UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science curriculum

Timo Tiirats

Object Recognition Using a Sparse 3D Camera Point Cloud

Master's Thesis (30 ECTS)

Supervisors:
Tambet Matiisen, MSc
Jan Bogdanov, MSc

TARTU

May 2023

UNIVERSITY OF TARTU

Abstract

Faculty of Science and Technology
Institute of Computer Science

Master of Science in Engineering

Object Recognition Using a Sparse 3D Camera Point Cloud

by Timo Tiirats

The demand for higher precision and speed of computer vision models is increasing in autonomous driving, robotics, smart city and numerous other applications. In that context, machine learning is gaining increasing attention as it enables a more comprehensive understanding of the environment. More reliable and accurate imaging sensors are needed to maximise the performance of machine learning models. One example of a new sensor is LightCode Photonics' 3D camera.

The thesis presents a study to evaluate the performance of machine learning-based object recognition in an urban environment using a relatively low spatial resolution 3D camera. As the angular resolution of the camera is smaller than in commonly used 3D imaging sensors, using the camera output with already published object recognition models makes the thesis unique and valuable for the company, providing feedback for LightCode Photonics' current camera specifications for machine learning tasks. Furthermore, the knowledge and materials could be used to develop the company's object recognition pipeline.

During the thesis, a new dataset is generated in CARLA Simulator and annotated, representing the 3D camera in a smart city application. Changes to CARLA Simulator source code were implemented to represent the actual camera closely. The thesis is finished with experiments where PointNet semantic segmentation and PointPillars object detection models are applied to the generated dataset. The generated dataset contained 4599 frames, of which 2816 were decided to use in this thesis. PointNet model applied to the dataset could predict the semantically segmented scene with similar accuracy as in the original paper. A mean accuracy of 44.15% was achieved with PointNet model. On the other hand, PointPillars model was unable to perform on the new dataset.

CERCS: T111 Imaging, image processing; T120 Systems engineering, computer technology; T181 Remote sensing; P176 Artificial intelligence

Keywords: 3D imaging; 3D sensors; object recognition; machine learning; CARLA Simulator; PointNet; PointPillars

TARTU ÜLIKOOL

Resümee

Loodus- ja täppisteaduste valdkond
Arvutiteaduse instituut

Tehnikateaduse magister

Objektituvastus kasutades hajusat 3D-kaamera punktipilve

Timo Tiirats

Autonoomsete sõidukite, robootika, targa linna ja paljude sarnaste rakenduste areng suurendab nõudlust täpsemate ja kiiremate raalnägemismudelite järele. Mainitud rakenduste puhul pälvib masinõpe üha enam tähelepanu, võimaldades ümbritsevat keskkonda terviklikumalt mõista. Masinõppemudelite efektiivsuse maksimeerimise eelduseks on töökindlate ja täpsete sensorite olemasolu, mille näitena võib tuua ettevõtte LightCode Photonics'i 3D-kaamera.

Magistritöös hinnatakse masinõppepõhise objektituvastuse võimekust linnakeskkonnas kasutades suhteliselt madala ruumilise eraldusvõimega 3D-kaamera andmeid. Töös käsitletud kaamera nurklahutusvõime on väiksem kui paljudel teistel laialdaselt kasutatavatel 3D-sensoritel, mistõttu on varem välja töötatud objektituvastusmudelite rakendamine kaamera andmetel unikaalne ja ettevõttele väärtuslik. Magistritöö põhjal on võimalik järeldada, kui hästi sobib LightCode Photonics'i praeguste spetsifikatsioonidega kaamera masinõppel põhinevatele rakendustele. Lisaks võimaldavad saadud kogemus ning koostatud materjalid edasi arendada ettevõtte objektituvastuse algoritme.

Lõputöö käigus genereeriti CARLA simulaatoris ja annoteeriti uus andmestik, mis kujutab 3D-kaamerat targa linna kasutusjuhul. CARLA simulaatori lähtekoodi muudeti, et tagada võimalikult suur ühtivus simuleeritud ja tegeliku kaamera vahel. Magistritöö viimane osa sisaldab eksperimentide analüüsi genereeritud andmestikule rakendatud PointNet'i semantilise segmenteerimise ja PointPillars'i objektituvastuse mudelite kohta. Loodud andmestik sisaldas 4599 kaadrit, millest 2816 otsustati käesolevas lõputöös kasutada. Andmestikule rakendatud PointNet'i mudeli ennustustäpsus semantiliselt segmenteeritud stseeni puhul sarnanes teiste autorite varasematele tulemustele. PointNet'i mudeliga saavutati keskmine täpsus 44.15%. Seevastu, PointPillars'i mudel uue andmestiku puhul oodatult ei töötanud.

CERCS: T111 Pilditehnika; T120 Süsteemitehnoloogia, arvutitehnoloogia; T181 Kaugseire; P176 Tehisintellekt

Märksõnad: 3D-kuva; 3D-sensorid; objektituvastus; masinõpe; Carla Simulator; PointNet; PointPillars

Acknowledgements

Firstly, I would like to express my deepest gratitude to my thesis supervisor Tambet Matiisen for the support and guidelines throughout the process. In addition, I thank my co-supervisor Jan Bogdanov for always finding the time to discuss thesis-related topics and providing feedback to go forward.

Secondly, I would like to thank LightCode Photonics team for their continuous support and understanding of the thesis's importance to me. Thank you, Thamasha Rasangi, for helping to annotate parts of the generated data.

Finally, I thank my family for supporting me throughout my studies and always motivating me to give my best.

Contents

Abbreviations

2D	Two-**D**imensional
3D	Three-**D**imensional
FPA	**F**ocal-**P**lain **A**rray
LiDAR	**Li**ght **D**etection **A**nd **R**anging
ML	**M**achine **L**earning
CGI	**C**omputational **G**host **I**maging
FoV	**F**ield-**o**f-**V**iew
ToF	**T**ime-**o**f-**F**light
iToF	**i**ndirect **T**ime-**o**f-**F**light
dToF	**d**irect **T**ime-**o**f-**F**light
AMCW	**A**mplitude-**M**odulated **C**ontinuous **W**ave
MEMS	**M**icro-**E**lectro**M**echanical **S**ystem
CMOS	**C**omplementary **M**etal-**O**xide **S**emiconductor
CNN	**C**onvolutional **N**eural **N**etwork
BEV	**B**irds-**E**ye-**V**iew
FPS	**F**rames **P**er **S**econd
API	**A**pplication **P**rogramming **I**nterface
TP	**T**rue **P**ositive
TN	**T**rue **N**egative
FP	**F**alse **P**ositive
FN	**F**alse **N**egative
IoU	**I**ntersection **o**ver **U**nion
mAP	**m**ean **A**verage **P**recision

Introduction

Machine learning-based computer vision solutions for object recognition have been gaining more and more attention over the last decade, enabling an increasingly comprehensive understanding of the observed environment together with precise localisation and tracking. It is possible to find computer vision tasks in numerous areas, including autonomous driving, robotics, visual surveillance and smart city [1]. Object recognition can be based on 2D data, for example, colour or grayscale images and 3D images containing distance data. While 2D-based machine learning (ML) models have been studied thoroughly, 3D-based models are still evolving and improving as more 3D data is becoming available through public datasets [2, 3], and each 3D sensor has unique characteristics. On the other hand, 3D data could improve the precision and reliability of ML models, especially in autonomous vehicles, to plan their movements or interactions and avoidance of objects to maximise efficiency and improve safety. In addition, reduce the number of edge case scenarios, severely deteriorating the performance of solely 2D-based ML models.

Therefore, 3D cameras and light detection and ranging (LiDAR) sensors are becoming increasingly important for environment perception [3], as they give a more accurate three-dimensional representation of an object or a scene, allowing for improved performance in diverse environments. As the system's, for example, a computer vision-based robot's, surrounding environments can be dynamic and have changes in illumination and weather conditions, then the efficiency of the system is highly dependable on the accuracy it can comprehend its surroundings [1]. Hence, many 3D vision companies, like Intel, Ouster, and Velodyne, constantly try to improve existing sensors and invent new technologies with competitive pricing, allowing to enhance the computer vision-based algorithms as more 3D data is available.

In this thesis, we use the existing object recognition ML models to conduct a performance analysis of a 3D camera based on a novel technology.

Aim of the Thesis

The main goal of this thesis is to evaluate the performance of machine learning-based object detection and semantic segmentation using a relatively low spatial resolution 3D camera in an urban environment. The LightCode Photonics camera is based on the novel design of computational ghost imaging (CGI). While the angular resolution of the 3D camera, and therefore its resolving power, i.e. its ability to reproduce object detail, is lower than in many other 3D imaging devices, the hypothesis is that using the camera point cloud for object recognition will yield comparable accuracy achieved in the original papers. Two previously published object recognition models are applied to a new dataset, which is created and annotated to validate the hypothesis.

The work is valuable for LightCode Photonics as it can be used for assessing the current camera capabilities and specifications for machine learning tasks. Additionally, a base algorithm and knowledge are created for future work to fuse 2D RGB and 3D point clouds or multiple 3D point clouds to improve the performance of object recognition algorithms. Last but not least, the resulting algorithm can be used as an input to LightCode Photonics customers to decrease the time-to-market.

Overview of the Thesis

The thesis is divided into three main chapters. The first chapter describes the thesis's theoretical aspects, giving background information related to 3D imaging and machine learning. Firstly, a comparative analysis of 3D imaging technologies is presented. Followed by machine learning-based object detection and semantic segmentation theoretical overview, including data representation format descriptions. Finally, an overview of the LightCode Photonics 3D camera is given.

The second chapter described the methodology used in the thesis. Firstly, an overview of the system setup on which the ML models were trained and tested is given. Furthermore, the data acquisition and preprocessing steps, including data annotating, are described. The second chapter is concluded with an overview of chosen object detection and semantic segmentation algorithms with an explanation of changes made to the original code.

In the last chapter, the results of object recognition models are evaluated. Firstly, an overview of the generated dataset is given. Secondly, the performance of ML models is described, including the accuracy and computational complexity metrics. The thesis closes with an outlook and a conclusion.

1 Background

In this chapter, the theoretical background of this thesis is given on previously published materials incorporated with thesis author assessments. Firstly, an overview of different 3D imaging technologies, including depth and lateral resolution finding categorisation, is given. In addition, LightCode Photonics 3D camera parameters and capabilities are specified. Secondly, the theoretical aspects of machine learning are discussed, focusing on the object recognition task on the 3D sensor point cloud. The section also introduces 3D imaging output data representation formats to support understanding each format's characteristics.

1.1 Overview of 3D imaging Technologies

3D imaging enables extracting semantics in many applications, including consumer, automotive, and industrial fields, for example, robotics, self-driving cars, AR/VR and security [4–6]. 3D imaging enhances the understanding of the world compared to 2D imaging as depth information is added. For example, a painted ball on a wall and a physical ball look the same in a 2D image [7].

3D-imaging technologies could be divided into categories based on multiple criteria. The first option is to classify them based on distance or range-finding technology. The second method divides them based on the method of obtaining lateral resolution. In the following sections, an overview of different distance-finding methods and will be followed by lateral resolution categorisation.

1.1.1 Distance-Finding Methods

There are multiple ways of acquiring depth information from the scene. The most common methods currently used are stereo vision, structured light, and time-of-flight (ToF) technologies [2, 4]. Both stereo vision and structured light use triangulation to estimate distances in the scene therefore, it can be computationally relatively expensive, and the accuracy depends on the imaging resolution and the distance between the bistatic imaging cameras. [2, 8]. On the other hand, time-of-flight technology measures the time it takes for the light to travel to the object and back to the camera [2]. Time-of-flight can be measured directly or indirectly, and the following two paragraphs give an overview of both techniques.

Direct time-of-flight (dToF) sensors typically use a sub-nanosecond electronic stopwatch, a pulsed light emission and measure the time it takes for the light to make a round-trip to the object and back to the sensors [4, 5, 9]. Using that time, the distance to the object can be found with the following formula:

$$D = \frac{\tau_R c}{2}, \tag{1.1}$$

where D is the object distance, τ_R is the measured flight time and c is the speed of flight [9]. In some cases, the back-scattered pulse is converted to an analog signal, for example, photocurrent [7, 8], which is then amplified, compared and linked to the flight time [5, 10]. More complex dToF sensors acquire a photon timing histogram for each pixel, showing how many photons the receiver collected in time, also known as time-correlated single-photon counting (TCSPC) [5, 6, 8]. Therefore using the dToF sensor for distance measuring enables simple discrimination of multipath echoes [5], which means that it is possible to measure the distance to multiple objects in the same pixel (Figure 1.1). At the same time, it allows seeing through (semi)transparent objects, for example, plexiglass. dToF is suitable for applications requiring long ranges, possibly up to kilometres, even while exposed to ambient light [4, 8].

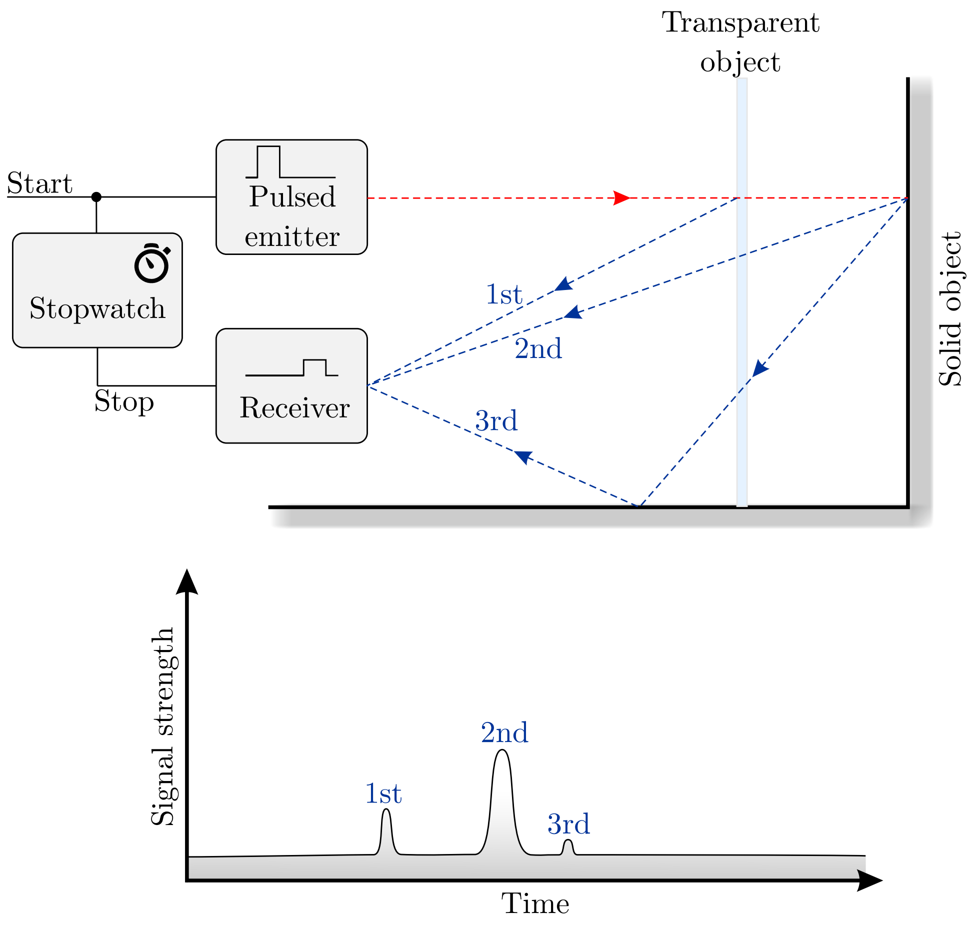

Figure 1.1: dToF sensor emits a short light pulse, synchronised to an electronic stopwatch. Emitted light pulse scatters on different objects on the scene and reflects back to the receiver side of the sensor. While doing so, it can take multiple different paths. A dToF receiver can differentiate return pulses from many different objects in its field-of-view (FoV) as long as the temporal resolution of the system is sufficient and an adequate amount of light is reflected from objects [5]. For simplification, the second reflection from the transparent object is not shown.

Indirect time-of-flight (iToF) sensors typically use amplitude or frequency-modulated continuous waves to measure the distance to objects [8, 11]. Of 3D imaging devices operating at optical frequencies, the currently most used iToF 3D sensors are amplitude-modulated continuous wave (AMCW) cameras [7]. The optical waves are periodic and have well-controlled fundamental frequencies, but the shape of the waveform may vary [7, 12]. To calculate distance, the AMCW sensor uses a homodyne photomodulator pixel structure to measure the phase difference between emitted and received optical signals (Figure 1.2) [5, 8, 9]. iToF sensors can operate at relatively fast frame rates with relatively high resolution while still having modest power consumption [5]. iToF technology's working range is usually up to tens of meters, limited due to the modulation frequency [8]. In addition, it cannot separate multiple reflections in the same pixel FoV as the multipath superimposes optical waves on each other [5, 7]. Therefore, there is a higher probability of erroneous distance measurements occurring, and in certain systems, an average distance value of these two reflections will be measured [13].

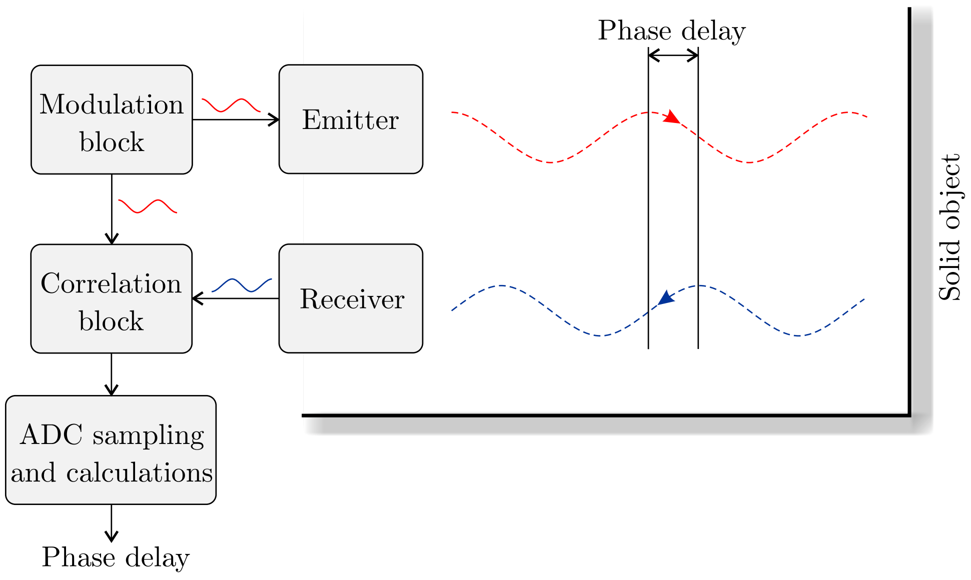

FIGURE 1.2: In an AMCW iToF measurement device amplitude modulated waveform is created. The optical signal is passed to the emitter of the device and to the correlation block as a local oscillator for the heterodyne demodulation of the return signal. The back-scattered signal is collected by the receiver and mixed with the local oscillator. The phase difference is outputted, which can be used for distance calculations [8, 12].

