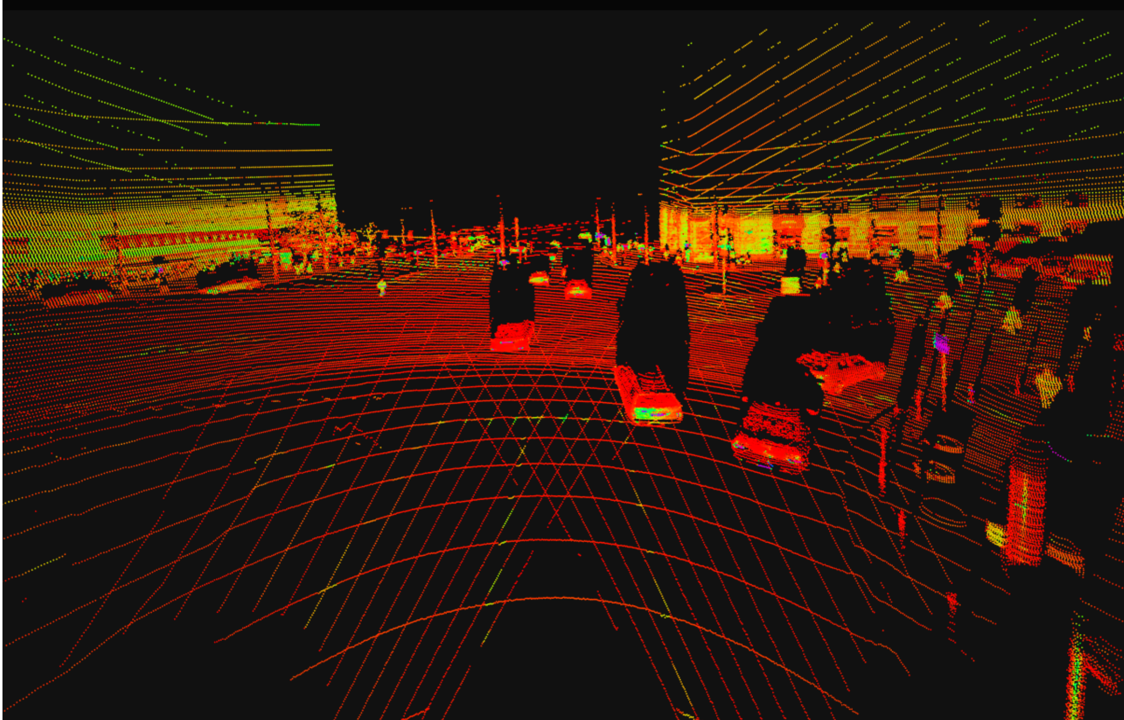
CHALMERS
UNIVERSITY OF TECHNOLOGY



TimePillars: Temporally-recurrent 3D LiDAR Object Detection

Deep Learning LiDAR perception for autonomous driving

Master's thesis in Systems, Control and Mechatronics

Bernardo Taveira & Ernesto Lozano Calvo

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

TimePillars: Temporally-recurrent 3D LiDAR Object Detection

Deep Learning LiDAR perception for autonomous driving

BERNARDO TAVEIRA & ERNESTO LOZANO CALVO

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2023

TimePillars: Temporally-recurrent 3D LiDAR Object Detection
Deep Learning LiDAR perception for autonomous driving

BERNARDO TAVEIRA ERNESTO LOZANO CALVO

© BERNARDO TAVEIRA, 2023. © ERNESTO LOZANO CALVO, 2023.

Supervisors: Niklas Gustafsson and Jonathan Larsson, Zenseact AB and Fredrik Kahl, Department of Electrical Engineering
Examiner: Fredrik Kahl, Department of Electrical Engineering

Master's Thesis 2023
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: LiDAR point cloud corresponding to a driving scene, obtained from Zenseact Open Dataset.

Typeset in $\LaTeX$
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

TimePillars: Temporally-recurrent 3D LiDAR Object Detection

3D LiDAR object detection for Autonomous Driving

BERNARDO TAVEIRA & ERNESTO LOZANO CALVO

Department of Electrical Engineering
Chalmers University of Technology

Abstract

Object detection applied to LiDAR point clouds is a relevant task in robotics, and particularly in autonomous driving. Single frame methods, predominant in the field, exploit information from individual sensor scans. Recent approaches achieve good performance, at relatively low inference time. Nevertheless, given the inherent high sparsity of LiDAR data, these methods struggle in long-range detection (e.g. 200m) and lack of temporal continuity, ignoring past information. We deem these characteristics to be critical in achieving safe automation.

Aggregating past data frames not only leads to a denser point cloud representation, but it also brings time-awareness to the system, and provides information about how the environment is changing. Solutions of this kind, however, are often highly problem-specific, demand careful data processing, and tend not to fulfil efficiency requirements.

In this context we propose TimePillars, a temporally-recurrent object detection pipeline which leverages the pillar representation of LiDAR data across time, respecting hardware integration efficiency constraints, and exploiting the diversity and long-range information of the novel Zenseact Open Dataset (ZOD). By performing extensive experimentation, we prove the benefits of having a recurrent scheme, and show how basic building blocks are enough to achieve robust and efficient results.

Keywords: Autonomous Driving, Deep Learning, LiDAR, point cloud, Recurrent Neural Network, convGRU.

Acknowledgements

The completion of this thesis sets an end to our higher education, and more specifically to our Masters' in Systems, Control and Mechatronics at Chalmers University of Technology. We feel thankful to have been given the chance to boost our knowledge further, and materialise it in this work.
We would like to directly thank our supervisors at Zenseact, Niklas Gustafsson and Jonathan Larsson, as well as our examiner at Chalmers, Fredrik Kahl, for their continuous support and always helpful guidance throughout the process.
We would also like to thank Zenseact for providing us with the right resources and knowledge necessary for this project. It is with warm heart that we part ways with all the amazing people at Zenseact who took us in and made us feel like any other member of their team. More specifically, we would like to thank the Deep Learning LiDAR Perception team (DLLP) with whom we shared workspace most days, guiding us and pushing us to go further.
Finally, we remember our families, who gave us all their day-to-day support, present as always. Thank you.

Bernardo Taveira and Ernesto Lozano Calvo, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

AD	Autonomous Driving
BEV	Bird's Eye View
CNN	Convolutional Neural Network
ConvGRU	Convolutional Gate Recurrent Network
GPU	Graphics Processing Unit
GRU	Gate Recurrent Unit
INS	Inertial Navigation Systems
LiDAR	Light Detection And Ranging
NN	Neural Network
RNN	Recurrent Neural Network
ZOD	Zenseact Open Dataset

Contents

List of Acronyms **ix**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Context 1
1.1.1 Autonomous driving 1
1.1.2 Perception 2
1.1.3 Challenges 2
1.2 Aim 3
1.3 Contributions and limitations 3
1.4 Content structure 4

2 Theoretical background **5**
2.1 Object detection 5
2.2 Deep learning 5
2.2.1 Supervised Learning 6
2.2.2 Fundamental NN building blocks 6
2.2.2.1 From neurons to fully-connected layers 6
2.2.2.2 Batches and their normalisation 7
2.2.3 Convolutional Neural Networks 7
2.2.4 Recurrent Neural Networks 8
2.2.4.1 GRU 8
2.3 Literature review 10
2.3.1 3D LiDAR object detection overview 10
2.3.1.1 Standard point cloud representation schemes 10
2.3.1.1.1 Projection methods 10
2.3.1.1.2 Volumetric methods 10
2.3.1.1.3 Point-based methods 11
2.3.1.2 The State Of The Art 11
2.3.2 Relevant work 12
2.4 Proposed solution 13

3 Data **15**
3.1 Sensing 15

3.1.1 LiDAR 15
3.2 Dataset 16
3.2.1 Sensor Setup 16
3.2.2 Frames 17
3.2.3 Sequences 19

4 Methods 21
4.1 Single frame 21
4.1.1 PointPillars 21
4.1.2 Proposed baseline 22
4.1.2.1 Input Preprocessing 23
4.1.2.2 Labels 24
4.1.2.3 NN Architecture 24
4.1.2.3.1 Pillar Feature Encoder 24
4.1.2.3.2 Backbone 25
4.1.2.3.3 Detection Head 25
4.1.2.4 Postprocessing 26
4.2 Multi-frame 26
4.2.1 LiDAR point clouds aggregation 26
4.2.1.1 Raw data EGO motion compensation 26
4.3 Temporally-aware 28
4.3.1 Time recurrency 28
4.3.1.1 convGRU 29
4.3.2 TimePillars 29
4.3.2.1 Model architecture 29
4.3.2.1.1 Feature EGO motion compensation 30
4.3.2.1.1.1 Interpolation-based 31
4.3.2.1.1.2 Convolutional-based 31

5 Experimental evaluation 33
5.1 Data Handling 33
5.2 Training 33
5.2.1 Class labels 33
5.2.2 Loss functions 34
5.2.2.1 Classification: Focal Loss 34
5.2.2.2 Regression: Huber Loss 34
5.2.2.3 Masking 35
5.2.3 Configuration 35
5.2.3.1 Hyperparameters 35
5.2.3.2 Transfer learning 36
5.2.3.3 Class imbalance weighting 36
5.2.4 TimePillars training step 37
5.3 Evaluation metrics 39
5.3.1 Precision 39
5.3.2 Recall 39
5.3.3 F1-Score 39
5.3.4 Average Precision 40

5.3.4.1 Mean Average Precision (mAP) 40
5.4 Results 40
5.5 Ablation studies 42
5.5.1 Longitudinal detection range 42
5.5.2 Long-term memory 43

6 Conclusions **45**
6.1 Discussion 45
6.2 Future work 45

Bibliography **47**

List of Figures

1.1 Sense-Plan-Act block diagram 1

3.1 Illustration of sensor position on acquisition vehicle [3] 16
3.2 Illustration of data in single frame 17
3.3 Geographical distribution of the frames subset [3] 18
3.4 Distribution of frames across time of day, road types and weather conditions [3] 18
3.5 Distance distribution of annotations compared to other popular datasets [3] 19

4.1 PointPillars block diagram [10] 22
4.2 Single Frame baseline block diagram 23
4.3 Ego motion compensation comparison for t (red) and $t-10$ (blue) . . 28
4.4 10 consecutive frames aggregation with EGO motion compensation . 28
4.5 TimePillars model architecture 30
4.6 Illustration of convolution-based transformation module and auxiliary task 32

5.1 Illustration of single training step. The cycle is not batched for simplicity. 38

List of Tables

5.1 Model-fixed hyperparameters. 36
5.2 Results of multiple models trained on ZOD frames dataset. All models were evaluated with the same number of scans as they were trained with. (*) Muti-Frame baseline built based on PointPillars and described in the Methods chapter. (†) Cyclist class includes all vulnerable vehicle types including motorcycles, wheelchairs and any human-powered, electric or motorised wheeled object that is used to transport people such as scooters. (‡) By "none", we mean that the network does not include any motion transformation layers but instead the data is motion compensated before being fed into the network. 41
5.3 Single frame baseline compared to two possible recurrent networks differing in the location of the recurrent module. (†) Like all other TimePillars models used within this project employ a convGRU after the backbone, unless otherwise specified. 42
5.4 Results from using or not the auxiliary task for the convolution based transform module. 42
5.5 Longitudinal detection range ablation study. We compare the system's performance for growing distances from the EGO vehicle. . . . 43
5.6 Average precision evaluated for different number of LiDAR scans. These models were trained without any motion compensation in the model itself, meaning that the data fed into the network both in inference and training was already motion compensated. 43

1
Introduction

1.1 Context

Driving-related incidents account for a significant proportion of casualties in contemporary society. Statistics show that fatalities can reach approximately 1.4 million annually and 50 million injuries [1]. Moreover, 94% of these incidents are attributed to human error [26]. Therefore, there exists a pressing need to enhance driving techniques while prioritising safety as the foremost consideration.

With the completion of this project, done in collaboration with Zenseact AB, we aim to contribute to the field, towards a safer and more sustainable future. "Towards zero. Faster." [1]

1.1.1 Autonomous driving

In search of a scenario with zero road accidents, safe automation comes as a way to decouple driving from its human-centred task nature. Autonomous driving, also known as self-driving or driverless driving, presents a machine-based solution that falls under the subfield of robotics to address this issue. In addition to enhancing safety, the adoption of autonomous driving can indirectly improve core concerns, such as mobility accessibility, traffic density, efficiency, and emissions [26].
Recent advancements in vehicle dynamics, computer vision, and machine learning, coupled with the parallel development of high-computing embedded systems, have led to significant progress in the area. The autonomous system aims to replace the human driver with a set of functional building blocks capable of translating information "from pixel to torque" [1], following a "Sense-Plan-Act" scheme (Fig. 1.1). In an iterative manner, the system shows capability of perceiving successive information from the environment, making decisions based on it, and commanding suitable vehicle's actuation.

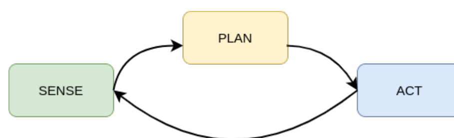


Figure 1.1: Sense-Plan-Act block diagram

As every emerging technology, self-driving comes along with a new ethical context

derived. Aside from technological advancements, social, political and economical issues influence its application [8]. The low tolerance to failures caused by autonomous vehicles may delay their introduction in the market, legislation playing a key role in the equation. Traditional assumptions about responsibilities in traffic issues are being subject to change in recent years. A responsibility shift is taking place, from drivers and other traffic participants, to managers, vehicle manufacturers, and others who contribute professionally to the traffic system platform [11]. This puts increase requirements on the development of robust perception system.

