POLITECNICO DI TORINO

Master's Degree in Electronic Systems

Master's Degree Thesis

Hardware and firmware tuning for point cloud object detection in embedded systems

Supervisors

Prof. Luciano LAVAGNO

Prof. Fernando LOPES

Candidate

Francesco DUPRÉ

July 2024

Summary

Object detection in point clouds is a central aspect of many robotics applications such as autonomous driving. Real-time technologies require very high speed devices and demand low complexity algorithms, at the expense of accuracy.
In this study we consider the trade-off between inference time and accuracy of an object detection model. In particular, our purpose is to answer the following question: how much can we reduce the computation time of said algorithm maintaining a sufficient accuracy and keep satisfactory performance?
To solve this problem we exploit the PointPillars algorithm, an encoder that utilizes PointNets to learn a representation of point clouds organized in vertical columns (pillars), which outperforms many other methods with respect to both speed and accuracy by a large margin.
Tuning is applied to some parameters of this model, such as the number of filters of the feature encoder and the number of layers of the backbone, without changing its global structure. Through training and testing performed with the KITTI benchmark, we obtain the trends of the accuracy versus time relations along the applied modifications.
Studying these tendency functions, we extrapolate the best solution which lowers the inference time without significantly reducing the performance. This solution consists in the reduction in the amount of layers in the backbone, and in the number of up-sample filters at its output.
On this basis, after building the environment to work on Nvidia Jetson Nano, an embedded system that contains a GPU for high-performance computing tasks, future work could concentrate on applying the improved model to this machine, with the aim of analysing its power efficiency and performance.

Acknowledgements

I would like to express my deepest appreciation to professor Fernando Lopes from ISEC (Instituto Superior de Ingeniería de Coímbra) who perfectly supervised the project with balance and excellent management skills. I also could not have undertaken this journey without the generous support of professor Luis Cruz from UC (Universidade de Coimbra) Pólo II, and the telecommunications laboratory team, Nuno Martins and Pedro Daniel Rocha, who provided knowledge and expertise to help me solve the hardest problems with clever solutions. Additionally I am deeply indebted to the Politecnico di Torino teaching staff, who has accompanied me for five years in this educational growth. In particular, this endeavor would not have been possible without professor Luciano Lavagno, who kindly accepted to tutor this project.
I am also grateful to my classmates and cohort members, and to my dearest and oldest friends, who always supported me in this path, sharing with me late-night feedback sessions and all the happy and the sad moments, and always giving me moral support. Special thanks should go to Irene, who believed in me and motivated me even when I could not believe in myself.
Lastly, I would be remiss in not mentioning my family, especially my parents who financed my studies making all of this possible, and my brother and sister. Their belief in me has kept my spirits and motivation high during this process.

Driving to the forefront of progress,
where technology transforms vision into reality.

Table of Contents

List of Tables

List of Figures

Acronyms

IEEE
: Institute of Electrical and Electronics Engineers

AP
: Average Precision

BEV
: Bird Eye View

CNN
: Convolutional Neural Network

GPU
: Graphic Processing Unit

SSD
: Single Shot Detector

VFE
: Virtual Feature Encoder

LiDAR
: Light Detection And Ranging

Chapter 1

Field review: object detection in point clouds, software and hardware tools, embedded systems

Over the past few years object detection has seen remarkable development driven by a rise in demand for solid perception systems in various applications, recently including autonomous driving.
Object detection techniques in point clouds stands as a pivotal area within computer vision, as they offer a unique perspective to better understand spatial relationships and contextual information in the environment. Point cloud data captures the geometric structure of scenes with extremely fine detail, making it well-suited for tasks such as obstacle detection, object recognition, and scene understanding. By analyzing the distribution of points and their attributes, object detection algorithms can accurately identify and localize objects within complex 3D scenes [22] .
Diverse software and hardware tools have been developed during all these years for allowing easy development of efficient object detection systems. Advanced software frameworks such as TensorFlow [9], PyTorch [10], and Open3D [11]provide rich ecosystems for prototyping, training, and deploying deep learning models tailored to point cloud data.
Furthermore, the integration of object detection algorithms into embedded systems unlocks new possibilities for edge computing applications. Embedded systems are specialized computing systems designed to perform specific tasks or functions within larger devices. Typically, they are tailored to one application and optimized towards performance, reliability, and low power consumption. Embedded platforms,

characterized by their limited computational resources and power constraints, require tailored solutions in order to enable real-time processing of point cloud data. Leveraging specialized hardware accelerators such as GPUs, alongside optimized software implementations, it is possible to achieve high-performance object detection on resource-constrained devices.

1.1 Object detection in point clouds

In Figure 1.1 it is shown from a BEV (Bird's Eye View) perspective the result of a real-time detection in point clouds. The targets (cars, pedestrians and cyclists) are surrounded by the bounding boxes and some additional information is added, like their speed.

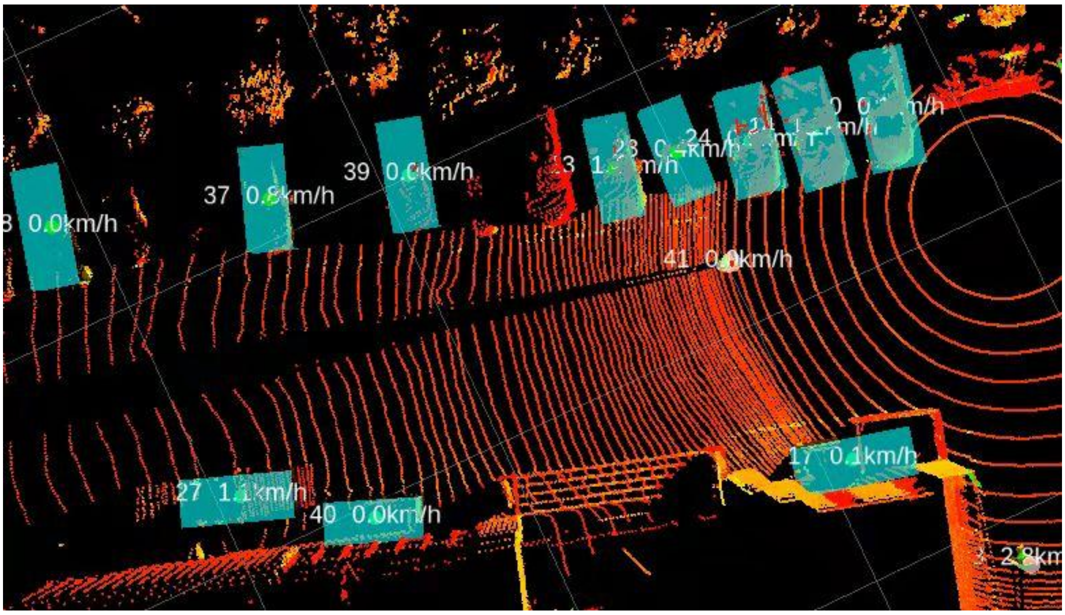

Figure 1.1: CUDA-PointPillars BEV image with bounding boxes

Object detection in point clouds is achieved by identifying and classifying objects in 3D space, based on the data points captured by sensor devices such as LiDAR [31]. Unlike traditional 2D image-based detection, point cloud data provides a rich view of the three-dimensional space around the environment, allowing for more accurate and reliable object detection. This proves to be particularly useful in applications such as autonomous driving, where understanding the spatial arrangement of objects is critical for navigation and safety.
Several key algorithms and approaches have been developed for object detection in point clouds. One popular method is VoxelNet [8], which segmentates the

point cloud into a voxel grid and then processes each voxel for features extraction. This approach leverages 3D convolutional neural networks (3D CNNs) to learn representations from the voxelized point cloud, thus allowing accurate object detection. Another widely used algorithm is PointPillars [1][2][3][4][5][6][27], which converts point cloud data into a pseudo-image and applies 2D CNNs for detection. This method simplifies the computational complexity and therefore real-time applications.

1.2 Software and Hardware tools

Designing and deploying object detection systems for point clouds requires robust software and hardware tools. Software frameworks such as TensorFlow [9] and PyTorch [10] provide powerful tools for building, training, and evaluating deep learning models. These frameworks offer pre-built modules and libraries that simplify the implementation of complex neural network architectures. Additionally, Open3D [11], an open-source library designed for 3D data processing, provides tools for point cloud manipulation, visualization, and integration with deep learning frameworks.

On the hardware side, platform selection is crucial to achieve the required performance. High-performance GPUs from NVIDIA, such as the Jetson Nano [17][18], offer parallel processing capabilities that significantly accelerate the training and inference of deep learning models. These chips are optimized to run complex neural networks and can handle the large computational demands of point cloud processing. Moreover, advancements in specialized hardware accelerators, such as Tensor Processing Units (TPUs) and Field Programmable Gate Arrays (FPGAs) [12], provide additional options for optimizing performance and power efficiency.

1.3 Embedded systems

Embedded systems play a vital role in bringing object detection capabilities to edge devices. These systems are designed to perform specific tasks with high efficiency and reliability. In the context of autonomous driving, embedded systems enable real-time processing of sensor data, allowing vehicles to make quick and informed decisions. Unlike general-purpose computing systems, embedded systems are optimized for low power consumption and small form factors, making them ideal for deployment in resource-constrained environments.

To achieve high-performance object detection on embedded systems, several strategies can be employed. One approach is to use hardware accelerators such as GPUs to offload computationally intensive tasks from the main processor [13]. By parallelizing the processing of point cloud data, GPUs can significantly reduce the time required for object detection. Additionally, optimizing software implementations for the specific architecture of the embedded system can further enhance performance. Techniques such as quantization [14], which reduces the precision of the model weights, and pruning, which removes unnecessary connections in the neural network, can be used to reduce the computational load without sacrificing accuracy.

Chapter 2

Identify a combination of a suitable object detection algorithm and an embedded platform

2.1 Object Detection Algorithms for Point Clouds

The following are some of the most performing algorithms for point clouds:

1. PointPillars [1][2][3][4][5][6][27]: this first method consists in a pillar-based representation, in which the point clouds are divided into pillars and each pillar's feature is encoded using a sparse 2D grid. It employs a two-stage architecture with a sparse convolutional backbone followed by region proposal and classification stages, achieving high efficiency and reducing the computational cost. Moreover, it is specifically designed for object detection in 3D point cloud data captured by LiDAR [31] sensors.
   In Figure 2.1 is showed the complete PointPillars network, starting from the point cloud data and ending with the final predictions. All the chain steps are present with a particular focus on the feature encoder and the backbone.

2. Frustum PointNet [15]: this algorithm selects frustum regions from the point cloud corresponding to 2D bounding boxes detected in an image. It utilizes the PointNet architecture for feature extraction from the frustum point clouds. Its main objective is to integrate 2D image data with 3D point cloud data to improve object detection accuracy. In summary, it trains the model in an end-to-end manner to jointly optimize 2D detection and 3D localization tasks.

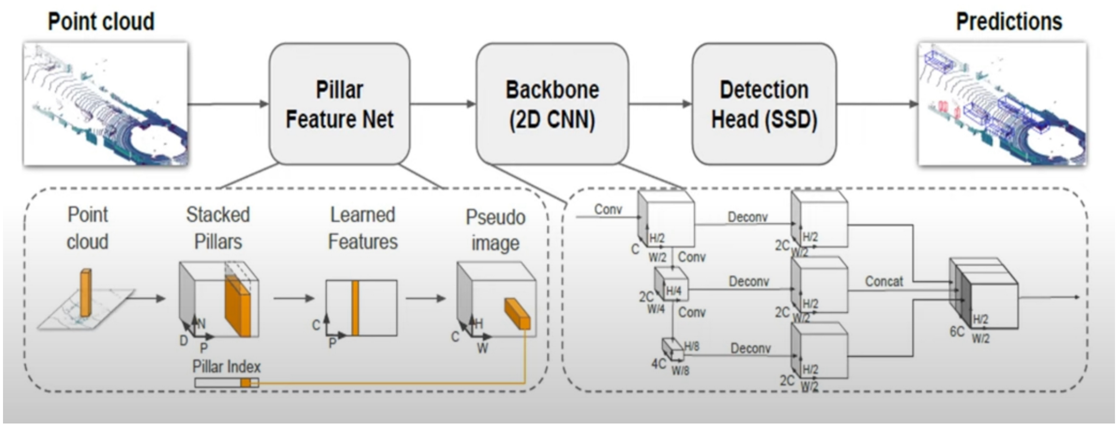

Figure 2.1: PointPillars network

3. 3D YOLO (You Only Look Once) [16]: this last solution performs object detection directly on the entire 3D point cloud in a single pass. It relays on a voxel-based representation, which consists in the division of point clouds into voxels followed by the prediction of object bounding boxes, confidence scores, and class labels for each voxel. It is a simple and efficient approach to object detection in point clouds, suitable for real-time applications, due to the trade-off between inference speed and detection accuracy, balanced adjusting model complexity and voxel resolution.

2.2 Embedded Platforms for Object Detection with Point Clouds

Here follow some of the most suitable platforms for object detection in point clouds:

1. Nvidia Jetson Nano [17][18]: this machine features a CUDA-enabled GPU for high-performance computing tasks, including deep learning inference. It is designed for energy-efficient operation, making it suitable for embedded applications with power constraints, an it offers a small and lightweight design ideal for deployment in edge computing devices. Furthermore, its compatibility with popular deep learning frameworks such as TensorFlow [9] and PyTorch [10], enables easy deployment of object detection models.
Figure 2.2 shows the aspect of a Nvidia Jetson Nano [17][18] device.

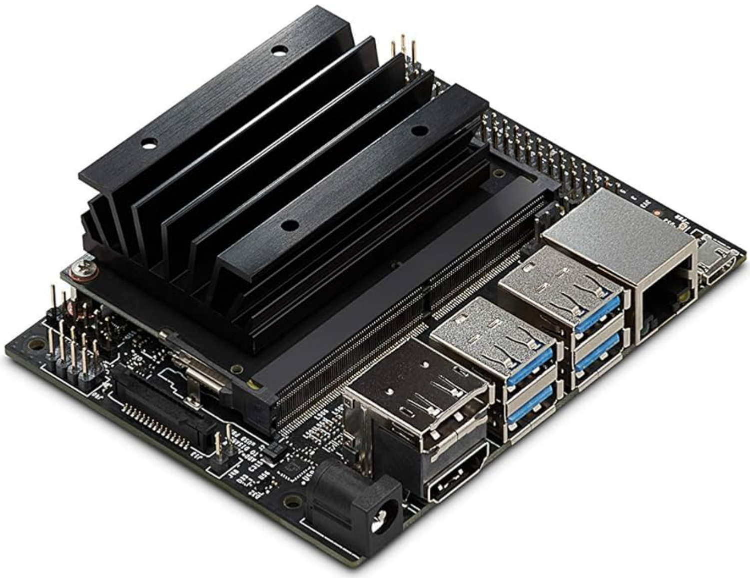

Figure 2.2: Nvidia Jetson Nano

2. Intel Neural Compute Stick (NCS) [19]: it is a system that plugs directly into a USB port and provides hardware acceleration for deep learning inference. Since it provides a unified interface for deploying and running deep learning models on the NCS, it simplifies integration with embedded systems. Its main advantage is the low latency, in fact it offers fast inference speeds with minimal delay, suitable for real-time object detection applications.

3. Raspberry Pi with Coral Edge TPU [20][21]: finally, this method integrates Google's Coral Edge TPU accelerator for high-performance deep learning inference, and it works seamlessly with Raspberry Pi single-board computers, offering a cost-effective solution for embedded object detection. Its compact size and low power consumption make it suitable for deployment in small and power-constrained devices, such as IoT sensors and edge devices.

These examples highlight some of the key object detection algorithms and embedded platforms used in the field of point cloud object detection, each with its own unique characteristics and suitability for different applications and deployment scenarios.
In our case, the most suitable combination would be the use of a Nvidia Jetson Nano [17][18] hardware along with the PointPillars [1][2][3][4][5][6][27] algorithm.

2.3 PointPillars

PointPillars [1][2][3][4][5][6][27] is a method for 3D detection:

- 3D object detection - recognition and determination of 3D information
- 2D convolutional layer - filter or kernel in a conv2D layer that slides over the 2D input data performing an element multiplication

The PointPillars network has a learnable encoder that uses PointNet to learn a representation of point clouds organized in pillars (vertical columns):

- PointNet - unified architecture that learns both global and local point features providing simple, efficient and effective approach for a number of 3D recognition tasks
- Point clouds - a huge collection of tiny individual points plotted in 3D space made up of a multitude of points captured using a 3D laser sensor

The network then runs a 2D convolutional neural network to produce network predictions, decode the predictions and generate 3D bounding boxes for different object classes such as car, pedestrian and cyclist.
In Figure 2.3 is shown an overview of the PointPillars algorithm, step-by-step, divided by color in the main layers. In light blue we have the input features and indices, in blue the feature encoder, in yellow the scatter layer, in purple the backbone and in red the dense head.

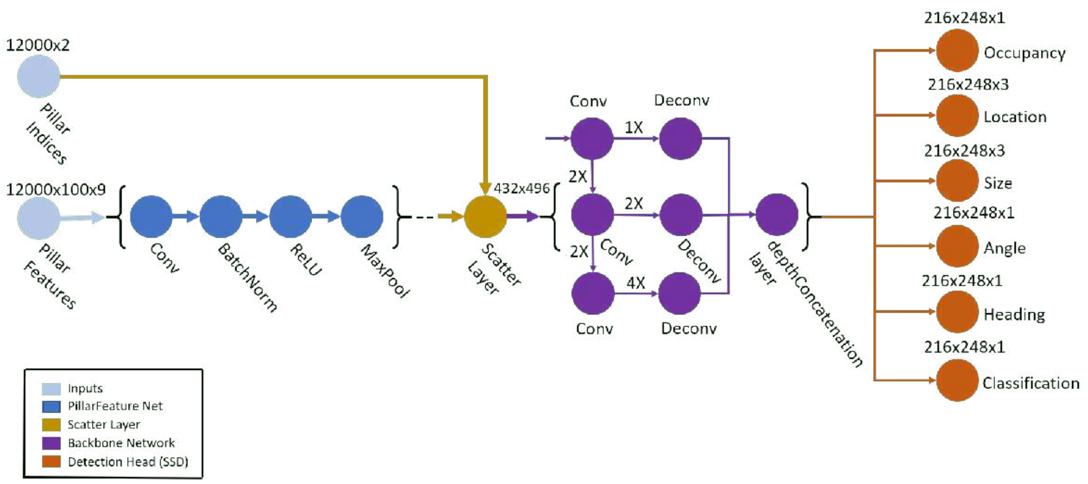

Figure 2.3: PointPillars network overview

2.3.1 Pillar feature net

The feature net is constituted by the Virtual Feature Encoder (VFE) that works following a series of steps:

1. Point clouds are generated by means of a Light Detection And Ranging (LiDAR) sensor [31], and they constitute datasets that represent objects or space using a three coordinates system (x, y, z).

2. The dataset is then divided into grids in the (x, y) plane, obtaining a set of pillars. We denote by l a 4-dimensional point in a point cloud with coordinates (x, y, z) and reflectance r.

3. Each point is then converted into a 9-dimensional vector, containing:

   - x_c, y_c, z_c: distance to the arithmetic mean of all points in the pillar.
   - x_p, y_p: offset from the pillar center in the (x, y) coordinates.

   The new point will be $D = [x, y, z, r, x_c, y_c, z_c, x_p, y_p]$.