The phase delay is calculated using the following formula:

$$\Delta\Phi = arctan(\frac{Q_3 - Q_4}{Q_1 - Q_2}), \tag{1.2}$$

where $Q_1 - Q_4$ is an analog-to-digital converter (ADC) sampled received signal intensities in four equally spaced points (Figure 1.3) [8, 9].

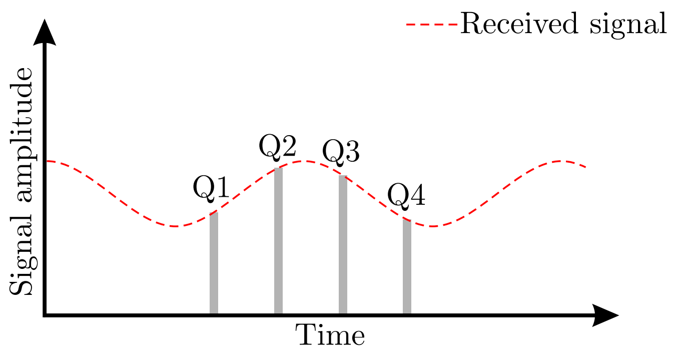

FIGURE 1.3: To determine the phase delay in iToF cameras, four equally spaced points are probed on the received signal by ADC continuously [8, 9, 12].

Using the calculated phase delay $\Delta\Phi$, the distance in the iToF AMCW sensor can be found in the following way:

$$D = \frac{c}{4\pi f_{laser}} * \Delta\Phi, \tag{1.3}$$

where c is the speed of light and f_{laser} is the frequecy of the laser [8].

1.1.2 Lateral resolution

Lateral resolution is defined as the imaging resolution perpendicular to the laser beam. Multiple technologies are available, including beam steering and focal-plane array (FPA) based imaging, which can be an active LiDAR sensor or rely on passive stereo vision. All mentioned technologies have advantages and disadvantages, and their characteristics will be discussed from the machine learning point of view. Subsequently, an overview of three different working principles is given.

Beam Steering Imaging

Beam steering-based 3D imagers are popular as they provide high-precision range information and usually have electro-optical architectures that achieve good signal-to-noise ratios enabling long-range operation [8, 11, 14]. In the case of beam steering, a single or relatively small number of detectors and laser emitters are usually used [14]. To enlarge the total FoV of the system comprising of singular or a small number of narrow FoV optical emitter-detector pairs, a mechanical system or optical beam steering is used [4, 8, 11]. Therefore all beam steering systems behave similarly to rolling shutter cameras, as they do not acquire data from their observable FoV simultaneously. The beam steering mechanisms can be divided into four categories: Opto-mechanical, electromechanical, micro-electromechanical (MEMS), and solid-state scanning systems. The first three mentioned technologies will be discussed, as the solid-state scanning methods are not mature enough to be used in real-world use cases [15]. All considered principles use the dToF method for measuring object distances in the scene [4].

Opto-mechanical systems use mirrors and prisms to change the direction of the emitted and back-scattered beam (Figure 1.4). For that, mostly galvanometer and gyroscopic mirrors or Risley prims are used. Alternating the optical beam angle using this method allows the system to be more lightweight at the cost of limited FoV. Decreasing the weight of moving parts in the scanner allows for reduced vibrations and lowers the torque specifications for the electrical motor moving the mirrors. Opto-mechanical systems could provide 2D and 3D information about the scene, but the difference is that for 2D, only one mirror is needed, and for 3D, two mirrors or prism are required [15].

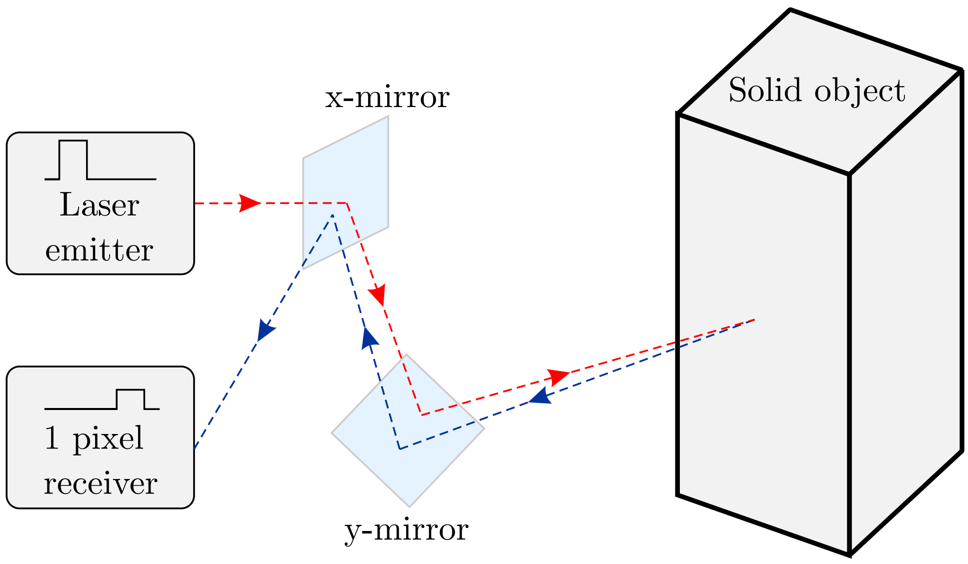

Figure 1.4: Opto-mechanically driven 3D imaging sensor that directs one laser beam towards the object in the scene using two orthogonally mounted mirrors. One mirror controls the elevation, and the second the azimuthal angle of the directed beam. One pixel-sized receiver is used to capture the ToF of the back-scattered light [8, 15].

Electromechanical systems are rather similar to opto-mechanical devices, but they use an electric motor to rotate the whole optical configuration, including linear detector arrays, around the mechanical axis [11, 15]. Therefore they could add one additional dimension to the original sensor, for example, enhancing 2D sensors to provide 3D data [15]. Their one major advantage is that they can perform 3D 360-degree scans of the environment on the azimuthal axis [11, 15]. In addition, they produce straight and parallel scan lines with a uniform scanning speed over the whole FoV, which makes them one of the most popular commercially available sensors available [11]. On the other hand, to achieve 360-degree FoV coverage, a complex design must be used as the data and power cables cannot be directly connected to the emitter-detector assembly [15], thus increasing the price of the device.

Micro-electromechanical scanning systems use small MEMS mirrors to control the direction of the laser beam. The mirrors are actuated by applying a stimulus, for example, voltage. They can be controlled in both azimuth and elevation axis, allowing for a whole scene scan with one mirror [11]. The size of the mirrors is only a few mm, for example, 4 mm x 4.5 mm x 1.6 mm and weighs 16 mg [11, 16]. Using smaller and fewer mechanical components makes the system compact and lightweight, which is favourable for many autonomous and robotics-related applications. [11, 15]. In addition, due to smaller mirrors, the resonance frequency is much larger than possible vibrations occurring in autonomous vehicles or robots [11]. On the contrary, MEMS-based sensors have relatively limited FoV due to not having rotating parts and might not be suitable for all 3D imaging tasks. Moreover, using a dual-axis mirror could potentially introduce unwanted crosstalk between the two-axis control signals, decreasing the control stability of the mirror [11, 15].

In conclusion, all the discussed systems in this section use direct time-of-flight technology, therefore, can separate multipath returns, which otherwise could introduce false positive distance measurements in the results. At the same time, they are rolling shutter imaging devices, which means that moving the sensor or targets in the scene could create the same object appearing in multiple places. As abnormal shapes could form, false objects or their distances can negatively impact machine learning algorithms. Electromechanical sensor result differs from opto-mechanical and micro-electromechanical sensors data as it performs uniform acquisition over the scene, while others scan back and forth through the scene. In the case of beam steering, the acquisition speed and framerate are negatively impacted by the desired angular resolution [8]. Lastly, electromechanical do not allow changing scanning patterns, while opto-mechanical and MEMS sensors allow for changeable patterns [11].

Focal-Plane Array Based Imaging

Focal-plane array sensor uses a linear array or a matrix of detectors to capture spatial information from the whole scene from a single or a series of acquisition periods without scanning [8, 11, 14]. Scanning is unnecessary, as each pixel in the detector is responsible for a certain subsection in the scene [8, 11] and the angular resolution is tied to the number of pixels and FoV of the sensor [8]. FPA-based imaging sensors are fully solid-state and could be used with a global shutter detector array. Usually, a solid-state operation is considered desirable as it reduces the sensor's size, weight and power consumption. Also, it requires less stabilisation as no moving part could be affected by resonance frequency [14]. The illumination schemes used with FPA arrays could be pulsed or continuous [11], depending on which distance measurement methods are used. In the context of this work, the most important ones are flash dToF and AMCW iToF sensors.

Flash dToF (Figure 1.5) sensor uses relatively short laser pulses to flood illuminate the scene [11]. This allows for very high framerates, but higher peak-illumination laser power is needed to gather enough back-scattered light from the scene. At the same time, it is constrained by eye-safety limits [8, 14]. Overall, the depth and angular resolution of flash dToF sensors are comparable to beam steering sensors. However, adding more pixels to the flash dToF sensor is complicated, as more sensitive detectors are needed compared to single-pixel scanning, thus increasing the size and cost of the focal-plane array [11, 14].

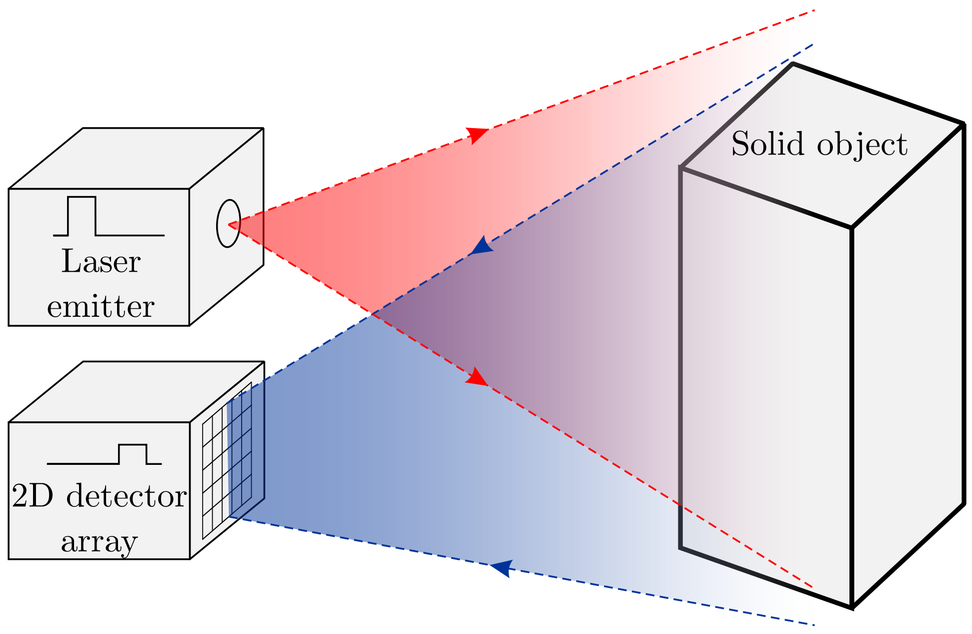

FIGURE 1.5: In the FPA flash dToF sensor, the FoV of the detector array is closely matched with the illumination area, usually using an optical system to shape the light beam and detector's FoV suitably. This allows to flood illuminate the whole scene with uniform pulsed lighting. Then, the light pulse is back-scattered from objects and collected by the 2D detector array, where each pixel collects the returning light. The distance to objects is calculated directly using the dToF method [11].

AMCW sensor continuously emits modulated light and measures the distances in the scene using the iToF method. The iToF technology-based sensors are mainly for short-distance measurements, because of the periodicity of the modulated light [11]. Compared to dToF flash sensors, the detectors used in AMCW sensors are based on well-known and frequent complementary metal-oxide semiconductor (CMOS) technology, allowing the detector to be small and low-cost. Each pixel or group of pixels in the AMCW detector incorporates its own phase delay measuring electronics, meaning there is a manufacturing limitation on the final number of pixels in the detector, similar to the dToF flash sensor [7, 11].

Overall, FPA sensors use both iToF and dToF distance measuring methods. iToF technology is susceptible to erroneous distance measurements due to the inability to distinguish multipath returns. At the same time, dToF has the potential to detect multipath returns and possibly could enhance the data and machine learning algorithms as more data is available. Compared to the beam steering approach, FPA-based imaging could be done using global shutter sensors, which allows for capturing the full scene with a single shot. This possibly allows for fewer errors in the final image due to moving targets or cameras. On the other hand, beam steering approaches apply spatial filtering intrinsically, thus improving the signal-to-noise ratio, allowing improved range with fewer captured frames [8].

Stereo vision

Stereo vision exploits the disparity between two monocular cameras for object distance calculations, similar to human vision (Figure 1.6) [12, 17–19]. Stereo vision usually uses standard CMOS imaging detectors, which makes the quality of the disparity map depend on image noise, lighting conditions and other uncertainties [18, 20]. In addition, low-textured materials harm the system's stability [17], and the disparity calculations are computationally expensive [8]. Therefore stereo vision can often achieve only a limited frame rate with an operating distance of a few meters [8]. On the positive side, stereo imaging can achieve high resolution [8] and denser data than compared to LiDAR sensors [17].

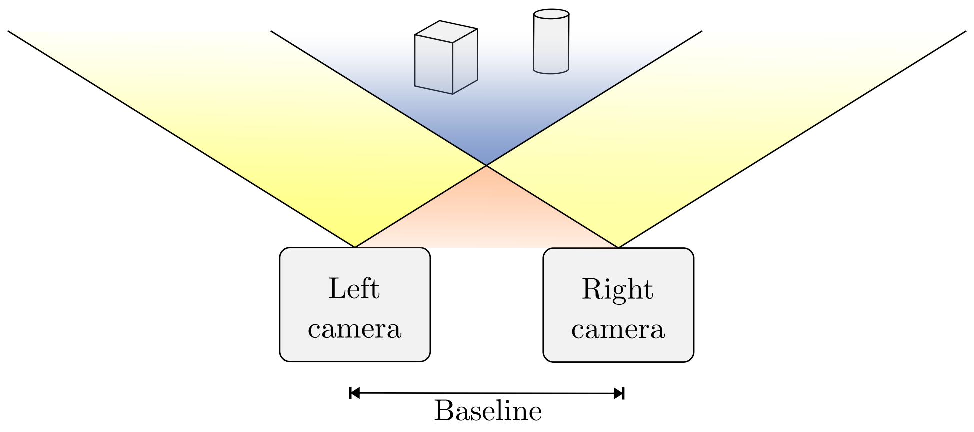

FIGURE 1.6: Stereo vision needs two cameras in a bistatic configuration that must be horizontally well aligned, and the baseline between the two cameras must be fixed [18]. Changing the baseline between cameras increases or decreases the sensor distance calculation range [8]. Stereo vision has multiple regions in its FoV. Firstly, the blue background shows where both cameras' FoVs match and triangulation can be done to find the object distance. Objects in the yellow area are present in different camera images, and disparity mapping is impossible. Lastly, the orange area shows the blind area in front of the sensor, where neither of the cameras sees the object [21].

Firstly, machine learning algorithms could benefit from stereo vision's denser data compared to previously discussed sensors as more data points are available due to the intrinsically higher native resolution of the two cameras. Secondly, as stereo imaging uses ordinary CMOS imaging detectors to create 3D data, it will provide 2D images simultaneously with 3D data. On the contrary, the dependency on ambient lighting makes the stereo camera output unstable under changing lighting conditions, which could affect ML model results.

1.1.3 LightCode Photonics Camera

For the practical part of the thesis, LightCode Photonics camera was chosen as the imaging sensor because the author worked as a Hardware Application Engineer in the company. In addition, the company was interested in the capability study of the camera related to machine learning applications. The table 1.1 shows the most relevant camera specifications. As the current camera specifications were relatively low compared to commonly used iToF or stereo sensors, testing and comparing the machine learning results were valuable to find the best application and setup for the camera.

Table 1.1: Overview of LightCode Photonics camera specifications

Attribute	**Value**
Field-of-view (°)	80×15
Resolution (px)	96×16
Angular resolution (°/px)	0.83×0.94
Refresh rate (Hz)	5
Range with 10% reflectivity target (m)	10

LightCode Photonics 3D camera uses a low-resolution FPA receiver in the camera's core to detect the back-scattered signal and enhances the lateral resolution using the CGI principle. Currently, the camera works as a near solid state sensor without any macroscopic moving parts, for example, larger than 1 mm pitch MEMS mirrors or prisms, to capture the image. In the near future, the technology has the potential to work as a fully solid-state system. The flash dToF measuring method is used for depth data, which gives the camera many capabilities compared to competing imaging sensors. It can acquire multiple returns from one pixel FoV, which could improve the effective resolution on the object's edges or with transparent objects. In addition, point cloud, depth image and intensity image are provided as output options. In the future, the hardware solution enables defining dynamic resolution with software, allowing to change the lateral resolution of the camera on the fly. The company believes that these features can improve machine learning-based operations.