1.1.2 Perception

Perception of the environment is at the heart of the autonomous driving challenge, representing the first block of the three mentioned above. To be able to interact with what is found around the vehicle, knowledge about the surrounding environment is first required.
From all the worthy enclosing elements, the task of object detection focuses the attention on physical bodies. In particular, it aims at locating (where) and classifying (what) surrounding objects. Given its relevance in the AD pipeline, it is the main task of study in this work.
Acting as "the eyes" of the vehicle, sensors are the input source of information to the system, and therefore they play a fundamental role in reliability and safety. Acquired data can be of diverse nature, providing alternative descriptions about the environment. Given the spatial dependence the driving task owns, 3D-awareness is crucial. From the existing sensors, one which has grown in popularity due to this requirement is the LiDAR. By building a point cloud of the surrounding objects through the measurement of a light beam emitted, the LiDAR is unmatched in terms of accuracy and resolution compared to other methods of 3D reconstruction. This sensor, despite the scepticism it brought in its early years due to the high cost, lack of weather resistance, and fragile architecture, has come a long way. More recently, new innovations such as solid-state architectures and clever solutions with lightweight mirrors made the future of LiDAR possible on consumer vehicles.

1.1.3 Challenges

As discussed, AD sets strict specifications, which constraint the validity of engineering solutions. Accuracy, speed and anticipation are some of the determinant features in the particular case of object detection. Firstly, the system is required to identify and locate objects correctly which is a vital benchmark as incorrect or missing detections could hypothetically lead to unsafe driving scenarios. Secondly, the algorithm execution needs to be fast (real-time performance). The trivial reason behind it is that the ego-vehicle, as well as the quickly-changing environment, demand fast decision-making. This constrains the development, imposing a compatibility need with integrated hardware accelerators. Finally, the degree of detectability of the system is a key feature conditioning the vehicle's reaction and anticipation capabilities. Long-range perception is crucial proven detecting objects as far as possible from the current position can make a significant difference at high speeds.

Sensor-wise, with the start of mass adoption of LiDAR, the demand for efficient perception solutions based on it is rising. Deep learning algorithms applied to 3D point clouds present multiple challenges. Unlike images, LiDAR data are highly sparse, meaning that the diversity of point density varies noticeably with distance. Furthermore, point clouds are spatially unordered and unstructured, therefore, properties like permutation and orientation invariance need to be taken care of when designing NN architectures, difficulting their processing.

1.2 Aim

In this context, and being aware of the existing challenges both at task and data levels, the main objective of this project is to design a novel 3D LiDAR object detection pipeline that obtains a more confident output than the work baseline, while fulfilling hardware integration constraints, i.e. in an efficient way. To satisfy such requirements, only simple and fundamental Machine Learning blocks are allowed to be used. The solution then passes by fully understanding and respecting the nature of the data involved.
To address the sparsity (varying density of points at greater distances) it is possible to consider a (stack) point clouds from different timestamps. This procedure, commonly termed as point clouds aggregation, leads to a denser data representation, which adds extra information of the environment. In addition, doing this can also contribute in mitigating problems of occlusion by closer objects coming into frame. The downside of such an approach, however, is the consequent increase in difficulty of maintaining an efficient execution for real-time onboard applications. Keeping the intuition behind this approach, we believe modelling and making use of the inherent time dependencies of the parameters involved may solve the problem.

1.3 Contributions and limitations

In this paper, we therefore propose a recurrent network that improves accuracy in object detection, especially at long distances. Our suggested architecture builds upon relevant past literature, and gets the most out of a novel LiDAR dataset: Zenseact Open Dataset.
Large emphasis is set on the extended annotation range, showcasing how the proposed network significantly enhances the detection of objects in distant areas, where point density is low and classification is challenging without multiple aggregated scans.
Furthermore, extensive experimentation was conducted, together with ablation studies on the network's continuous inference stability, and various solutions and techniques to enhance detection for extended sequences were presented.
In order to achieve practicality on modern embedded hardware, our project was designed to operate within the constraints of current-generation vehicle hardware. As a result, our network prioritised compatibility with a broad range of devices by relying solely on common operations supported by popular optimisers and tools such as TensorRT. We avoided additional operations that might not be supported

by deep learning accelerators.
As a reference deployment architecture, we take the latest generation of NVIDIA Orin, where we want to optimise the operations with the Orin DLA in mind.
Despite these limitations, using mostly 2D convolutional operations we were able to improve the performance of a established reference in the field, PointPillars [10], without the need of additional complex operations such as 3D convolutions, sparse convolutions, or self-attention blocks.

1.4 Content structure

To present the work performed, the following content structure is taken:

- A literature background is provided, introducing the reader to the existing relevant work in the field. At that point of familiarisation, the proposed solution is then detailed and contrasted with the latter.
- A data chapter follows. Its aim is to provide an overview in perception sensors, with focus on the utilised LiDAR; and to further describe the dataset used, of high novelty in the field.
- The next section holds the core contribution. The methods implemented are structured based on the way they deal with sensor data frames, as detailed later.
- Evaluation of the method follows. The adopted training configurations are provided, a closer look is taken to the classical choices of evaluation metrics, and then extensive results and ablation studies analyse the validity of the solutions.
- Finally, some insights are addressed as a conclusion, and future work paths are pointed out.

2
Theoretical background

This chapter is used to provide a theoretical context in the directly-related topics of this work. A brief introduction to Deep learning and common building blocks is provided. It follows a detailed relevant literature review. Only then, our proposed method is introduced and contrasted with previous work in the research area.

2.1 Object detection

Object detection is a computer vision technique that involves identifying and locating objects of interest from a multitude of possible sensors such as camera, radar and LiDAR.
The goal of object detection is to not only classify the objects present in an image or video but also to provide their precise location and spatial extent, generally represented with a bounding box. Object detection algorithms typically follow a two-step process:

- Localization: This step involves determining the position of objects in the image or video by defining bounding boxes around them. The bounding boxes are usually represented by their coordinates, such as the top-left and bottom-right corner coordinates, and in the case of 3D detection, the orientation angle.
- Classification: After localizing the objects, the algorithm assigns class labels to each bounding box to identify the type or category of the object.

In this work, 3D object detection is performed, having LiDAR point cloud data as input. Furthermore, we focus on dynamic objects, typically found in driving scenarios (vehicles, cyclists, pedestrians and animals). Deep learning is used as the tool to turn the algorithms into reality, proven its strengths dealing with highly complex data.

2.2 Deep learning

As a subfield of machine learning, and the latter, respectively, of artificial intelligence, deep learning aims to enable computers to learn and make intelligent decisions by processing vast amounts of data.
Deep learning algorithms are designed to learn hierarchical representations of data through a set of layers of interconnected nodes, often known as neurons. These layers form what is called a deep neural network. Neurons can be seen as systems, with input and output signals. Output signals correspond to a mathematical function of the input, which is passed to the neurons in the next layer. The connections between

the neurons have weights and biases which act as learning parameters associated to them. These are adjusted during a training phase to optimise the network's performance.

2.2.1 Supervised Learning

From the existing learning typologies, we focus on supervised learning. The training process in supervised deep learning involves presenting the network with a large labelled dataset. This implies that problem solutions, represented as ground-truth data, are available as reference for the network to learn. The learning takes place at the training phase, and it allows the network to identify patterns and relationships in the data by adjusting the weights of the connections. The network learns to recognise features at the early networks layers, and combines them to form more complex representations at deeper layers. This hierarchical feature extraction enables deep learning models to automatically learn and extract relevant features from raw data of diverse nature.
Deep learning under supervision has demonstrated exceptional performance in various domains, including computer vision, natural language processing, speech recognition, and recommendation systems. It has achieved state-of-the-art results in tasks such as image classification, object detection, machine translation, and voice synthesis.
One of the reasons deep learning has gained significant attention is the availability of large datasets and powerful computing resources, which allow for training deep neural networks with millions or even billions of parameters. Additionally, advancements in hardware, such as graphics processing units (GPUs), have accelerated the training and inference processes.

2.2.2 Fundamental NN building blocks

As a background to this project, the most basic NN blocks are introduced. These are commonly used in the practical majority of networks, and we believe they act as fundamental knowledge in the field.

2.2.2.1 From neurons to fully-connected layers

The most basic unit of a neural network is often denoted as perceptron, or neuron. Its expression, shown below (see Eq. 2.1), controls linearly the input x through learnable parameters (W, b), and expresses the output as a non-linear mapping of the latter using an activation function σ.

$$\mathbf{y} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}) \tag{2.1}$$

The choice of activation function defines the non-linear characterisation of the output. Common activations employed are the Sigmoid function (Eq. 2.2), the Hyperbolic tangent (Eq. 2.3), and the Rectified Linear Unit (Eq. 2.4):

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

$$f(x) = \max(0, x) \tag{2.4}$$

Neurons are fundamental in the creation of more complex and deeper architectures. The clearest example of this are fully-connected layers. Formed by the replication of multiple neurons, they take every neuron in the current layer, and connect it to every neuron in the next layer.

2.2.2.2 Batches and their normalisation

In deep learning, when training a neural network, the dataset is typically divided into smaller data subsets, or batches. Batches are processed together during the training phase. Instead of updating the model's parameters after each individual sample, or after processing the dataset as a whole, the parameters are updated based on the average of the gradients computed over the batch. The benefits of batches extend to efficiency, convergence and generalisation, among others.
Batch normalisation is a technique commonly used to improve the training and performance of neural networks. It addresses the challenge of covariate shift, which has to do with the change in the distribution of the network activations as the parameters are updated during training, and keeping the labels fixed.
The main idea behind batch normalisation is to normalise the inputs to each layer of the network by subtracting the batch mean and dividing by the batch standard deviation. This operation is applied to the inputs of a layer within a mini-batch of training examples, and has the effect of uniforming the representation, and compacting data in a way that prevents large spread, which turns beneficial in the learning process.

2.2.3 Convolutional Neural Networks

Fully-connected layers require flattened data as input to allow their utilisation. When applying a flattening transformation to tensor data, spatial-awareness is lost. In applications like computer vision, where spatial data are key, the need of layers which respect it is high.
Convolutional Neural Networks were precisely born to exploit the spatial locality of data, while doing so in an efficient way. By applying convolutional operations to the input, data are scanned across with a set of filters or kernels. Each filter detects local patterns and extracts relevant features. Parameter sharing holds, meaning the same filter is applied across the entire input to detect the same feature in different regions. This sharing of weights reduces the number of learnable parameters and enables the network to learn spatial hierarchies of features, while remaining efficient.
The output size of a convolutional layer, with respect to the input, is given in Eq. 2.5, and reasons how the different layer-specific parameters need to be configured to achieve the wanted output behaviour.

$$\text{Out_size} = \frac{\text{In_size} - \text{Filter_size} + 2 \cdot \text{Padding}}{\text{Stride}} + 1 \tag{2.5}$$

$$\begin{aligned}
\text{Out_size} &: \text{Size of the output feature map.} \\
\text{In_size} &: \text{Size of the input image or feature map.} \\
\text{Filter_size} &: \text{Size of the convolutional filter (receptive field).} \\
\text{Padding} &: \text{Amount of zero-padding applied to the input.} \\
\text{Stride} &: \text{Step size of the filter as it convolves across the input.}
\end{aligned}$$

CNNs often include pooling layers to downsample the feature maps and reduce their spatial dimensions. Pooling operations, such as max pooling or average pooling, summarise the information within a local neighborhood and retain the most salient features.
Given the mentioned relevance and adequacy of CNNs to computer vision, their proven efficiency, and the flexibility they provide (fully-connected layers can be expressed as 1×1 convolutions), CNNs are the main typology utilised in this work.

2.2.4 Recurrent Neural Networks

RNNs are a class of artificial NN designed to process sequential data. Unlike traditional feedforward networks, RNNs have a memory component that allows them to capture and utilize information from previous inputs in the sequence. This memory enables RNNs to model and understand the temporal dependencies present in the data, making them particularly effective for tasks involving sequences, therefore ideal for a driving scenario.
The fundamental idea behind RNNs is the concept of shared weights and hidden states. In a standard feedforward network, each input is processed independently, and there is no notion of time or order. However, when working with sequential data, the order of the elements matters, and the relationship between elements is crucial for understanding the data.
To overcome these limitations, RNNs introduce the notion of recurrent connections. These connections allow the hidden state (memory) from the previous time step to be fed again into the network at the current time step, creating a loop-like structure. This recurrent connection enables the network to maintain a form of memory and consider past information when processing the current input.
At each time step, an RNN takes an input vector and combines it with the previous hidden state to produce an output and an updated hidden state. The output can be used for prediction or fed as an input to the next time step. This recurrent process allows the network to capture dependencies over time and model complex sequences.

