SDC

IMPLEMENTATION OF 3D OBJECT DETECTION PAPER

February 15, 2022

Name: Yomna Gamal El-Din Mahmoud El-Sayed

OVERVIEW

1. Project Description

3D Object Detection from Lidar point cloud and RGB image

Many LiDAR-based methods for detecting large objects, single-class object detection, or under easy situations were claimed to perform quite well. However, their performances of detecting small objects or under hard situations did not surpass those of the fusion-based ones due to failure to leverage the image semantics.

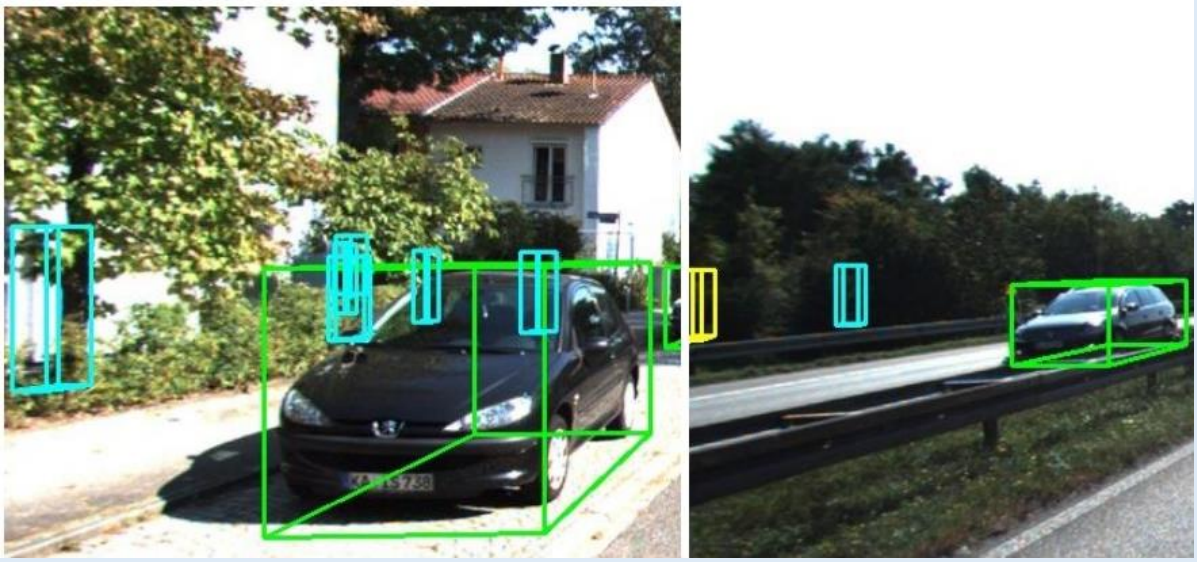


Figure 1 PVRCNN result

A paper called Voxel-Pixel Fusion Network for Multi-class 3D Object Detection 2021, give a solution for this problem.

Their proposed solution is a deep learning (DL)-embedded fusion-based multi-class 3D object detection network which admits both LiDAR and camera sensor data streams (VPFNet). Inside this network, a key novel component is called Voxel-Pixel Fusion (VPF) layer, which takes advantage of the geometric relation of a voxel-pixel pair and fuses the voxel features and the pixel features with proper mechanisms. The proposed method is evaluated on the KITTI benchmark for multi-class 3D object detection task under multilevel difficulty, and is shown to outperform all state-of-the-art methods in mean average precision (mAP). It is also noteworthy that our approach here ranks the first on the KITTI leaderboard for the challenging pedestrian class.

There are NO Official Code for the paper

2. Dataset

It's evaluated on KITTI Vision Benchmark (Geiger, Lenz, and Urtasun 2012), which provides 7,481 training samples and 7,518 testing samples for the 3D and BEV object detection tasks.

Each sample contains a LiDAR point cloud, camera RGB image and calibration matrix, then size of data we need is:

- *left color images of object data set (12 GB)*
- *Velodyne point clouds, as they use laser information (29 GB)*
- *camera calibration matrices of object data set (16 MB)*

For training samples, labels are a plus. Obstacle categories for evaluation are car, pedestrian, and cyclist. The difficulties are easy, moderate, and hard, representing fully visible and slightly truncated, partly occluded and moderate truncation, challenging to see and severe truncation, respectively.

Since it does not provide a validation set and only provides a training set and testing set, we split the training set into training split and testing split. Therefore, the original training set, including 7,481 frames, is separated into a 3,712-training split and 3,769 validations split.

3. Model Description

First, the LiDAR stream voxelizes the point cloud and utilizes a 3D backbone network to extract features.

At the same time, the camera stream utilizes a 2D backbone network to extract features.

Then, their feature maps are fed into the voxel-pixel fusion layer to fuse.

Finally, the fused features are passed to the region proposal network and detection network to obtain the detection.

Besides, the 2D detection task in the camera stream is auxiliary for training, which can be removed during inference time.

