Eskild Brobak
Peder Espen

Training object detection models for AVs using simulated RGB and LiDAR image data

Master's thesis in Informatics
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Eskild Brobak
Peder Espen

Training object detection models for AVs using simulated RGB and LiDAR image data

Master's thesis in Informatics
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Abstract

Interest and investment in autonomous vehicle research has increased rapidly in the last decade. A major roadblock which has received increased attention is the area of object detection: making vehicles able to see and understand their surroundings using various tools like cameras and deep learning algorithms.

To implement object detection, algorithms can be trained using annotated data. These algorithms usually require large datasets to perform adequately, which is typically images hand-annotated by humans. This is a costly and time-consuming practice. Using images from a simulated environment in place of real images could help mediate the cost and time spent significantly. With a simulated environment one has full control over all objects, which enables automatic annotation of images. One such simulator is Carla, an open source project specifically developed for research around autonomous driving.

This thesis investigates whether automatically annotated simulator data from Carla can be used to train object detection models which can detect real life traffic objects. As a large part of the thesis revolves around collecting properly automatically annotated training data from the simulator, it also investigates whether tightness is important for the bounding boxes. In the first six experiments, the results show that Carla data alone is not sufficient to properly train the chosen models. However, fine-tuning a Carla trained model using some real data shows promising results. The results from the seventh experiment indicate that bounding box tightness is important for training the models properly.

Sammendrag

Interesse og investering i forskning rundt autonome kjøretøy har økt kraftig det siste tiåret. En stor hindring som har fått økt oppmerksomhet er området objekt-deteksjon: få kjøretøy til å se og forstå deres omgivelser ved å bruke diverse verktøy som kameraer og dyp-læring algoritmer.

For å implementere objekt-deteksjon kan algoritmer bli trent ved å bruke annotert data. Disse algoritmene krever vanligvis store datasett for å oppnå god nok ytelse, som vanligvis er bilder hånd-annotert av mennesker. Dette er en kostbar og tidkrevende øvelse. Å bruke bilder fra et simulert miljø i stedet for ekte bilder kunne redusert kostnader og tidsbruk betydelig. Med et simulert miljø har man full kontroll over alle objekter, som gjør det mulig å automatisk annotere bilder. En slik simulator er Carla, et prosjekt med åpen kildekode som er spesifikt utviklet for forskning rundt autonome kjøretøy.

Denne avhandlingen undersøker om automatisk annotert simulert data fra Carla kan brukes for å trene objekt-deteksjon modeller som kan detektere ekte trafikk-objekter. Siden en stor del av oppgaven handler om å samle inn automatisk annotert trenings-data fra simulatoren, undersøker avhandlingen også hvor viktig tettheten til avgrensningsboksene er. I de første seks eksperimentene viser resultatene at data fra Carla alene ikke er nok til å trene de utvalgte modellene ordentlig. Å bruke noe ekte data til å finpusse en modell som hovedsakelig er trent på Carla ga lovende resultater. Resultatene fra det syvende eksperimentet indikerte at tettheten til avgrensningsboksene var viktig for å trene modellene ordentlig.

Preface

This thesis is a part of the research conducted within the NTNU Autonomous Perception Laboratory (NAPLab).

We would like to thank our supervisor Frank Lindseth for his guidance and help with the thesis. We would also like to thank his co-supervisor Gabriel Kiss for providing assistance with the technical aspects of the thesis.

Contents

1 Introduction . . . **1**

1.1 Background and Motivation . . . 1

1.2 Research Goal and Questions . . . 3

1.3 Contributions . . . 4

1.4 Report Structure . . . 5

2 Background and related work . . . **6**

2.1 Deep Learning . . . 6

2.1.1 Convolutional Neural Networks . . . 6

2.1.2 Overfitting a model . . . 7

2.1.3 Transfer learning . . . 8

2.2 Computer Vision . . . 8

2.2.1 Object detection . . . 9

2.2.2 Semantic and instance segmentation . . . 10

2.2.3 Mean Average Precision . . . 11

2.3 Object detection models . . . 12

2.3.1 Yolov5 . . . 12

2.3.2 Faster R-CNN 14
2.4 LiDAR 16
2.5 Carla Simulator 16
2.5.1 Python API 18
2.5.2 Relevant python packages 18
2.5.3 Types of sensors 19
2.5.4 Carla Client bounding boxes 20
2.5.5 Ouster LiDAR sensor and LiDAR images 20
2.6 Related Work 22
3 Methodology **25**
3.1 Fixing the Carla instance segmentation LiDAR sensor 25
3.2 Collecting simulator data and creating datasets 27
3.2.1 Defining dynamic object types 27
3.2.2 Methods of gathering sensor data 28
3.2.3 Extracting RGB data from Carla 29
3.2.4 Generating RGB images 36
3.2.5 Extracting LiDAR data from Carla 41
3.2.6 Generating LiDAR images 46
3.2.7 Data collection pipeline 53
3.3 Training and testing object detection models 60
3.3.1 Yolov5 60
3.3.2 Faster R-CNN via MMDetection 63

3.3.3 Metric evaluation 66

4 Experiments & Results 67

4.1 Training baseline models 67

4.2 Experiment 1: Small RGB dataset, default settings 68

4.3 Experiment 2: Increase number of epochs 69

4.4 Experiment 3: Increase dataset size 71

4.5 Experiment 4: Fine-tuning with Kitti images 72

4.6 Experiment 5: Fine-tuning on pre-trained COCO models 73

4.7 Experiment 6: LiDAR images from scratch 74

4.8 Experiment 7: Fine-tuning with LiDAR images 75

4.9 Experiment 8: Comparing bounding box tightness 76

5 Discussion 78

5.1 Object detection in RGB images 78

5.1.1 Research Question 1 78

5.1.2 Research Question 2 80

5.2 Object detection in LiDAR images 81

5.2.1 Research Question 3 81

5.2.2 Research Question 4 82

5.3 Bounding box tightness 83

5.4 Shortcomings 84

5.5 Reflections 84

6 Conclusion and Future work 86

6.1 Conclusion 86
6.2 Future Work 87
A Faster R-CNN **93**

Figures

2.1 CNN architecture. Source: Keiron O'Shea, Ryan Nash[9] 7

2.2 2D bounding box (left) vs 3D bounding box (right). Source: https://towardsdatascience.com/orientation-estimation-in-monocular-3d-object-detection-f850ace91411 9

2.3 A scene processed using semantic and instance segmentation. Source: https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339 10

2.4 Yolov5 architecture. Source: https://github.com/ultralytics/yolov5/issues/280 13

2.5 Faster R-CNN architecture. source: https://blog.paperspace.com/faster-r-cnn-explained-object-detection/ 15

2.6 Carla Vehicle bounding box example. Source: https://carla.readthedocs.io/en/0.9.5/measurements/ 20

2.7 Example of Ouster LiDAR images. Top is range, middle intensity and bottom ambient. Source: https://ouster.com/blog/the-camera-is-in-the-lidar/ 21

3.1 Code snippet from ActorRegistry.cpp 26

3.2 Carla PythonAPI Example 28

3.3 Pygame window in Carla 29

3.4 Synchronous loop from ExtractRGB.py 30

3.5 Camera calibration code snippet from client_bounding_boxes.py . . 31

3.6 Example of a bounding box text file 32

3.7 Part of the .json file used for labeling objects 33

3.8 Example of bounding boxes being too large 35

3.9 Example of cyclist bounding box in Carla 36

3.10 Code snippet for converting the actor ID into BGR colors 37

3.11 Pipeline showing the extraction of the G and B channels of an instance segmentation image, and how bit-shifting is used to match the ID of the object bounding box. 38

3.12 Code snippet for filtering out objects not in the scene 39

3.13 Occluded bounding box example 39

3.14 Code snippet showing how the bounding boxes where tightened . . 40

3.15 Example of bounding box stretching on the edge of the image . . . 41

3.16 Code snippet from LidarExtract.py 42

3.17 SaveActorLabels function 43

3.18 Example of a .ply file generated from the semantic LiDAR sensor . . 45

3.19 processPointCloud() function 49

3.20 Example range image (top) and intensity image (bottom) 50

3.21 Lidar range image without bounding box threshold (top) vs with threshold (bottom) 52

3.22 Segmentation LiDAR image showing the shifted bounding boxes . . 52

3.23 Python script to count the number of different labels for the project datasets 57

3.24 Class imbalance for 500 image test dataset 57

3.25 Blueprint lists from spawn.py 58

3.26 Class imbalance for 500 image test dataset with balanced spawning 59

3.27 Class imbalance for the Carla LiDAR training dataset (4000 images) 59

3.28 Yolov5 Pytorch dataset structure (top) and .yaml file contents (bottom) 61

3.29 Differences between Yolov5 variations. Source: `https://github.com/ultralytics/yolov5` 62

3.30 Folder structure of the COCO format used in MMDetection Faster R-CNN 64

3.31 Example of a image information within a .json file 64

3.32 Example of an object and its information within a .json file 65

5.1 Intensity images from Carla (top) vs Ouster (bottom) 82

A.1 Class structure inside of a .json annotation file used in MMDetection Faster R-CNN. 94

Acronyms

AI Artificial Intelligence. 8

ANN Artificial Neural Network. 6

AP Average Precision. 11, 12, 61, 68, 79, 83, 84

AV Autonomous Vehicles. 1, 2

CNN Convolutional Neural Network. x, 6–8, 12, 13

CV Computer Vision. 3, 6, 8, 9, 24, 56

IoU Intersect over Union. 12, 68, 87

LiDAR Light Detection and Ranging. x, xi, 4, 6, 16, 19–22, 28, 41–43, 45, 46, 48, 50–55, 59, 60, 67, 74, 75, 81–84, 88

mAP Mean Average Precision. 11, 12, 66, 68, 74, 79, 83

ML Machine Learning. 3, 12, 17

RBN Region Based Network. 14–16

ReLu Rectified Linear Unit. 7

VM Virtual Machine. 17

Yolo You Only Look Once. 12, 13

Chapter 1

Introduction

This chapter has four sections. Section 1.1 explains the background and motivation behind the report. Section 1.2 describes the goal and research questions that form the basis of the thesis. Section 1.3 talks about the contribution this paper makes. Section 1.4 summarizes the structure of the report.

1.1 Background and Motivation

According to a study published in 2018 by The World Health Organization, traffic accidents was the leading cause of deaths among children and young adults aged 5-29 years old [1]. The same report also stated that an estimated 1.35 million people die every year in various traffic related accidents. Another study by the U.S. Department of Transportation published in 2017 claimed that between 94-96% of all traffic accidents were caused by human error [2]. As more aspects of society is being automated, Autonomous Vehicles (AV) might be a solution which can help reduce these numbers.

There are additional benefits to AVs. According to Zhong et al, AVs could reduce commuters' value of travel time by 18-32% [3], allowing passengers to spend their time on activities other than driving, while reducing time spent on commutes. A study by Stern et al. stated that having 5% of a vehicle fleet be AVs designed to

help stabilize traffic flow could help reduce the overall CO2-emissions of the fleet by 15% [4]. For Europe, roughly 22% of total CO2 emissions comes from road traffic [5], so implementing AVs in existing traffic solutions could help significantly reduce emissions.

A central part of making cars autonomous is making them able to observe their surroundings. A human driver has to see and understand a large variety of different types of objects that can appear on the road, like other cars, pedestrians or other road users like cyclists. Making a vehicle able to accurately detect these objects is a vital part of making autonomous vehicles viable. This paper focuses on a specific aspect of this expansive and evolving topic: object detection of dynamic traffic objects. In this paper, dynamic traffic objects are defined as traffic objects which can move, like cars or pedestrians.

Object detection in images can be summed up in two questions:

- **What kind of object is it?** You want to identify different objects in the image. This is typically done by separating objects into classes or labels, such as cars, dogs, people etc. Also called **Image Classification**.
- **Where is the object?** Where the detected objects are located in the image. This is especially important for autonomous driving, as knowing your surroundings is essential to be able to predict and react to dynamic situations. Also called **Object Localization**.

One of the challenges when training a model in object detection is having a big enough dataset. For example, the ImageNet dataset contains over 14 million images divided into roughly 22,000 categories [1]. When training an object detection model, the model needs annotated data in the form of labels which specifies where objects are located in the images. This is done by using bounding boxes, a rectangle that determines the spatial location of the object in the image. Traditionally, images are manually annotated by hand, which is time consuming and repetitive.

The first part of this project explores whether a simulator can be used to collect photo-realistic images that can be precisely automatically annotated. Building on

[1] https://www.image-net.org/

previous work by Jang et al. [6], an open-source autonomous driving simulator named Carla (Car Learning to Act) is used to collect automatically annotated images. They created scripts for collecting RGB images from the simulation, along with 3D bounding boxes. These 3D bounding boxes were then converted into 2D, using additional information to correct the boxes.

The bounding boxes collected using the scripts from the paper showed to often be inaccurate, being larger than the boundaries of the objects. A paper by Kervadec et al. published in 2020 showed the importance of tight bounding boxes in a Machine Learning (ML) scenario [7]. Also, the original project only separated objects into two categories: vehicles and pedestrians. This limited the variety of the dataset, as Carla contains several sub-types of spawnable objects such as trucks and cyclists. The team, being particularly interested in Computer Vision (CV) and related subjects, were interested in improving the groundwork laid by the paper to produce scripts capable of making more robust and balanced datasets. This would be done in part by implementing the new **Instance Segmentation Image sensor** added in the then newest version of Carla (0.9.13) to achieve more precise bounding boxes.

In order to test whether the collected data is valuable for object detection tasks, the second part of the project explores if the generated datasets can be used to train a model to detect traffic objects in real life images. Two object detection models (Yolov5 and Faster R-CNN) were chosen to get comparative results, and several experiments were conducted with various parameters. The models were tested against the Kitti dataset [2], a popular dataset for traffic detection.

1.2 Research Goal and Questions

The research goal for this paper is:

> **Investigate whether automatically annotated images from the Carla simulator can be used to train object detection models for real traffic scenarios.**

[2] `http://www.cvlibs.net/datasets/kitti/eval_object.php`

To achieve this goal, three research questions were posed:

- **RQ 1:** How well can an object detection model detect real traffic objects when trained on simulated RGB image data collected from the Carla simulator?
- **RQ 2:** Can RGB image data collected from the Carla simulator be used to reduce the amount of real images required by fine-tuning an object detection model trained primarily on the simulated data?
- **RQ 3:** How well can an object detection model detect real traffic objects in LiDAR images when trained on simulated LiDAR image data?
- **RQ 4:** To what degree can simulated LiDAR image data be used to reduce the number of real LiDAR images required by fine-tuning a model pre-trained on the simulated data?
- **RQ 5:** To what degree do tight bounding boxes improve the detection of objects?

1.3 Contributions

While using simulator data to train object detection models for real life usage is not a new concept, there is not a great amount of research using Carla specifically. Particularly using the included 3D bounding boxes from the simulator and projecting them onto 2D images. The CarFree project is one example, which was improved upon during this project [6]. The bounding boxes produced by the script were improved by tightening as well as extending the boxes where the objects were partly occluded. This improved model performance significantly.

Training object detection models solely on Carla data did not produce viable results. Different model variations like increasing training time or dataset size did not significantly impact the results either. However, the RGB and LiDAR models trained on primarily Carla data (with fine-tuning using a small real dataset) showed promising results, getting an improved score compared to solely using the small real dataset. While the models did not achieve similar performance as their respective baselines, the improvement suggests that the data collected from Carla

could be used as a majority of the training data, which reduces time and cost of annotating by hand.

1.4 Report Structure

- **Introduction** discusses the practical problems and motivations as to why this topic is worthwhile doing research on.
- **Background** presents the theoretical background for the different aspects of the thesis. Related work will also be presented in this chapter.
- **Methodology** is split into two parts. The first part explores whether a simulator can be used to collect photo-realistic images that can be precisely automatically annotated. The second part of the project looks at whether this dataset can be used to train models to detect dynamic traffic objects in real life images.
- **Experiments and Results** presents the different experiments that were done, along with their results.
- **Discussion** elaborates and discusses the different results from the experiments. This is done in part by addressing the research questions. Shortcomings of the thesis are also discussed.
- **Conclusion** will conclude the thesis and present the most important findings, along with future work that could be done.

Chapter 2

Background and related work

This chapter details the most relevant theory behind the thesis. Section 2.1 describes concepts like Convolutional Neural Network (CNN)s, neural networks and transfer learning. Section 2.2 explains the most relevant theory in Computer Vision (CV). Section 2.3 talks about the two object detection models chosen for the thesis. Section 2.4 briefly explains the theory behind how a LiDAR scanner works. Section 2.5 looks into the most relevant mechanisms of the Carla simulator. Finally, section 2.6 investigates related work in the form of previously published papers.

2.1 Deep Learning

2.1.1 Convolutional Neural Networks

A popular method of implementing object detection is through using **CNN**s. The concept gained recognition when Alex Krizhevsky and his team won the ImageNet classification challenge by a substantial margin in 2012 [8]. CNNs specialize in analyzing data that have a grid-like structure, such as images. They are built on the idea of an Artificial Neural Network (ANN) which are networks inspired by how the nervous system and the human brain functions [9]. CNNs are built using a high number of neurons [9]. These neurons receive an input and then performs

an operation based on that input. From the initial input of raw image vectors to the final output, the weight of the network will be expressed as a score function.

The CNN architecture is consists of primarily three types of layers: the convolutional layer, the pooling layer and the fully connected layer [9]. This is illustrated in image 2.1.

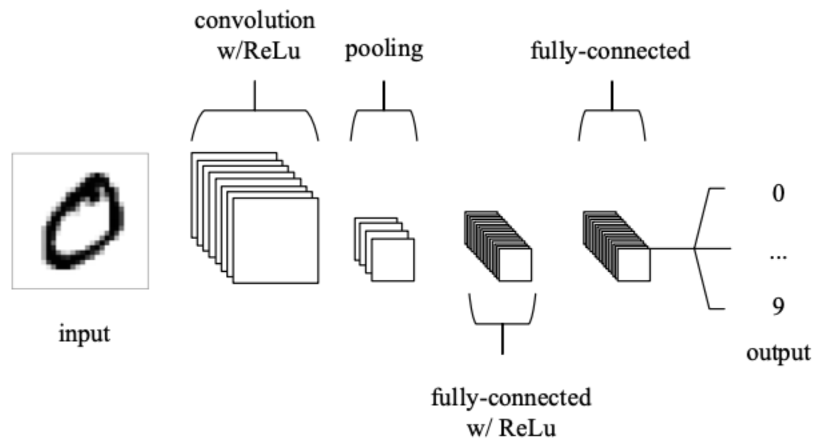


Figure 2.1: CNN architecture. Source: Keiron O'Shea, Ryan Nash[9]

An input is passed through the CNN which holds the pixel values of the input image. The **convolutional layer** will calculate the scalar product between the weights and input region. This will then determine the outputs of neruons that are connected to those regions. A Rectified Linear Unit (ReLu) will apply an activation function. Next the **pooling layer** will reduce the number of parameters in the activation from the ReLu. This is done by using downsampling in the spatial dimensionality of the input. Lastly, a **fully connected layer** will produce class scores from these activations, which are used for classification.

2.1.2 Overfitting a model

Overfitting can be a challenge when it comes to deep neural networks. It might seem like simply adding more layers to the model will improve performance indefinitely. However, when adding more complexity to a neural network overfitting

can happen. Overfitting happens when the model fits exactly to the training data, and is unable to generalize on objects it has not seen before [10]. This may result in a model having a reduced ability to do the task of pinpointing features, not only in the training set but also in the test and validation sets [9].