2.2.4.1 GRU

A Gated Recurrent Unit is a specific type of RNN that addresses some of the limitations of traditional RNNs, such as difficulties in capturing long-term dependencies and vanishing/exploding gradients.
In a standard RNN, the hidden state is updated at each time step by directly incorporating the current input and the previous hidden state. However, doing the

latter has the implicit risk of suffering from the problem of vanishing/exploding gradients, i.e. the gradients getting very close to zero, or to infiniy, respectively, and therefore damaging the NN's ability to learn long-term dependencies in sequential data. This issue led to the development of more sophisticated RNN variants like GRU, or Short-Term Memory (LSTM), not considered in this work because of being fairly similar, but slightly more computationally demanding.
The GRU introduces the concept of gating mechanisms, which allows it to selectively update and utilize information from previous time steps. It achieves this through two main gates: the reset gate (r) and the update gate (z), which condition the new hidden state to be computed.
The reset gate determines how much of the previous hidden state should be forgotten. It is computed based on the concatenation of the previous hidden state and the current input. The reset gate helps the GRU adaptively decide which past information to discard, enabling it to focus on more recent and relevant information, see Equation 2.6.
The GRU also calculates a candidate hidden state $\tilde{h}(t)$. This candidate state represents the new information that can be added to the previous hidden state. It is computed based on the concatenation of the reset-gated previous hidden state and the current input. The candidate hidden state captures the potential new information that the GRU considers relevant, as Equation 2.7 shows.
The update gate (Equation 2.8) controls the balance between the new candidate hidden state and the previous hidden state. It decides how much of the previous hidden state should be retained and how much of the new information should be incorporated. The update gate is also computed based on the concatenation of the previous hidden state and the current input.
Finally, the current hidden state is computed by combining the previous hidden state and the candidate hidden state, weighted by the update gate. This calculation ensures that the GRU selectively updates the hidden state with the most relevant and useful information, while preserving important information from the past (Equation 2.9).

$$r(t) = \sigma(W_r \cdot [h(t-1), x(t)] + b_r) \tag{2.6}$$

$$\tilde{h}(t) = \tanh(W_h \cdot [r(t) \cdot h(t-1), x(t)] + b_h) \tag{2.7}$$

$$z(t) = \sigma(W_z \cdot [h(t-1), x(t)] + b_z) \tag{2.8}$$

$$h(t) = (1 - z(t)) \cdot h(t-1) + z(t) \cdot \tilde{h}(t) \tag{2.9}$$

2.3 Literature review

An overview of the existing work in the area of LiDAR 3D object detection is presented. Provided background in the task as a whole, we then point at relevant work directly related to our proposed solution, i.e. touching the particular issues of point clouds aggregation, time awareness and efficiency.

2.3.1 3D LiDAR object detection overview

Despite the success of 2D object detection methods, the need for robust perception has promoted the exploration of 3D alternatives, which take object poses and dimensions into consideration. Deep learning applied to 3D LiDAR object detection is therefore a research topic receiving high attention these days.
To analyse literature from various perspectives, we first introduce the multiple type of point cloud representations commonly employed, as presented in [2]. Having that background, advanced state-of-the-art architectures are shown.

2.3.1.1 Standard point cloud representation schemes

Considering the data processing particularities mentioned previously, when dealing with LiDAR point clouds the particular choice of data representation plays a critical role. We introduce the main styles of point cloud representation available in literature.

2.3.1.1.1 Projection methods

Given the far level of development reached in NN applied to classical computer vision tasks, the most trivial approach suggests bringing the LiDAR data into the image domain. By rasterizing the point cloud data, a pseudo-image (2D) is obtained and standard image detection architectures can be utilised directly. Diverse projection methods can be found on literature, e.g. Front's View, Perspective View, Bird's Eye View, etc. From these, the latter one (BEV) has received the largest attention. The reason is, due to cars motion being given exclusively on the ground plane, the height component can be disregarded without severe information loss. Furthermore this representation can help with occlusion, it rarely presents bounding-boxes overlapping, and it maintains the natural metric space (physical dimensions). A successful example of such approach is PIXOR [25], which proved the use of BEV projection is suitable and enough for obtaining an efficient single-stage object detector. Alternatively, multi-view settings could also be considered, combining the best of several projection views. An example of such philosophy can be taken from [29], which combines a Perspective View (high density of points), with BEV (sparser representation).

2.3.1.1.2 Volumetric methods

An alternative way of dealing with LiDAR data is by discretizing the sparse and unstructured 3D point clouds into a volumetric 3D grid of voxels (volume elements).

This allows subdividing the point cloud into discrete series, easing its processing. In analogy with images, voxels can be thought as 3D pixels because they do contain depth information. This fact makes them be a more detailed mean of representation than Projection methods. Nevertheless, because of the sparsity nature of point clouds, a large amount of grid cells are often empty, which leads to a waste of computational power. Moreover, due to being a discretization, a trade-off between grid resolution and memory exists.
VoxelNet [30] was an innovative early proposal which achieved successful results with a voxel-based approach. It was, however, deemed still too slow for real-time applications. SECOND [24] employed sparse-convolutions and achieved a much shorter inference time and improved performance. A substantial change in voxelization philosophy was, however, brought by PointPillars [10]. Instead of using regular voxels, point clouds were discretized in vertical columns, defining Pillars. This way of representation, apart from bringing efficiency benefits, allowed the removal of the need of manually tuning the vertical axis binning.

2.3.1.1.3 Point-based methods

As the first big step in deep learning on raw point clouds, Qi et al. [19] proposed a network architecture known as PointNet. Different from other representation methodologies, which transform the raw point cloud data into computationally easier domains, PointNet proved to be capable of learning directly on an unordered set of points. Since the release of PointNet, several approaches were developed to try to face its flaws, e.g. [20], [21]. Despite the accuracy increase that comes from respecting the nature of data more than the alternative methods, the inherent need of point subsampling eventually leads to errors. However, the main negative aspect of Point-based architectures is their increased computational cost, not being precise for real time applications.

2.3.1.2 The State Of The Art

The current SOTA methods are mainly based on the seen forms of representation. It is by adding extra building blocks how novel performance is achieved.
Transformer-based architectures are state-of-the-art for most Deep Learning fields nowadays, which is not an exception when talking about LiDAR point clouds. Former 3D self-attention approaches set the emphasis in point-based ([28], [7]) and voxel-based ([27], [16]) representations. Their development built the foundations for examples applied to the specific object detection problem (MLCVNet [23], 3DETR [18]) proving to be highly accurate. In spite of Transformers' present relevance, and even though there exist efficient Transformer networks (LighTN [22]), the highly strict constraint of embedded-integration in the car makes them lose adequacy in this project.
A totally different approach was recently proposed in [6]. The authors start by renaming the established projection and voxel-based methods as dense detectors, i.e. result of making sparse features denser. In the paper, it is claimed these typologies work fine when the LiDAR detection range is below 75 m, but they are impractical when aiming for higher ranges. This has to do with the fact object centers (com-

monly taken as object references) fall in empty areas because of the high degree of data sparsity, and the convolutions applied to these just diffuse the representations, but carrying the problem along. To address this main issue, they suggest using a fully-sparse pipeline, inspired on Point-based methods, utilizing sparse voxel encoders or sparse attention blocks, reaching State Of The Art results.

2.3.2 Relevant work

In addition to the literature content covered so far, it is left to add up contributions in the directly-related topics of point cloud aggregation, as well as in temporal-awareness.

It is known that aggregation of different frames can help improving overall performance thanks to obtaining a denser point cloud. However, the risk of redundant information being aggregated exists. To contribute to the topic, Fast and Furious (FaF) [15] demonstrated how considering several scans across time can greatly improve the object detection accuracy, with benefits extending to other fields like tracking and motion forecasting. The implementation used a BEV representation of the LiDAR and analysed the differences between aggregating the point clouds following an Early Fusion scheme (at the beginning of the network) or a Late Fusion one (successive deeper layers). Later on, [4] replaced the frame fusion model, which owned 3D convolutions, by a more efficient module based on 2D convolutions and including a map as input alongside the LiDAR scans. Recently, [6] set the emphasis in aggregating only meaningful information. To do that, they introduced the term residual points, to name points which change between consecutive frames, i.e. which are informative. The combination of these points and previous foreground points (history information) define a worthier point cloud.

Alternatively, the use of information from several frames can be performed not only explicitly (via aggregation methods), but instead by considering time relations. It turns out modelling time dependencies is a way of using meaningful past information to enjoy more accurate current predictions. The clearest example of such model architecture is a Recurrent Neural Network. RNNs contain a hidden state which acts as memory, promoting the use of worthy knowledge across time. As the first relevant recurrent proposal in literature, [9] presented a conv-LSTM approach. Voxelized point clouds were fed into a U-Net backbone containing sparse convolutions, and from there the hidden state was updated. EGO-motion compensation of the hidden state was performed manually before computing the current state. After the mentioned success of PointPillars [10], [17] proposed the addition of a convLSTM in the PointPillars pipeline, after the backbone, although the results were not noticeably improving the former architecture. Most recently, FS-GRU [5] was presented, based on the same concept as the two mentioned, but having two substantial changes: a convGRU instead of a convLSTM was used to model time dependencies and enable feature sharing, making the inference time independent from the length of past information used; and the proposal of hidden state EGO-motion compensation within the network and with the help of an auxiliary task.

2.4 Proposed solution

As mentioned previously, our goal is to improve detection results at high ranges through the use of multiple consecutive LiDAR scans.
Along with the evident gains in point density, which are crucial for classification at high ranges, the concatenation of point clouds can also bring improvements in other tasks, such as motion forecasting, as illustrated in [15]. However, approaches of this nature suffer from a linear increase in the amount of data utilised and processed, leading to a significant rise in computation time during inference when the number of past scans used increases.
In contrast, we utilise a recurrent approach to offer a computationally efficient inference solution while taking advantage of past information. The recurrent module allows for the retention of information across time, from prior execution steps, allowing the network to merge new information with internally stored past memory. Such a recurrent approach, despite presenting an increase in convergence difficulties during training, has the potential to solve the problem.
We choose the PointPillars [10] architecture as the basis for our object detection network, owing to its enduring strong performance and efficiency since its original release five years ago. Additionally, the architecture predominantly employs 2D convolutions, which are widely supported by optimisation platforms and deep learning accelerators.
Similar to the approach taken in FS-GRU [5], we propose the incorporation of a Gate Recurrent Unit as the recurrent module in our network. Specifically, we utilised a convolution-based Gate Recurrent Unit, known as convGRU. However, our approach differs from existing methods by utilising heavily processed and latent information from the backbone as the hidden state, instead of relying on encoded pillars. This approach has demonstrated improved precision in detections, despite requiring a more extensive training cycle. Furthermore, we employ a simple 2D convolutional layer as a transformation model for the hidden state. This module is necessary, as we assume that the vehicle collecting the data is in motion, and therefore, the hidden state requires a motion transformation to align with the new LiDAR scan in space. We refer to this network architecture as TimePillars.
The Zenseact Open Dataset [3], containing a considerable number of LiDAR scans arranged in short sequences, is utilized for training and evaluation purposes. Additionally, we establish a baseline with single-frame and non-recurrent multi-frame approaches to facilitate comparison.

3
Data

3.1 Sensing

In the context of autonomous driving, sensors play a critical role by acting as the interface between the software and the real-world environment. The selection of sensors is particularly crucial in perception, as it is not only the most influential decision but also the most limiting factor. Any system designed to perceive the surrounding environment is built upon the sensor package, whether it is based on cameras, LiDAR, or other sensing technologies.
In the case of autonomous driving intended for commercial use, the most commonly used sensors are cameras, radars, LiDAR, and inertial navigation systems (INS). Cameras have become a standard component in any autonomous driving/advanced driver-assistance system application, paving the way for complex systems that require less driver intervention. The rapid progress of machine learning coupled with recent advancements in embedded hardware capable of running increasingly complex neural networks has enabled remarkable achievements in this field.
Radar has also become an ubiquitous sensor in most passenger cars. With a reasonable price range and easier-to-process data when compared to cameras, it has played a crucial role in safety functionalities. Unlike cameras, radar information is directly useful in its raw format by providing distance to obstacles in the horizon. It is also very robust to adverse weather and lighting conditions. However, the resolution of the data obtained is insufficient to detect small objects or classify the environment.
Due to the demand for precise 3D environment perception and robustness to lighting conditions, LiDAR has become increasingly popular. In this work, we focused mainly on this sensor, attempting to process and classify objects in the 3D point cloud it provides.

3.1.1 LiDAR

LiDAR, also known as Light Detection and Ranging, offers a 3D point cloud of a given area by emitting and receiving light beams, with technological specifics differing among manufacturers and models. The principle objective remains the same: to determine the distance to surrounding objects by measuring the reflected light beam.
Mechanical LiDAR, which scans the environment by spinning, has been used in various applications, but its high cost has limited its application in the past. In addition, the low durability and high maintenance required by these precise and fast spinning devices make them unsuitable for widespread use.

Solid state LiDAR, on the other hand, has recently emerged as a viable alternative for mass production environments. The technology varies significantly among manufacturers and the concept of solid state has been expanded and experimented with, but the main objective is to minimise movement and fragile components in the sensor. With many car brands announcing the addition of these sensors to their lineups, there has never been a greater need to solve the object detection problem with them.

3.2 Dataset

This study utilised the Zenseact Open Dataset [3], a multi-modal autonomous driving dataset collected over a period of 2 years across 14 countries. Two principal subsets of the dataset, namely frames and sequences, were employed for our analysis.
In addressing the three-dimensional problem tackled in this paper, the LiDAR scans and their corresponding 3D bounding box annotations were utilised for both training and validation. To account for temporal awareness in our approach, we incorporated odometry position, which was either utilised to transform the scans or fed into the network for internal transformation.

3.2.1 Sensor Setup

The utilised data acquisition vehicle includes a variety of sensors, such as LiDARs, cameras, radars, GNSS, and additional ones. The arrangement of sensors in the vehicle is illustrated in Figure 3.1.
For this project the combined point cloud from the 3 LiDARs was used. This sensor package includes a high-resolution Velodyne VLS128 mounted on the top of the vehicle, which generates the majority of the point cloud data owing to its outstanding resolution and detection range. Moreover, a Velodyne VLP 16 is positioned on either side of the vehicle to enhance the point cloud data in the immediate vicinity of the car, thereby filling in the blind spots left by the larger top sensor.