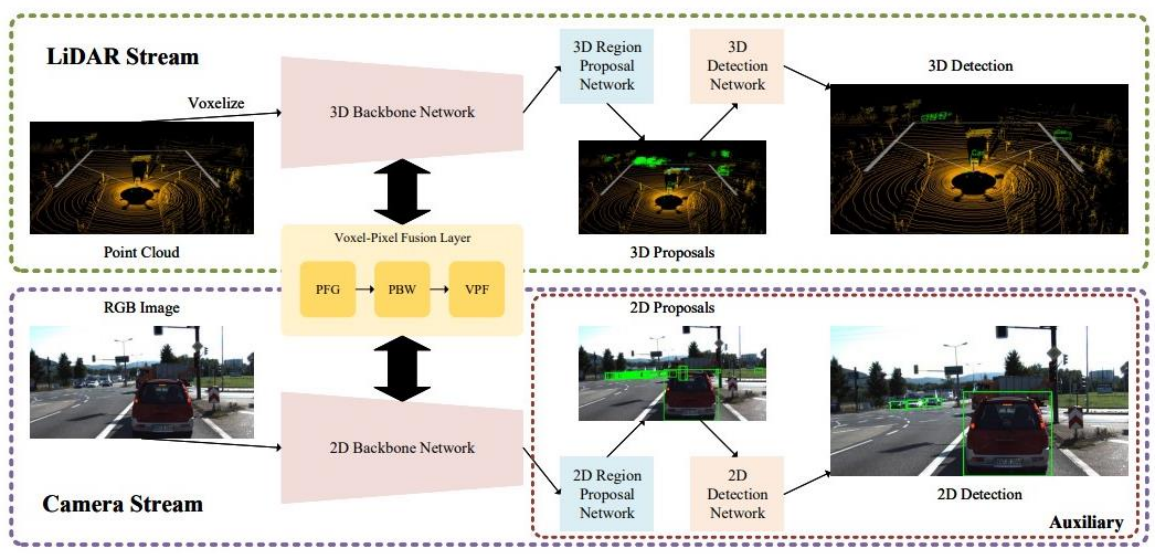


Figure 2 VPFNet Model

4. Training

Data augmentation can synthesize data artificially by modifying the original one, and this strategy is quite effective and mature for a single sensor.

However, it is challenging for fusion-based methods. Therefore, they pretrain the LiDAR stream and camera stream separately with data augmentation enabled.

*They first pre-train PV-RCNN (Shi et al. 2020) about **60** epochs for the LiDAR stream on the KITTI dataset.*

*Then, they pre-train Cascade R-CNN (Cai and Vasconcelos 2021) about **six** epochs for the camera stream.*

On the contrary, Cascade R-CNN uses the pre-trained model of the COCO (Lin et al. 2014) dataset and is finetuned on the KITTI dataset.

*After pre-training, the voxel-pixel fusion layer is joined, and the complete VPFNet continues to be trained about **five** epochs without using data augmentation.*

*Besides, the total batch size is 24 on **eight** NVIDIA Tesla V100 GPUs for all the training settings, and the cosine annealing strategy is adopted to adjust the learning rate dynamically. In the pre-training stage, learning rates for both the LiDAR stream and camera stream are set at 0.01. Finally, the VPFNet is trained with a learning rate of 0.002.*

Since resources we have is lower than they used "8 NVIDIA Tesla V100 GPUs", we might need to reduce number of epochs or increase time needed for training.

IMPLEMENTATION

1. Dataset Preparation

1. *Download the 3D KITTI detection dataset include:*
    - *Velodyne point clouds (29 GB)*
    - *Training labels of object data set (5 MB)*
    - *Camera calibration matrices of object data set (16 MB)*
    - *Left color images of object data set (12 GB)*
2. *In this project, we use the cropped point cloud data. Point clouds outside the image coordinates are removed.*
   Using preprocessing part of an unofficial implementation of VoxelNet, and update it to fit that project. And run crop.py to generate cropped data.
   This help in reducing size of the data from 41 GB to 15.6 GB which could be saved on drive.
3. *Split the training set into training and validation set as in the paper.*

The Notebook contains the code of this part is here

When visualizing result of this part, point cloud become as:

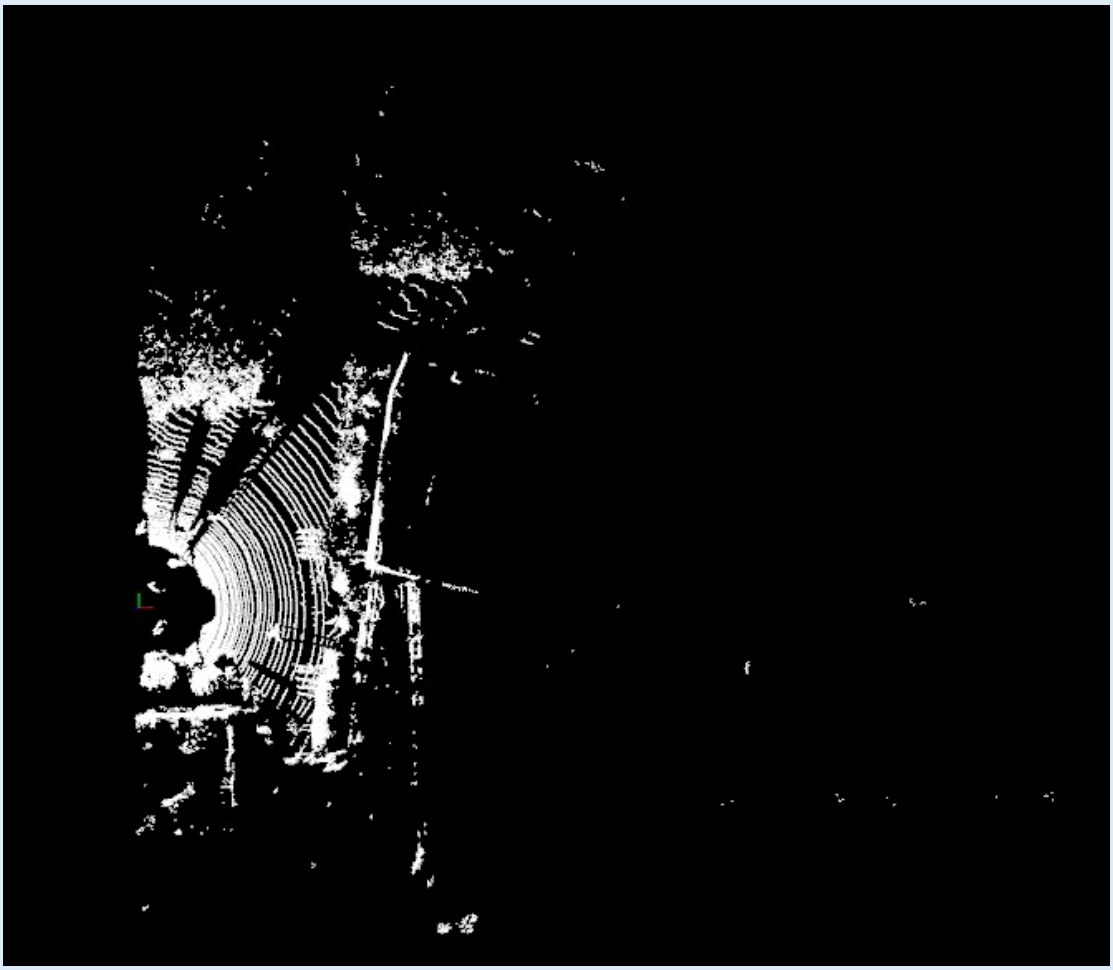


Figure 3 Point cloud sample

2. Visualization of Point Cloud

Using same visualization used in OpenPCDet to visualize both processed data and result of PVRCNN.

Visualizing point cloud not working on Colab, it works locally.

The code for this part is here.

3. 2D Backbone Network → Cascade R-CNN

*In the paper, they use Cascade R-CNN uses the pre-trained model of the COCO dataset and is finetuned on the KITTI dataset by pre-train it for about **six** epochs for the camera stream.*

Here, using Cascaded RCNN model in MMDet detector and then train it for 4 epochs.

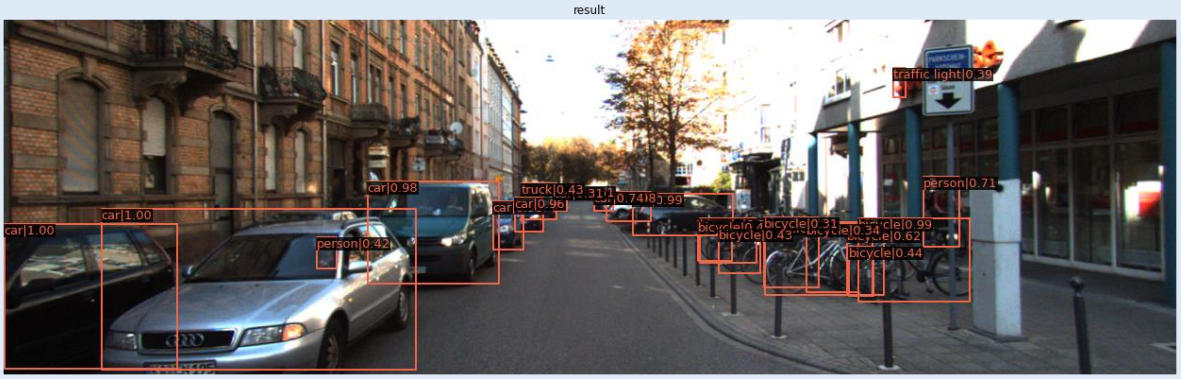


Figure 4 Result before finetune

1st Epoch

Class	Gts	Dets	Recall	AP
Car	14385	38216	0.962	0.909
Pedestrian	2280	7299	0.796	0.675
Cyclist	893	3171	0.767	0.574
mAP				0.719

