20-05.

[51] An illustration how discrete convolution on a pixel works. River Trail documentation. Available at: `http://intellabs.github.io/RiverTrail/tutorial/`, accessed 2021-20-05.

[52] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[53] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, January 2009.

[54] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[55] Dave Hershberger, David Gossow, and Josh Faust. Rviz, 3d visualization tool for ros. 2019. Available at: `http://wiki.ros.org/rviz`, accessed 2021-20-05.

[56] Fei Xia. Pytorch implementation for pointnet. Github. Available at: `https://github.com/fxia22/pointnet.pytorch`, accessed 2021-20-05.

Appendix

CD Content

Directory name	Description
text	bachelor thesis in pdf
sources	source codes
datasets	annotated datasets used for training and evaluation

Table 4: CD Content