2.1.3 Transfer learning

Typically, there is an assumption that training and testing data for detection models is taken from the same domain. However, in many cases training data can be expensive to collect. Because of this, having models pre-trained on data from different domains could be helpful. The pre-trained model acts as a foundation, and the model uses the old features to enhance new learning. This method is called **transfer learning** [11]. This enables models to be trained quickly and effectively despite having small or limited datasets.

2.2 Computer Vision

CV is a field in Artificial Intelligence (AI) about enabling computers to derive meaningful information from visual inputs, and take actions or make recommendations based on said information [12]. Humans rely on large amounts of context to be able to recognize objects, determine depth and whether objects are moving. Computers can learn this by using images, videos and other visual inputs to differentiate objects and separate them into categories.

The two essential technologies used to achieve this is **deep learning** and **CNNs**. Features can be extracted from images to describe different objects, like their shape, size and texture. CV encompasses many different areas, with this paper focusing primarily on object detection.

2.2.1 Object detection

Object detection is a CV task dealing with detecting instances of visual objects of different classes in digital mediums like images or videos [13]. The aim of object detection is to develop computer models that can determine what objects are and where they are located in the scene. Many other CV tasks rely on object detection as a building block, including object tracking and semantic/instance segmentation, which is detailed further in section 2.2.2.

In order to train an object detection model, labeled data has to be provided. A common implementation is to use **bounding boxes**. For images, bounding boxes are traditionally drawn in 2D as rectangles. They can also be drawn in 3D. A comparison can be seen in figure 2.2:

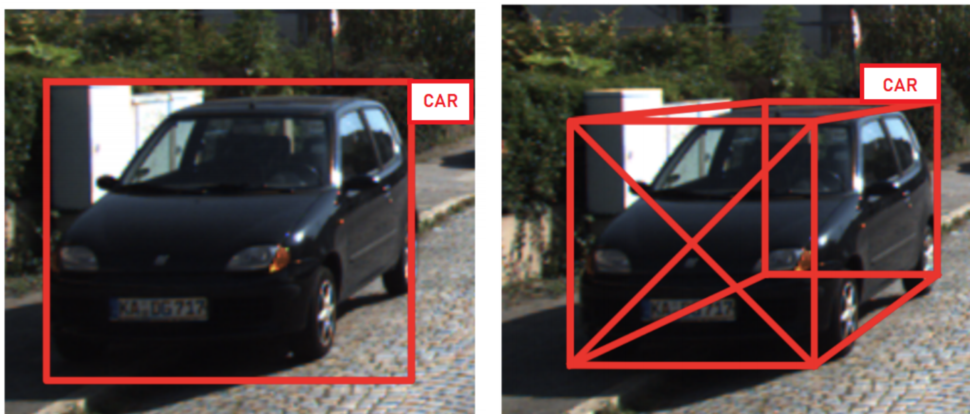


Figure 2.2: 2D bounding box (left) vs 3D bounding box (right). Source: `https://towardsdatascience.com/orientation-estimation-in-monocular-3d-object-detection-f850ace91411`

The bounding box lets the model know where objects are placed in the image, and the associated label provides the object class. The trained model will then attempt to draw its own bounding boxes to determine where the different objects are in the scene. Bounding boxes offer some advantages over other annotation methods like brush strokes or scribbles, due to their fast placement and them spatially constraining the object [14].

One of the big challenges with the bounding box solution is the need for manual

annotation by humans to create datasets for training. Annotating images is an expensive and time-consuming process, and datasets often need to be large to cover the variance in the data[14]. As such, being able to automate this process could save a substantial amount of time and money, while also making the creation of larger datasets easier.

2.2.2 Semantic and instance segmentation

Semantic segmentation is the process of assigning a label to every pixel in an image [15]. Unlike regular object detection, where one might focus on only specific objects in the images, segmentation requires every pixel in the image to be assigned a specific class.

Instance segmentation takes this concept further, by assigning a unique label to every instance of the different objects as well. For example, with semantic segmentation the pixels representing all cars in an image would be assigned the same label. For instance segmentation, all the cars would be assigned different labels like Car1, Car2 etc. Figure 2.3 visualizes the difference between the methods:

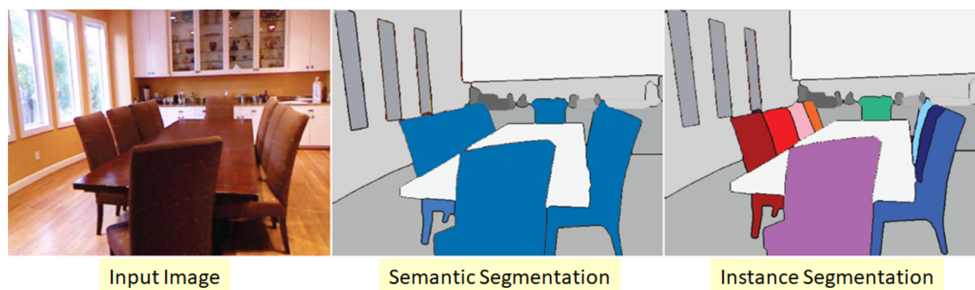


Figure 2.3: A scene processed using semantic and instance segmentation. Source: `https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339`

According to Hafiz et al. [15], instance segmentation could have significant applications in robotic automation, autonomous driving and surveillance among other areas. Many popular object detection models like Faster R-CNN (see section 2.3.2) have the ability to perform instance segmentation on images in addition to object

detection. Newer datasets like COCO [1] and Cityscapes [2] also provide segmentation images which can be used for training and validating models.

2.2.3 Mean Average Precision

A common way to measure the performance of an object detection model is by using Mean Average Precision (mAP) over the classes [16]. As mentioned, the goal of object detection is to localize objects in the images, and assign the correct classes to said objects.

To understand mAP, one needs to understand the **precision-recall** curve [17]. **Precision** is a measurement of how many objects the model gets correct out of the ones it guesses. The formula for precision can be expressed as:

$$Precision = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}, \tag{2.1}$$

where True Positives is the amount of objects the model correctly guesses, and False Positives is the amount of objects the model did not manage to guess. **Recall** is a measurement of how many objects the model managed to guess out of the total amount of existing objects.

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}, \tag{2.2}$$

where False Negatives is the amount of objects where the model guessed incorrectly. Most models also implement the concept of **confidence**, which is a threshold used to adjust what the model should prioritize. A higher confidence value means the model values avoiding False Positives over avoiding False Negatives. The precision-recall curve is a plot of the models' recall and precision values as a function of its confidence threshold.

There are different types of metrics associated with the precision-recall curve, with the most relevant one for this project being the **Average Precision (AP)**

[1] `https://cocodataset.org/#home`
[2] `https://www.cityscapes-dataset.com/`

curve. It is calculated as the weighted mean of precision achieved at each confidence threshold. The increase in recall from each previous threshold is used as the weight.

To get the mAP, one also needs to consider Intersect over Union (IoU). This is a threshold which determines how tight the bounding boxes need to be in comparison to the ground truth label. For example, if the IoU is set to 0.5, at least 50% of the bounding box needs to overlap with the ground truth to count as a correct guess. The IoU threshold depends on the task at hand. For example, the COCO dataset uses ten thresholds, starting at 0.5 and increasing by 0.05 until 0.95 [3]. It could be useful to combine these different thresholds into one metric, resulting in the mAP metric. The AP for each class is calculated across all the different IoU thresholds, and the mAP for all classes is averaged to arrive at the final metric value.

2.3 Object detection models

2.3.1 Yolov5

You Only Look Once (Yolo) is a popular ML algorithm which uses neural networks to do real-time object detection. It is named You Only Look Once because it only requires a single forward propagation through a neural network to detect objects. This is also known as a **Single shot detector**. It employs CNNs in a regression method to provide class probabilities of provided images. The first version of Yolo was released in 2015 by Redmon et al. [18]. Since then many new versions have been released, with the newest mainstream release being Yolov5 as of May 2022.

Yolov5 is currently the newest mainstream iteration of Yolo. It was released in 2020, and is being updated regularly on its Github page[4]. The main difference between this and the previous version, Yolov4, is that it is much faster, albeit at the cost of some prediction accuracy. Unfortunately, as of May 2022, there is no paper detailing how Yolov5 works in detail. However, there is some information

[3] `https://cocodataset.org/#detection-eval`

[4] `https://github.com/ultralytics/yolov5`

about the model architecture available on the Github page[5]. Figure 2.4 shows a simplified version of the model architecture.

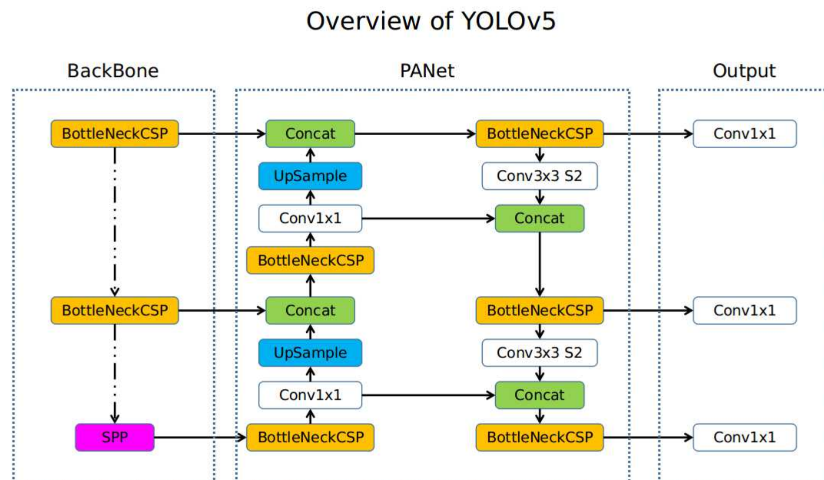


Figure 2.4: Yolov5 architecture. Source: `https://github.com/ultralytics/yolov5/issues/280`

Much like other CNN-based detectors, it consists of many stages of convolution at different scales. The input images get upscaled and downscaled at various stages in the pipeline, and results are concatenated.

Yolov5 also uses a variety of image augmentation techniques during training. Mosaics (splitting the image into smaller parts and distributing them randomly across the image) are used to ensure that small objects are well recognized, as well as increasing image variance. Other augmentations are also applied randomly, like image flips, rotation and scale variance.

An important aspect of Yolov5 is its ease of use compared to previous versions of Yolo. In an interview with Roboflow.com, Glenn Jocher (the lead developer on the Yolov5 project), shared some insights on what makes it easier to use[6]. Yolo predicts bounding boxes as deviatons from a list of anchor box dimensions. Anchors are pre-exisiting areas of the images where the model assumes objects will occur.

[5] `https://github.com/ultralytics/yolov5/issues/6998`
[6] `https://blog.roboflow.com/yolov5-improvements-and-evaluation/`

These boxes were initially created based on the COCO dataset, which Yolov5 is customized for. By creating thousands of candidate anchor boxes per image and using loss functions to reward correct boxes, the model can learn efficiently. The new contribution in Yolov5 was the introduction of genetic anchors. Instead of using pre-existing anchors, the model uses genetic learning algorithms to create new anchors based on the bounding boxes of the custom dataset. This enables the model to train using custom datasets much more easily than previous versions, with no modification of the model needed.

2.3.2 Faster R-CNN

Faster R-CNN is a network composed of two main modules. These two modules are the Region Based Network (RBN) and the Fast R-CNN module that detects the objects within that region [19]. Faster R-CNN is the third iteration of its family, having R-CNN and Fast R-CNN as its predecessors. Faster R-CNN is the fastest of the three, but also the most efficient computationally due to sharing convolutional computations across both the RBN module and the Fast R-CNN module.

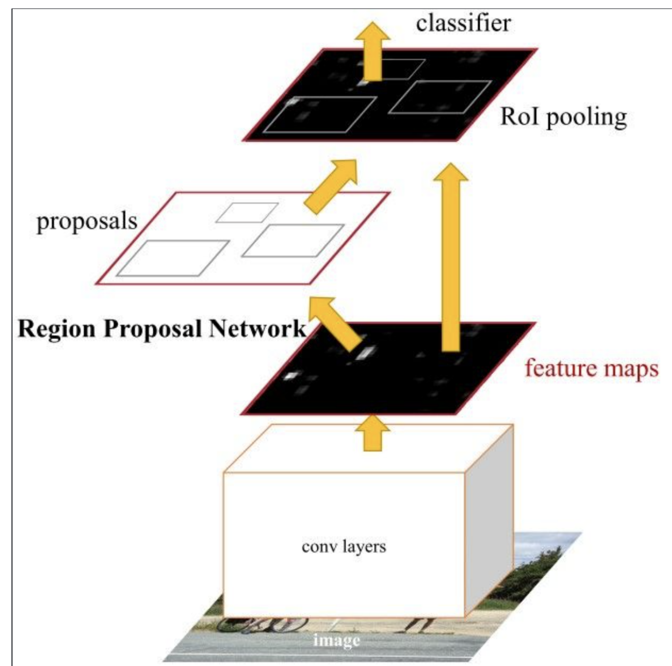


Figure 2.5: Faster R-CNN architecture. source: `https://blog.paperspace.com/faster-r-cnn-explained-object-detection/`

As seen in figure 2.5, the RBNs job is to generate proposals based on different regions. It will then apply neural networks using attention to guide the Fast R-CNN detection module. This will help the module to look for the specific objects in the image [19]. Looking at the figure, one can see that the convolutional layers are shared across both the RBNand the Fast R-CNN module.

Region Based Network

The previous models used a Selective Search algorithm for the region proposals, whereas Faster R-CNN opted for a network that itself can produce these regional proposals [19]. It takes an image as input and returns rectangular object proposals. Generating these region proposals is done by sliding a small network across the feature map, as seen on figure 2.5. One of the main advantages of RBN is that it processes the image using the same convolutional layer that was used in the Fast R-CNN module. Because of this, the RBN will not use any extra time to produce regional proposals. This is also one of the reasons why Faster R-CNN is

faster than its predecessors, as mentioned in the sections above. The RBN module and the Fast R-CNN module can be seen as a single unified network.

2.4 LiDAR

Light Detection and Ranging (LiDAR) is a technology that uses laser beams to create 3D representations of its surrounding environment. The technology is used in many different areas, such as infrastructure, robotics, mapping and autonomous driving [20]. The LiDAR sensor sends out light waves into the environment, where these beams bounce back when hitting an object of any sort and returns to the sensor. The sensor will then measure the distance from the sensor to the hit object, by calculating the laser beams' time traveled. This step is then repeated millions of times per second to give a visual 3D representation of an environment surrounding the sensor [20]. When driving in an urban area a LiDAR can map everything from tall buildings and trees to smaller objects like pedestrians and vegetation.

Systems that make use of LiDAR have the advantage of creating a high-resolution 3D map (also called point cloud) of the surroundings. This means it can get accurate distance measurements compared to other systems using regular cameras or stereo vision [20]. There are millions of points generated at a high speed, which is one of the reasons for its accuracy.

2.5 Carla Simulator

Using simulators is a common way to allow for testing models for autonomous driving without requiring expensive equipment and tools. One such simulator is CARLA; an open-source autonomous driving simulator built using Unreal Engine [7]. According to their website, the goal of CARLA is to "**..help democratize autonomous driving R&D, serving as a tool that can be easily accessed and customized by users".** It uses Python and C++ as its main programming languages, allowing users to create scripts which modify and control the open world

[7] `https://carla.readthedocs.io/en/0.9.12/start_introduction/`

simulator.

The Carla simulator is built using a scalable client-server architecture. The server is responsible for handling everything related to simulating; computation of physics, sensor rendering, the world-state and the actors etc. The client consists of different client modules which control the actors and world conditions. These can be manipulated and interacted with using the Carla Python API, acting as a communication-layer between the server and client.

The team behind Carla highly recommend using a dedicated GPU for the server part of the simulator. For the current newest version (0.9.13 as of May 2022) they recommend at least 6GB of dedicated video memory for the GPU, especially if ML will be used when simulating. This ended up being somewhat problematic for the team, due to limited access to hardware options. The team had access to one laptop with a 4GB GPU, which was able to run Carla version 0.9.11 in a limited capacity. It was decided that using the newest version should be a priority to ensure access to the most updated features. The team got access to an upgraded Virtual Machine (VM) provided by NTNU, which had sufficient hardware. However, only one VM was provided, which had to be split between both team members. To resolve this, one team member usually remoted into their own desktop PC which had sufficient specifications for Carla. This ensured that the project could be carried out with the newest version of Carla at the time (0.9.13).

As mentioned previously, the Carla simulator is split into a server part (which acts as a spectator overview for the simulation) and a client part. The client part uses the Pygame-package (see section 2.5.2) in Python to allow additional windows to be opened for further control. For example, when running the **manual_control.py** script from the included examples from the Carla github page [8], a second window pops up where the user can control a car using their keyboard.

[8] https://github.com/carla-simulator/carla/blob/master/PythonAPI/examples/manual_control.py

2.5.1 Python API

The primary way a user can interact with the Carla simulator is by using the Python API[9]. This allows for manipulation of the simulation, and is essential if one is to extract simulator data. The API allows for interaction with and manipulation of most aspects of the world, including:

- **Actors**: These are the objects that interact with the simulation in any way. Primarily vehicles, pedestrians and sensors.
- **Sensor**: Various objects that can be attached to actors for data collection and observation.
- **World**: The world that the simulation happens in. Several attributes can be changed, like time of day, the current map or weather conditions like rain and wind.

2.5.2 Relevant python packages

A variety of python packages were used during the project, some of which are required to use the Carla Python API. This section contains a brief summary of the most relevant ones.

- **Numpy[10]:** Package used for scientific computing in Python, primarily used for efficient multidimensional array operations. This package is required to use the Carla simulator.
- **Pygame[11]:** Package for creating games in Python. Used by the Carla PythonAPI for rendering a controllable client.
- **OpenCV[12]:** Image processing package for Python. Used to read and write images, as well as find shapes and draw bounding boxes on said images.
- **Open3D[13]:** Library which supports rapid development of software that deals with 3D data. Used to read and display point cloud files.

[9] https://carla.readthedocs.io/en/latest/python_api/
[10] https://numpy.org/
[11] https://www.pygame.org/wiki/about
[12] https://opencv.org/
[13] http://www.open3d.org/docs/release/getting_started.html

- **Matplotlib pyplot[14]:** Interface package for Matplotlib, which allows for visualization of data. Used for displaying and saving images to disk.

2.5.3 Types of sensors

Various sensors were used to collect data from the Carla simulator[15]. Carla contains a library of different sensor types, like cameras, collision detectors and LiDAR sensors. This next section will briefly explain some of the sensors that were used in this project to collect data in Carla. All of the sensors can be placed freely inside the simulator, but are usually attached to the player vehicle in a certain position.

- **RGB camera sensor:** Acts as a typical camera, capturing images from the scene. A variety of options can be adjusted, like image resolution, iso and other camera settings.
- **Semantic segmentation camera:** Classifies every object in camera view by displaying them in different colors. Objects are pre-labeled depending on their type (e.g. pedestrians are red, cars are blue). Around 20 different classes are available.
- **Instance segmentation camera:** Introduced in version 0.9.13, this sensor operates similarly to the semantic segmentation camera, but creates unique colors for each actor based on their unique ID.
- **LiDAR sensor:** Simulates a LiDAR sensor by using ray-casting. This is a fairly complex sensor, and has a lot of parameters which can be adjusted for effect. See section 3.2.5 for more details.
- **Semantic LiDAR sensor:** Similar to the LiDAR sensor, with two main differences: the raw data collected contains more data per point (primarily semantic information about objects hit by the LiDAR) but does not include the intensity, drop-off or noise model attributes.

[14] `https://matplotlib.org/3.5.0/api/_as_gen/matplotlib.pyplot.html`
[15] `https://carla.readthedocs.io/en/latest/ref_sensors/`

2.5.4 Carla Client bounding boxes

The Carla simulator has specific functionality for getting the 3D bounding boxes of spawned actors in the simulation[16]. Specifically, bounding boxes for spawned vehicles and pedestrians can be obtained through their **bounding_box** attribute. The information includes the object location, orientation and the eight box outer points in an (X, Y, Z) format.

