










Article

Framework for Representing, Building and Reusing Novel State-of-the-art 3D Object Detection Models in Point Clouds Targeting Self-Driving Applications

António Silva^{1,*}, Pedro Oliveira^{1,*}, Dalila Durães^{1,*}, Duarte Fernandes¹, Rafael Névoa², João Monteiro¹, Pedro Melo-Pinto^{1,3}, José Machado^{1,4} and Paulo Novais^{1,4}

- ¹ Algoritmi Centre, University of Minho, Guimarães, Portugal; asilva@algoritmi.uminho.pt (A.S.); pedro.jose.oliveira@algoritmi.uminho.pt (P.O.); dad@di.uminho.pt (D.D.); pedro.melo@algoritmi.uminho.pt (P.M.P.); joao.monteiro@dei.uminho.pt (J.M.); jmac@di.uminho.pt (J.Mac.) pjon@di.uminho.pt (P.N.)
² Bosch Car Multimédia, Braga, Portugal; rafael.nevoa@pt.bosch.com (R.N.)
³ University of Trás-os-Montes and Alto Douro, Vila Real, Portugal; pmelo@utad.pt
⁴ LASI - Intelligent System Associate Laboratory, Guimarães, Portugal
* Correspondence: asilva9116@gmail.com, daliladurães@gmail.com
‡ These authors contributed equally to this work.

Abstract: The rapid development of Deep Learning brought novel methodologies for 3D Object Detection using LiDAR sensing technology. These improvements in precision and inference speed performances lead to notable high performance and real-time inference, which is especially important for self-driving purposes. However, the developments carried by these approaches overwhelm the research process in this area since new methods, new technologies, and software versions lead to different project necessities, specifications and requirements. Moreover, the improvements brought by the new methods may be due to improvements in newer versions of deep learning frameworks and not just the novelty and innovation of the model architecture. Thus, it became crucial to create a framework with the same software versions, specifications and requirements that accommodate all these methodologies and allow the easy introduction of new methods and models. A framework is proposed that abstracts the implementation, reusing and building of novel methods and models. The main idea is to facilitate the representation of state-of-the-art (SoA) approaches and simultaneously encourage the implementation of new approaches by reusing, improving and innovating modules in the proposed framework, which has the same software specifications to allow a fair comparison. This makes it possible to determine if the key innovation approach outperforms the current SoA by comparing models in a framework with the same software specifications and requirements.

Keywords: Autonomous Driving; Deep learning Methods; LiDAR Sensing Technology; 3D Object Detection

1. Introduction

The field of computer vision has seen significant advancements in recent years, particularly in the area of 3D object detection from point cloud data. However, there is still a need for a general representation framework that can be applied to a wide range of 3D object detection tasks, regardless of the specific sensor or application domain. The development verified in recent years of the computational power offered by cutting-edge GPUs allowed the application of deep learning algorithms to detect objects in several domains. One such domain is autonomous driving using Light Detection And Ranging (LiDAR) data, representing a considerable gain in detection efficiency, precision, and inference speed [1].

In recent years, there has been significant progress in 3D object detection models based on LIDAR data for self-driving applications. A multitude of frameworks and projects have been proposed, each with its own unique approach to addressing the challenges of detecting and tracking objects in a 3D environment. However, this diversity also poses a

challenge when it comes to deploying these models for onboard inference in a self-driving vehicle [2,3].

One major issue is the enormous variation in software versions, libraries, and supported platforms, making it difficult to assemble and deploy these models correctly. Additionally, self-driving requirements must be taken into consideration, such as the need for operationalisation with different modules and the limited computational resources available in onboard systems.

Regardless, the 3D object detection models discussed in the literature take point clouds as input and are known to be more complex. These models have a deeper pipeline and process a more significant amount of data. For example, a point cloud usually comprises between 100k-120k [3], where each point holds data related to the Euclidean distance and signal reflection, that is, 128 bits to translate each information of each point.

The literature includes recent research such as [3–6], it has been suggested that the minimum operating requirements for self-driving applications should include an overall class classification of at least 60 mAP and an inference time of less than 100 ms.

In this context, the need for a standardized and optimized framework for 3D object detection based on LIDAR data becomes even more important. Such a framework could simplify the deployment process, enable better interoperability across different systems, and facilitate the development of more efficient and effective self-driving systems.

1.1. Our Contribution

This paper aims to propose a general SoA representation framework for 3D object detection from the point cloud. It supports multiple SoA 3D object detection methods with highly refactored codes for both one-stage and two-stage methods. Also, it enables the implementation and reusing of different approaches with less manual engineering efforts by proposing an abstract way of building object detectors. At the same time, it facilitates the implementation of new methods in each module of the framework. By implementing different SoA we are trying to facilitate a new approach for the scientific community. In this way, it will be possible to offer a framework for real-time testing inference and measure the trade-off between metrics (mAPvs inference time) in one single framework 3D model objects applying for self-driving applications. Therefore, the contributions proposed in this paper are as follows:

- An abstract framework for the implementation/representation of edge for 3D object detection models using LIDAR data.
- Less engineer effort to implement new methods in different framework models.
- A simpler way to change hyperparameters and retrain models using YML files.
- An easily represented model using these YML files automatically.

The organisation of this paper is as follows: In the next Section 2, some of the state-of-the-art works related to 3D object detection systems and hardware platforms for their implementation are presented. Section 3 shows a four-step methods used to select, train, and tune a deep learning model for deployment on a hardware device. The following section, 4, presents the selected 3D object detection model, as well as its deep learning components, specifying the details about the architecture of the target hardware device and the implementation of the hardware components and software. The presentation of performance evaluation results, comparison of results and discussion of these results occur in Section 7. Finally, Section 8 presents the main results achieved in this paper and future work.

2. Related Work

In recent years, object detection models in point cloud presented in the literature have been highly improved, and more and more detection performance has been achieved. Based on the literature, the most discussed models are divided into two broad categories: approaches based on CNN 3D and approaches based on CNN 2D, where different data

representations, backbone networks and multiscale resource learning techniques can be adopted [3].

When it comes to 3D object detection approaches, they can be classified into three types. The first category is based on volumetric representation. The second is based on Pillars. Finally, the third is based on raw points. Furthermore, they are novel models recognised by the scientific community that provide innovation in the diverse architecture pipeline, high accuracy and performance in 3D object detection.

The first category, which can be divided into one-stage or two-stage, is usually based on the volumetric representation to discretize the point cloud. The one-stage representation only has a single stage, and SECOND [7] is an example. This 3D convolution-based technique produces item class prediction, bounding box regression, and orientation classification. The two-stage representation got the same results as the single stage but fine-tuned the bounding box. Examples of two-stage are P-RCNN [8], VoxelRCNN [9] and *PointA²* [10]. Usually, these methods require more resources in terms of computing power because they either use the costly volumetric representation of the point cloud or rely on computationally intensive 3D convolutions.

The second category of models fall under one-stage methods and use 2D convolutions in place of the computationally intensive 3D convolutions. PointPillar [11] is an example of this approach. To decrease the high computational cost of handling 3D LiDAR data, these models usually compress the data into a 2D projection or organize it into Pillars [11]. While these methods are quicker and suitable for real-time applications, they sacrifice detection capabilities by losing some information. This highlights the trade-off between inference time and accuracy.

The third category of methods, such as Point RCNN [12], utilizes a two-stage approach based on raw point data and voxel representation to take advantage of their respective benefits. In the first stage, the network uses voxel representation as input and performs light convolutional operations, which results in a small number of high-quality initial predictions. An attention mechanism effectively combines coordinate and indexed convolutional features of each point in the initial forecast, maintaining both accurate localization and contextual information. The second stage uses the fused feature of interior points to refine the prediction [13].

3. Methodology

To implement/represent the 3D Object Detection models based on Deep Learning in the framework, we employed a three-step methodology, which is depicted in Figure 1. (1) Firstly, a set of model architecture and hyper-parameter specifications are defined in different configuration files. These files define the specifications of the components of each module in the framework (described in section 4) as well as the training and test specifications that are then used to build, train and test the object detectors. We chose the models for 3D Object Detection based on a review of existing literature, which is outlined in section 2 and elaborated further in [3]. The framework, described in the 4 section, was developed to facilitate the representation of any Object Detection model.

Once the object detector is built, it is subjected to a training and evaluation pipeline (2), where various optimizations can be performed to enhance the accuracy metrics and fulfill the inference time requirements. In our project, since different components need to operate simultaneously, such as the SLAM algorithm and object detector, we define an overall mAP of 60% and an inference time of less than 100ms (metrics are always subject to trade-offs). The training and evaluation step can be done by changing the training specification in the respective model configuration. The concept behind defining the training and testing parameters in these configuration files is to make it easier to modify them and subsequently submit the object detector to the same training and evaluation pipeline. The pipeline was executed on a server-side node with an Intel Core i9 processor, 64GB of RAM, and a Quadro RTX 8000 GPU. Therefore, the proposed workflow follows an iterative process, where the model is fine-tuned. The training and evaluation steps are repeated whenever necessary

until it meets the requirements and satisfies the application requirements. The evaluation and comparison process is carried out using KITTI benchmarks using the validation set on the aforementioned server node. In conclusion, this workflow guarantees that the models meet the application requirements and attain the highest possible accuracy. This procedure identifies a group of potential Object Detection models for the subsequent step.

After completing step (2) workflow, a comparison phase of the resulting models (step 3) is conducted to select the model that can ensure a better balance between precision and inference time. The subsequent section presents information on the architecture of the framework, the chosen Deep Learning models, and the parameters used in the fine-tuning process.

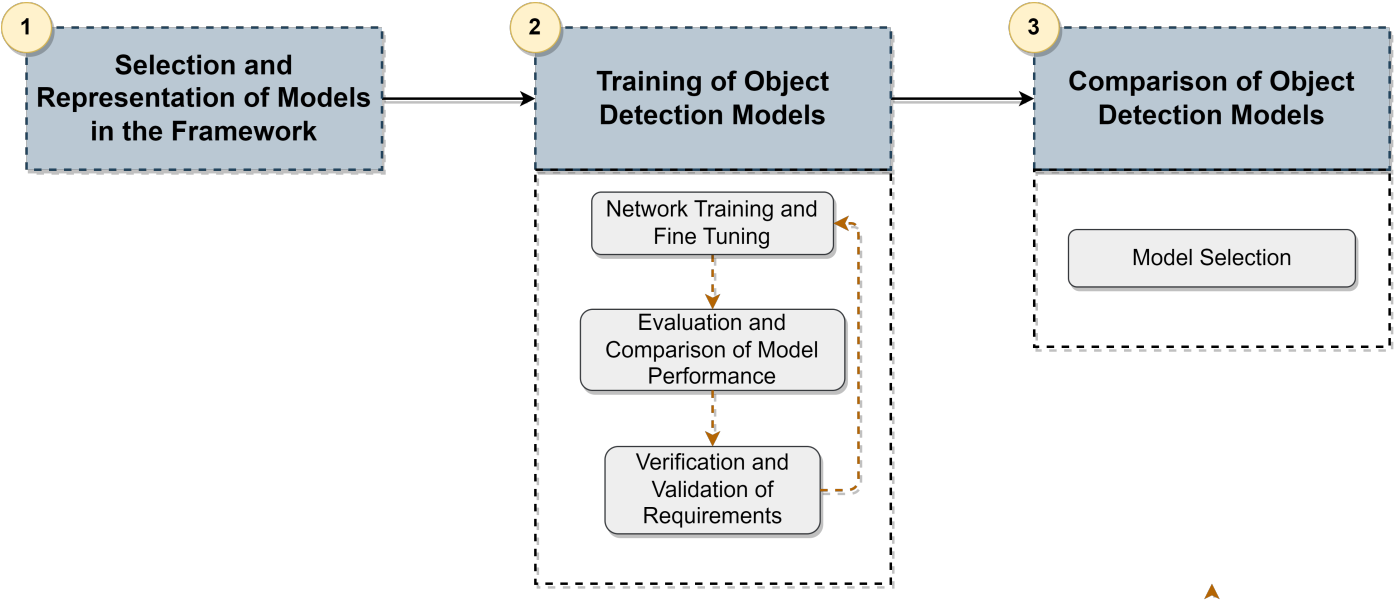


Figure 1. Methodology for object detection model fine-tuning.