1.2 Overview of Object Recognition

Object recognition, including object detection and semantic segmentation, is a common computer vision research topic [1, 22]. It is used in various applications, including autonomous driving, robotics, medical diagnosis, fashion, etc., and it aims to extract all the objects of interest in the scene, identify them and determine their locations, therefore improving the understanding of the scene [1, 22–24]. For object detection tasks, the number of objects in the scene is not fixed and, therefore, could vary from zero to many. In addition, the objects might be partially hidden or vary in scale [1]. Object detection using ML algorithms has been developing rapidly in recent years, allowing the detection of objects based on 2D and 3D data. However, 2D object detection methods are much more mature at this time [1, 23]. 2D and 3D object detection methods work on the same principles, but 3D object detection estimates the object positions in the 3-dimensional scene. In contrast, the 2D method estimates the object locations on the image plane. Furthermore, it is possible to use 2D camera images for 3D object detection to estimate the object's exact position in 3D space. Still, as the 2D images lack depth information, the efficiency of these methods is low [1].

Different input data formats could be used for object recognition, and some of those will be discussed in the next section. This is followed by an overview of two object recognition methods: object detection with bounding boxes and semantic segmentation. The third section covers the description of two specific object recognition algorithms. Lastly, semantic segmentation and object detection evaluation metrics are discussed.

1.2.1 Input Data Representations

3D imaging sensors are can output different data in multiple formats. In addition to distance information, some sensors can output intensity images, raw histograms, velocity maps, and other data types. One of the most common formats to represent data is using point clouds [23], where each point marks the contact position of the laser beam and the object surface. The point is described with three coordinates (X, Y, Z) [25], with respect to the sensor [1]. The downside with point clouds is that they are unordered, sparse and irregularly distributed in the world space [2, 23, 26], therefore they cannot be efficiently processed by a convolutional neural network (CNN) machine learning models [1]. Usually, preprocessing is applied to make the data more structured and suitable for machine learning models [1]. Still, these methods may ineluctably ignore information in one dimension, for example, projecting the point cloud to birds-eye-view (BEV) [2, 23]. Overall, the methods could be divided into three categories of how the point cloud data is used in machine learning object detection tasks. Firstly, point-based methods use point cloud 3D representation directly, without applying previous conversions, to learn the geometrical relationships between points in 3D space [1, 23, 26, 27]. Secondly, projection-based methods, which create different point cloud projections, for example, a BEV image, to make the data more structured, which would allow the use of the same concepts and operators as for 2D object detection machine learning models [1, 23, 26]. Lastly, voxel-based methods, sometimes referenced as grid-based methods, divide the point cloud into predefined-size volumetric voxels that include the information about all the points inside the volume to improve the irregularity of raw point cloud data [1, 26]. This means that the precision of voxel-based methods depends on the sampling density of the grid [26].

In addition to the 3D point cloud, LiDAR sensors can capture the reflectivity of the targets in the scene [1, 4, 25], which can be measured with an active pulse or in some cases using ambient light present in the imaging spectrum [4, 28]. The reflectivity, also known as return intensity, is evaluated based on the strength of the returning pulse from the diffusely reflected signal (Figure 1.1) [4]. Intensity values can be added to the point cloud directly as a fourth parameter [25] or used to generate separate 2D intensity images, where each pixel is coloured based on the object's intensity in pixels FoV [4]. In the latter case, the intensity images can be used similarly to other 2D coloured images used for object detection; for example, in You Only Look Once (YOLO) algorithms [4, 25, 29].

It is worth noting that the intensity image's quality depends on the level of contrast in colour and textures between objects and the background [4].

Similarly to intensity, depth could also be represented as a 2D image, where each pixel colour value represents the distance to a point in the 3D scene. This allows it to be used with popular 2D object detection neural networks [1]. On the other hand, the depth images alone may not supply enough features for object detection due to sparse sensor data [25] and the depth resolution depends on the number of bits used for describing each pixel. The same two problems could be present when using intensity images. Also, in the depth image, objects that are side-by-side at the same distance could be merged, lowering the possibility of correct predictions.

More complex dToF sensors can produce raw time histograms (Figure 1.1), which could also be used for object detection. Time histograms for the whole image frame could be presented as a 3D cube with $X*Y*time$ dimensions. It has been tested that in some cases, using histogram data for object detection outperforms object detection using intensity or depth images, even if the images have the same or a higher lateral resolution. The higher performance is due to multipath return discrimination and the time histogram's ability to capture salient details in the scene [4].

Some 3D imaging sensors can measure the object velocity in the pixel FoV relative to the camera, for example, frequency-modulated continuous wave (FMCW) LiDARs. They can measure the Doppler shift generated by object motion [8], based on what the velocity image could be generated. Velocity image could help to improve the object detection performance [30], as it is easier to differentiate between different moving or/and static objects. For example, at longer distances, the shape of pedestrians and telephone poles is similar [24], but having the velocity information supports the understanding that the moving object cannot be a telephone pole.

3D object detection performance could be enhanced with sensor fusion, merging different types of sensor data [1, 24, 26], usually giving up processing speed [1]. One option is to improve the sparsity and relatively low resolution of 3D sensor data, with the 2D RGB imaging sensor, as their data is well structured, relatively high resolution and gives more contextual information. At the same time, RGB sensors lack depth information, which 3D sensors have [24–26].

1.2.2 Object Detection and Semantic Segmentation

Object detection and semantic segmentation are strongly related visual recognition tasks, and all previously discussed machine learning aspects relate to both methods. Object detection is a task to predict a minimum bounding box around every object of interest in the scene with a predicted class. Optionally a confidence factor may be provided for each predicted object. Semantic segmentation is defined as pixel-level object detection, where the class is predicted for each pixel in the image or point in the point cloud, including the background (Figure 1.7) [1, 22, 31, 32]. Semantic segmentation is particularly suitable for robotic or autonomous driving use cases, as it incorporates object detection, shape recognition and classification tasks. It is important to note that semantic segmentation does not distinguish two objects in the scene of the same class, while object detection predicts a bounding box for each object in the scene. Therefore object detection could be useful for object counting and tracking [1, 33].

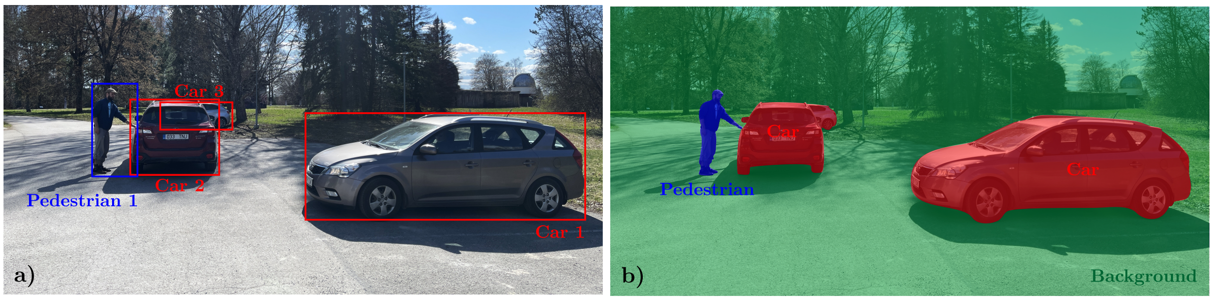

FIGURE 1.7: a) In the case of object detection, a minimum bounding box is predicted for all classified objects. The bounding boxes may overlap, for example, between a pedestrian and a car or between two cars, as shown in the figure. In addition, object detection cannot provide essential attributes about the object, for example, shape. b) Conversely, the predicted semantic segmentation areas cannot overlap as each point in a scene is classified individually [1, 22]. Therefore providing additional information about objects and the scene.

Object classification deep learning models, which are a part of the object detection pipeline, use CNN architecture having only downsampling layers and ending with fully connected layers for predictions [1]. On the other hand, semantic segmentation uses a fully convolutional network (FCN) which usually does not contain any fully connected layers. FCN consists of two sets of layers, downsampling and upsampling, and is used for dense predictions. The input data is encoded into smaller feature maps during downsampling to capture deep contextual information corresponding to the semantics [1, 33]. To save spatial location information during downsampling layers, strides can be used in exchange for max pooling layers [33]. Upsampling decomposes the low-resolution feature maps into high-resolution data, recovering the spatial information and enabling precise localisation. This denotes that the output prediction has the same shape as the input data, allowing every input data point to have a classification output [1, 33].

3D semantic segmentation and object detection deep learning models could use raw point clouds as input, but the label format for training and testing varies. For the semantic segmentation model, a point class must be provided for each point in the point cloud, and the class could be described with one-hot or integer encoding [27, 33]. On the contrary, object detection needs a list of object bounding boxes in the current scene. The bounding box structure commonly includes bounding box class, centre coordinate, size and rotation around the vertical axis, as it is considered that objects in the scene are parallel to the ground [1].

1.2.3 PointNet

PointNet is a pioneer in point-based 3D object classification and semantic segmentation architectures, which can directly consume point clouds [2, 27, 34]. The model supports object classification, part segmentation and scene segmentation tasks (Figure 1.8). Without modifications, the model can consume points described with X, Y and Z coordinates, but additional dimensions could be added, for example, point intensities or normals [27]. PointNet model is invariant to point cloud permutation, i.e. data feeding order and geometric transformations or rotations [27], but it cannot capture the local structure induced by the metric [34]. In addition, PointNet model can resist small corruption, such as outliers or missing data, in the input point cloud and summarise the object's shape based on a sparse set of key points [27].

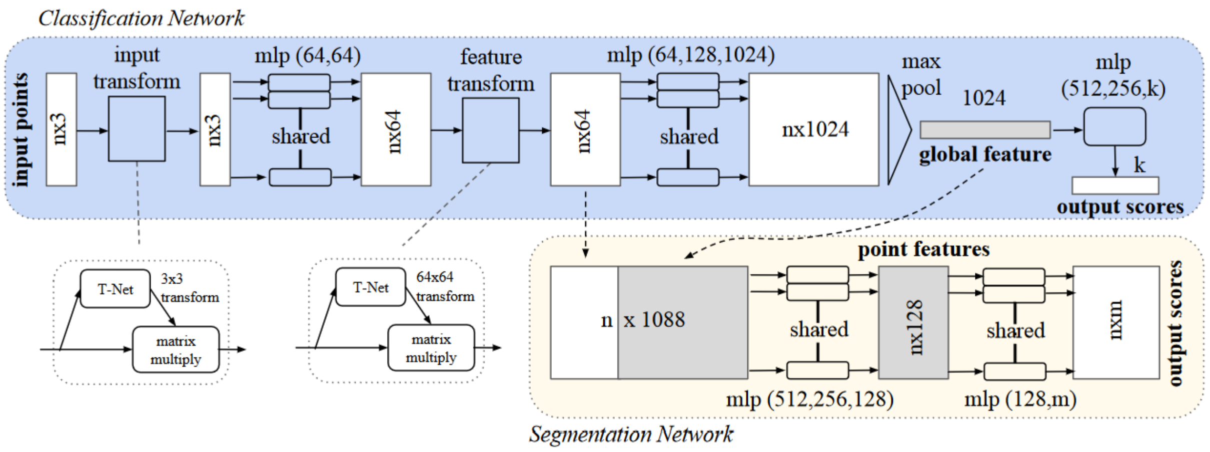

FIGURE 1.8: PointNet model architecture. The classification network is used for object classification tasks, and the intermediate result of the network is also utilised in the upsampling part of the segmentation network. The segmentation network can be used both for part and scene segmentation tasks [27].

In a board term, PointNet architecture (Figure 1.8) learns the spatial encoding of each point and aggregates them with max pooling symmetric function to form a global point cloud signature [27, 34] while saving the reason for selecting each point [27]. PointNet uses T-Nets to capture the affine transformations of the input points. T-Net is a special network which consists of basic modules for feature extraction, max pooling, batch normalisation and fully connected layers [27, 35]. Inside PointNet model, T-Nets are applied two times: first, to make the point cloud invariant to transformations and second, to align the features from different point clouds. It is followed by a max pooling layer aggregating the local features from the T-Net to extract global features. Between T-Nets, multi-layer perception (MLP) layers are used. In the case of semantic segmentations, the network can predict per-point scores based on local and global information, as the local features are concatenated with global features to generate a combined point feature map for each point. For semantic segmentation, the model will output $n \times m$ scores, where n is the number of points and m is the number of the predicted semantic classes. Based on the evaluations conducted by the PointNet paper authors, PointNet can process more than a million points per second and therefore have the potential to work in real-time application [27].

1.2.4 PointPillars

PointPillars is a novel real-time 3D object detection model that uses point clouds as input and generates vertical pillars based on the BEV of the input cloud, which enables to use standard 2D convolutional architectures. The model inputs 4-dimensional point cloud points, including X, Y, and Z coordinates and intensity values. The PointPillars model (Figure 1.9) consists of three main stages. Firstly, Pillar Feature Network, where the input point cloud is divided into evenly spaced vertical pillars (columns) in the x-y plane and then encoded to a sparse pseudo image by simplified PointNet architecture. The pseudo image can then be processed with 2D-convolutional layers. Each point, added to the pillar, is extended to a 9-dimensional vector, including the point's original position, intensity and five parameters describing the point's position relative to the pillar. Also, the maximum point count for each pillar is fixed, and in case of more points, they will be randomly sampled to fit the pillar. If the pillar is empty, zero padding is applied to keep the tensor size. Secondly, the Backbone stage uses the pseudo image and 2D convolutional layers to produce a high-level input representation. The 2D convolutional operations are extremely efficient on GPU, enabling low inference time compared to models using 3D convolutional layers. Thirdly, a Single Shot Detector (SSD) predicts the final 3D bounding boxes. In addition, it is possible to use the PointPillars network for semantic segmentation tasks if the SSD detection head is changed to the appropriate one [36].

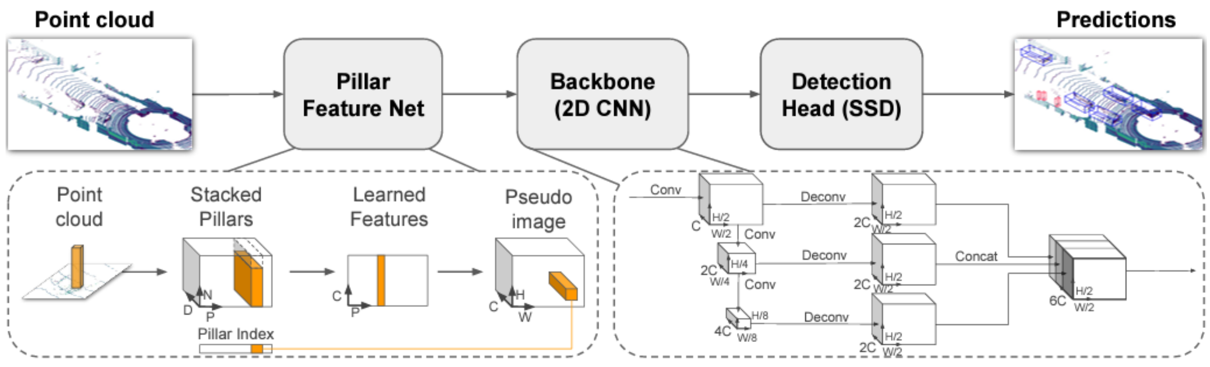

FIGURE 1.9: PointPillars model architecture [36].

Three losses are measured during the model training and validation: the model's localisation, heading and classification error. In addition, the model performance is measured with BEV and 3D mAP metric, where BEV shows that the bounding boxes IoU is evaluated in the 2D space, while with 3D mAP, the IoU is evaluated in the 3D space. It is shown that data augmentation is important for good performance, therefore, three types of augmentations are added to input data. Firstly, random insertion of previously saved bounding boxes with corresponding points included in the bounding box. Secondly, adding rotations and translations to all bounding boxes and lastly, global augmentations, including mirroring, rotation, scaling and translation, are applied to both point clouds and bounding boxes. In some cases, the more the pedestrians' bounding boxes are augmented, the lower the accuracy. It is mentioned that detecting pedestrians and cyclists is challenging as they might be confused between each other, and pedestrians can be similar to poles or tree trunks. On the other hand, detecting cars should be rather accurate. The model has shown running speeds up to 62 Hz with the normal-sized model and 105 Hz with the simplified version. Lastly, PointPillar authors claim the model should not require hand-tuning with multiple lidar or radar point clouds [36].

1.2.5 Evaluation metrics

Evaluation metrics measure the ML model's performance in generalisation and optimisation. The performance metrics of the ML model can be divided into two categories: accuracy metrics, measuring the model's effectiveness and computational complexity metrics, measuring the model's scalability for deploying in resource-limited systems [1]. In the following two sections overview of multiple metrics is discussed, starting with accuracy and ending with computational complexity metrics.

Accuracy metrics

To measure the effectiveness of the object detection and semantic segmentation model, localisation and classification accuracy should be evaluated [1]. There is no perfect metric; therefore, multiple metrics can be combined to improve the understanding of the model's generalisation power. The following abbreviations are used for the number of true positive (TP), true negative (TN), false positive (FP) and false negative (FN) predictions in one frame. In the case of object detection, the bounding boxes, where the area or volume between ground truth and the predicted bounding box is above the localisation threshold, are classified as TP predictions. All incorrectly positioned bounding boxes are counted as FP and undetected ground truth boxes make up FN predictions, while TN predictions are not counted in the context of object detection [37]. In semantic segmentation, separate pixels are divided into TP, TN, FP and FN predictions [1]. To apply binary classification metrics to a multi-class problem, as usually object detection and semantic segmentation are, metrics for each class can be calculated separately and then averaged together, also known as a macro-averaged result [38]. This allows the treatment of each class as a binary classification problem, and therefore the TP, TN, FP and FN confusion matrix can be found. This section describes numerous metrics for evaluating object detection and semantic segmentation models, including binary and multi-class classification metrics formulas.

One of the most intuitive ways to evaluate the model performance is to use binary accuracy (A), which calculates the ratio between correct predictions and the total number of predictions [1]. The accuracy only measures the classification aspect of the model without considering the localisation performance. The accuracy can be calculated in the following way:

$$\mathrm{A} = \frac{\textit{number of correct predictions}}{\textit{total number of predictions}} = \frac{TP + TN}{TP + TN + FP + FN}[1,\ 38] \tag{1.4}$$

One of the biggest drawbacks of accuracy is that it may falsely indicate the model's work if the input data set has imbalanced classes; for example, 90% of data points belong to class 0 and 10% to class 1. In the given example accuracy of 90% is achieved by predicting all objects to class 0. In this case, and with a multi-class problem, it is advisable to use mean, also known as a macro, accuracy (mA), which calculates the accuracy for each class separately and then averages over all classes [1]. The mean accuracy can be calculated in the following way:

$$\mathrm{mA} = \frac{1}{n} * \sum_{i=1}^{n} A_i, \tag{1.5}$$

where A_i is the accuracy for i'th class and n is the total number of classes [1].