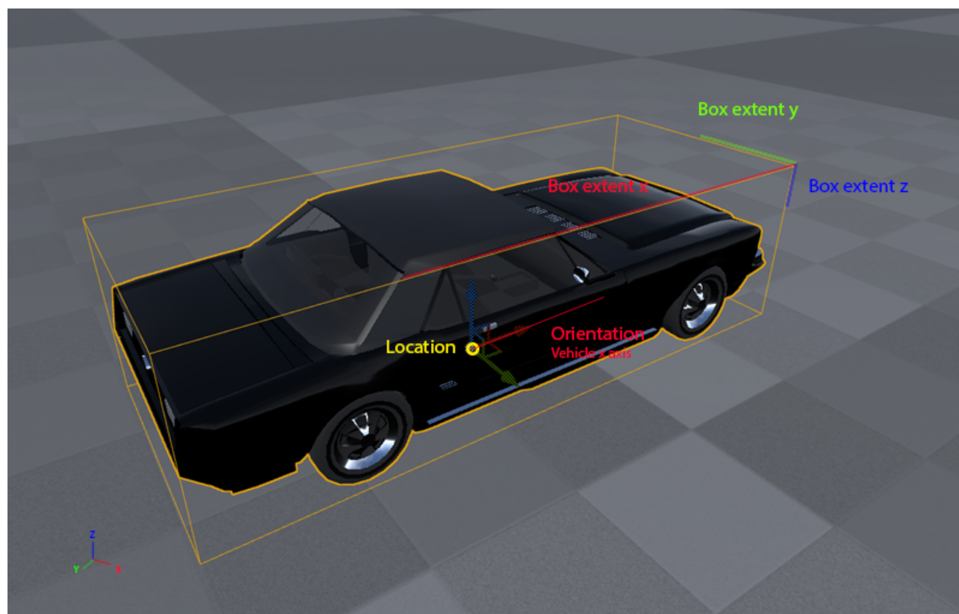


Figure 2.6: Carla Vehicle bounding box example. Source: `https://carla.readthedocs.io/en/0.9.5/measurements/`

2.5.5 Ouster LiDAR sensor and LiDAR images

NTNUs NAPLab[17] has a car fitted with a variety of sensors, including a LiDAR sensor developed by the company Ouster. Ouster enables its users to get various images as an output from their LiDAR sensor [18]. These images are made by projecting the 3D point cloud created by the sensor into 2D images using information from the points in the cloud. Ouster provides three types of images: range, intens-

[16] `https://carla.readthedocs.io/en/0.9.5/measurements/`
[17] `https://www.ntnu.edu/idi/naplab`
[18] `https://ouster.com/blog/the-camera-is-in-the-lidar/`

ity and ambient. For the range images, each pixel value represents the range from the sensor to the projected point. For intensity, the pixel values are typically calculated based on factors like reflectivity and by measuring the amount of light that returns to the sensor. For the ambient images, Ouster uses a combination of various data collected from the sensor to create a more realistic image. An example of these images can be seen in figure 2.7:

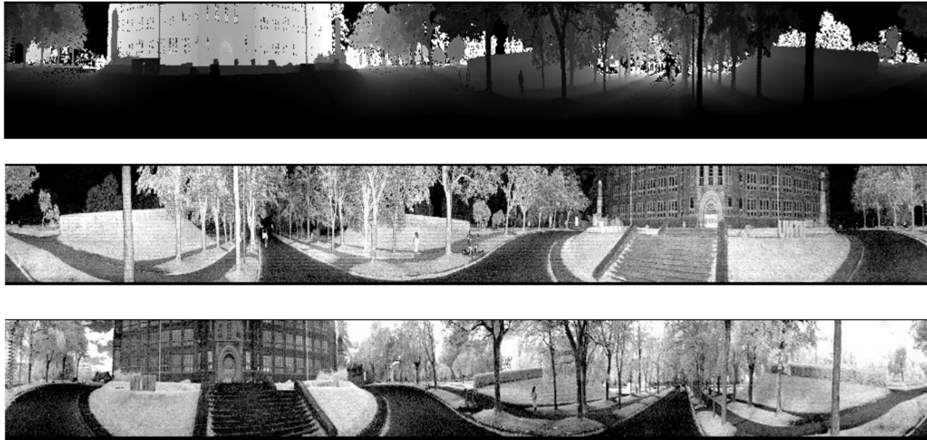


Figure 2.7: Example of Ouster LiDAR images. Top is range, middle intensity and bottom ambient. Source: `https://ouster.com/blog/the-camera-is-in-the-lidar/`

The Carla simulator has several LiDAR sensors available. The primary LiDAR sensor[19] can be attached to a vehicle for data collection. In order to create images from the LiDAR data collected from the Carla simulator, a sensor with similar settings to the Ouster LiDAR would need to be used. Following this one could try to generate similar images that can be used for training. Unfortunately, much of the Ouster image processing is not public domain, as they develop and provide it as a part of their business. As such, part of the data collection process for this project included creating LiDAR images similar to the Ouster LiDAR images that were collected using the NTNU Ouster LiDAR.

[19] `https://carla.readthedocs.io/en/latest/ref_sensors/#lidar-sensor`

2.6 Related Work

A paper published by Jang et al. details a method to automatically gather annotated image data using the Carla simulator, as well as using semantic segmentation to correct the collected bounding boxes [6]. The paper aimed at making the collection of synthetic data from Carla easier and more robust. One of the provided scripts were used to gather images and 3D bounding box information from the simulator, while the other was used to convert the 3D bounding boxes to 2D and further process them to achieve more tightness. The second part was done by using segmentation images retrieved from the simulation, which when visible provide pixel-perfect boundaries of each object. Their scripts were used as a baseline for parts of this project, with some modifications and improvements to fit the assignment specifications.

There have been attempts in the past to find out if simulator data can be used for training object detection models for the use of real-life detection. Dworak et al. investigated whether LiDAR data collected from the Carla simulator could be used to train deep learning object detection models [21]. They found that while combining the real and simulated data for training did not improve the models, it could still be used for fine-tuning. This was done by first training a model using the simulated data, then fine-tuning this model using a portion of the real world data. This created a model with similar performance to the baseline, while requiring less real world data to achieve similar performance. They also suggest that the simulation data could be used for validating new solutions.

A paper published in 2021 by Niranjan et al. investigated whether data collected from the Carla simulator could be used to train object detection models [22]. They collected around 1000 images from Carla which were then annotated by hand. They used 5 classes; vehicles, bicycles, motorbikes, traffic lights and traffic signs. SSD Mobilenet was chosen as the object detection model to train using the collected dataset. Testing the model, they measured an accuracy of 82.81%, leading them to claim that Carla can be utilized effectively to train and test object detection algorithms. They argue that Carla has advantages over other simulators in the form of its open source nature, compatibility and real world environment modelling.

Another paper using Carla simulator data was published in 2021 by Bu et al. [23]. They wanted to investigate whether data extracted from Carla could be used to train object detection models. They managed to create a pipeline to automatically annotate the data. To prove the ease of creating automatically annotated data using Carla, they used objects not included in the semantic segmentation label list by adding their own objects in the form of fire hydrants and crosswalks. They then used the ray-tracing capabilities of Unreal Engine in combination with the RGB camera sensor to capture annotated images. To get a baseline, they implemented a Cut-Paste method, in which images of real objects are pasted in various places, sizes and rotations on background images. This was done in an attempt to show that Carla data could outperform poorly made datasets made from real life images. ResNet was chosen as the object detection model, and the trained model was benchmarked on several real life datasets. They conclude by saying that data from the Carla simulator can be effectively used to train models to detect real life objects. While not achieving performance close to the models trained on proper datasets, their model performed above the baseline, showing that Carla data could outperform poor real-life datasets.

A paper by Tremblay et al. explored whether synthetic data could be used to train deep networks [24]. Synthetic images were created by placing 3D models of objects such as cars in a 3D scene in random positions and orientations. Textures and colors of the objects were randomized, and shapes were randomly placed to allow the network to learn which objects were not of interest. Backgrounds were selected from real images. Another relevant aspect of the paper is the usage of random perturbing to force the models to focus on learning important features over photo-realistic ones. The networks trained on purely synthetic data performed compellingly. The best performance was found when fine-tuning with real data, which performed better than using real data alone. According to the paper, this shows how synthetic data can replace parts of real datasets to reduce the amount of annotating that has to be done.

Creating and using 3D objects in place of real objects is a fairly popular alternative to using real data. The main challenge lies in creating realistic and photo-realistic backgrounds, which requires significant time and effort. To remedy this, Alhaija et al. attempted a hybrid solution; using real life images as backgrounds, and

rendering photo-realistic 3D objects into the scenes in a realistic manner [25]. They argue that this is possible because not all aspects of the scene are equally important when training a detection model. The generated dataset was trained on an object detection model and compared to synthetic data as as well as a small real-life dataset. They found that the model trained on their augmented dataset generalized better than both the real-life and synthetic datasets.

A similar method was developed by Tsirikoglou et al., where highly realistic synthetic data with per-pixel accurate annotations was generated to aid in CV tasks related to semantic segmentation [26]. Their image generation pipeline was based on procedural world modeling, using light transport simulation with path tracing techniques. They estimate that the time spent generating their dataset was three to four times shorter than other synthetic options, while achieving similar performance. This was done by focusing on important features in the image, while also managing low-level features like anti-aliasing and motion blur. The paper also found that the best relative merits for comparing synthetic datasets comes from training on the synthetic data alone, without initializing weights or fine-tuning.

Richter et al. explored using the video game GTA V, a realistic open-world game, to create pixel-accurate semantic label maps for images extracted from the game [27]. They produced a dataset with 25,000 images, and used this to train a detection model. Their results showed that a model supplemented with the synthetic data achieved significantly better results, while also being able to reduce the amount of hand-labeled data. A model trained with their dataset and only 1/3 of a real-life dataset outperformed a model trained solely on the real-life data.

Chapter 3

Methodology

This chapter is split into two main parts. The first part details the collection of simulator data in order to create a dataset. The second part describes how this dataset was used to train two different object detection models. For more details on how specific experiments were conducted and the results, see chapter 4. For the Python script sections, select parts of the code is presented. All of the code used for the project can be found on the teams' Github repository [1].

3.1 Fixing the Carla instance segmentation LiDAR sensor

In version 0.9.13 (released November 16, 2021), the Carla team added an instance segmentation camera sensor [2]. This sensor could be useful to perform automatic annotation of the collected RGB images, since having a unique color for each actor would make separating the different objects easier. The sensor would, according to the announcement, store the ID of the actors in the G and B channels of the image, with the red channel storing the semantic tag (similar to the semantic segmentation camera sensor). Unfortunately the sensor was released without any relevant documentation. The Carla sensor reference[3] did not contain any information on

[1] https://github.com/PederEspen/master-thesis
[2] http://carla.org/2021/11/16/release-0.9.13/
[3] https://carla.readthedocs.io/en/latest/ref_sensors/

the new sensor. Several posts on the Carla forum and Github page complained about this[4]. Inspecting the Carla source code [5], it seemed like the instance segmentation sensor used a different ID for encoding the green and blue pixel values compared to other sensors like the semantic Lidar sensor. The instance segmentation images were created by encoding the IDs into the green and blue channel of the image, see section 3.2.4 for more information. The semantic Lidar sensor[6] utilized Carla's ActorRegistry, an zero-indexed ID system used to assign IDs to spawned actors in the simulation. The code snippet seen in figure 3.1 from the source file **ActorRegistry.cpp** shows how the ID was incremented for each actor spawned in the simulation.

```
IdType Id = ++FActorRegistry::ID_COUNTER;

if (DesiredId != 0 && Id != DesiredId) {
  // check if the desired Id is free, then use it instead
  if (!Actors.Contains(DesiredId))
  {
    Id = DesiredId;
    if (ID_COUNTER < Id)
      ID_COUNTER = Id;
  }
}
```

Figure 3.1: Code snippet from ActorRegistry.cpp

The ID used by the Instance Segmentation sensor was fetched directly from Unreal Engine, using the function **GetUniqueID**[7]. This was an ID assigned by Unreal Engine to spawned objects. Because this ID was different from the Carla ActorRegistry ID the sensor was effectively not usable, since there was no way to match the Carla actor IDs to the IDs calculated from the image pixel values.

Luckily a user on the Carla Github page found a solution to this [8]. The user found that while they could not change the instance sensor to use the ActorRegistry ID,

[4] https://github.com/carla-simulator/carla/issues/4938

[5] https://github.com/carla-simulator/carla/tree/master/Unreal/CarlaUE4/Plugins/Carla/Source/Carla

[6] https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-Lidar-sensor

[7] https://docs.unrealengine.com/4.26/en-US/API/Runtime/CoreUObject/UObject/UObjectBase/GetUniqueID/

[8] https://github.com/carla-simulator/carla/discussions/5047

they could change it the other way around, i.e. make Carla use the Unreal Engine ID for its actors. This would make it so that one could retrieve the correct ID from the pixel values, and match this to the actor bounding boxes retrieved from the simulation.

Fixing this issue would require a change of the source code, and one would need to build Carla from source to be able to apply the change. Originally, only the pre-packaged version of Carla had been used during the project. Building Carla from source proved to be a complicated process. The documentation contained instructions for how to build on both Windows and Linux, although Linux was recommended. Initially, Windows was chosen as this was the operating system of choice for both team members. Building on Windows proved quite difficult, with several errors showing up along the process. A post was made on the Carla Discord server to try and get some help with the error messages. The team decided to instead build using Linux as it was the recommended option.

Building on Linux was done successfully, and some testing was performed to ensure that the ID was changed properly for the other sensors that were used (like the semantic Lidar sensor, which provides an object ID along with the point cloud information). Testing showed that the method worked, and a precise method to tighten bounding boxes using instance segmentation could be implemented.

3.2 Collecting simulator data and creating datasets

3.2.1 Defining dynamic object types

Performing object detection required the selection of object classes. As this project focused on detecting dynamic objects in a traffic situation, the following four classes were chosen:

- 0 - Car (regular passenger cars)
- 1 - Truck (bigger cars like vans or ambulances)
- 2 - Cyclist
- 3 - Pedestrian

The classes were partly chosen based on what objects were available in the Carla simulator and partly based on what classes were typically found in available real-life datasets for comparison.

3.2.2 Methods of gathering sensor data

There are primarily two ways one could go about gathering data from the Carla simulator: Letting the player vehicle drive around on its own for a specified amount of time, or by manually controlling the player vehicle. Typically one would use a specific route to collect data on, and do repeated sessions with different parameters to gather a variety of data.

The player vehicle would be fitted with select sensors, which were accessible through the Python API. For example, the following snippet attached a collision sensor to the vehicle and gathered collision data every game update:

```
col_bp = world.get_blueprint_library().find('sensor.other.collision')
col_location = carla.Location(0,0,0)
col_rotation = carla.Rotation(0,0,0)
col_transform = carla.Transform(col_location,col_rotation)
ego_col = world.spawn_actor(col_bp,col_transform,
                attach_to=ego_vehicle, attachment_type=carla.AttachmentType.Rigid)
def col_callback(colli):
    print("Collision detected:\n"+str(colli)+'\n')
ego_col.listen(lambda colli: col_callback(colli))
```

Figure 3.2: Carla PythonAPI Example

The **listen()** function ran every game update and checked for new information. In this case it simply output the information gathered to console, but one could for example write the information to a .csv file to be used for later analysis. It could also be used to save data to disk, like in the case of the RGB image or LiDAR sensors.

3.2.3 Extracting RGB data from Carla

The first step of the data collection process was to gather raw data from the simulator. This was done using the file **ExtractRGB.py**. This file was adapted from a file (**extract.py**) from the CarFree project [6]. The script spawned a player vehicle used for collecting data, which then could be interacted with via a Pygame window. The primary controls included the ability to drive the car manually using the arrow keys, having the car drive using auto pilot, and being able to either manually and automatically capture images. Figure 3.3 shows the Pygame window.

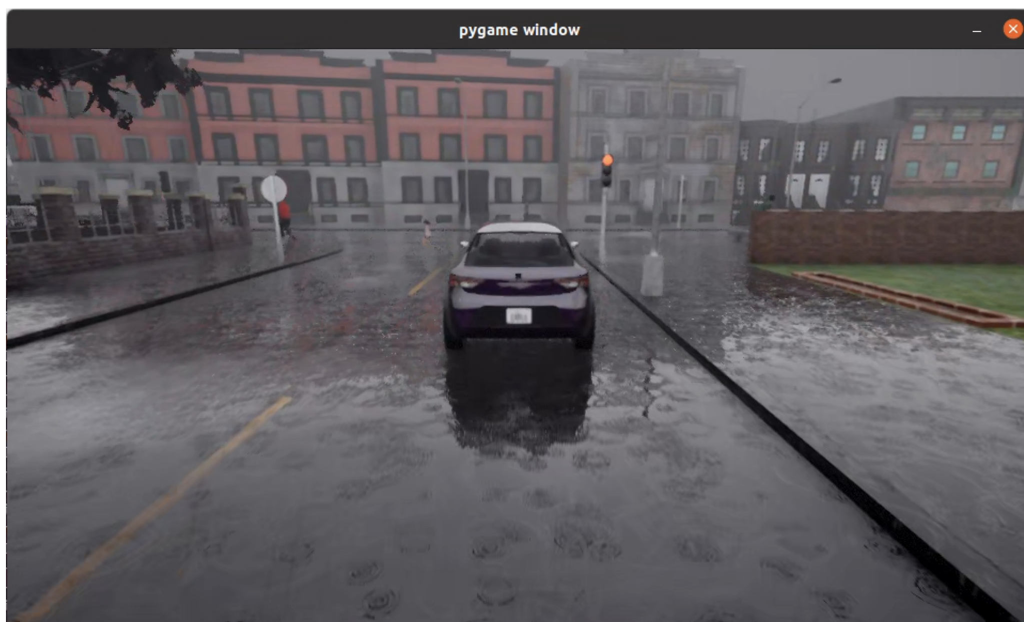


Figure 3.3: Pygame window in Carla

For the majority of the data collection, the autopilot and automatic capture of images was used. The manual control and image capture was occasionally used during development to debug or get images from specific angles or situations. The images were captured with a resolution of 960x540.

After launching the Carla client, vehicles and pedestrians needed to be spawned. This was done using the **spawn.py** file, which comes included with the Carla simulator. The file spawned a specified amount of actors into the simulation, where the amount of vehicles and pedestrians could be chosen separately. For more details,

see the subsection regarding dataset imbalance in section 3.2.7.

The player vehicle was spawned with the RGB camera sensor and Instance Segmentation camera sensor attached. Both sensors were attached to the car via the Python API, and were shifted in the x and z directions slightly, along with a pitch rotation of -15 degrees. This was to emulate a camera being attached to the inside of the front windshield of the car, looking forward.

The script applied a synchronous mode to the client. This was done to ensure consistency when collecting data. Normally, data would be collected every game update of the simulation when using the **listen()** function in an asynchronous fashion. Creating a separate synchronous loop would prevent this, and only allow data capture when certain requirements were met. Figure 3.4 shows the loop which ran continuously when starting the script. Using this loop, images were captured every 5 seconds.

```
self.set_synchronous_mode(True)

while True:
    self.world.tick()

    self.capture = True
    pygame_clock.tick_busy_loop(60)

    self.render(self.display)

    self.time_interval += 1
    if ((self.time_interval % args.CaptureLoop) == 0 and self.loop_state):
        self.image_count = self.image_count + 1
        self.rgb_record = True
        self.seg_record = True
        actor_bbox_record = True
        count = self.image_count
        print("-------------------------------------------------")
        print("ImageCount - %d" %self.image_count)

    if self.screen_capture == 1:
        self.image_count = self.image_count + 1
        self.rgb_record = True
        self.seg_record = True
        actor_bbox_record = True
        count = self.image_count
        print("-------------------------------------------------")
        print("Captured! ImageCount - %d" %self.image_count)
```

Figure 3.4: Synchronous loop from ExtractRGB.py

Whenever conditions were met, the script would set the sensor record variables to true. This in turn would allow the **listen()** functions to record and save the data collected at that specific time in the simulation.