4. Framework for Representing 3D Object Detection Models

Our framework’s key innovation is that it facilitates the representation of any object detector through YAML configuration files that define their module specifications in each framework component. Moreover, this framework, shown in Figure 2, aims to facilitate the implementation and integration of new modules in each framework component to allow the comprehensive representation of the different state-of-the-art 3D object detectors.

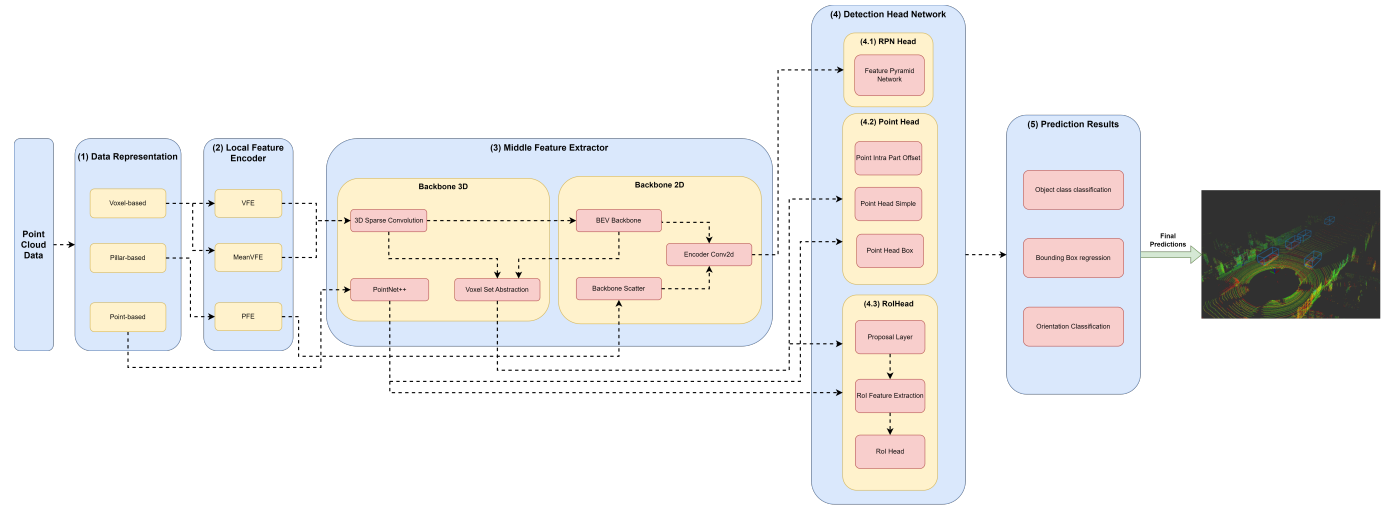


Figure 2. Framework used for the implementation/representation of Object Detection models.

The first component, (1) Data Representation, receives the set of points and discretises them in a set of data structures, such as Pillars or Voxels or only passes the set of points to be used by the Middle Extractor module (3). (2) The local feature encoder receives as input these data structures, more specifically, the set of Pillars or Voxels and encodes and concatenates their features. Then, in the Middle Extractor (3), 3D and/or 2D Backbones extract features from local encoded features, which are used by the (4) Detection Head to predict object class, bounding box offsets and direction (5). (4.1) This Detection Head based on RPN can be assisted by two modules, a (4.2) Point Head module and (4.3) Region of Interest (RoI) Head module, which refines the predicted bounding box offsets and orientation. (4.2) Point Head module is composed of three networks: a Point Intra Part Offset Head [10], a point-based segmentation head for keypoint segmentation [14], and another point-based segmentation head based on [12]. The (4.3) RoiHead module is defined for each state-of-the-art model based on their specificities, but typically it is composed of a Proposal Layer, which proposes a set of RoIs, a RoI Feature Extraction that pooled the RoI features and a RoI Head that predicts RoI class and bounding box offsets.

4.1. Point Cloud Data Representation

We receive an unordered set of points $PC = \{p_1, p_2, p_3 \dots p_n\}$, where $n > 0$ and each point p is represented as (p^x, p^y, p^z, p^r) , where p^x, p^y and p^z correspond to coordinates in the three-dimension cartesian axis and p^r is the reflectance value provided by the LiDAR sensor. A point cloud range PC_R is a tuple (L, H, W) , where L consists of (x_{min}, x_{max}) , H consists of (y_{min}, y_{max}) , and W consists of (z_{min}, z_{max}) . We denote a point cloud subset with respect to PC_R as $PC_R = \{p_i : p_i \in PC, x_{min} \leq p_i^x \leq x_{max}, y_{min} \leq p_i^y \leq y_{max}, z_{min} \leq p_i^z \leq z_{max}\}$.

4.1.1. Pillar Representation

The framework receives the points in PC_R and discretises them in the X-Y axis thus creating a set of pillars $PL_p = \{pl_1, pl_2, pl_3, \dots, PL_p\}$, where $p = mp$, mp is the max number of Pillars and $mp \in \mathbb{N}^+$. Each PL_p has a fixed size in PC_R , and it is represented by a tuple $S_{PL_p} = (w, h)$, where w is the width of the Pillar along the x axis, and h is the height of the Pillar along the y axis. The points are grouped accordingly with the Pillar that resides.

To deal with the sparsity problem and save computation, a max number of points per pillar NP is defined. The points are randomly sampled if the number of points in each pillar is higher than NP . On the other hand, zero padding is added in case of less than NP points.

4.1.2. Voxel-based Representation

The voxelisation process assumes a similar way as proposed in Pillar discretisation; however, the received points are discretised in the X-Y-Z axis. It allows the creation of a set of voxels $VL_j = \{vl_1, vl_2, vl_3 \dots vl_j\}$, where $j = mv$, mv means the max number of Voxels and $mv \in \mathbb{N}^+$ and each VL_j assumes a fixed size in PC_R , and a tuple represents it $S_{VL_j} = (w, h, d)$. w is the width of the voxel along the x axis, h is the height of the voxel along the y axis, and d is the depth of the voxel along the z axis.

A Random Sampling strategy is also applied to save computation, and a max number of points per voxel NV is also used. The strategy to sample points or apply zero padding is the same as the Pillar Representation.

4.1.3. Point-based

The idea in the point-based strategy is to pass the cropped point cloud, herein denoted as PC_R , to the Middle Feature Encoder.

4.2. Local Feature Encoder

The Local Feature Encoder receives the data representation structures DS , such as Pillars denoted as PL , Voxels VL or just the set of points of cropped area PC_R . Then, a set

of methods are applied to obtain features and produce dense tensors in the case of Pillar Feature Network (PFN) and Voxel Feature Encoder (VFE) or calculate these features by simply calculating the mean values of point coordinates within each voxel using Mean VFE method.

4.2.1. Pillar Feature Network

The features of each Pillar, PL , are augmented in a tensor $D = (x, y, z, r, x_c, y_c, z_c, x_{plc}, y_{plc})$, where c describes the distance to the arithmetic mean of all points in PL , and plc is the offset distance from the $Pl_{x,y}$ center.

For this purpose, (1) Pillar Feature Network (PFN) receives the Pillar augmented features as input and applies linear transformations to each point, herein described as $linear(Pl_{in}) = Pl_{out}$, where Pl_{in} corresponds to the initial tensor $Pl_{in} = (P, N, D_{in})$, and Pl_{out} to the output tensor. In Pl_{out} , all but the last dimension are the same shape as the input. Dimension D_{out} results from the linear transformation of D_{in} , thus producing $Pl_{in} = (P, N, D_{out})$. Then, Batch-Norm and ReLU are applied to this tensor. Afterwards, all resulting features are aggregated. This process allows generating a dense tensor to represent the Pillar as a tuple (D, P, N) , where D is the above-mentioned augmented point, P is the number of non-empty pillars per batch, and N is the number of points per Pillar. Next, max pooling operations over the channels are used to generate a tensor of size (D_{out}, P) .

4.2.2. Voxel Feature Encoder

Similar to PFN, the points in each Voxel, $VL_j = \{pt_i = (x_i, y_i, z_i, r_i) \in \mathbb{R}^4\}, i = \{1, 2, \dots, NV\}$, are augmented by calculating offset distance of the point to the $VL_{x,y,z}$ center herein denoted as vlc , which generates the tensor $VL_j = \{pt_i = (x_i, y_i, z_i, r_i, x_{vlc}, y_{vlc}, z_{vlc}) \in \mathbb{R}^7\}, i = \{1, 2, \dots, NV\}$, where NV as mentioned before is the max number of points per Voxel. Afterwards, each pt_i is subject VFE Layers $VFEL_l$, where $l \geq 1$. Each $VFEL_l$ is composed by a set of transformations, where linear transformation, Batch-Norm, and ReLU are applied. Then, all points features of VL_j , resulting from the above-mentioned transformations, herein described as pf_j , are aggregated. Each pf_j can be described as $pf_j \in \mathbb{R}^{out}$, where out is the output dimension that results from the linear transformation of all points pt_i . The output size out resulting from the linear transformation can be described as $out_l = F_o/2$, where $F_o = \{f_1, f_2, \dots, f_o\}, F_o \in \mathbb{N}^+$ means the output features of a specific $VFEL$ index l . After, all point features $PF, pf_j \in PF$ are subject to a max pooling operation over the channels. The output tensor is described as $pfr_m \in \mathbb{R}^{out}$ where $m = 1$. Afterwards, a repeat process of the above tensor is performed $repeat(pfr, k)$ in VL_j , which means the repeat point feature resulted from max pooling k times, where $k = \{1, 2, 3, \dots, NV\}$. Each pfr_k is augmented with pf_j to generate $pfo_j = (pfr_k, pf_j) \in \mathbb{R}^{2out}, k = \{1, 2, \dots, NV\}$ and $j = \{1, 2, \dots, NV\}$. The set of features for each voxel can be described by the tuple $VL_{out} = \{VL_j = stack(pfo_j)\}$, where $j = \{1, 2, 3, \dots, NV\}$, $stack = (pfo_1 \times pfo_2 \times \dots pfo_j)$, and applies linear, Batch-Norm, ReLU and max pooling to each VL_j . Thus, $VL_j \in \mathbb{R}^F$, means that VL_j has output dimension F , the output feature of the last VFE layer.

Finally, it generates a list of obtained voxel features VLA_{out} , $VLA_{out} = \{VL_j = \{vl_1, vl_2, \dots, vl_j\}, VL_j \in \mathbb{R}^F, F = f_o$, where VLA_j is the above-mentioned augmented features of all voxels.

4.2.3. Mean Voxel Feature Encoder

Mean VFE receives a set of Voxels VL , sums all points residing in each Voxel in a specific axis, and divides by the number of points of each one. This operation can be described as $VLM_{out} \triangleq \{mean(vl_j)_{k=0}^{nv} = \{ptf_i = (\sum(ptx)_{i=1}^{nv} | count(ptx), \sum(pty)_{i=1}^{nv} | count(pty), \sum(ptz)_{i=1}^{nv} | count(ptz), \sum(ptr)_{i=1}^{nv} | count(ptr)) \in \mathbb{R}^4\}, k = [0, NV], count(ptx) \rightarrow ptx \in vl_j, count(pty) \rightarrow pty \in vl_j, count(ptz) \rightarrow ptz \in vl_j, count(ptr) \rightarrow ptr \in vl_j. nv$ corresponds to the total number of points of the Voxel vl_j in VL in a given axis, mv the max number of Voxels and $ptf_i \in VLPF$ corresponds to a resulting point. This strategy considers the voxel-wise features a new Voxel center $VL_{x,y,z,r}$ and approximately

equivalent to raw point cloud data. The idea herein is to process the voxel-wise features in Middle Feature Encoder more efficiently, especially by the 3D sparse convolutions since it generates mv (max number of voxels as described in section 4.1) number of non-empty Voxels.

4.3. Middle Feature Extractor

The (3) Middle Feature Extractor is responsible for extracting more features from the (2) Local Feature Encoders to provide more context for the shape description of objects for the networks of the *Detection Head* module. Various methods are used; herein, we separated into 3D Backbones and 2D Backbones, which will be described in more detail above.

4.3.1. Backbone 3D

A variety of methods resort to 3D backbones based on sparse convolutions (sparse CNN), such as SECOND (Figure 4), PV-RCNN (Figure 6), PartA² (Figure 7) and Voxel-RCNN (Figure 8). In particular, PV-RCNN uses a Voxel Set Abstraction 3D backbone, which is used to encode the multiscale semantic features obtained by sparse CNN to *keypoints*. PointRCNN (Figure 5) uses PointNet++ [8] with multiscale grouping for feature extraction and gets more context to the shape of objects and then passes these features to (4) Detection Head module.