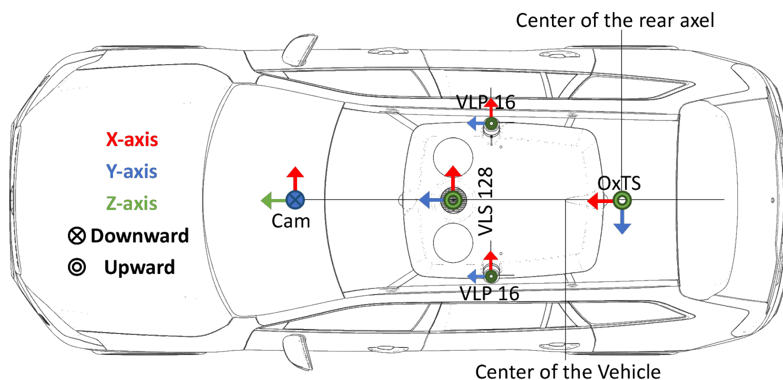


Figure 3.1: Illustration of sensor position on acquisition vehicle [3]

3.2.2 Frames

The models outlined in this paper were trained primarily using frames, a subset of data comprising 100,000 camera-LiDAR pairs. It is important to note that these pairs are not temporally related to other pairs in the dataset, meaning they represent distinct data, temporally and spatially sparse. However, each of the 100,000 pairs includes a core frame, which is the annotated pair itself, as well as one second before and after of unlabelled data. As a result, each frame, includes a total of 21 LiDAR scans captured at approximately 10 Hz: 10 scans captured prior to the core frame and 10 scans captured post, resulting in a total of 2.1 million LiDAR scans. Of these, only 100,000 have annotations (the core scan). The vehicle's pose and orientation are provided for each LiDAR scan, facilitating point cloud aggregation around the core frame. Figure 3.2 illustrates the data contained within a single frame.

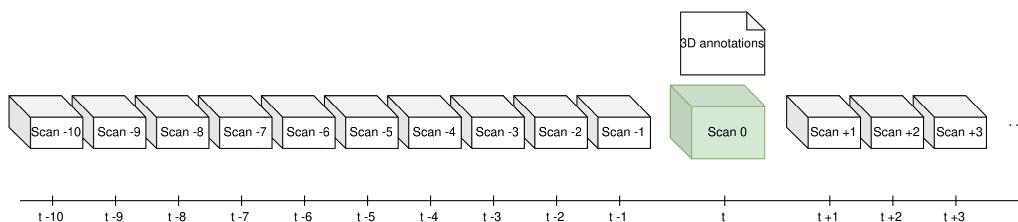


Figure 3.2: Illustration of data in single frame

As previously stated, these frames were captured over a period of two years in 14 different countries. The geographic dispersion of the frames is depicted in Figure 3.3. Furthermore, Figure 3.4 illustrates the distribution of the data with respect to the time of day, road type, and weather conditions.

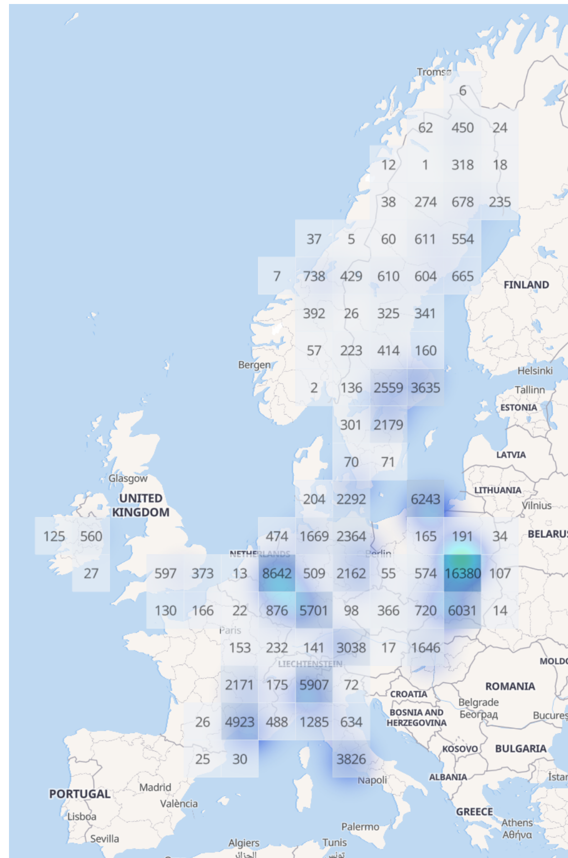


Figure 3.3: Geographical distribution of the frames subset [3]

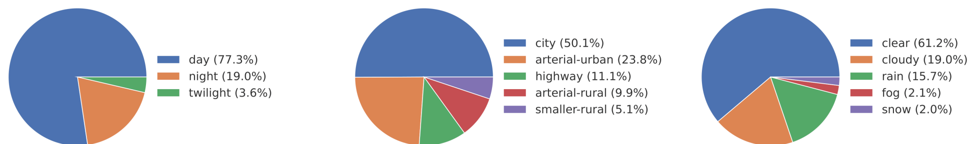


Figure 3.4: Distribution of frames across time of day, road types and weather conditions [3]

The introduction of this novel dataset is particularly intriguing as it presents a comparative analysis of annotation distribution across distance from the care with other commonly used datasets (refer to Figure 3.5). In addition to the substantially higher number of frames, the strong presence of annotations at greater distances was a crucial factor in our decision to utilize this dataset. Our aim of efficiently detecting objects on aggregated point clouds necessitated the use of high-range annotations, particularly where the point clouds are sparser and aggregation can yield the greatest benefits.

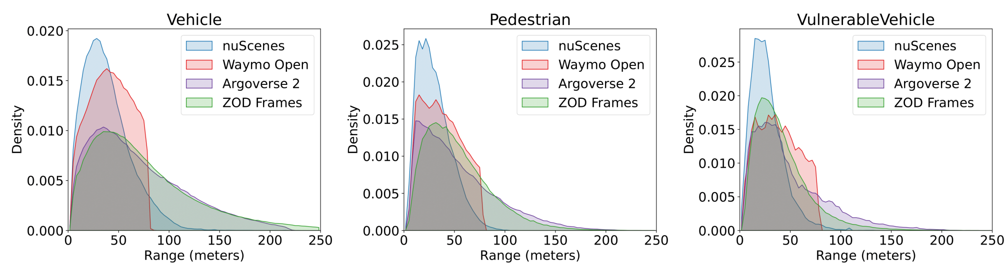


Figure 3.5: Distance distribution of annotations compared to other popular datasets [3]

3.2.3 Sequences

The dataset's sequences subset comprises of 1473 segments of 20 seconds in length. Analogous to the frames subset, as of the composition of this report, solely the keyframe at the center is annotated, with a duration of 10 seconds before and after it.
One benefit of this particular subset is its increased duration, which allows for the testing and evaluation of networks over longer time sequences.

4

Methods

This chapter aims to provide further details on the main focus of study, including the methods utilised to establish a baseline and contribute to the research. The proposals presented in this section are categorised based on their approach to handling LiDAR data frames.

4.1 Single frame

The first section discusses single-frame methods, which are characterised by their reliance on a single LiDAR scan as input. Although such methods initially suffer from sparsity issues arising from the limited amount of data available from a single sensor frame, they offer the advantage of being computationally simple and flexible with respect to input. Moreover, they typically have the lowest inference times. It should be noted that the single frame used corresponds to the core frame in the dataset, as described in Chapter 3.

As per Section 2.4, we leverage the existing literature and extend the PointPillars methodology [10] as our primary reference for a single frame. Additionally, we utilise it as the primary benchmark for evaluating the proposed methods described later. While retaining the fundamental components of the original technique, we introduce minor modifications to establish our single frame baseline.

4.1.1 PointPillars

The architecture of PointPillars can be observed in Figure 4.1. At a high level, input point clouds are directly fed into a Pillar Feature Net block, which encodes the raw data into a 2D pseudo-image. This pseudo-image is then is passed through a 2D convolutional Backbone which generates a low-level feature representation. Finally, a Detection Head takes care of outputting the prediction bounding boxes for the object detection task.

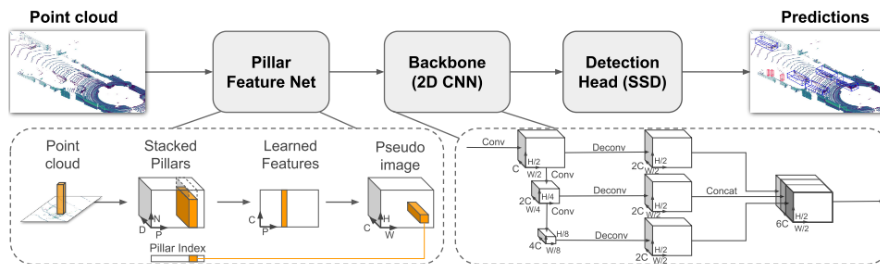


Figure 4.1: PointPillars block diagram [10]

The Pillar Feature Net module maps the input point cloud to a pseudo-image. In this block it resides most of the innovation from the original paper. Unlike standard voxel methods, pillarisation is used as discretization method instead. Pillars can be seen as vertical column slices, spanning only in the ground dimensions (x and y) while having a fixed height (z) value along the vertical dimension. This approach eliminates the need for z-axis binning in voxelization and captures the essential vertical information while simplifying the computational pipeline. It enables efficient processing using 2D convolutions and ensures consistent input dimensions for the neural network.

In this setting, the point cloud is first discretized into an evenly spaced x-y grid, and a set of pillars are obtained. By stacking them, a dense tensor can be originated, which acts as input to a simplified PointNet [19] feature encoder. Using previously-stored grid positional indices, the learned encoded features can be scattered back into the original grid, achieving the desired 2D pseudo-image representation.

The image is then passed through a 2D CNN backbone. Note the ease of 3D convolutions avoidance, without major accuracy drop, due to pillarisation. The network, not far from following a U-Net philosophy, shrinks the latent space in its downsampling part, and upscales it back using transposed convolutions and concatenating progressive reconstructions. This results in an input-output scaling factor of 1/2, i.e. halving the grid.

A Detection Head makes the final network predictions. The original work uses Single Shot Detector (SSD) [13], producing anchor boxes for the chosen outputs: the localisation regression targets (position, size and angle) and the classification ones (object class and discretized heading direction).

4.1.2 Proposed baseline

To enhance our single frame baseline, we propose some modifications from the original PointPillars paper. These modifications aim to improve performance or simplify the architecture in terms of computational complexity. Specifically, we focused on enhancing the data preprocessing and the Pillar Feature Net block. We believe that improvements in these areas, which are crucial stages of the pipeline closer to the raw data, can have a significant impact on the performance of the established PointPillars architecture.

An overview of the system is shown in Figure 4.2 and further detailed below.

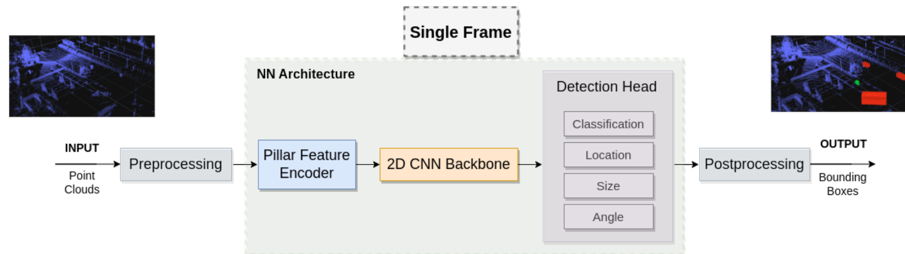


Figure 4.2: Single Frame baseline block diagram

We start by describing the preprocessing applied to the data, most critically on the input, but also required by the ground truths for data loading concerns. This way, and for the sake of completeness, the labels encoding utilized is also presented.

4.1.2.1 Input Preprocessing

Pillarization brings many benefits, as discussed, which reasons why we use it as starting point. However, as claimed by the authors of the paper, a large number of pillars from the resulting set are empty, or they own very high sparsity levels (around 97%, [10]). This is a large limitation, commonly faced when dealing with LiDAR point clouds, which the paper attempts to tackle by setting a limit on the number of non-empty pillars per sample (denoted as P), sampling out or using zero-padding to reach the desired limits.

Dynamic voxelization was first introduced in [29]. Differently from the standard "hard voxelization", it suggests that there is no need to set a predefined number of points per voxel (pillar). Instead, dynamically-created voxels allow using every existing point in the grid. We borrow the intuition behind Dynamic Voxelization, and apply it in the pillars context.

The preprocessing stage, aware of this modification, firstly defines the grid of interest. Consecutively, the cloud points are augmented from their basic representation (x, y, z, reflectance) with the additional channels information suggested in the original paper: distance to the arithmetic mean of all the points in the pillar (x_c, y_c, z_c), and offset to the pillar center (x_p, y_p). At this point, the pillar indices corresponding to the extended points are stored for future scattering. The output of preprocessing, and therefore the input of the NN, turns out to be the truncation/padding of both the points and their matching indices to a desired number. This value, representing the total number of points, N_t, is set as $N_t = 200.000$ after exhaustive testing. Note the key difference in formulation here, as one of the fundamental peculiarities of Dynamic Voxelization: the points are treated as a whole when doing the latter, not per pillar.

Denote N_p as the number of points per pillar, D as the number of channel dimensions, and P as the number of non empty pillars per sample. Our preprocessing leads to a tensor of points of shape (N_t, D), differently from (D, P, N_p) in the original paper, thanks to Dynamic Voxelization. This level of abstraction has several beneficial implications: information loss is minimized (all points available for pillarisation are used, no sampling is applied per pillar); the computational cost is reduced due to not pillar-specific padding; and the resulting embeddings are deterministic (overcoming the stochastic nature of points dropout).