4. The set of pillars will be mostly empty due to sparsity of the point cloud, and the non-empty pillars will in general have few points in them. For this reason, to exploit sparsity, a limit on P and N is imposed, where:

   - P is the number of non-empty pillars.
   - N is the number of points per pillar.

   A dense tensor of size (D, P, N) is obtained.
   If a sample of a pillar holds too much information to fit in this tensor, the data is sampled. On the other hand if the sample has too little data to populate the tensor, zero padding is applied.

5. By means of a simplified version of PointNet, a linear layer (1x1 convolution across the tensor) is applied to each point, followed by BatchNorm and ReLU to obtain high-level freatures of dimension (C, P, N). This is followed by a max pool operation that converts it to a (C, P) dimensional tensor.

2.3.2 Scatter layer

Once encoded, the features are scattered back to the original pillar locations to create a pseudo-image of size (C, H, W) where H and W indicate the height and width of the canvas.

2.3.3 Backbone network

The backbone is constituted by sequential 2D convolutional layers (2D CNN) to learn features from the transformed input. The input is the feature map generated by the feature encoder and the scatter layer. The backbone network is divided in three fully convolutional blocks:

- The first layer of each block down-samples the feature maps by half by means of convolution of stride 2, followed by a sequence of convolutions of stride 1.
- After each convolution layer, BatchNorm and ReLU operations are applied.
- The output of every block is up-sampled to a fixed size via deconvolution.
- The final output features are concatenated to obtain the high-resolution feature map.

2.3.4 Detection head

A Single Shot Detector (SSD) setup is used to perform 3D object detection:

- SSD network's objective is to generate bounding boxes on the features coming from the backbone layer.
- The task of object localisation is done in a single forward pass of the network using a multi-box for bounding box regression technique.
- The detector also classifies the detected objects by means of the class anchors generator.
- Non-maximum suppression is used to filter out noisy predictions.

Chapter 3

Performance evaluation of the selected algorithm in a desktop PC without resource limitations

3.1 Experimentation setup

To establish a base reference on our performance benchmark, we tested the original PointPillars method in a non-limiting processing capabilities environment, using a machine with the following characteristics:

Operating System: Kubuntu 22.04
KDE Plasma Version: 5.24.7
KDE Frameworks Version: 5.92.0
Qt Version: 5.15.3
Kernel Version: 6.5.0-28-generic (64-bit)
Processors: 24 × AMD Ryzen 9 5900X 12-Core Processor
Memory: 62,7 GiB of RAM
Graphics: NVIDIA RTX 3090 24GB [26]
nvcc: NVIDIA (R) Cuda compiler driver
Driver Version: 550.90.07
CUDA Version: 12.4

For what concerns the dataset, we use the KITTI object detection benchmark [23][29] dataset, which contains samples that have both LiDAR [31] point clouds

and images.
The KITTI benchmark [23][29] is divided in two macro categories, training set and testing set. The training set is used to train models, while the testing set is used to test them after the training. This last portion of data also contains the validation set, which is used for validations along the training.
Moreover, the objects in the KITTI dataset [23][29] are also categorized as easy, moderate and hard data difficulties. This separation is made on the number of targets in the data, their position with respect to the source and their tendency to be spotted.

3.2 Model configuration file

To better understand the PointPillars method, in Listing 3.1 we focus on the model part in the configuration file. This portion of the file describes every step of the Pointpillars network. We can see the number of filters in the VFE which is then used as number of BEV features in the scatter layer, and then as input filters of the backbone. In the latter, it is particularly relevant the number of layers, since it will characterize all the backbone structure.

Listing 3.1: PointPillar.yaml

```
...
MODEL:
    NAME: PointPillar

    VFE:
        NAME: PillarVFE
        WITH_DISTANCE: False
        USE_ABSLOTE_XYZ: True
        USE_NORM: True
        NUM_FILTERS: [64]

    MAP_TO_BEV:
        NAME: PointPillarScatter
        NUM_BEV_FEATURES: 64

    BACKBONE_2D:
        NAME: BaseBEVBackbone
        LAYER_NUMS: [3, 5, 5]
        LAYER_STRIDES: [2, 2, 2]
        NUM_FILTERS: [64, 128, 256]
        UPSAMPLE_STRIDES: [1, 2, 4]
        NUM_UPSAMPLE_FILTERS: [128, 128, 128]

    DENSE_HEAD:
```

```
        NAME: AnchorHeadSingle
        CLASS_AGNOSTIC: False
...
```

The full configuration file can be found in Appendix A.

3.3 Results

After building the environment on the "limitless" resources machine, following the OpenPCDet tutorial [24], we are able to train and test the model performance. The training file *"train.py"* can be found in Appendix B.
We will consider only the BEV results, since it is the encoding technique used in the model that we are exploiting.
Figure 3.1, Figure 3.2 and Figure 3.3 show the accuracy behavior along the training validations for cars respectively for easy, moderate and hard data difficulties. Figure 3.4, Figure 3.5 and Figure 3.6 show the accuracy behavior along the training validations for cyclists respectively for easy, moderate and hard data difficulties. Figure 3.7, Figure 3.8 and Figure 3.9 show the accuracy behavior along the training validations for cars respectively for easy, moderate and hard data difficulties. These graphs obtained by means of the Tensorboard [9][25] visualization tool, show the accuracy (y-axis) in percentage, changing along the number of epochs (x-axis), starting from epoch 50 of 80.

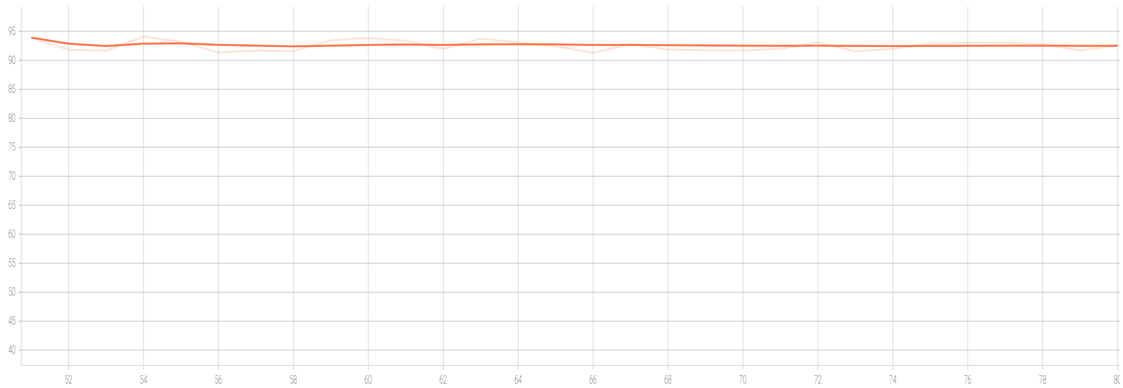

Figure 3.1: Car accuracy [%] for easy data difficulty along the epochs

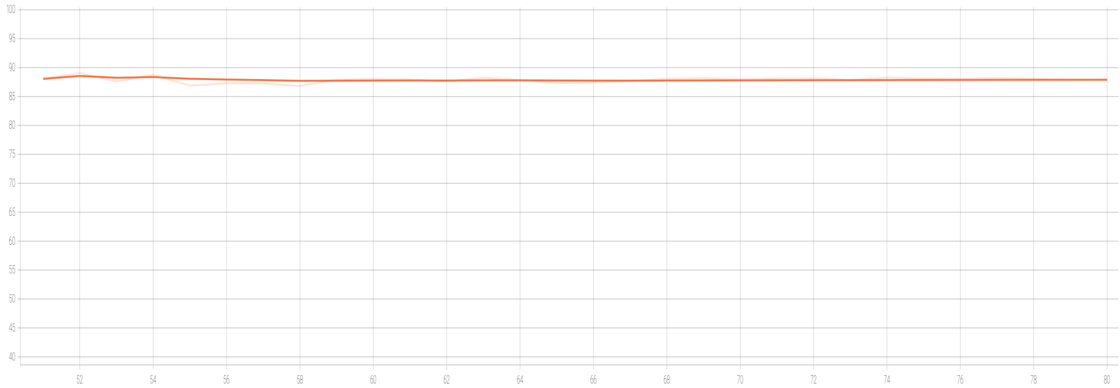

Figure 3.2: Car accuracy [%] for moderate data difficulty along the epochs

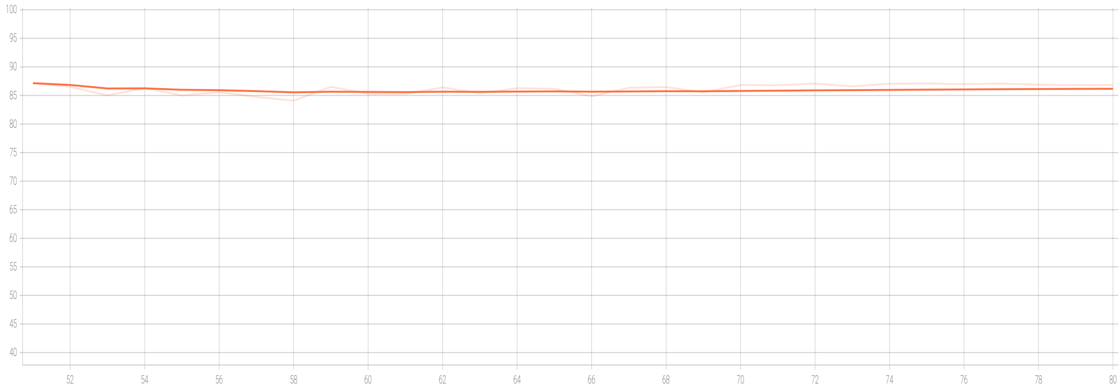

Figure 3.3: Car accuracy [%] for hard data difficulty along the epochs

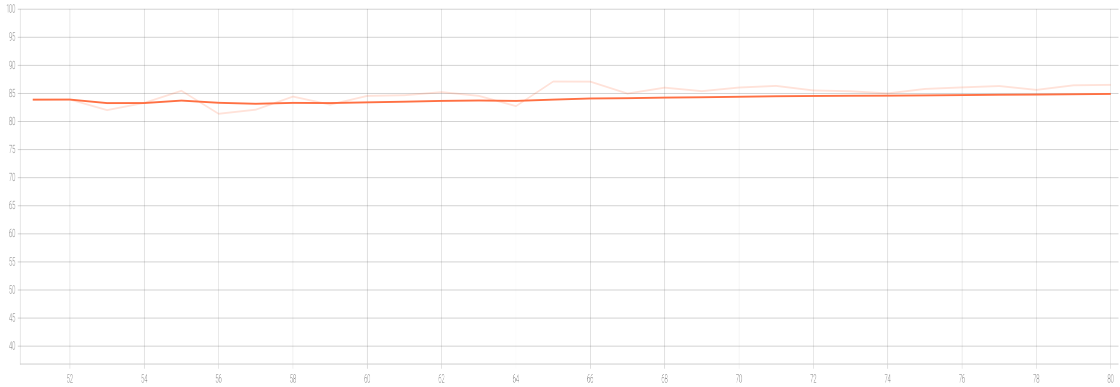

Figure 3.4: Cyclist accuracy [%] for easy data difficulty along the epochs

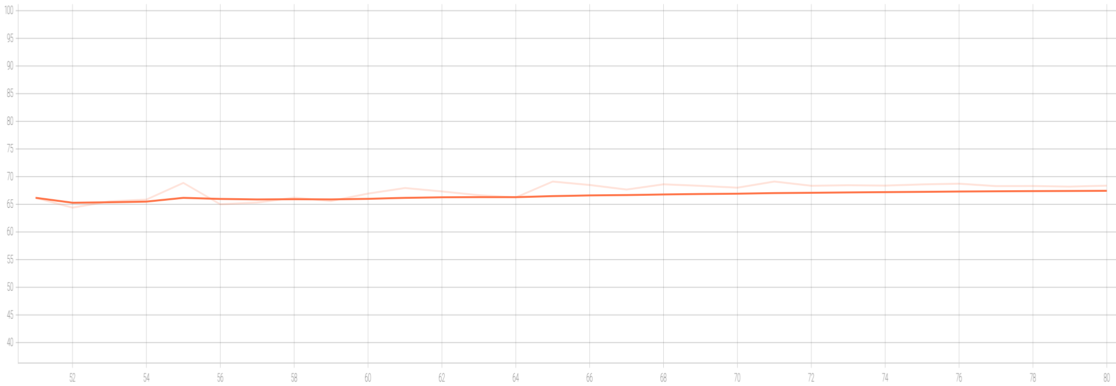

Figure 3.5: Cyclist accuracy [%] for moderate data difficulty along the epochs

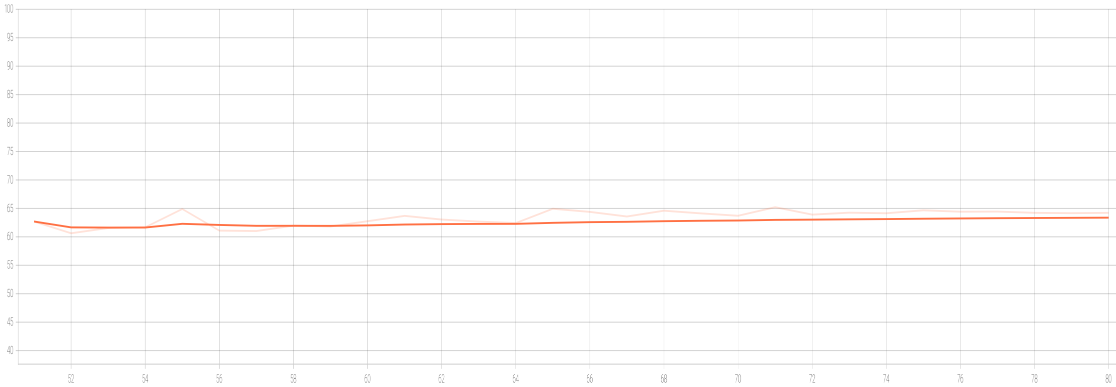

Figure 3.6: Cyclist accuracy [%] for hard data difficulty along the epochs

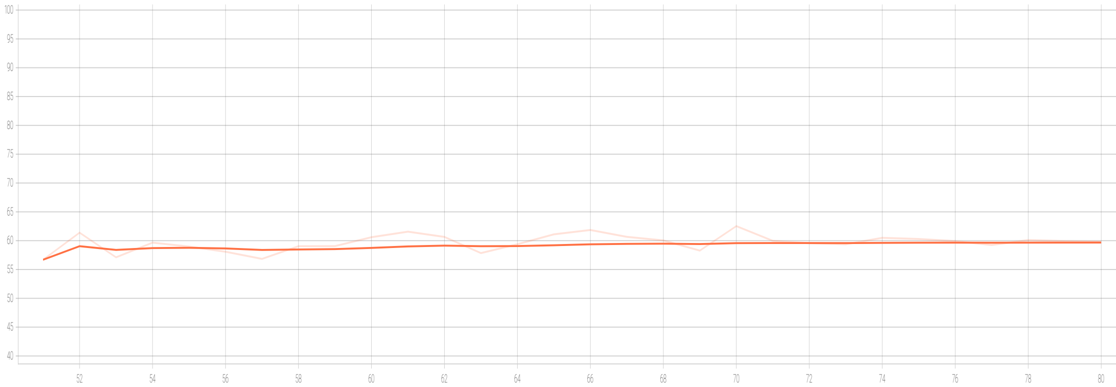

Figure 3.7: Pedestrian accuracy [%] for easy data difficulty along the epochs

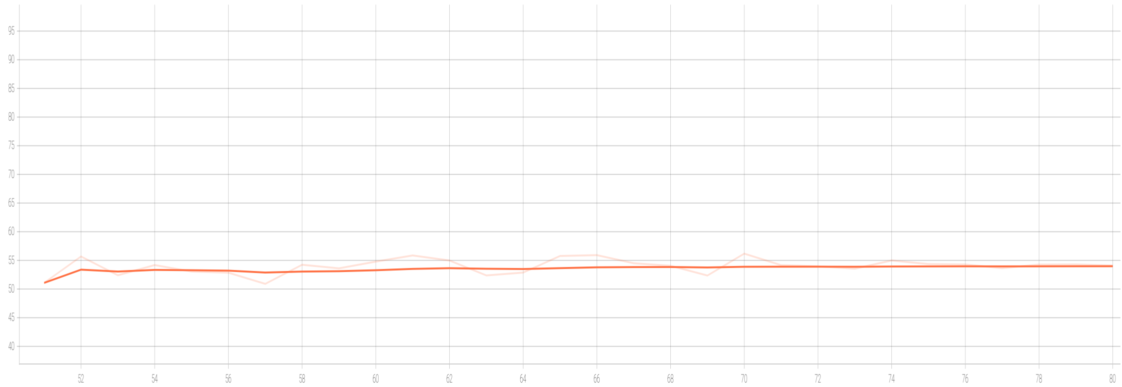

Figure 3.8: Pedestrian accuracy [%] for moderate data difficulty along the epochs

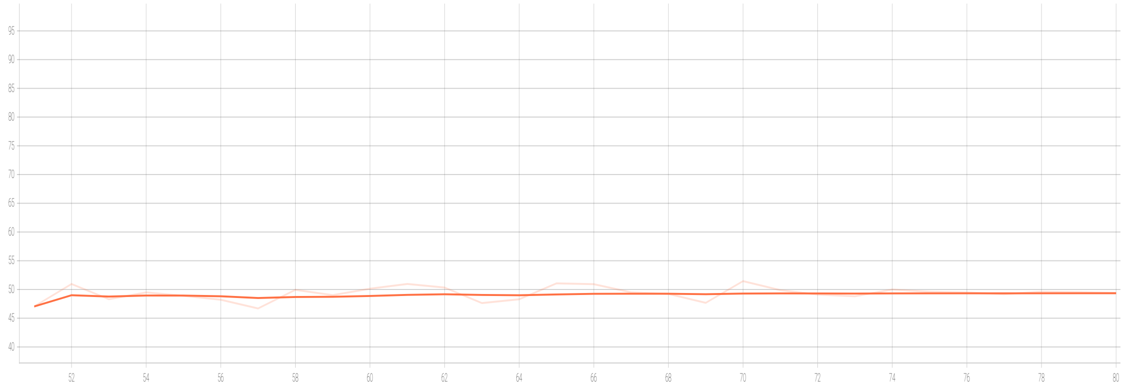

Figure 3.9: Pedestrian accuracy [%] for hard data difficulty along the epochs

These results show clearly how for every subject (car, cyclist, pedestrian) the accuracy decreases from easy difficulty data, to moderate difficulty data, to finally reach its minimum with hard difficulty data. One more consideration: since pedestrians are the hardest to spot, the model accuracy for them is lower, while it is a little better for cyclists and it is the best for cars.

3.3.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 3.1.

	easy	hard	moderate
car	92.63	86.81	87.95
cyclist	86.53	64.22	68.37
pedestrian	59.83	49.39	54.10

Table 3.1: Accuracy table of first training BEV [%]

3.3.2 Loss behavior

Focusing then on the loss, we obtain the results shown in Figure 3.4.
As we can see, it decreases significantly along the training, with a minimum value around 0.4.