3D Sparse Convolution

The 3D Sparse Convolution method receives the voxel-wise features of VFE, VLA_{out} or Mean VFE, VLM_{out} .

This backbone is represented as a set of blocks BLC , in the form $\{blc_1, blc_2, \dots, blc_m\}$, where $m = 6$. Each block $blc_j \in BLC, j = m$ can be defined by a set of Sparse Sequential operations denoted as $SSQ_s = \{ssq_1, ssq_2, ssq_3, \dots, ssq_s\}, s \geq 1$. Each SSQ_s is described by $((SuM \rightarrow \neg SpC) \vee (SpC \rightarrow \neg SuM)), Bn, RL$, where SuM means Submanifold Sparse Convolution 3D [15], SpC Spatially-sparse Convolution 3D [16], Bn 1D Batch Normalisation operation, and RL represents ReLU method. The last method assumes the standard procedure as mentioned in [17].

In our framework, the set of blocks assumes the following configurations:

- The input block blc_1 can be described by $blc_1 = \{sq_1 = (SuM, BN, RL)\}$;
- The next block is represented in the form $blc_2 = \{sq_1 = (SuM, BN, RL)\}$;
- The block 3 is represent as $blc_3 = \{sq_1 = (SpC, BN, RL), sq_2 = (SuM, BN, RL), sq_3 = (SuM, BN, RL)\}$;
- The block 4 is denoted as $blc_4 = \{sq_1 = (SpC, BN, RL), sq_2 = (SuM, BN, RL), sq_3 = (SuM, BN, RL)\}$;
- The block 5 is denoted as $blc_4 = \{sq_1 = (SpC, BN, RL), sq_2 = (SuM, BN, RL), sq_3 = (SuM, BN, RL)\}$;
- The last block is defined by $blc_6 = \{sq_1 = (SpC, BN, RL)\}$.

The Batch Normalisation Bn element is defined by (InB, ep, mn) , which represents the formula in [18]. InB represents the input features, which are the output features of Submanifold Sparse or Spatially-sparse Convolutions 3D, so that $(OutS \rightarrow \neg OutM \vee OutM \rightarrow \neg OutS)$. ep represents the eps, and mn the momentum values. These values are defined in the following Table 1.

Table 1. Values used in Bn .

Bn Element	Value
ep	0.001
mn	0.01

The element SpC can be represented as $(InS, OutS, KsS, StS, PdS, DIS, OpS)$. InS represents the input features of SpC and it is denoted as $SpC \in \mathbb{N}^+$, $InS = OutM$ where $OutM$ represents the output features of Submanifold Sparse Conv3D. The element $OutS$

represents the output features resulting from applying SpC . KsS means kernel size of a Spatially-sparse Convolution 3D and it is denoted as $KsS_s = \{KsS_1, KsS_2, KsS_s\}$ where $s = \{1, \dots, 3\}$, $KsS_s \in \mathbb{N}^+$ and $ksS_s = ksS_{s+1}$. The stride StS can be described as a set $StS_r = \{sts_1, sts_2, sts_r\}$, $r = \{1, \dots, 3\}$, $StS_r \in \mathbb{N}^+$ and $sts_r = sts_{r+1}$. PdS designates padding, and a set can define it $PdS_v = \{pds_1, pds_2, pds_v\}$, $v = \{1, \dots, 3\}$, $Pd_v \in \mathbb{N}^+$, $pds_v = pds_{v+1}$. DlS means dilation and can be defined as a set $DlS_l = \{dls_1, dls_2, dls_l\}$, $l = \{1, \dots, 3\}$, $DlS_l \in \mathbb{N}^+$, $dls_l = dls_{l+1}$. The output padding OpS is represented as a in the form $OpS_a = \{ops_1, ops_2, ops_a\}$, $a = \{1, \dots, 3\}$, $OpS_a \in \mathbb{N}^+$ and $ops_a = ops_{a+1}$. The configurations used in our framework are represented in Table 2.

Table 2. Configurations used in SpC for each element.

SpC Element	Value
KsS_t	3
StS_r	1
PdS_v	1
DlS_l	1
OpS_a	0

SuM is represented by $(InM, OutM, ksM, StM, PdM, DlM, OpM)$ [15]. InM represents the input features passed by (2) Local Feature Encoder or by the last Sparse Sequential block Sq_s , and $OutM$ the output features of SuM . Thus, $InM \in \mathbb{N}^+$, $InM = 4$ in the case of the Local Encoder be Mean VFE, otherwise $In = F$, where F are the output features of the VFE network. Also, InS can be represented by $InM = OutM$ and $InM = OutS$, where $OutS$ represents the output features of a SpC . The element Ks represents the kernel size that can be defined as $Ks_t = \{ks_1, ks_2, ks_t\}$ where $t = \{1, \dots, 3\}$, $Ks_t \in \mathbb{N}^+$ and $ks_t = ks_{t+1}$. StM means stride and can be defined as a set $St_r = \{st_1, st_2, st_r\}$, $r = \{1, \dots, 3\}$, $St_r \in \mathbb{N}^+$ and $st_r = st_{r+1}$. PdM represents padding, and a set can describe it in the form $Pd_p = \{pd_1, pd_2, pd_p\}$, $p = \{1, \dots, 3\}$, $Pd_p \in \mathbb{N}^+$, $pd_p = pd_{p+1}$. Dl means dilation and can be described as a set $Dl_d = \{dl_1, dl_2, dl_d\}$, $d = \{1, \dots, 3\}$, $Dl_d \in \mathbb{N}^+$, $dl_d = dl_{d+1}$. Op represents the output padding, and a set describes it in the form $Op_u = \{op_1, op_2, op_p\}$, $u = \{1, \dots, 3\}$, $Op_u \in \mathbb{N}^+$ and $op_u = op_{u+1}$. The configurations used in our framework are represented in Table 3.

Table 3. Configurations used in SuM and SpC for each block. N.A. - Not applicable.

SuM Element	InS	OutS	InM	OutM	Ks	St	Pd	Dl	Op
$b1c_1 \wedge sq1 \rightarrow SuM$	N.A.	N.A.	4	16	3	1	1	1	0
$b1c_2 \wedge sq1 \rightarrow SuM$	N.A.	N.A.	16	16	3	1	0	1	0
$b1c_3 \wedge sq1 \rightarrow SpC$	16	32	N.A.	N.A.	3	2	1	1	0
$b1c_3 \wedge sq2 \rightarrow SuM$	N.A.	N.A.	32	32	3	1	0	1	0
$b1c_3 \wedge sq3 \rightarrow SuM$	N.A.	N.A.	32	32	3	1	0	1	0
$b1c_4 \wedge sq1 \rightarrow SpC$	32	64	N.A.	N.A.	3	2	1	1	0
$b1c_4 \wedge sq2 \rightarrow SuM$	N.A.	N.A.	64	64	3	1	0	1	0
$b1c_4 \wedge sq3 \rightarrow SuM$	N.the A.	N.A.	64	64	3	1	0	1	0
$b1c_5 \wedge sq1 \rightarrow SpC$	64	64	N.A.	N.A.	3	2	0	1	0
$b1c_5 \wedge sq2 \rightarrow SuM$	N.A.	N.A.	64	64	3	1	0	1	0
$b1c_5 \wedge sq3 \rightarrow SuM$	N.A.	N.A.	64	64	3	1	0	1	0
$b1c_6 \wedge sq1 \rightarrow SpC$	64	128	N.A.	N.A.	3	2	0	1	0

The hyperparameters used in each $b1c_j$ are defined in Table 7.

Finally, the output spatial features SP is defined by $SP \in \mathbb{R}$, where SP is defined by a tuple (B, C, D, H, W) . B represents the batch size, C the output features of blc_5 represented in SpC as $OutS$, D depth, H height and W width.

PointNet++

We use a modified version of PointNet++ [8] based on [12] to learn undiscretised raw point cloud data (herein denoted as PC_R) features in multi-scale grouping fashion. The objective is to learn to segment the foreground points and contextual information about them. For this purpose, a Set Abstraction module herein denoted as SA_M is used to sub-sampling points at a continuing increase rate, and a Feature Proposal module, described as FP_M , is used to capture feature map per point with the objective of point segmentation and proposal generation. A SA_M is composed by $SA_M = \{ptn_1, ptn_2, \dots, ptn_g\}, g \in \mathbb{N}^+, g = \{1, 2, \dots, 4\}$, where ptn means PointNet Set Abstraction module operations. Each $ptn_g \in PTN$ is represented by (QGL, ML) , where QGL corresponds to Query and Grouping operations to learn multi-scale patterns from points, and ML are the set of specifications of the PointNet before the global pooling for each scale.

QGL means ball query operation QL followed by a grouping operation GL . It can be defined by the set $\{qgl_1, qgl_2\}$ where qgl_1 and qgl_2 correspond to two query and group operations. A ball query QL is represented as (R, NS, P, CP) , where R means the radius within all points will be searched from the query point with an upper limit $NS, NS \in \mathbb{N}^+$, in a process called ball query, P means the coordinates of the point features in the form $PF = \{pf_n = (x_n, y_n, z_n) \in \mathbb{R}^3\}, n \in \mathbb{N}$ that are used to gather the point features, CP represents the coordinates of the centers of the ball query in the form $CP = \{cp_p = (xc_p, yc_p, zc_p) \in \mathbb{R}^3\}, p \in \mathbb{N}^+, p \leq n, p = \{1, 2, \dots, 4\}$, where xc , yc , and zc are center coordinates of a ball query. Thus, this ball query algorithm search for point features P in a radius R with an upper limit of NS query points from the centroids (or ball query centers) CP . This operation generates a list of indices ID in the form $\{id_1, id_2, \dots, id_x\}, x \geq 1, id_x \in ID, id_x \in \mathbb{N}^{NCP \times NS}$, where NCP corresponds to the number of CP . ID represents the indices of point features that form the query balls. After, a grouping operation GL is performed to group point features and can be described by (PF, ID) , in which PF and ID correspond to point features and indices of the features to group with, respectively. In each QGL of a ptn , the number of centroids NCP will decrease, so that $NCP_p > NCP_{p+1}, p = \{1, 2, \dots, 4\}, NCP \in \mathbb{N}^+$, and due to the relation of the centroids in ball query search, the number of indices NID and corresponding point features will also decrease. Thus, in each ptn the number of points features is defined by $NP_n > NP_{n+1}, NP_{n+1} = NCP_p, p > 1$. The number of centroids defined in QGL during ptn operations is defined in Table 4.

Table 4. Configurations used in NCP for each element

NCP Element	Value
ncp_1	4096
ncp_2	1024
ncp_3	256
ncp_4	64

Afterwards, a ML is performed, defined by a set of specifications of the PointNet before the QGL operations. The idea herein is to capture point-to-point relations of the point features in each CP local region. The point features coordinates translation to the local region relative to the centroid point is performed by the operation $LR = \{fr_f = (px_f - xc_f, py_f - yc_f, pz_f - zc_f) \in \mathbb{R}^3\}, f = \{1, 2, \dots, NS\}$. px , py , and pz are coordinates of point features PF as mentioned before, and xc , yc , and zc are coordinates of the centroid center. ML can be defined by a set $SQ = \{sq_1, sq_2\}$ that represent two sequential methods. Each SQ is represented by the set of operations $OP = \{ops = (C2D, Bn2D, RL)\}, s = \{0, 1, \dots, 3\}$, where $C2D$ means Convolution 2D, $Bn2D$ 2D Batch Normalisation, and RL represents the ReLU method. $C2D$ is defined by $(InC2D, OutC2D, KsC2D, SC2D)$.

$InC2D$, where $InC2D \in \mathbb{N}^+$, represents the input features that can be received by QGL or by the output features $OutC2D, OutC2D \in \mathbb{N}^+$ of the op_{s-1} , $KsC2D$ the kernel size, and $SC2D$ the stride of the Convolution 2D. The kernel size $KsC2D$ is defined by the set $\{ksc2d_1, ksc2d_2\}, ksc2d_1 = ksc2d_2$ and $\forall op_s \in SQ, SQ \in ML, ML \in PTN, ksc2d_1 = 1$. Also, the stride $SC2D$ is represented by a set $\{sc2d_1, sc2d_2\}, sc2d_1 = sc2d_2$, and $sc2d_1 = 1$, with $\forall op_s \in SQ, SQ \in ML, ML \in PTN$. The set of specifications used in our models regarding OP are summarised in Table 5. $ptn_i \in PTN$ can be defined as:

$$PTN \triangleq \{ptn_i = \max(ML(SG(pf_i)))\} \quad (1)$$

, where \max denotes max pooling, SG denotes random sampling of pf_i features, ML multi-layer perceptron network to encode features and relative locations.