In addition to saving the RGB and instance segmentation images, bounding box data had to be collected and saved. The PythonAPI allowed direct access to an actors bounding box via the **bounding_box.extent** attribute. This was used initially to get the 8 extents of the bounding boxes in (x,y,z) format. The next step was to transform these coordinates with respect to the sensor coordinates. Each bounding box had coordinates with respect to its own center, as explained in section 2.5.4. The bounding boxes were only useful if they were located with respect to the camera view, so a transformation of the coordinates was done. This was done in two steps; first by transforming the actor coordinates to the world coordinates, and then from the world coordinates to the sensor coordinates.

After this, the coordinates needed to be projected from 3D into 2D to achieve 2D bounding boxes. To do this, camera calibration was used. Parts of the CarFree code was based on an example script from the Carla Github page, **client_bounding_boxes.py** [9]. Figure 3.5 shows how the camera calibration was implemented.

```
calibration = np.identity(3)
calibration[0, 2] = VIEW_WIDTH / 2.0
calibration[1, 2] = VIEW_HEIGHT / 2.0
calibration[0, 0] = calibration[1, 1] = VIEW_WIDTH / (2.0 * np.tan(VIEW_FOV * np.pi / 360.0))
self.camera.calibration = calibration
```

Figure 3.5: Camera calibration code snippet from client_bounding_boxes.py

The eight bounding box extents were produced with respect to the camera view, with an x and y image coordinate value. These were then saved to disk in .txt files. The format of these files is shown in figure 3.6.

[9] `https://github.com/carla-simulator/carla/blob/master/PythonAPI/examples/client_bounding_boxes.py`

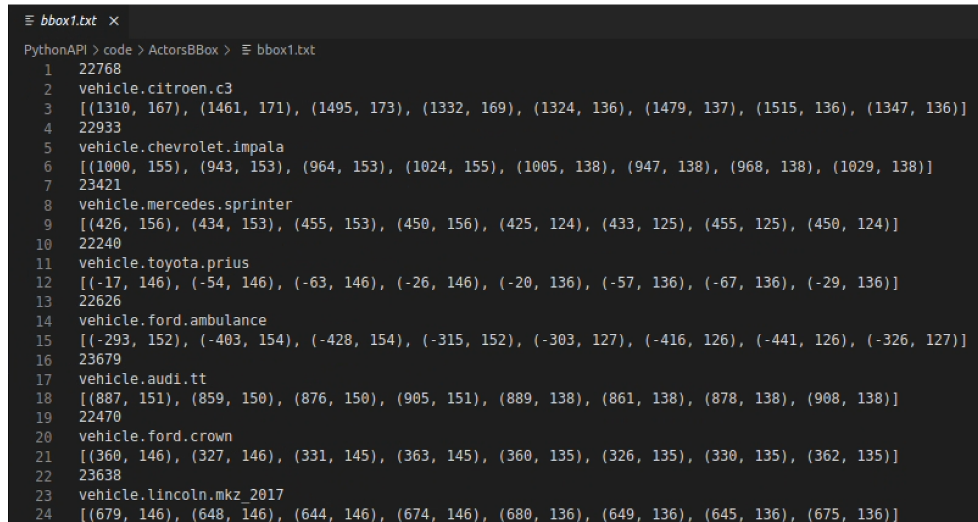


Figure 3.6: Example of a bounding box text file

In the original CarFree code, each line contained a separate bounding box. As semantic segmentation was used, no additional information was needed to achieve bounding boxes, as Carla already separated vehicles from pedestrians with the semantic tag. However, for the instance segmentation implementation, the format had to be altered. In the new version, every three lines contained information about one actor. The first line contained the actor ID. This was used later when tightening bounding boxes to match the actor box ID to the ID calculated from the instance segmentation images. Additionally, the custom labeling (see section ”Creating correct labels” below) required the actors **type_id**, which got stored on the second line. The third line contained the coordinates of the bounding box. The reason for formatting the files this way was to make it easier to read the information when processing the files later.

Creating correct labels

CarFree only used two labels: vehicles and pedestrians. As mentioned in section 3.2.1, this project aimed at 4 labels for the dynamic objects. This meant a different approach had to be taken to add additional labels. Information about the actors could be retrieved from the simulation along with the bounding boxes.

This included the name or **type_id** of the spawned actor, written in the format: *vehicle.tesla.model3*, as an example. The full list of actor names can be found in the Blueprint library[10].

Using this information, a .json file was created which contained all spawnable vehicles and pedestrians, as well as an associated class. This idea was inspired by a project which aimed at converting 3D bounding boxes to 2D[11]. The structure of the .json file is shown in figure 3.7.

```
{
    "reference": {
        "car": 0,
        "truck": 1,
        "bicycle": 2,
        "pedestrian": 3,
        "others": 9
    },
    "classification": {
        "vehicle.audi.a2": 0,
        "vehicle.audi.etron": 0,
        "vehicle.audi.tt": 0,
        "vehicle.bh.crossbike": 2,
        "vehicle.bmw.grandtourer": 0,
        "vehicle.carlamotors.carlacola": 1,
        "vehicle.carlamotors.firetruck": 1,
        "vehicle.chevrolet.impala": 0,
        "vehicle.citroen.c3": 0,
        "vehicle.diamondback.century": 2,
        "vehicle.dodge.charger_2020": 0,
        "vehicle.dodge.charger_police": 0,
        "vehicle.dodge.charger_police_2020": 0,
        "vehicle.ford.ambulance": 1,
        "vehicle.ford.crown": 0,
        "vehicle.ford.mustang": 0,
        "vehicle.gazelle.omafiets": 2,
        "vehicle.jeep.wrangler_rubicon": 0,
        "vehicle.lincoln.mkz_2017": 0,
```

Figure 3.7: Part of the .json file used for labeling objects

The file acted as a dictionary, with the reference containing the different classes and their associated label values. The classification section contained all the type_ids

[10] https://carla.readthedocs.io/en/latest/bp_library/

[11] https://github.com/MukhlasAdib/CARLA-2DBBox

and their labels. When storing the bounding box data, the actor type_id was also stored. Then, for each bounding box the .json file was iterated over and when a match was found, it was assigned the associated class. See section 3.2.4 for more information. Doing this, the dataset could be labeled in different ways by modifying the structure of the .json file. The **others** tag was included to compensate for a bug which would happen occasionally in Carla, where no proper type_id was retrieved. These instances were ignored when iterating over the bounding boxes.

Challenges surrounding bounding boxes and labels

When bounding boxes were fetched on a certain timestamp, the simulator returned all of the currently spawned objects, not only those that were visible to the player vehicle. As such, bounding boxes of objects outside the camera view had to be removed. Also, certain objects could technically be in the camera view, but could be completely occluded by other vehicles, buildings and so on. These occluded objects also needed to be removed.

After these steps, only bounding boxes that were visible inside the camera view were left. The next step was to select min/max X and Y values to create 2D bounding boxes for each visible actor. CarFree solved this by picking the min and max values from the eight 3D bounding box points. The problem with the simple min max approach was that the bounding boxes were often too large, as seen in figure 3.8.

Figure 3.8: Example of bounding boxes being too large

This was due to the rudimentary way of selecting min and max values of the 3D bounding box coordinates. Different angles would change how the bounding box appears due to perspective.

It was also found that the lack of certain features in the Carla simulator limited the selection of classes. An issue was encountered with the **cyclist** class. Typically, this class would be defined as a bicycle, along with a person on top riding it. However, Carla did not spawn pedestrian actors on top of the bicycles. Instead, static pedestrian-looking objects were spawned on top, which meant the bounding boxes did not extend across the entire object, only the bicycle part. Figure 3.9 shows an example of a bounding box drawn for one of these objects. Note how the bounding box only extends to the top of the bicycle, and does not include the person on top. The team did not find a satisfactory solution to this issue, so it was determined that the bounding boxes would have to suffice.

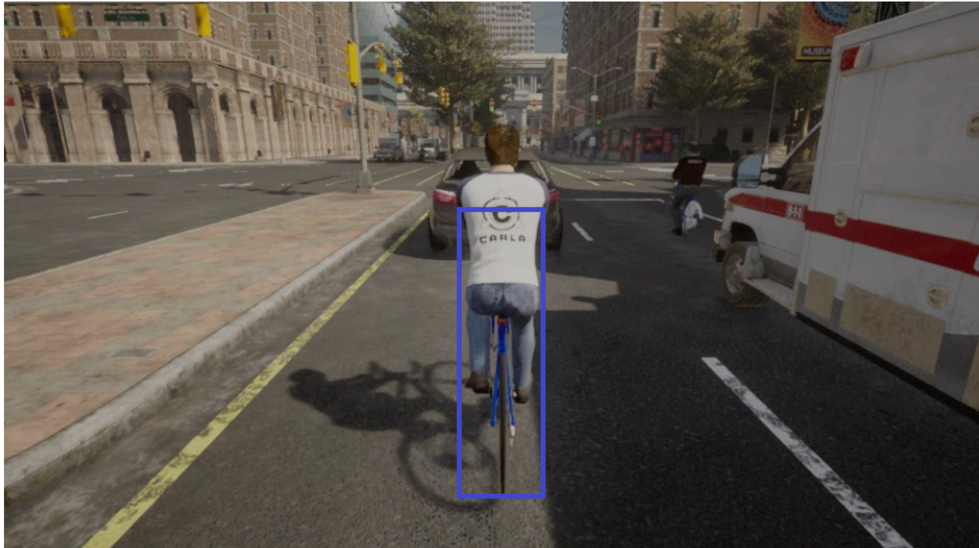


Figure 3.9: Example of cyclist bounding box in Carla

The project supervisor Frank Lindseth mentioned that having a **Bus** class would also be helpful, but it was not possible to spawn any such objects in the simulator. However, bigger vehicles like fire trucks and ambulances were available, and are as mentioned labeled as trucks.

3.2.4 Generating RGB images

Since one of the RQs was to investigate how well a model could be trained on data collected from a virtual environment such as Carla, it was important that the bounding boxes were correctly placed and tight to achieve optimal results. Building Carla from source (see section 3.1) enabled this since it became possible to extract the ID of the different actors from the instance segmentation images. These IDs could then be used to match the different actors to their respective bounding boxes, and get well placed, tight bounding boxes.

To do this post-processing, the file **GenerateRGB.py** was used. The file iterated over the stored RGB and instance images, as well as the bounding box .txt files. Bit-shifting was used to get the colors of the different actors based on their ID, as seen on figure 3.10. It should be noted that the color space used for the project

was BGR (Blue-Green-Red), as this was the default format for the OpenCV package used to read and write images.

```
actor_id = int(a_bbox[0])
actor_color_red =  10
actor_color_green = (actor_id & 0x00ff)
actor_color_blue = (actor_id & 0xff00) >> 8

actor_color = (actor_color_blue, actor_color_green, actor_color_red)
```

Figure 3.10: Code snippet for converting the actor ID into BGR colors

Bit-shifting is done by shifting the integer representation of an ID to either left or right. For the instance segmentation, each object had an associated bounding box ID. The next step was to calculate the colors from that ID, where bit-shifting to the right was done. Applying **(& 0x00ff)** to the ID returns a hexadecimal representation for green, whereas **(& 0xff00)** with the ID gives the hexadecimal representation for blue. There is no need to shift the green to the right, because it is already shifted all the way to the right. Although shifting the hexadecimal for blue is needed to get the blue color into a number between 0-255. Figure 3.11 shows the pipeline that was executed to match the bounding box IDs to their respective instance segmentation colors.

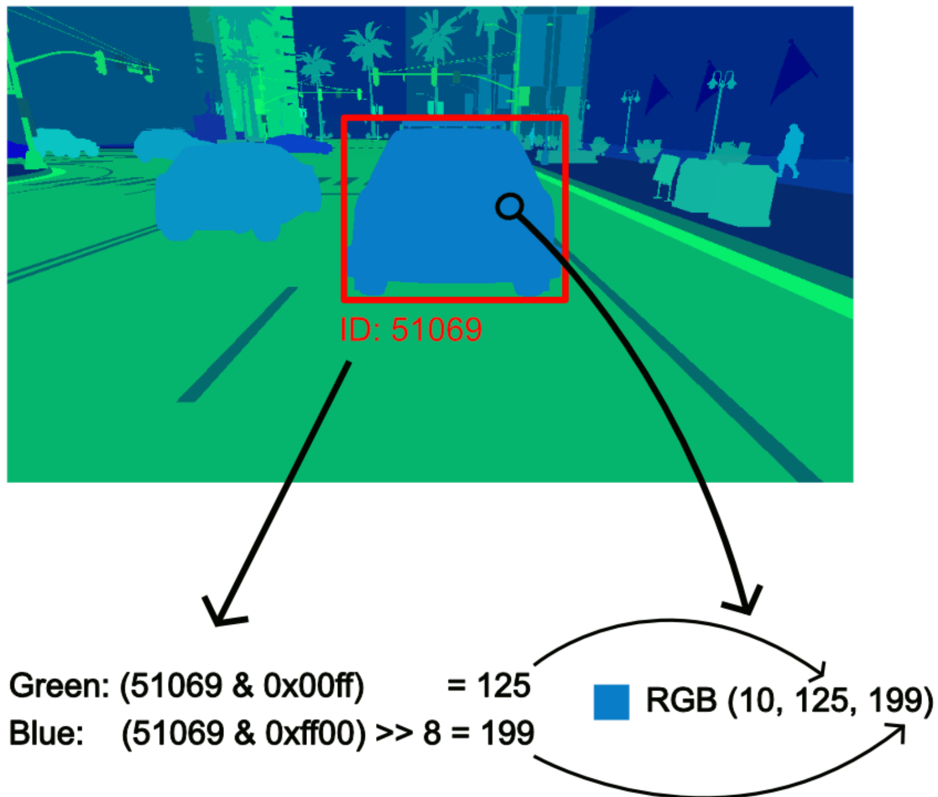


Figure 3.11: Pipeline showing the extraction of the G and B channels of an instance segmentation image, and how bit-shifting is used to match the ID of the object bounding box.

The function **filtering(array, color)** shown in figure 3.12 would filter the actors not in the scene. The **array** argument would be the bounding box points, while the **color** would be calculated based on the id contained in the bounding box .txt file. Every pixel within the boundary of the bounding box in the image would be checked using a double loop, and if a match was found, it meant that the object was both in the scene and also not entirely occluded. Using this function, entirely occluded objects would not pass the filter.

```
# Filters objects not in the scene
def filtering(array, color):
    global seg_info
    for y in range(array[2], array[3]):
        for x in range(array[0], array[1]):
            if seg_info[y, x][0] == color[0] and seg_info[y, x][1] == color[1]:
                return True
    return False
```

Figure 3.12: Code snippet for filtering out objects not in the scene

Correcting partially occluded objects

After removing entirely occluded objects, the next step was correcting the objects that were partly occluded by other objects. After talking to Frank Lindseth, the project supervisor, it was decided that the partially covered objects should have an extended bounding box to show the entire object boundary. Figure 3.13 shows an example of a bounding box occluded by a building.

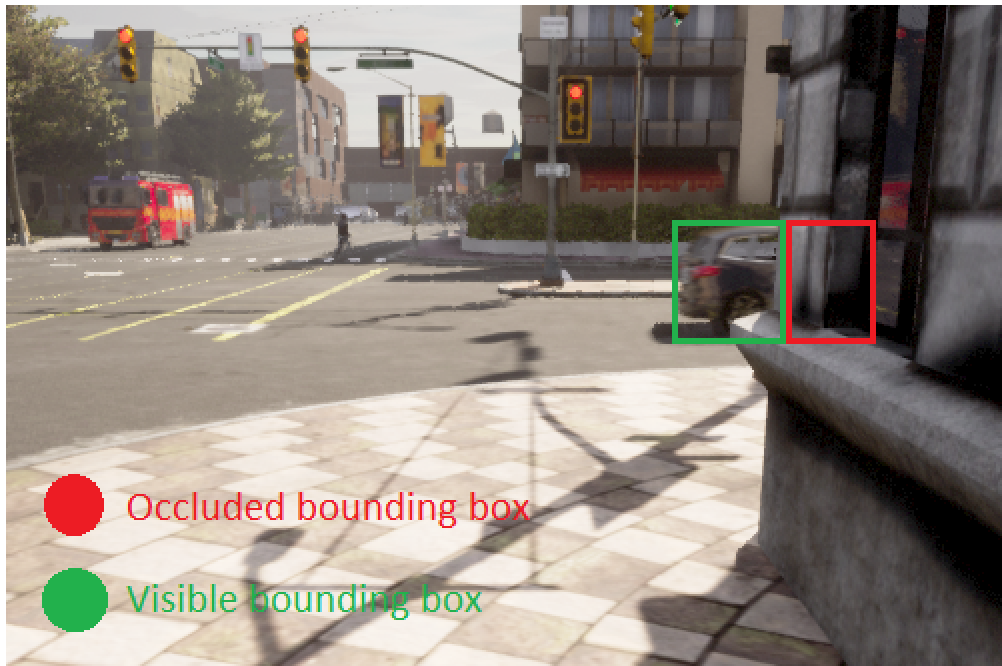


Figure 3.13: Occluded bounding box example

The problem was solved by looking at how much of the objects' bounding box was

covered by something different than the object itself. Looking at the code snippet in Figure 3.14,

```
tightened_percent_min_x = abs((min_x - cali_min_x) / (max_x - min_x))
tightened_percent_max_x = abs((max_x - cali_max_x) / (max_x - min_x))
tightened_percent_min_y = abs((min_y - cali_min_y) / (max_y - min_y))

if tightened_percent_max_x > 0.3:
    cali_max_x = max_x

if tightened_percent_min_x > 0.3:
    cali_min_x = min_x

if tightened_percent_min_y > 0.3:
    cali_min_y = min_y

cali_max_x = max_x
cali_max_y = max_y
cali_min_x = min_x
cali_min_y = min_y
```

Figure 3.14: Code snippet showing how the bounding boxes where tightened

if 30% or more of the bounding box was occluded in the x-axis, the bounding box kept its original bounding box values. However, if less than 30% of the bounding box was occluded, this usually meant that the bounding box was simply too large, and the bounding box was fitted with the instance segmentation image coordinates. The reason for this threshold was because there were images where the original bounding box extended more than 30% outside of the actual object itself. Having this threshold made the large bounding boxes fit pixel-perfect.

A related issue was found specifically for the y-axis, where bounding boxes for objects placed on the sides of the image extended outside of the objects actual bounding box, even extending all the way to the bottom of the of the screen. See figure 3.15 for an example. This was because the bounding box for these objects extended outside the image itself, so to not lose the x-values they were set to either 0 (for the left side of the image) or the max x value of the image (for the right side).

Figure 3.15: Example of bounding box stretching on the edge of the image

Fixing this required a change in the threshold. The best solution the team found was removing the threshold in the bottom part of the y-axis. This meant that fitting the bottom part of the bounding box would happen regardless of any other scenario. This caused its own issue with larger vehicles that were slightly visible behind the top of other vehicles or objects, where their bounding box would become very slim in the y dimension. However, this happened significantly less frequently than the previously mentioned issue, and was seen as an acceptable compromise for the issue.