2nd Epoch

Class	Gts	Dets	Recall	AP
Car	14385	26851	0.953	0.906
Pedestrian	2280	4460	0.761	0.663
Cyclist	893	1830	0.684	0.576
mAP				0.715

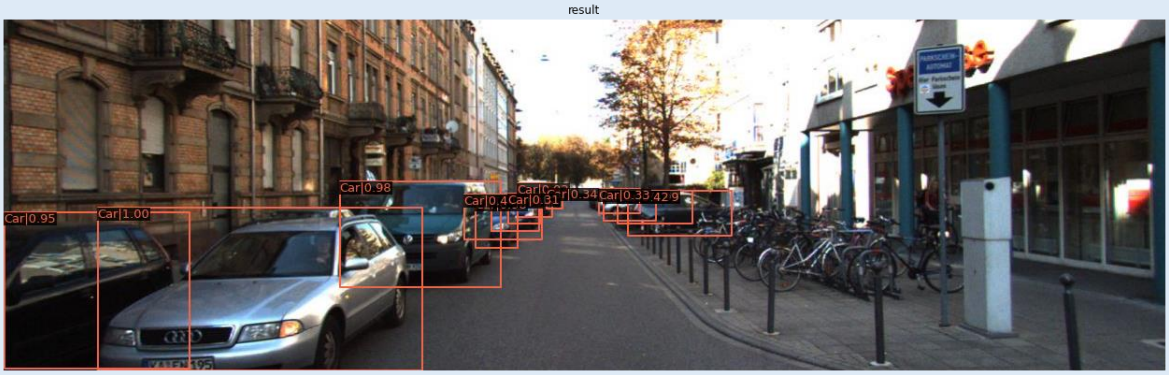


Figure 5 Result after 2nd epoch

4th Epoch

Class	**Gts**	**Dets**	**Recall**	**AP**
Car	*14385*	*21304*	*0.927*	*0.89*
Pedestrian	*2280*	*4054*	*0.75*	*0.661*
Cyclist	*893*	*1170*	*0.594*	*0.525*
mAP				*0.692*

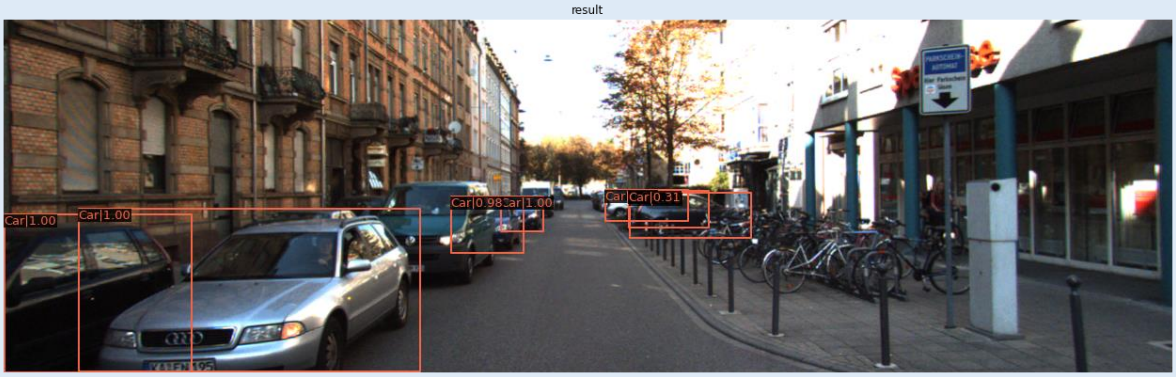


Figure 6 Result after 4th epoch

Code for this part is here

Note: while training, number of detections became closer to ground truth but recall and AP became lower. Maybe it needs to train more and use different learning rate and different threshold as well. Currently used threshold is 0.5.