The next widely used option is the intersection over union (IoU) metric, which evaluates the overlap between ground truth and prediction boundaries or regions, showing the model's localisation performance. The IoU is actively used in object detection and semantic segmentation deep learning models [1, 4]. In the case of object detection, the IoU is calculated based on the area between predicted and ground truth bounding boxes as:

$$\mathrm{IoU} = \frac{\textit{predicted} \cap \textit{truth}}{\textit{predicted} \cup \textit{truth}}[1,\ 4] \tag{1.6}$$

In the case of semantic segmentation, the IoU is calculated in the following way based on the number of TP, FP and FN classified pixels:

$$\mathrm{IoU} = \frac{TP}{TP + FP + FN}[1] \tag{1.7}$$

Similarly to accuracy, there is an option to use mean IoU (mIoU) with a multi-class problem [1]. The mIoU can be calculated as:

$$\mathrm{mIoU} = \frac{1}{n} * \sum_{i=1}^{n} IoU_i, \tag{1.8}$$

where IoU_i is the IoU for i'th class and n is the total number of classes[1].

The last accuracy metrics discussed in this section are precision, recall and F-score. Precision and recall make up a Precision-Recall Curve (PRC) which can be used to discriminate between FP and FN predictions, an opportunity that classification accuracy

[1]`https://www.tensorflow.org/api_docs/python/tf/keras/metrics/MeanIoU`

and IoU lack [1]. Similarly to previous metrics, a macro-averaged result can be found in a multi-class problem. Precision (P) can be calculated as:

$$\text{P} = \frac{\textit{true positives samples}}{\textit{positive samples}} = \frac{TP}{TP + FP}[1,\ 38,\ 39] \tag{1.9}$$

Recall (R) can be computed as:

$$\text{R} = \frac{\textit{true positives samples}}{\textit{relevant samples}} = \frac{TP}{TP + FN}[1,\ 38,\ 39] \tag{1.10}$$

In addition, PRC enables the evaluation of one of the most used single-value metrics in object detection or semantic segmentation tasks, called average precision (AP). The area under the PRC curve defines the value of average precision. In the case of a multi-class problem, mean average precision (mAP) is used [1].

Lastly, F-score measures the harmonic mean between ground truth and predictions by combining precision and recall. Also, it is possible to measure the mean F-score similarly to the previously mentioned metrics [39]. The following formula is used to calculate the F-score:

$$\text{F-score} = 2 * \frac{\textit{precision * recall}}{\textit{presicion + recall}} = 2 * \frac{P * R}{P + R}[4,\ 39] \tag{1.11}$$

Computational complexity metrics

Multiple metrics are available for evaluating the computational complexity of the ML model. In the scope of this work, the most important metric is the inference speed, evaluated in frames per second (FPS). Evaluating the inference speed is especially important if the model is meant to work in real-time, for example, for autonomous driving vehicles. In addition, there are metrics for memory usage and for calculating floating point operations, which the model does during the operation [1]. They will become more important if the system resources limit the memory capacity and computational speed. For example, if the model is deployed on a microcontroller.

2 Methods

This chapter describes the work conducted in the practical part of this thesis. Firstly, system setups used to run machine learning models and data acquisition software are described. A description of the data generation and annotation process follows it. Lastly, a chosen semantic segmentation and object detection algorithms are discussed, including the changes made to be compatible with the author's generated data.

2.1 System Setup

In the course of this thesis, two computer systems were used due to the ease of access in different working locations and the difference in computing power. The first system (system A) was used for data generation and annotation, and the other (system B) for training and testing the machine learning model. The system A parameters were the following:

- Operating system: Ubuntu 20.04.5 LTS
- CPU: Intel Core i5-11400F @ 2.60GHz
- GPU: NVIDIA GeForce RTX 3060 Ti 8 GB
- RAM capacity: Kingston 16 GB @ 3200 MHz
- Storage: WD Green SSD SN350 1 TB

The system B parameters were the following:

- Operating system: Ubuntu 20.04.5 LTS
- CPU: Intel Core i9-11900K @ 5.0 GHz
- GPU: MSI GeForce RTX 3090 SUPRIM X 24 GB
- RAM capacity: G.Skill Trident Z Neo 64 GB @ 3600 MHz
- Storage: Kingston SSD SNV2S 500 GB

The most noticeable differences were in the CPU and GPU memory capacity, which can significantly affect the machine learning model training speed. On the other hand, system A, which had lower specifications, suited its data generation and annotation tasks competently.

While writing the thesis, Grammarly[1] was used to verify the correctness of spelling and to improve the text comprehensibility. The educational subscription was provided by the University of Tartu, and British English was chosen as the base language.

2.2 Data Acquisition

The data acquisition pipeline consisted of multiple steps: generation, annotation and filtering. All mentioned steps will be discussed in the following few sections. Due to the camera's technological readiness level during the writing of the thesis, it was preferable to use simulated data to conduct the rest of the practical work. It was essential that simulation software allowed the creation of a relevant environment and represented the actual camera use case and data format as close as possible to LightCode Photonics camera.

CARLA Simulator version 0.9.13[2], based on Unreal Engine 4, was selected as the simulation software, an open-source simulator mainly for autonomous driving applications. It already provides free premade assets, including standard sensors, maps, cars, etc., with the flexibility to add or modify current assets. In addition, it supports pedestrian and traffic managers, traffic rules, and intersections that make the simulator mimic the real-world as close as possible. Lastly, CARLA Simulator can provide bounding boxes for all dynamic assets in the environment, simplifying and accelerating the data generation process [40].

Using simulated data compared to real-world data has multiple advantages and disadvantages. The negative side of using simulated sensor data is that it does not support the multiple returns feature, which could increase the effective resolution of the sensor in some cases. Furthermore, premade camera sensors do not output the reflectivity of objects in the scene, and the range limit is abrupt, i.e. does not depend on the targets' reflectivity. Lastly, the horizontal and vertical angular resolutions have to be equal but might differ in LightCode Photonics 3D camera case. On the positive side, simulated data can be less noisy and less affected by environmental conditions, e.g. sunlight intensity or direct reflections, that might otherwise blind the sensor. In addition, the environment, including weather, pedestrians, traffic, etc., is highly controllable.

CARLA Simulator is developed as a server-client system, meaning the server side is responsible for rendering the scene and running the simulator environment. The Client-side is responsible for interacting between the client application and the server, creating

[1]`https://www.grammarly.com/`
[2]`https://carla.org/2021/11/16/release-0.9.13/`

sensors and managing the returned sensor data. The server-client system allows the client to run on different machines than the server, as communication between the server and client happens through sockets. Also, the server supports multiple clients at the same time [40]. In the scope of this thesis, CARLA Simulator server and client-side were on the system A (Section 2.1) setup.

2.2.1 Server-Side

In CARLA Simulator, available assets are named blueprints, each having its input and output attributes. By default, the simulator supports 14 different types of sensor blueprints[3], including RGB camera, LiDAR sensor and depth camera. None of the available sensors directly represents LightCode Photonics camera output and data acquisition type. The closest ones are the depth camera and LiDAR sensor. The depth camera returns a 2D array of distances that should be converted to a point cloud but does not include reflectivity values nor have the option to set the maximum working range. On the other hand, the LiDAR sensor outputs a point cloud with reflectivity values and allows to set the operating range, but it represents 360-degree scanning LiDAR acquisition format that differs compared to LightCode Photonics camera-like acquisition. Therefore, it was decided that to use LightCode Photonics camera in CARLA Simulator conveniently, it should be implemented in the server as an available blueprint. In addition, it helps to have cleaner code in the client and share the camera blueprint with customers using CARLA Simulator.

To add LightCode Photonics camera blueprint to CARLA Simulator source code, it was forked from the original GitHub repository [41] and modified accordingly to CARLA Simulator documentation[4]. The blueprint was set up similarly to a depth camera, requesting a depth BGRA image array from the underlying Unreal Engine and converting it to a 2D distance array. All distances over the maximum operating range were discarded, and the distance array was converted to a point cloud using the pseudocode shown in listing 2.1. All the modified server-side code is accessible through appendix A.

```
int hRes = horizontal_resolution_pixels
int vRes = veritcal_resolution_pixels
float FoV = field_of_view_degrees

# Camera intrinsic parameters.
int cx = hRes / 2
int cy = vRes / 2
float fx = hRes / (2.0 * tan(FoV * PI / 360))
float fy = fx

# Row and Column values represent the pixel position on the image.
# Calculate the 3D coordinate for each distance measurement in the array.
for row, column, distance in distances_2d_array:
    float x_coordinate = distance
    float z_coordinate = ((cy - row) / fy) * x_coordinate
    float y_coordinate = -((cx - column) / fx) * x_coordinate
```

Listing 2.1: Pseudocode of converting distances to 3D points

[3]https://carla.readthedocs.io/en/0.9.13/ref_sensors/

[4]https://carla.readthedocs.io/en/0.9.13/tuto_D_create_sensor/

The table 2.1 shows the camera blueprint's basic attributes and is accessible to clients with path *sensor.camera.lightcode*. In addition, it is possible to set camera lens distortion attributes similar to depth or RGB cameras.

Table 2.1: LightCode Photonics camera blueprint input attributes

Blueprint attribute	**Type**	**Description**
image_size_x	int	Image width in pixels.
image_size_y	int	Image height in pixels.
fov	float	Horizontal field-of-view in degrees.
range	float	Maximum operating range in meters.
sensor_tick	float	Simulation seconds between sensor captures.

The LightCode Photonics camera blueprint outputs data shown in the table 2.2.

Table 2.2: LightCode Photonics camera blueprint output attributes

Output attribute	**Type**	**Description**
frame	int	Frame number of the measurement.
timestamp	double	Simulation time during measurement in seconds.
transform	carla.Transform	Location and rotation in world coordinates.
width	int	Image width in pixels.
height	int	Image height in pixels.
fov	float	Horizontal field-of-view in degrees.
range	float	Maximum operating range in meters.
raw_data	bytes	Left-handed array of 3-dimensional points.

The raw data returned by the camera blueprint can be converted to a 3D Numpy array with the Python code in listing 2.2. It is important to note that the point cloud is left-handed, meaning the Y-axis must be inverted to convert it to the right-hand rule.

```python
xyz_coordinates = np.ndarray(shape=((lightcode_image.width * lightcode_image.height),
                            3), dtype=np.float32, buffer=lightcode_image.raw_data)
```

Listing 2.2: Python code to convert raw data to array of 3D points

CARLA Simulator must be built from the source using the forked repo code to use the created LightCode Photonics camera blueprint. After which, the blueprint functions and parameters are accessible through CARLA Simulator Python application programming interface (API) to set up the environment and sensors.

2.2.2 Client-Side

On the client-side, users can choose the simulation environment, set up suitable traffic, interact with autonomous agents, spawn and use specific sensors. It was decided that this thesis focuses on testing the camera capabilities in the urban environment for detecting different road users, like cars, pedestrians, bicycles, etc. Different intersections were chosen to generate data using a statically placed LightCode Photonics cameras. The simulator was configured to run in synchronous mode with a fixed refresh rate of 20 Hz. From the available premade maps in CARLA Simulator *Town10HD* was selected as it has the most background details available. The automatically generated traffic consisted of different category cars, motorcycles, bicycles and pedestrians and all of them used automatic path planning. For deterministic behaviour, a random seed was used for the traffic manager during multiple runs. It was observed that the CARLA Simulator's different runs do not always behave deterministically and are not repeatable, even if all the conditions based on the documentation were fulfilled. The thesis author also reported an issue to the official CARLA Simulator GitHub repository about this bug[5]. The default weather settings were not altered, as different conditions would not interfere with LightCode Photonics blueprint behaviour in the simulator. The client code was written using Python API, and figure 2.1 shows a high-level view of the code. All the client-side code is available in the appendix C.

[5]`https://github.com/carla-simulator/carla/issues/5360`

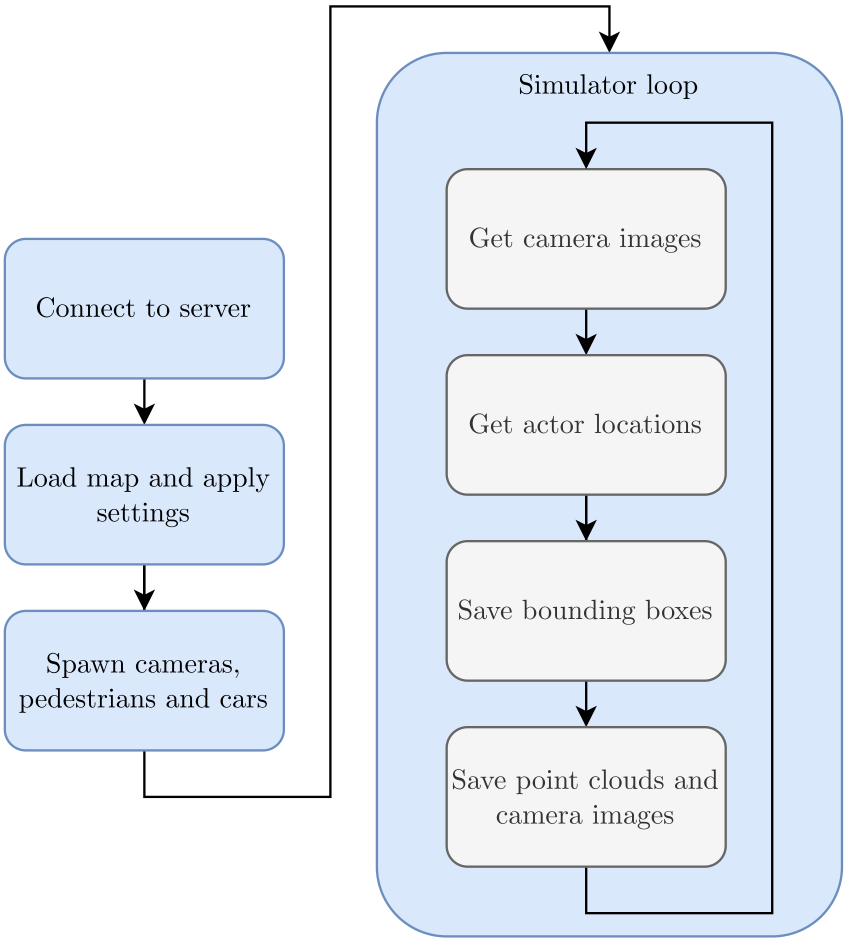
Connect to server
Load map and apply settings
Spawn cameras, pedestrians and cars
Simulator loop
Get camera images
Get actor locations
Save bounding boxes
Save point clouds and camera images

FIGURE 2.1: High-level graph of the client Python code. Firstly, a connection to the server and traffic manager is established. Followed by choosing a suitable environment and applying world settings. Thirdly, all actors are spawned into the environment, including pedestrians, sensors, and cars. Lastly, a simulator loop is started, where each received frame' bounding boxes and point clouds are saved.

In the client code, LightCode Photonics cameras were spawned together with RGB cameras to have an additional reference to the point cloud data. The RGB images improve the annotation process quality and speed, making checking the bounding box boundaries and object type more precise. Spawned camera parameters are given in the table 2.3. As the simulator cannot run at a low refresh rate due to physics substepping[6] limits, then to achieve a 5 Hz refresh rate for cameras, the intermediate frames had to be filtered out by client code. Another option would be to use *sensor_tick* camera attribute, but then it was noted that the synchronisation between server and client was not stable enough for data collection. LightCode Photonics camera increased range compared to the original specification shown in table 1.1 is due to the desire to test the increased range advantages

[6]`https://carla.readthedocs.io/en/0.9.13/adv_synchrony_timestep/`

in the smart city use case and also due to the strict limit of the range in the generated data. In addition, the FoV of LightCode Photonics camera is 80 × 13.3 degrees, as the horizontal and vertical axis angular resolutions must match.

Table 2.3: Spawned LightCode Photonics and RGB camera specifications

Attribute	**LightCode Photonics camera**	**RGB camera**
Field-of-view (°)	80 × 13.3	80 × 13.3
Resolution (px)	96 × 16	800 × 134
Angular resolution (°/px)	0.83	0.1
Refresh rate (Hz)	5	5
Range (m)	35	unlimited

In a total of 6 cameras, 3 from both categories were placed in the same location in the world with different rotations. The cameras' FoVs were aligned in the horizontal plane, covering 240-degree FoV. To mimic the possible position in the real-world use case, the cameras were placed on a virtual post with a height of 2 meters. Figure 2.2 shows one possible positioning of LightCode Photonics and RGB cameras.

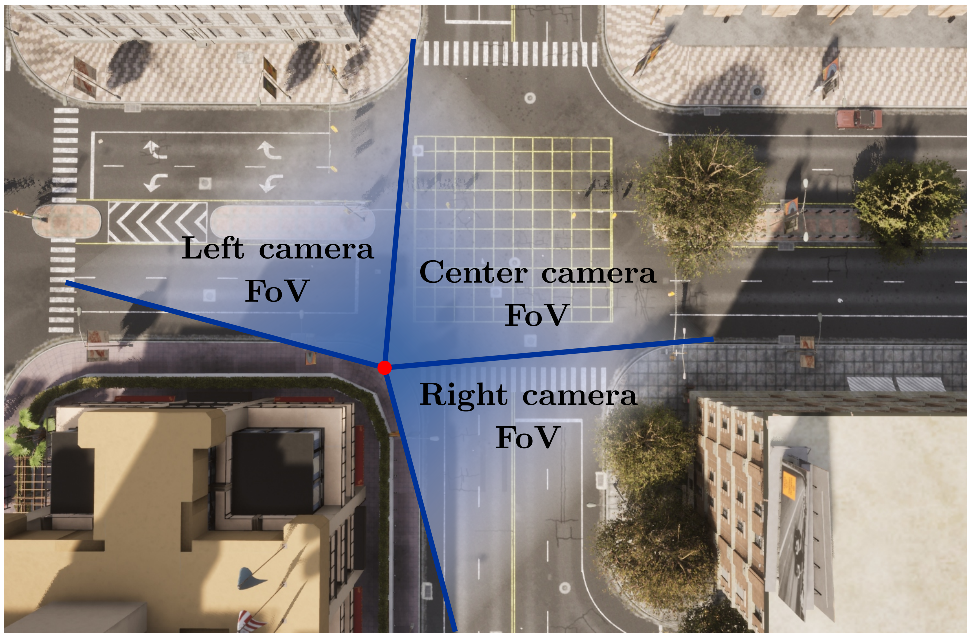

Figure 2.2: Top view of an example intersection showing the possible positioning of cameras. The cameras' position is shown with a red dot and FoV boundaries with blue lines. Each LightCode Photonics camera covers horizontally 80-degree FoV and is accompanied by an RGB camera with the same location and rotation.