Table 5. Set of configurations used in OP of a specific SQ of the ML element in a specific PTN .

NCP Element	InC2D	OutC2D
$op_1 \wedge sq_1 \wedge ptn_1$	4	16
$op_2 \wedge sq_1 \wedge ptn_1$	16	16
$op_3 \wedge sq_1 \wedge ptn_1$	16	32
$op_1 \wedge sq_2 \wedge ptn_1$	4	32
$op_2 \wedge sq_2 \wedge ptn_1$	32	32
$op_3 \wedge sq_2 \wedge ptn_1$	32	64
$op_1 \wedge sq_1 \wedge ptn_2$	99	64
$op_2 \wedge sq_1 \wedge ptn_2$	64	64
$op_3 \wedge sq_1 \wedge ptn_2$	64	128
$op_1 \wedge sq_2 \wedge ptn_2$	99	64
$op_2 \wedge sq_2 \wedge ptn_2$	64	96
$op_3 \wedge sq_2 \wedge ptn_2$	96	128
$op_1 \wedge sq_1 \wedge ptn_3$	259	128
$op_2 \wedge sq_1 \wedge ptn_3$	128	196
$op_3 \wedge sq_1 \wedge ptn_3$	196	256
$op_1 \wedge sq_2 \wedge ptn_3$	259	128
$op_2 \wedge sq_2 \wedge ptn_3$	128	196
$op_3 \wedge sq_2 \wedge ptn_3$	196	256
$op_1 \wedge sq_1 \wedge ptn_4$	515	256
$op_2 \wedge sq_1 \wedge ptn_4$	256	256
$op_3 \wedge sq_1 \wedge ptn_4$	256	512
$op_1 \wedge sq_2 \wedge ptn_4$	515	256
$op_2 \wedge sq_2 \wedge ptn_4$	256	384
$op_3 \wedge sq_2 \wedge ptn_4$	384	512

Finally, a Feature Proposal FP_M is applied employing a set of feature proposal modules $\{fp_1, fp_2, \dots, fp_m\}, m = \{1, 2, \dots, 4\}, m \in \mathbb{N}^+$. Each $fp_m \in FP_M$ is defined by the element SQ as defined above. Also, the element SQ assumes a set $\{sq_1, sq_2\}$ and each SQ has the same operations with the only difference in the element s that describes the number of operations, assuming $s = \{1, 2\}$ instead of $s = \{1, 2, 3\}$. The configurations used in our models are summarised in Table 6.

Table 6. Set of configurations used in *OP* of a specific *SQ* in a specific *FP_M*.

<i>NCP Element</i>	<i>InC2D</i>	<i>OutC2D</i>
$op_1 \wedge sq_1 \wedge fp_1$	257	128
$op_2 \wedge sq_2 \wedge fp_1$	128	128
$op_1 \wedge sq_1 \wedge fp_2$	608	256
$op_2 \wedge sq_2 \wedge fp_2$	256	256
$op_1 \wedge sq_1 \wedge fp_3$	768	512
$op_2 \wedge sq_2 \wedge fp_3$	512	512
$op_1 \wedge sq_2 \wedge fp_4$	1536	512
$op_2 \wedge sq_2 \wedge fp_4$	512	512

Voxel Set Abstraction

This method aims to generate a set of keypoints from given point cloud PC_R and use a keypoint sampling strategy based on Farthest Point Sampling. This method generates a small number of keypoints that can be represented by $K \triangleq \{p_j = (x_j, y_j, z_j) \in \mathbb{R}^{B*3}\}, j = [1, NK]$, where NK is the number of points features that have the largest minimum distance, and B the batch size. The Farthest Point Sampling method is defined according to a given subset $PA \triangleq \{pa_j = (xa_j, ya_j, za_j)\}, j = \{1, 2, \dots, M\}, PA \subset PF$, where M is the maximum number of features to sample, and subset $PB \triangleq \{pb_k = (xb_k, yb_k, zb_k)\}, k = \{0, 1, 2, \dots, N\}, PB \subseteq PF$, where N is the total number of points features of PF , the point distance metric is calculated based on $D \triangleq \{d_i = \{(xb_k - xa_j)^2 + (yb_k - ya_j)^2 + (zb_k - za_j)^2\}\}, i \leq M$. Based on D a operation $SM \triangleq \{sm_k = \{\min(d_i, sm_{i-1})\}\}, k \leq M, i \leq N$ is performed, which calculates the smallest value distance between d_i and sm_{i-1} . $sm_k \in SM, k < N$ and SM represents the list of the last known largest minimum distance of point features. Assuming $sm_k = sm_{i-1} \mid d_i < sm_{i-1}$, it returns the index $IDX = \{idx_k = (i - 1)\}$. Based on $sm_k = \{d_i \mid d_i > sm_{i-1}\}$, thus $IDX = \{idx_k = (i)\}$. Finally, this operation generates a set of indexes in the form $IDX \triangleq \{idx_0, idx_1, \dots, idx_m\}, idx_m \in IDX, m \leq M$, and $IDX \in \mathbb{R}^{B*M}$, where B corresponds to the batch size and M the maximum number of features to sample. The keypoints K are given by $K \triangleq \{pf_{idx_0}, pf_{idx_1}, \dots, pf_{idx_m}\}$

These keypoints K are subject to an interpolation process utilising the semantic features encoded by the 3D Sparse Convolution as SP . In this interpolation process, these semantic features are mapped with the keypoints to the voxel features VL that reside. Firstly, this process defines the local relative coordinates of keypoints with Voxels VL by means $VLI \triangleq \{vli_i = (\frac{(kx_i - PC_{Rx_{min}})}{vlix_k}, \frac{(ky_i - PC_{Ry_{min}})}{vliy_k}) \in \mathbb{R}^2\}, k = [0, NK[, i = [0, NV[$. Then, a bilinear interpolation is carried out to map the point features SP from 3D Sparse Convolution in a radius R with the VLB , the local relative coordinates of keypoints. This is perform $PR \triangleq \{\forall sp \leq R, sp \in SP \mid R = (xr, yr) \in \mathbb{R}^2, sp_i = (pfx_i, pfy_i)\}, i = [0, NK[$. Afterwards, indexes of points are defined according to $vli_a \in VLI \mid vli_a = vli_i$ in the form (xa, ya) and another $vli_b \triangleq (xb = (xa + 1), yb = (ya + 1))$. The expression that gives the features sp_i from the BEV perspective based on vli_a and vli_b is the following:

- $SBEVA \triangleq (sp_{vliax}, sp_{vliay})$
- $SBEVB \triangleq (sp_{vlibx}, sp_{vliay})$
- $SBEVC \triangleq (sp_{vliax}, sp_{vliby})$
- $SBEVD \triangleq (sp_{vlibx}, sp_{vliby})$

Thus, the weights between these indexes vli_a, vli_b and vli_i are calculated, as follows:

- $WA \triangleq \{(vlix_i - prx_i) \times (vliy_i - vliy_i)\};$
- $WB \triangleq \{(vlix_i - prx_i) \times (vliy_i - vliya_i)\}$
- $WC \triangleq \{(vlix_i - prax_i) \times (vliby_i - vliy_i)\}$
- $WD \triangleq \{(vlix_i - prax_i) \times (vliy_i - vliay_i)\}$

Finally, the bilinear expression that gives the features sp_i from the BEV perspective is $PFBEV \triangleq (sbeva_i * wa_i) + (sbevb_i * wb_i) + (sbevc_i * wc_i) + (sbevd_i * wd_i)$, where $sbeva_i \in SBEVA$, $sbevb_i \in SBEVB$, $sbevc_i \in SBEVC$, $sbevd_i \in SBEVD$. Also, $wa_i \in WA$, $wb_i \in WB$, $wc_i \in WC$, $wd_i \in WD$, and $i = [0, NV[$.

The local features of $pfbev_j \in PFBEV$ is indicated by $vli_i = |vli_k - sp_i|$, $k = [0, NK[$, $i = [0, NV[$ and aggregated using PointNet++ according with their specification defined above. It will generate PTN that are voxel-wise features within the neighbouring voxel set vli_i of sp_i , transforming using PointNet++ specifications. This generates $ptn_i \in PTN$ according $PTN \triangleq ptn_i = ptn_0, \dots, ptn_{NK}$ and each ptn_i are aggregate features of 3D Sparse Convolution sp_i with $pfbi$ from different levels according to Table 4.

Backbone 2D

PointPillars (Figure 3) uses only a 2D Backbone since they require fewer computational resources when compared to 3D Backbones. However, they introduce information loss that can be mitigated by readjusting the objects back to LiDAR's Cartesian 3D system with minimal information loss. For this purpose, the features resulting from the PFN are used by the Backbone Scatter component, which scatters them back into a 2D pseudo-image. The next component, the Detection Head, then uses this 2D pseudo-image.

Other models, such as SECOND (Figure 4), PV-RCNN (Figure 6), PartA² (Figure 7) and Voxel-RCNN (Figure 8) compress the information into Bird's-eye view (BEV) after using a 3D Backbone for feature extraction. After, they perform feature encoding and concatenation using an Encoder Conv2D. After this process, the resulting features are passed to the Detection Head module.

Backbone Scatter

The features resulting from the PFN are used by the PointPillars Scatter component, which scatters them back to a 2D pseudo-image of size (D_{out}, H, W) , where H and W denote height and width, respectively.

BEV Backbone

BEV Backbone module receives 3D feature maps from 3D Sparse Convolution and reshapes them to BEV feature map. Admitting the given sparse features $SP \triangleq (B, C, D, H, W)$, the new sparse features are $(B, C \times D, H, W)$. The BEV Backbone is represented as a set of blocks BLC , in the form $blic_1, blic_2, \dots, blic_m$, where $m \geq 1$. Each block $blic_j \in BLC$, $j \leq m$, is represented by (n, F, U, S) . The element n represents the number of convolutional layers in BLC_j . The set of convolutional layers C in BLC_j is described as a set $\{c_1, c_2, c_3, \dots, c_n\}$, where $n \geq 1$. F represents the number of filters of each $c_i \in C$, $i \leq n$, U is the number of upsample filters of c_i . Each of the upsample filters has the same characteristics, and their outputs are combined through concatenation. S denotes the stride in c_1 . If $S > 1$, we have a downsampled convolutional layer (c_1). There are a certain convolutional layers (c_i , such that $i > 1$) that follow this layer. BatchNorm and ReLU layers are applied after each convolutional layer.

The input for this set of blocks BLC is spatial features extracted by 3D Sparse Convolution or Voxel Set Abstraction modules and reshaped to BEV feature map.

Table 7. The different block configuration ($blic_j \in BLC$) used. N.A. - Not Applicable

Models	$blic_1$	$blic_2$	$blic_3$
PointPillars	(3, 64, 128, 2)	(5, 128, 128, 2)	(5, 128, 128, 2)
SECOND	(5, 64, 128, 1)	(5, 128, 256, 2)	N.A.
PV-RCNN	(5, 64, 128, 1)	(5, 128, 256, 2)	N.A.
PointRCNN	N.A.	N.A.	N.A.
PartA ²	(5, 128, 256, 2)	(5, 128, 256, 2)	N.A.
VoxelRCNN	(5, 128, 256, 2)	(5, 128, 256, 2)	N.A.

Encoder Conv2D

Based on features extract in each block blc_j and after upsampled based on $U = 2D$, where D means downsample factor of the convolution layer C , upsample features $u_j \in U$, $j = [0, m]$ are concatenate, such that $UF \triangleq \text{cat}(u_j)$, where cat means $u_j + u_{j+1}$, $j = [0, m]$.

4.4. Detection Head

After that, the (4) Detection Head component receives the 2D encoded features as input and performs operations based on three modules: RPN Head, Point Head, and RoI Head.