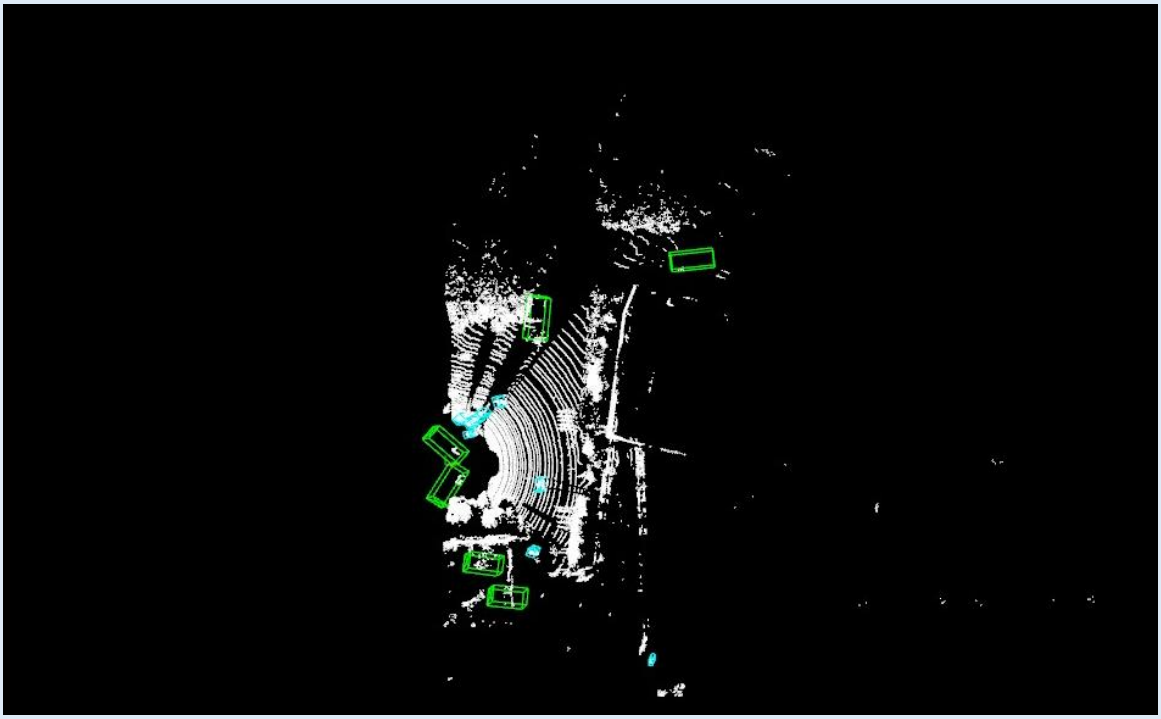
4. 3D Backbone Network → PV-RCNN

*In the paper, They first pre-train PV-RCNN about **60** epochs for the LiDAR stream on the KITTI dataset.*

Here, using a pre-trained model, trained on KITTI for 80 epochs and its AP is (83.61, 57.90, 70.47) for Car, Pedestrian and cyclist respectively.

So, what done in this part is update some parts in the code to make it working on Colab and save the results to download it and see it locally,

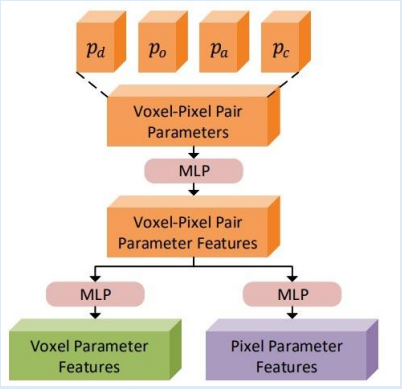
Its result is like:



5. Parameter Feature Generating Module

It's to generate parameter features to represent the deterministic characteristics of each voxel-pixel.

- *Density parameter*
- *Occlusion parameter*
- *Area parameter*
- *Contrast parameter*



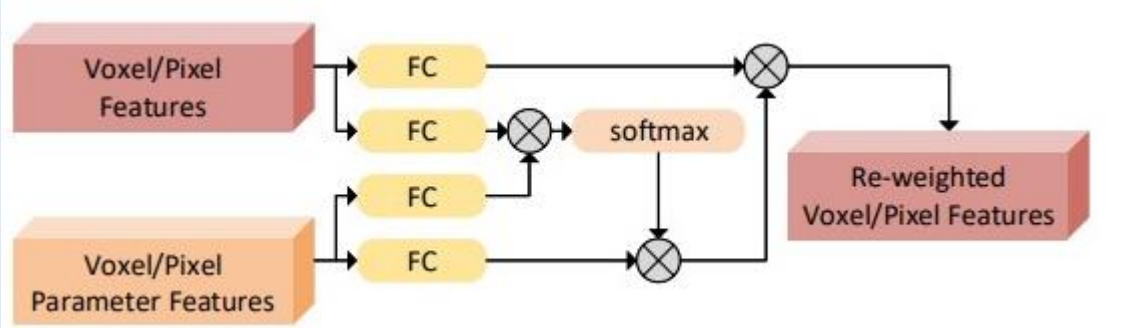
p` = M(p) where R4 → R16
Pv = M(p`) where R16 → RCv
Pp = M(p`) where R16 → RCp

```
class PFG(nn.Module):
    def __init__(self):
        super(PFG, self).__init__()
        self.m = nn.Linear(4, 16)
        self.mv = nn.Linear(16,Cv)  # Cv stands for the number of channels for a voxel
        self.mp = nn.Linear(16,Cp)  # Cp stands for the number of channels for a pixel

    def forward(self, pd, po, pa, pc):
        # voxel-pixel pair parameter
        p = [pd, po, pa, pc]
        # voxel-pixel pair parameter features
        p_dash = self.m(p)
        # voxel parameter features and pixel parameter features
        pv = self.mv(p_dash)
        pp = self.mp(p_dash)
        return pv, pp, p_dash
```