The data was generated in two parts. The first time the automatic annotation process was not used, all frames were annotated manually. Before the second time, an automatic annotation[7] code was added to the client to improve its speed. It was observed that the automatic annotation process does not entirely remove the hand correction step, as some bounding boxes were still slightly off compared to generated point clouds (Figure 2.3). In addition, CARLA Simulator had two large actor groups, vehicles and walkers, i.e. pedestrians, which meant the bicycle and motorcycle object types must be manually changed afterwards. The server and client applications had to be well synchronised for the automatic annotation process to work. Otherwise, offsets between camera images and bounding boxes occurred. Also, the version of CARLA Simulator did not allow to filter out bounding boxes that were hidden behind other obstacles. For example, a bounding box for a small car was still returned even if it was completely hidden behind a large truck. Lastly, the distance between the camera transforms, and the actors' bounding boxes were calculated and compared to select bounding boxes close to the cameras.

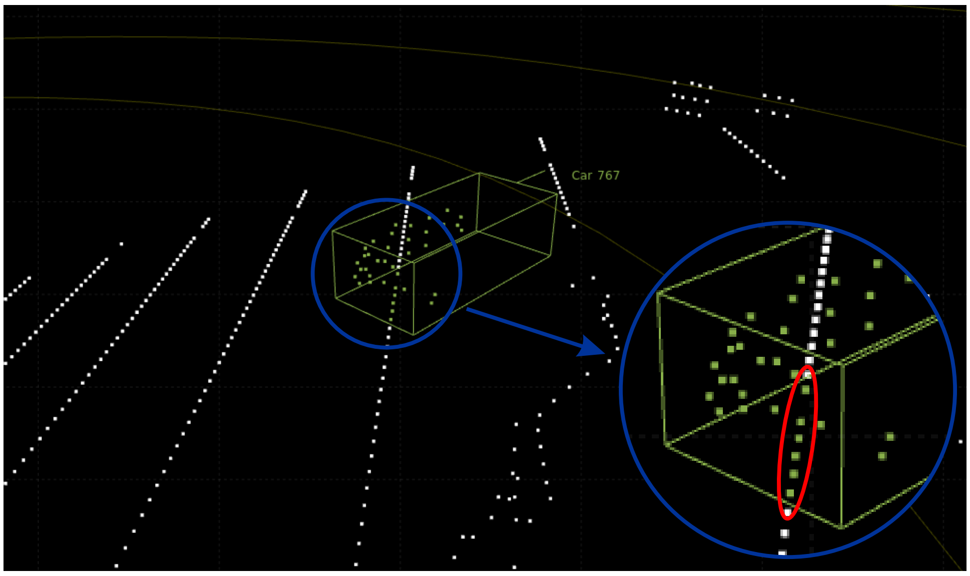

FIGURE 2.3: Perspective view of a point cloud in the annotation tool showing an automatically generated bounding box for a car. Points included in the car' bounding box are marked green, and background points are shown as white. A few green points are included in the bounding box, shown with a red ellipse, which are incorrect and should be excluded.

In order to be compatible with the chosen data annotation tool, the client code had to output four types of data in each simulator frame:

- Three RGB images, i.e. each RGB camera output was saved to separate .jpg image file
- Three LightCode Photonics camera point clouds, i.e. each camera' generated output was saved to separate .pcd file

[7]`https://carla.readthedocs.io/en/0.9.14/tuto_G_bounding_boxes/`

- One point cloud .pcd file, where all LightCode Photonics cameras' point clouds have been merged.
- One .json file containing automatic annotations for all cameras.

The need for a separate point cloud file containing merged camera point clouds was specific to the annotation tool, as it expected one .pcd point cloud and one .json annotation file for each frame. In addition, keeping the point clouds together allowed tracking the object through multiple cameras FoV, improving the overall annotation speed. Lastly, for seamless workflow, it was important that the output format of CARLA Simulator annotations must match the annotation tool format.

2.2.3 Data Processing Pipeline

After being generated by the client code, the data is processed in multiple steps to prepare the data for object detection and semantic segmentation models. They can be divided into the annotation, separation, filtration and adding intensity phases. The overall pipeline is shown in figure 2.4. The following paragraphs will describe the pipeline parts more comprehensively.

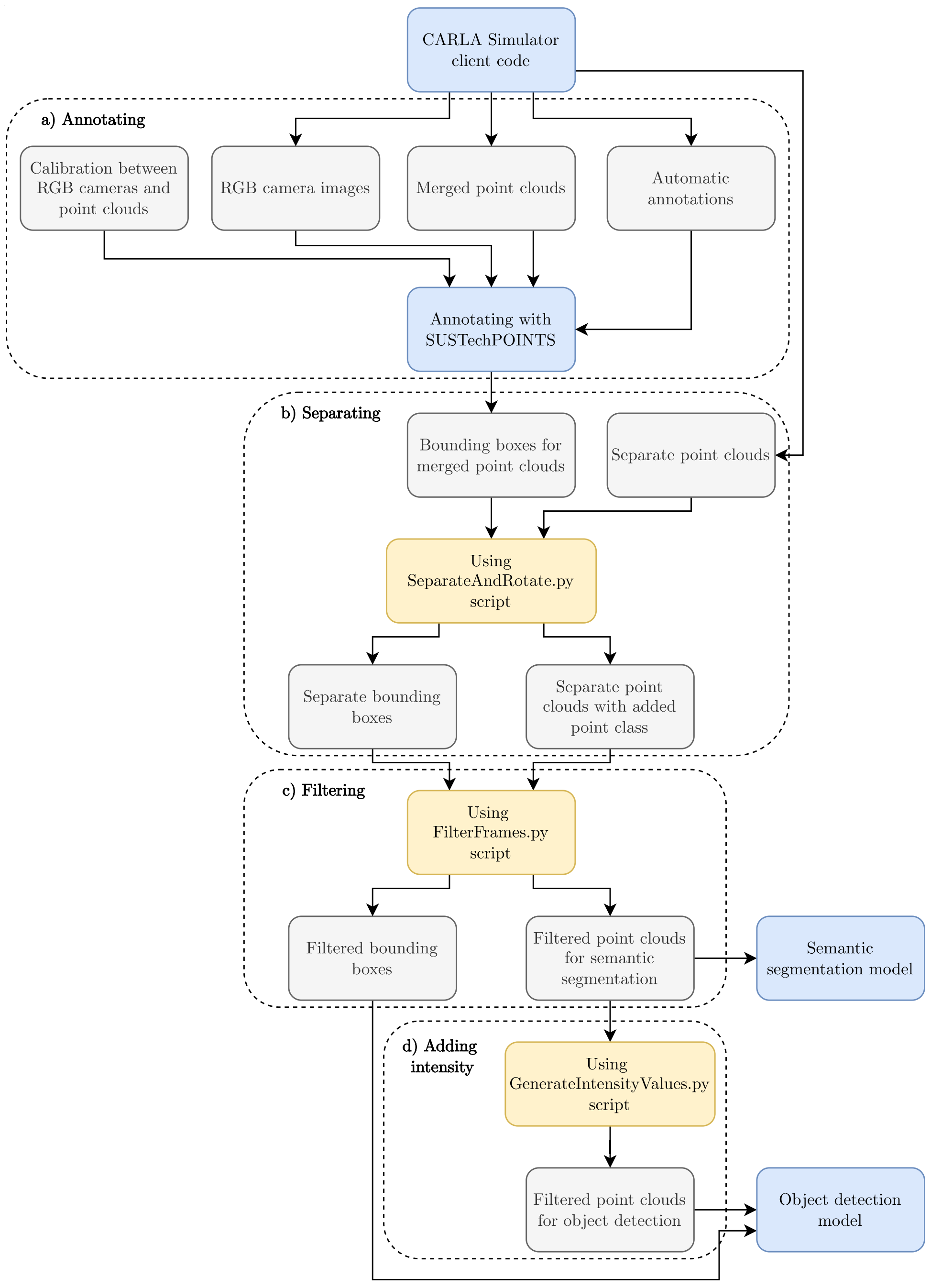
CARLA Simulator client code
a) Annotating
Calibration between RGB cameras and point clouds
RGB camera images
Merged point clouds
Automatic annotations
Annotating with SUSTechPOINTS
b) Separating
Bounding boxes for merged point clouds
Separate point clouds
Using SeparateAndRotate.py script
Separate bounding boxes
Separate point clouds with added point class
c) Filtering
Using FilterFrames.py script
Filtered bounding boxes
Filtered point clouds for semantic segmentation
Semantic segmentation model
d) Adding intensity
Using GenerateIntensityValues.py script
Filtered point clouds for object detection
Object detection model

FIGURE 2.4: The data processing pipeline starts with gathering data from CARLA Simulator. a) The next step is data annotation which can be done manually or assisted by automatic annotations from CARLA Simulator. b) The improved and still merged annotations, i.e. bounding boxes, are then separated into different files that can be linked to each camera's point cloud and used to add a semantic class for each point. c) It is followed by filtering out the frames containing no objects. d) Lastly, an intensity field is added to the point cloud as it is one of the requirements of the selected object detection algorithm.

Data annotation (Figure 2.4, part a) was done with SUSTechPOINTS open-source tool [42, 43]. It is a semi-automatic 3D point cloud annotation tool with RGB camera image support that, for each frame, generates a list of bounding boxes to a .json file with the layout shown in listing 2.3. In addition to point clouds and RGB images, the tool needs calibration files that determine the RGB cameras' extrinsic and intrinsic parameters to match the bounding boxes between RGB images and point clouds. The calibration files were calculated and generated manually. The tool also supported automatic annotating using a pre-trained ML model for new point clouds, but it was not working stable enough with the generated data, therefore mostly interpolation between two fixed bounding boxes was used. It was suspected that it might be due to the lower resolution point cloud compared to what the model is trained on. The road users were divided into four categories: car, pedestrian, bicycle and motorcycle, where the car category included trucks, vans and passenger cars.

```json
{
    "obj_id": "1",
    "obj_type": "Car",
    "psr": {
        "position": {
            "x": -7.627991017406543,
            "y": 21.434290470313428,
            "z": -1.0838902804892578
        },
        "rotation": {
            "x": 0,
            "y": 0,
            "z": -3.115412714809878
        },
        "scale": {
            "x": 4.174792711505839,
            "y": 1.7091842164074056,
            "z": 1.408580385606108
        }
    }
}
```

Listing 2.3: Annotated object in .json file. SUSTechPOINTS enables to use of 9 Degrees of Freedom bounding boxes.

The separation phase (Figure 2.4, part b) used the bounding boxes from SUSTechPOINTS annotation tool and separate point clouds generated by CARLA Simulator. The phase aimed to divide the bounding boxes between three separate point clouds and discard bounding boxes with too few points. The minimum number of points the bounding box must contain to be saved is 12 for car and 6 for pedestrian, bicycle and motorcycle classes. During the separation, the bounding box might be present in the middle of the separation line and, therefore, was saved in both separated bounding box output files. An example based on one frame is shown in figure 2.5. The phase outputted three .json files containing bounding boxes in each camera point cloud and three point cloud .txt files, where a class number was added as a fourth parameter to a point, resulting in point format: X, Y, Z, class. For semantic segmentation, an additional background class was added, as all the points must be classified. A custom-made script was used to process the bounding boxes and point clouds, called *SeparateAndRotate.py*, available in appendix C.

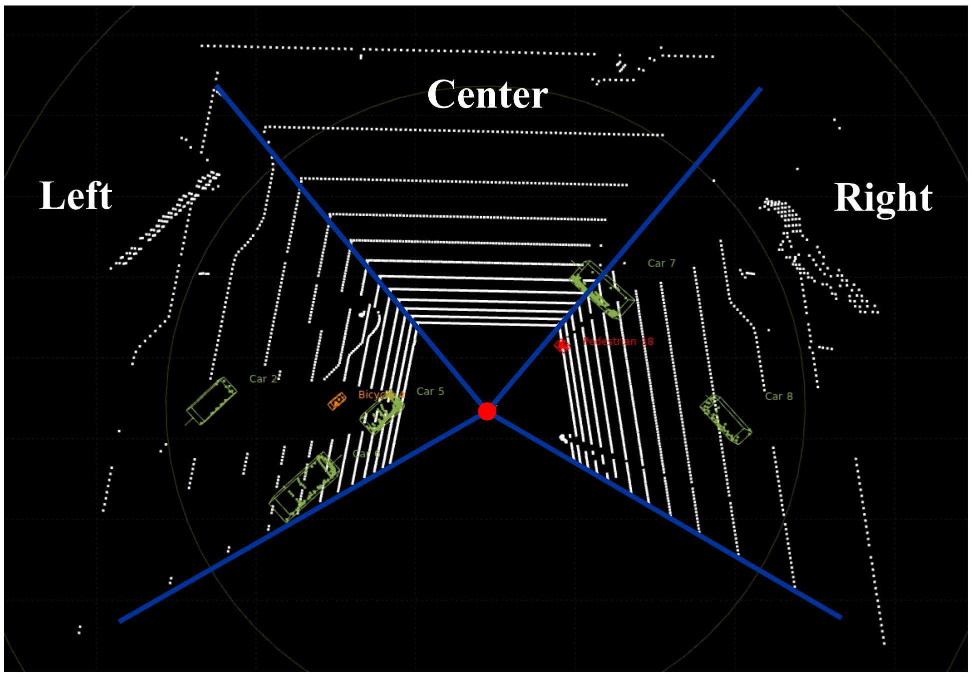

FIGURE 2.5: The figure shows a top view of the annotation tool output. In the separation phase, the output is divided into three sections, corresponding to the right, centre and left camera. In this case, the right camera annotation file contains three, centre one and left four bounding boxes after the separation, as the *car 7* bounding box is in the right and centre point clouds.

Additional data filtering (Figure 2.4, part c) was applied to the generated data to remove empty frames, i.e. frames without any object. Leaving these frames into the input dataset might lead to quicker ML model overfit, therefore, lower generalisation power. The filtering was done using a custom-made *FilterFrames.py* (Appendix C) Python script that scanned through point clouds and annotations, saving only appropriate files. The outputted point cloud files were directly usable with a semantic segmentation model.

Lastly, point clouds were required to contain intensity values in the data to use the selected object detection model (Figure 2.4, part d). As the defined LightCode Photonics camera in CARLA Simulator could not output intensity values, the class parameter in the point cloud files was replaced with zeros indicating the absence of that variable. Therefore the point cloud files for object detection resulted in a point format: X, Y, Z, 0. The processing was done using the *GenerateIntensityValues.py* (Appendix C) script.

2.3 Semantic Segmentation and Object Detection Models

In the practical part of this thesis, two different object recognition models were considered. Firstly, PointNet [27] semantic segmentation model and secondly, PointPillars [36] object detection model. The two architectures were chosen as they both can run in real-time applications and allow the evaluation of both object detection and semantic segmentation task capability. In the longer term, LightCode Photonics must offer a solution that can be used in real-time, as it is crucial for robotic applications. In addition, to the thesis author's knowledge, these models have not been tested with very sparse point clouds, as the output of LightCode Photonics camera was. These models were trained and tested on system B (Section 2.1). The following two sections will describe the changes made to the models' implementations to get the models working with previously generated data, starting with PointNet, followed by PointPillars.

2.3.1 PointNet

The PointNet model was trained and tested using the Jupyter Notebook file from the available Keras implementation [35]. The Keras version was chosen as it offers a more user-friendly API than the original TensorFlow framework. The Keras API enables faster and more convenient code testing and modifying while still allowing to use more advanced features, for example, defining custom metrics. The following paragraphs will overview the most significant changes to the original code. All the changes described in the following paragraphs are visible in the appendix C.

Firstly, the input data reading code (Listing 2.4) was modified to parse point clouds and ground truth labels correctly. The labels were saved as a fourth parameter for each point in the generated data, therefore was no need to read additional files. In addition, the original code discarded all the point clouds that did not meet the minimum required point count. The mentioned section was replaced with data padding, allowing to use point clouds with variable sizes. In order to do so, the point cloud size was fixed to 1536, which comes from the LightCode Photonics camera resolution, which was 96×16 pixels (Table 1.1). Repeated data was used from the original point cloud to standardise the point count, i.e. already existing points were duplicated to increase the point count. The solution was proposed by one of the authors of the PointNet [27] paper under this GitHub issue[8].

```

LABELS = ["Car", "Pedestrian", "Bicycle", "Motorcycle", "Background"]
NUM_SAMPLE_POINTS = 1536
points_dir = "InputData/lidar_semantic/*.txt"

point_clouds = []
labels = []
labels_str = []

# Read point cloud files and separate them to point cloud and label arrays
for point_file in tqdm(glob(points_dir)):

    point_cloud_with_label = np.loadtxt(point_file, dtype=np.float32)

    point_cloud_points = point_cloud_with_label.shape[0]
    if point_cloud_points < NUM_SAMPLE_POINTS:

```

[8]https://github.com/charlesq34/pointnet/issues/161

```
        # Padding the point cloud with repeated data.
        # According to this: https://github.com/charlesq34/pointnet/issues/161
        missing_point_count = NUM_SAMPLE_POINTS - point_cloud_points
        point_cloud_with_label = np.append(point_cloud_with_label,
                              point_cloud_with_label[:missing_point_count], axis=0)

    label_column = np.array(point_cloud_with_label[:,[3]], dtype=np.int64)

    # String array for labels
    label_map = ["none"] * len(label_column)
    for i, label in enumerate(label_column):
        label_map[i] = LABELS[label[0]]

    point_clouds.append(point_cloud_with_label[:,:3])
    labels.append(label_column)
    labels_str.append(label_map)
```

Listing 2.4: Python code to read from the generated data and extract point clouds and ground truth labels. In addition, point cloud padding is done to align the point count in all the point clouds. The code outputs three arrays containing: point clouds, labels and labels in string form for better visualisation.