4.4.1. RPN Head

Based on 2D encoded features, a set of convolutions to predict class labels, regression offsets, and direction are performed. Thus, a set of 1×1 convolutions $C1x = \{c1x_1, c1x_2, \dots, c1x_k\}$, where $k = 3$, is applied. Each $c1x_k$ can be represented by $C2D \triangleq (IC, OC, KS)$, where $C2D$ mean Convolution 2D, IC input channels, OC output channels and KS kernel size. $c1x_1$ is the class prediction convolution and can be described by $(UF, NA \times NC, 1)$, where NA means number of anchor per location and NC number of target classes to predict. $c1x_2$ is the convolution for bounding box offset regression and can be defined by $(UF, NA \times NC \times 7, KS)$ where it generates 2 anchors NA for each class NC and 7 are the number of bounding box offsets. Finally, $c1x_3$ is performed based on $(UF, NA \times NB, KS)$ where NA represent the same number of anchor per location as previously mentioned, NB the number of bins per anchor location and KS kernel size.

The figure representing our baseline network for each block can be seen in Figure 2. We use three blocks with a BEV backbone for PointPillars, while for the other models, we use two blocks. Each block is represented as described in Table 7. Table 8 describes the configuration of the RPN Head.

Table 8. The different RPN configurations ($c1x_k \in C1x$) used. N.A. - Not Applicable

Models	$c1x_1$	$c1x_2$	$c1x_3$
PointPillars	(512, 18, 1)	(5, 128, 128, 2)	(5, 128, 128, 2)
SECOND	(512, 18, 1)	(512, 42, 1)	N.A.
PV-RCNN	(512, 18, 1)	(512, 42, 1)	N.A.
PartA ²	(512, 18, 1)	(512, 42, 1)	N.A.
VoxelRCNN	(5, 128, 256, 2)	(5, 128, 256, 2)	N.A.

4.4.2. Point Head

Different implementations of Point Head have been proposed to refine RPN predictions or generate class labels, bounding box regression offsets, and direction. It can be composed of a class layer regression CR in the form $CR \triangleq \text{linear}(IN, OT)$ and bounding box layer BBR described as $PR \triangleq \text{linear}(IN, OT)$. The point class layer CR provides the segmentation score of foreground points, and PR gives the relative location of foreground points as $PR \triangleq \{pr_p = (x_f, y_f, z_f)\}$ and calculated based on a foreground point $fpp = (x_p, y_p, z_p)$ using $\{(x_t = \frac{(x_p - x_c)}{w} + 0.5, \frac{y_t = (y_p - x_c)}{l} + 0.5, z_t = \frac{(z_p - z_c)}{h} + 0.5), (\cos(\theta)_p - \cos(\theta)_c, \sin(\theta)_p - \sin(\theta)_c)\}$, where x_c, y_c, z_c are center coordinates of the bounding box, h, w , and l means height, width, length of bounding box respectively, and θ is the box orientation in bird-view.

Firstly, bounding box targets are normalised in a canonical coordinate system by first checking if the given points $PT \triangleq p_i = (x_i, y_i, z_i)$, $PT \in bb_k$ are within the Bounding box $bb_k \triangleq (xc_i, yc_i, zc_i, dx_i, dy_i, dz_i, \theta_i)$ by performing $((\frac{|x_i - xc_k|}{2} + 0.00001) \mid x_i - xc_k < dx_i \& \frac{|y_i - yc_k|}{2} + 0.00001 \mid y_i - yc_k < dy_i)$, where if the given statement is true the local lxn_i and lyn_i are calculated. The operation is $lxn_i = ((x_i - xc_k) \times (\cos(-\theta_i))) + ((y_i - yc_k) \times (-\sin(-\theta_i)))$ and $lyn_i = ((x_i - xc_k) \times (\sin(-\theta_i))) + ((y_i - yc_k) \times (\cos(-\theta_i)))$. After, we determine the local relative coordinate of p_i concerning bounding box bb_k in X-Y by

means $lr_i = ((x_i - xc_k) \times (\cos(-\theta_i))) + ((y_i - yc_k) \times (-\sin(-\theta_i)))$, $lyn_i = ((x_i - xc_k) \times (\sin(-\theta_i))) + ((y_i - yc_k) \times (\cos(-\theta_i)))$ and then determine if a point belongs and returning respective index to bounding box by $(\square(\ln x_i < \frac{dx_i}{2} + 0.00001 \vee \frac{ln y_i}{2} + 0.00001 < dy_i) \rightarrow id = i)$. After getting the points indexes within the bounding Boxes, all inside points are aggregated with PointNet++.

Point Intra Part Offset

It is both *CR* and *PR* to predicted point class labels and point bounding box offsets.

Point Head Simple

It is only composed of *CR*. However, it has modifications to its architecture $CR \triangleq \{cr_1, cr_2, cr_3\}$ where each *cr* is represented by a tuple (LR, BN, RL) where *LR* means linear regression, *BN* batch normalisation, and *RL* ReLU method. *BN* can be defined by (NF) where *NB* means the number of features and typically assumes the same value as *OT*.

Point Head Box

It is composed of *CR* and *PR* with architecture modifications. $CR\{cr_1, cr_2\}$ where $CR \triangleq (LR, BN, RL)$ where *LR* means linear regression, *BN* batch normalisation, and *RL* ReLU method. *PR* is composed of $PR \triangleq \{pr_1, pr_2\}$, where each *pr* is defined by the same tuple (LR, BN, RL) .

4.4.3. RoI Head

The Regions of Interest (RoI) Head is responsible for taking the RoI features of each box proposal of the RPN Head and then optimising the imperfect bounding box proposals by predicting and fixing the size and location (centre and orientation) residuals relative to the input bounding box predictions. Besides each model's specificities, any RoI Head is composed of a proposal layer that generates/refines a set of RoIs based on RPN RoIs, denoted as *PL*, an RoI feature extraction method *RF*, and Head module *HM* that can be composed but not restricted to Shared Fully Connected Layer *SFC*, up-down layer *UL* and *DL*, class layer *CL*, regression layer *RL*, RoI Point Pool 3D layer (*RoIPL*), RoI Grid Pool layer (*RoIGL*), RoI-aware Pool 3D layer (*RoiAP3D*), and a convolution part (*CnvP*) and convolution RPN (*CnvRPN*).

SFC are responsible for feature extraction and can be defined by a set $\{sfc_0, \dots, sfc_f\}$, $f = [0, 2]$, and $sfc_f \in SFC$ and sfc_f is represented by a tuple $(C1D, BN1D, RL, DRO)$, where *C1D* means convolution 1D, *BN1D* means batch normalisation 1D, *RL* ReLU, and *DRO* means dropout. *CL* can be defined by the set $\{cl_0, \dots, cl_c\}$, $c = [0, 2]$ and each cl_c by $(C1D, BN1D, RL, DRO)$. *RL* produces box predictions and is composed by the set $\{rl_0, \dots, rl_r\}$, $r = [0, 2]$, where each rl_r is defined by $(C1D, BN1D, RL, DRO)$. *DL* and *UL* mean bottom-up box generation proposal layers from foreground points. A sequence of Convolution 2D and ReLU methods can define the *DL*. A *UL* is represented as ul_1, ul_2 and each *ul* by the same sequence of Convolution 2D and ReLU methods.

RoIPL specifically pool 3D points and their corresponding point features according to the location of each 3D proposal of *PL*. Admitting the given output of bounding boxes *BB* and a specific bounding box $bb_n \in BB$, where $BB \triangleq \{bb_n = (x_n, y_n, z_n, h_n, w_n, l_n, \theta_n)\}$, where x, y, z are center coordinates of the predicted bounding box, h, w , and l denotes the height, width, and length of the bounding box, and θ the orientation of bounding box. Herein the *RoIPL* produces an enlarged set of $bbe_n \in BBE$ that can be defined by $(x_n, y_n, z_n, h_n + \eta, w_n + \eta, l_n + \eta, \theta_n)$, where η represents a constant value to resize the bounding box. The depth information loss for each bounding box proposal is compensated by including the distance information to the LiDAR sensor to the $uf_p \in UF$ that are BEV spatial features. Each uf_p is augmented with $d_b \triangleq \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2}$, $d_b \in D$, where x_p, y_p , and z_p correspond to coordinates of point features of Local Encoder module and x_c, y_c , and z_c center coordinates of LiDAR sensor. Thus, it generates a tensor in the form (VLM_{out}, D) that is fed to PointNet++ as described in section 4.3.1 to encode the augmented tensor with local features with global semantic BEV features *UF*. This generates a feature vector for confidence classification and box refinement.

The idea of *RoIGL* is to aggregate the keypoint features to the RoI-grid points with multiple receptive fields. Grid points are uniform sampling and can be described by $GP \triangleq \{gp_1, gp_2, \dots, gp_s\}, s = 216$, which means that a grid $6 \times 6 \times 6$ are usually adopted. Firstly, the identification of neighbouring keypoints to grid gp_i in a radius R is performed by means $GF \triangleq \{\forall p \leq r, p \in K \mid R = (xr, yr, zr) \in \mathbb{R}^3, p_j = (px_j, py_j, pz_j) \in \mathbb{R}^3 \mid gp_s = (gpx_j, gpy_j, gpz_j) \in \mathbb{R}^3 \mid \|p_j - gp_s\|^2\}, i = [0, NK[$. After all, a PointNet block is used to aggregate neighbouring keypoint set GF in the same way of Equation 2:

$$PTN \triangleq \{ptn_i = \max(ML(SG(gf_i)))\} \quad (2)$$

Then, the two MLP layers, $SFC(PTN)$ and $SC(PTN)$, are performed.

RoIAP3D aims to provide bounding box score confidence and refinement by aggregating the local feature information (VLM_{out}) with global semantic BEV features (UF) within the proposals. Two operations are performed within the point features pf_i of bounding boxes BB , such that $BB \triangleq \{bb_k = \{pf_i \in \mathbb{R}^C\}\}, i = [0, m[, pf_i \in PF$ and scattered to the voxel data structures $VLB \triangleq \{vlb_k = (x_j, y_j, z_j), i = [0, m[$ where x_j, y_j, z_j are encoded in canonical coordinates using Point Head module, m are the number of inside points within bounding box bb_k . The objective is to solve the problem of different proposals generating the same pooled points. For this purpose, average pooling for pooled part features operation denoted as PPF , and max pooling for pooled RPN features defined as $PRPN$ are adopted and can be described as $PPF \triangleq RoIMax(VLB, PF, BB), PPF \in \mathbb{R}^{S_x \times S_y \times S_z \times C}$ and $PRPN \triangleq RoIAvg(VLB, PF, BB), PRPN \in \mathbb{R}^{S_x \times S_y \times S_z \times C}$ where S_x, S_y, S_z are the resolution of Voxels spatial shape. The operations $RoIMax$ and $RoIAvg$ can be described more specifically:

$$RoIMax = \begin{cases} \max(\{pf_i \in vlb_k\}), & \text{if } count(PPF) > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$RoIAvg = \begin{cases} \frac{\sum_{i=0}^{count(PPF)} pf_i}{count(PPF)}, pf_i \in vlb_k(\{pf_i \in vlb_k\}), & \text{if } count(PPF) > 0 \\ 0, & \text{otherwise} \end{cases}$$

5. 3D Object Detection Model Specifications

Herein, we will specify each model in the different module frameworks. These models were selected based on the requirements established and defined in 1, since they are the models that best guarantee the trade-off between metrics (mAP and inference time). The set of models and their specificities concerning the developed framework is illustrated in Figures 3, 4, 5, 6, 7 and 8. The modules of each model are represented in the figures as green boxes, and the flow of the tensors occurs in the direction of the orange arrows.