4.1.2.2 Labels

The ground truth data are also processed to match the configuration we believe is best to promote learning. This step accounts for the four outputs involved: classification, location, size and angle.
Unlike PointPillars and most object detection algorithms, we do not use anchor boxes. Instead, a prediction is outputted for each of the cells in the grid directly. Despite losing single cell multi-object detection capabilities, this way avoids the tedious and commonly imprecise tuning of anchor boxes parameters, and smooths processing and inference computation.
The first step in the creation of labels concerns the extraction of the grid cell indices which are overlapped by an object. This is fundamental, as these are the cells which own the associated ground truth. We do this by defining a region of interest coefficient, which acts as a factor defining the fraction of the overlapping area taken into consideration. The latter is set as 1.0 for the background class and 0.5 for the rest. Orientation is not taken into account when defining the region of interest, i.e. the area defined follows the x and y axis directions.Once obtained the indices, the labels can be created.
The size output presents the simplest processing. The length, width and height of the ground truth bounding boxes are encoded directly, without further work.
The classification labels correspond to the one-hot encoded class ground truths.
The angle ground truths, conversely, need special care given that the dataset provides the object's orientation in its quaternion representation. The heading (yaw) angle is extracted and from it, the value is encoded by its sine and cosine counterparts. Note this is more beneficial than direct angle regression because the order of magnitude of the tensors (scaling factor) does not constrain the decoded heading value.
Finally, the locations of the centres of ground truth objects are encoded as an offset from the centres of each of the overlapping cells in the grid, i.e. the distance is regressed, not the absolute location value, delimiting the engineering problem.

4.1.2.3 NN Architecture

4.1.2.3.1 Pillar Feature Encoder

The Pillar Feature Encoder takes the (preprocessed) input tensor defined above, encodes it and scatters the points back to their original grid positions, creating a BEV pseudo-image. Given the utilization of Dynamic Voxelization, the simplified PointNet is slightly adapted accordingly.
The original paper applies a 1x1 Convolution (linear layer) to the input, followed by BatchNorm and ReLU, obtaining a (C, P, N_p) tensor, where C stands for the number of channels. Right before the final scatter max layer, a max pooling operation is applied to the channels, leading to a latent space of shape (C, P).
This is where we differ in architecture. Given our tensor was initially encoded as (N_t, D), leading to (N_t, C) after the former layers, we remove the max pooling operation as it loses adequacy, considering our approach treats all the points as a whole and therefore there is no longer a need for a dimensionality reduction in

the channels. However, the concept of taking the features which maximise the representation remains, and it is implicit in the scatter max layer which follows. Its application leads to a new tensor where feature maximum values are scattered to specific grid positions.

The resulting BEV pseudo-image is of shape (L, W, C), where L and W correspond to the length and width of the grid, respectively. These values are chosen based on the ideal driving perception capabilities, the dataset, and computational constraints. The grid cell size is set to 0.2 m (0.16 m in the original paper) to ease data processing. Aiming for long-range longitudinal detections, we enjoy enjoy objects range of (0, 120)[m], giving $L = 600$; and consider objects at a width range of (-40, 40)[m], bringing $W = 400$.

4.1.2.3.2 Backbone

The architecture from the original paper was respected. A 2D CNN is utilized. No large improvements were considered in this module, as the network suggested owns a suitable depth and the needed elements to extract valuable features, while remaining efficient.

Three downsampling blocks (Conv-batchnorm-ReLU) are utilized. Strides are the chosen mean of spacial reduction. Each of the blocks present the downsampling stride (S) in the first convolution to accommodate the input, and a stride of 1 in the remaining layers. Kernel sizes are 3×3 for all the convolutional layers, and the number of channels $C = 64$. The blocks then result in: Down1(S, C), Down2($2S$, $2C$), and Down3($4S$, $4C$).

Three upsampling blocks follow. Through transposed convolutions, they apply the corresponding upsampling strides to each of the progressively downsampled versions: Up1(S, $2C$), Up2($2S$, $2C$), Up3($4S$, $2C$) so that each of the output sizes are $(L/2, W/2, 2C)$. A final concatenation of the different strided versions leads to a tensor of shape $(L/2, W/2, 6C)$.

4.1.2.3.3 Detection Head

A Single-Shot detector philosophy is followed, meaning the algorithm aims to detect objects in a single pass, and no particular region proposal algorithm is employed.

The features from the backbone are then fed into linear layers (1x1 convolutions) which accommodate the individual outputs for classification and localization regression. Each of the layers corresponds to a certain output of the object detection task, being these: classification, location, size and angle.

It is to be noted that, of all regression outputs, only the size has an activation function, and the latter is a ReLU. This is done to prevent the size from taking negative values, which is not suitable for the other outputs. Alternatively, the classification convolution is followed by a Softmax activation layer which outputs the class probabilities.

4.1.2.4 Postprocessing

The NN output tensors require additional postprocessing effort in order to filter and extract the final bounding boxes.
First of all, the predictions need to be decoded from their label-encoding format seen above. Classification scores are read from the Softmax output, obtaining the predicted classes. The locations of the predicted centres, encoded as offsets from the grid cell centres, are also extracted back. The final angle values are the result of applying the quadrant respecting arc-tangent operation (atan2) to the encoded sine and cosine versions.
Once decoded, class score filtering is applied to the tensors, followed by Non-Maximum Suppression (NMS). The former removes low confidence predictions, while the latter is the key to obtain a single (the best) bounding box associated to an object. 2D rotated Intersection Over Union (IoU) is used to decide the degree of overlap between two rotated bounding boxes.

4.2 Multi-frame

So far, the method presented exploits information just one frame at a time. Even though the performance of single frame approaches has been proven to be high, making use of information from multiple consecutive frames could bring a boost in learning, given more information is available, facing the sparse nature of LiDAR data.

4.2.1 LiDAR point clouds aggregation

With the aim of exploring the area and establishing a multi-frame baseline to reflect on the results, we aggregate point clouds from a number of previous time instants, up to the current one. The latter is done as part of preprocessing, meaning the main pipeline elements presented above are not affected.
The aggregation is done via concatenation. A parameter accounting for the number of desired LiDAR sweeps, denoted as N_{sweeps}, is defined. We take the core frame in the dataset as current-time reference (newest information) given it contains the ground truth annotations. From it, $N_{sweeps} - 1$ scans are considered. Considering the total number of points for a single frame was as explained above, $N_t = 200.000$, the multi-frame network takes as input, instead, $N_{sweeps} \cdot N_t$ points.
To provide extra temporal information to the network, we concatenate timestamp data in the input channels dimension. The timestamp information is given as an offset from the core (current) frame and expressed in the range $timestamp \in [0, 1]$, where the core takes a zero offset.

4.2.1.1 Raw data EGO motion compensation

The steps provided so far lead to a successful aggregation of LiDAR point clouds, but miss a crucial detail: the EGO-vehicle is moving at a certain speed while the sensor collects data, and therefore one must compensate for its motion.

From the existing typologies of compensation scenarios that can be found, this baseline compensates the raw point clouds directly, and as part of preprocessing (outside the NN). Alternative approaches are studied in Section 3 below.

Furthermore, it is important to mention we do this in an scan-wise fashion. The latter means that the points are moved to a frame taken as reference (core frame as mentioned), accounting for the spacial offset inherent to belonging to different sensor scans. An additional compensation scheme could be also applied: point-wise compensation. Such an approach considers the addition of point cloud unrolling, i.e. moving the points within each scan to compensate for the ego motion that happened during the scan itself. This is not object of work in this baseline, left open for future work if intended.

In line with Chapter 3, the dataset counts with orientation data for each frame. The vehicle's poses across time are known and correspond to 4×4 transformation matrices from a fixed dataset (world) reference. Furthermore, LiDAR calibration data are available. The calibration acts as a mapping from the LiDAR coordinate system into the EGO vehicle, and therefore it is assumed to be constant (fixed sensor mounting).

In this context, the steps to achieve EGO motion compensation are the following:

- Consider $T^{w}_{ego_{pre}}$ and $T^{w}_{ego_{core}}$ to be the individual poses of a previous frame, and the current (core) frame, respectively, both expressed in the world coordinate system. The relative transformation between a previous frame, and the current one can be computed as follows:

$$T^{ego_{core}}_{ego_{pre}} = (T^{w}_{ego_{core}})^{-1} \cdot T^{w}_{ego_{pre}} \tag{4.1}$$

- Express the point cloud to be compensated, given implicitly in the sensor coordinate system, PC_{sensor} in the EGO-vehicle one through the LiDAR calibration, T^{sensor}_{ego}:

$$PC_{ego} = PC_{sensor} \cdot T^{sensor}_{ego} \tag{4.2}$$

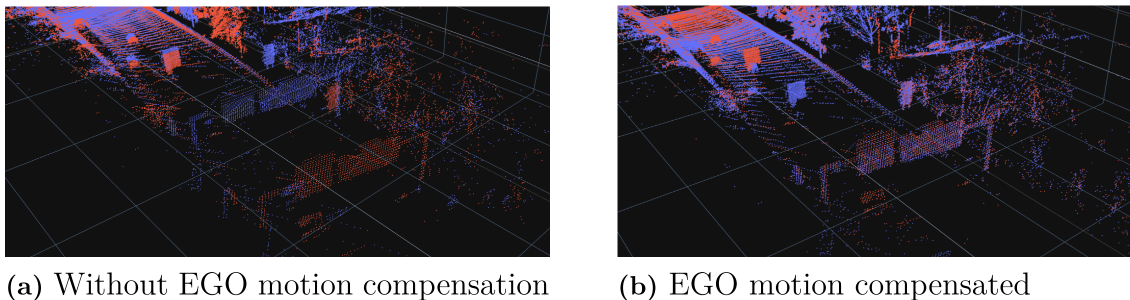
- Use the odometry computed beforehand above (relative frames transformation) to transform the previous frame point cloud, compensating for EGO motion:

$$PC_{ego_{compensated}} = PC_{ego} \cdot T^{ego_{core}}_{ego_{pre}} \tag{4.3}$$

- Express the point cloud back in the sensor coordinate system through the calibration:

$$PC_{sensor} = PC_{ego_{compensated}} \cdot (T^{sensor}_{ego})^{-1} \tag{4.4}$$

Figure 4.3 shows the difference in appearance of the point cloud with/without the application of motion compensation. The point cloud in red corresponds to the core (current) frame, while the one in blue is associated to 10 frames in the past, i.e. 1s before. A clear difference is noticed when looking at image (b), in which the scan-wise compensation manages to aggregate the information from both frames at the correct spatial location, giving a richer point cloud representation, while the opposite does not hold.



(a) Without EGO motion compensation **(b)** EGO motion compensated

Figure 4.3: Ego motion compensation comparison for t (red) and $t-10$ (blue)

It is by the addition of several consecutive frames when the point cloud gets noticeably denser, counting with more points representing the same object in space, which might in turn ease the object detection task, see Figure 4.4.

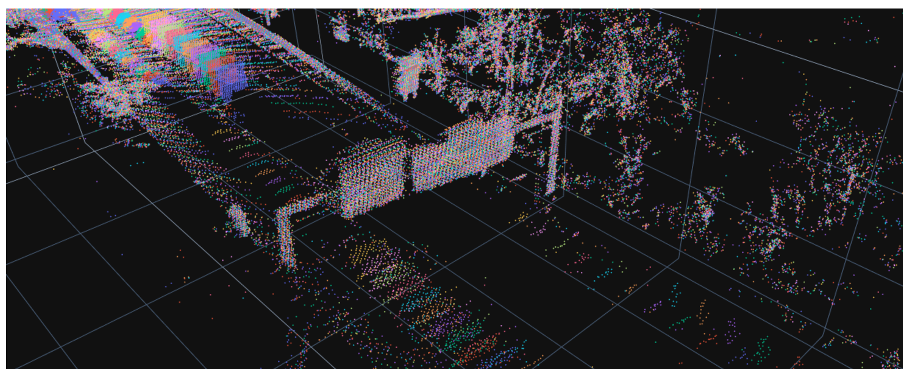


Figure 4.4: 10 consecutive frames aggregation with EGO motion compensation

4.3 Temporally-aware

Multi-frame methods served as ground introduction about the intuition behind aggregation, and the possible benefits of also considering past data in the learning process. Nevertheless, it can trivially be seen how the number of points quickly grows with the utilisation of several frames in the past. This would demand highly complex problem-specific engineering solutions to try to figure out an efficient input representation of LiDAR point clouds that would boost aggregation, while remaining computationally cheap.

4.3.1 Time recurrency

We believe a robust solution, owning a high degree of generalisation and flexibility, could only come if a shift of thinking were produced. Such a change in approach would consist of avoiding heavy preprocessing, aiming instead for a NN realisation that could benefit from end-to-end Deep Learning. In this context, we think the solution passes by endowing the NN with a memory, which could keep track of worthy features across time, and disregard non useful information in the long term, based on previous experience and knowledge about the environment.

The building block which materialises the previous idea has been introduced before: Recurrent Neural Network.

4.3.1.1 convGRU

Vanilla GRU architectures are suitable for a wide range of tasks (e.g. NLP), but are not ideal to use directly in the context of this project. Considering the network works with (pseudo-) image features, where spatial information matters, convolutions stand out as the preferred way of processing data. In addition, the efficiency of convolutions can be substantially superior than applying RNNs directly on spatial data; and convolutions bring translation ínvariance robustness, which improve generalisation to inputs with spatial transformations associated (translations, rotations, and scaling).
That is the reason why we consider, instead, the use of convGRU, which is not other but its convolutional version. Furthermore, to leverage efficiency, we reduce the number of required convolutional layers thanks to merging operations of similar nature. In particular (see Eq. 4.5 - 4.7), by looking at the previous equations (Eq. 2.6 - 2.9) one can see both the reset and update gates share the same input variables (just owning different learnable parameters). This allows the application of a single convolution (*), of duplicated number of filters, via input concatenation.

$$[r(t)\,, z(t)] = \sigma([W_r\,, W_z)] * [h(t-1), x(t)] + [b_r\,, b_z]) \tag{4.5}$$

$$\tilde{h}(t) = \tanh(W_h * [r(t) \cdot h(t-1), x(t)] + b_h) \tag{4.6}$$

$$h(t) = (1 - z(t)) \cdot h(t-1) + z(t) \cdot \tilde{h}(t) \tag{4.7}$$

The convGRU architecture, with its gating mechanisms and the substitution of products by convolution operations, allows the model to adaptively learn which information to forget, which to update, and which new data to incorporate. This enables the capture of long-term dependencies in sequential data.