Also, due to data padding, there was no need to discard points over the point count threshold, previously done in the preprocessing step to standardise the point clouds. Therefore, no points were lost from the original point cloud after the data padding implementation. Before generating database objects, the only pre-processing step was normalising the point cloud to make them range and position invariant [33].

Secondly, two custom metrics were added: mean accuracy and mean F1-score, to evaluate the accuracy performance of the single-label categorical classification model, as neither is implemented in the Keras API. The calculation of the mean accuracy metric is shown in the listing 2.5. In the case of mean F1-score, TensorFlow Addon API binary F1-score[9] was used as a base metric and the custom metric class was responsible for formatting the inputs and returning the mean value. The mean F1-score metric implementation can be found in appendix C. The original PointNet paper [27] uses mean IoU and binary accuracy to evaluate the model's performance; therefore, it was essential to include mentioned metrics as well to be able to compare the thesis author's model performance to the original model.

```
class MacroAccuracy(tf.keras.metrics.Metric):
    def __init__(self, num_classes, **kwargs):
        super().__init__(name="macroaccuracy", **kwargs)
        self.num_classes = num_classes
        self.correct_positives = self.add_weight(name="correct_positives",
                                                 shape=[self.num_classes],
                                                 initializer="zeros",
                                                 dtype=tf.float32)
        self.total_predictions = self.add_weight(name="total_predictions",
                                                 shape=[self.num_classes],
                                                 initializer="zeros",
                                                 dtype=tf.float32)

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true_one_hot = tf.one_hot(y_true, depth=tf.shape(y_pred)[2])
        y_true_reshaped = tf.reshape(tf.cast(y_true_one_hot, dtype=tf.float32),
```

[9]https://www.tensorflow.org/addons/api_docs/python/tfa/metrics/F1Score

```
                                             [-1, self.num_classes])
        y_pred_reshaped = tf.reshape(tf.cast(y_pred, dtype=tf.float32),
                                             [-1, self.num_classes])
        y_pred_one_hot = tf.one_hot(tf.argmax(y_pred_reshaped, axis=-1),
                                             depth=self.num_classes)

        self.correct_positives.assign_add(tf.reduce_sum(y_true_reshaped *
                                                        y_pred_one_hot, axis=0))
        self.total_predictions.assign_add(tf.reduce_sum(y_true_reshaped, axis=0))

    def result(self):
        return tf.reduce_mean(tf.math.divide_no_nan(self.correct_positives,
                                                     self.total_predictions))

    def reset_state(self):
        reset_value = tf.zeros([self.num_classes], dtype=tf.float32)
        self.correct_positives.assign(reset_value)
        self.total_predictions.assign(reset_value)
```

Listing 2.5: Custom Keras metric class to calculate the mean accuracy. The metric was updated after each batch in the *update_state()* function, and the mean value was returned in the *result()* function. The labels must be converted to a one-hot format to reduce the true positive calculations to *AND* function between ground truth labels and predictions. Before that, an *argmax()* function is used on predictions to get the highest confidence label.

Thirdly, implementing the K-fold cross-validation strategy to evaluate model performance was one of the most significant changes to the original code. The K-fold was chosen as it enables to use a smaller dataset for model performance evaluation and helps to reduce errors related to the dataset splitting. The implementation was done based on the book [33]. The code of K-fold implementation is visible in the appendix C, and the figure 2.6 shows how the data was divided during the model training and evaluation. Data shuffling happens after diving into the testing, training and validation datasets, not the whole dataset, as the consecutive frames were strongly correlated in the point cloud data. The shuffling of whole data would increase the accuracy performance of the model but does not show the generalisation power of the model on new data. Shuffling the training data reduces the possibility that the model will start to overfit after the first samples.

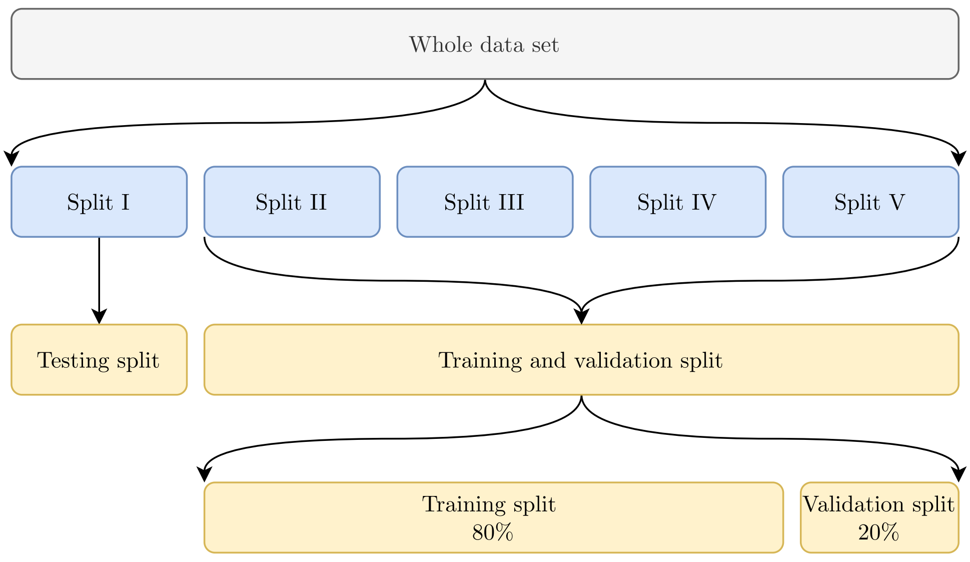

FIGURE 2.6: The whole dataset is divided into five splits, shown with blue boxes. At each fold, different splits are chosen to form the testing, training and validation dataset, shown with yellow boxes. One of the splits is always chosen as the testing dataset, which is used to evaluate the model after training. The training and validation datasets are used during model training and formed based on the other four splits, with a ratio of 80:20.

Lastly, an option to run one point cloud inference at a time was added to measure the computational complexity of the model during deployment. Therefore, one previously trained model was chosen to generate predictions and measure the inference time with Python *time.time_ns()* function that has a resolution of 84 ns on the Linux operating system[10].

2.3.2 PointPillars

The tested PointPillars model uses the TensorFlow implementation from Open3D-ML repository [44], which is an extension to the popular Open3D library for 3D data processing [45]. One of the main reasons for using Open3D-ML implementation is that the package already includes tools for point cloud and bounding box visualisation, simplifying the process of checking the bounding boxes of input and output data from the model. In addition, the Open3D-ML package includes other object recognition model implementations, allowing validation of data in multiple models in the future. The following paragraphs will overview the most notable changes to the Open3D-ML repository code. In this section, all mentioned file paths are shown from the repository root folder, and the fully modified code is available in the appendix B.

A new dataset class, *BBLightCode*, was added to support SUSTechPOINTS annotation tool bounding boxes and the generated point cloud files. The class was implemented in

[10]`https://peps.python.org/pep-0564/#annex-clocks-resolution-in-python`

the file *ml3d/datasets/bblightcode.py*, and for correct implementation, a provided template and KITTI [46] dataset examples were used. The *BBLightCode* class was used for loading and saving the predicted bounding boxes, as well as using the bounding boxes and point clouds for visualisation and training. The default Open3D-ML allows to read in and use bounding boxes defined in a different coordinate system than point clouds. For example, KITTI dataset 3D bounding boxes have been defined in the RGB camera coordinate system. In the implemented class, the calibrations were replaced with an identity rotation matrix to reflect that no rotations are needed in the point cloud and bounding box coordinate systems. The KITTI dataset label format is findable in the *object development kit* in their project webpage[11].

The Open3D-ML pipeline has been built so that model functions, for example, loss and forward pass, have been implemented in a Python script, and the model configuration is loaded from a separate .yml file. This allows to use different configurations and datasets without modifying the model's source code. The object detection training pipeline for the generated dataset was started using *scripts/train_scripts/pointpillars_lightcode.sh* script (Listing 2.6). As with PointNet, the K-fold cross-validation was used with the PointPillars model, and the generated data was split as shown in figure 2.6. On the other hand, in the case of PointPillars model, the K-fold splits were manually separated and copied to the correct folders.

```
1 #!/bin/bash
2 #SBATCH -p gpu
3 #SBATCH -c 4
4 #SBATCH --gres=gpu:1
5
6 cd ../..
7 python scripts/run_pipeline.py tf -c ml3d/configs/pointpillars_lightcode.yml \
8 --dataset_path /home/timo/Open3D/LightCode_Dataset --pipeline ObjectDetection
```

Listing 2.6: Listing shows the content of the *scripts/-train_scripts/pointpilars_lightcode.sh* script that starts the TensorFlow object detection training pipeline with LightCode Photonics configuration and dataset. In case of testing, an additional *--split test* command-line argument was added to line 8

The configuration file allows changing all the parameters relating to model training, validating or testing, including the model's hyperparameters. In the case of the PointPillar model, the most important ones were related to detectable object properties and point cloud size. Followingly, some noteworthy discoveries are given about the configuration found during the training and testing process:

- The *point_cloud_range* and *voxel_size* parameters are strongly connected as the *voxel_size* is used to divide the point cloud into equal-sized pillars. Therefore the *point_cloud_range* array elements must be multiples of the *voxel_size* elements. In addition, the *voxel_size* third element value must equal the point cloud height[12].

[11] `https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d`
[12] `https://github.com/isl-org/Open3D-ML/issues/473`,

- The anchors minimum and maximum z-axis values describing the allowed object ranges in the model head configuration should be the same and preferably have an average value of all the objects in the dataset[13].
- A separate pickle (.pkl) file must be generated to use the *ObjectSample()* augmentation function, which randomly places objects from the dataset to the point cloud to increase the variety in the training data. The pickle file can be generated with provided Python script, called *scripts/collect_bboxes.py*[14].

The default testing pipeline provided in the Open3D-ML Tensorflow implementation, in file *ml3d/tf/pipelines/object_detection.py*, was modified to calculate model evaluation metrics using the testing dataset. Previously, the validation split was used to evaluate the model performance metrics and testing split only for predictions. The modification allows a more accurate overview of the model's generalisation power as the metrics are calculated on data that has never been revealed to the model. In addition to testing sequence modifications, it was essential to make the bounding boxes available on the testing data, as before they were discarded. Therefore two additional keywords to retrieve the dataset split were added *testing_with_bb* and *test_with_bb*, allowing the testing split ground truth data to be used. The support for these keywords was added to the *BBLightCode* dataset class and PointPillars model file, *ml3d/tf/models/point_pillars.py.* Lastly, the predicted bounding box saving code was finished, which makes it possible to save the bounding boxes with the point cloud corresponding name. This enables the prediction comparison with ground truth data in the Open3D-ML built-in visualiser.

[13]See footnote 12.

[14]`http://www.open3d.org/docs/0.14.1/python_api/open3d.ml.torch.datasets.augment.ObjdetAugmentation.html`

3 Results

This chapter gives an overview of the thesis results, including a discussion of the formed hypothesis. The analysis is based on applying two object recognition ML models on the newly generated dataset. Firstly, a section describing the dataset statistics is presented. Secondly, an overview of PointNet model performance is given, and the chapter is concluded with a discussion of PointPillars model.

3.1 Dataset

The dataset was generated in CARLA Simulator in two parts. The first time, a total of 1851 frames, i.e. 617 frames for each camera and three cameras on one intersection, were generated and hand-annotated. An additional 2748 frames with automatic annotation were generated on a different intersection to increase the initial data variety and count. That makes a total of 4599 frames, and after data filtering (Figure 2.4, part c) 2816 frames remained, forming the final dataset for object detection and semantic segmentation models. As the first batch of data was only hand-annotated, and automatic annotation with hand reviewing was used with the second batch, the size and position of bounding boxes might differ slightly relative to an annotated object.

The point cloud points were divided into five classes for semantic segmentation (Table 3.1). As the class distribution was rather unbalanced, the background class made up approximately 92% of all the points, while bicycle and motorcycle classes remained lower than 0.5%, a mean accuracy metrics were included in the model evaluation. On the generated dataset, a dummy code without any ML that predicted all the points to the background class would achieve an accuracy of 92.3%. In the case of mean accuracy, the metric would be 18.46%, as the binary accuracy for all other classes would be 0%.

Table 3.1: The point distribution to semantic classes in the filtered dataset.

Class	**Count**	**Percentage**
Car	214 243	5.97
Pedestrian	51 301	1.43
Bicycle	5 626	0.16
Motorcycle	4 955	0.14
Background	3 311 871	92.30
TOTAL	**3 587 996**	**100.00**

The filtered dataset contains 5451 bounding boxes in different object classes (Table 3.2). As with semantic segmentation, the lowest number of bounding boxes is for bicycle and motorcycle objects, which might affect the predicting accuracy for these classes. On average, 1.9 objects are present in one frame, while a car is present in each frame, and every second frame contains a pedestrian-class object. In comparison, KITTI dataset for 3D object detection and tracking contains an average of 5.4 objects per frame, which is relatively higher and therefore has many more training examples with the same number of frames [1].

Table 3.2: Bounding boxes count per object class in the filtered dataset.

Class	**Count**
Car	3 169
Pedestrian	1 540
Bicycle	435
Motorcycle	307
TOTAL	**5451**

The raw generated and filtered datasets are available in appendix C. Publishing the raw dataset enables processing and filtering the data differently, for example, by changing the minimum point count required for each class, as the annotations in the raw dataset have been done with fewer required points. Therefore, providing the basis for finding the best model on the generated data.

3.2 PointNet

The model training and evaluation consisted of creating five models, each with 7.36M trainable parameters and training to 150 epochs. No changes to the learning rate, batch size or model hyperparameters were made to preserve the model as close to the original implementation as possible. The batch consisted of 32 point clouds and labels during training, so the epoch contained 71 batches. Each model's best weights were saved, which produced the highest validation mean IoU, and during the evaluation, the best weights were loaded for testing the model on the testing data split.

The average model (Figure 3.1) started to overfit the training data from around 20 epochs, at the point where the training curve started to grow much faster than the validation curve. The validation curve showed a slight average improvement until it stabilises in approximately 115 epochs. With the validation dataset, the average mean accuracy with the best models was around 42-48%, which matches the evaluation results with testing data afterwards.

[1] `https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d`

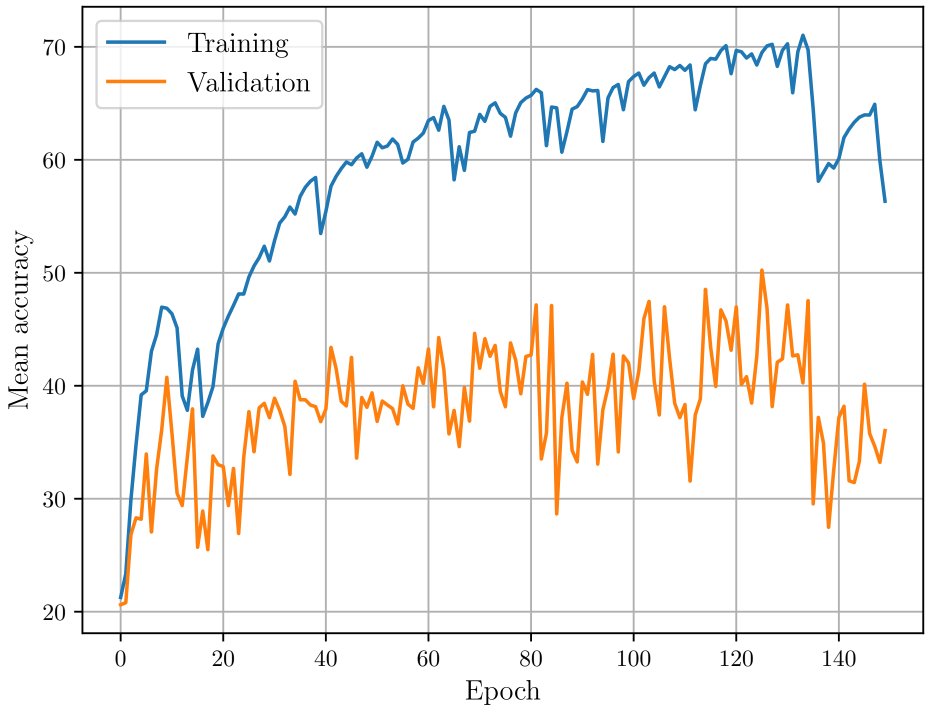

FIGURE 3.1: PointNet model training and validation mean accuracy curves averaged over five cross-validation folds. Based on the figure, the model can potentially have even higher mean accuracy and generalisation power, as the validation curve is much lower than training, referring to overfitting.

The five models were evaluated with mean accuracy, mean IoU and mean F1-score metrics. The mean accuracy results for each fold are given in figure 3.2. It is possible to see that the evaluation results fluctuate in relatively large intervals, between 32.69% and 59.89%. This might suggest that the generated dataset has unevenly spread classes in addition to an uneven representation of classes (Table 3.1), therefore some dataset splits had easier scenes, producing higher accuracy. For example, the bicycle and motorcycle class objects can be present in some scenes in the centre of the dataset but not on the edges. Therefore might not be selected for training and vice versa. This ensures that the decision to implement the K-fold strategy was correct for the model accuracy evaluation.

The mean accuracy value over all five evaluations was 44.15%, which expresses that the model performs significantly better than the dummy model, only predicting background class with a mean accuracy of 18.46%. This shows the model can generalise on the generated dataset, at least to some extent. The original PointNet paper [27] does use accuracy and mean IoU to represent the evaluation results. The accuracy metric is not directly comparable, as in the generated dataset, the classes were highly biased towards the background class. However, the paper results reveal that the original model was able to achieve 78.62% for accuracy, while the model running on the generated dataset achieved an accuracy of 93.89% averaged over five folds. This surpasses the PointNet paper's accuracy by a significant margin. The IoU metric results were much more comparable and close as well. With the generated dataset, the average mean IoU was 41.87% and in the original paper, it was 47.71%. The average mean F1-score for the models was 43.38%, which evaluates the combined score of the model's precision and recall.