6. Parameter-Based Weighting Module

These features would be used to initiatory re-weight the voxel features from the LiDAR feature map and the pixel features from the camera feature map.
Features of a voxel coming from the LiDAR/Camera feature map are expressed by bv/bp.



$$b'_v = \mathrm{softmax}(\mathrm{B}^1_v(b_v) \circ \mathrm{P}^1_v(p_v)) \circ \mathrm{P}^2_v(p_v) \circ \mathrm{B}^2_v(b_v), \quad (3)$$

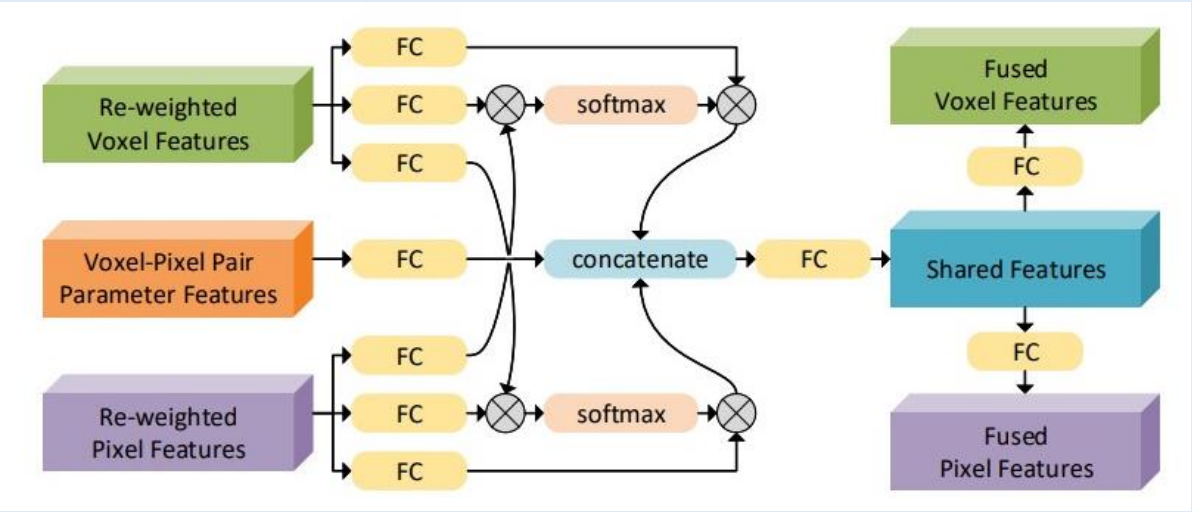
$$b'_p = \mathrm{softmax}(\mathrm{B}^1_p(b_p) \circ \mathrm{P}^1_p(p_p)) \circ \mathrm{P}^2_p(p_p) \circ \mathrm{B}^2_p(b_p), \quad (4)$$

```
class PBW(nn.Module):
    def __init__(self):
        super(PBW, self).__init__()
        self.Bv1 = nn.Linear(Cv, Cv)
        self.Bv2 = nn.Linear(Cv, Cv)
        self.Pv1 = nn.Linear(Cv, Cv)
        self.Pv2 = nn.Linear(Cv, Cv)
        self.sv = nn.Softmax()
        self.Bp1 = nn.Linear(Cp, Cp)
        self.Bp2 = nn.Linear(Cp, Cp)
        self.Pp1 = nn.Linear(Cp, Cp)
        self.Pp2 = nn.Linear(Cp, Cp)
        self.sp = nn.Softmax()
    def forward(self, bv, bp, pv, pp):
#       Features of a voxel coming from the LiDAR feature map are expressed by bv
#       features of a pixel coming from the camera feature map are symbolized by bp
#       The corresponding voxel parameter feature vector coming from the PFG is represented as pv
#       The corresponding pixel parameter feature vector coming from the PFG is represented as pp
        bv_dash = self.sv(self.Bv1(bv)* self.Pv1(pv))*self.Pv2(pv)*self.Bv2(bv)
        bp_dash = self.sp(self.Bp1(bp)* self.Pp1(pp))*self.Pp2(pp)*self.Bp2(bp)
        return bv_dash, bp_dash
```

7. Voxel-Pixel Fusion Module

The re-weighted voxel features, re-weighted pixel features, and voxel-pixel pair parameter features are combined and transformed into the fused voxel features and pixel features.

The fused voxel features and fused pixel features, which serve as residuals, are appended to the original features. Finally, fused feature maps substitute the original feature maps and return to backbone networks.