3.2.5 Extracting LiDAR data from Carla

The script **ExtractLidar.py** created for LiDAR data collection was based on a tutorial script from the Carla documentation, **tutorial_ego.py**[12]. Initially, the same base script as for **ExtractRGB.py** was used, but the synchronous client did not work well with the Lidar sensor, so the asynchronous option was chosen instead. Similar to ExtractRGB.py, vehicles and pedestrians were spawned using a separate spawning script. After this, the script was started to collect data. The script spawned a player vehicle at an available spawn point which would save the point

[12] `https://carla.readthedocs.io/en/latest/tuto_G_retrieve_data/#tutorial-scripts`

cloud files based on the **sensor_tick** attribute (see the next paragraph for more information). Figure 3.16 shows part of the code which retrieved the semantic LiDAR sensor blueprint, set various attributes and attached it to the player vehicle. Each time an image was saved, a counter would be incremented. The game loop would then stop whenever the counter reached a specified threshold, typically a few thousand images collected.

```
ego_vehicle.set_autopilot(True)

lidar_sem_bp = world.get_blueprint_library().find('sensor.lidar.ray_cast_semantic')
lidar_sem_bp.set_attribute('channels',str(128))
lidar_sem_bp.set_attribute('points_per_second',str(2621440))
lidar_sem_bp.set_attribute('upper_fov',str(11.25))
lidar_sem_bp.set_attribute('lower_fov',str(-11.25))
lidar_sem_bp.set_attribute('rotation_frequency',str(20))
lidar_sem_bp.set_attribute('range',str(240))
lidar_sem_bp.set_attribute('sensor_tick',str(5))
lidar_location = carla.Location(0,0,2)
lidar_rotation = carla.Rotation(0,0,0)
lidar_transform = carla.Transform(lidar_location,lidar_rotation)

#lidar_sen = world.spawn_actor(lidar_bp,lidar_transform,attach_to=ego_vehicle,attachment_type=carla.AttachmentType.Rigid)
lidar_sem_sen = world.spawn_actor(lidar_sem_bp,lidar_transform,attach_to=ego_vehicle,attachment_type=carla.AttachmentType.Rigid)
#lidar_sen.listen(lambda point_cloud: point_cloud.save_to_disk('./lidar_output/%.6d.ply' % point_cloud.frame))
lidar_sem_sen.listen(lambda point_cloud: point_cloud.save_to_disk('./semantic_lidar_output/%.6d.ply' % point_cloud.frame))


# --------------
# Game loop. Prevents the script from finishing.
# --------------
while True:
    world_snapshot = world.wait_for_tick()
```

Figure 3.16: Code snippet from LidarExtract.py

As the RGB images were collected at a rate of one every 5 seconds, this was done for the point cloud files as well. The rate could be controlled via the **sensor_tick attribute**, which determined how many seconds should pass between each action the sensor takes (in this case, saving a .ply file).

When starting a data collection session, a file named **ids_labels.txt** was generated. The function which created this file can be seen in figure 3.17.

```
def saveActorLabels(vehicles, pedestrians):
    actors = []
    for vehicle in vehicles:
        actors.append([vehicle.id, vehicle.type_id])
    for pedestrian in pedestrians:
        actors.append([pedestrian.id, pedestrian.type_id])
    f = open('ids_labels.txt', 'w')
    for actor in actors:
        f.write(str(actor[0]) + ',' + str(actor[1]) + '\n')
    f.close()
```

Figure 3.17: SaveActorLabels function

The generated file contained the ID of all the spawned actors, as well as their type_id. The type_id described the actor type, for example **'vehicle.tesla.model3'**. This file was used later to match actor IDs to their respective type when creating labels for the dataset.

Creating images from Lidar point clouds

The Carla LiDAR sensor had several attributes that needed to be tweaked before collecting data:

- **Channels**: This attribute determined the number of vertical lasers. The Ouster images had a vertical resolution of 128 pixels, so this value was set to 128 (one laser per vertical pixel).
- **Range**: Determined the range of the lasers in meters. Ouster used 240m by default, which was chosen for the Carla sensor as well.
- **Upper FOV and lower FOV**: The upper and lower angle of the sensor. The NTNU Ouster sensor had a -11.25 to 11.25 degree field of view according to the project co-supervisor Gabriel Kiss, for a total of 22.5 degrees.
- **Dropoff general rate, Dropoff intensity limit and Dropoff zero intensity**: These values introduce general random loss to the sensor, to mimic real life data loss which can occur. Playing around with these values, it was found that removing the loss completely was the most similar to the Ouster images.
- **Rotation frequency**: How many times the sensor rotated completely per step. This was chosen to be the same as the FPS of the simulation (which

was chosen to be 20, more on that below), to always get a complete 360 degree point cloud. If i.e. half the FPS value was chosen for this setting, the point clouds would be 180 degrees only.

- **Points per second:** How many points the sensor should register per second. The amount of point cloud files (.ply) generated per second always equaled the simulation FPS. This setting was per second, meaning one would need 30 times as many points as one would want for each file. The desired output images had a resolution of 1024x128 pixels, which lead to this setting being set to 1024 * 128 * 20 = **2,621,440**.

Using these settings, 20 .ply files were generated each second. The team discussed how many images it was necessary to collect. Ultimately, it was decided that collecting images at a similar pace to the RGB images was the natural approach (one image every 5 seconds). This would allow for a greater variance in the images, as 20 images each second would result in many very similar images. As mentioned previously, this was controlled via the **sensor_tick** attribute.

The **Points per second** attribute was an especially important factor to consider. Because one could only choose the points per second and not per point cloud file captured, a huge amount of points needed to be collected every second (even if the point cloud is only saved every 5 seconds). The simulation ran at 60 fps by default. According to the documentation it should be possible to run at a fixed time-step by applying the parameters **"-benchmark -fps=X"**, where X is the desired FPS [13]. Unfortunately, the team could not get this to work as expected. A workaround was found by using the NVIDIA Control Panel application [14] on Windows, where a max FPS for all applications on the computer can be set. The minimum configurable FPS was 20, which is what was chosen for the simulation. This would make the simulation FPS match the rotation frequency, and complete 360 degree point clouds could be collected every 5 seconds.

[13] `https://carla.readthedocs.io/en/stable/configuring_the_simulation/`

[14] `https://www.microsoft.com/en-us/p/nvidia-control-panel/9nf8h0h7wmlt?activetab=pivot:overviewtab`

Point cloud file format

The LiDAR sensor generated .ply (Polygon File format) files, with each file representing one point cloud. The format of the .ply files is shown in figure 3.18.

```
ply
format ascii 1.0
element vertex 130041
property float32 x
property float32 y
property float32 z
property float32 CosAngle
property uint32 ObjIdx
property uint32 ObjTag
end_header
-100.9829 0.0000 20.0867 0.8544 0 1
-100.6350 -0.6172 20.0179 0.8574 0 1
-100.2895 -1.2303 19.9503 0.8603 0 1
-99.9463 -1.8393 19.8839 0.8632 0 1
-99.6054 -2.4442 19.8187 0.8660 0 1
-99.2439 -3.0445 19.7501 0.4551 0 1
-68.0406 -2.5051 13.5433 0.9751 0 9
-68.0615 -2.9240 13.5508 0.8938 0 9
-98.1853 -4.8216 19.5538 0.8770 0 1
-68.3465 -3.7767 13.6157 0.7435 0 9
-68.1125 -4.1829 13.5739 0.7984 0 9
-97.2785 -6.5732 19.3940 0.8850 0 1
-69.7607 -5.1438 13.9139 0.2523 0 9
-69.1045 -5.5218 13.7895 0.2094 0 9
-67.9491 -5.8491 13.5659 0.7670 0 9
-67.1546 -6.1959 13.4146 0.8640 0 9
-67.3472 -6.6305 13.4610 0.9458 0 9
-67.3357 -7.0466 13.4670 0.8055 0 9
-69.4461 -7.6984 13.8983 0.8392 0 9
-69.1149 -8.0911 13.8417 0.3274 0 9
```

Figure 3.18: Example of a .ply file generated from the semantic LiDAR sensor

The first lines describe the format of each row value in the file, as well as how many points were contained in the file (the **element vertex** value, in this case 130041). Each line represents a point in the cloud, with the first three values being its X, Y and Z values. For the regular LiDAR sensor, the fourth value was the returned intensity, normalized between 0 and 1. For the semantic LiDAR sensor, the fourth value was the cosine of the incident angle, while the fifth and sixth value represented the actor ID and semantic tag respectively.

Autopilot issue

Gathering the LiDAR data proved to be a computationally intensive task. This lead to some inconsistencies with the script. The primary issue encountered was the player vehicle occasionally getting stuck. It was observed that all of the spawned vehicles would have issues with their autopilots, which did not occur when gathering RGB images. When turning, the vehicles would turn too far, and end up crashing into objects on the side of the road like lamp posts. If there were no objects to crash into, the vehicles would be able to correct their path and keep driving. However, if the player vehicle got stuck, it would still collect .ply files, but the scene would remain the same. This issue rarely happened, and only on the map Town02, where the vehicle always got stuck in the same spot. However, since this behavior was also observed in the other vehicles to some extent, it warranted a mention. It was not determined what caused this issue, but the team believes it happened due to a performance limitation with the Carla simulator, or possibly some issue with the default asynchronous mode.

3.2.6 Generating LiDAR images

This file was responsible for reading the point cloud files, generating the LiDAR images and creating bounding box files for the dataset.

Handling .ply files

The LiDAR sensor saved the point clouds in a .ply file format (See section 3.2.5 for more information on the file format). The team initially used the **open3D**[15] Python package to read the .ply files. This package was found be insufficient, as it could only read the X, Y and Z values, and did not support the reading of additional parameters like the intensity or actor ID values. As such, another method was required. The team ended up using a script from a Github repo[16] by the user **daavoo** which could be used to read .ply files and store the result in a Pandas

[15] `http://www.open3d.org/docs/0.9.0/tutorial/Basic/file_io.html`
[16] `https://github.com/daavoo/pyntcloud/blob/master/pyntcloud/io/ply.py`

Dataframe. As this project used numpy arrays for the data processing, the pandas function **.to_numpy()**[17] was used to convert the dataframe into a numpy array. Each value in the returned array contained another array with the values of each line in the .ply file. This array could then be iterated over to process all the points from the point clouds.

Projecting 3D points to 2D image space

In order to create an image, the 3D point cloud needed to be projected to a 2D image space. Wu et al. describes one method to achieve this [28]. This is done by calculating where the 3D points will fit as pixel coordinates, as well as what the pixel value should be.

Following the paper, the following formulas were implemented in Python. The image range value was calculated using Euclidean vector distance, using the formula:

$$r = \sqrt{x^2 + y^2 + z^2}$$

The azimuth angle was calculated using the formula:

$$\phi = arctan(x/y)$$

The elevation angle was calculated using the formula:

$$\theta = arcsin(z/r)$$

For the column index **u** of the image, this was calculated with the formula:

$$u = \lfloor \frac{1}{2}(1 + \phi/\pi) \cdot w \rfloor,$$

where w was the desired width of the image.

For the row index **v** of the image, the paper presented two choices. The first option was using projection by laser ID (PBID), which assigned each laser to its corresponding image row, with the elevation angle being equal to the laser angle. The other option was projection by elevation angle (PBEA), where each row value was

[17] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_numpy.html`

calculated based on the minimum and maximum elevation angles. The formula for calculating the value was as follows:

$$v = \lfloor(\theta_{up} - \theta)/(\theta_{up} - \theta_{down}) \cdot h\rfloor,$$

where h was the desired height of the image.

The paper found that PBEA gave a more consistent and accurate result, so this method was chosen.

One issue was found regarding the calculation of the row index **v**. When using the floor operator, an entire row in the middle of the image would not get indexed. After examination, it seemed like all the calculated values before flooring would be just above .5 in their respective bracket, including for 0. Because of this, the row on angle 0 would not get any values. This was fixed by rounding the values instead of flooring, which caused a more even distribution of values.

For the pixel intensity values, this was initially found by multiplying the normalized value from the .ply file generated by the normal LiDAR sensor by 255 to receive a pixel value between 0 and 255. However, as this value was not provided by the semantic LiDAR sensor, one would have to run the normal LiDAR and semantic LiDAR sensors concurrently and match the points to get the correct intensity values. Luckily, some testing showed that the inverted range image was identical to the calculated intensity image. Because of this, the final version of the script simply inverted all the calculated range values to create the intensity image.

After obtaining index values for the rows and columns of the image, the range/intensity values were set at these values. Figure 3.19 shows the function **processPointCloud()** which handles everything explained in this section up until this point:

```
def processPointCloud(filename):
  img_range = np.zeros((1024, 128), dtype=np.uint8)
  img_seg = np.zeros((1024, 128, 3), dtype=np.uint8)
  img_labeled = np.zeros((1024, 128))
  img_tag = np.zeros((1024, 128))
  pcd_points = readPLY(filename)['points'].to_numpy()
  pcd_array = []
  for point in pcd_points:
    x = point[0]
    y = point[1]
    z = point[2]
    object_id = point[4]
    object_tag = point[5]
    pcd_array.append([x, y, z])

    r = np.linalg.norm([x, y, z])
    r_norm = (r/240) * 255
    azimuth = math.atan2(x, y)
    elevation = math.asin(z / r)
    elevation_up = math.radians(11.25)
    elevation_down = math.radians(-11.25)

    u = round(0.5 * (1 + azimuth/math.pi) * 1024)
    v = math.floor(((elevation_up - elevation) / (elevation_up - elevation_down)) * 128)

    img_range[u-1][v-1] = r_norm
    img_seg[u-1][v-1][:] = seg_colors[object_tag]
    img_labeled[u-1][v-1] = int(object_id)
    img_tag[u-1][v-1] = object_tag

  img_intensity = np.invert(img_range)
  return img_range, img_intensity, img_seg, img_labeled, img_tag
```

Figure 3.19: processPointCloud() function

As mentioned, the file reads a .ply file, calculates [u,v] pixel coordinates and fills these coordinates with range and intensity values. The **img_labeled** array was later used to annotate the images (essentially acting as an instance segmentation image mask, but using IDs instead of colors), while the **img_tag** was used to create semantic segmentation images to visualize results.

The images generated were then saved to disk using the **imsave()** function from the **matplotlib.pyplot**-package. Figure 3.20 shows an example of a typical generated range and intensity image.

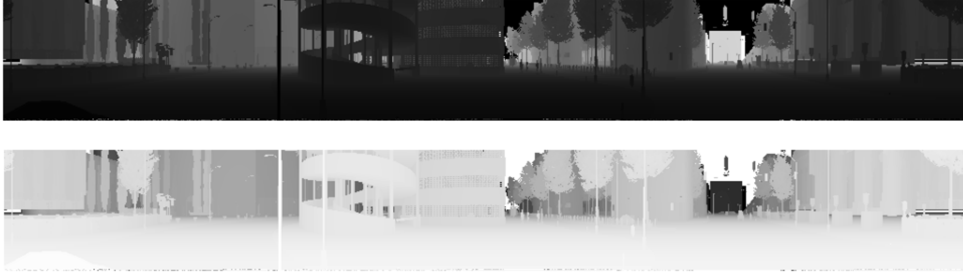


Figure 3.20: Example range image (top) and intensity image (bottom)

An issue was encountered with the images, which can be observed in the figure above. Towards the bottom the LiDAR images got noisy, with some values not being filled in. While this only happened for the bottom 3-4 rows of pixels, the team did some testing in an attempt to correct this issue. The bottom pixels corresponded to the point cloud coordinates closest to the sensor, which meant the points had very similar coordinates. The further away from the sensor the points were, the more spread out they became. As the 2D projection was an imperfect estimation and values were rounded, this lead to many values overwriting one another instead of filling every available space. The team believed this to be a limitation of the Carla sensor, as well as the method of 3D to 2D projection. As this only happened in the lower part of the images and did not significantly impact visibility, the team decided to ignore this issue for the data collection.

Automatic annotation for Lidar images

As mentioned previously, one of the advantages of using a simulator to collect data is that one has full control over all objects in the scene, which can be used for automatic annotation. Manual annotation is time consuming and tedious, so being able to do this automatically would save time.

A similar method to the RGB images was applied when generating the LiDAR bounding boxes. While the semantic LiDAR sensor did not provide instance colors for each object, it did provide the actor ID directly. As such, a slightly different approach had to be used compared to the RGB images.

While projecting the 3D points to 2D image coordinates, an array named **img_labeled** was also created with the same dimensions as the input image. This array was then filled with the actor IDs obtained from the .ply file. What was created was effectively an image mask of the actor IDs. For the next step, the **ids_labels.txt** file was iterated over, and for each actor ID the entire image was scanned. The relevant min/max X and Y values were gathered, which represented the boundaries of the bounding box. A check was implemented to ensure that only objects that were present in the image were checked (as the ids_labels.txt file contained all the spawned actors in the entire simulation).

A case was found where certain objects could appear on both sides of the image. Since the image was 360 degrees, an actor could appear split on both ends of the image horizontally. This would cause issues with said objects' bounding box, as the bounding box would stretch across the entire image. To avoid this, any object where the difference between max and min X was above 900 pixels were ignored. Another option would be to treat these two parts as separate objects, but this seemed like a sub-optimal solution from an annotation perspective.

As with the RGB images, bounding boxes that were too small were removed. However, unlike for the RGB images, where a percentage of the x and y values was chosen as a threshold for removal, the area of the bounding box was calculated and used as a threshold. Compared to the RGB images, the LiDAR images had more slim and tall objects, particularly the pedestrians. Using an area as the threshold preserved these bounding boxes while still removing small boxes that could be a detriment to model training. Figure 3.21 shows the effect of using the threshold on a range image. Note the smaller objects being removed in the bottom image.

Figure 3.21: Lidar range image without bounding box threshold (top) vs with threshold (bottom)

Occluded bounding boxes for Lidar images

Using the bounding boxes retrieved from the simulation, it would be possible to draw full boxes for objects partially occluded by other objects. This was successfully done for the RGB images (see section 3.2.4), but proved to be difficult for the LiDAR images. A similar method was attempted to transform the bounding box coordinates to the sensor coordinates. If one could get the bounding box coordinates to match values from the LiDAR sensor cloud points, one could use the same method for projecting the 3D points to 2D image coordinates to get 2D bounding boxes directly on the image. However, unlike for the camera sensor, there was no camera calibration available. The 3D to 2D projection warps the image towards the edge, and this was observed to happen with the bounding boxes as well. This meant that objects close to the center of the image would have correct bounding boxes, while ones closer to the edge would have somewhat shifted bounding boxes. See figure 3.22 for an example:

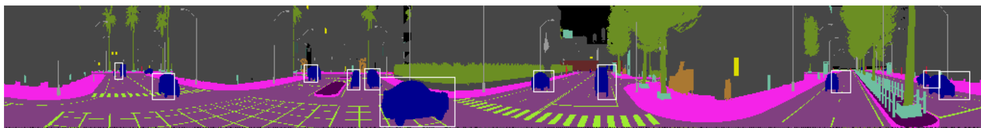


Figure 3.22: Segmentation LiDAR image showing the shifted bounding boxes

The team was not able to find a solution to this issue. The majority of found bounding boxes were shifted enough to be unusable, which caused this part of the project to not be implemented. Instead, a purely visual instance segmentation

was done, using only the labeled image as described above. This worked well most of the time, only failing to get bounding boxes occluded by other objects in the scene. As the LiDAR images were to be used mostly as secondary training data, this was considered acceptable by the team.

3.2.7 Data collection pipeline

The team created a pipeline to collect the training datasets. First the Carla server needed to be started, and town and weather conditions for the session were determined. To change towns, an included Carla configuration file **config.py** was executed by issuing the command:

```
python config.py --map ${preferred town}
```

The weather conditions were changed by using the included **environment.py** file and the command:

```
python environment.py --weather ${preferred weather condition}
```

After changing the desired map and weather conditions, the actors needed to be spawned. The actors were spawned using the command:

```
python spawn.py
```

where the number of actors was modified in the file directly. These could also be passed as parameters if necessary.