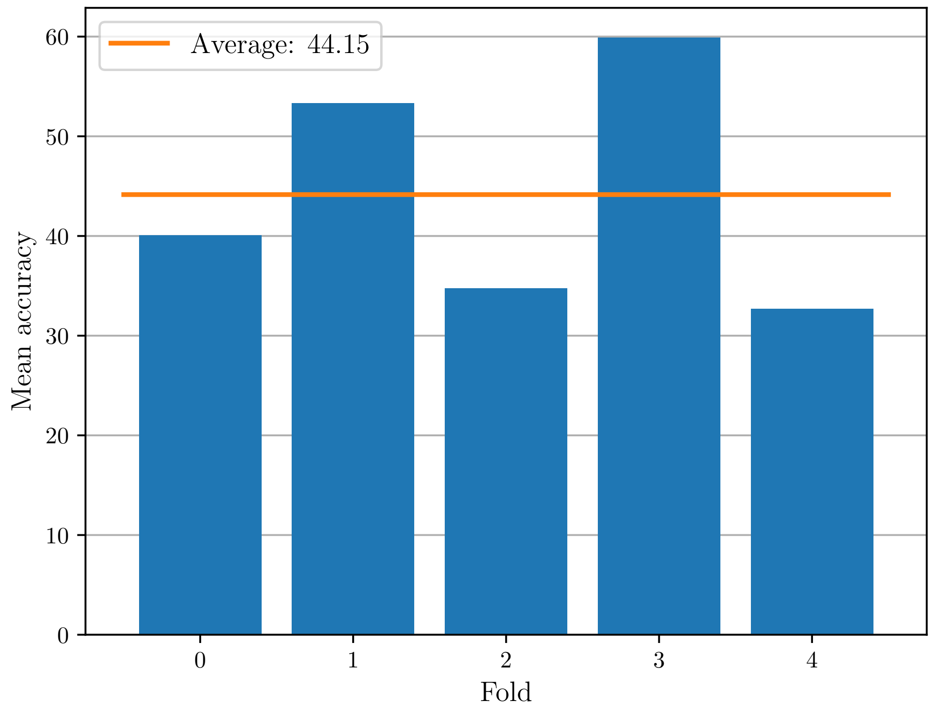

Figure 3.2: PointNet model evaluation mean accuracy for each cross-validation fold. The biggest mean accuracy difference is between folds 3 and 4, a total of 27.2%. The average value suggests the model performance when trained on the whole data and then deployed to a similar environment.

The model's fold training time was 20 minutes and 6 seconds with 8.04 seconds per epoch. The model inference speed was evaluated with the 0-fold model, which used the first 563 samples from the generated dataset as the testing data. All the rest was used for training and validation beforehand. The inference time (Figure 3.3) was measured with Python time module and directly before and after the *model.predict()* function call, as it replicates the real-world use case most accurately. On the other hand, the measured time might depend slightly on what the computer is doing in the background. Still, all other applications besides the Jupyter Notebook server were closed to minimise the risk. Based on the experiment, an average inference time of 30.03 milliseconds was measured, which makes the model run at 33.3 FPS. This can already be considered to be a real-time model for LightCode Photonics 3D camera, as it exceeds the currently expected maximum framerate of the camera. Furthermore, the model median inference time was 29.33 milliseconds, and the first inference time was 343.36 milliseconds. It was noted that the GPU usage during the inference was around 15% and power usage 120 W, which suggests that GPU computing capacity is not the bottleneck for the model inference speed. On the other hand, during training, PointNet was able to maximise the computing power of GPU by an average of 98% without significant fluctuations.

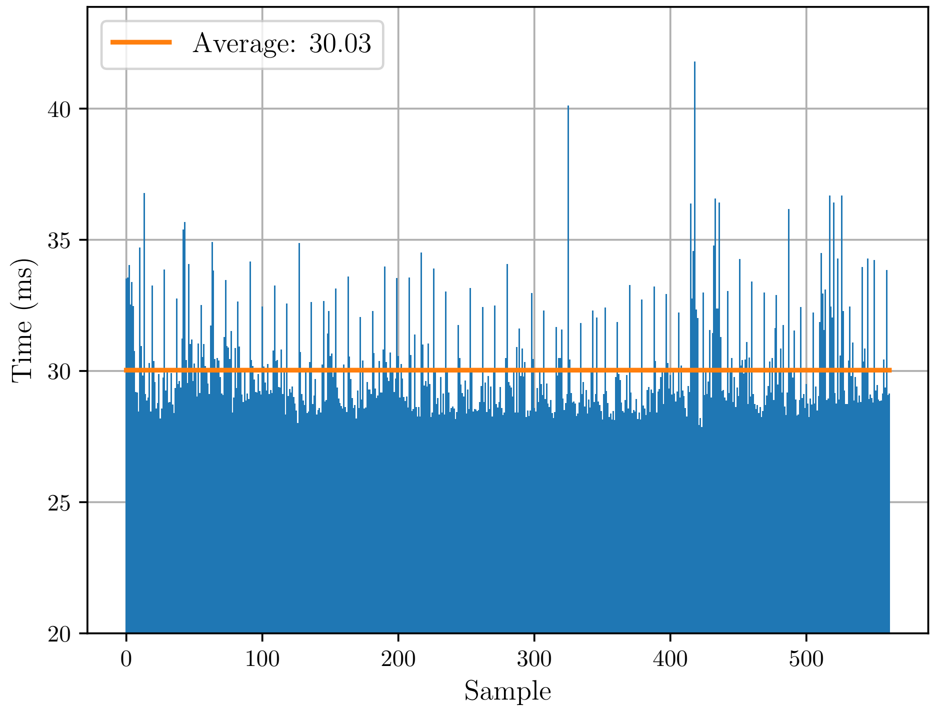

FIGURE 3.3: PointNet model inference time for 562 frames as the first inference time containing model warmup is excluded from the figure. Interestingly, there is a small peak in the inference time after approximately every eight predictions, even if the point clouds were predicted one at a time. It is suspected to represent some kind of context change in the GPU or Python garbage collection.

In conclusion, as the model is overfitting to the generated dataset and the mean accuracy could be increased even more, the next task would be to decrease the model size or add more regularisation to make it more generic. Another way to increase the model accuracy would be to train the model with class weight, depending on the class frequency of occurrence. The PointNet original paper [27] suggested that their model is capable of real-time operations, which was confirmed during the experiments with the generated dataset. In general, the PointNet model seems to be a promising semantic segmentation model to use with sparse point clouds on the actual 3D camera.

3.3 PointPillars

The model training consisted of training five new models on different data splits, while each model had 4.8M parameters to 61 epochs. The configuration file was taken from the KITTI dataset example and modified accordingly to the point cloud range and position of the bounding boxes to determine the area where the point cloud is converted to pillars. As this thesis aimed to evaluate the capability to run unchanged models on the generated dataset, as few parameters were changed as possible. The configuration file is in the appendix B.

The models' average bounding box training and validation loss curves are shown in figure 3.4, while the other two losses, directional loss and classification loss, behaved similarly. From the figure, it is possible to see that the model training loss is stably decreasing while the validation loss is relatively constant in time, confirming a significant overfit right from the start. The models were trained with a batch size of 6 and with 5000 steps per epoch, which means during one epoch, all the training data was processed 2.8 times by the model. This allows the insertion of more augmented bounding boxes into the training data, increasing the number of augmented samples and the effectiveness of the augmentation. The downside is that a large number of steps per epoch may increase the risk of model overfitting. To rule out the possibility that the step count caused the overfit, the 0-fold model was tested with two configurations, firstly with 5000 and secondly with 1802 steps. The experiment containing fewer steps per epoch produced more moderate training loss curves while having similar validation losses and, in some cases, even lower validation mAP. Therefore, it was decided to leave the parameter to its default configuration.

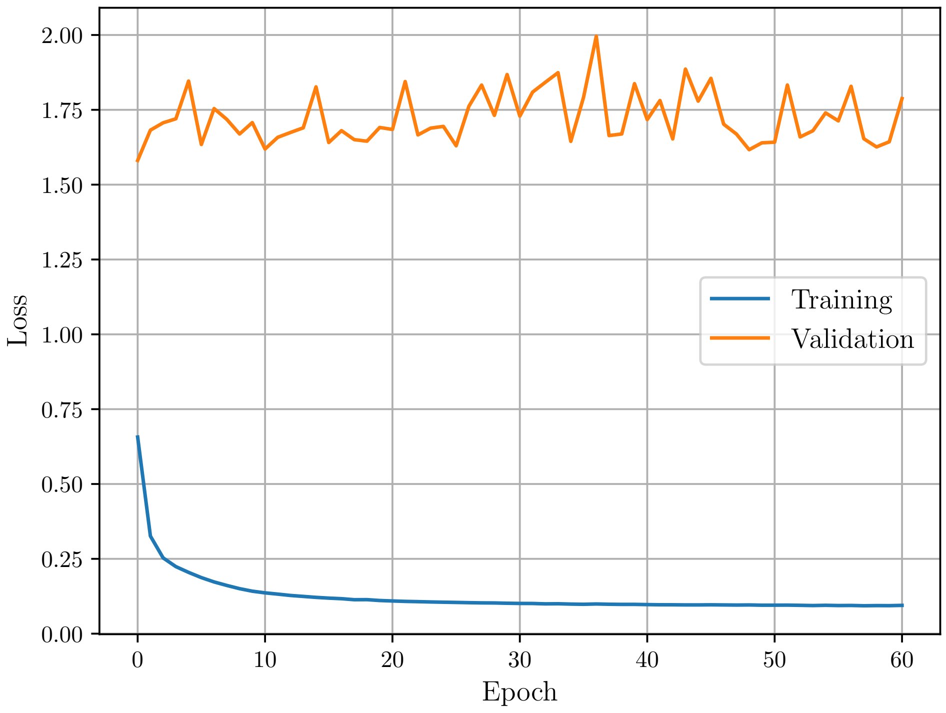

FIGURE 3.4: Averaged training and validation bounding box loss curves over five models. At the end of the training, the training loss is around 0.1, while the validation loss is approximately 1.7, which makes the validation loss 17 times larger than the training loss, referring to overfit.

As the model was heavily overfitting to the generated data, several techniques were tested to increase the model's performance. The hypothesis was that adding regularisation or changing the model size should make the model more generic and less prone to overfit the training data. The regularisation was added with a weight decay parameter passed to the optimiser. In the Tensorflow Open3D-ML implementation, it was essential to uncomment a section in the *ml3d/tf/models/point_pillars.py* (Appendix B) file in function *get_optimizer()* to make the weight decay parameter usable. In all the cases, the difference between training and validation loss stayed approximately in the same position. Therefore the thesis does not include an analysis of those experiments.

At this point, it was confirmed that the model works and can produce predictions with the KITTI 3D object detection dataset[2]. It achieved similar results described in the Open3D-ML GitHub [44] and the original paper [36]. This allows to believe that the selected implementation does not cause the overfitting problem but is tied to the generated dataset.

[2]`https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d`

The validation BEV and 3D mAP (Figure 3.5) showed an increasing trend during the training. The growth of BEV mAP slowed down approximately at epoch 30, while the same happened to 3D mAP around epoch 16. One of the possible reasons why the 3D mAP stabilised earlier is that the metric needed more precise bounding boxes, as the IoU evaluation also used the height and placement of the bounding box in the z-axis, suggesting that further learning would not reduce the error, as the model has exhausted its learning capacity on the dataset. The figure shows that both BEV and 3D mAP lines are fluctuating, which might indicate too few samples are available, therefore change in one prediction affects the mAP result significantly.

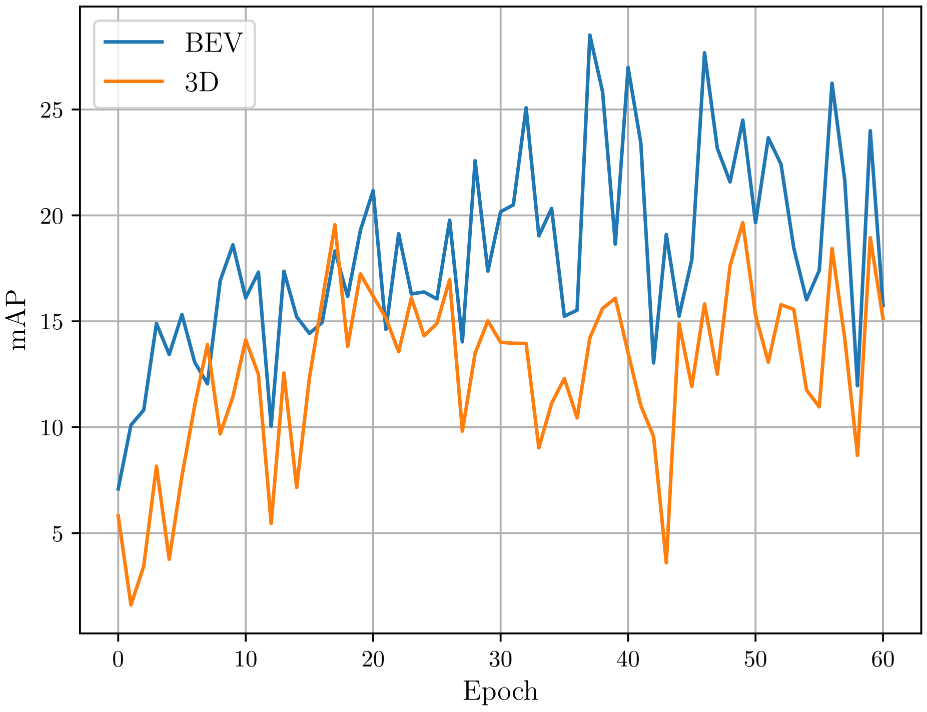

FIGURE 3.5: The validation BEV and 3D mAP, averaged over five models.

The model checkpoint, which produced the highest validation BEV mAP score, was chosen as the model to evaluate the accuracy performance of that fold. The evaluation results for five models are shown in figure 3.6. The average BEV and 3D mAP scores for detecting pedestrians were 30.54% and 26.0%, respectively, while mAPs for cars were 17.76% and 8.41%. The evaluation mAP scores for the pedestrian class were more stable than for the car class. One proposed hypothesis is that from the top view, the pedestrian bounding boxes are square-shaped, but the car class bounding boxes are rectangular, therefore, are more affected by the direction of the bounding box while evaluating the IoU. In addition, in a few cases, the BEV and 3D mAP were not correlating or had significant differences. For example, fold 3 car class BEV mAP is around 50% and 3D 25%, which is likely caused by having too few objects in the dataset, which aligns with the discovery made based on the validation data. Therefore, the change in one prediction affects the result in greater amplitude. Compared to KITTI dataset, the generated dataset had 2.8 times fewer objects per scene (Section 3.1). Furthermore, it was noted that in some cases, for example, fold 1 pedestrian scores, the BEV mAP score is lower than 3D mAP, which raised suspicions, as the IoU value for the bounding box in the 3D cannot be larger than in the BEV view, therefore no additional TP bounding boxes are present in the scene. Further investigation discovered that it directly comes from the mAP calculations, as different bounding box confidence levels are used for BEV and 3D average precision calculations. Lastly, the overall average score, including bicycle and motorcycle classes, is 13.61% for BEV and 8.6% for 3D mAP.

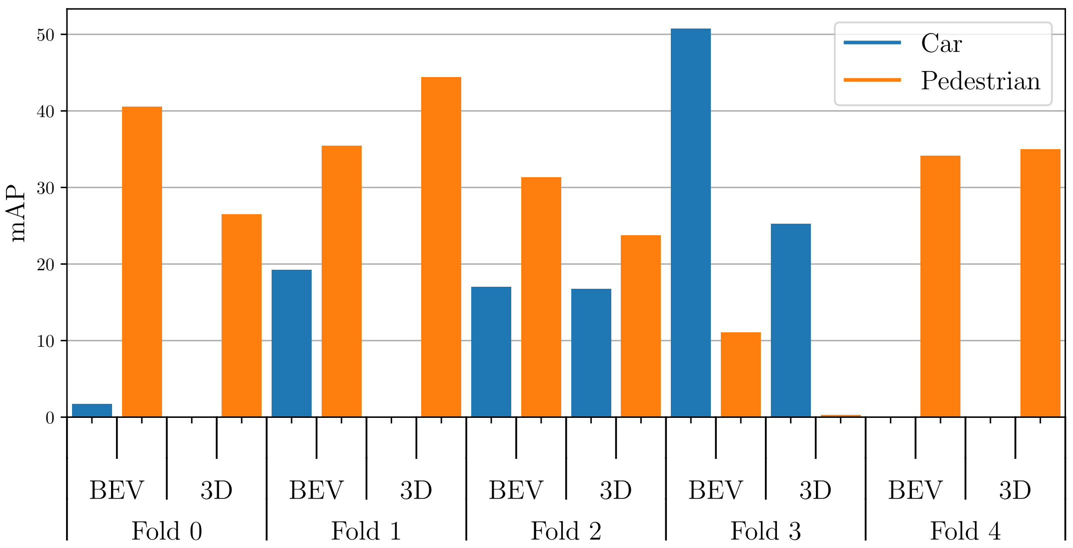

FIGURE 3.6: The evaluation BEV and 3D mAP for car and pedestrian classes for each fold. The missing columns represent a zero mAP value for that class. The bicycle and motorcycle classes are excluded from the figure as they produced close to zero mAP values in all folds.

The model's fold training time was 11 hours, 59 minutes and 53 seconds, which makes 11 minutes and 48 seconds for one epoch. As with PointNet, the 0-fold model was used to test the inference speed (Figure 3.7). The average inference time was 68.78 milliseconds, i.e. 14.5 FPS, significantly lower than the original PoitnPillars paper stated using Nvidia GTX 1080Ti for evaluations [36]. That might come down to the Open3D-ML PointPillars implementation, including post-processing steps, for example, non-maximum-suppression and how the code measures the time. Still, four times lower FPS was not expected while predicting bounding boxes, and further investigation must be conducted to validate the bottleneck for future use.

It was observed that during training, the model did not produce a constant load to the GPU, and the utilisation varied between 5-52%, which correlated with the training batch change. Therefore, suspected to be related to the small batch size. The GPU utilisation during inference was 21%, requiring 145 watts, therefore the computing power of the GPU was not the bottleneck for the model's low inference speed.