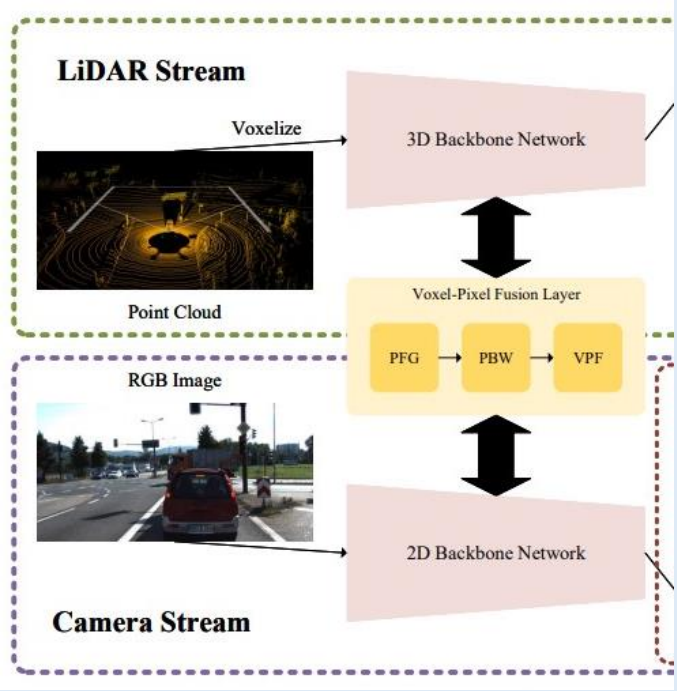

```
class VPF(nn.Module):
    def __init__(self):
        super(VPF, self).__init__()
        self.Bv3 = nn.Linear(Cv, Cv)
        self.Bv4 = nn.Linear(Cv, Cv)
        self.Bv5 = nn.Linear(Cv, Cv)
        self.Bp3 = nn.Linear(Cp, Cp)
        self.Bp4 = nn.Linear(Cp, Cp)
        self.Bp5 = nn.Linear(Cp, Cp)
        self.P = nn.Linear(16, 16)
        self.sp = nn.Softmax()
        self.sv = nn.Softmax()
        self.S = nn.Linear(Cv+Cp+16, Cv+Cp+16)
        self.Sp = nn.Linear(Cv+Cp+16, Cp)
        self.Sv = nn.Linear(Cv+Cp+16, Cv)

    def forward(self, bv_dash, bp_dash, p_dash):
        bvt = self.sv(self.Bv3(bv_dash)*Bp3(bp_dash))*(Bv5(bv_dash))
        bpt = self.sp(self.Bv4(bv_dash)*Bp4(bp_dash))*(Bp5(bp_dash))
        pt = self.P(p_dash)
        s = self.S(torch.cat((bvt,bpt,pt),0))
        bv_2d = self.Sv(s)
        bp_2d = self.Sp(s)
        return bv_2d, bp_2d
```

8. Voxel-Pixel Fusion Layer

As shown in this figure, they put a layer between the 2 backbone networks to apply fusion.

In the paper, for 3D backbone network side, they mentioned that they insert the VPF layer to the first, the second, or the third down-sampling layer. The results show that bridging the VPF layer on the second down sampling layer pair for all the classes is the best choice.



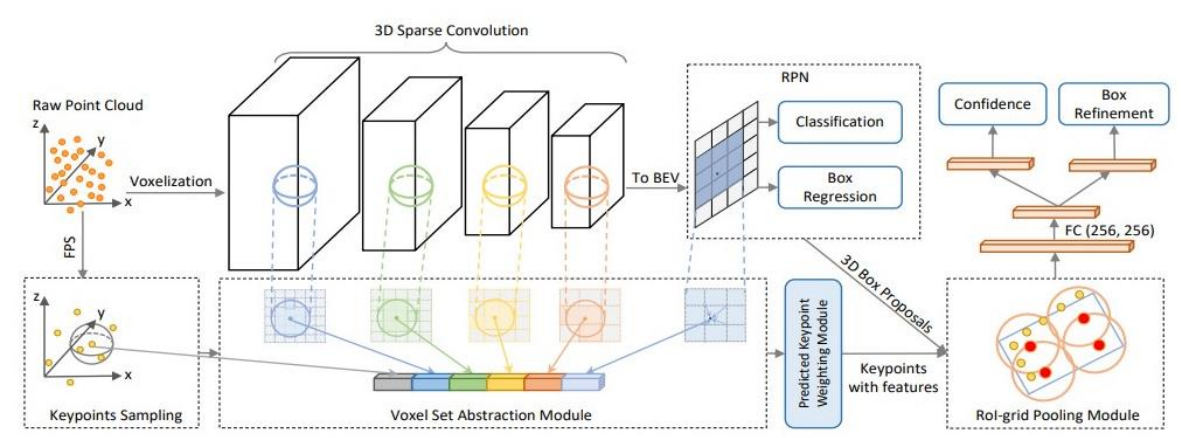


Figure 7 PV-RCNN Model, 3D Sparse convolution part is that responsible for down sampling

For the other side, 2D backbone layer, they didn't mention where they put the layer, so I decided to put it after the conv network
"I" is input image,
"conv" backbone convolutions,
"pool" region-wise feature extraction,
"H" network head,
"B" bounding box,
"C" classification.
"B0" is proposals in all architectures.

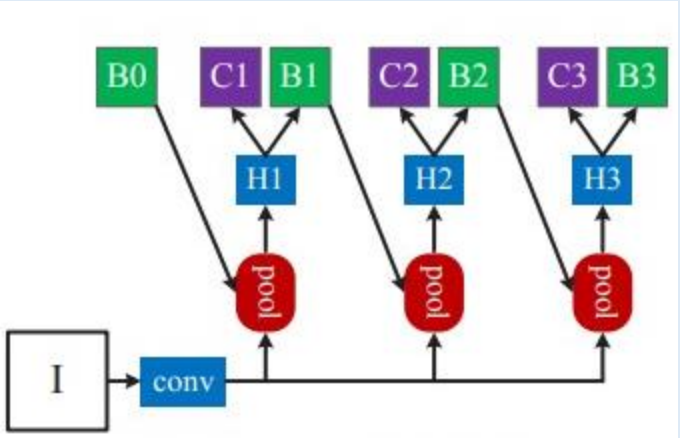
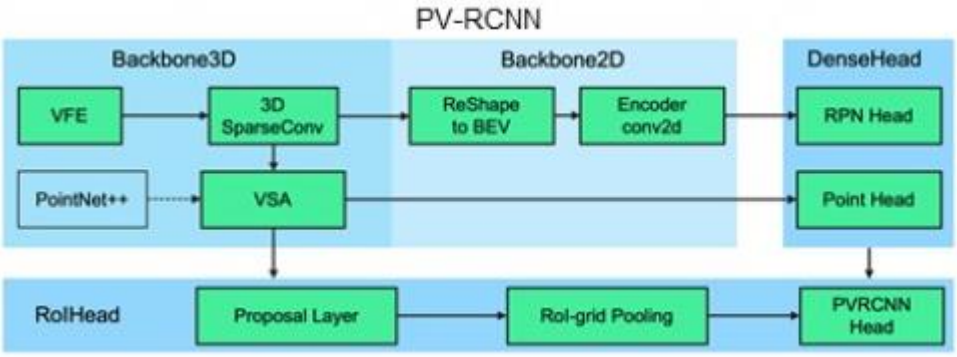


Figure 8 Cascaded RCNN Model

Unresolved Problems Faced in This Part

- Hard to get all voxels from PV-RCNN to calculate occlusion parameter and area parameter as I couldn't understand all its implementation steps in right way.



  This figure is the illustration of the implementation in OpenPCDet, I couldn't understand outputs of some modules which made me not able to use it.

- Dimensions problem when implementing forward function for VPFNet due to differences between PV-RCNN and Cascaded RCNN as they used different dimensions and I couldn't change them.

Code for these parts "from PFG module to VPF layer", here

CODE

In this folder on drive https://drive.google.com/drive/folders/1KbWYcQFVECETxJfH1M0zf8MEy_ZlQ5y2
Data and models uploaded in it, and there are 5 notebooks:

1. Data preparation
2. Visualization “work only locally”
3. Cascaded RCNN “to run and fine-tune it”
4. PV RCNN “to run the demo of the model”
5. VPFNet “to build the model, its modules and use previous two models”

Note: they will run correctly with OpenPCDet and mmdetection that are in the drive folder as some changes I made in it to make it work correctly on colab.

REFERENCES

VPFNet Paper: https://arxiv.org/abs/2111.00966

Cascaded RCNN
Paper: https://arxiv.org/abs/1906.09756
Code: https://github.com/open-mmlab/mmdetection/tree/588536de9905feb7f37c2c977d146a64c74ef28e
Helping Links:
https://colab.research.google.com/github/open-mmlab/mmdetection/blob/master/demo/MMDet_Tutorial.ipynb#scrollTo=Wi4LPmsR66sy
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/finetuning_torchvision_models_tutorial.ipynb#scrollTo=rxx4jNElzrbQ

PV RCNN
Paper: https://arxiv.org/abs/1912.13192
Code: https://github.com/open-mmlab/OpenPCDet
Helping Links: https://www.youtube.com/watch?v=a0jR_xChKfA
https://github.com/open-mmlab/OpenPCDet/issues/273

Dataset
Download: http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d
VoxelNet code to preprocess: https://github.com/gkadusumilli/Voxelnet

Split models and reuse specific layers:
https://medium.com/analytics-vidhya/how-to-add-additional-layers-in-a-pre-trained-model-using-pytorch-5627002c75a5
https://cameleonx.com/blog/2018/03/26/how-to-split-a-pre-trained-torch-model/
https://github.com/mortezamg63/Accessing-and-modifying-different-layers-of-a-pretrained-model-in-pytorch
https://discuss.pytorch.org/t/dynamically-insert-new-layer-at-middle-of-a-pre-trained-model/66531/3

Another helping links:
https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/
https://github.com/traveller59/second.pytorch/tree/3aba19c9688274f75ebb5e576f65cfe54773c021