The amount of actors depended on the size of the map that data was collected from. If not enough actors were spawned in relation to the map size, many of the collected images would have few or no objects in view most of the time due to them being spread out. To ensure a consistent number of labels in the collected images, maps with a relatively similar size was chosen. The chosen maps were **Town01**, **Town02** and **Town11**.

After setting up the Carla simulator with actors, map and weather conditions, the next step was to run the code for extracting the data. For the RGB images, this was done by executing the command:

```
python ExtractRGB.py
```

This opened a secondary pygame window. The team would press "P" to enable the auto-pilot and then "l" to start capturing images. Every five seconds the script would save an instance segmentation image, RGB image and bounding box information for all the actors present in the current image view of the simulator.

The next step was to run the command:

```
python GenerateRGB.py
```

which looped through all of the above images, tightened the bounding boxes and saved these in the Darknet format (see section 3.3 for more information on the dataset formats).

For the LiDAR data collection, the process was similar. However, there was no need to manually start the player vehicle or saving of images, as this happened automatically.

Real RGB images for testing

The team needed a dataset with properly annotated real images to test the models. A popular dataset used for 2D traffic object detection is the **Kitti** dataset[18]. It contains roughly 7500 annotated images. This dataset was chosen due to the quality of the annotations, as well as it having annotations for all the relevant label types used in this project. For the Carla dataset, the team initially collected 500 test images to review the dataset before collecting a larger sample. The images were all collected from Town11.

[18] http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d

Real LiDAR images for testing

As with the RGB images, a test dataset was needed to test the generated LiDAR images. During the 2021 fall semester, the team members completed the course Visual Intelligence[19] (subject code TDT17) in preparation for the master thesis. The course included a mini-project, with the aim of performing object detection on annotated LiDAR images from the NTNU Ouster sensor. The data was provided in the form of short video files with a framerate of 30 FPS. Annotations were provided in several different formats, including Darknet and COCO. The annotations were divided into 4 classes; **car**, **person**, **rider (cyclist)** and **bus**.

The videos were divided into single frames using simple Python scripts to create a dataset consisting of images. The team decided to use Yolov5 as the object detection model for the project. After some discussion with the project supervisor Frank Lindseth, he suggested using this dataset to test the LiDAR images generated from the Carla point cloud files. Roughly 5000 images were provided during the project, which the team considered sufficient to use for training, validating and testing purposes.

One issue that had to be addressed was the mismatching labels between the simulated and real data. The data provided for the project did not contain any **truck** class, however it did contain a **bus** class. The team decided to simply change the class name in the project dataset, with the reasoning that some of the trucks in Carla like the fire truck or ambulance look fairly similar to buses.

Dataset format conversion

The team decided to use two object detection models to test the generated datasets; Yolov5 and Faster R-CNN. Yolov5 uses the **Darknet Pytorch** dataset format, while Faster R-CNN through MMDetection uses the **COCO** format. See section 3.3 for more information on the specific formats. **GenerateRGB.py** and **GenerateLidar.py** only saved the bounding box information to the regular Darknet

[19] `https://i.ntnu.no/wiki/-/wiki/Norsk/TDT17+-+Visual+Intelligence`

format. To convert between these formats, a website called Roboflow was used[20]. Roboflow specializes in CV tasks, having image annotation, dataset splitting and dataset format conversions available online. While they have paid models, their free plan was sufficient for this project. The datasets were uploaded in regular Darknet format to the website and converted and downloaded in the Darknet Pytorch and COCO formats respectively.

Dataset imbalance

The team wanted to make sure that the collected datasets did not suffer from significant class imbalance. According to Oksuz et al. [29], an imbalanced dataset can have adverse effects on the final detection performance of object detections models. As mentioned previously, a test dataset consisting of 500 images was initially collected. The team created a simple Python script to count the number of labels in the dataset, seen in figure 3.23:

[20] `https://roboflow.com/`

```
def countLabels():
  path = 'train/labels'
  car_count = 0
  truck_count = 0
  cyclist_count = 0
  pedestrian_count = 0

  with os.scandir(path) as it:
      for entry in it:
          fo = open(entry.path, 'r')
          lines = fo.readlines()
          for line in lines:
            if line[0] == '0':
              car_count = car_count + 1
            if line[0] == '1':
              truck_count = truck_count + 1
            if line[0] == '2':
              cyclist_count = cyclist_count + 1
            if line[0] == '3':
              pedestrian_count = pedestrian_count + 1

  print('Car count:', car_count)
  print('Truck count:', truck_count)
  print('Cyclist count:', cyclist_count)
  print('Pedestrian count:', pedestrian_count)
```

Figure 3.23: Python script to count the number of different labels for the project datasets

The script iterated over all the .txt label files and counted the number of different labels (label type was determined by the first number on each line). At the end the result would be printed to console.

Using this script, it was discovered that the 500 image test dataset suffered from a significant class imbalance, which is illustrated in figure 3.24.

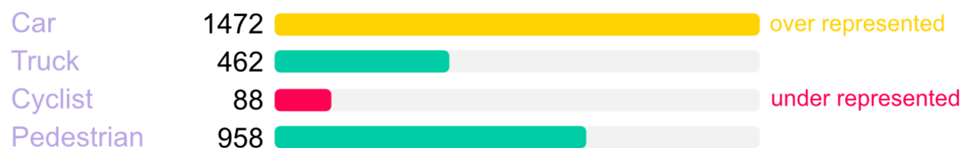


Figure 3.24: Class imbalance for 500 image test dataset

As the figure shows, the **cyclist** class was severely under represented in the dataset. Additionally, **car** was over represented. The team investigated the origin of the class imbalance in an attempt to fix it. The issue was determined to be with the **spawn.py** file, used to spawn actors in the simulation. As mentioned, Carla separated actors into two categories: vehicles and pedestrians. Both trucks and cyclists counted as vehicles, as could be seen from their **type_id**, for example with the cyclist type_id "**vehicle**.diamondback.century". When running the **spawn.py** file, it would randomly select vehicles from the list of vehicle blueprints and spawn these. The Carla blueprint library contained a variety of vehicle types, with the following counts:

- **Car:** 25
- **Truck:** 7
- **Cyclists:** 3

The library had significantly more car blueprints than trucks or cyclists. Because blueprints were randomly chosen, this would lead to fewer trucks and cyclists being spawned.

To fix this issue, the team decided to balance the spawning. This was done by modifying the **spawn.py** file to spawn a certain amount of each type of vehicle. A list was made of all the blueprint names of the different vehicles, seen in figure 3.25. The team decided that 20 of each vehicle type was a sufficient amount for the size of the selected maps, for a total of 60 vehicles.

```
car_list = ['audi.a2', 'audi.etron', 'audi.tt', 'bmw.grandtourer', 'chevrolet.impala', 'citroen.c3',
'dodge.charger_2020', 'dodge.charger_police', 'dodge.charger_police_2020', 'ford.crown', 'ford.mustang',
'jeep.wrangler_rubicon', 'lincoln.mkz_2017', 'lincoln.mkz_2020', 'mercedes.coupe', 'mercedes.coupe_2020',
'micro.microlino', 'mini.cooper_s', 'mini.cooper_s_2021', 'nissan.micra', 'nissan.patrol', 'nissan.patrol_2021',
'seat.leon', 'tesla.model3', 'toyota.prius']
truck_list = ['carlamotors.carlacola', 'carlamotors.firetruck', 'ford.ambulance', 'mercedes.sprinter', 'tesla.cybertruck',
'volkswagen.t2', 'volkswagen.t2_2021']
bicycle_list = ['bh.crossbike', 'diamondback.century', 'gazelle.omafiets']
```

Figure 3.25: Blueprint lists from spawn.py

After updating the file (also renaming it to **spawnBalanced.py**, a sample of 500 images were collected from Town11 to test the new class balance. After counting the labels, the team discovered there was still some class imbalance, as can be seen in figure 3.26.

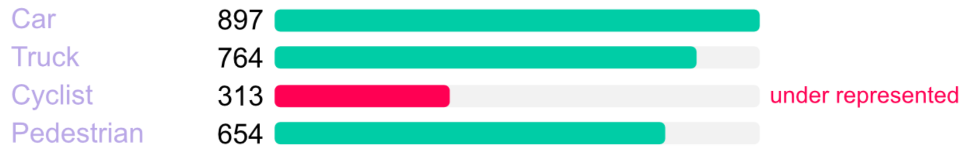


Figure 3.26: Class imbalance for 500 image test dataset with balanced spawning

However, it was an improvement over the original dataset. While **cyclist** was still under represented, it was significantly improved over the previous version. **Car** was also not tagged as over represented anymore.

Inspecting the generated images, it was discovered that the cyclists would not consistently have bounding boxes drawn. While the bounding boxes did appear to be extracted from the simulation, their coordinates were often inaccurate, to the point where the bounding box would not appear inside the camera view. This was determined to be an issue with the Carla simulator itself, due to the inconsistency and it not happening with the cars, trucks and pedestrians. The team did not manage to find a solution to this problem. See section 5.3 for further discussion.

The LiDAR images were collected after addressing the class imbalance. The label count from the complete LiDAR dataset can be seen in figure 3.27:

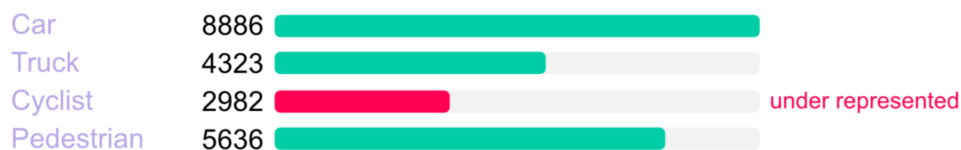


Figure 3.27: Class imbalance for the Carla LiDAR training dataset (4000 images)

While the a class imbalance still existed (primarily for the **cyclist** class), it was much more consistent compared to the original dataset.

After the imbalance was addressed, 7500 images from the Carla simulator were gathered to match the Kitti dataset size. The dataset had a similar class imbalance

to the 500 image test dataset. The images were equally split between the three maps, with each map having three separate weather conditions: clear, night and rain. Both datasets were split, with 80% reserved for training, 10% for testing and 10% for validation.

For the data collection of LiDAR images from the Carla simulator, the team opted for a total of 5000 images to match the testing dataset size, with 2500 range and 2500 intensity images. This was done by collecting an equal amount of .ply files from the same three towns as the RGB images. The LiDAR images would not be affected by world conditions like rain and lighting, so these settings were not considered. The same 80/10/10 split was used for this dataset.

3.3 Training and testing object detection models

3.3.1 Yolov5

One of the reasons for picking Yolov5 for this project was its ease of use in everything from training on custom data to testing the trained models. The team also had previous experience using it for the mini-project during the TDT17 course. First, the Yolov5 Github repository[21] was cloned. Requirements were installed using pip with the included requirements.txt file.

Dataset format

Yolov5 uses the Pytorch Darknet format, where train/test/valid are separated into folders, with sub-folders separating the images into one folder and labels in another. The bounding box information and labels are saved in a .txt format, with each image having a corresponding label file with the same name. See figure 3.28 for an example.

[21] `https://github.com/ultralytics/yolov5`

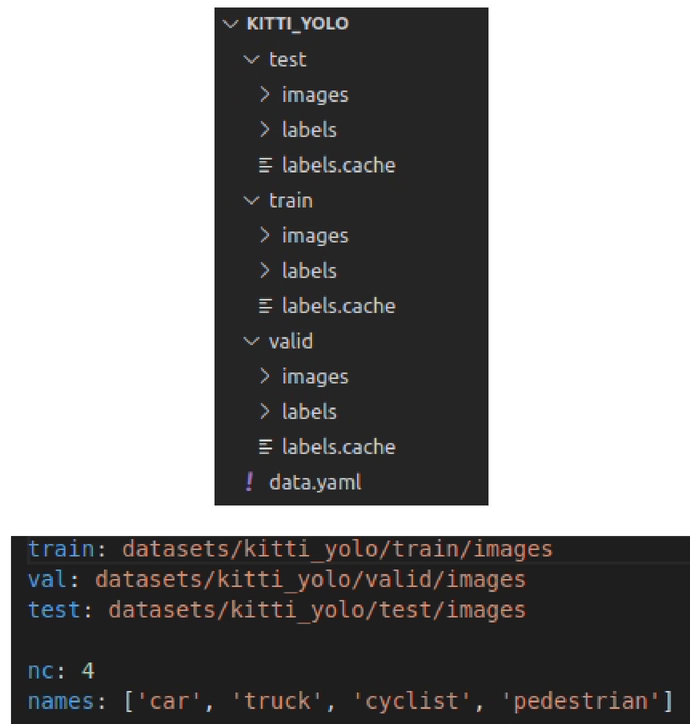
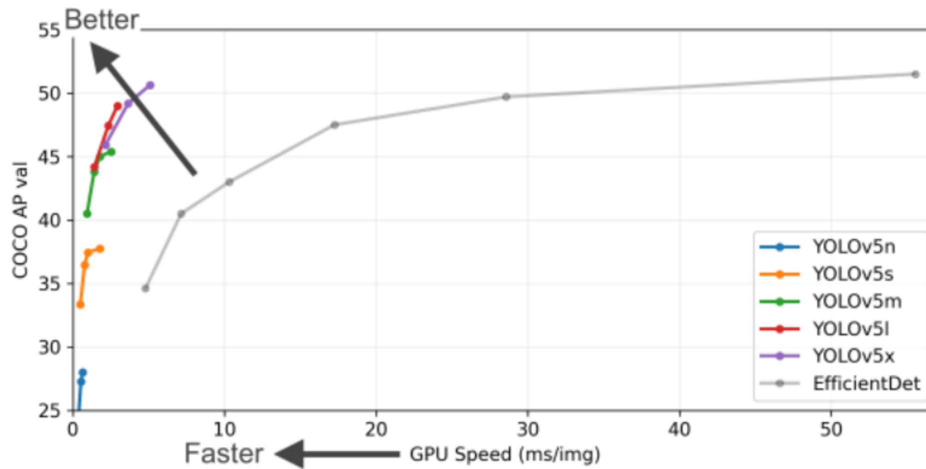


Figure 3.28: Yolov5 Pytorch dataset structure (top) and .yaml file contents (bottom)

The .yaml file contains the file paths to each part of the dataset, along with the number of classes and their names respectively.

Training a Yolov5 model

There are several versions of Yolov5 available; Yolov5n (nano), Yolov5s (small), Yolov5m (medium), Yolov5L (large) and Yolov5X (XL). The primary difference is a tradeoff in speed vs. accuracy. The smaller models like nano and small could run inference (detect objects in images) faster, but are not as accurate as the larger models. Figure 3.29 shows a comparison between the speed and AP on the COCO dataset.



Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7

Figure 3.29: Differences between Yolov5 variations. Source: `https://github.com/ultralytics/yolov5`

The team decided that a balance between speed and accuracy was the best choice, resulting in **Yolov5m** being chosen.

To start training, the included **train.py** file from the Yolov5 Github repository had to be executed. The file required parameters in the form of **–data**, which should point to the path of the .yaml file of the chosen dataset. It also required the choice of Yolov5 model, passed via the **–cfg** argument.

The file also had a variety of optional parameters. In order to train a model from scratch, this could be done by passing an empty string as the **–weights** parameter. Otherwise, pretrained COCO weights for the corresponding model would be downloaded and used. Specific weight files could also be passed using this argument to do transfer learning on a model trained on another dataset. Additionally,

other parameters such as batch size and number of epochs could be specified.

An example of the full command issued to start training:

```
python train.py --data datasets/kitti/data.yaml --cfg yolov5m
--weights ''
```

Testing a Yolov5 model

For testing the trained models, the included **val.py** file was used. The file took similar arguments to train.py: **–data** being the .yaml file of the dataset used as test data, and **–weights** being the weights of the trained model. One important additional argument was **–task**, which could be used to specify what type of task the script should carry out. By default, the task was **val**, meaning the model would be tested on the validation part of the dataset. The team wanted to use the **test** part of the dataset, which was achieved by passing **–task test** to the file.

An example of the full command issued to test the trained models:

```
python val.py --data datasets/kitti/data.yaml
--weights runs/kitti300epochs/weights/best.pt --task test
```

3.3.2 Faster R-CNN via MMDetection

MMDetection[22] is an object detection framework for working with different object detection models. The framework supports several different object detection models. The team opted for Faster R-CNN, because there was a need to train the model from scratch, and this model was already implemented from scratch in the included files from the MMDetection repository. Faster R-CNN is also a widely documented and used object detection model.

[22] https://github.com/open-mmlab/mmdetection

Dataset format

Faster R-CNN through MMDetection used the COCO format, having a folder structure as shown in figure 3.30. It was important to correctly specify file and folder paths in order to get the code running correctly.

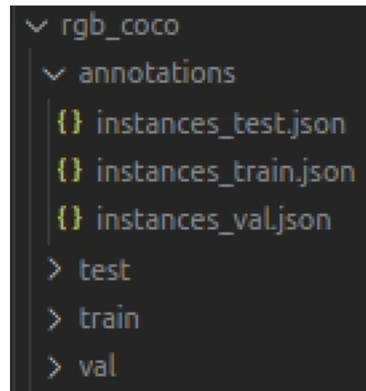


Figure 3.30: Folder structure of the COCO format used in MMDetection Faster R-CNN

As mentioned in section 3.2.7, a dataset split of 80% train, 10% test and 10% validation images was used. Using the COCO format, information about the images and their bounding boxes was located in the annotation .json files. These files contained the file name of every image within the database, shown in figure 3.31, along with all the annotations, shown in figure 3.32. The images and annotations were linked with a unique ID.

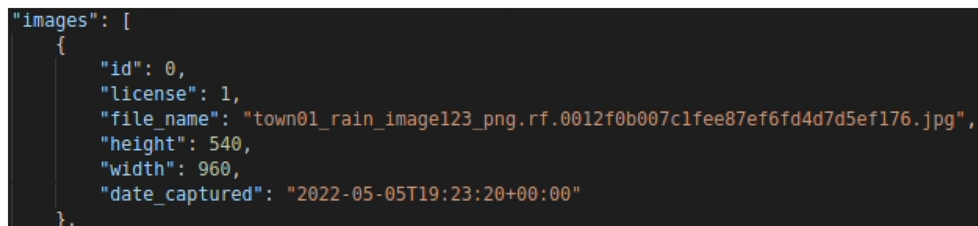


Figure 3.31: Example of a image information within a .json file

```
"annotations": [
    {
        "id": 0,
        "image_id": 0,
        "category_id": 1,
        "bbox": [
            240,
            168,
            478,
            370
        ],
        "area": 176860,
        "segmentation": [],
        "iscrowd": 0
    },
```

Figure 3.32: Example of an object and its information within a .json file

Training from scratch using the framework proved to be challenging. There was little to no documentation on how to fix errors that occurred or what files to edit in order to make the code run properly. The training tutorial[23] only mentioned what code to run from terminal and therefore indirectly what files had to be used. Several files in different locations needed to be changed to train models successfully. See appendix A for detailed instructions.

In order to start training, the following command was executed:

```
python tools/train.py ${configuration file}
```

To be able to do transfer learning with MMDetection, the line **load_from = None** in the **default_runtime.py** file needed to be changed to the desired .pth file. This line allows the user to load a weights file (in .pth format), and then continue training on top of that file.

[23]`https://github.com/open-mmlab/mmdetection/blob/master/docs/en/1_exist_data_model.md`

Testing Faster R-CNN with MMDetection

To test models with MMDetection, the following command needed to be executed from terminal:

```
python tools/test.py ${configuration file} ${checkpoint file}
```

The configuration file being same one that was used during training, whereas the checkpoint file being weights file generated after training was finished.