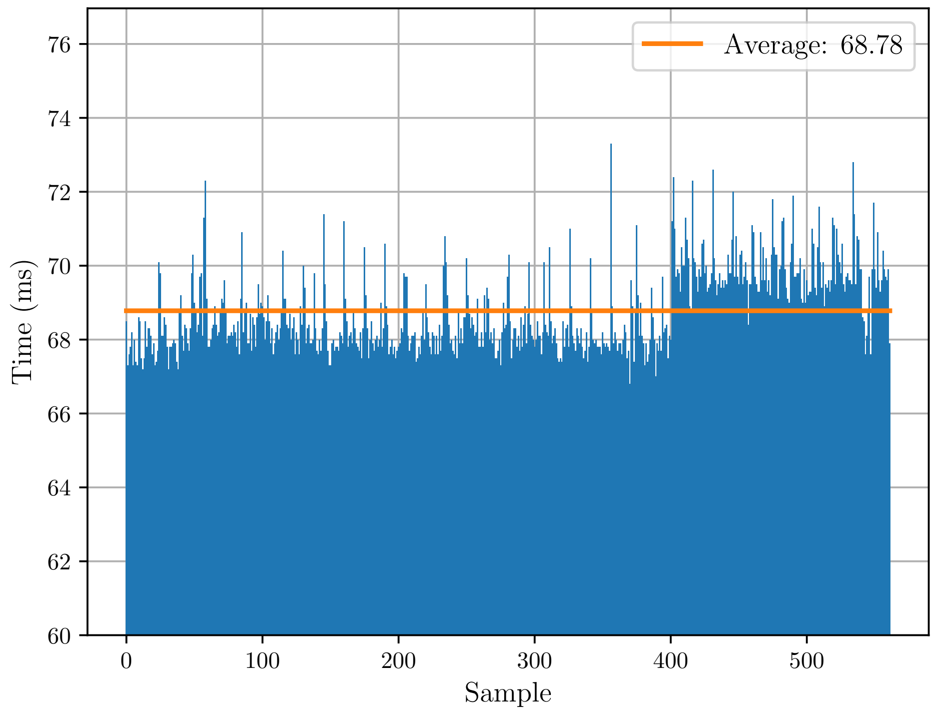

FIGURE 3.7: The inference times for 562 frames with the 0-fold model with an average value of 68.78 and median value of 68.38 milliseconds. The first frame inference time of 1.58 seconds has been excluded as it includes model warmup time.

It was suspected but not fully proven that during model training, a similar issue already reported[3] in the GitHub issues section appeared. The issue states that point clouds having an x-coordinate smaller than zero can produce false positive results randomly in the scene. Therefore additional steps were added to the data processing pipeline file *SeparateAndRotate.py*, discussed in section 2.2.3 to rotate the point clouds and transform their position.

In conclusion, the PointPillar model could not perform as expected, with a few possible reasons, such as too sparse data, missing intensity values or low object count. The data sparsity problem is relevant, as discussed in the PointPillars paper [36], the model sometimes cannot differentiate between pedestrians and other narrow vertical features, leading to false positive detections. The model can be further tested with new data containing intensity values with increased object count to decrease the overfitting problem.

[3]`https://github.com/isl-org/Open3D-ML/issues/551`

In addition, the inference time is much lower than reflected in the PointPillars paper [36], even if the system B (Section 2.1) parameters exceed the original setup by a significant margin. Lastly, one possible reason why PointNet model provided higher accuracy than PointPillars model with sparse data is that in semantic segmentation, the loss is calculated for each point, therefore yielding a higher training signal. On the other hand, in object detection, the loss is calculated for each object, but as the object count is smaller than the point count, the training signal is lower, decreasing the learning capacity with sparse data.

Outlook

The results of this thesis will be used in the two following projects in LightCode Photonics company. Firstly, to apply the same 3D object recognition models, PointNet and PointPillars, to LightCode Photonics camera-generated point cloud in a warehouse environment. A new dataset is being gathered to test models' performances in the mentioned environment, which allows to include additional features that were missing from the simulated data due to technical reasons, for example, intensity values, multiple returns and velocity map. Therefore, a performance increase is expected, especially with PointPillars model, as it was unable to show stable results on the simulated data.

Secondly, the created dataset could already be used for proof-of-concept 2D RGB and 3D point cloud fusion tasks, as it contains both point clouds and RGB images. The project objective would be to test the accuracy increase, as the RGB camera offers more semantic information compared to the point cloud. On the other hand, LightCode Photonics camera point cloud contains depth information and is more resilient to lighting conditions.

Conclusion

The importance of object recognition is increasing in the modern world, with numerous applications ranging from robotics to medical imaging; therefore, being one of the core visual research topics. Furthermore, imaging sensor manufacturers must continuously improve the data quality and compatibility with existing and future machine learning pipelines to improve the overall user experience, safety and application performance. This leads to the release of new imaging sensors, tested with existing state-of-the-art models to be competitive in the market.

The aim of the thesis was to evaluate the performance of machine learning-based object detection and semantic segmentation using a relatively low spatial resolution 3D camera in an urban environment. A hypothesis was formulated that using the camera-generated point for object recognition would produce at least similar accuracy to previously published original results. In addition, the thesis aimed to improve the company's understanding of its camera's capability for machine learning-based object recognition using only 3D point clouds and, with it, create a basis for future product development decisions.

In the theoretical part of the thesis, an overview of different 3D imaging sensor technologies was given, including their categorisation in depth and lateral resolution measurement, followed by an overview of object recognition-related subjects. To verify the hypothesis, a new dataset was generated in CARLA Simulator, compared to the data from the actual camera, due to the current technological state of the 3D camera. This required additional changes to CARLA Simulator source code to incorporate LightCode Photonics camera blueprint into the default sensor list, which also enables the use of the camera by other companies. The generated dataset was annotated with SUSTechPOINTS annotation tool and processed to prepare the data for object recognition models. Finally, PointNet semantic segmentation and PointPillars object detection models were modified according to the data layout and applied to the dataset. Furthermore, a k-fold cross-validation code was added to PointNet model together with evaluation pipeline changes in PointPillars model.

After the experiments with object recognition models, it was concluded that PointNet semantic segmentation model, which was applied to the generated dataset, could perform with similar accuracy to the original model. On the other hand, PointPillars model was not able to produce stable predictions with the dataset. It was suspected that the instability might come from a low number of training samples, data sparsity or the lack of correct intensity values. Furthermore, both models did overfit the dataset, allowing to believe that improvements to the accuracy can still be made. During testing PointNet model achieved an inference speed of 33.3 FPS, while PointPillars only produced 14.5 FPS.

The LightCode Photonics 3D camera is a promising future imaging technology, and most likely, many of its applications will be tied to machine learning and object recognition. At the time of writing, the future specifications of the camera have changed, already using some of the knowledge and feedback from this thesis. The work on improving the object recognition models continues. Furthermore, due to the sparsity of data, incorporating 2D RGB with 3D point clouds is a promising architecture to improve the overall object recognition performance.

Bibliography

[1] Alexandros Iosifidis and Anastasios Tefas, editors. *Deep learning for robot perception and cognition.* Academic Press, San Diego, CA, 2022.

[2] Eman Ahmed, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Bjorn Ottersten. A survey on Deep Learning Advances on Different 3D Data Representations. *arXiv:1808.01462*, 2019.

[3] Jing Li, Rui Li, Jiehao Li, Junzheng Wang, Qingbin Wu, and Xu Liu. Dual-view 3D object recognition and detection via Lidar point cloud and camera image. *Robotics and Autonomous Systems*, 150:103999, 2022. doi: 10.1016/j.robot.2021.103999.

[4] Germán Mora-Martín, Alex Turpin, Alice Ruget, Abderrahim Halimi, Robert Henderson, Jonathan Leach, and Istvan Gyongy. High-speed object detection with a single-photon time-of-flight image sensor. *Optics Express*, 29(21):33184, 2021. doi: 10.1364/OE.435619.

[5] Istvan Gyongy, Neale A. W. Dutton, and Robert K. Henderson. Direct Time-of-Flight Single-Photon Imaging. *IEEE Transactions on Electron Devices*, 69(6):2794–2805, 2022. doi: 10.1109/TED.2021.3131430.

[6] Istvan Gyongy, Germán Mora Martín, Alex Turpin, Alice Ruget, Abderrahim Halimi, Robert K. Henderson, and Jonathan Leach. High-speed vision with a 3D-stacked SPAD image sensor. In Mark A. Itzler, K. Alex McIntosh, and Joshua C. Bienfang, editors, *Advanced Photon Counting Techniques XV*, volume 11721, page 1172105. SPIE, 2021. doi: 10.1117/12.2586883.

[7] Cyrus Bamji, John Godbaz, Minseok Oh, Swati Mehta, Andrew Payne, Sergio Ortiz, Satyadev Nagaraja, Travis Perry, and Barry Thompson. A Review of Indirect Time-of-Flight Technologies. *IEEE Transactions on Electron Devices*, 69(6):2779–2793, 2022. doi: 10.1109/TED.2022.3145762.

[8] Federica Villa, Fabio Severini, Francesca Madonini, and Franco Zappa. SPADs and SiPMs Arrays for Long-Range High-Speed Light Detection and Ranging (LiDAR). *Sensors*, 21(11):3839, 2021. doi: 10.3390/s21113839.

[9] François Piron, Daniel Morrison, Mehmet Rasit Yuce, and Jean-Michel Redouté. A Review of Single-Photon Avalanche Diode Time-of-Flight Imaging Sensor Arrays. *IEEE Sensors Journal*, 21(11):12654–12666, 2021. doi: 10.1109/JSEN.2020.3039362.

[10] *LIDAR Pulsed Time of Flight Reference Design (Rev. B)*. Texas Instruments, 2016. URL `https://www.ti.com/lit/ug/tiducm1b/tiducm1b.pdf`. Accessed: 2023-04-13.

[11] Santiago Royo and Maria Ballesta-Garcia. An Overview of Lidar Imaging Systems for Autonomous Vehicles. *Applied Sciences*, 9(19):4093, 2019. doi: 10.3390/app9194093.

[12] Silvio Giancola, Matteo Valenti, and Remo Sala. *A Survey on 3D Cameras: Metrological Comparison of Time-of-Flight, Structured-Light and Active Stereoscopy Technologies*. Springer International Publishing, 2018. doi: 10.1007/978-3-319-91761-0.

[13] *CE30-C Solid State Array LiDAR Operation Manual*. Benewake, s.a. URL `https://www.manualslib.com/manual/1502459/Benewake-Ce30-C.html`. Accessed: 2023-04-13.

[14] Paul McManamon. *Field guide to lidar*. SPIE Press, 2015. doi: 10.1117/3.2186106.

[15] Thinal Raj, Fazida Hanim Hashim, Aqilah Baseri Huddin, Mohd Faisal Ibrahim, and Aini Hussain. A Survey on LiDAR Scanning Mechanisms. *Electronics*, 9(5):741, 2020. doi: 10.3390/electronics9050741.

[16] Dingkang Wang, Stephan Strassle Rojas, Alexander Shuping, Zaid Tasneem, Sanjeev Koppal, and Huikai Xie. An Integrated Forward-View 2-Axis Mems Scanner for Compact 3D Lidar. In *2018 IEEE 13th Annual International Conference on Nano/Micro Engineered and Molecular Systems (NEMS)*, pages 185–188. IEEE, 2018. doi: 10.1109/NEMS.2018.8557009.

[17] Enrique Marti, Miguel Angel de Miguel, Fernando Garcia, and Joshue Perez. A Review of Sensor Technologies for Perception in Automated Driving. *IEEE Intelligent Transportation Systems Magazine*, 11(4):94–108, 2019. doi: 10.1109/MITS.2019.2907630.

[18] Kai Yit Kok and Parvathy Rajendran. A Review on Stereo Vision Algorithm: Challenges and Solutions. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 13(2):112–128, 2020. doi: 10.37936/ecti-cit.2019132.194324.

[19] Abdelmoghit Zaarane, Ibtissam Slimani, Wahban Al Okaishi, Issam Atouf, and Abdellatif Hamdoun. Distance measurement system for autonomous vehicles using stereo camera. *Array*, 5:100016, 2020. doi: 10.1016/j.array.2020.100016.

[20] *Direct Time-of-Flight Depth Sensing Reference Designs*. OnSemi, 2021. URL `https://www.onsemi.com/pub/Collateral/TND6341-D.PDF`. Accessed: 2023-04-13.

[21] Asim Bhatti, editor. *Stereo Vision*. InTech, 2008. doi: 10.5772/89.

[22] Daniel Ruiz, Gabriel Salomon, and Eduardo Todt. Can giraffes become birds? an evaluation of image-to-image translation for data generation. In *Anais do XI Computer on the Beach - COTB '20*. Universidade do Vale do Itajaí, 2020. doi: 10.14210/cotb.v11n1.p176-182.

[23] Ming Zhu, Chao Ma, Pan Ji, and Xiaokang Yang. Cross-Modality 3D Object Detection. In *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 3771–3780. IEEE, 2021. doi: 10.1109/WACV48630.2021.00382.

[24] Wentao Chen, Wei Tian, Xiang Xie, and Wilhelm Stork. RGB Image- and Lidar-Based 3D Object Detection Under Multiple Lighting Scenarios. *Automotive Innovation*, 5(3):251–259, 2022. doi: 10.1007/s42154-022-00176-2.

[25] Lin Bai, Yiming Zhao, and Xinming Huang. Enabling 3-D Object Detection With a Low-Resolution LiDAR. *IEEE Embedded Systems Letters*, 14(4):163–166, 2022. doi: 10.1109/LES.2022.3170298.

[26] Ruddy Théodose, Dieumet Denis, Thierry Chateau, Vincent Fremont, and Paul Checchin. A Deep Learning Approach for LiDAR Resolution-Agnostic Object Detection. *IEEE Transactions on Intelligent Transportation Systems*, 23(9):14582–14593, 2022. doi: 10.1109/TITS.2021.3130487.

[27] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *arXiv:1612.00593*, 2016.

[28] Yunze Man, Xinshuo Weng, Prasanna Kumar Sivakumar, Matthew O'Toole, and Kris Kitani. Multi-Echo LiDAR for 3D Object Detection. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3743–3752. IEEE, 2021. doi: 10.1109/ICCV48922.2021.00374.

[29] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv:1804.02767*, 2018.

[30] Bence Major, Daniel Fontijne, Amin Ansari, Ravi Teja Sukhavasi, Radhika Gowaikar, Michael Hamilton, Sean Lee, Slawomir Grzechnik, and Sundar Subramanian. Vehicle Detection With Automotive Radar Using Deep Learning on Range-Azimuth-Doppler Tensors. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 924–932. IEEE, 2019. doi: 10.1109/ICCVW.2019.00121.

[31] Di Feng, Christian Haase-Schütz, Lars Rosenbaum, Heinz Hertlein, Claudius Gläser, Fabian Timm, Werner Wiesbeck, and Klaus Dietmayer. Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges. *IEEE Transactions on Intelligent Transportation Systems*, 22(3): 1341–1360, 2021. doi: 10.1109/TITS.2020.2972974.

[32] Jian Dong, Qiang Chen, Shuicheng Yan, and Alan Yuille. Towards Unified Object Detection and Semantic Segmentation. In *Computer Vision – ECCV 2014*, pages 299–314. Springer International Publishing, 2014. doi: 10.1007/978-3-319-10602-1_20.

[33] François Chollet. *Deep learning with Python.* Manning Publications, New York, NY, second edition, 2021.

[34] Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv:1706.02413*, 2017.

[35] Soumik Rakshit and Sayak Paul. Keras documentation: Point cloud segmentation with PointNet, 2020. URL `https://keras.io/examples/vision/pointnet_segmentation/`. Accessed: 2023-04-06.

[36] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. PointPillars: Fast Encoders for Object Detection From Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12689–12697. IEEE, 2019. doi: 10.1109/CVPR.2019.01298.

[37] Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. A survey on performance metrics for object-detection algorithms. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 237–242. IEEE, 2020. doi: 10.1109/iwssip48289.2020.9145130.

[38] Eduardo Fernandez-Moral, Renato Martins, Denis Wolf, and Patrick Rives. A new metric for evaluating semantic segmentation: Leveraging global and contour accuracy. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1051–1056. IEEE, 2018. doi: 10.1109/ivs.2018.8500497.

[39] Zoltan Rozsa and Tamas Sziranyi. Object Detection From a Few LIDAR Scanning Planes. *IEEE Transactions on Intelligent Vehicles*, 4(4):548–560, 2019. doi: 10.1109/TIV.2019.2938109.

[40] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16. PMLR, 2017.

[41] carla, 2021. URL `https://github.com/carla-simulator/carla/tree/0.9.13`. Accessed: 2023-04-13.

[42] E Li, Shuaijun Wang, Chengyang Li, Dachuan Li, Xiangbin Wu, and Qi Hao. SUSTech POINTS: A Portable 3D Point Cloud Interactive Annotation Platform System. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1108–1115. IEEE, 2020. doi: 10.1109/IV47402.2020.9304562.

[43] SUSTechPOINTS, 2022. URL `https://github.com/naurril/SUSTechPOINTS/tree/522d1daef1e829f433cada6f420189ab3daee249`. Accessed: 2023-04-13.

[44] Open3D-ML, 2022. URL `https://github.com/isl-org/Open3D-ML/tree/e469075d4450bb73458671e8e42c60a203b6a908`. Accessed: 2023-04-13.

[45] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

[46] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012. doi: 10.1109/CVPR.2012.6248074.

Appendix A – Modified CARLA Simulator Server Code

The code is in GitHub repository: `https://github.com/TimoTiirats/carla_lightcode/tree/lightcode`

Appendix B – Modified Open3D-ML Point-Pillars Code

The code is in the GitHub repository: `https://github.com/TimoTiirats/Open3D-ML/tree/lightcode`

Appendix C – Python Codes and Dataset

Python and generated dataset files are attached to this thesis in a *AppendixC.zip* file with the following layout:

- *Carla/* - includes Carla Simulator client code
  - *CarlaClient.py*
- *Processing/* - includes data processing pipeline Python files
  - *SeparateAndRotate.py*
  - *FilterFrames.py*
  - *GenerateIntensityValues.py*
- *PointNet/* - includes the PointNet model files
  - *PointNet_K-Fold.ipynb*
  - *metadata.json*
- *Dataset.zip* - includes the generated raw and processed datasets