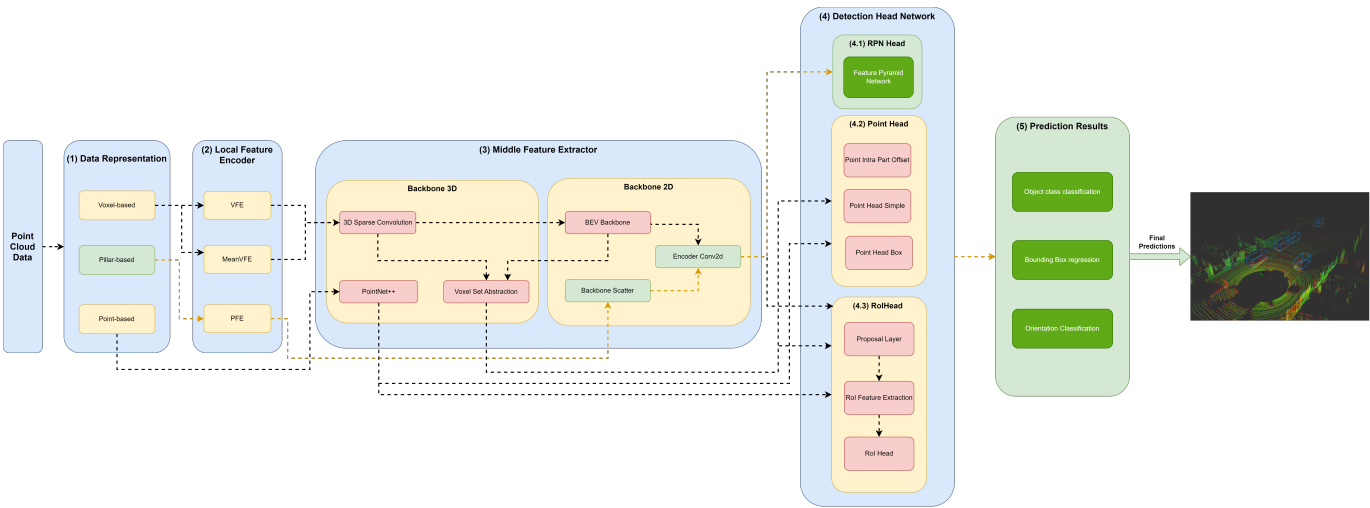


Figure 3. Structure of the PointPillars model represented in the developed framework.

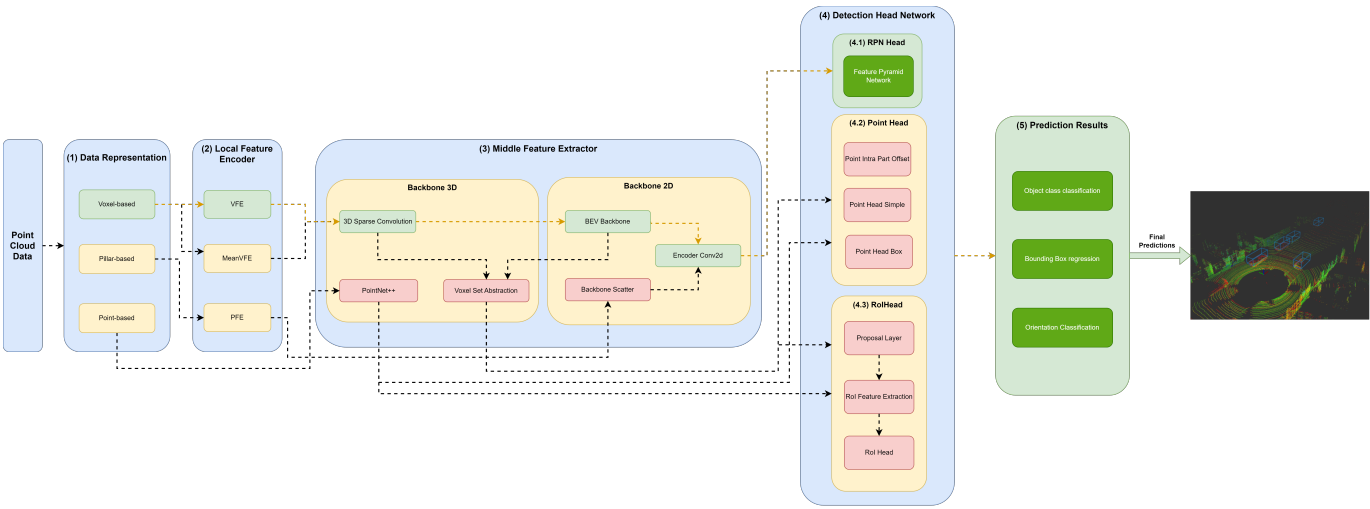


Figure 4. Structure of the SECOND model represented in the developed framework.

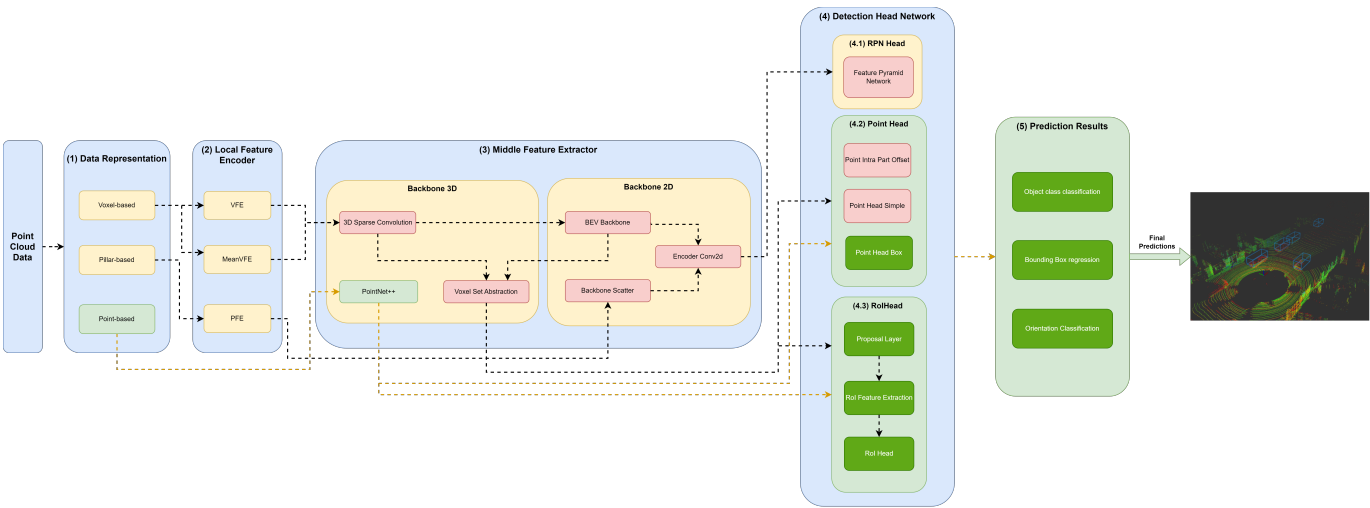


Figure 5. Structure of the PointRCNN model represented in the developed framework..

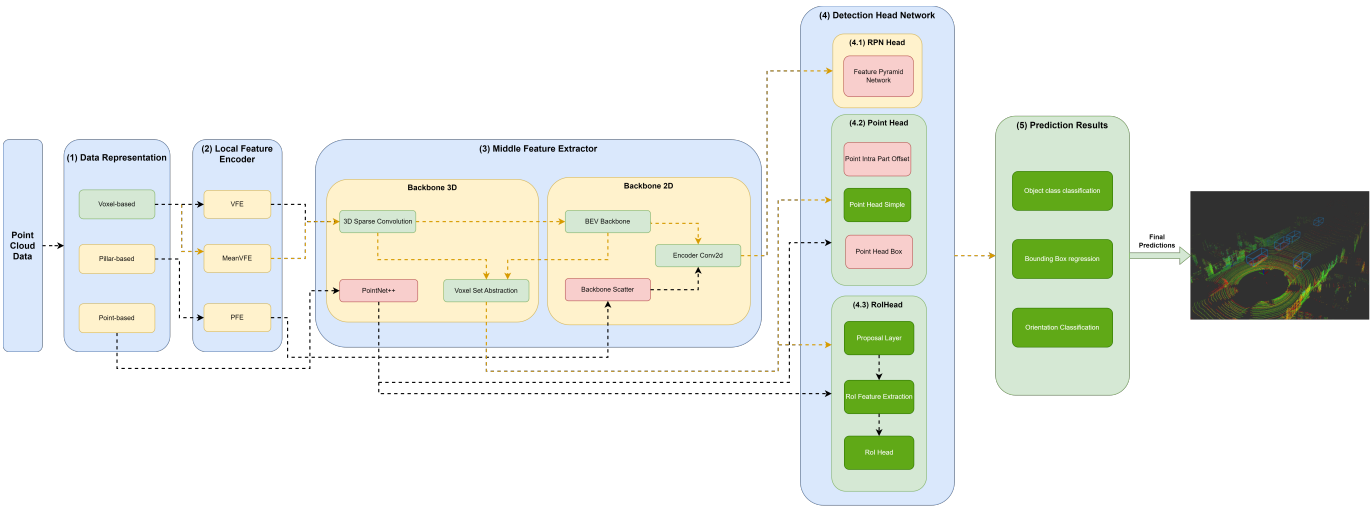


Figure 6. Structure of the PV RCNN model represented in the developed framework.

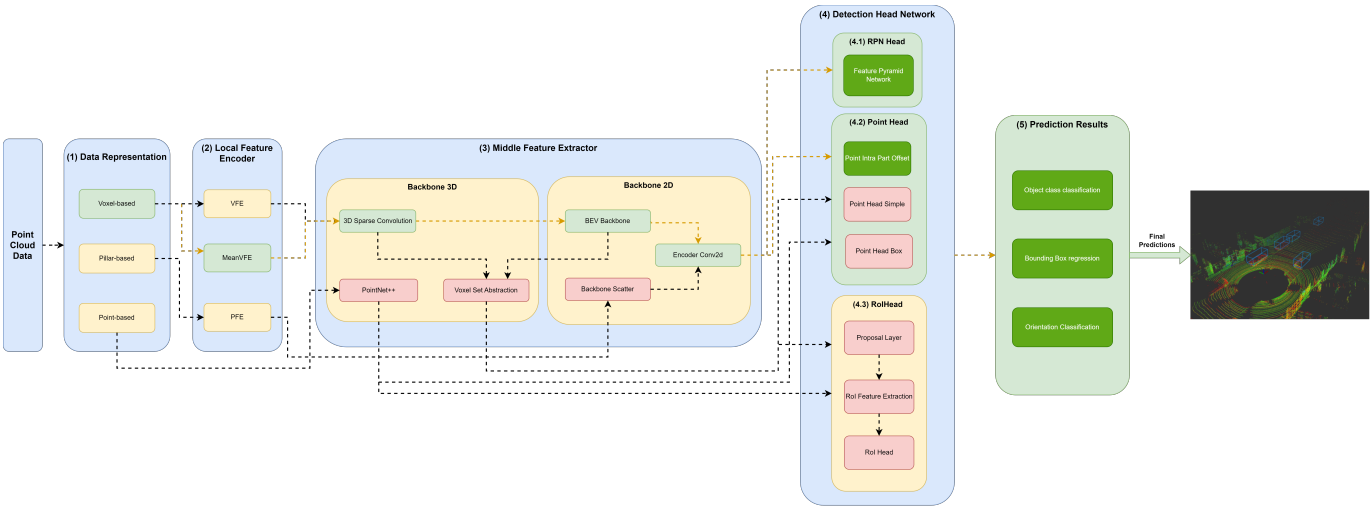


Figure 7. Structure of the PartA² model represented in the developed framework.

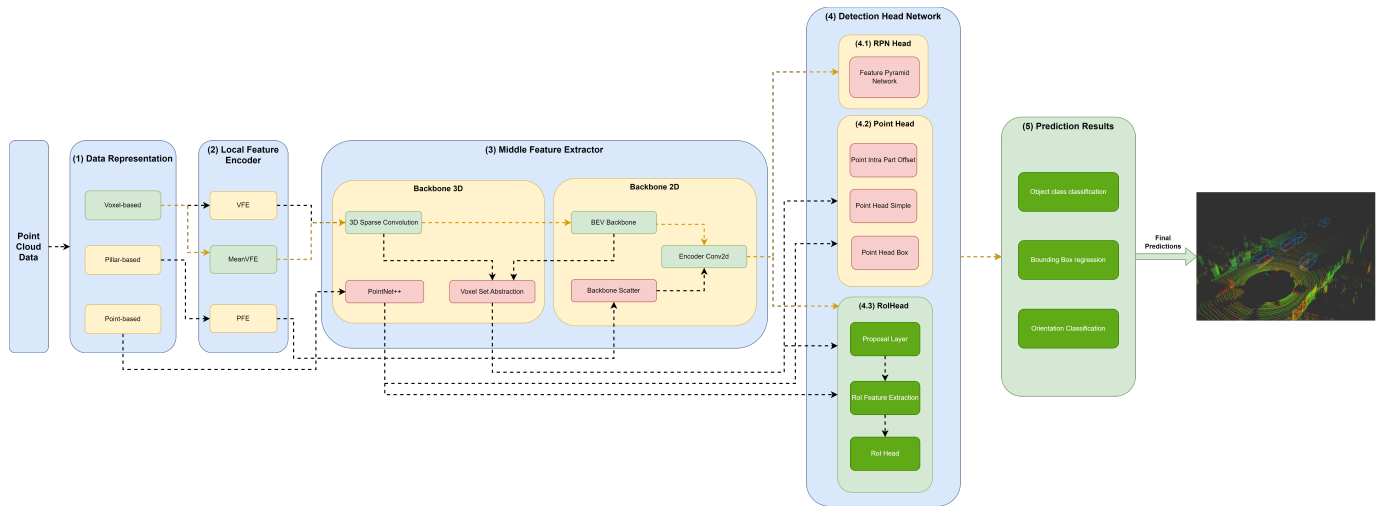


Figure 8. Structure of the VoxelRCNN model represented in the developed framework.