3.3.3 Metric evaluation

Testing both Yolov5 and Faster R-CNN through MMDetection produced a variety of information and metrics about the models. A common way to measure a models' performance is by using its mAP score (see section 2.2.3 for details). The team decided to use this as the main metric to compare the results from the experiments. Average Precision for each class was also measured, as it could show discrepancies among different classes and how they could affect the mAP value.

Each dataset had its own validation set, which could provide the end-user with useful metrics when the training was finished. These metrics could also be used during training to fine-tune the model. However, the metrics were calculated using the validation set of the dataset itself, making it potentially less relevant in regards to the research questions. The mAP training values for a model trained exclusively on Carla would likely not be very useful when the objective is to check how well a model trained on simulator data can perform on a real dataset. Instead, the models were tested using the test portions of the Kitti and Ouster LiDAR datasets.

Chapter 4

Experiments & Results

This section details the experiments that were carried out in an attempt to answer the research questions posed in section 1.2. Section 4.1 explains the baseline models which were used for comparison. The following sections present the eight experiments which were conducted. Experiments 1, 2 and 3 addresses RQ1, while experiment 4 and 5 addresses RQ2. Experiment 6 and 7 addresses RQ3 and RQ4 respectively. Finally, experiment 8 attempts to answer RQ5.

4.1 Training baseline models

Before any experiments were conducted, four baseline models were created; two Yolov5 models and two Faster R-CNN models, both using RGB and LiDAR data from the hand-annotated datasets. As mentioned previously, the Kitti dataset contained roughly 7500 annotated images. As per the 80/10/10 split, 6000 of the images were used to train the baseline models, with the remaining 1500 images divided equally into the validation and testing sets. For the Ouster LiDAR data, the team had access to 5000 images, of which 4000 were used for training, and 500 each for test/valid. All trained models were tested on the test portion of their respective hand-annotated datasets.

Experiment parameters:

- **Training dataset:** Kitti (6000 images) and Ouster (4000 images)
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)

Table 4.1 shows the mAP scores for the baseline models:

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Yolov5 (Kitti)	0.936	0.703	0.812	0.822	0.680	0.497
Faster R-CNN (Kitti)	0.864	0.564	0.702	0.688	0.482	0.382
Yolov5 (Ouster)	0.877	0.522	0.691	0.498	0.475	0.425
Faster R-CNN (Ouster)	0.723	0.443	0.635	0.445	0.346	0.347

Table 4.1: Results from baseline models trained on Kitti RGB and Ouster LiDAR images.

The **mAP0.5** score is the mAP score (see section 2.2.3 for details) at an IoU threshold of 0.5. **mAP0.5:0.95** is the standard COCO performance measurement, being the average mAP across 10 IoU thresholds from 0.5 to 0.95. The four last columns show the individual AP scores for each class.

For the baseline models, Yolov5 performed better than Faster R-CNN across all metrics. In particular, the **cyclist** class got a relatively higher score for Yolov5. As mentioned, these scores represented the baseline against which the other experiments were compared.

4.2 Experiment 1: Small RGB dataset, default settings

The purpose of this experiment was to observe how the two models performed with their default settings and a relatively small dataset, consisting of 6000 training images from Carla to match the Kitti dataset. The Yolov5 model was trained using 300 epochs, while Faster R-CNN was trained using 30 epochs, being the default amount for each model.

Experiment parameters:

- **Training dataset:** Carla RGB (6000 images)
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
Yolov5	0.168	0.078	0.221	0.052	0.005	0.037

Table 4.2: Yolov5 RGB results compared to Yolov5 baseline.

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
Faster R-CNN	0.150	0.065	0.151	0.052	0.005	0.052

Table 4.3: Faster R-CNN RGB results compared to Faster R-CNN baseline.

As table 4.2 and table 4.3 show, the results were poor compared to the baseline for both models. Comparing the two models, Yolov5 performed slightly better on **mAP0.5**, **mAP0.5:0.95** and the **Car** class. For **Truck** and **Cyclist** both models performed equally, whereas Faster R-CNN performed slightly better on **Pedestrian**.

4.3 Experiment 2: Increase number of epochs

This experiment was conducted to investigate if increasing the number of epochs would affect model metrics. The team chose to train two versions of each model using an increased amount of epochs and comparing them to the baseline and the default epochs for each model. Yolov5 was trained using 600/900 epochs and Faster R-CNN using 60/90 epochs.

Experiment parameters:

- **Training dataset:** Carla RGB (6000 images)

- **Testing dataset:** Kitti dataset
- **Epochs:** 300/600/900 (Yolov5), 30/60/90 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
300 epochs	0.168	0.078	0.221	0.052	0.005	0.037
600 epochs	0.189	0.088	0.235	0.060	0.008	0.049
900 epochs	0.190	0.091	0.237	0.061	0.009	0.049

Table 4.4: Yolov5 results compared to the Yolov5 baseline.

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
30 epochs	0.150	0.065	0.151	0.052	0.005	0.052
60 epochs	0.145	0.062	0.136	0.048	0.001	0.065
90 epochs	0.122	0.057	0.148	0.042	0.000	0.035

Table 4.5: Faster R-CNN results compared to Faster R-CNN baseline.

Table 4.4 shows that using 900 epochs for Yolov5 resulted in slightly better scores than using 600 epochs, and they both outperformed the baseline 300 epochs. **Pedestrian** was the only class that did not get an increased score for 600 and 900 epochs.

In table 4.5, Faster R-CNN trained using 90 epochs produced slightly worse results than 60 epochs and the default 30 epochs, except from the **car** class, which achieved a marginally better score. This was possibly due to overfitting, which is discussed more in chapter 5.1.1.

4.4 Experiment 3: Increase dataset size

The next experiment was conducted to see if changing the amount of training data could affect the model performance. As mentioned in section 4.2, the amount of images for the baseline training was 6000. The team decided to double the size of the training set to 12000 images in order to investigate how it might affect the training.

Experiment parameters:

- **Training dataset:** Carla RGB (6000/12000 images)
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
6K images	0.168	0.078	0.221	0.052	0.005	0.037
12K images	0.178	0.087	0.247	0.052	0.006	0.043

Table 4.6: Bigger dataset for Yolov5 compared to Yolov5 baseline.

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
6k images	0.150	0.065	0.151	0.052	0.005	0.052
12k images	0.247	0.114	0.277	0.095	0.002	0.083

Table 4.7: Bigger dataset for Faster R-CNN compared to Faster R-CNN baseline.

Both table 4.6 and 4.7 show an improved score across most metrics when training with a bigger dataset. For Yolov5, the scores improved across all classes apart from **truck**, which did not change. Most of the scores improved by between 10-20%. The improvement was more significant for Faster R-CNN, nearly doubling its scores on every class except from **cyclist**, which worsened.

4.5 Experiment 4: Fine-tuning with Kitti images

Experiment 4 was conducted to investigate whether the Carla data could be used to train a baseline model, which could then be fine-tuned using a smaller subset of the Kitti data. 2000 images were selected randomly from the Kitti training dataset, and transfer learning was done on top of pre-trained Carla weights (6000 images, 300/30 epochs). Models were also trained from scratch using the same 2000 Kitti images to check if the pre-trained weights had any effect at all.

Experiment parameters:

- **Training datasets:** Carla RGB (6000 images), Kitti (2000 images)
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
Kitti 2k	0.863	0.592	0.748	0.711	0.537	0.374
Kitti Transfer	0.898	0.644	0.776	0.764	0.591	0.447

Table 4.8: Baseline vs. Kitti trained on 2k images vs. Kitti transfer learned on Carla for Yolov5

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
Kitti 2k	0.071	0.016	0.027	0.014	0.021	0.003
Kitti Transfer	0.089	0.024	0.045	0.034	0.015	0.002

Table 4.9: Baseline vs. Kitti trained on 2k images vs. Kitti transfer learned on Carla for Faster R-CNN

As table 4.8 shows, the Yolov5 model trained using the pre-trained Carla weights showed an improvement across all metrics compared to the one trained solely on

2000 Kitti images. While not reaching the performance of the 6000 image Kitti model, the improvement was significant. The **mAP0.5:0.95** score improved by roughly 10%, while the **Pedestrian** class showed the biggest improvement overall.

Comparatively, Faster R-CNN performed poorly. The scores did improve from the 2K Kitti model, but they were still far off compared to the Yolov5 model. The team initially suspected something went wrong during training, and training was repeated twice to check if something was done incorrectly. Despite this, the results did not change.

4.6 Experiment 5: Fine-tuning on pre-trained COCO models

Experiment 5 was conducted to investigate whether models pre-trained on the COCO dataset could be used to enhance the data collected from Carla. For Yolov5, the pre-trained weights were automatically downloaded when specifying their usage in training, while for Faster R-CNN they were included in the cloned repository.

Experiment parameters:

- **Training datasets:** Carla RGB (6000 images), Pretrained COCO weights
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
Carla RGB	0.168	0.078	0.221	0.052	0.005	0.037
COCO weights + Carla RGB	0.283	0.144	0.352	0.13	0.005	0.088

Table 4.10: Baseline vs. Carla RGB vs. Carla RGB transfer learned on pretrained COCO weights for Yolov5

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
Carla RGB	0.150	0.065	0.151	0.052	0.005	0.052
COCO weights + Carla RGB	0.190	0.074	0.181	0.063	0.001	0.050

Table 4.11: Baseline vs. Carla RGB vs. Carla RGB transfer learned on pretrained COCO weights for Faster R-CNN

Table 4.10 shows the result for the Yolov5 model. A substantial increase in mAP can be seen, even doubling the value for the **mAP0.5:0.95** metric. The only class which did not benefit from the pre-trained weights was **cyclist**. However, the results for Faster R-CNN seen in table 4.11 differ, where cyclist did not improve with the transfer learning. It also showed an improvement across most categories, though notably **Pedestrian** got a poorer score when pre-trained on COCO. It also improved less overall compared to Yolov5.

4.7 Experiment 6: LiDAR images from scratch

This experiment was performed identically to Experiment 1, replacing the RGB Kitti images with Ouster LiDAR images. Default settings were used for both models, and the training set consisting of 4000 Ouster images was used. The models were tested against the test part of the Ouster dataset.

Experiment parameters:

- **Training datasets:** Carla LiDAR (4000 images)
- **Testing dataset:** Ouster dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Ouster dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.877	0.522	0.691	0.498	0.475	0.425
LiDAR	0.038	0.018	0.048	0.001	0.005	0.015

Table 4.12: Yolov5 LiDAR results compared to Yolov5 baseline.

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.723	0.443	0.635	0.445	0.346	0.347
LiDAR	0.040	0.019	0.056	0.001	0.002	0.016

Table 4.13: Faster R-CNN LiDAR results compared to Faster R-CNN baseline.

As table 4.12 and 4.13 show, both models performed quite poorly compared to the baseline trained on Ouster image data. The **truck** class in particular received a very poor score, which is understandable considering that these labels were marked as **bus** in the baseline Ouster dataset (see section 3.2.7 for more information).

4.8 Experiment 7: Fine-tuning with LiDAR images

Experiment 7 was conducted to investigate whether the Carla LiDAR images could be used to reduce the amount of real images required when training with LiDAR images. A subset of the Ouster training dataset consisting of 2000 randomly chosen images were used to fine-tune the models from experiment 6 by doing transfer learning. Models were also trained using the 2000 images alone to check whether the transfer learning had any effect.

Experiment parameters:

- **Training datasets:** Carla LiDAR (4000 images), Ouster (2000 images)
- **Testing dataset:** Ouster dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Ouster dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.877	0.522	0.691	0.498	0.475	0.425
Ouster 2k	0.631	0.346	0.442	0.469	0.245	0.202
Ouster Transfer	0.670	0.402	0.476	0.569	0.305	0.250

Table 4.14: Baseline vs. Ouster trained on 2000 images vs. Ouster transfer learned on pretrained Carla LiDAR weights for Yolov5

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.723	0.443	0.635	0.445	0.346	0.347
Ouster 2k	0.004	0.004	0.008	0.002	0.001	0.004
Ouster Transfer	0.008	0.005	0.010	0.002	0.002	0.006

Table 4.15: Baseline vs. Ouster trained on 2000 images vs. Ouster transfer learned on pretrained Carla LiDAR weights for Faster R-CNN

For Yolov5, the fine-tuned model saw substantial gains in its metrics compared to the model trained only on the 2k Ouster images. As table 4.14 shows, all scores increased by between 5-14%, with the **truck** class seeing the largest increase. However, the scores did not reach the baseline, and the gains were not as significant as the Kitti model in experiment 4 4.5.

Similar to experiment 4, transfer learning on Faster R-CNN with 2000 Ouster images produced poor results. Again, training was repeated twice to see if the metrics would change, which they did not. The team was not able to find a solution to these poor scores, though it only seemed to happen when the training set was small.

4.9 Experiment 8: Comparing bounding box tightness

The final experiment was carried out to investigate whether the tightened bounding boxes developed as part of this project had any impact on model performance. To test this, 6000 RGB images were collected using the original CarFree code (apart from having the extra classes added) and models were trained us-

ing the same parameters as the base Carla RGB models (6000 images, 300/30 epochs). The results were compared to the baseline and the trained models from experiment 1.

Experiment parameters:

- **Training datasets:** Carla RGB (6000 images), CarFree (6000 images)
- **Testing dataset:** Kitti dataset
- **Epochs:** 300 (Yolov5), 30 (Faster R-CNN)
- **Baseline:** Trained on Kitti dataset, 300 epochs (Yolov5) 30 epochs (Faster R-CNN)

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.936	0.703	0.812	0.822	0.680	0.497
CarFree	0.074	0.028	0.065	0.022	0.005	0.002
Tight bboxes	0.168	0.078	0.221	0.052	0.005	0.037

Table 4.16: Results from comparing bounding box tightness with Yolov5

Model	mAP0.5	mAP0.5:0.95	Car	Truck	Cyclist	Pedestrian
Baseline	0.864	0.564	0.702	0.688	0.482	0.382
CarFree	0.077	0.030	0.077	0.008	0.012	0.023
Tight bboxes	0.150	0.065	0.151	0.052	0.005	0.052

Table 4.17: Results from comparing bounding box tightness with Faster R-CNN

As table 4.16 and table 4.17 show, there was a substantial improvement when using the tightened bounding boxes. For Yolov5, the **mAP0.5:0.95** nearly tripled, with significant gains for cars, pedestrians and trucks. The cyclist class did not see any improvement. For Faster R-CNN, all classes improved apart from cyclist, which saw reduced results. It should be noted that this improvement was relative, and the scores were still poor when compared to the baseline model.

Chapter 5

Discussion

This chapter discusses the results from the experiments and highlights the most interesting findings. The chapter is divided into four parts; the first three discussing the results in the context of the research questions, with the last part discussing the shortcomings of the thesis.

5.1 Object detection in RGB images

5.1.1 Research Question 1

The first research question was:

> **How well can an object detection model detect real traffic objects when trained on simulated RGB image data collected from the Carla simulator?**

This question was posed to figure out whether or not it was possible to use a simulated environment, more specifically the Carla simulator, to train an object detection model which could detect traffic objects in real images. Looking at experiments 1 through 4, the results were generally poor. Both models performed badly when trained exclusively on simulated data, and the results were not suffi-

cient to be considered usable in real-life scenarios.

Increasing the epochs in experiment 2 was done to see whether it could improve performance while keeping the other parameters the same. In a paper by Dai et al. where several object detection models were tested, larger epochs resulted in improved scores in most cases [30]. For example, for Faster R-CNN their AP score improved from 40.2% to 42.0% when increasing the epochs threefold. The results from experiment 2 indicate that generally, more epochs would improve model performance. For Yolov5, the increase was most noticeable between 300 and 500 epochs. The model only slightly improved between 500 and 700 epochs, suggesting that the model would not stand to gain from training much longer than that. For Faster R-CNN, most classes got worse results when using increased epochs. Only the **pedestrian** class improved between 30 and 60 epochs, while the remaining results were significantly impacted. One reason for this could be **overfitting**, which can happen because the model is unable to generalize because it fits exactly to its training data [10]. Because the model is too well trained to detect a specific variety of data, it is unable to detect similar objects from other datasets. In this case it could mean that the model was trained to only recognize objects in Carla images, and not the similar looking objects in the Kitti images, thus being unable to generalize.

Through experiment 3, the team attempted to improve the results by using a larger dataset. Zhu et al. found that having additional data is helpful, but mainly with correcting regularization and reducing noise and outliers in the dataset[31]. The results show that increasing the size of the dataset did improve the metrics. Faster R-CNN improved significantly, with the mAP scores nearly doubling, allowing the model to outperform Yolov5. However, the results were still not on par with the baseline Kitti model, and would not be considered sufficient for real life use.

Experiment 5 showed that even though the scores were poor, transfer learning on pre-trained COCO models produced increased scores, even doubling the mAP0.5:0.95 at its best. From the conducted experiments, it becomes clear that it is difficult to train an object detection model with Carla images alone and test it on real life images from Kitti. Even though experiment 5 showed improved scores, the results overall were way below the baseline and what would be acceptable for a prop-

erly trained object detection model. The best results across all experiments came from Yolov5, scoring a map0.5:0.95 value of 14.4%, which was extremely low compared to the baseline at more than 70%.

5.1.2 Research Question 2

The team formulated research question 2 as:

> **Can RGB image data collected from the Carla simulator be used to reduce the amount of real images required by fine-tuning an object detection model trained primarily on the simulated data?**

Experiment 5 was conducted to investigate this question. For Yolov5, a significant improvement to the results was observed when training a base model on Carla data and fine-tuning with a small portion of the Kitti dataset. This is consistent with previous work from Dworak et al. and Tremblay et al., which showed that simulator data could be used effectively to fine-tune models [21] [24]. Being able to significantly reduce the size of the manually annotated dataset could save time and money. These results show that the data collected from Carla could have an application in training the building blocks of Yolov5 object detection models used in traffic environments.

As a contrast to these results, the model trained using Faster R-CNN achieved poor performance. As mentioned previously, the experiment was conducted two additional times because the team thought something had gone wrong during dataset preparation or training, but the results were consistent. It is not quite clear to the team where the issue lies. One suggestion is that transfer learning does not work as well for Faster R-CNN as for Yolov5, but experiment 5 (which used the pre-trained COCO weights for transfer learning) got results as expected for both models. The model trained on 2k Kitti images for comparison also achieved very poor performance, despite not using transfer learning. It is possible that Faster R-CNN simply needs much more data in general than Yolov5 to perform decently, but the team could not find any research indicating this.

5.2 Object detection in LiDAR images

5.2.1 Research Question 3

The third research question was:

> **How well can an object detection model detect real traffic objects in LiDAR images when trained on simulated LiDAR image data?**

As mentioned in section 3.2.7, the Ouster dataset was provided as part of a mini-project from the course TDT17. The dataset was hand-annotated by researchers at NAPLab. This slow and tedious process was part of the reason why the project supervisor Frank Lindseth suggested exploring the option to use Carla to generate LiDAR images which could replace parts of the hand-annotated dataset.

The results from experiment 6 show that solely using the Carla LiDAR images to train the models did not produce sufficient results. While the experiment parameters were limited (no increased dataset, using default number of epochs), the results were poor enough that they would likely not increase significantly with these changes.

One reason responsible for the poor results could be the difference in image quality between the datasets. The Carla LiDAR images are severely lacking in detail, with the main distinguishing feature between objects being their shape. The Ouster images (especially ambient) have much higher detail with different objects being easily distinguishable. See figure 5.1 for a comparison. While some details are available in both images, the Ouster images have more details like shadows and object texture.