4.3.2 TimePillars

Take as starting point the Single-Frame baseline presented above, and consider the insertion of a Recurrent Neural Network to it, at the right pipeline place and surrounded by the needed components. The resulting architectural modification now allows the system leverage information across time. This is, on a high level, the substantial addition behind our approach, which we term as TimePillars.

4.3.2.1 Model architecture

The model architecture that we propose is depicted in Figure 4.5.

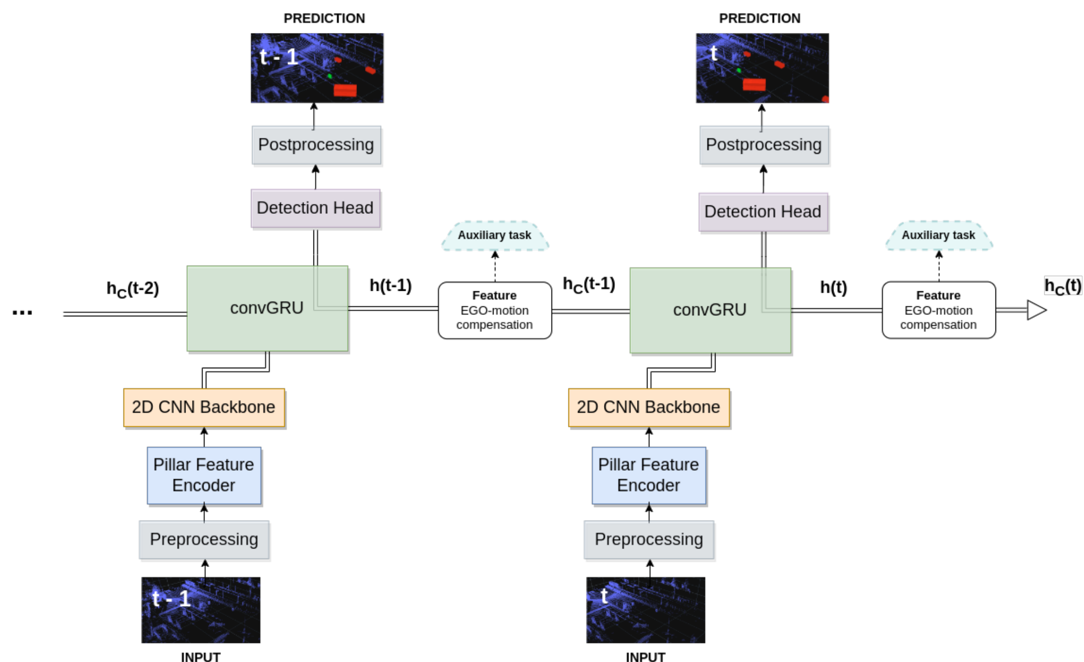


Figure 4.5: TimePillars model architecture

As seen in the single frame section, input point clouds from a given time instant are passed through a preprocessing stage which sets the foundation of Dynamic Pillarisation, as well as prepares the remaining input data for future utilization by the NN. The latter is followed by a Pillar Feature Encoder, which allows the point cloud discretization into a BEV pseudo-image; and consecutively by a 2D CNN backbone, which boosts the features representation.
At this point (after the backbone), the NN has shrunk the data into a low-level latent space which synthesises the relevant information about the driving scenario. That is the key reason why we consider the placement of the convGRU in such pipeline position, as further experimental evaluation to be introduced below shows. The hidden state from the considered time step is fed into the Detection Head and postprocessed, leading to the bounding box predictions associated to that timestamp.

4.3.2.1.1 Feature EGO motion compensation

The hidden state features outputted by the ConvGRU are given expressed in the coordinate system of the previous frame. If stored and utilized directly for the computation of the next prediction, a spatial mismatch, as explained before, would occur.
Raw point cloud EGO motion compensation, as intended in the multi-frame baseline presented, solved the compensation problem when using multiple input point clouds, despite it being a computationally-demanding operation.
The solution in the recurrent context, however, demands for the compensation to take place at feature level. This, in turn, makes the hypothetical solution more efficient, but hardens the problem.

We modify the preprocessing to store, as an additional (optional) NN input, odometry data, i.e. relative transformation matrices from source to target consecutive frames.
At this point, two alternative approaches to face the problem are explored: interpolation compensation; and convolutional-based compensation.

4.3.2.1.1.1 Interpolation-based One way to fix the feature EGO motion compensation problem is to use classical interpolation methods. In particular, (pseudo) image bilinear interpolation can be applied to obtained the transformed features.
To do so, the first step concerns the computation of the location of the cell-center coordinates at the previous time step (t-1). Once obtained, through the previously stored odometry information, the coordinates of the previous frame are transformed to the current one. These are the coordinates which are then used to perform bilinear interpolation. More specifically, we take the hidden state features from the previous time instant and interpolate them to the newly obtained coordinates. The output of the latter is a feature map ready to be used at the current time (t), compensated for EGO motion.

4.3.2.1.1.2 Convolutional-based Feature interpolation achieves the desired feature transformation, but it suffers from efficiency problems (problem-specific interpolation), as well as innacuracy (linear approximation). More advanced interpolation techniques could be explored, of higher polinomial order, but we believe the nature of the solution might not be the most suitable one.
Instead, and inspired by the FS-GRU paper [5], we perform a convolutional-based approach to transform the hidden state features. The original paper, however, provides limited information about it, and therefore we have implemented our custom architecture.
The approach we take provides the NN with the needed information to perform the feature transformation via a convolutional layer. We start by taking the odometry transformation matrix, i.e. the needed operation that one would need to perform to successfully transform the features. Next, we extract from it just the 2D information (rotational and translational counterparts) of the matrix:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & tx_{14} \\ r_{21} & r_{22} & r_{23} & ty_{24} \\ r_{31} & r_{32} & r_{33} & tz_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

i.e. r_{11}, r_{12}, r_{21}, r_{22} for the rotational part, and tx_{14}, ty_{24} for the translational, respectively.
This simplification avoids the main matrix constants and works in the 2D (pseudo-image) domain, simplifying it from 16 values (4×4) to just 6.
We then flatten the matrix, and extend it to match the shape of the hidden features to compensate: (N_frames - 1, length/2, width/2, 6). This representation makes it suitable for each latent pillar to be concatenated in the channels dimension of the hidden features to be compensated.

Finally, the hidden state features are fed into a 2D convolutional layer which fits the transformation process. A 3×3 filter is used, and no activation constrains the output to provide operational freedom.
Note a key aspect that derives: the execution of the convolution does not guarantee that the transformation is performed. Channel concatenation just provides the network with extra information about how it could eventually be performed. This is the main reason why we define an auxiliary task, whose aim is to guide the network through the transformation process, to guarantee a successful compensation is achieved.

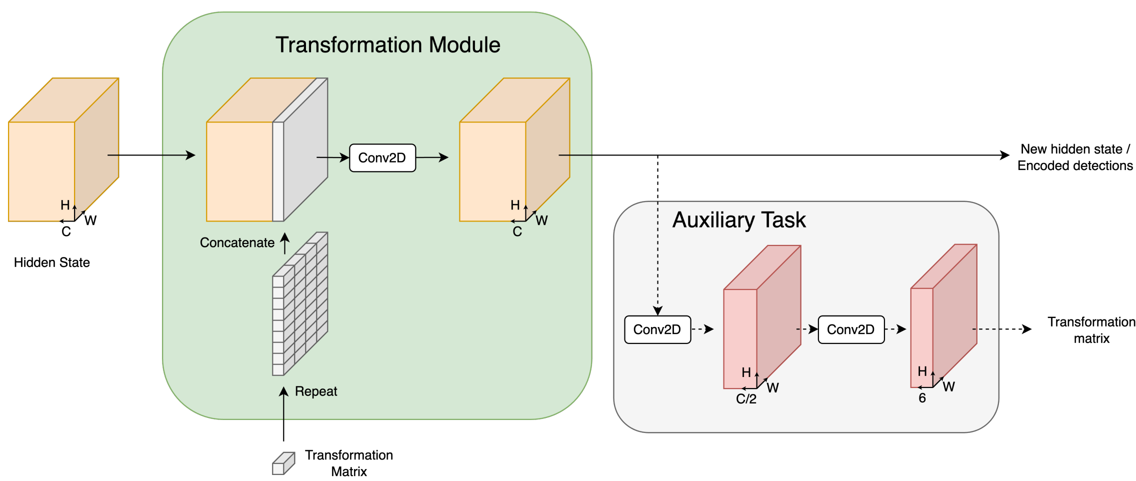


Figure 4.6: Illustration of convolution-based transformation module and auxiliary task

- **Auxiliary task:**

The auxiliary task receives as input the output of the convolutional layer, which encoded the transformation process. The features are fed into a lightweight CNN composed by just two convolutional layers. Their aim is to exploit the latent space representation under supervision. Over time, the resulting features, provided the effect of the auxiliary task, become succesfully compensated for EGO motion.
The use of the auxiliary task is restricted exclusively to the training process. Once the network has learnt to properly transform the features, it loses adequacy. That is why the module is not considered at inference.
Further detailed experimental work, contrasting the utilization or not of the auxiliary task, can be found in the next chapter.

5
Experimental evaluation

Extensive experimental work is presented in this section. Firstly, its aim is to present the relevant details behind data processing and training. Then, the evaluation metrics used are introduced. With them, results achieved with the methods proposed are shown. Finally, their robustness to changes and generalisation capabilities are tested through ablation studies.

5.1 Data Handling

Dealing with point cloud data demands careful processing. A especial concern is the file size data own, which does not allow storing in memory several data samples simultaneously.
After conducting an initial analysis, it was determined that the primary bottleneck in the process was the input pipeline. This was, as introduced, due to the file reading time and the complex structure of the large point cloud files, which resulted in an inefficient training loop.
To address this issue, the dataset was converted to TFRecords format offline, given its immense size. This conversion effectively eliminated data loading as a bottleneck in the training pipeline, allowing us to split the dataset into multiple files and only load the necessary data into memory. Moreover, the use of TFRecords enabled fully in-graph execution for data loading, which further improved IO speed during the training process.

5.2 Training

We provide additional valuable training-specific information. First, we mention the class labels used. The loss functions employed for each of the object detection variables are described. We then mention hyperparameter values which boost the learning, and focus on configurations that help with the class imbalance context. Lastly, we provide some details about how a successful recurrent training on the novel ZOD can be accomplished.

5.2.1 Class labels

The classes we train on are the following: vehicle, vulnerable vehicle, pedestrian, animal, inconclusive and background.

We focus on the performance of the first three with respect to the background. This choice has to do with their predominant presence in the dataset, and literature relevance. However, we believe it is important that networks see other sources of data, like animals, and consider fundamental to have an inconclusive class to account for samples whose classification is not trivial, even at human-level performance.

5.2.2 Loss functions

5.2.2.1 Classification: Focal Loss

In object detection, typically, the number of background (negative) examples is very large compared to the number of foreground (positive) samples. This class imbalance situation can lead to challenges in training, as the presence of many easy negative examples can dominate the learning process.
The Focal Loss, first introduced in [12] is picked as the classification loss in this work. Its choice aims to alleviate the impact of class imbalance by assigning higher weights to hard, misclassified, examples while down-weighting easy examples.
Equation 5.1 shows its mathematical formulation:

$$\text{FocalLoss}(p_t) = -\alpha_t (1 - p_t)^\gamma \log(p_t) \tag{5.1}$$

Where:

$$\begin{aligned} \text{CrossEntropy}(p_t) &= -\log(p_t) \\ \text{FocalTerm}(p_t) &= (1 - p_t)^\gamma \\ \alpha_t &: \text{Balancing parameter} \\ \gamma &: \text{Focusing parameter} \\ p_t &: \text{Predicted probability of the positive class} \end{aligned}$$

The loss function is therefore a scaled cross-entropy loss, where the scaling factor decays dynamically to zero as confidence in the correct class grows. It achieves this by introducing two additional parameters: the balancing parameter, α_t, and the focusing parameter, γ. The latter determines the rate at which easy examples are down-weighted compared to hard examples, while the former scales the loss depending on the desired importance of positive and negative samples in the process. The values utilized correspond to the common choice of $\alpha_t = 0.5$ and $\gamma = 2$.

5.2.2.2 Regression: Huber Loss

The Huber Loss, frequently named also as Smooth-L1 loss, is chosen as our regression loss. It therefore takes care of the following object detection bounding box variables: location, size and angle.
The presence of outliers is common in the task of object detection. This means a standard choice of regression loss, like the L1 Loss (non-differentiable and the difference of errors is not squared) or the MSE (highly sensitive to outliers) are not suitable loss candidates. The Huber Loss, conversely, combines the best properties of L1 and L2 losses. It provides a smooth and robust loss function that is both

differentiable and less sensitive to outliers. The key lies behind the fact that a quadratic mapping is employed when values are within a threshold, and a linear one takes place in a region in which outliers are found.
Its expression is presented in 5.2:

$$\text{Huber}(y_i, \hat{y}_i) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{5.2}$$

where y_i and $\hat{y}_i$ stand for the ground truth and prediction samples, respectively, and the parameter δ is the threshold that determines when the loss transitions from quadratic to linear.
The adopted threshold values are of $\delta = 1.0$ for the location and size, while $\delta = 3.0$ is used for the angle, considering a higher outlier tolerance proved to be of help to encourage the learning of objects' rotation values.