5.1. Data Representation

Typically the models of Figures 4, 6, 7 and 8 choose to represent the point cloud in Voxels. In this data structure, the point cloud is delimited (using the cropping technique), and a grid is produced where the data is discretised along the X-Y-Z axis.

Only PointPillars, illustrated in Figure 3, discretises this delimited space of the point cloud on the X-Y axis, creating a set of Pillars.

In the case of the PointRCNN model (Figure 5), it provides the delimited point cloud without any data discretisation and structuring process for the Middle Feature Encoder.

5.2. Local Feature Encoders

As illustrated in the Figures, three strategies are used by the models to improve the efficiency of the object detectors in the feature extraction of the data structures. Typically, these modules are responsible for the local feature extraction and then, via concatenation, aggregate these features. Three networks are used: VFE for SECOND (Figure 4), PFE for PointPillars (Figure 3) and Mean VFE for PV RCNN (Figure 6), PartA² (Figure 7), and VoxelRCNN (Figure 8).

5.3. Middle Feature Extractor

A variety of methods use 3D Backbones based on sparse and sub-manifold convolutions, such as SECOND (Figure 4), PV-RCNN (Figure 6), PartA² (Figure 7) and Voxel-RCNN (Figure 8). In the case of PV-RCNN, the 3D Voxel Set Abstraction backbone encodes the multiscale semantic features obtained by the 3D sparse CNN for keypoints. PointRCNN (Figure 5) uses PointNet++ [8] with multiscale clustering for feature extraction and get more context to the shape of objects and then passes these features to the Detection Head module.

Only PointPillars (Figure 3) uses 2D Backbones since they require fewer computational resources when compared to 3D Backbones. However, they introduce a loss in the information that is easily mitigated since it is possible to readjust the objects again to the Cartesian 3D system of LiDAR with fewer loss of information. For this purpose, the resulting PFE features are used by the Backbone Scatter component, which scatters them back into a 2D pseudo-image. The next Detection Head component then uses this 2D pseudo-image.

Other models, such as SECOND (Figure 4), PV RCNN (Figure 6), PartA² (Figure 7) and Voxel-RCNN (Figure 8) compress the information in Bird's-eye view (BEV) using the BEV Backbone for feature extraction then encode and concatenate the features using the Encoder Conv2D component. After this process, the resulting features are passed to the Detection Head.

5.4. Detection Head

As mentioned earlier, this module comprises three networks: RPN Head, Point Head and RoI Head.

All models except PointRCNN use the RPN Head to generate RoIs using a low-level algorithm called Selective Search [19] to produce proposed regions per frame of the point cloud. Selective Search generates sub-segments to generate many candidate regions and, following bottom-up grouping, recursively combines similar regions into larger regions to provide more accurate final candidate proposals. Each of these regions is submitted independently to the CNN module. The output feature map is then fed to an SVM classifier to predict the object class within the candidate RoI. Along with object class prediction, the algorithm also predicts four Bounding Box offset values.

The Point Head is used to assist the RPN Head, as illustrated in Figures 6 and 7, or generate predictions of object classes and predict four values that are the Bounding Box offsets, as shown in Figures 5 and 8. Point Head generates various masks of objects or parts of objects in a multiscale way, followed by a simple bounding box inference to generate proposals, also called point proposals, using each point to contribute to the reconstruction of the 3D geometry of the object.

The RoI Head used by the PointRCNN (Figure 5), PV-RCNN (Figure 6), PartA² (Figure 7) and Voxel-RCNN (Figure 8), naturally uses the RoI features of each bounding box proposed in the RPN, and then optimises the imperfect bounding boxes from previous stages, predicting and correcting the size and location (center and orientation) in relation to the predictions of the input bounding boxes.

6. Network Training and Fine-tuning

The models described in this document were trained using the KITTI data sets. In addition, the models were evaluated based on the KITTI benchmarks, namely for detecting 3D objects and BEV, considering a validation set. Regarding the number of epochs used in the training phase, a methodology spread by the literature was considered. Thus we use 200 epochs, considering the data described in Table 13. Considering training hyperparameters, We define the initial learning rate of 0.01, learning rate decay of 0.1, decay epoch methodology, weight decay of 0.01, gradient clipping normalisation with a max value of 10, beta1 of 0.95 and beta2 of 0.85. We use learning rate decay, weight decay, and gradient clipping normalisation as regularisation procedures to prevent overfitting. The evaluation metrics in the results were based on the official KITTI evaluation detection metrics. Hence, the metrics used were mAP for a BEV and 3D Object Detection. The partition of the training data used in this work consisted of a division discussed in [2]. This approach divides the 7481 training examples that are provided, into a training set of 3712 samples, with the remaining 3769 samples belonging to the evaluation set. Moreover, the benchmarks presented in this article are based on the evaluation set only.

We select three target classes in all experiments: car, pedestrian and cyclist. Typically, all the models described herein generate two separate networks. One network is optimised for predicting cars and another for pedestrians and cyclists. However, this approach can be improper in self-driving car applications since low-edge devices with few resources must cope with two parallel models. For this reason, we trained all classes in a one-single model for all 3D Object Detectors.

For the fine-tuning process, we save the results of the mAP for each epoch to understand when models converge. Herein, we provide a study with the consequences of the number of samplings and min points per class sampling compared with the study made in [20]. In [20], we used different class sampling strategies but without changing the number of min points for class sampling.

Sampling Instance Strategy. We focus on optimising the number of sampling instances and min points per class sampling. The main objective of the sampling strategy is to soften the KITTI dataset imbalance issue. During training, the point cloud is randomly fed with these instances, which means they are placed into the current point cloud. Although this is

true, the min points affect whether or not a certain instance can be used for sampling. If we increase the min number of points in the training process, instances such as pedestrians and cyclists are less sample because few points exist to describe their shape. On the other hand, if we decrease too many min points, the model suffers in distinguishing between the foreground and the background points. In our experiments, we use the configurations described in Table 9. The min point for class sampling was fixed per class as 5 instead of 10 points for pedestrian and cyclist classes and 5 points for the car.

Point Cloud Range. The point cloud range affects any Object Detector's detection range, limiting its detection range. Our research uses the original point cloud range for all models to represent the location of ground truth objects for all frames in the KITTI dataset frame. For example, in cars, it is possible to verify in terms of depth information that most ground truth instances are between the 0 and 70 meters. After 70 meters from the LiDAR sensor centre, the number of instances drastically decreases. This can be explained by the fact that after this range, the number of points to describe an object's shape is very few, making detecting objects difficult. Thus, this experiment compares the point cloud range of PointPillars and the other models where the detection range is not compromised. The point cloud ranges are depicted in Table 10. Also, we analyse the number of data structures (max number of Pillars or Voxels) compared with the study in [20].

Table 9. Number of sampling instances (SI) per class.

SI Configuration	Car	Pedestrian	Cyclist
SI_1	15	10	10
SI_2	25	20	20

Table 10. The different point cloud ranges (PC_R) configurations used in fine-tuning.

PC_R Configuration	X_{min}	X_{max}	Y_{min}	Y_{max}	Z_{min}	Z_{max}
PC_{R1}	0	69.12	-39.68	39.68	-3	1
PC_{R2}	0	70	-40	40	-3	1

Data structure sizes. The object detection model receives the points in PC_R and discretises them in the X-Y axis, thus creating a set of pillars, or discretises in X-Y-Z and creates a set of Voxels. Each data structure DS has a fixed size in PC_R . The data structure size directly impacts model accuracy and inference time. Increasing the data structure size can result in too much data being encoded and consequently randomly sampled, leading to information loss (the maximum number of points per data structure is set for computational saving purposes). On the other hand, reducing the data structure size can increase the number of non-empty data structures, increasing memory usage and inference time. Two DS configurations were used in our fine-tuning process, as shown in Table 11.

Table 11. Pillar size (S_{DS}) configurations used in fine-tuning.

S_{DS} Configuration	$S_{DS}length$	$S_{DS}height$	$S_{DS}depth$
S_{DS16}	0.16	0.16	1
S_{DS5}	0.05	0.05	0.1

Number of Data Structures. Max number of data structures is defined to explore the KITTI dataset sparsity problem since most data structures will be empty. Using a large number of Data Structures can result in most of them being filled with zeros (to create a dense tensor), making it inefficient for inference time purposes. Based on the distribution of the number of points per data structure in the KITTI dataset, a max number of points is also defined, as shown in Table 12.

Table 12. Total number of data structures used in fine-tuning.

<i>P</i> Configuration	Total Number of <i>DS</i>	Max Number of Points Per <i>DS</i>
P_{12K}	12K	100
P_{16K}	16K	5

7. Performance Evaluation, Comparison, and Discussion

This section reports the set of experiments, which results from the random search methodology, used to achieve a better trade-off between accuracy and inference time performance metrics. Table 13 depicts the experiments and corresponding network configurations and models. PointPillars settings and their results are also provided to understand the impact of producing a model optimised to produce three-class output rather than separating into two distinct networks (one for cars and another for pedestrians and cyclists).

Table 13. The set of experiments conducted and respective network configurations.

Experiment	Model Config.	PC_R Config.	SI Config.	No. Output Classes	S_{PL} Config.	P Config.
1	PointPillars	PC_{R1}	SI_1	3	S_{DS16}	P_{12K}
2	SECOND	PC_{R2}	SI_1	3	S_{DS5}	P_{16K}
3	PV-RCNN	PC_{R2}	SI_1	3	S_{DS5}	P_{16K}
4	PointRCNN	PC_{R2}	SI_1	3	S_{DS5}	P_{16K}
5	Part A ²	PC_{R2}	SI_1	3	S_{DS5}	P_{16K}
6	VoxelRCNN	PC_{R2}	SI_1	3	S_{DS5}	P_{16K}
7	PointPillars	PC_{R1}	SI_2	3	S_{DS16}	P_{12K}
8	SECOND	PC_{R2}	SI_2	3	S_{DS5}	P_{16K}
9	PV-RCNN	PC_{R2}	SI_2	3	S_{DS5}	P_{16K}
10	PointRCNN	PC_{R2}	SI_2	3	S_{DS5}	P_{16K}
11	Part A ²	PC_{R2}	SI_2	3	S_{DS5}	P_{16K}
12	VoxelRCNN	PC_{R2}	SI_2	3	S_{DS5}	P_{16K}

The following Tables 14, 15, 16, and 17 provide the results of experiments of Table 13 in terms of AP for three difficulty levels (Easy, Moderate and Hard) and different Intersection over Union (IOU) thresholds, according to KITTI benchmarks. For cars, IOU is 70%, while for pedestrians and cyclists, it is required IOU of 50%. Table 18 presents the comparative results of the experiments carried out in this study with the original results in the literature. The comparison considers the overall three identified classes, both for 3D and BEV. The results presented for the developed experiments consider the overall values per class for the best detection metric.

Table 14. Results in validation set for BEV detection metric for experiment 1-6.

Model	Epoch	Experiment	Car			Cyclist			Pedestrian			Overall
			Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard	
Voxel R-CNN	197	6	96.9	94.89	95.08	73.03	77.68	80.3	85.03	85.54	85.97	87.12
Part A ²	187	5	97.64	96.72	96.6	81.37	83.02	83.38	90.21	90.81	90.95	90.31
PointPillars	160	1	76.29	79.05	80.80	57.52	58.01	58.10	77.75	72.52	73.62	70.84
PointRCNN	24	4	92.83	88.64	88.55	80.71	79.85	80.9	89.35	89.03	88.67	86.04
PV-RCNN	92	3	94.52	93.91	93.58	78.65	79.46	80.65	80.83	80.32	80.59	84.94
SECOND	154	2	87.97	83.75	84.43	71.29	76.0	78.23	77.99	78.96	79.55	80.74

Table 15. Results in validation set for 3D detection metric for experiment 1-6.