Figure 5.1: Intensity images from Carla (top) vs Ouster (bottom)

The team believes one reason for this difference is the lack of data available from the Carla sensor compared to the Ouster sensor. The Ouster sensor could produce more information which enables the creation of more detailed images. It is also likely that the Ouster sensor is more precise, which allows for more detailed calculations of pixel values. Also, Ouster might have a more advanced method of projecting the point clouds to 2D images, while doing more processing on the images to enhance clarity . Unfortunately, as mentioned in section 2.5.5, their code is not public domain which makes this difficult to confirm.

5.2.2 Research Question 4

The fourth research question was:

> **To what degree can simulated LiDAR image data be used to reduce the number of real LiDAR images required by fine-tuning a model pre-trained on the simulated data?**

Experiment 7 was conducted to investigate this RQ. While solely using Carla LiDAR images produced poor results, the team wanted to check if the data could be used to reduce the amount of hand-annotated images required. The results from Yolov5 increased across all metrics. Interestingly, the **truck** class saw the largest increase, despite the Carla data not having many objects which could be considered similar to a bus (the most similar object available in Carla would be a fire truck).

When looking at the results from Faster R-CNN, the metrics were similar to the ones achieved when using Kitti in experiment 4. Comparing the results from the 2000 Kitti images with the 2000 images from the Ouster dataset, it can be observed that training with few images and iterations works poorly when using Faster R-CNN through MMDetection. This comparison and the results from experiment 3, where the metrics almost doubled when training with the double amount of training data, also adds to the idea that Faster R-CNN and MMDetection may not work well with few images.

While the team would not consider these results sufficient to use in real scenarios, they show that the Carla LiDAR images could have a use in reducing the amount of real images required while still achieving decent model performance.

5.3 Bounding box tightness

Research question 3 was posed as:

> **To what degree do tight bounding boxes improve the detection of objects?**

A major part of this project was centered around improving the bounding boxes from the CarFree paper [6]. Experiment 7 was conducted to see how the new bounding boxes compared to the old ones. The results showed a significant increase in detection scores across most mAP and individual class AP values. The most notable result was the increase in the highly relevant **mAP0.5:0.95** score, which for Yolov5 nearly tripled, and for Faster R-CNN more than doubled. While the scores were still poor compared to the baseline Kitti dataset, the relative increase shows that the improved bounding boxes did improve the detection scores in this scenario significantly.

A caveat can be seen in the **cyclist** class for both models. Yolov5 did not see any improvement for this class, while Faster R-CNN actually got a decreased cyclist score with the tighter bounding boxes. As mentioned in section 3.2.3, the bounding boxes for the cyclists in Carla were not always consistent. There was also the

issue from section 3.2.7 where cyclists would not get bounding boxes properly drawn sometimes. Furthermore, across all experiments cyclists would consistently get the lowest AP score compared to the other classes. These might be some of the reasons why the cyclist class was an outlier in experiment 7.

5.4 Shortcomings

The team recognize several shortcomings with the thesis. The process of converting 3D bounding boxes from Carla into 2D turned out to be a fairly complicated and lengthy process. The team would like to see the Carla team implement 2D bounding boxes directly to avoid issues. Also, the team did not manage to implement the semantic LiDAR sensor in the synchronized client used to collect the RGB images. The synchronized client has some features like the secondary window and controls which would be helpful for the LiDAR data collection as well.

Another shortcoming of the thesis is the lack of variety in the testing method. The team were surprised to see the poor results on the Carla trained models. While the Kitti images differ somewhat in their light levels and scenery, the general outlines and shapes of the desired objects look fairly similar. The team should have tested the models on several other datasets if possible. These datasets would likely have a different composition to Kitti, which could give a more detailed and balanced conclusion to the research goal.

5.5 Reflections

The team had previous experience working with 2D bounding boxes, which is why 2D object detection was chosen as the focal point for this project. About halfway through the project, the teams' supervisor Frank Lindseth suggested investigating 3D object detection, which would circumvent the need to transform the Carla 3D bounding boxes to 2D. The team spent a couple of days investigating this option. While the team found interesting previous work, it was determined that there was not enough time to completely change the scope of the thesis. The

team acknowledges that more time could have been spent on research early in the semester to avoid these issues.

A significant amount of time was spent improving the bounding boxes from the CarFree project [6]. Fixing the Instance Segmentation sensor by building Carla from scratch took several weeks alone. This led to the team having less time to train and test models than initially planned. If the team had better planned and researched the project initially this could have been avoided, and better results could have been achieved. Earlier testing could have exposed bad datasets or issues with the models (like Faster R-CNN not training properly with the smaller datasets), and the team could have addressed these challenges.

Chapter 6

Conclusion and Future work

6.1 Conclusion

The research goal of this thesis was:

> **To investigate whether automatically annotated images from the Carla simulator can be used to train object detection models for real traffic scenarios.**

The results from the conducted experiments seem to indicate that Carla data by itself is not sufficient to train models which can perform well on real data. Models trained on both RGB and Lidar images from Carla performed poorly when tested on real datasets, and changing parameters like dataset size and number of epochs trained did not significantly improve the results. Similarly to Dworak et al. and Tremblay et al., using the Carla data to train a base model which was then fine-tuned using a small amount of real data showed a decent improvement, suggesting that the data is most suited for this purpose [21] [24].

A significant portion of this project was spent improving the bounding boxes from the CarFree project [6]. The improved and tightened bounding boxes achieved significantly better results in the conducted experiment. This seems to show that bounding box tightness has a large impact on the performance of an object detec-

tion model, especially for higher IoU thresholds.

6.2 Future Work

The team believes that certain aspects of the data collection process could be further improved. One aspect is inspecting the differences between small and big objects, and whether having more or less size variety would improve the results. Also, collecting images at various resolutions could be worth investigating. Training with smaller images is generally quicker, so one could potentially gather a large amount of smaller resolution images to get better results while training the same amount of time. Conversely, larger images could have more detail which could also improve detection.

Testing the trained models using additional real datasets could be beneficial. By doing this one could eliminate the possibility that testing solely with Kitti and Ouster data was partly responsible for the bad results from the different experiments. This idea could also be expanded to training, where Carla data could be combined with other simulators, like rFpro[1]. This could create a greater variance in the dataset and produce an even more robust dataset.

Improving the bounding boxes further is another possibility. As seen in section 5.3, the improved bounding boxes produced better results. Improving these bounding boxes further could be done by for example fixing the issue with the cyclist bounding boxes. The issue with occluded bounding boxes in the y dimension should also be addressed. Most importantly, adding in the possibility of extracting 2D bounding boxes directly from Carla instead of having to project the 3D bounding boxes into the 2D space could be a major improvement. This is something that the developers of Carla would have to look into and implement directly into the simulator with a feature that would enable the extraction of the 2D bounding boxes directly. It would make it easier to create bounding boxes for occluded objects, and avoid bugs such as bounding boxes extending outside of the objects themselves in the y-axis, as presented in section 3.2.4.

[1] `https://www.rfpro.com/`

As mentioned in section 5.5, the team spent some time researching 3D object detection. Because the Carla bounding boxes are available directly in a 3D format, this could be a better approach to extract data from the Carla simulator. Both RGB image 3D object detection and LiDAR point cloud object detection could be possible applications for this data.

Finally, the team believes improving the Carla LiDAR images could increase model performance substantially. Particularly, being able to recreate the ambient images that Ouster provide would be helpful as these are most similar to real images and contain the most detail. In order to do this, the Carla team would likely need to add more functionality to their LiDAR sensors like the possibility to extract reflectivity data.

All code is available at `https://github.com/PederEspen/master-thesis`.

Bibliography

[1] World Health Organization, *Global status report on road safety 2018*, 2018. [Online]. Available: https://www.who.int/publications/i/item/WHO-NMH-NVI-18.20.

[2] US Department of Transportation, *2016 fatal motor vehicle crashes: Overview*, 2017. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812456.

[3] H. Zhong, W. Li, M. W. Burris, A. Talebpour and K. C. Sinha, 'Will autonomous vehicles change auto commuters' value of travel time?' *Transportation Research Part D: Transport and Environment*, vol. 83, p. 102 303, 2020, ISSN: 1361-9209. DOI: https://doi.org/10.1016/j.trd.2020.102303. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361920919311010.

[4] R. E. Stern, Y. Chen, M. Churchill, F. Wu, M. L. Delle Monache, B. Piccoli, B. Seibold, J. Sprinkle and D. B. Work, 'Quantifying air quality benefits resulting from few autonomous vehicles stabilizing traffic,' *Transportation Research Part D: Transport and Environment*, vol. 67, pp. 351–365, 2019, ISSN: 1361-9209. DOI: https://doi.org/10.1016/j.trd.2018.12.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361920918304383.

[5] European Environment Agency, *Co2 emissions from cars: Facts and figures (infographics)*, 2019. [Online]. Available: https://www.europarl.europa.eu/news/en/headlines/society/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics.

[6] J. Jang, H. Lee and J.-C. Kim, 'Carfree: Hassle-free object detection dataset generation using carla autonomous driving simulator,' *Applied Sciences*, vol. 12, no. 1, 2022, ISSN: 2076-3417. [Online]. Available: `https://www.mdpi.com/2076-3417/12/1/281`.

[7] H. Kervadec, J. Dolz, S. Wang, E. Granger and I. B. Ayed, *Bounding boxes for weakly supervised segmentation: Global constraints get close to full supervision*, 2020. DOI: `10.48550/ARXIV.2004.06816`. [Online]. Available: `https://arxiv.org/abs/2004.06816`.

[8] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks,' *Advances in neural information processing systems*, vol. 25, 2012.

[9] K. O'Shea and R. Nash, 'An introduction to convolutional neural networks,' *arXiv preprint arXiv:1511.08458*, 2015.

[10] IBM.com, *What is overfitting?* [Online]. Available: `https://www.ibm.com/cloud/learn/overfitting`.

[11] K. Weiss, T. Khoshgoftaar and D. Wang, 'A survey of transfer learning,' *Journal of Big Data*, vol. 3, May 2016. DOI: `10.1186/s40537-016-0043-6`.

[12] IBM.com, *What is computer vision?* [Online]. Available: `https://www.ibm.com/topics/computer-vision`.

[13] Z. Zou, Z. Shi, Y. Guo and J. Ye, *Object detection in 20 years: A survey*, 2019. DOI: `10.48550/ARXIV.1905.05055`. [Online]. Available: `https://arxiv.org/abs/1905.05055`.

[14] M. Rajchl, M. C. H. Lee, O. Oktay, K. Kamnitsas, J. Passerat-Palmbach, W. Bai, M. Damodaram, M. A. Rutherford, J. V. Hajnal, B. Kainz and D. Rueckert, *Deepcut: Object segmentation from bounding box annotations using convolutional neural networks*, 2016. DOI: `10.48550/ARXIV.1605.07866`. [Online]. Available: `https://arxiv.org/abs/1605.07866`.

[15] A. M. Hafiz and G. M. Bhat, 'A survey on instance segmentation: State of the art,' *International Journal of Multimedia Information Retrieval*, vol. 9, no. 3, pp. 171–189, Jul. 2020. DOI: `10.1007/s13735-020-00195-x`. [Online]. Available: `https://doi.org/10.1007%2Fs13735-020-00195-x`.

[16] P. Henderson and V. Ferrari, *End-to-end training of object class detectors for mean average precision*, 2016. DOI: 10.48550/ARXIV.1607.03476. [Online]. Available: https://arxiv.org/abs/1607.03476.

[17] Roboflow.com, *What is mean average precision (map) in object detection?* [Online]. Available: https://blog.roboflow.com/mean-average-precision/.

[18] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, 'You only look once: Unified, real-time object detection,' Jun. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[19] S. Ren, K. He, R. Girshick and J. Sun, 'Faster r-cnn: Towards real-time object detection with region proposal networks,' *Advances in neural information processing systems*, vol. 28, 2015.

[20] velodyne.com, *What is lidar?* [Online]. Available: https://velodynelidar.com/what-is-lidar/.

[21] D. Dworak, F. Ciepiela, J. Derbisz, I. Izzat, M. Komorkiewicz and M. Wójcik, 'Performance of lidar object detection deep learning architectures based on artificially generated point cloud data from carla simulator,' in *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)*, 2019, pp. 600–605. DOI: 10.1109/MMAR.2019.8864642.

[22] D. Niranjan, B. C. VinayKarthik and Mohana, 'Deep learning based object detection model for autonomous driving research using carla simulator,' in *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, 2021, pp. 1251–1258. DOI: 10.1109/ICOSEC51865.2021.9591747.

[23] T. Bu, X. Zhang, C. Mertz and J. M. Dolan, 'Carla simulated data for rare road object detection,' in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 2794–2801. DOI: 10.1109/ITSC48978.2021.9564932.

[24] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon and S. Birchfield, 'Training deep networks with synthetic data: Bridging the reality gap by domain randomization,' in *2018*

IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2018, pp. 1082–10 828. DOI: 10.1109/CVPRW.2018.00143.

[25] H. A. Alhaija, S. K. Mustikovela, L. Mescheder, A. Geiger and C. Rother, *Augmented reality meets computer vision : Efficient data generation for urban driving scenes*, 2017. DOI: 10.48550/ARXIV.1708.01566. [Online]. Available: https://arxiv.org/abs/1708.01566.

[26] A. Tsirikoglou, J. Kronander, M. Wrenninge and J. Unger, *Procedural modeling and physically based rendering for synthetic data generation in automotive applications*, 2017. DOI: 10.48550/ARXIV.1710.06270. [Online]. Available: https://arxiv.org/abs/1710.06270.

[27] S. R. Richter, V. Vineet, S. Roth and V. Koltun, *Playing for data: Ground truth from computer games*, 2016. DOI: 10.48550/ARXIV.1608.02192. [Online]. Available: https://arxiv.org/abs/1608.02192.

[28] T. Wu, H. Fu, B. Liu, H. Xue, R. Ren and Z. Tu, 'Detailed analysis on generating the range image for lidar point cloud processing,' *Electronics*, vol. 10, no. 11, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10111224. [Online]. Available: https://www.mdpi.com/2079-9292/10/11/1224.

[29] K. Oksuz, B. C. Cam, S. Kalkan and E. Akbas, *Imbalance problems in object detection: A review*, 2019. DOI: 10.48550/ARXIV.1909.00169. [Online]. Available: https://arxiv.org/abs/1909.00169.

[30] Z. Dai, B. Cai, Y. Lin and J. Chen, 'Up-detr: Unsupervised pre-training for object detection with transformers,' in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2021, pp. 1601–1610.

[31] X. Zhu, C. Vondrick, D. Ramanan and C. Fowlkes, 'Do we need more training data or better models for object detection?' *BMVC 2012 - Electronic Proceedings of the British Machine Vision Conference 2012*, vol. 3, Jan. 2012. DOI: 10.5244/C.26.80.

Appendix A

Faster R-CNN

The project datasets only had 4 classes, which required overwriting the default number of classes (which was taken from COCO). This was done by editing the file **mmdetection/configs/scratch/faster_rcnn_r50_fpn_gn-all_scratch_6x_coco.py**, by adding the line *num_classes=4*.

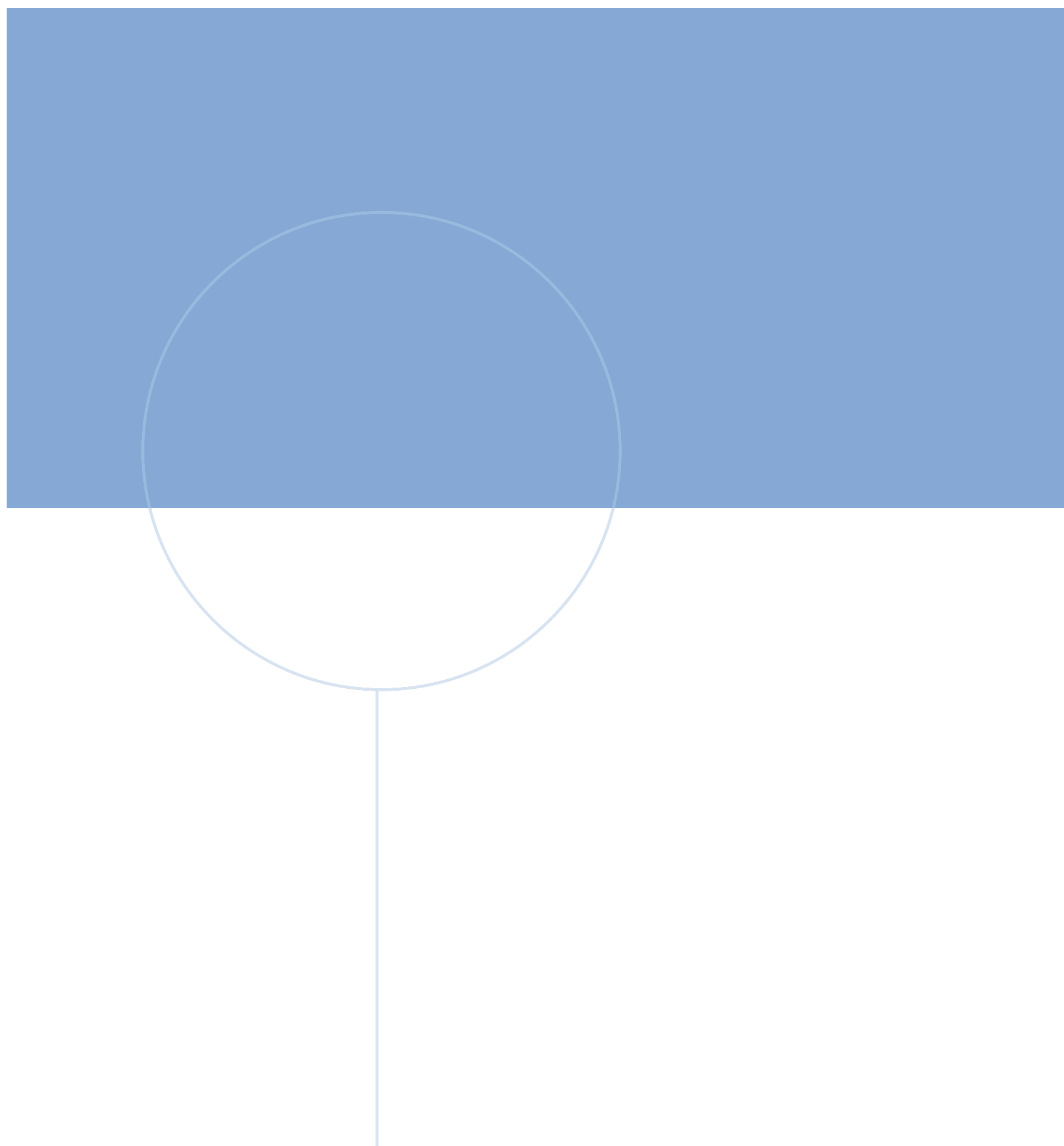
The next step was to add the correct class names in the **MMDetection/mmdet/-core/evaluation/class_names.py** file and also adding the classes and some color palettes in the **MMDetection/mmdet/datasets/coco.py** file. The color palettes were required and used to draw different class bounding boxes during inference.

After changing all of the class names in the files mentioned above, the class names in the annotation files under **MMDetection/data/coco/annotations** had to be changed (see the section about the dataset format below for more information). The class order in the annotation files had to be the same as in the previous files, which is shown in figure A.1.

```
],
"categories": [
    {
        "id": 0,
        "name": "traffic",
    },
    {
        "id": 1,
        "name": "car",
    },
    {
        "id": 2,
        "name": "truck",
    },
    {
        "id": 3,
        "name": "cyclist",
    },
    {
        "id": 4,
        "name": "pedestrian",
    }
```

Figure A.1: Class structure inside of a .json annotation file used in MMDetection Faster R-CNN.

The last step required was in **MMDetection/configs/_base_/datasets/coco_detection.py**, specifying where the different train, test and validation folders were located.



NTNU

Kunnskap for en bedre verden