5.2.2.3 Masking

Seen the loss functions employed, details regarding their masking are provided.
Particularly, we aim to remove the influence of the inconclusive and background classes in the final loss. This is done by applying a mask to each of the regression outputs (not to the classification one) to filter out samples which are associated to ground-truth background or inconclusive labels. Doing this is key in search of a successful training loop, as we want to minimise the loss exclusively with respect to the model's performance on the relevant classes.

5.2.3 Configuration

The configuration parameters that, as a result of tuning, brought best training performance and results are provided.

5.2.3.1 Hyperparameters

We start by introducing the hyperparameters whose choice was extended to all the models trained, once the best configuration was adopted. These therefore remain fixed across different trainings. They are contained in Table 5.1 for ease of consultation, and subdivided in point cloud/BEV-specific, learning, and weighting. The values presented proved to be the most suitable for each role.
The justification for the BEV-related settings can be found in the method section 4.1.
It is worth remarking the utilisation of AdamW [14] as optimiser, given it manages to decouple weight decay regularisation from the learning rate setting. The latter (weight decay) is used as regularisation, instead of techniques like Dropout, considering it is more deterministic and therefore intuitive to tune. A learning rate scheduler is used, which scales the hyperparameter by a factor of 0.8 based on validation loss performance, and with a patience value of 2 epochs.
To give some learning-tasks more importance than to others, we consider additional loss task weights. The highest weight was set to the focal loss, as it proved to the beneficial, considering a well-performing classifier helps on the remaining tasks.

The same importance was given to the three regression variables. Alternatively, the auxiliary task employed in the convolutional-based version of TimePillars, showed that a low weight is enough for the transformation learning to take place.

Hyperparameter	Value
Total number of points (N_t)	200,000
BEV cell size	0.2
BEV resolution	400 × 600
Longitudinal detection range	120 m
Optimiser	AdamW
Learning rate decay	0.8
Regulariser	Weight decay
Focal loss weight	3.0
Location loss weight	2.0
Size loss weight	2.0
Angle loss weight	2.0
Auxiliary task loss weight	1.5

Table 5.1: Model-fixed hyperparameters.

The remaining parameters, not mentioned so far, correspond to the ones whose values were adopted differently depending on the model. A clear case is the learning rate, chosen as $lr = 0.001$ for the single frame (PointPillars) implementation given its training from scratch, but often reduced a decade to values like $lr = 4e - 04$ when fine-tuning. Similarly, the regularisation effect through weight-decay was kept small when training from the beginning $decay = 1e - 05$, but increased when transfer learning was employed, e.g. $decay = 0.1$ brought the best TimePillars results achieved.
Networks were trained, most often, for around 30 epochs, usually long-enough given the large size of ZOD Frames. The weights were stored for each epoch, available to be loaded back at inference. To extract the best results from each trained model, a script analysing metrics performance was run on the test set for each saved epoch.

5.2.3.2 Transfer learning

Transfer learning plays a centre role in the training of our models. We train the single frame baseline from scratch, and early stop when overfitting is detected. At that point, transfer learning from the single frame context is then used as starting point for the training of TimePillars models. In particular, we freeze all the layers from the Pillar Feature Encoder, as well as from the Backbone, except the last six layers of the latter (i.e. the upsampling operations). This proved to be the best configuration, given the implicit accuracy-efficiency trade-off.

5.2.3.3 Class imbalance weighting

Aware of the issues related to the already introduced problem of foreground/background class imbalance, we perform additional measurements to help with the situation.

In particular, we propose the application of class weights, and explore a suitable classifier's bias initialisation.
We start by calculating the frequency of each class in the dataset, and with it the number of samples per class, N_i. This allows getting a quantitative idea about how the distribution of classes looks like.
Numbers reinforce the heavy class imbalance problem, with the background class representing around 99.6% of the dataset, vehicle nearly 0.3 %, and the remaining 0.05% being shared by cyclists and pedestrians. The rest (Animal and inconclusive) result negligibly small.
Knowing the context, the weights for the i-th class, W_i, out of $N_{classes}$, and considering the total length of the dataset, N_T, can be calculated:

$$\mathrm{W}_i = k_i \frac{N_T}{N_i \cdot N_{classes}} \tag{5.3}$$

An extra factor, k_i, allows further class scaling in a flexible manner. Its introduction was considered after analysing confusion matrix weighted results, and observing there still existed margin to increase the intensity of the weighting, with respect to the background predictions.
Alternatively, we consider a classifier's bias initialisation. Its aim is to account for the initial biased scenario that the class imbalance originates. We want the softmax output to reflect the dataset situation. To calculate the bias, the frequency of each class is used as softmax output. By doing this, we are incorporating the class distribution information directly into the model. The higher the frequency of a class, the higher its corresponding softmax output will be. This helps the model assign more importance to the underrepresented classes during training.

$$\mathrm{frequency}_i = \frac{e^{b_i}}{\sum_j e^{b_j}} \tag{5.4}$$

Solving the previous equation brings the bias initialisation values. Setting these correctly speeds up convergence and avoids the network learn the bias in the early iterations.

5.2.4 TimePillars training step

The dataset's structure, as previously described, affects the recurrent network training loop. It is important to consider that the frames in the dataset were acquired over a span of two years and across multiple countries, and thus, there exists no spatial or temporal relationship between them.
Consequently, a specific order cannot be defined to loop the dataset for training the recurrent network and backpropagating every few iterations. However, the past 10 LiDAR scans are available for every frame, although not annotated. Given that these past scans cannot be directly used for training due to lack of annotations, they can be utilised to populate the hidden state with pertinent information.
To accomplish this, we run the network for a fixed number of iterations with scans prior to the core frame, ignoring the network output. Then, we perform a final pass through the network with the core frame as input, followed by computation of the

loss and backpropagation to update the weights. An illustration of this train step can be observed in Figure 5.1.

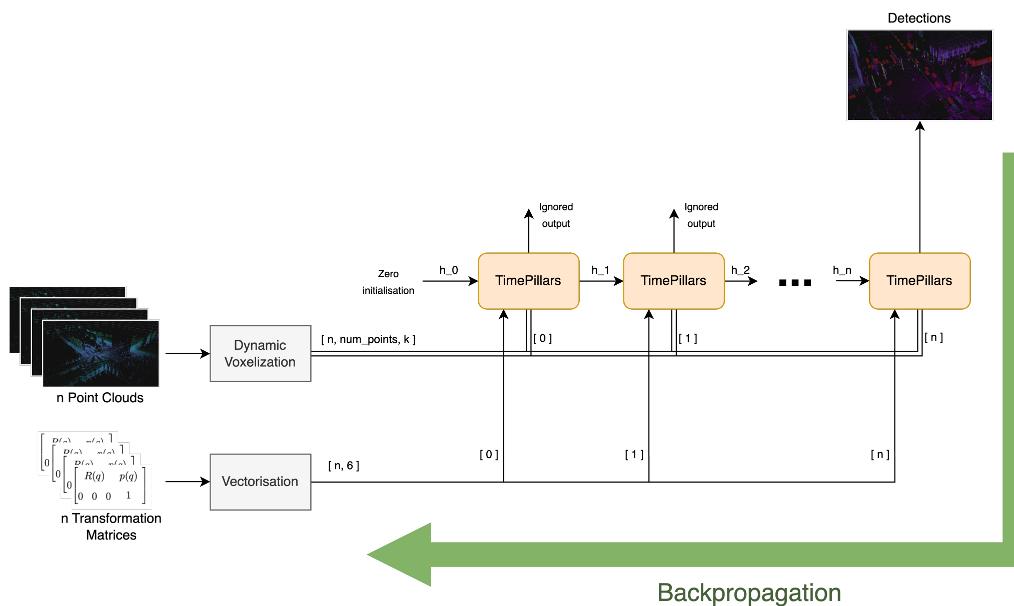


Figure 5.1: Illustration of single training step. The cycle is not batched for simplicity.

With this train cycle we can take advantage of the massive dataset size without repeating data or needing to run at a specific order. The ability to randomise the epoch steps is beneficial in generalisation of the network.

On the other hand this approach results in a long computation time per step given that it runs the network multiple times per weight updates. Furthermore, the memory consumption is quite high due to the increased gradient through the multiple LiDAR scans and network executions.

Memory optimisation then became the big limiting factor in training TimePillars. To address this problem we run the training process in mixed precision. The use of float16 within the network reduces the memory consumption by around half, enabling the use of a higher batch size and faster training.

Additionally, to speed up and ease the training process we transfer the network weights from the single frame PointPillars model trained earlier. The pillar feature encoder is frozen as well as all the downsampling layers of the backbone. This still gives freedom for the network to train output of the backbone, the convGRU and respective transformation module and lastly the detection head. By reducing the amount of weights to be updated the memory usage is reduced and the training stability improved.

For testing and comparison purposes we also trained a model without any transformation inside the network. In this case the transformation is done in the preprocessing of the data. The main goal of this network is to evaluate the performance of the convGRU when transformation of the moving frame is not in question.

5.3 Evaluation metrics

The evaluation metrics used in this work are presented. They aim to establish an objective and quantitative way to examine and compare the performance of the methods proposed.
Precision, recall and F1-Score focus on the classification task, while (mean) Average Precision, standard choice in the field, is used to evaluate the object detection accuracy as a whole.

5.3.1 Precision

Precision measures the proportion of correctly predicted positive instances (true positives, TP) out of the total instances predicted as positive (true positives and false positives, FP). It then focuses on indicating how precise or accurate the model is in predicting positive instances, i.e. how many of the model predictions are actually correct. Its expression is calculated as follows (Eq. 5.5)

$$Precision = \frac{TP}{(TP + FP)} \tag{5.5}$$

The precision score ranges from 0 to 1, with 1 being the score to aim for.

5.3.2 Recall

Conversely, recall measures the proportion of correctly predicted positive instances (true positives) out of the total actual positive instances (true positives and false negatives, FN), see Eq. 5.6:

$$Recall = \frac{TP}{(TP + FN)} \tag{5.6}$$

Its aim is therefore to measure the sensitivity to classification correctness, and the score produced also ranges from 0 to 1, with 1 being the best possible one.

5.3.3 F1-Score

As a way to merge the previous two metrics into one, F1-Score proposes the calculation of the harmonic mean of precision and recall.
Its expression is calculated as:

$$F1Score = \frac{2 \cdot (Precision \cdot Recall)}{(Precision + Recall)} \tag{5.7}$$

providing a balanced assessment, especially when both types of classification errors (false positives and false negatives) need to be taken into account. This metric is therefore particularly useful when the dataset is imbalanced, as it prevents overemphasis on the majority class. As mentioned, the latter is achieved by combining precision and recall, giving equal weight to both. Similarly to the precision and recall metrics, its score ranges from 0 to 1, with 1 being the best achievable value.

5.3.4 Average Precision

The metrics above focus on the classifier's performance, but do not take into account the remaining bounding box variables, which are of regression nature (centre locations, size dimensions and rotation angles).
The average precision metric, conversely, uses the 3D Intersection over Union (Eq. 5.8) between the ground-truth and prediction boxes as a threshold to filter out non accurate predictions.

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}} \tag{5.8}$$

The final score, ranging from 0 to 1, and being the higher the better, comes from the calculation of the area below the precision-recall curve, for each class independently. Note this operation corresponds to the average of picking the (interpolated) precision scores, at retrieved recall values. It formulation can be written as follows:

$$\text{AP} = \frac{1}{N_s} \sum_{k=1}^{N_s} \text{Precision}(k) \cdot \text{Recall}(k) \tag{5.9}$$

where N_s denotes the total number of samples taken, and Recall(k) is the recall value at the k-th retrieved sample.

5.3.4.1 Mean Average Precision (mAP)

In some contexts, AP is calculated for each class and averaged to get its mean value, the mAP. Nevertheless, one can often find work which does not make any distinction in meaning between AP and mAP.
We use the former definition. Therefore, in this paper, the mAP we name corresponds to the mean of the AP obtained for each class i of N_c, and at a certain 3D IoU threshold (0.5 in the case of this work).

$$\text{mAP}_{IoU} = \frac{1}{N_c} \sum_{i=1}^{N_c} \text{AP}_i \tag{5.10}$$

5.4 Results

As mentioned before we train a regular PointPillars Network in our dataset and use it as a benchmark to ensure that the addition of recurrency results in significant performance improvements.
The comparison of the average precision between the single frame benchmark and diverse versions of TimePillars can be seen in Table 5.2. Three alternative methods of performing the motion compensation transformation are shown, as described in the Methods chapter.
The improvements over single frame methods are very noticeable, particularly on smaller sized classes such as Cyclists and Pedestrians. As expected, the added information of multiple scans greatly increases the average precision.
As expected, the network which does not have to transform the hidden state performs the best. This network not only is simpler but is fed perfectly corrected data

and therefore does not need to learn to account for transformation or embedding of this transformation in the hidden state. Despite this, the two methods of transformation inside the network perform quite close to it.
This transformation in the preprocessing that yields the best results is however not our proposal given that it would require that in inference, at every step, we would need to reset the hidden state, transform the previous clouds and feed them all through the entire network. Not only is this inefficient but also defeats the purpose of using a recurrent network.
Additionally, as demonstrated in the table, besides more efficient, the conv-based transformation of the hidden state produces significantly better results than its affine interpolation alternative.
Lastly, we also present an alternative baseline of multi-frame nature, but non-recurrent, based on PointPillars, but employing aggregation of multiple scans, as seen in section 4.2 of the method chapter. Despite the massive computational power and memory required to train this model, we were able to achieve results that improve the single frame PointPillars benchmark. The results reinforce the benefits in accuracy when considering past information in the system, and the need of recurrency to levarage performance and remain efficient.