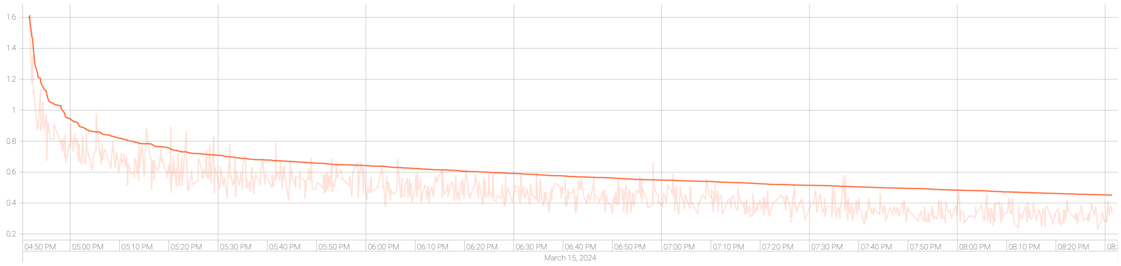

Figure 3.10: Loss behavior along the training

3.3.3 Results visualization

It is then possible to visualize, by means of the Open3d module [11], the BEV images with the insertion of bounding boxes. For example, in Figure 3.11 and Figure 3.12 we have the BEV image number 8 from the velodyne section of the KITTI dataset [23][29] evaluated firstly with the pre-trained PointPillars model, and secondly evaluated on the 80^{th} checkpoint of our training.

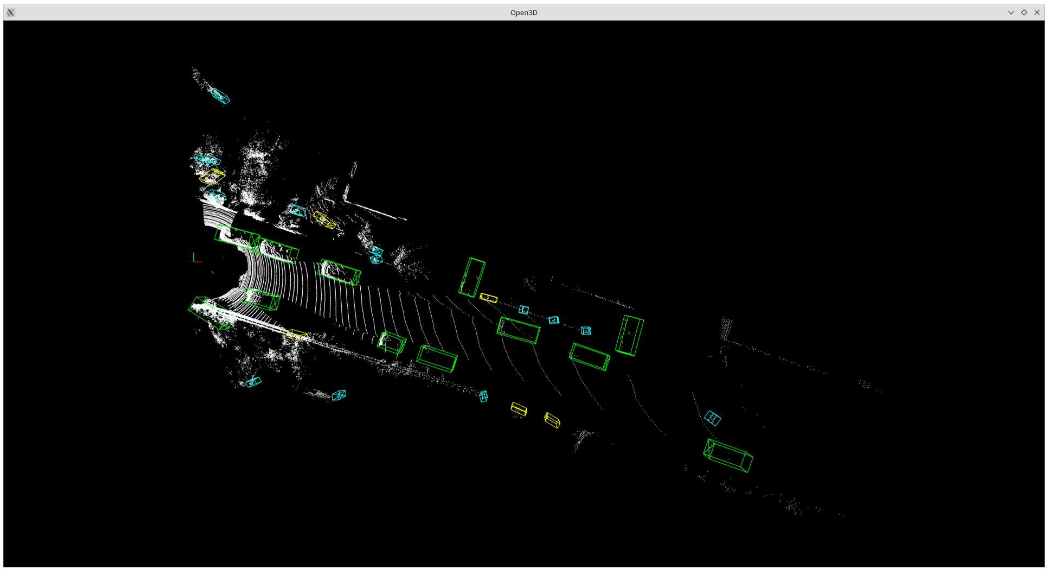

Figure 3.11: Pre-trained BEV with bounding box of image 8 of the dataset

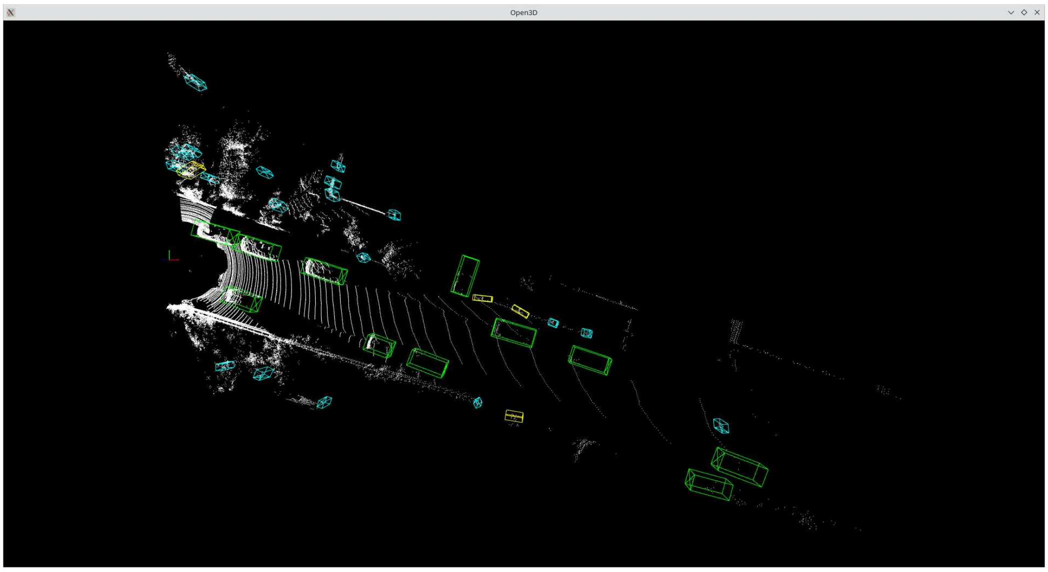

Figure 3.12: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

Figure 3.11 and Figure 3.12 have some visible differences: some of the bounding boxes are present only in the the newly trained version and vice-versa. Nevertheless, the number of errors increases with the distance from the source, and it is easy to see how results are less accurate for pedestrians (blue bounding boxes) and cyclists (yellow bounding boxes) with respect to cars (green bounding boxes).

3.3.4 More examples

To better visualize the comparison of the new training with the pre-trained model, here are some more results obtained with Open3D, applied on other images of the dataset:

- Image number 3000 of the velodyne KITTI [23][29] dataset is compared in Figure 3.13 (pre-trained) and Figure 3.14 (newly trained).
- Image number 5000 of the velodyne KITTI [23][29] dataset is compared in Figure 3.15 (pre-trained) and Figure 3.16 (newly trained).

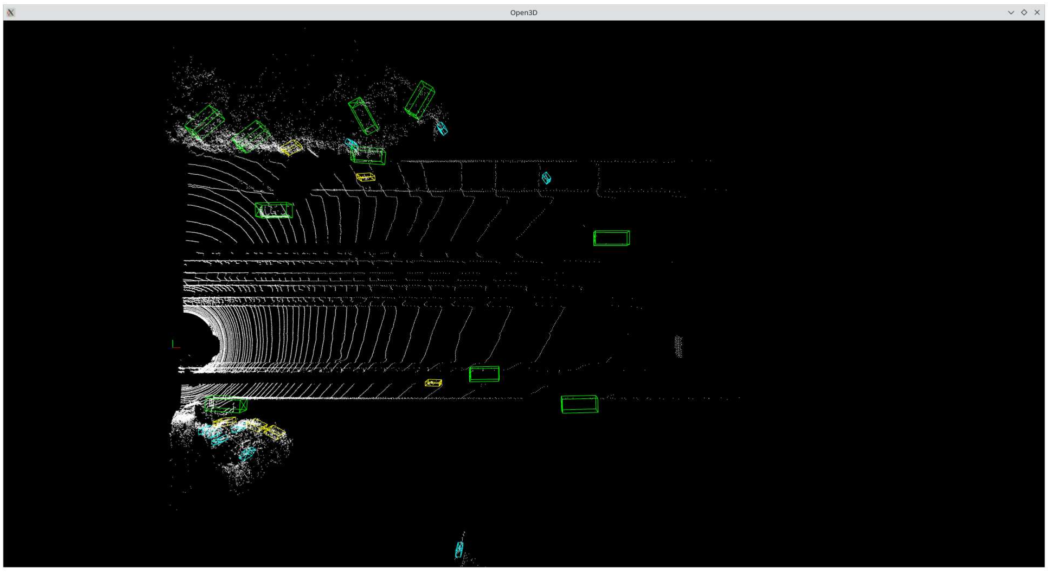

Figure 3.13: Pre-trained BEV with bounding box of image 3000 of the dataset

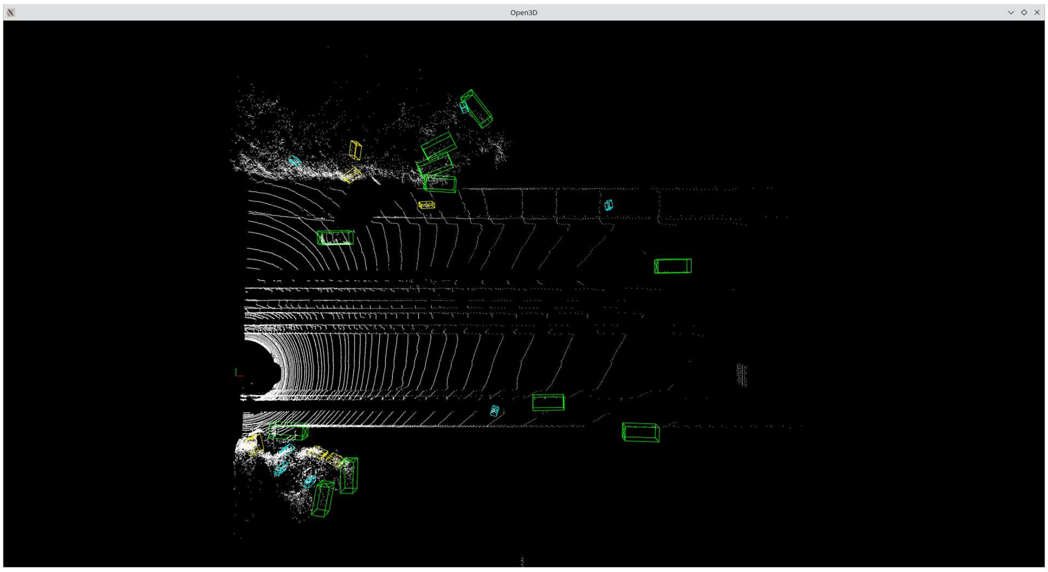

Figure 3.14: 80^{th} checkpoint BEV with bounding box of image 3000 of the dataset

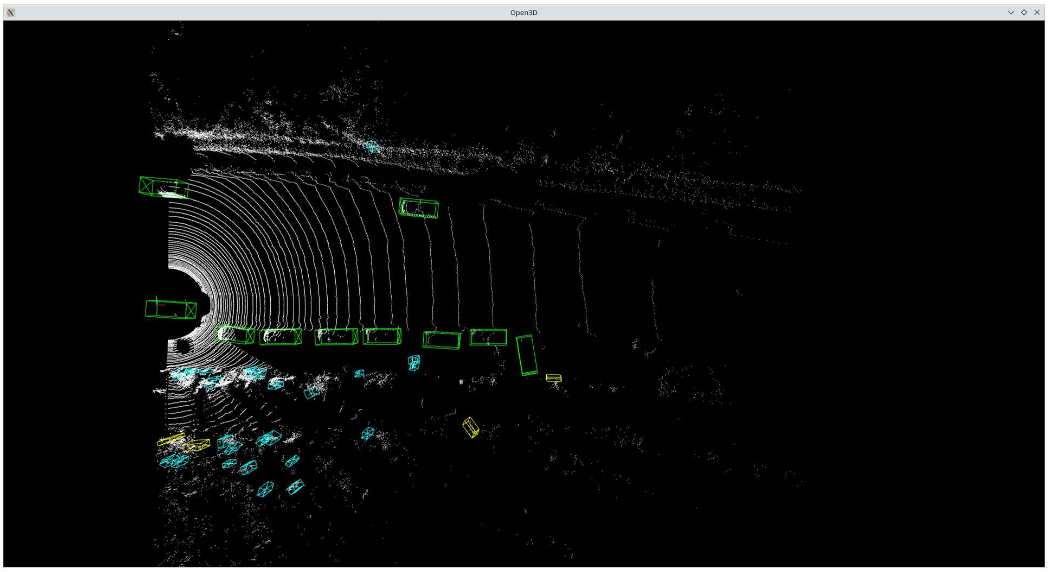

Figure 3.15: Pre-trained BEV with bounding box of image 5000 of the dataset

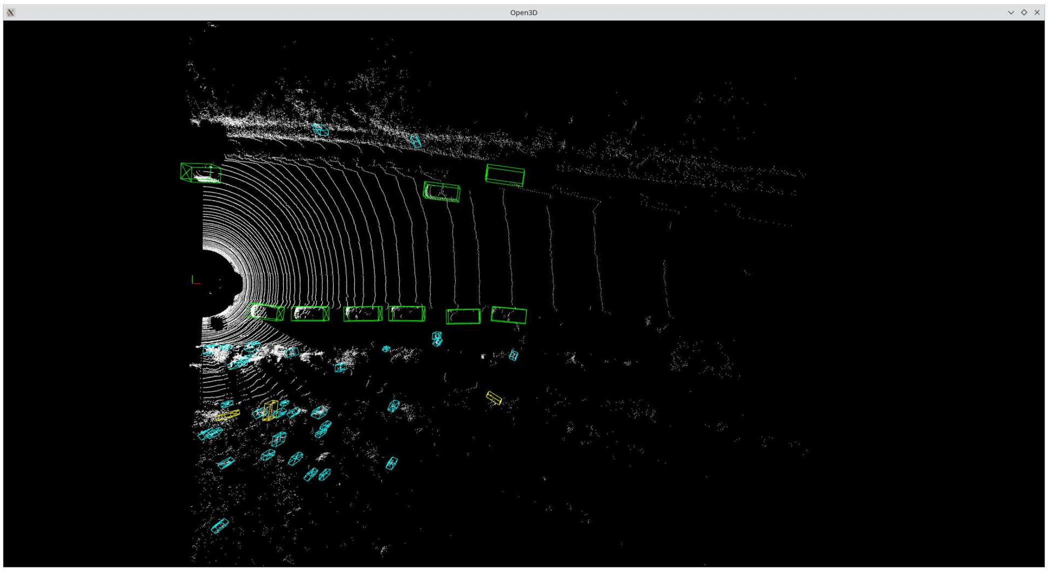

Figure 3.16: 80^{th} checkpoint BEV with bounding box of image 5000 of the dataset

For what concerns Figure 3.13 and Figure 3.14, the two images are almost identical, with some exceptions. The pre-trained model in fact has a lower number of bounding boxes, which means that it results in some false negatives.
Moving to Figure 3.15 the problem is the opposite, in fact the pre-trained model here adds some false positives, resulting in more bounding boxes than Figure 3.16. Overall, the number of differences between the two trainings is low, and most importantly, the wrong results are far from the source.

Chapter 4

Main task: porting of the detection algorithm to the embedded system, making use of firmware simplifications

In this chapter we will try to simplify the model to gain computational speed trying to maintain a good trade-off with the accuracy. We will work on the pillar feature net, scatter layer and backbone, without modifying the Conv2D-BatchNorm-ReLU structure used in the encoding-decoding stages of the model.
To obtain the best result, a large number of trainings has to be performed, with different configurations each time. The parameters to modify are the number of layers in the backbone and the number of filters in the different stages of the backbone (refer to Appendix A).
Inference time, number of parameters and accuracy will be taken into account and the best results will be compared with Section 3.3.

4.1 Training behavior with fixed number of filters

The aim of this section is to fix one parameter, the number of filters, and see how the resulting variables change modifying the number of layers.
For this purpose, we firstly fix the filters to the number in the original model, referring to lines 58, 62, 68 and 70 of the configuration file in Appendix A, secondly

we half that number, then we half only the number of input filters and finally we half only the up-sample filters. In every scenario, the number of layers analyzed changes between [2,4,4], [3,5,5] and [4,6,6].

4.1.1 Accuracy with filters fixed to their original number

In Figure 4.1, Figure 4.2 and Figure 4.3 the behavior of the 80^{th} checkpoints accuracy is shown for the different difficulties of the dataset. We can observe that they are almost constant, with a non-significant decrease along the number of layers for cars and cyclists, and a slight growth for pedestrians.

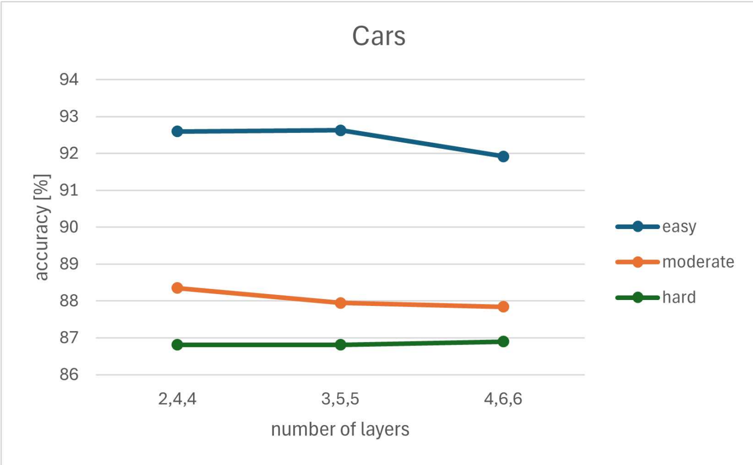

Figure 4.1: Accuracy [%] for cars versus the number of layers with filters fixed to their original number

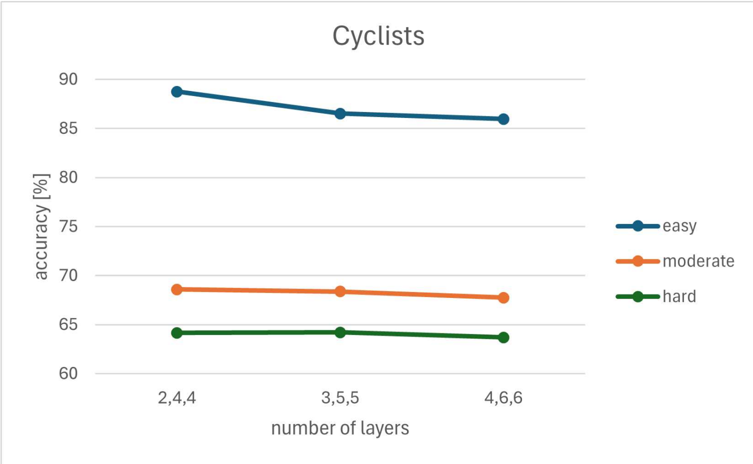

Figure 4.2: Accuracy [%] for cyclists versus the number of layers with filters fixed to their original number

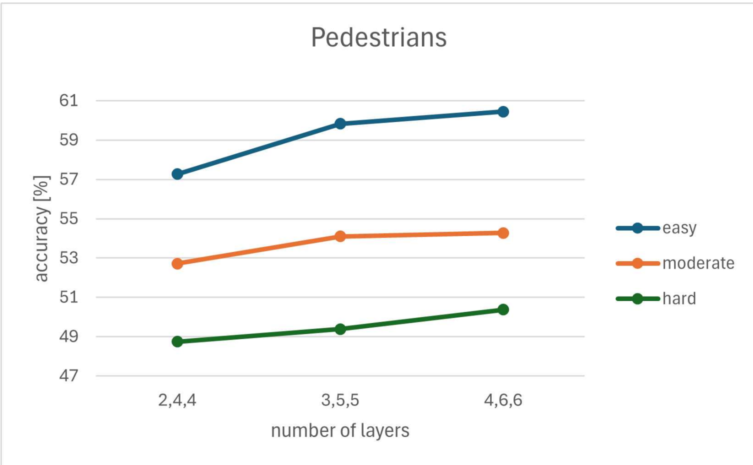

Figure 4.3: Accuracy [%] for pedestrians versus the number of layers with filters fixed to their original number

4.1.2 Accuracy with filters fixed to half of their original number

When we half the number of filters we obtain the 80^{th} checkpoints accuracy of Figure 4.4, Figure 4.5 and Figure 4.6. This time the change in values is more evident, especially for pedestrians. The best case scenario in this case would be then with a number of layers between [2,4,4] and [3,5,5]. Nevertheless, in general the values are lower than those in Section 4.1.1.

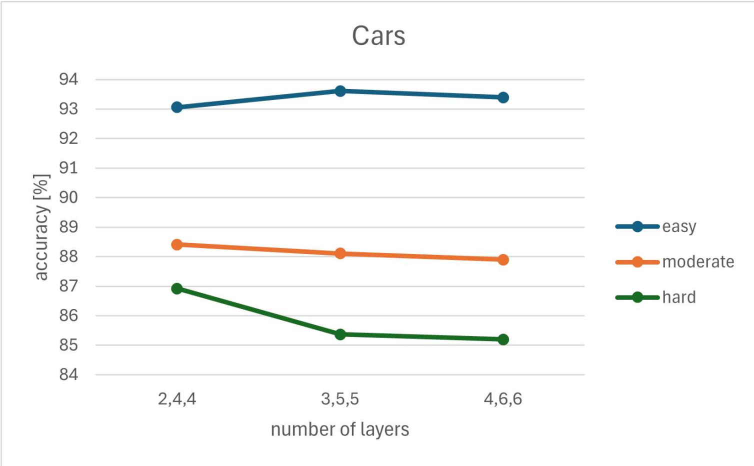

Figure 4.4: Accuracy [%] for cars versus the number of layers with half of the total number of filters

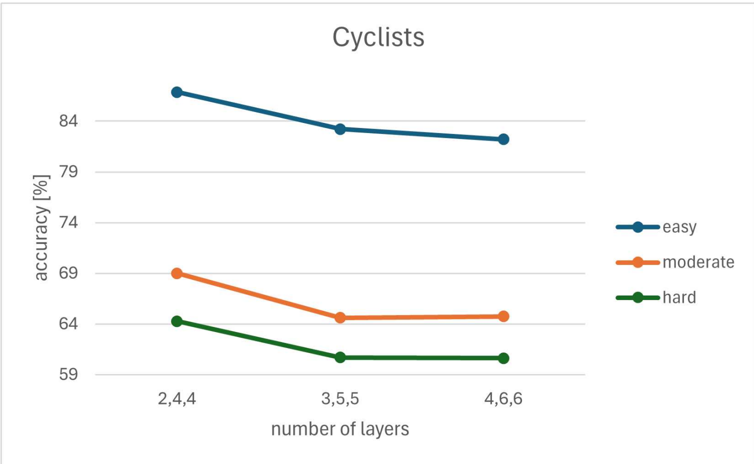

Figure 4.5: Accuracy [%] for cyclists versus the number of layers with half of the total number of filters

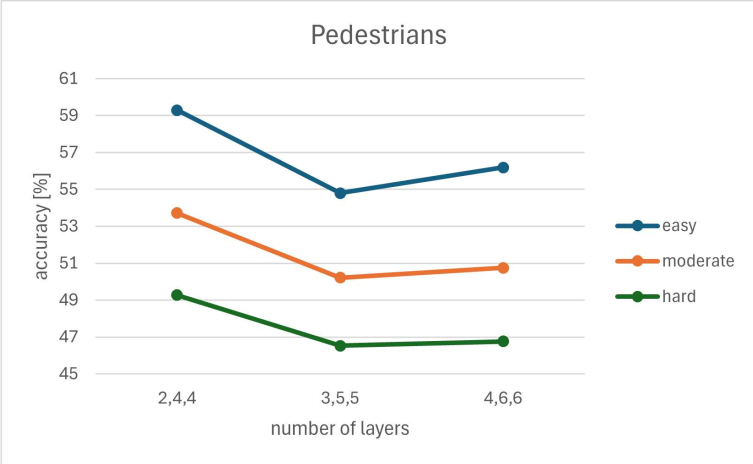

Figure 4.6: Accuracy [%] for pedestrians versus the number of layers with half of the total number of filters

4.1.3 Accuracy with up-sample filters fixed to half of their original number

Now we half only the up-sample filters. Figure 4.7, Figure 4.8 and Figure 4.9 show that the 80^{th} checkpoints accuracy generally decreases with this configuration (with an exception for cyclists that have a small growth). It is clear that the favorable number of layers is closer to [2,4,4].

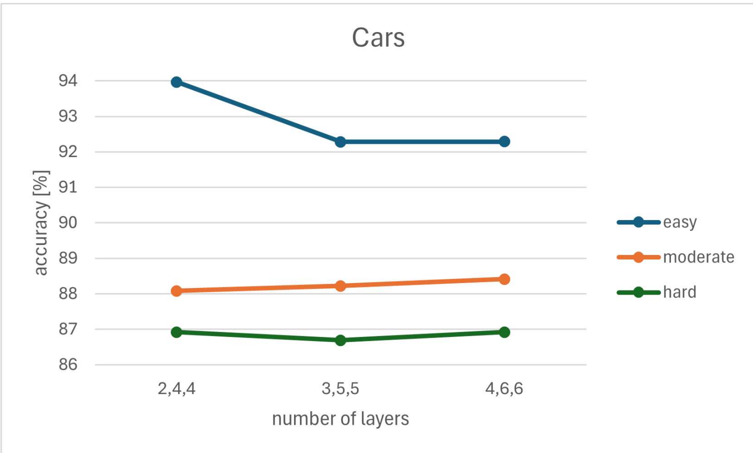

Figure 4.7: Accuracy [%] for cars versus the number of layers with half of the number of up-sample filters

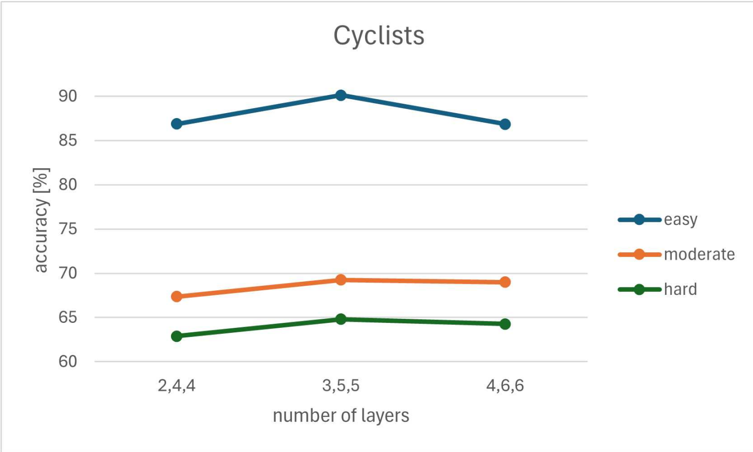

Figure 4.8: Accuracy [%] for cyclists versus the number of layers with half of the total number of up-sample filters

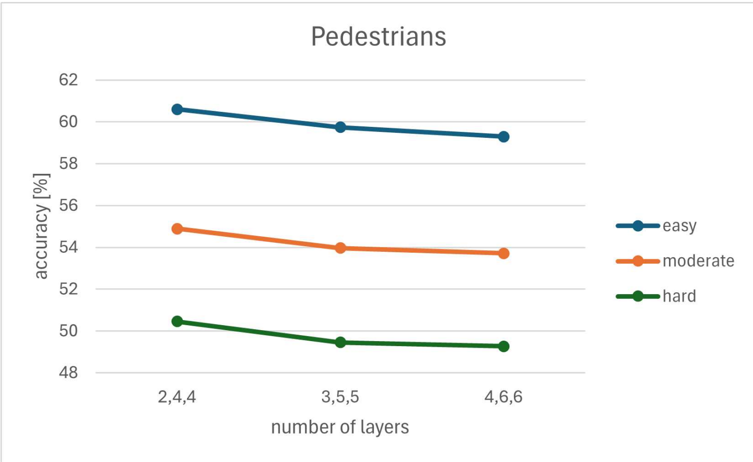

Figure 4.9: Accuracy [%] for pedestrians versus the number of layers with half of the total number of up-sample filters

4.1.4 Accuracy with input filters fixed to half of their original number

This time we half only the input filters. Results are shown in Figure 4.10, Figure 4.11 and Figure 4.12. Except for cyclists, the function visibly increases, especially for pedestrians, which have a very low accuracy for low number of layers. For this reason, with this amount of filters the best number of layers would be between [3,5,5] and [4,6,6].

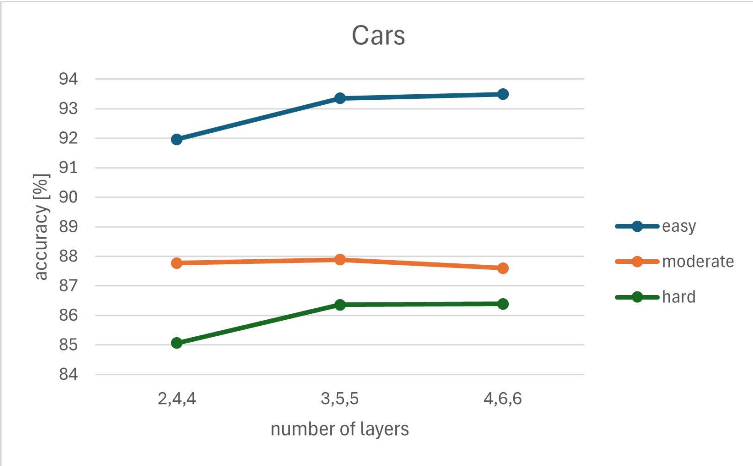

Figure 4.10: Accuracy [%] for cars versus the number of layers with half of the number of input filters

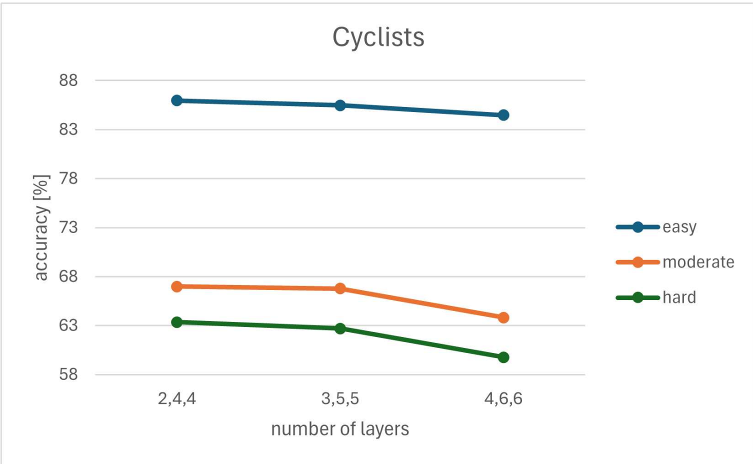

Figure 4.11: Accuracy [%] for cyclists versus the number of layers with half of the total number of input filters

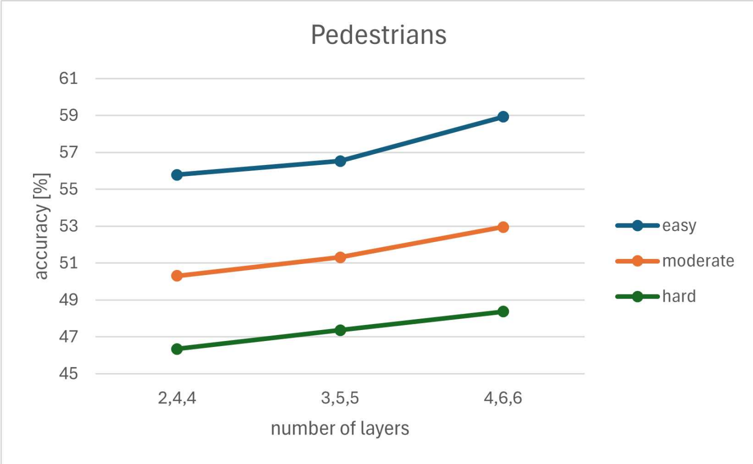

Figure 4.12: Accuracy [%] for pedestrians versus the number of layers with half of the total number of input filters

4.1.5 Number of parameters and inference time compared for the different configurations

We now consider the number of parameters and the inference time for the different setups.

- In terms of the number of parameters, it only depends on the number of layers, so for every filter configuration it will be the same, as described in Figure 4.13. We can observe that it grows linearly, starting from 109 millions for [2,4,4] layers, 127 millions for [3,5,5] and 145 millions for [3,6,6], therefore, to have better performances we need fewer layers. For this reason, we can exclude from consideration every configuration related to [4,6,6] layers.

- For what concerns the inference time, it will change according to the number of parameters, because they increase the complexity of computation. As a consequence we want to reduce as much as possible the number of layers

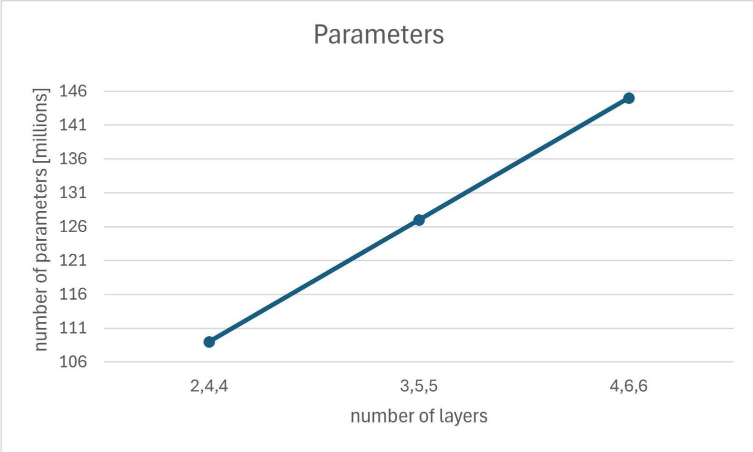

Figure 4.13: Number of parameters versus number of layers

4.2 Training behavior with fixed number of layers

Now we fix the number of layers in the backbone and analyse how the resulting variables change with the number of filters. In light of this, we will initially fix the number of layers to [3,5,5], referring to line 66 of the configuration file in Appendix A, and afterward we will lower it to [2,4,4]. We will not consider [4,6,6] layers since we ruled them out in Section 4.1.5. The variable is the number of filters that is going to change from half of the original value to double of that value.

4.2.1 Accuracy with backbone layers fixed to [3,5,5]

In figure 4.14, Figure 4.15 and Figure 4.16 we can see the graphs for a number of layer fixed to [3,5,5]. We can observe that the resulting slopes are more steep then the ones in Section 4.1, especially for cyclists and pedestrians. In general, doubling the number of filters we obtain better accuracy, with the exception of cars on easy and moderate difficulty dataset.

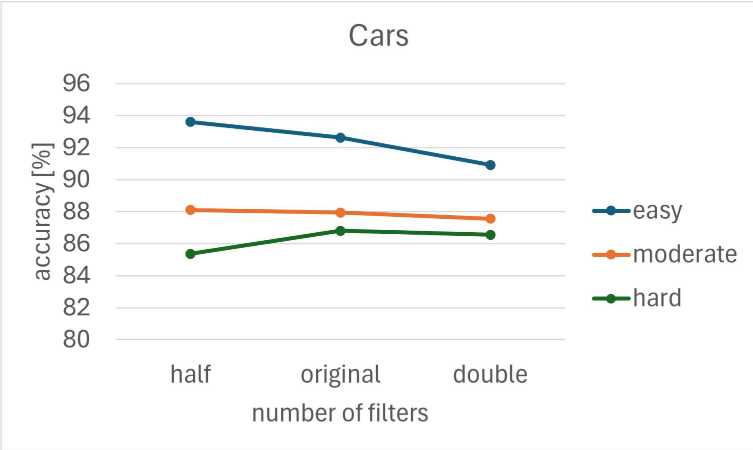

Figure 4.14: Accuracy [%] for cars versus the number of total filters with [3,5,5] layers

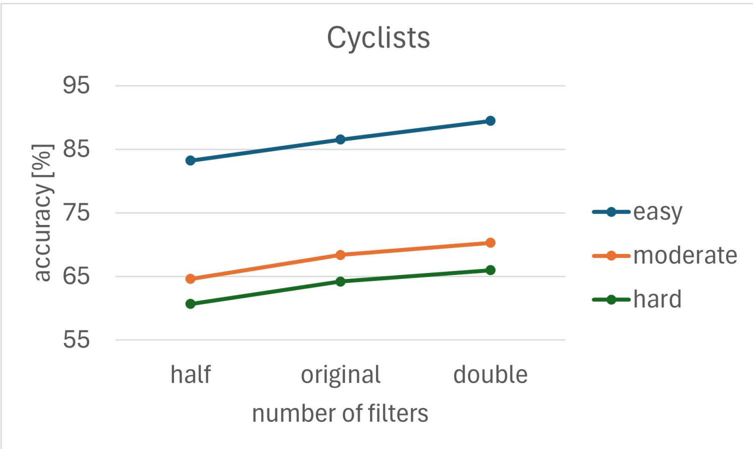

Figure 4.15: Accuracy [%] for cyclists versus the number of total filters with [3,5,5] layers

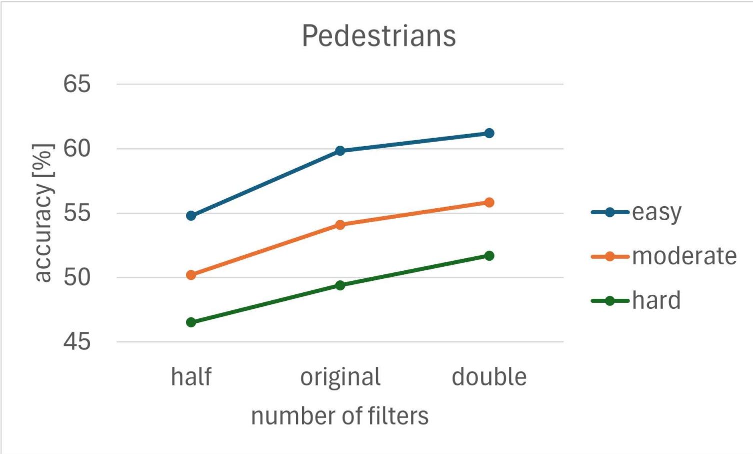

Figure 4.16: Accuracy [%] for pedestrians versus the number of total filters with [3,5,5] layers

4.2.2 Accuracy with backbone layers fixed to [2,4,4]

Now we reduce the number of layers, fixing them to [2,4,4]. The graphs obtained in Figure 4.17, Figure 4.18 and Figure 4.19 are almost constant and in general better than those in Section 4.2.1.

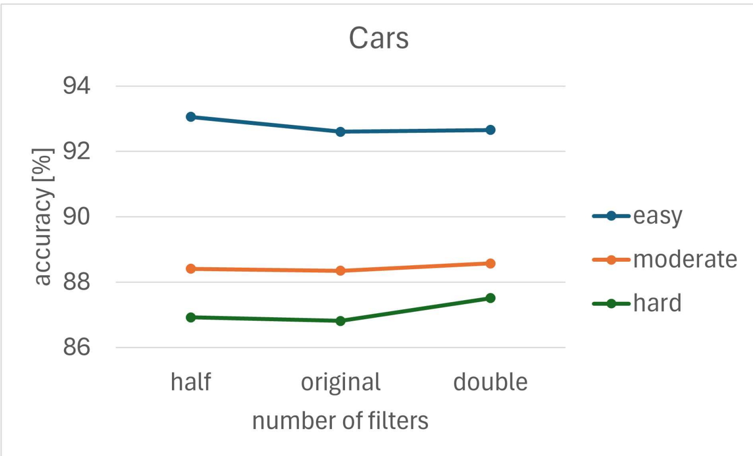

Figure 4.17: Accuracy [%] for cars versus the number of total filters with [2,4,4] layers

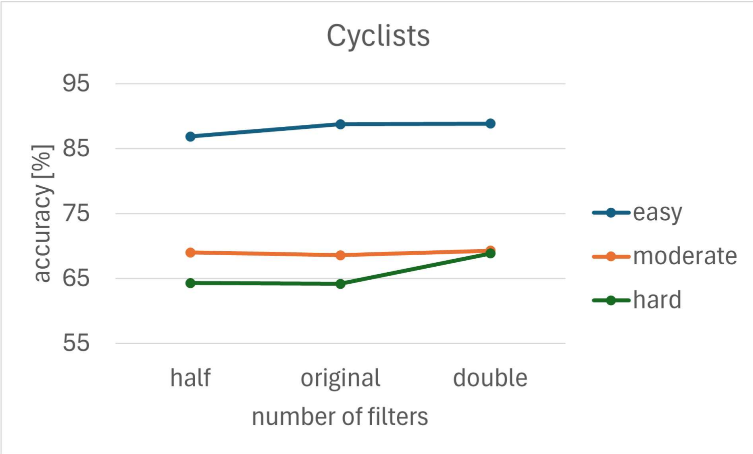

Figure 4.18: Accuracy [%] for cyclists versus the number of total filters with [2,4,4] layers

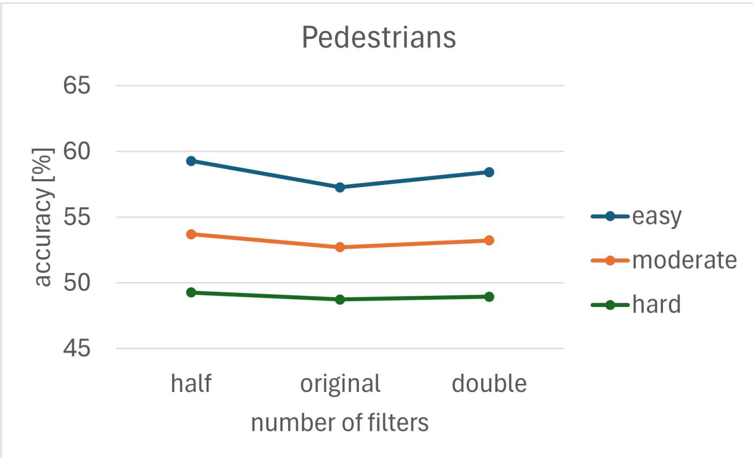

Figure 4.19: Accuracy [%] for pedestrians versus the number of total filters with [2,4,4] layers

4.2.3 Number of parameters and inference time compared for the different configurations

Finally, we want to look at the number of parameters and the training time.

- As we said in Section 4.1.5, the number of parameters does not depend on the amount of filters; therefore it will always be 127 million for [3,5,5] layers and 109 million for [2,4,4].

- For what concerns the duration of computation, we observe in Figure 4.20 that it grows almost exponentially with the number of filters, for both amounts of layers. For this reason we do not want to double this value, on the other hand we do not need to reduce it significantly since lowering will not result in a much faster computation, but it could affect negatively the performances.

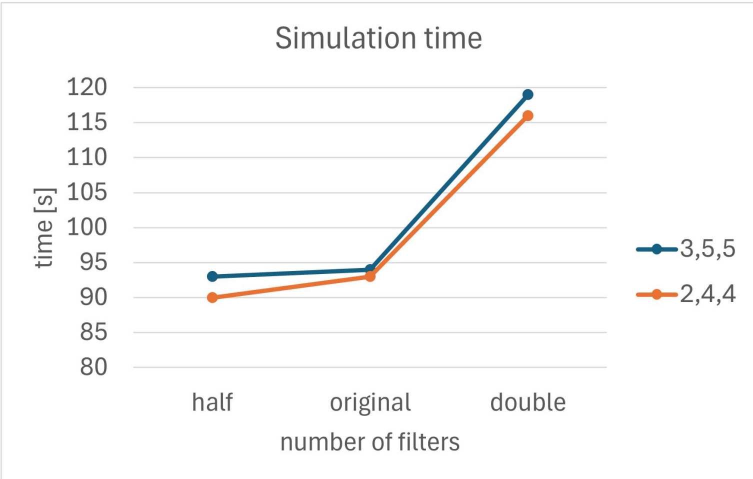

Figure 4.20: Inference time versus number of layers

Chapter 5

Prototype: performance evaluation

For each graph in Chapter 4 we extrapolate the configuration with the best results, not including every training with [4,6,6] layers and doubled filters, because of the considerations in Section 4.1.5 and Section 4.2.3.
The results obtained in Chapter 4 will allow us to limit our options to the best six configurations. To finally find the only optimal solution, we want to evaluate their losses and visualize by means of Open3D [11] their resulting image with bounding boxes, to compare it with the one in Section 3.3.3.

5.1 Reducing the number of layers in the backbone to [3,4,4]

In Listing 5.1 is provided the modified part of the configuration file (see Appendix A).

Listing 5.1: modified configuration file

```
BACKBONE_2D:
    NAME: BaseBEVBackbone
    LAYER_NUMS: [3, 4, 4]
    LAYER_STRIDES: [2, 2, 2]
    NUM_FILTERS: [64, 128, 256]
    UPSAMPLE_STRIDES: [1, 2, 4]
    NUM_UPSAMPLE_FILTERS: [128, 128, 128]
```

5.1.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.1.

	easy	hard	moderate
car	92.24	85.95	87.35
cyclist	86.39	63.80	68.17
pedestrian	59.33	49.87	52.93

Table 5.1: Accuracy table of the 80^{th} checkpoint [%]

5.1.2 Losses

The obtained losses remain very close to Figure 3.4, as observed in Figure 5.1.

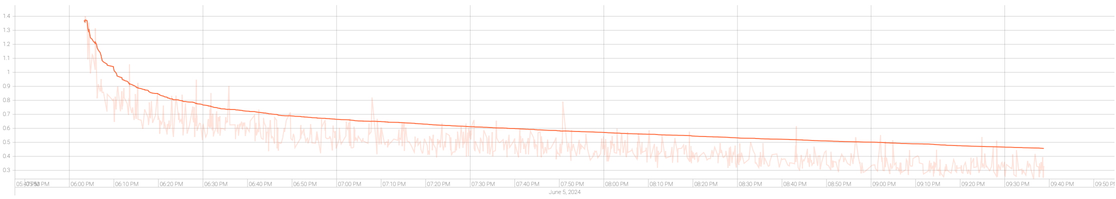

Figure 5.1: Loss behavior along the training

5.1.3 Results visualization

To better visualize these results, in Figure 5.2 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

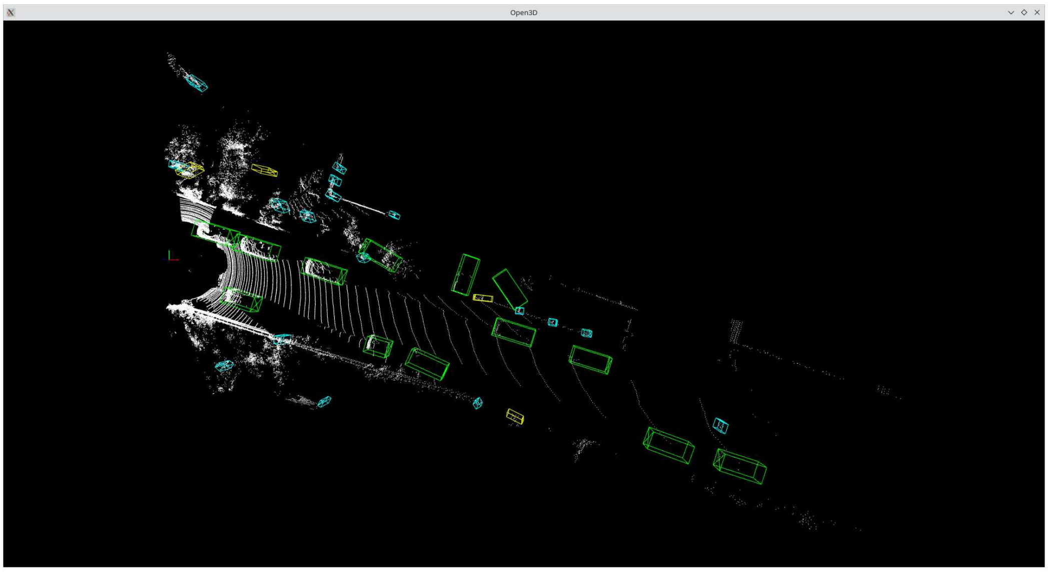

Figure 5.2: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.2 Reducing the number of layers in the backbone to [2,4,4] and halving the number of filters

In Listing 5.2 is provided the modified part of the configuration file (see Appendix A).

Listing 5.2: modified configuration file

```
    VFE:
        NAME: PillarVFE
        WITH_DISTANCE: False
        USE_ABSLOTE_XYZ: True
        USE_NORM: True
        NUM_FILTERS: [32]

    MAP_TO_BEV:
        NAME: PointPillarScatter
        NUM_BEV_FEATURES: 32

    BACKBONE_2D:
        NAME: BaseBEVBackbone
        LAYER_NUMS: [2, 4, 4]
        LAYER_STRIDES: [2, 2, 2]
```

```
        NUM_FILTERS: [32, 64, 128]
        UPSAMPLE_STRIDES: [1, 2, 4]
        NUM_UPSAMPLE_FILTERS: [64, 64, 64]
```

5.2.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.2.

	easy	hard	moderate
car	93.06	85.12	87.74
cyclist	81.47	60.04	64.13
pedestrian	58.44	47.53	51.76

Table 5.2: Accuracy table of the 80^{th} checkpoint [%]

5.2.2 Losses

The obtained losses have significantly increased, as observed in Figure 5.3, excluding this configuration from candidates.

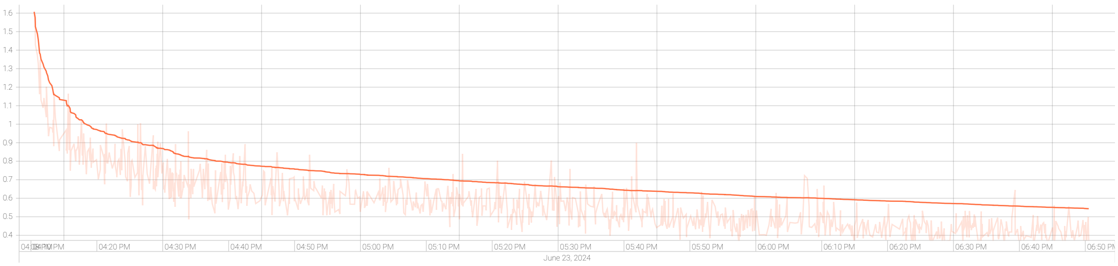

Figure 5.3: Loss behavior along the training

5.2.3 Results visualization

To better visualize these results, in Figure 5.4 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

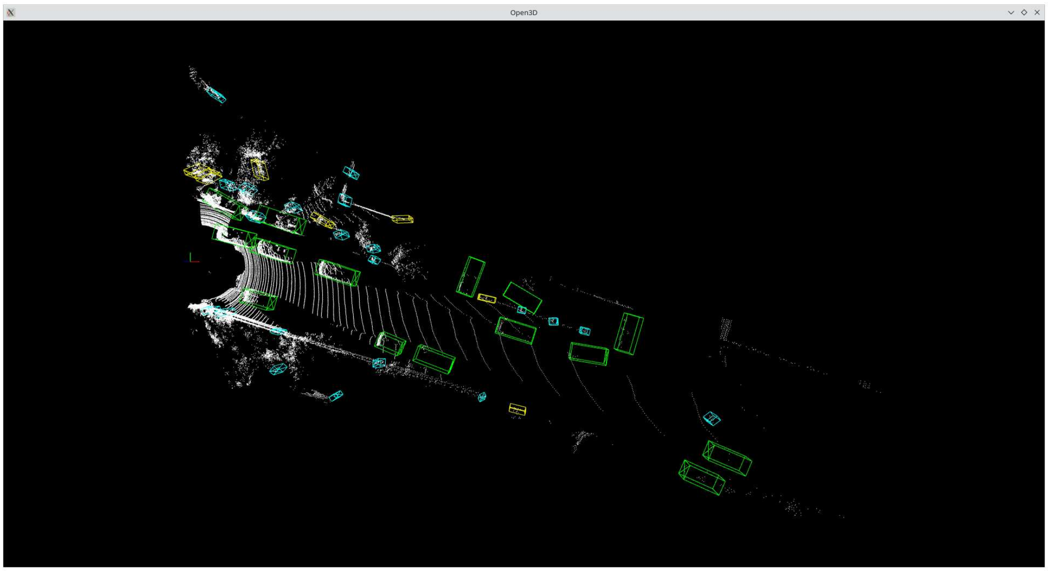

Figure 5.4: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.3 Halving the number of input filters

In Listing 5.3 is provided the modified part of the configuration file with (see Appendix A).

Listing 5.3: modified configuration file

```
VFE:
    NAME: PillarVFE
    WITH_DISTANCE: False
    USE_ABSLOTE_XYZ: True
    USE_NORM: True
    NUM_FILTERS: [32]

MAP_TO_BEV:
    NAME: PointPillarScatter
    NUM_BEV_FEATURES: 32

BACKBONE_2D:
    NAME: BaseBEVBackbone
    LAYER_NUMS: [3, 5, 5]
    LAYER_STRIDES: [2, 2, 2]
    NUM_FILTERS: [32, 128, 256]
    UPSAMPLE_STRIDES: [1, 2, 4]
    NUM_UPSAMPLE_FILTERS: [128, 128, 128]
```

5.3.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.3.

	easy	hard	moderate
car	93.35	86.36	87.89
cyclist	85.48	62.69	66.78
pedestrian	56.53	47.37	51.32

Table 5.3: Accuracy table of the 80^{th} checkpoint [%]

5.3.2 Losses

The obtained losses are lower then Figure 5.3, as observed in Figure 5.5, but they are still too high, excluding also this configuration from candidates.

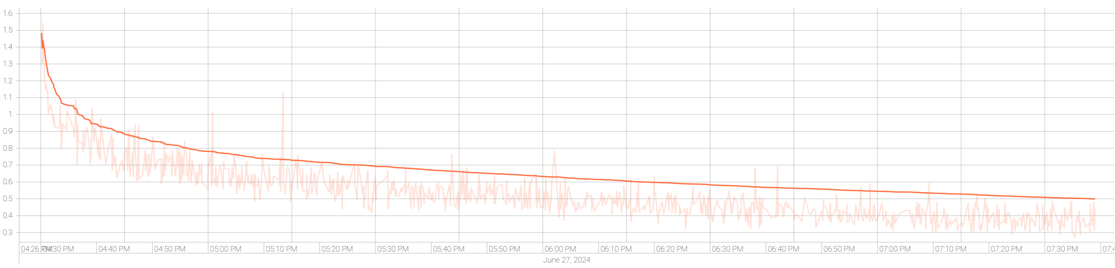

Figure 5.5: Loss behavior along the training

5.3.3 Results visualization

To better visualize these results, in Figure 5.6 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

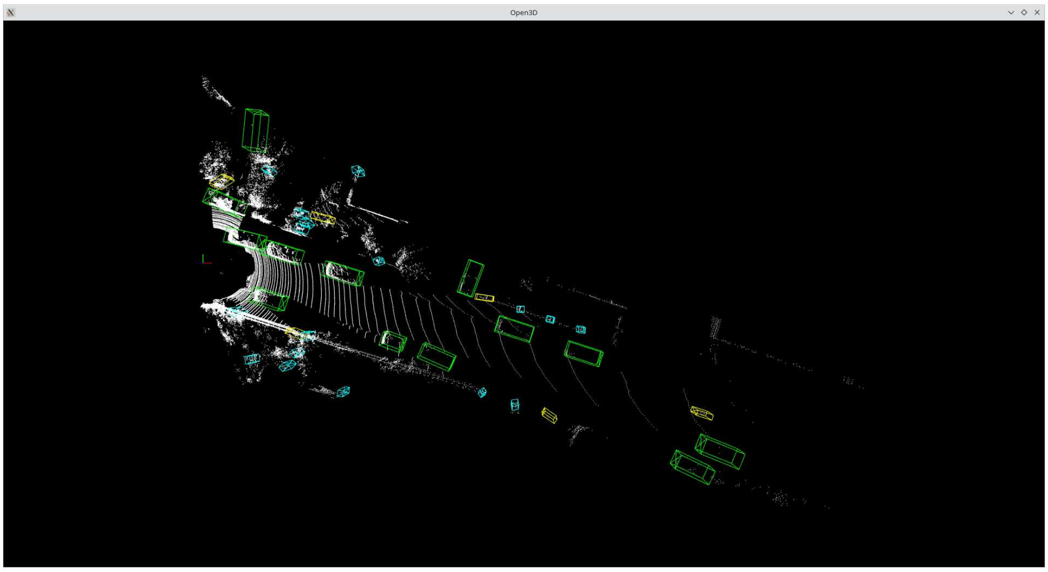

Figure 5.6: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.4 Reducing the number of layers in the backbone to [3,4,4] and halving the number of input filters

In Listing 5.4 is provided the modified part of the configuration file with (see Appendix A).

Listing 5.4: modified configuration file

```
VFE:
    NAME: PillarVFE
    WITH_DISTANCE: False
    USE_ABSLOTE_XYZ: True
    USE_NORM: True
    NUM_FILTERS: [32]

MAP_TO_BEV:
    NAME: PointPillarScatter
    NUM_BEV_FEATURES: 32

BACKBONE_2D:
    NAME: BaseBEVBackbone
    LAYER_NUMS: [3, 4, 4]
    LAYER_STRIDES: [2, 2, 2]
```

```
        NUM_FILTERS: [32, 128, 256]
        UPSAMPLE_STRIDES: [1, 2, 4]
        NUM_UPSAMPLE_FILTERS: [128, 128, 128]
```

5.4.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.4.

	easy	hard	moderate
car	93.61	86.87	88.30
cyclist	87.86	62.82	97.14
pedestrian	60.33	49.67	54.07

Table 5.4: Accuracy table of the 80^{th} checkpoint [%]

5.4.2 Losses

The obtained losses, as observed in Figure 5.7, are still too high, excluding also this configuration from candidates.

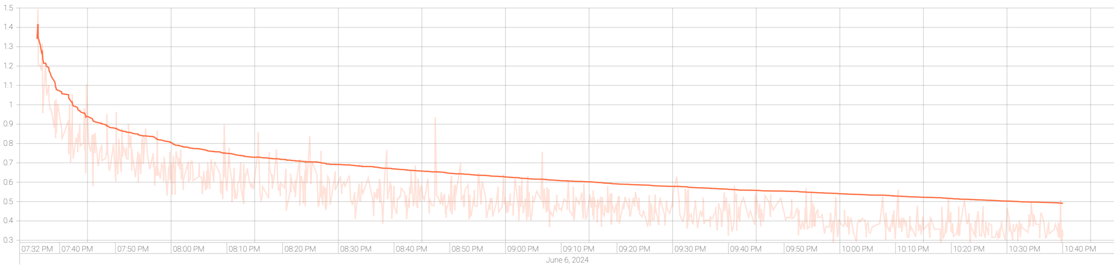

Figure 5.7: Loss behavior along the training

5.4.3 Results visualization

To better visualize these results, in Figure 5.8 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

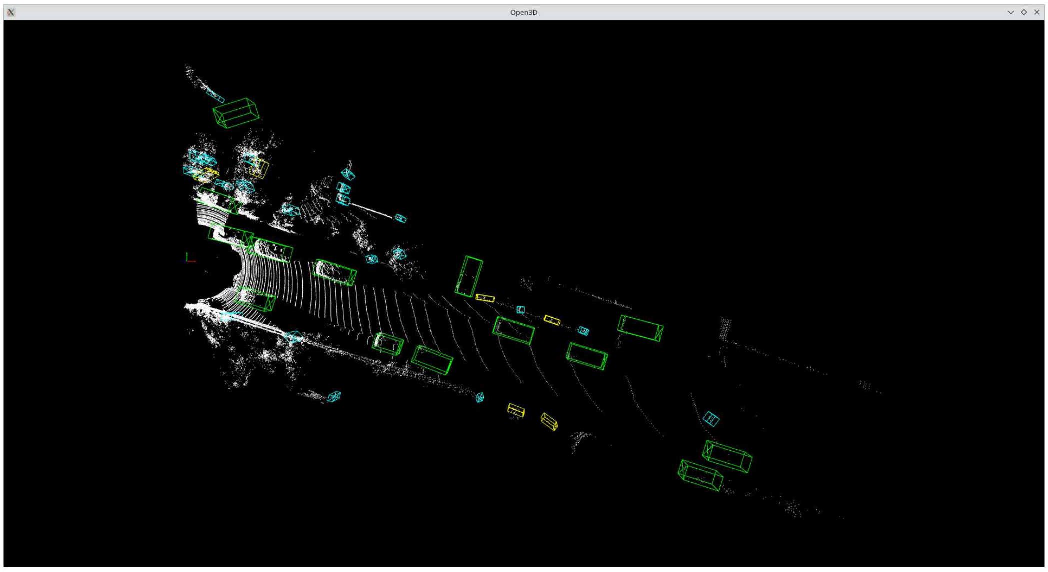

Figure 5.8: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.5 Halving the number of up-sample filters

In Listing 5.5 is provided the modified part of the configuration file with (see Appendix A).

Listing 5.5: modified configuration file

```
BACKBONE_2D:
    NAME: BaseBEVBackbone
    LAYER_NUMS: [3, 5, 5]
    LAYER_STRIDES: [2, 2, 2]
    NUM_FILTERS: [64, 128, 256]
    UPSAMPLE_STRIDES: [1, 2, 4]
    NUM_UPSAMPLE_FILTERS: [64, 64, 64]
```

5.5.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.5.

	easy	hard	moderate
car	91.88	86.60	87.82
cyclist	89.16	66.09	70.73
pedestrian	60.67	50.30	54.77

Table 5.5: Accuracy table of the 80^{th} checkpoint [%]

5.5.2 Losses

The obtained losses are close to Figure 3.4, as observed in Figure 5.9.

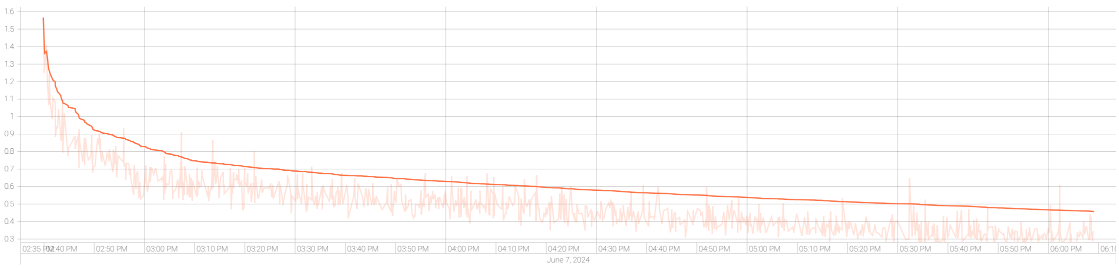

Figure 5.9: Loss behavior along the training

5.5.3 Results visualization

To better visualize these results, in Figure 5.10 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

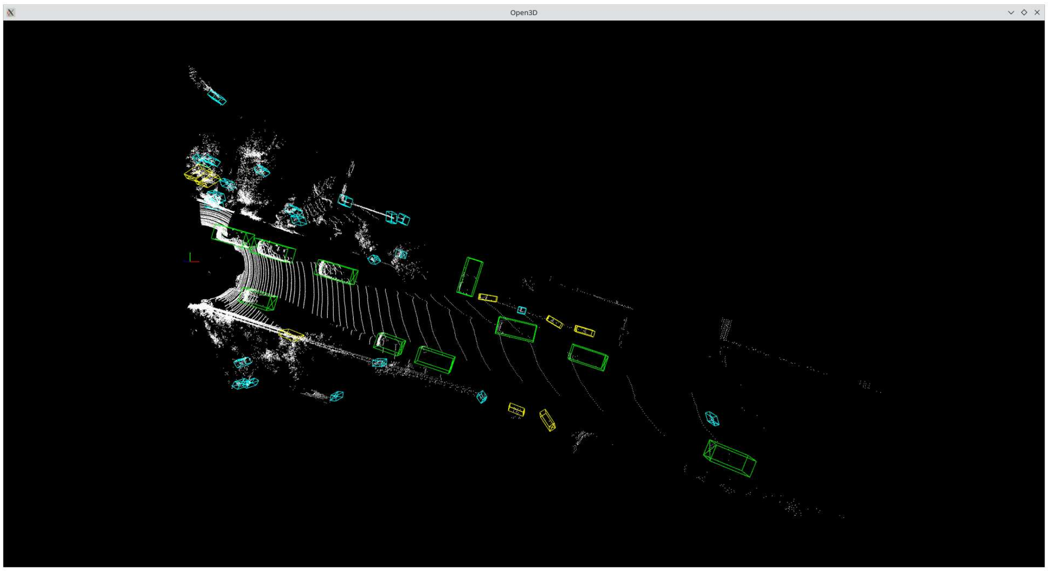

Figure 5.10: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.6 Reducing the number of layers in the backbone to [2,4,4] and halving the number of up-sample filters

In Listing 5.5 is provided the modified part of the configuration file with (see Appendix A).

Listing 5.6: modified configuration file

```
BACKBONE_2D:
    NAME: BaseBEVBackbone
    LAYER_NUMS: [2, 4, 4]
    LAYER_STRIDES: [2, 2, 2]
    NUM_FILTERS: [64, 128, 256]
    UPSAMPLE_STRIDES: [1, 2, 4]
    NUM_UPSAMPLE_FILTERS: [64, 64, 64]
```

5.6.1 Accuracy

The final values of the 80^{th} checkpoint of the training are listed in Table 5.6.

	easy	hard	moderate
car	93.67	86.92	88.08
cyclist	86.87	62.88	67.35
pedestrian	60.60	50.46	54.89

Table 5.6: Accuracy table of the 80^{th} checkpoint [%]

5.6.2 Losses

The obtained losses are very close to Figure 3.4, as observed in Figure 5.11.

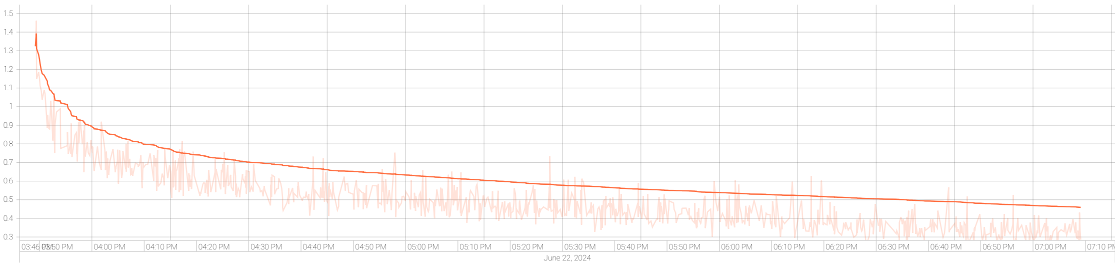

Figure 5.11: Loss behavior along the training

5.6.3 Results visualization

To better visualize these results, in Figure 5.12 we have the same image as Figure 3.12, but with the bounding boxes obtained with this last configuration.

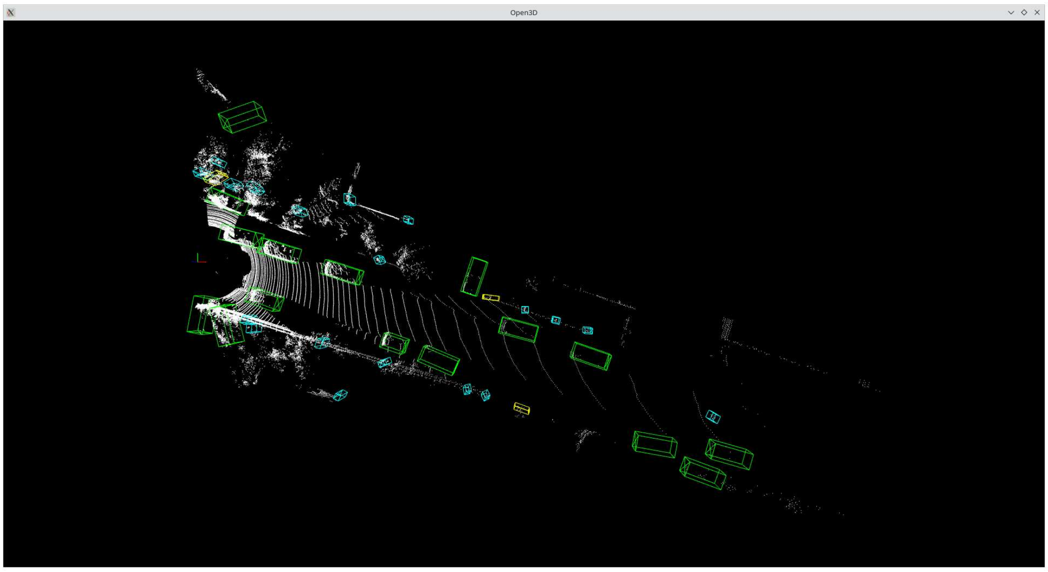

Figure 5.12: 80^{th} checkpoint BEV with bounding box of image 8 of the dataset

5.7 Optimal solution

To understand the results, we compare the data of the original model and of Section 5.1, Section 5.2, Section 5.3, Section 5.4, Section 5.5 and Section 5.6 in an accuracy versus inference time graph to have a clear idea of which one is the best solution. On the x-axis we have the testing time while on the y-axis we have the validation accuracy of the trained model.
We want the highest accuracy possible with the lowest amount of time, so the closest we are to the top left corner of the accuracy-delay graphs, the better.
Looking at Figure 5.13, Figure 5.14, Figure 5.15, Figure 5.16, Figure 5.17, Figure 5.18, Figure 5.19, Figure 5.20 and Figure 5.21, it is easy to see that the best configuration is the one described in Section 5.6, which allows us to have the best performances to work in real-time conditions without increasing losses, in fact Figure 5.11 has not increased significantly with respect to Figure 3.10. Furthermore, as shown in Section 4.1.5, reducing the number of CNNs, also the number of parameters is low and Figure 5.12 has very high compatibility with Figure 3.12 (with the exception of very few cyclists and pedestrians wrongly spotted, but far from the source).
For these reasons, reducing the number of layers in the backbone to [2,4,4] and halving the number of up-sample filters is the best solution for our problem.

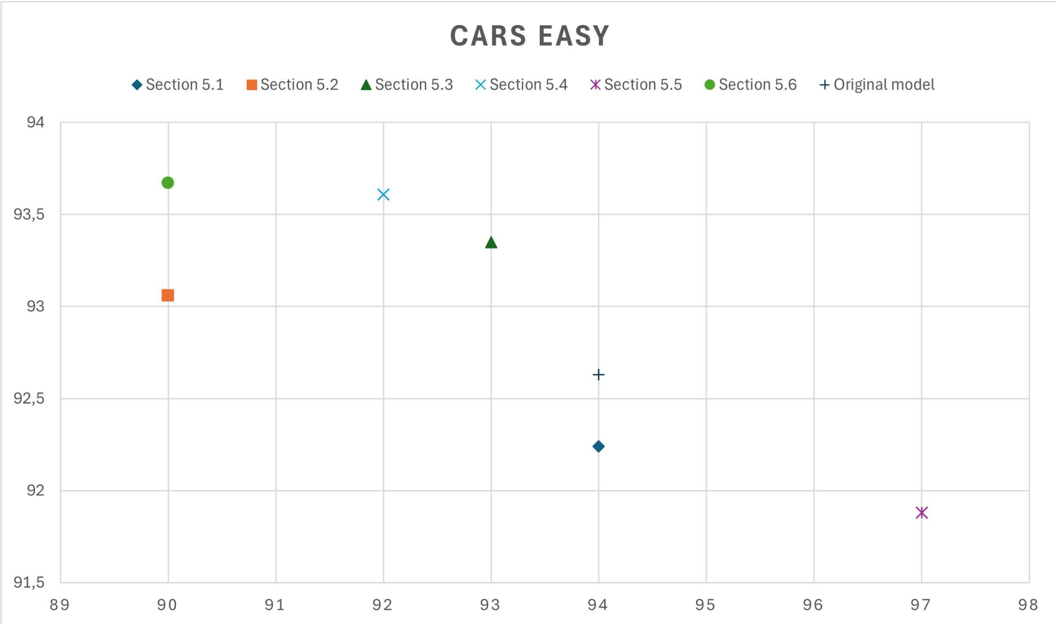

Figure 5.13: Accuracy-delay graph for cars evaluated on easy difficulty dataset

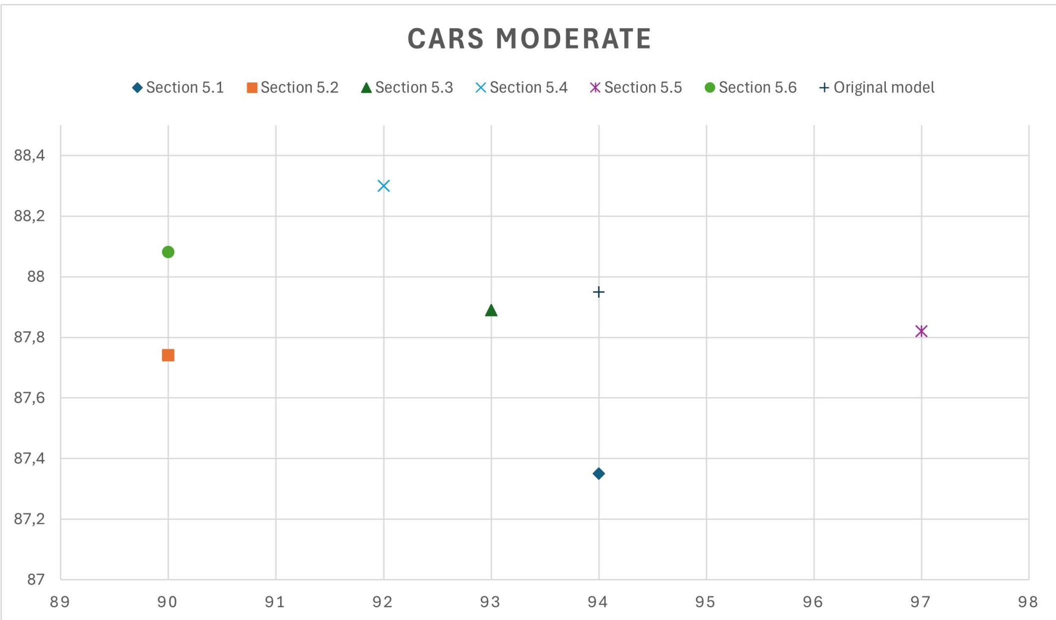

Figure 5.14: Accuracy-delay graph for cars evaluated on moderate difficulty dataset

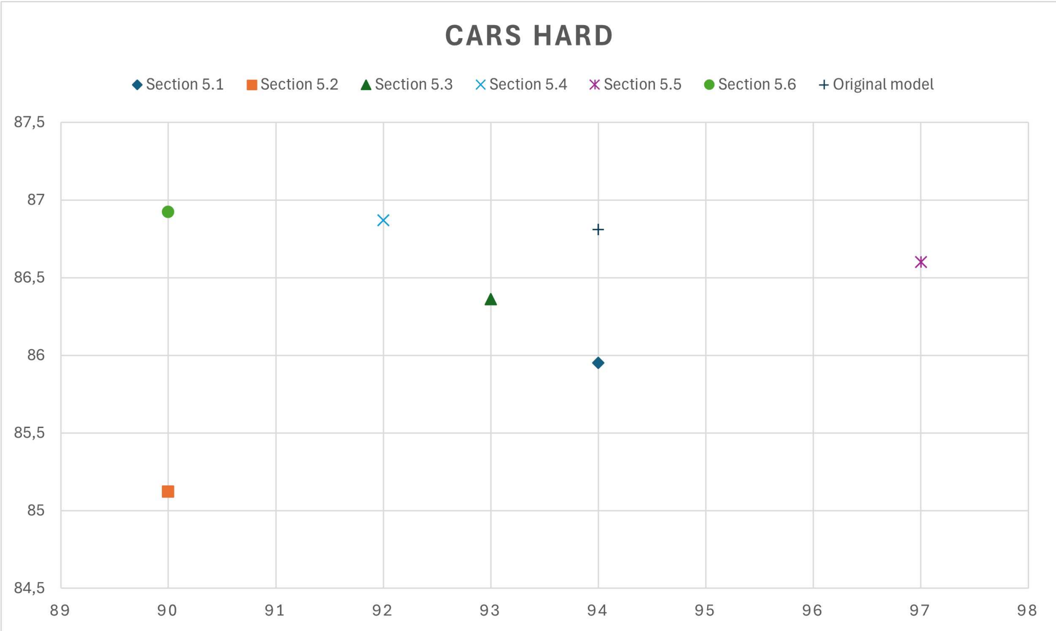

Figure 5.15: Accuracy-delay graph for cars evaluated on hard difficulty dataset

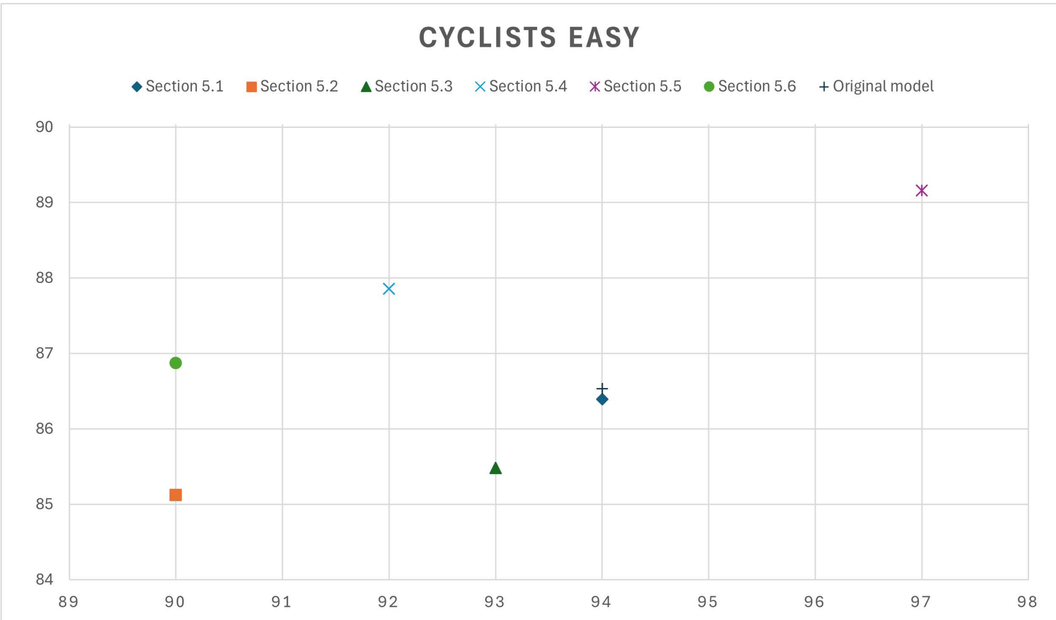

Figure 5.16: Accuracy-delay graph for cyclists evaluated on easy difficulty dataset

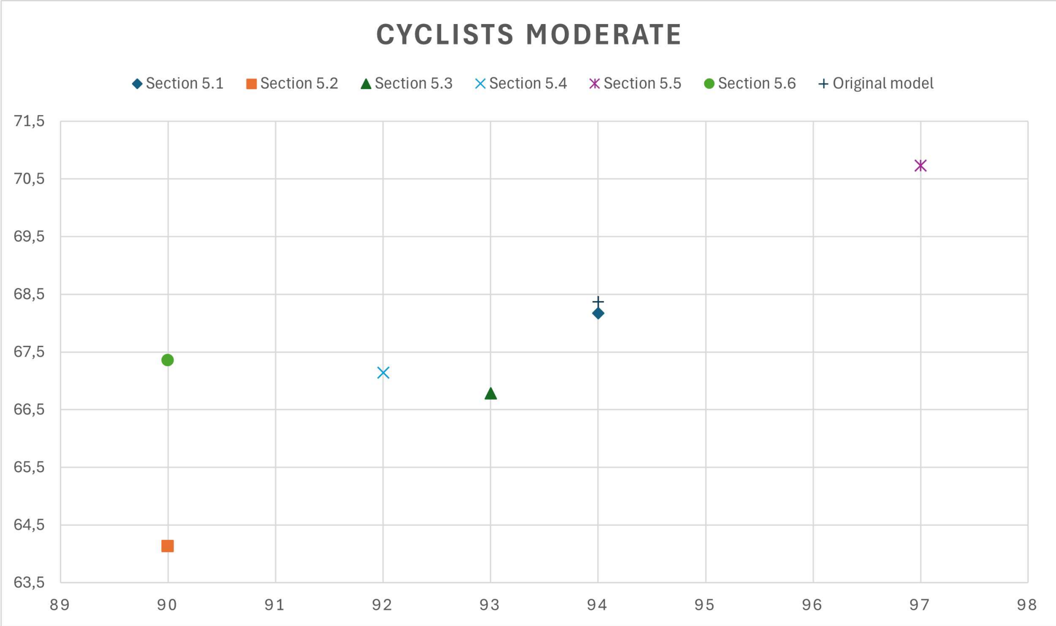

Figure 5.17: Accuracy-delay graph for cyclists evaluated on moderate difficulty dataset

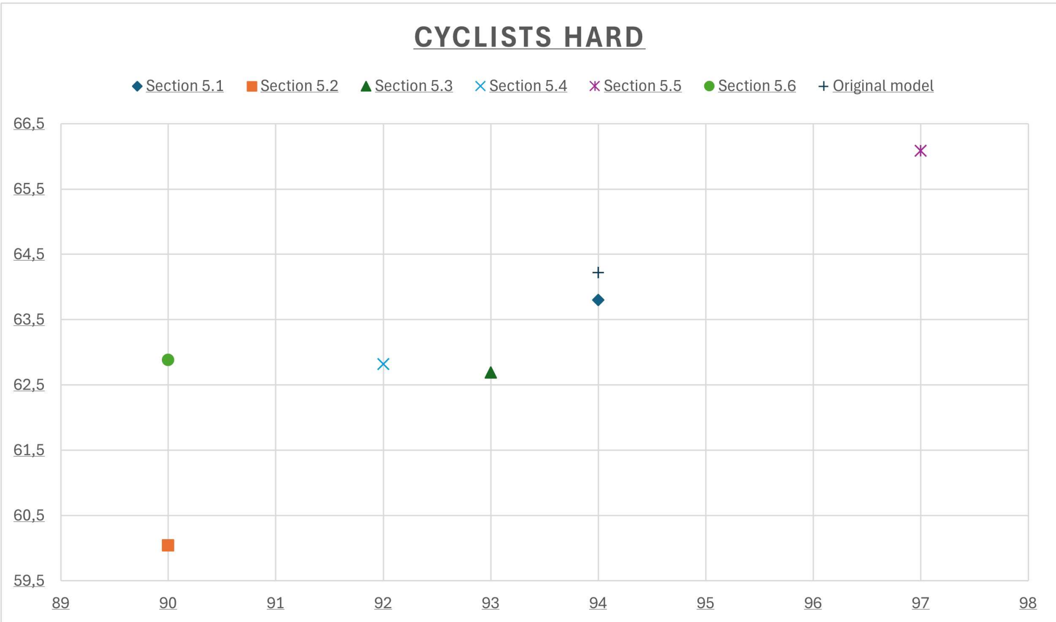

Figure 5.18: Accuracy-delay graph for cyclists evaluated on hard difficulty dataset

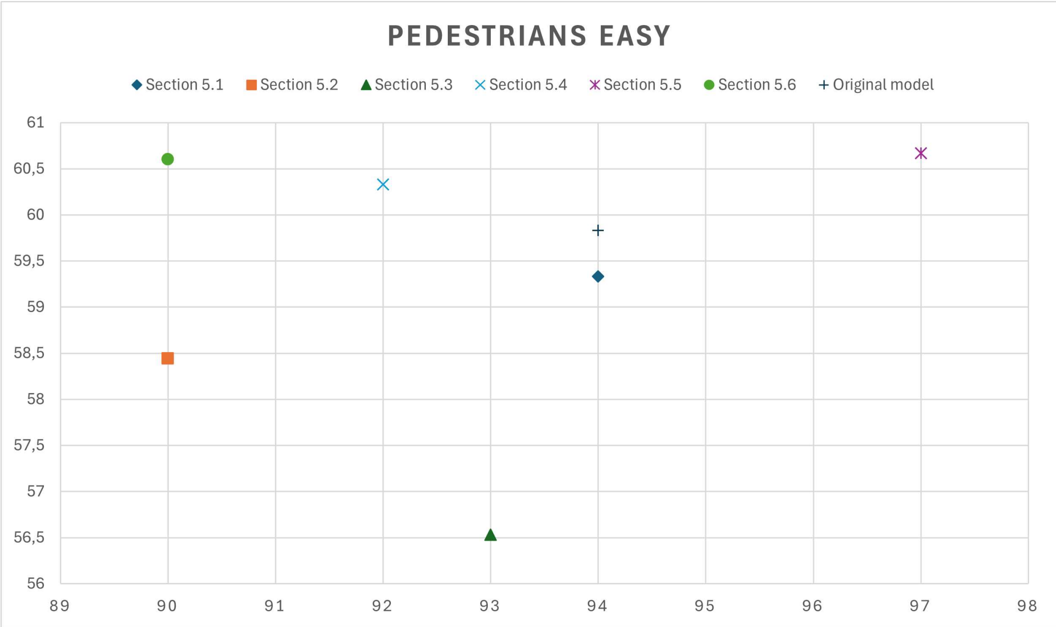

Figure 5.19: Accuracy-delay graph for pedestrians evaluated on easy difficulty dataset

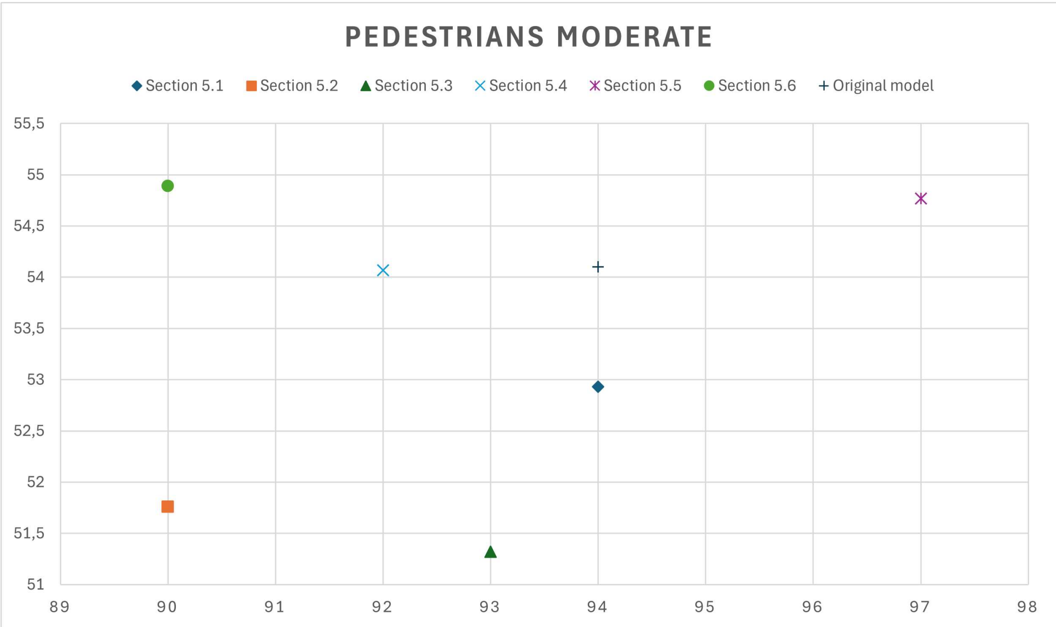

Figure 5.20: Accuracy-delay graph for pedestrians evaluated on moderate difficulty dataset

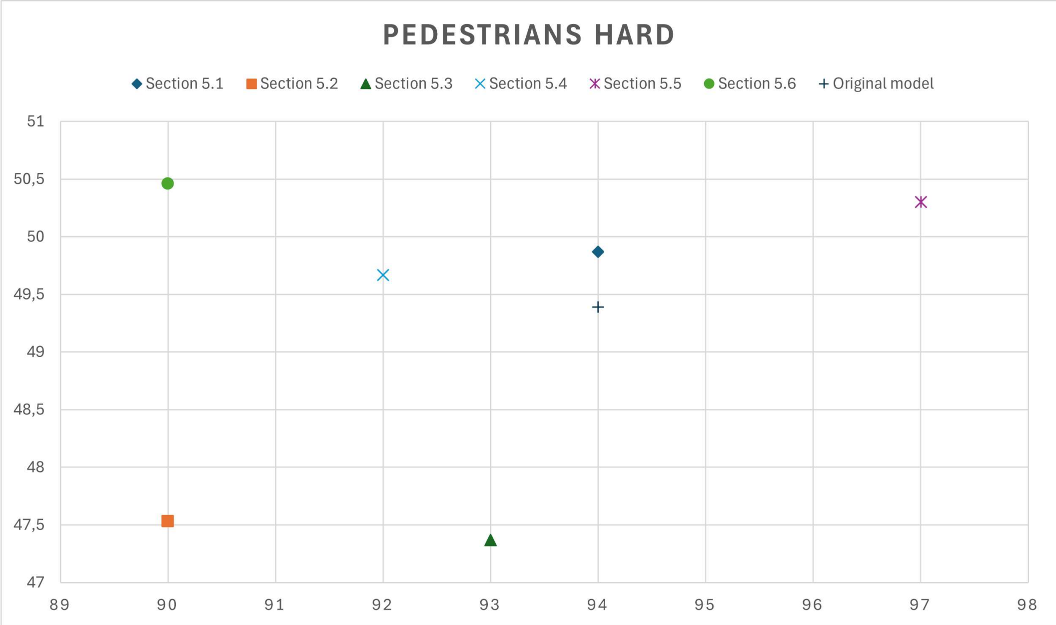

Figure 5.21: Accuracy-delay graph for pedestrians evaluated on hard difficulty dataset

Chapter 6

Configure the embedded system workspace to work with point cloud files and deep learning

We decided in Section 2.2 to work with a Nvidia Jetson Nano[17][18], which will have the following features:

Operating System: Kubuntu 18.04
KDE Plasma Version: 5.12 LTS
Memory: 3.9 GiB of RAM
Processors: ARMv8 Processor rev 1 (v8l) x 4
Graphics Processor: NVIDIA Tegra X1 (nvgpu)/integrated OS type 64-bit
Disk: 14.7 GB
nvcc: NVIDIA (R) Cuda compiler driver
CUDA version: 10.2

6.1 Environment setup

To build the environment, we follow the steps described in the CUDA-PointPillars tutoria [28]. In particular, we execute the commands shown in Listing 6.1.

Listing 6.1: Environment Setup

```
cd CUDA-PointPillars && . tool/environment.sh
mkdir build && cd build
cmake .. && make -j\$(nproc)
cd ../ && sh tool/build\_trt\_engine.sh
cd build && ./pointpillar ../data/ ../data/ --timer
```

When the last command is executed without errors, the environment is built and some outputs are produced as an example of encoded results of object detection in point clouds.
These outputs are text files, which contain numbers representing the coordinates of the predictions. An example is shown in Listing 6.2.
Each of these lines represent an encoded bounding box, in particular:

- Three bounding box center coordinates (x, y, z) in floating point
- Three shift amounts from the box center (dx, dy, dz) in floating point
- The heading (box direction)
- The class (either car, cyclist or pedestrian) represented by an integer number (respectively 0, 1 or 2).
- The score (certainty of the prediction) in a floating point value between 0 an 1

Listing 6.2: Predictions

```
34.8439 -3.1248 -1.42732 4.00212 1.57296 1.46907 6.32502 0 0.849222
15.8563 3.68577 -1.03334 1.70029 0.537016 1.74738 3.06188 2 0.358915
23.747 3.41975 -0.87707 0.866645 0.622608 2.03322 4.38542 1 0.289773
6.48975 3.94685 -0.868623 1.81853 0.576962 1.67918 6.25642 2 0.2587
7.84963 -2.72028 -0.802734 0.849104 0.665763 1.82079 6.11424 1
    0.190122
32.0105 -4.36819 -1.10008 0.748601 0.701814 1.97403 1.87401 1
    0.164248
44.941 1.49198 -1.76096 3.83123 1.6851 1.53399 2.83324 0 0.163445
7.77657 -3.42381 -0.875381 0.832478 0.599927 1.69983 2.81982 1
    0.154695
10.5924 3.96306 -0.828076 0.860478 0.559364 1.86522 4.38151 1
    0.152158
14.4884 3.95217 -0.990264 0.649761 0.660461 1.83227 1.32964 1
    0.146575
38.2134 -4.42721 -1.10261 0.730813 0.564112 1.94177 5.43962 1
    0.131954
60.3677 -5.19389 -0.995332 0.840058 0.559364 1.77506 4.72441 1
    0.119203
34.4795 3.96642 -1.1533 0.480744 0.684887 1.97403 6.36588 1 0.104112
```

This type of encoding could be visualized with some tools, like 3D-Detection-Tracking-Viewe [30], to obtain an image like Figure 6.1.

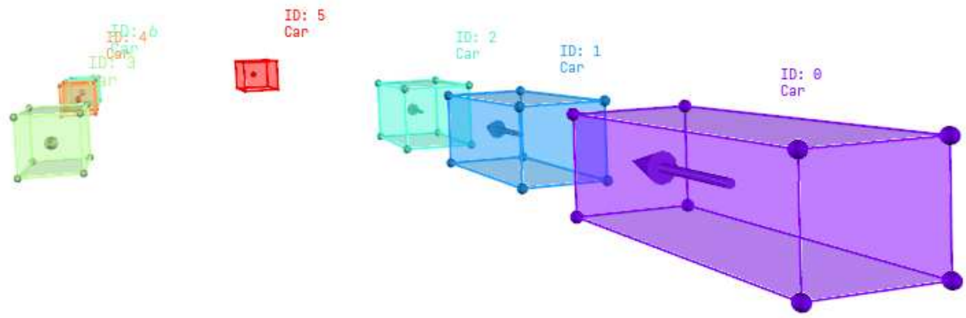

Figure 6.1: Visualization of a decoded prediction

Once the environment is ready, it will be possible to port the algorithm to the Nvidia Jetson Nano [17][18] to work in real-time conditions.

Chapter 7

Conclusion

Object detection in point clouds is increasingly crucial for many applications such autonomous driving. The need of faster and more accurate devices is nowadays of the highest importance for real-time applications.

This study conducted a systematic performance maximization of the PointPillars algorithm, a novel deep network and encoder that can be trained end-to-end on LiDAR point clouds, focusing on reducing the inference time without significantly increasing losses and maintaining accuracy to a proper level. This optimization process was limited to the tuning of number of filters and number of layers in the backbone structure of the model, and every configuration was trained and tested on the KITTI benchmark.

The results are promising, showing that reducing the number of layers in the backbone to [2,4,4] and halving the number of up-sample filters we lower by four seconds the inference time, without significantly lowering the accuracy, and with a negligible increase in loss. This configuration guarantees Pareto optimality (for Figure 5.13, Figure 5.15, Figure 5.19, Figure 5.20 and Figure 5.21, Pareto dominance is obtained).

This study being the first step, future experiments could put into practice the implementation of this resulting compact design on the Nvidia Jetson Nano device, which is a embedded platform suitable for deep learning inference, analysing its power efficiency and performance.

Appendix A

Configuration file

Listing A.1: PointPillar.yaml

```yaml
CLASS_NAMES: ['Car', 'Pedestrian', 'Cyclist']

DATA_CONFIG:
    _BASE_CONFIG_: cfgs/dataset_configs/kitti_dataset.yaml
    POINT_CLOUD_RANGE: [0, -39.68, -3, 69.12, 39.68, 1]
    DATA_PROCESSOR:
        - NAME: mask_points_and_boxes_outside_range
          REMOVE_OUTSIDE_BOXES: True

        - NAME: shuffle_points
          SHUFFLE_ENABLED: {
            'train': True,
            'test': False
          }

        - NAME: transform_points_to_voxels
          VOXEL_SIZE: [0.16, 0.16, 4]
          MAX_POINTS_PER_VOXEL: 32
          MAX_NUMBER_OF_VOXELS: {
            'train': 16000,
            'test': 40000
          }
    DATA_AUGMENTOR:
        DISABLE_AUG_LIST: ['placeholder']
        AUG_CONFIG_LIST:
            - NAME: gt_sampling
              USE_ROAD_PLANE: True
              DB_INFO_PATH:
                  - kitti_dbinfos_train.pkl
              PREPARE: {
```

```
                    filter_by_min_points: ['Car:5', 'Pedestrian:5', 'Cyclist:5'],
                    filter_by_difficulty: [-1],
                }

                SAMPLE_GROUPS: ['Car:15','Pedestrian:15', 'Cyclist:15']
                NUM_POINT_FEATURES: 4
                DATABASE_WITH_FAKELIDAR: False
                REMOVE_EXTRA_WIDTH: [0.0, 0.0, 0.0]
                LIMIT_WHOLE_SCENE: False

              - NAME: random_world_flip
                ALONG_AXIS_LIST: ['x']

              - NAME: random_world_rotation
                WORLD_ROT_ANGLE: [-0.78539816, 0.78539816]

              - NAME: random_world_scaling
                WORLD_SCALE_RANGE: [0.95, 1.05]

MODEL:
    NAME: PointPillar

    VFE:
        NAME: PillarVFE
        WITH_DISTANCE: False
        USE_ABSLOTE_XYZ: True
        USE_NORM: True
        NUM_FILTERS: [64]

    MAP_TO_BEV:
        NAME: PointPillarScatter
        NUM_BEV_FEATURES: 64

    BACKBONE_2D:
        NAME: BaseBEVBackbone
        LAYER_NUMS: [3, 5, 5]
        LAYER_STRIDES: [2, 2, 2]
        NUM_FILTERS: [64, 128, 256]
        UPSAMPLE_STRIDES: [1, 2, 4]
        NUM_UPSAMPLE_FILTERS: [128, 128, 128]

    DENSE_HEAD:
        NAME: AnchorHeadSingle
        CLASS_AGNOSTIC: False

        USE_DIRECTION_CLASSIFIER: True
        DIR_OFFSET: 0.78539
        DIR_LIMIT_OFFSET: 0.0
```

```
        NUM_DIR_BINS: 2

        ANCHOR_GENERATOR_CONFIG: [
            {
                'class_name': 'Car',
                'anchor_sizes': [[3.9, 1.6, 1.56]],
                'anchor_rotations': [0, 1.57],
                'anchor_bottom_heights': [-1.78],
                'align_center': False,
                'feature_map_stride': 2,
                'matched_threshold': 0.6,
                'unmatched_threshold': 0.45
            },
            {
                'class_name': 'Pedestrian',
                'anchor_sizes': [[0.8, 0.6, 1.73]],
                'anchor_rotations': [0, 1.57],
                'anchor_bottom_heights': [-0.6],
                'align_center': False,
                'feature_map_stride': 2,
                'matched_threshold': 0.5,
                'unmatched_threshold': 0.35
            },
            {
                'class_name': 'Cyclist',
                'anchor_sizes': [[1.76, 0.6, 1.73]],
                'anchor_rotations': [0, 1.57],
                'anchor_bottom_heights': [-0.6],
                'align_center': False,
                'feature_map_stride': 2,
                'matched_threshold': 0.5,
                'unmatched_threshold': 0.35
            }
        ]

        TARGET_ASSIGNER_CONFIG:
            NAME: AxisAlignedTargetAssigner
            POS_FRACTION: -1.0
            SAMPLE_SIZE: 512
            NORM_BY_NUM_EXAMPLES: False
            MATCH_HEIGHT: False
            BOX_CODER: ResidualCoder

        LOSS_CONFIG:
            LOSS_WEIGHTS: {
                'cls_weight': 1.0,
                'loc_weight': 2.0,
                'dir_weight': 0.2,
                'code_weights': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

```yaml
            }

    POST_PROCESSING:
        RECALL_THRESH_LIST: [0.3, 0.5, 0.7]
        SCORE_THRESH: 0.1
        OUTPUT_RAW_SCORE: False

        EVAL_METRIC: kitti

        NMS_CONFIG:
            MULTI_CLASSES_NMS: False
            NMS_TYPE: nms_gpu
            NMS_THRESH: 0.01
            NMS_PRE_MAXSIZE: 4096
            NMS_POST_MAXSIZE: 500

OPTIMIZATION:
    BATCH_SIZE_PER_GPU: 4
    NUM_EPOCHS: 80

    OPTIMIZER: adam_onecycle
    LR: 0.003
    WEIGHT_DECAY: 0.01
    MOMENTUM: 0.9

    MOMS: [0.95, 0.85]
    PCT_START: 0.4
    DIV_FACTOR: 10
    DECAY_STEP_LIST: [35, 45]
    LR_DECAY: 0.1
    LR_CLIP: 0.0000001

    LR_WARMUP: False
    WARMUP_EPOCH: 1

    GRAD_NORM_CLIP: 10
```

Appendix B

Python training and evaluating file

Listing B.1: train.py

```python
import _init_path
import argparse
import datetime
import glob
import os
from pathlib import Path
from test import repeat_eval_ckpt

import torch
import torch.nn as nn
from tensorboardX import SummaryWriter

from pcdet.config import cfg, cfg_from_list, cfg_from_yaml_file, log_config_to_file
from pcdet.datasets import build_dataloader
from pcdet.models import build_network, model_fn_decorator
from pcdet.utils import common_utils
from train_utils.optimization import build_optimizer, build_scheduler
from train_utils.train_utils import train_model


def parse_config():
    parser = argparse.ArgumentParser(description='arg parser')
    parser.add_argument('--cfg_file', type=str, default=None, help='specify the config for training')

    parser.add_argument('--batch_size', type=int, default=None, required=False, help='batch size for training')
```

```python
    parser.add_argument('--epochs', type=int, default=None, required=False, help='number of epochs to train for')
    parser.add_argument('--workers', type=int, default=4, help='number of workers for dataloader')
    parser.add_argument('--extra_tag', type=str, default='default', help='extra tag for this experiment')
    parser.add_argument('--ckpt', type=str, default=None, help='checkpoint to start from')
    parser.add_argument('--pretrained_model', type=str, default=None, help='pretrained_model')
    parser.add_argument('--launcher', choices=['none', 'pytorch', 'slurm'], default='none')
    parser.add_argument('--tcp_port', type=int, default=18888, help='tcp port for distrbuted training')
    parser.add_argument('--sync_bn', action='store_true', default=False, help='whether to use sync bn')
    parser.add_argument('--fix_random_seed', action='store_true', default=False, help='')
    parser.add_argument('--ckpt_save_interval', type=int, default=1, help='number of training epochs')
    parser.add_argument('--local_rank', type=int, default=None, help='local rank for distributed training')
    parser.add_argument('--max_ckpt_save_num', type=int, default=30, help='max number of saved checkpoint')
    parser.add_argument('--merge_all_iters_to_one_epoch', action='store_true', default=False, help='')
    parser.add_argument('--set', dest='set_cfgs', default=None, nargs=argparse.REMAINDER,
                        help='set extra config keys if needed')

    parser.add_argument('--max_waiting_mins', type=int, default=0, help='max waiting minutes')
    parser.add_argument('--start_epoch', type=int, default=0, help='')
    parser.add_argument('--num_epochs_to_eval', type=int, default=0, help='number of checkpoints to be evaluated')
    parser.add_argument('--save_to_file', action='store_true', default=False, help='')

    parser.add_argument('--use_tqdm_to_record', action='store_true', default=False, help='if True, the intermediate losses will not be logged to file, only tqdm will be used')
    parser.add_argument('--logger_iter_interval', type=int, default=50, help='')
    parser.add_argument('--ckpt_save_time_interval', type=int, default=300, help='in terms of seconds')
    parser.add_argument('--wo_gpu_stat', action='store_true', help='')
```

```
    parser.add_argument('--use_amp', action='store_true', help='use mix precision training')


    args = parser.parse_args()

    cfg_from_yaml_file(args.cfg_file, cfg)
    cfg.TAG = Path(args.cfg_file).stem
    cfg.EXP_GROUP_PATH = '/'.join(args.cfg_file.split('/')[1:-1])  # remove 'cfgs' and 'xxxx.yaml'

    args.use_amp = args.use_amp or cfg.OPTIMIZATION.get('USE_AMP', False)

    if args.set_cfgs is not None:
        cfg_from_list(args.set_cfgs, cfg)

    return args, cfg


def main():
    args, cfg = parse_config()
    if args.launcher == 'none':
        dist_train = False
        total_gpus = 1
    else:
        if args.local_rank is None:
            args.local_rank = int(os.environ.get('LOCAL_RANK', '0'))

        total_gpus, cfg.LOCAL_RANK = getattr(common_utils, 'init_dist_%s' % args.launcher)(
            args.tcp_port, args.local_rank, backend='nccl'
        )
        dist_train = True

    if args.batch_size is None:
        args.batch_size = cfg.OPTIMIZATION.BATCH_SIZE_PER_GPU
    else:
        assert args.batch_size % total_gpus == 0, 'Batch size should match the number of gpus'
        args.batch_size = args.batch_size // total_gpus

    args.epochs = cfg.OPTIMIZATION.NUM_EPOCHS if args.epochs is None else args.epochs

    if args.fix_random_seed:
        common_utils.set_random_seed(666 + cfg.LOCAL_RANK)

```

```python
    output_dir = cfg.ROOT_DIR / 'output' / cfg.EXP_GROUP_PATH / cfg.TAG / args.extra_tag
    ckpt_dir = output_dir / 'ckpt'
    output_dir.mkdir(parents=True, exist_ok=True)
    ckpt_dir.mkdir(parents=True, exist_ok=True)

    log_file = output_dir / ('train_%s.log' % datetime.datetime.now().strftime('%Y%m%d-%H%M%S'))
    logger = common_utils.create_logger(log_file, rank=cfg.LOCAL_RANK)

    # log to file
    logger.info('**********************Start logging**********************')
    gpu_list = os.environ['CUDA_VISIBLE_DEVICES'] if 'CUDA_VISIBLE_DEVICES' in os.environ.keys() else 'ALL'
    logger.info('CUDA_VISIBLE_DEVICES=%s' % gpu_list)

    if dist_train:
        logger.info('Training in distributed mode : total_batch_size: %d' % (total_gpus * args.batch_size))
    else:
        logger.info('Training with a single process')

    for key, val in vars(args).items():
        logger.info('{:16} {}'.format(key, val))
    log_config_to_file(cfg, logger=logger)
    if cfg.LOCAL_RANK == 0:
        os.system('cp %s %s' % (args.cfg_file, output_dir))

    tb_log = SummaryWriter(log_dir=str(output_dir / 'tensorboard')) if cfg.LOCAL_RANK == 0 else None

    logger.info("----------- Create dataloader & network & optimizer -----------")
    train_set, train_loader, train_sampler = build_dataloader(
        dataset_cfg=cfg.DATA_CONFIG,
        class_names=cfg.CLASS_NAMES,
        batch_size=args.batch_size,
        dist=dist_train, workers=args.workers,
        logger=logger,
        training=True,
        merge_all_iters_to_one_epoch=args.merge_all_iters_to_one_epoch,
        total_epochs=args.epochs,
        seed=666 if args.fix_random_seed else None
    )

```

```python
    model = build_network(model_cfg=cfg.MODEL, num_class=len(cfg.CLASS_NAMES), dataset=train_set)
    if args.sync_bn:
        model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model)
    model.cuda()

    optimizer = build_optimizer(model, cfg.OPTIMIZATION)

    # load checkpoint if it is possible
    start_epoch = it = 0
    last_epoch = -1
    if args.pretrained_model is not None:
        model.load_params_from_file(filename=args.pretrained_model, to_cpu=dist_train, logger=logger)

    if args.ckpt is not None:
        it, start_epoch = model.load_params_with_optimizer(args.ckpt, to_cpu=dist_train, optimizer=optimizer, logger=logger)
        last_epoch = start_epoch + 1
    else:
        ckpt_list = glob.glob(str(ckpt_dir / '*.pth'))

        if len(ckpt_list) > 0:
            ckpt_list.sort(key=os.path.getmtime)
            while len(ckpt_list) > 0:
                try:
                    it, start_epoch = model.load_params_with_optimizer(
                        ckpt_list[-1], to_cpu=dist_train, optimizer=optimizer, logger=logger
                    )
                    last_epoch = start_epoch + 1
                    break
                except:
                    ckpt_list = ckpt_list[:-1]

    model.train()  # before wrap to DistributedDataParallel to support fixed some parameters
    if dist_train:
        model = nn.parallel.DistributedDataParallel(model, device_ids=[cfg.LOCAL_RANK % torch.cuda.device_count()])
    logger.info(f'---------- Model {cfg.MODEL.NAME} created, param count: {sum([m.numel() for m in model.parameters()])} ----------')
    logger.info(model)

    lr_scheduler, lr_warmup_scheduler = build_scheduler(
        optimizer, total_iters_each_epoch=len(train_loader), total_epochs=args.epochs,
```

```python
        last_epoch=last_epoch, optim_cfg=cfg.OPTIMIZATION
    )

    # -----------------------start training---------------------------
    logger.info('**********************Start training %s/%s(%s)**********************'
                % (cfg.EXP_GROUP_PATH, cfg.TAG, args.extra_tag))

    train_model(
        model,
        optimizer,
        train_loader,
        model_func=model_fn_decorator(),
        lr_scheduler=lr_scheduler,
        optim_cfg=cfg.OPTIMIZATION,
        start_epoch=start_epoch,
        total_epochs=args.epochs,
        start_iter=it,
        rank=cfg.LOCAL_RANK,
        tb_log=tb_log,
        ckpt_save_dir=ckpt_dir,
        train_sampler=train_sampler,
        lr_warmup_scheduler=lr_warmup_scheduler,
        ckpt_save_interval=args.ckpt_save_interval,
        max_ckpt_save_num=args.max_ckpt_save_num,
        merge_all_iters_to_one_epoch=args.merge_all_iters_to_one_epoch,
        logger=logger,
        logger_iter_interval=args.logger_iter_interval,
        ckpt_save_time_interval=args.ckpt_save_time_interval,
        use_logger_to_record=not args.use_tqdm_to_record,
        show_gpu_stat=not args.wo_gpu_stat,
        use_amp=args.use_amp,
        cfg=cfg
    )

    if hasattr(train_set, 'use_shared_memory') and train_set.use_shared_memory:
        train_set.clean_shared_memory()

    logger.info('**********************End training %s/%s(%s)**********************\n\n\n'
                % (cfg.EXP_GROUP_PATH, cfg.TAG, args.extra_tag))

    logger.info('**********************Start evaluation %s/%s(%s)**********************' %
                (cfg.EXP_GROUP_PATH, cfg.TAG, args.extra_tag))
    test_set, test_loader, sampler = build_dataloader(
```

```
            dataset_cfg=cfg.DATA_CONFIG,
            class_names=cfg.CLASS_NAMES,
            batch_size=args.batch_size,
            dist=dist_train, workers=args.workers, logger=logger, training=False
        )
        eval_output_dir = output_dir / 'eval' / 'eval_with_train'
        eval_output_dir.mkdir(parents=True, exist_ok=True)
        args.start_epoch = max(args.epochs - args.num_epochs_to_eval, 0)  # Only evaluate the last args.num_epochs_to_eval epochs

        repeat_eval_ckpt(
            model.module if dist_train else model,
            test_loader, args, eval_output_dir, logger, ckpt_dir,
            dist_test=dist_train
        )
        logger.info('**********************End evaluation %s/%s(%s)**********************' %
                    (cfg.EXP_GROUP_PATH, cfg.TAG, args.extra_tag))


if __name__ == '__main__':
    main()
```

Bibliography

[1] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang and O. Beijbom. PointPillars Fast Encoders for Object Detection from Point Clouds.
[2] A. Paigwar, D. Sierra-Gonzalez, O. Erkent and C. Laugier. Frustum-PointPillars: A Multi-Stage Approach for 3D Object Detection using RGB Camera and LiDAR.
[3] L. Zhang, H. Meng, Y. Yan and X. Xu. Transformer-Based Global PointPillars 3D Object Detection Method.
[4] J. Tu, P. Wang and F. Liu. PP-RCNN: Point-Pillars Feature Set Abstraction for 3D Real-time Object Detection.
[5] K. Vedder, and E. Eaton. Sparse PointPillars: Maintaining and Exploiting Input Sparsity to Improve Runtime on Embedded Systems.
[6] L- Fan and L. Li. Detecting Objects in Point Clouds with NVIDIA CUDA-Pointpillars
[7] K. Luo, H. Wu, K. Yi, K. Yang, W. Hao and R. Hu. Towards consistent Object Detection via LiDAR-Camera Synergy.
[8] Y. Zhou and O. Tuzel. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection.
[9] Tensorflow: an end-to-end platform for machine learning. More information at the link `https://www.tensorflow.org/`
[10] PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. More information at the link `https://pytorch.org/`
[11] Open3D: A Modern Library for 3D Data Processing. More information at the link `https://www.open3d.org/`
[12] J. Vandendriessche, N. Wouters, B. da Silva, M. Lamrini, M. Y. Chkouri and A. Touhafi. Environmental Sound Recognition on Embedded Systems: From FPGAs to TPUs.
[13] F. Fusco, M. Vlachos, X. Dimitropoulos and L. Deri. Indexing million of packets per second using GPUs.
[14] R. Bal, A. Bakshi and S. Gupta. Performance Evaluation of Optimization Techniques with Vector Quantization Used for Image Compression.
[15] P. Cao, H. Chen, Y. Zhang and G. Wang. Multi-View Frustum Pointnet for Object Detection in Autonomous Driving.
[16] M. Simony, S. Milzy, K. Amendey and H.M. Gross. Complex-YOLO: An

Euler-Region-Proposal for Real-time 3D Object Detection on Point Clouds
[17] S. Valladares, M. Toscano, R. Tufiño, P. Morillo and D. Vallejo-Huanga. Performance Evaluation of the Nvidia Jetson Nano Through a Real-Time Machine Learning Application.
[18] Nvidia Jetson Nano Developer Kit. More information at the link https://developer.nvidia.com/embedded/jetson-nano-developer-kit.
[19]G. Dinelli, G. Meoni, E. Rapuano, G. Benelli and L. Fanucci. An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick.
[20] E. Dubois. Shared learning among distributed edge devices using coral edge tpu machine learning engines.
[21] J. C. Shovic. Raspberry Pi IoT Projects.
[22] S. A. Bello, S. Yu, C. Wang, J. M. Adam and J. Li. Review: Deep Learning on 3D Point Clouds
[23] Y. Liao, J. Xie and A. Geiger. KITTI-360: A Novel Dataset and Benchmarks for Urban Scene Understanding in 2D and 3D
[24] OpenPCDet Toolbox for LiDAR-based 3D Object Detection. More information at the link https://github.com/open-mmlab/OpenPCDet
[25] TensorBoard: TensorFlow's visualization toolkit. More information at the link https://www.tensorflow.org/tensorboard
[26] Nvidia GeForce RTX 3090. More information at the link https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/rtx-3090-3090ti/
[27] Matlab. Get Started with PointPillars. More information at the link www.mathworks.com/help/lidar/ug/get-started-pointpillars.html
[28] A project demonstrating how to use CUDA-PointPillars to deal with cloud points data from lidar. More information at the link https://github.com/NVIDIA-AI-IOT/CUDA-PointPillars?tab=readme-ov-file
[29] The KITTI Vision Benchmark Suite. More information at the link https://www.cvlibs.net/datasets/kitti/
[30] 3D detection and tracking viewer (visualization) for KITTI and waymo dataset. More information at the link
https://github.com/hailanyi/3D-Detection-Tracking-Viewer
[31] B. Behroozpour, P. A. M. Sandborn, M. C. Wu and B. E. Boser. Lidar System Architectures and Circuits