Model	Epoch	Experiment	Car			Cyclist			Pedestrian			Overall
			Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard	
Voxel R-CNN	140	6	89.55	83.37	82.63	69.72	72.7	73.53	72.16	71.38	72.71	76.29
Part A ²	182	5	79.15	77.31	77.25	73.6	74.84	76.11	72.63	74.94	76.01	76.46
PointPillars	179	1	63.49	58.98	59.27	52.27	60.16	63.0	41.06	40.38	38.99	53.75
PointRCNN	89	4	84.87	79.86	79.37	68.96	71.11	71.35	76.55	75.01	74.36	75.03
PV-RCNN	139	3	88.86	83.57	82.89	71.52	73.21	74.39	64.34	64.53	64.28	73.86
SECOND	147	2	75.55	72.19	72.43	55.23	62.36	65.06	61.77	62.05	61.34	66.28

Table 16. Results in validation set for BEV detection metric for experiment 7-12.

Model	Epoch	Experiment	Car			Cyclist			Pedestrian			Overall
			Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard	
Voxel R-CNN	199	12	97.19	96.11	96.32	74.43	77.55	79.92	88.53	88.29	88.42	88.22
Part A ²	195	11	97.75	96.71	96.61	78.23	80.9	82.74	89.99	90.41	90.76	90.04
PointPillars	21	7	85.76	81.04	82.87	67.04	73.04	75.8	55.39	57.19	58.58	72.42
PointRCNN	16	10	96.3	90.84	90.83	78.31	78.51	79.01	85.88	85.24	85.32	85.05
PV-RCNN	190	9	96.4	93.45	94.08	69.05	72.34	74.74	78.77	80.17	80.7	83.17
SECOND	162	8	90.61	86.51	86.05	78.66	79.76	79.91	66.27	73.66	76.79	80.92

Table 17. Results in validation set for 3D detection metric for experiment 7-12.

Model	Epoch	Experiment	Car			Cyclist			Pedestrian			Overall
			Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard	
Voxel R-CNN	186	12	83.72	81.21	81.33	68.44	71.01	73.69	67.62	69.28	70.42	75.15
Part A ²	187	11	83.29	82.53	82.87	74.13	75.38	76.2	69.46	70.95	70.82	76.63
PointPillars	21	7	69.49	66.31	66.94	47.58	52.72	56.98	37.4	36.91	39.48	54.57
PointRCNN	39	10	89.96	83.36	81.59	68.66	71.26	71.32	73.52	74.04	72.66	75.19
PV-RCNN	44	9	83.42	80.46	80.61	63.75	67.41	70.22	63.18	63.38	63.45	71.42
SECOND	162	8	76.02	70.24	72.77	56.1	63.59	65.77	56.2	58.87	58.14	65.56

Table 18. Our results in KITTI validation set vs Original results in KITTI test set for 3D and BEV detection metrics.

Model	Our results (Overall per Class)						Original results (Overall per Class)					
	3D			BEV			3D			BEV		
	Car	Cyc.	Ped.	Car	Cyc.	Ped.	Car	Cyc.	Ped.	Car	Cyc.	Ped.
Voxel R-CNN	85.18	71.98	72.08	96.54	77.3	88.41	83.19	-	-	89.94	-	-
Part A ²	82.9	75.24	70.41	96.99	82.59	90.66	79.94	66.54	45.50	88.03	71.34	34.92
PointPillars	67.58	52.43	37.93	83.22	71.96	57.05	75.29	62.56	44.09	86.48	66.07	50.67
PointRCNN	84.97	70.41	73.41	90.01	80.49	89.02	77.77	62.10	41.12	87.41	70.03	47.91
PV-RCNN	85.11	73.04	64.38	94.0	79.59	80.58	82.83	66.65	45.25	90.59	71.26	52.39
SECOND	73.39	60.88	61.72	87.72	79.44	72.24	79.20	62.56	44.09	88.4	68.36	47.63

Table 19. Our inference time metric results.

Model	Total (ms) ~	Speed (Hz) ~
PointPillars	17.25	57.97
SECOND	34.1	29.33
PV-RCNN	118.03	8.47
PointRCNN	97.83	10.22
Part A ²	82.66	12.10
VoxelRCNN	59	16.95

Table 20. Original model inference time metric results.

Model	Total (ms) ~	Speed (Hz) ~
PointPillars	16	62.5
SECOND	110	9.09
PV-RCNN	80	12.5
PointRCNN	100	10
Part A ²	80	12.5
VoxelRCNN	40	25

As demonstrated in the before-mentioned results, the improvements introduced for three-class trained models produced better mAP and very close inference time results (Tables 19 and 20). However, it is clear that there is always a cost in terms of mAP for producing three-class inference models and our results are for the KITTI validation set, and the original results are for the KITTI test set. Regarding the point cloud range in our networks, we reproduced original configurations for all models, with fewer *DS* when compared with the study in [20] since most *DS* will be empty. This improvement drastically decreases the inference time when comparing PointPillars with the same research. Although the original model inference times are better when compared with our results (Tables 19 and 20), this can be explained by the fact that original models obtained their results by training separated networks, one for cars and another for pedestrians and cyclist (a standard literature practice on KITTI benchmarks). By training three-class models, gradients are affected by all those instances, which leads our models to lose the specialisation for prediction. However, as mentioned in [20], producing separate networks is impractical for self-driving applications.

Reducing the minimum points to consider a sample instance brought gains in terms of mAP and for the same model architecture since more instances can be used for data augmentation. This allows for expanding the diversity of the training data and our models to learn more patterns from data.

8. Conclusion

The research about deep learning methods for 3D Object Detection on LiDAR data has increased tremendously in recent years, with many models, repositories, and different technologies being developed. Although this benefits scientific development in this area, the various technologies, software, repositories and models are a bottleneck for testing and improving the current methods.

To cope with this limitation, we develop a framework for representing multiple SoA 3D object detectors with highly refactored codes for both one-stage and two-stage methods. The main idea of this framework is to facilitate the implementation, reusing and implementation of new techniques in each framework module with less manual engineering effort. In conclusion, it enables the abstract implementation, reusing and building of any object detector in one single 3D object detector framework.

Nonetheless, it is evident that creating three-class inference models comes with a trade-off in terms of mAP. Our study's results are based on the KITTI validation set, while

the original findings were obtained using the KITTI test set. We replicated the original network configurations for all models concerning the point cloud range, but with fewer DS and than the research mentioned in previous section. The improvement mentioned earlier leads to a considerable reduction in the inference time when PointPillars is compared to the same research.

Author Contributions: Conceptualization, A.S., P.O., and D.D.; methodology, A.S., P.O., and D.D.; software, A.S., and P.O.; validation, P.M.P., J.Mac., P.N., A.S., P.O., D.D., D.F. R.N., and J.M.; formal analysis, A.S., P.O., D.D., J.Mac., P.N., D.F., R.N., P.M.P., J.M.; investigation, A.S., P.O., and D.D.; resources, J.Mac., P.N., P.M.P., and J.M.; data curation, A.S., P.O. and R.N.; writing—original draft preparation, A.S., P.O., and D.D.; writing—review and editing, A.S., P.O., D.D., R.N., D.F., J.Mac., P.N., J.M. and P.M.P.; visualization, A.S., P.O., D.D., D.F., J.Mac., P.N., P.M.P. and J.M.; supervision, J.Mac., P.N., P.M.P., and J.M.; project administration, J.Mac., P.N., J.M., and P.M.P.; funding acquisition, J.Mac., P.N., J.M. and P.M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by FCT—Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020 and the project “Integrated and Innovative Solutions for the well-being of people in complex urban centers” within the Project Scope NORTE-01-0145-FEDER-000086. The work of Pedro Oliveira was supported by the doctoral Grant PRT/BD/154311/2022 financed by the Portuguese Foundation for Science and Technology (FCT), and with funds from European Union, under MIT Portugal Program.

Institutional Review Board Statement: x.

Informed Consent Statement: x.

Data Availability Statement: x.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Beltrán, J.; Guindel, C.; Moreno, F.M.; Cruzado, D.; Garcia, F.; De La Escalera, A. Birdnet: a 3d object detection framework from lidar information. In Proceedings of the 2018 21st International Conference on Intelligent Transportation Systems (ITSC). IEEE, 2018, pp. 3517–3523.
2. Chen, X.; Ma, H.; Wan, J.; Li, B.; Xia, T. Multi-view 3d object detection network for autonomous driving. In Proceedings of the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1907–1915.
3. Fernandes, D.; Silva, A.; Névoa, R.; Simões, C.; Gonzalez, D.; Guevara, M.; Novais, P.; Monteiro, J.; Melo-Pinto, P. Point-cloud based 3D object detection and classification methods for self-driving applications: A survey and taxonomy. *Information Fusion* **2021**, *68*, 161–191. <https://doi.org/https://doi.org/10.1016/j.inffus.2020.11.002>.
4. Cosmas, K.; Kenichi, A. Utilization of FPGA for onboard inference of landmark localization in CNN-Based spacecraft pose estimation. *Aerospace* **2020**, *7*, 159.
5. Ngadiuba, J.; Loncar, V.; Pierini, M.; Summers, S.; Di Guglielmo, G.; Duarte, J.; Harris, P.; Rankin, D.; Jindariani, S.; Liu, M.; et al. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology* **2020**, *2*, 015001.
6. Sharma, H.; Park, J.; Amaro, E.; Thwaites, B.; Kotha, P.; Gupta, A.; Kim, J.K.; Mishra, A.; Esmaeilzadeh, H. Dnnweaver: From high-level deep network models to fpga acceleration. In Proceedings of the the Workshop on Cognitive Architectures, 2016.
7. Yan, Y.; Mao, Y.; Li, B. Second: Sparsely embedded convolutional detection. *Sensors* **2018**, *18*, 3337.
8. Qi, C.R.; Yi, L.; Su, H.; Guibas, L.J. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In Proceedings of the Advances in neural information processing systems, 2017, pp. 5099–5108.
9. Zhou, Y.; Tuzel, O. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection, 2017, [arXiv:cs.CV/1711.06396].
10. Shi, S.; Wang, Z.; Shi, J.; Wang, X.; Li, H. From points to parts: 3d object detection from point cloud with part-aware and part-aggregation network. *IEEE transactions on pattern analysis and machine intelligence* **2020**.

11. Lang, A.H.; Vora, S.; Caesar, H.; Zhou, L.; Yang, J.; Beijbom, O. Pointpillars: Fast encoders for object detection from point clouds. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 12697–12705.
12. Shi, S.; Wang, X.; Li, H. Pointtrcnn: 3d object proposal generation and detection from point cloud. In Proceedings of the Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 770–779.
13. Chen, Y.; Liu, S.; Shen, X.; Jia, J. Fast point r-cnn. In Proceedings of the Proceedings of the IEEE/CVF international conference on computer vision, 2019, pp. 9775–9784.
14. Shi, S.; Guo, C.; Jiang, L.; Wang, Z.; Shi, J.; Wang, X.; Li, H. PV-RCNN: Point-voxel Feature Set Abstraction for 3D Object Detection. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 10529–10538.
15. Graham, B.; van der Maaten, L. Submanifold Sparse Convolutional Networks. *CoRR* **2017**, *abs/1706.01307*, [[1706.01307](#)].
16. Graham, B. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070* **2014**.
17. Lu, L.; Shin, Y.; Su, Y.; Karniadakis, G.E. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733* **2019**.
18. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the International conference on machine learning. PMLR, 2015, pp. 448–456.
19. Uijlings, J.R.; Van De Sande, K.E.; Gevers, T.; Smeulders, A.W. Selective search for object recognition. *International journal of computer vision* **2013**, *104*, 154–171.
20. Silva, A.; Fernandes, D.; Névoa, R.; Monteiro, J.; Novais, P.; Girão, P.; Afonso, T.; Melo-Pinto, P. Resource-Constrained Onboard Inference of 3D Object Detection and Localisation in Point Clouds Targeting Self-Driving Applications. *Sensors* **2021**, *21*, 7933.