Method	Motion transf.	Scans	$mAP_{0.5}$	Vehicle	Cyclist†	Pedestrian
PointPillars	-	1	0.520	0.918	0.414	0.228
MF PointPillars*	None‡	3	0.548	0.923	0.443	0.278
TimePillars	None‡	3	**0.624**	**0.927**	**0.547**	0.397
TimePillars	Interpolation	3	0.574	0.905	0.452	0.366
TimePillars	Conv-based	3	0.609	0.909	0.519	**0.400**

Table 5.2: Results of multiple models trained on ZOD frames dataset. All models were evaluated with the same number of scans as they were trained with. (*) Muti-Frame baseline built based on PointPillars and described in the Methods chapter. (†) Cyclist class includes all vulnerable vehicle types including motorcycles, wheelchairs and any human-powered, electric or motorised wheeled object that is used to transport people such as scooters. (‡) By "none", we mean that the network does not include any motion transformation layers but instead the data is motion compensated before being fed into the network.

During the development of the network, the placement of the recurrent module was a crucial point of consideration. If added after the backbone, the training loop would be computationally more expensive since it would repeatedly run the backbone, which is the most computationally intensive section. However, we believe that feeding already processed data could result in better merge of scans.
To verify this hypothesis, we have implemented an additional model similar to the one presented by FS-GRU [5] where the recurrent module is fed the encoded pillars. The results comparing with our suggested approach can be seen in Table 5.3, including our single frame benchmark.
Our solution clearly outperforms in all classes. Most noticeable is the fact that the network with convolutional GRU before the backbone performs similarly to the

single frame baseline. To note that the single frame was trained and fine tuned with a greater care.

Method	GRU location	Scans	$mAP_{0.5}$	Vehicle	Cyclist	Pedestrian
PointPillars	-	1	0.520	**0.918**	0.414	0.228
TimePillars	bef. backbone	3	0.516	0.903	0.413	0.232
TimePillars	aft. backbone†	3	**0.609**	0.909	**0.519**	**0.400**

Table 5.3: Single frame baseline compared to two possible recurrent networks differing in the location of the recurrent module. (†) Like all other TimePillars models used within this project employ a convGRU after the backbone, unless otherwise specified.

We prove the effectiveness of the auxiliary task by training a model without it and comparing the average precision. The results shown in Table 5.4 confirm the performance improvement when using the auxiliary task in training.

Method	Aux. task	Scans	$mAP_{0.5}$	Vehicle	Cyclist	Pedestrian
TimePillars		3	0.588	0.906	0.475	0.382
TimePillars	✓	3	**0.609**	**0.909**	**0.519**	**0.400**

Table 5.4: Results from using or not the auxiliary task for the convolution based transform module.

5.5 Ablation studies

To put the methods' robustness under test, relevant ablation studies are performed.

5.5.1 Longitudinal detection range

As mentioned before, unlike other datasets in the field, ZOD owns novel long range LiDAR data, up to 200 m.

The main results presented in this work, most of them shown previously already, correspond to the utilisation of a longitudinal range of 120 m. Such a value was chosen keeping in mind the clear existing trade-off between long range and accuracy. Note relevant literature mainly focus on ranges below 80 m, e.g. PointPillars used 75 m.

Table 5.5 presents evaluation results for the Single Frame baseline trained at various detection ranges. The values achieved confirm the trade-off (longer range, lower mAP), but also prove the robustness of the method implemented, as the drop is not significant.

Range	$mAP_{0.5}$	Vehicle	Cyclist	Pedestrian
80 m	**0.526**	**0.927**	0.411	**0.239**
120 m	0.520	0.918	**0.414**	0.228
160 m	0.496	0.904	0.359	0.226
200 m	0.495	0.890	0.381	0.215

Table 5.5: Longitudinal detection range ablation study. We compare the system's performance for growing distances from the EGO vehicle.

5.5.2 Long-term memory

Stability for long sequences is crucial for recurrent networks in inference. It enables the use of the model continuously in real world applications without the need to reset the hidden state. To this end we evaluate multiple models and how they behave for longer sequences.
Performing inference with 5 LiDAR scans on a network trained for 3 produces a drop in accuracy. To test if a higher number of sweeps could produce more stable results, we have also trained an additional model with 10 LiDAR scans. As shown in Table 5.6, performing inference on that model with, instead, 11 scans produces a significant reduction in average precision.

Training Scans	Inference Scans	$mAP_{0.5}$	Vehicle	Cyclist	Pedestrian
3	3	0.624	0.927	0.547	0.397
3	5	0.304	0.837	0.043	0.033
10	10	0.611	0.929	0.523	0.382
10	11	0.552	0.925	0.403	0.329

Table 5.6: Average precision evaluated for different number of LiDAR scans. These models were trained without any motion compensation in the model itself, meaning that the data fed into the network both in inference and training was already motion compensated.

We can then assert that our trained models do not generalise well for longer sequences than what they were trained for. Further analysis show that the longer we train, despite improving the performance, the worse the performance gets at higher inference number of scans. In other words, the model seems to overfit to this number. Several possible solutions were theorised, mostly changing the way the training cycle is done. However, due to data structure limitations and time, they were not experimented on. A further analysis of possible improvements in this area are shown later in the Future Work section.
To avoid this decrease in performance as more LiDAR scans are fed into the network, in real life applications one could introduce a circular buffer of size equal to the number of scans used in training. Every time a new point cloud is acquired we feed it through the pillar feature encoder and backbone and then add this encoded feature space to the buffer (removing the oldest one there). Then we execute the convolutional GRU for each of the encoded features in the buffer, essentially building

up the hidden state to with the right amount of past scans, and end by computing the detection head.
The trade off of this approach is a slight increase in computation time due to the multiple convGRU executions and the increase in memory usage due to the circular buffer. Despite these setbacks, it ensures a significant improvement in performance and still prevents the need to feed past LiDAR scans through the pillar feature encoder and backbone, which consists of the most costly operations. The use of recurrency would then still be justified.

6

Conclusions

6.1 Discussion

In view of the results, the validity of the work can be confirmed. The outcome fulfils the objectives, and answers the set of initial research questions.
First of all, the method followed leads to the successful realisation of a suitable model for the task of 3D LiDAR object detection.
An important conclusion follows: taking past sensor data into account proves to be superior than just exploiting information at the present. The access to previous information about the driving environment faces the sparsity nature of LiDAR point clouds, and leads to more accurate predictions. Furthermore, the work shows the adequacy of recurrent networks as a mean to achieve the latter. In contrast to point cloud aggregation methods, which consider the creation of denser data representations by heavy processing, it turns out endowing the system with memory brings a more robust solution.
Our main proposal, TimePillars, materialises a way to solve the recurrent problem. Unlike relevant recurrent literature, extensive experimentation suggests the placement of convGRU in the pipeline after the backbone block works best. In addition, our method establishes a concrete way to compensate for vehicle motion across scans, and at feature level. Moreover, the work shows the direct benefits of achieving it via an auxiliary task specifically-designed for it.
Evaluation presents a performance growth with respect to the model taken as baseline (PointPillars). The latter, known by its efficiency, is just augmented with three extra convolutional layers during inference. This modification proves that basic NN building blocks are enough to achieve significant results, and guarantees the existing efficiency and hardware-integration specifications are met.
Finally, to the best of our knowledge, the work acts as a first baseline of results for the 3D object detection task on the newly introduced Zenseact Open Dataset. The dataset exhibits a level of dimensional and contextual diversity that exceeds that of any other currently released datasets. As such, we can assert with confidence that the network's performance and generalizability extend to real-world applications beyond the dataset employed.

6.2 Future work

The main question that remains to be answered is how to train and tune the network architecture to achieve a stable execution when exposed to long sequences. As

mentioned in the ablation studies, our trained model overfits to the number of LiDAR scans provided in training. When executed for longer sequences than the ones trained, the average precision drops significantly. This could be an indication of the GRU not fully learning to forget.
We believe that a different training strategy could fix this problem, such as a stateful approach to the training cycle where, for a long sequence, we do backpropagation and weight updates every few scans but not resetting the hidden state. This could hypothetically force the network to learn to forget irrelevant or outdated data.
Another potential solution could be training or fine-tuning the model on significantly larger sequences per training step, which could prove to be challenging memory-wise.
It is important to remark that the latter, however, does not prevent the work from being applied in a real product. Proven the benefits of using recent past frames, this philosophy can perfectly be kept at inference. Furthermore, previously-computed backbone outputs, input to the recurrent module, can be stored and updated across time to avoid the need of recomputing past frames.
We, therefore, leave open a hypothetical future model acceleration and hardware implementation that would quantitatively assess the efficiency of the proposed solution, and consider it for real-life application.

Bibliography

[1] Zenseact's mission, people at heart webpage. https://zenseact.com/. Accessed: 31-05-2023.

[2] Simegnew Yihunie Alaba and John E. Ball. A Survey on Deep-Learning-Based LiDAR 3D Object Detection for Autonomous Driving. *Sensors*, 22(24):9577, 12 2022.

[3] Mina Alibeigi, William Ljungbergh, Adam Tonderski, Georg Hess, Adam Lilja, Carl Lindström, Daria Motorniuk, Junsheng Fu, Jenny Widahl, Petersson Zenseact, and Sweden Gothenburg. Zenseact Open Dataset: A large-scale and diverse multimodal dataset for autonomous driving. 5 2023.

[4] Sergio Casas, Wenjie Luo, and Raquel Urtasun. IntentNet: Learning to Predict Intention from Raw Sensor Data. 1 2018.

[5] Zhikai Chen, Yafei Wang, Xulei Liu, and Xinchang Wang. FS-GRU: Continuous Perception and Prediction with inter Frame Feature Sharing. pages 517–522, 11 2022.

[6] Lue Fan, Yuxue Yang, Feng Wang, Naiyan Wang, and Zhaoxiang Zhang. Super Sparse 3D Object Detection. 1 2023.

[7] Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R. Martin, and Shi-Min Hu. PCT: Point cloud transformer. 12 2020.

[8] Sven Ove Hansson, Matts Åke Belin, and Björn Lundgren. Self-Driving Vehicles—an Ethical Overview. *Philosophy and Technology*, 34(4):1383–1408, 12 2021.

[9] Rui Huang, Wanyue Zhang, Abhijit Kundu, Caroline Pantofaru, David A Ross, Thomas Funkhouser, and Alireza Fathi. An LSTM Approach to Temporal 3D Object Detection in LiDAR Point Clouds. 7 2020.

[10] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. PointPillars: Fast Encoders for Object Detection from Point Clouds. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June:12689–12697, 12 2018.

[11] Patrick Lin. Why Ethics Matters for Autonomous Cars. In *Autonomes Fahren*, pages 69–85. Springer Berlin Heidelberg, 2015.

[12] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal Loss for Dense Object Detection. 8 2017.

[13] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. 12 2015.

[14] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. 11 2017.

[15] Wenjie Luo, Bin Yang, and Raquel Urtasun. Fast and Furious: Real Time End-to-End 3D Detection, Tracking and Motion Forecasting with a Single Convolutional Net. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3569–3577, 12 2018.

[16] Jiageng Mao, Yujing Xue, Minzhe Niu, Haoyue Bai, Jiashi Feng, Xiaodan Liang, Hang Xu, and Chunjing Xu. Voxel Transformer for 3D Object Detection. 9 2021.

[17] Scott Mccrae and Avideh Zakhor. 3D OBJECT DETECTION FOR AUTONOMOUS DRIVING USING TEMPORAL LIDAR DATA. Technical report.

[18] Ishan Misra, Rohit Girdhar, and Armand Joulin. An End-to-End Transformer Model for 3D Object Detection. 9 2021.

[19] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January:77–85, 12 2016.

[20] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. *Advances in Neural Information Processing Systems*, 2017-December:5100–5109, 6 2017.

[21] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. PointRCNN: 3D Object Proposal Generation and Detection from Point Cloud. 12 2018.

[22] Xu Wang, Yi Jin, Yigang Cen, Tao Wang, Bowen Tang, and Yidong Li. LighTN: Light-weight Transformer Network for Performance-overhead Tradeoff in Point Cloud Downsampling. 2 2022.

[23] Qian Xie, Yu-Kun Lai, Jing Wu, Zhoutao Wang, Yiming Zhang, Kai Xu, and Jun Wang. MLCVNet: Multi-Level Context VoteNet for 3D Object Detection. 4 2020.

[24] Yan Yan, Yuxing Mao, and Bo Li. SECOND: Sparsely Embedded Convolutional Detection. *Sensors 2018, Vol. 18, Page 3337*, 18(10):3337, 10 2018.

[25] Bin Yang, Wenjie Luo, and Raquel Urtasun. PIXOR: Real-time 3D Object Detection from Point Clouds. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 7652–7660, 2 2019.

[26] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020.

[27] Cheng Zhang, Haocheng Wan, Xinyi Shen, and Zizhao Wu. PVT: Point-Voxel Transformer for Point Cloud Learning. 8 2021.

[28] Hengshuang Zhao, Li Jiang, Jiaya Jia, Philip Torr, and Vladlen Koltun. Point Transformer. Technical report.

[29] Yin Zhou, Pei Sun, Yu Zhang, Dragomir Anguelov, Jiyang Gao, Tom Ouyang, James Guo, Jiquan Ngiam, and Vijay Vasudevan. End-to-End Multi-View Fusion for 3D Object Detection in LiDAR Point Clouds. 10 2019.

[30] Yin Zhou and Oncel Tuzel. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 11 2017.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY