Using a Waffle Iron for Automotive Point Cloud Semantic Segmentation — Supplementary Material —

Gilles Puy[1] Alexandre Boulch[1] Renaud Marlet[1,2]

[1]valeo.ai, Paris, France [2]LIGM, Ecole des Ponts, Univ Gustave Eiffel, CNRS, Marne-la-Vallée, France

Contents

A. WaffleIron Implementation

We present in Listing 1 an example of a code implementing the WaffleIron backbone in PyTorch [3]. We recall that this backbone takes as input point tokens provided by an embedding layer and outputs updated point tokens used in a linear classification layer for semantic segmentation. The implementation consists of applications of basic layers directly to each point tokens (batch normalizations, 1D and 2D convolutions, matrix-vector multiplications).

The step which is, maybe, the most technical to implement is the construction of the sparse matrices (line 56 of Listing 1) for projections from 3D to 2D. For completeness, we provide the corresponding code as well in Listing 2. Creating these sparse matrices requires computing the mapping between each 3D point and each 2D cell. Note that the sole computations needed to get this mapping reduces to lines 15 and 17 of Listing 2. The rest and majority of the code concerns the creation of arrays to build the corresponding sparse matrices.

B. Instance Cutmix and Polarmix on SemanticKITTI

In complement to Sec. 4.5 in the main paper, we show in Fig. 4 the benefit of combining the augmentations polarmix [4] and instance cutmix [6, 5] over instance cutmix alone for training on SemanticKITTI [1]. The combination allows to us to improve the mIoU% by 1.5 point on average, with the most notable improvements in the classes bicycle, other-vehicle and person.

Architecture & Training hyparameters	mIoU%
$F = 256$ & 3-dim $\boldsymbol{h}_i$	76.1
$F = 256$ & 5-dim $\boldsymbol{h}_i$	76.6
$F = 384$ & 5-dim $\boldsymbol{h}_i$	76.9
$F = 384$ & 5-dim $\boldsymbol{h}_i$ & Stoch. depth	77.6

Table 5. Influence of F, input vectors $\boldsymbol{h}_i$, and stochastic depth on the performance of WaffleIron on nuScenes. We train and evaluate WaffleIron-48-F backbones on the official train and validation set, respectively. We report the average mIoU% obtained at the last training epoch of two independent runs.

C. Input Features, dimension F, and Stochastic Depth on nuScenes

We present in Tab. 5 the interest of successively: using 5-dimensional input vectors $\boldsymbol{h}_i$ (intensity, x, y, z, and range of $\boldsymbol{p}_i$) instead of 3-dimensional input vectors (intensity, height=z, range); increasing F from 256 to 384; and using stochastic depth during training on nuScenes. We notice that each of these ingredients improves the mIoU% to finally reach 77.6 on average over two independent training.

D. Visual Inspections

Segmentation results. We present in Fig. 5 and Fig. 6 visualizations of semantic segmentation results obtained with our method on the validation set of nuScenes [2] and SemanticKITTI [1], respectively. The official color codes for these visualizations are recalled in Fig. 7. We notice that, overall, the segmentation are of good quality. Nevertheless, we remark sometimes confusion between the sidewalk and the road on nuScenes (row 1 and 3 in Fig. 5). We notice as well some wrongly classified points when the vegetation overlaps a building in the last row of Fig. 5. On SemanticKITTI, we notice essentially some confusion between terrain and vegetation, especially in row 1 and 3 of Fig. 6.

2D features maps. For illustration purposes, we provide

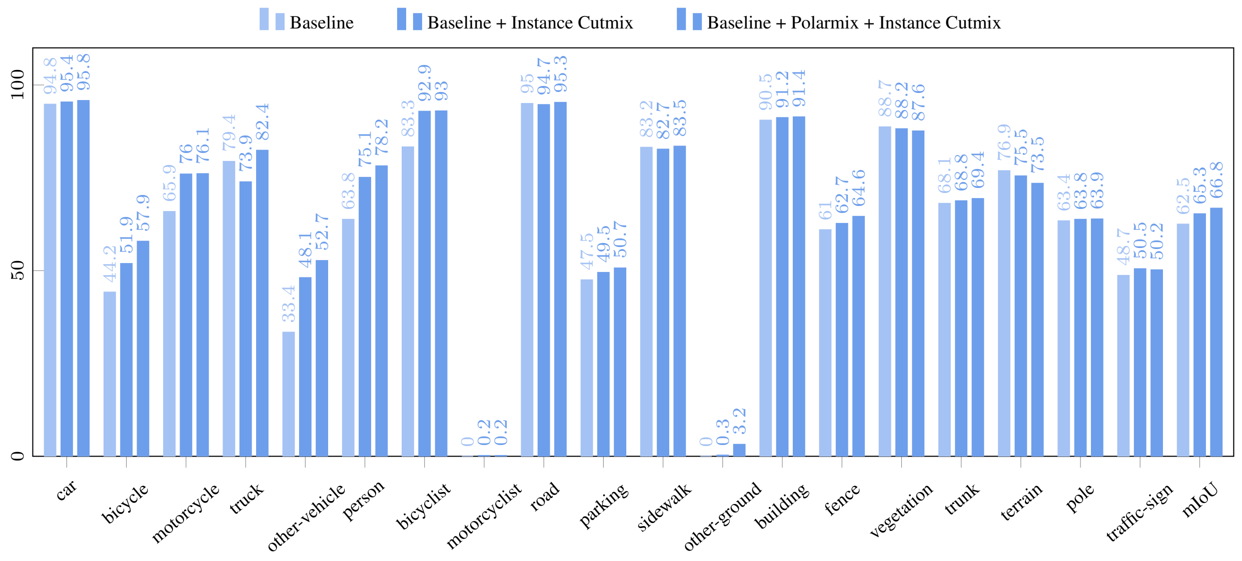

Figure 4. Performance of WaffleIron when using baseline augmentations (rotation, flip axis, scaling), when adding instance cutmix, or when adding instance cutmix and polarmix together. We train a WaffleIron-48-256 backbone on the train set of SemanticKITTI and compute the mIoU on the corresponding validation set. We report the average mIoU% obtained at the last training epoch of two runs.

in Fig. 8 and Fig. 9 visualizations of 2D feature maps obtained after projection at different layers ℓ of WaffleIron, for nuScenes and SemanticKITTI, respectively.

E. Number of Parameters and Inference time

The largest WaffleIron models that we trained in this work, WaffleIron-48-256 and WaffleIron-48-384, contain only 6.8 M and 15.1 M trainable parameters, respectively. This stays smaller than, e.g., Cylinder3D [7], which has more than 50 M parameters.

The inference time for WaffleIron-48-256 is provided in the core of the paper. This inference time does not include data pre-processing. Yet, the only noticeable extra step required in our method for data pre-processing, compared to networks using sparse convolutions, is the nearest neighbor search required to compute the point tokens in the embedding layer. Note that this embedding layer is not tied to our proposed backbone WaffleIron; one could design other embedding layers not requiring this nearest neighbor search.

In order to further accelerate inference while keeping the simplicity of implementation of WaffleIron, we can think of the following possibilities which we leave for future work.

- Reduce the number of point tokens by increasing the voxel size used for voxel-downsampling during pre-processing. We used square voxels of size 10 cm, while the 2D grids in the WI blocks have a resolution of 60 cm on nuScenes, and 40 cm on SemanticKITTI. We can probably downsample the point clouds further during pre-processing with limited impact on the performance.
- Construct a new embedding layer that outputs a reduced number of point tokens, especially in regions that are highly sampled by the lidar and that contain redundant information.

```python
 1 import torch
 2 import numpy as np
 3 import torch.nn as nn
 4 
 5 
 6 class ChannelMix(nn.Module):
 7     def __init__(self, channels):
 8         super().__init__()
 9         # Number of channels denoted by F in the paper
10         F = channels
11         # Layers in channel mixing step
12         self.norm = nn.BatchNorm1d(F)
13         self.mlp = nn.Sequential(nn.Conv1d(F, F, 1), nn.ReLU(), nn.Conv1d(F, F, 1))
14         self.layerscale = nn.Conv1d(F, F, 1, bias=False, groups=F)
15 
16     def forward(self, tokens):
17         return tokens + self.layerscale(self.mlp(self.norm(tokens)))
18 
19 
20 class TokenMix(nn.Module):
21     def __init__(self, channels, grid_shape):
22         super().__init__()
23         # Shape of 2D grid on projection plane
24         self.H, self.W = grid_shape
25         # Number of channels denoted by F in the paper
26         F = channels
27         # Layers in token mixing step
28         self.norm = nn.BatchNorm1d(F)
29         self.ffn = nn.Sequential(
30             nn.Conv2d(F, F, 3, padding=1, groups=F), nn.ReLU(), nn.Conv2d(F, F, 3, padding=1, groups=F)
31         )
32         self.layerscale = nn.Conv1d(F, F, 1, bias=False, groups=F)
33 
34     def forward(self, tokens, sp_mat):
35         B, C, N = tokens.shape
36         # Flatten
37         residual = torch.bmm(sp_mat["flatten"], self.norm(tokens).transpose(1, 2)).transpose(1, 2)
38         # FFN with channel-wise 2D convolutions with kernels of size 3 x 3
39         residual = self.ffn(residual.reshape(B, C, self.H, self.W)).reshape(B, C, self.H * self.W)
40         # Inflate
41         residual = torch.bmm(sp_mat["inflate"], residual.transpose(1, 2)).transpose(1, 2)
42         return tokens + self.layerscale(residual.reshape(B, C, N))
43 
44 
45 class WaffleIron(nn.Module):
46     def __init__(self, channels, depth, grids_shape):
47         super().__init__()
48         self.grids_shape = grids_shape
49         self.channel_mix = nn.ModuleList([ChannelMix(channels) for _ in range(depth)])
50         self.token_mix = nn.ModuleList(
51             [TokenMix(channels, grids_shape[l % len(grids_shape)]) for l in range(depth)]
52         )
53 
54     def forward(self, tokens, non_zeros_ind):
55         # Build projection matrices
56         sp_mat  = [build_proj_matrix(ind, tokens.shape[0], np.prod(sh))
57                    for ind, sh in zip(non_zeros_ind, self.grids_shape)]
58         # Forward pass in backbone
59         for l, (smix, cmix) in enumerate(zip(self.token_mix, self.channel_mix)):
60             tokens = smix(tokens, sp_mat[l % len(sp_mat)])
61             tokens = cmix(tokens)
62         return tokens
```

Listing 1. **Pytorch implementation of WaffleIron**. This backbone takes as input point tokens provided by an embedding layer, and outputs updated point tokens used in a linear classification layer for semantic segmentation. The implementation of the embedding layer and the classification layer are not presented here. The code to construct the sparse projection matrices on line 57 is presented in Listing 2.

```python
def get_non_zeros_ind(point_coord, plane_axes, grid_shape, fov_xyz_min, resolution):
    """
    Mapping between point indices and 2D cell indices for the projection from 3D to 2D.
    Inputs:
      'point_coord': xyz-coordinates of the points to project (array of size num_points x 3).
      'planes_axes': axis encoding of projection planes, e.g., 'planes_axes=(0,1)' for the (x,y)-plane.
      'grid_shape': shape of 2D grid on projection plane, e.g., 'grid_shape=(128,128)'.
      'fov_xyz_min': lowest xyz-bounds of the FOV (array of size 1 x 3)
      'resolution': resolution of 2D grid (scalar)
    Output:
      indices of non-zero entries in sparse matrix for the projection from 3D to 2D.
    """

    # Quantize point cloud coordinates at desired resolution
    quant = ((point_coord - fov_xyz_min)[:, plane_axes] / resolution).astype('int')
    # Transform quantized coordinates to 2D cell indices
    cell_indices = quant[:, 0] * grid_shape[1] + quant[:, 1]

    # Indices of non-zeros entries in sparse matrix for projection from 3D to 2D.
    num_points = quant.shape[0]
    indices_non_zeros = torch.cat([
        # Batch index (batch size of 1 here)
        torch.zeros(1, num_points).long(),
        # Index of corresponding 2D cell for each point
        torch.from_numpy(cell_indices).long().reshape(1, num_points),
        # Index of each point
        torch.arange(num_points).long().reshape(1, num_points)
    ], axis=0)

    return indices_non_zeros


def build_proj_matrix(indices_non_zeros, batch_size, num_2d_cells):
    """
    Construct sparse matrices for the projection from 3D to 2D and vice versa.
    Inputs:
      'indices_non_zeros': indices of non-zero entries in sparse matrix for projecting from 3D to 2D.
      'batch_size':  batch size.
      'num_2d_cells': number of cells in the 2D grid.
    Outputs:
      sparse projection matrices for the Flatten and Inflate steps.
    """
    num_points = indices_non_zeros.shape[1]
    matrix_shape = (batch_size, num_2d_cells, num_points)

    # One non-zero coefficient per point (set to 1) in sparse matrix for inflate step
    ones = torch.ones(batch_size, num_points, 1, device=indices_non_zeros.device)

    # Sparse projection matrix for Inflate step
    inflate = torch.sparse_coo_tensor(indices_non_zeros, ones.reshape(-1), matrix_shape)
    inflate = inflate.transpose(1, 2)

    # Count number of points in each cells (used in Flatten step)
    num_points_per_cells = torch.bmm(inflate, torch.bmm(inflate.transpose(1, 2), ones))

    # Sparse projection matrix for Flatten step (projection & average in each 2d cells)
    weight_per_point = 1. / num_points_per_cells.reshape(-1)
    flatten = torch.sparse_coo_tensor(indices_non_zeros, weight_per_point, matrix_shape)

    return {"flatten": flatten, "inflate": inflate}
```
Listing 2. Code to construct the sparse projection matrices used in WaffleIron. Note that we build two matrices for efficiency: one for the Flatten step ('flatten') and one for the Inflate step ('inflate'). The matrix 'flatten' combines (i) projection to 2D and (ii) averaging in each 2D cell, i.e., implements Eq. (4) directly. The matrix 'inflate' corresponds to S in Eq. (5).

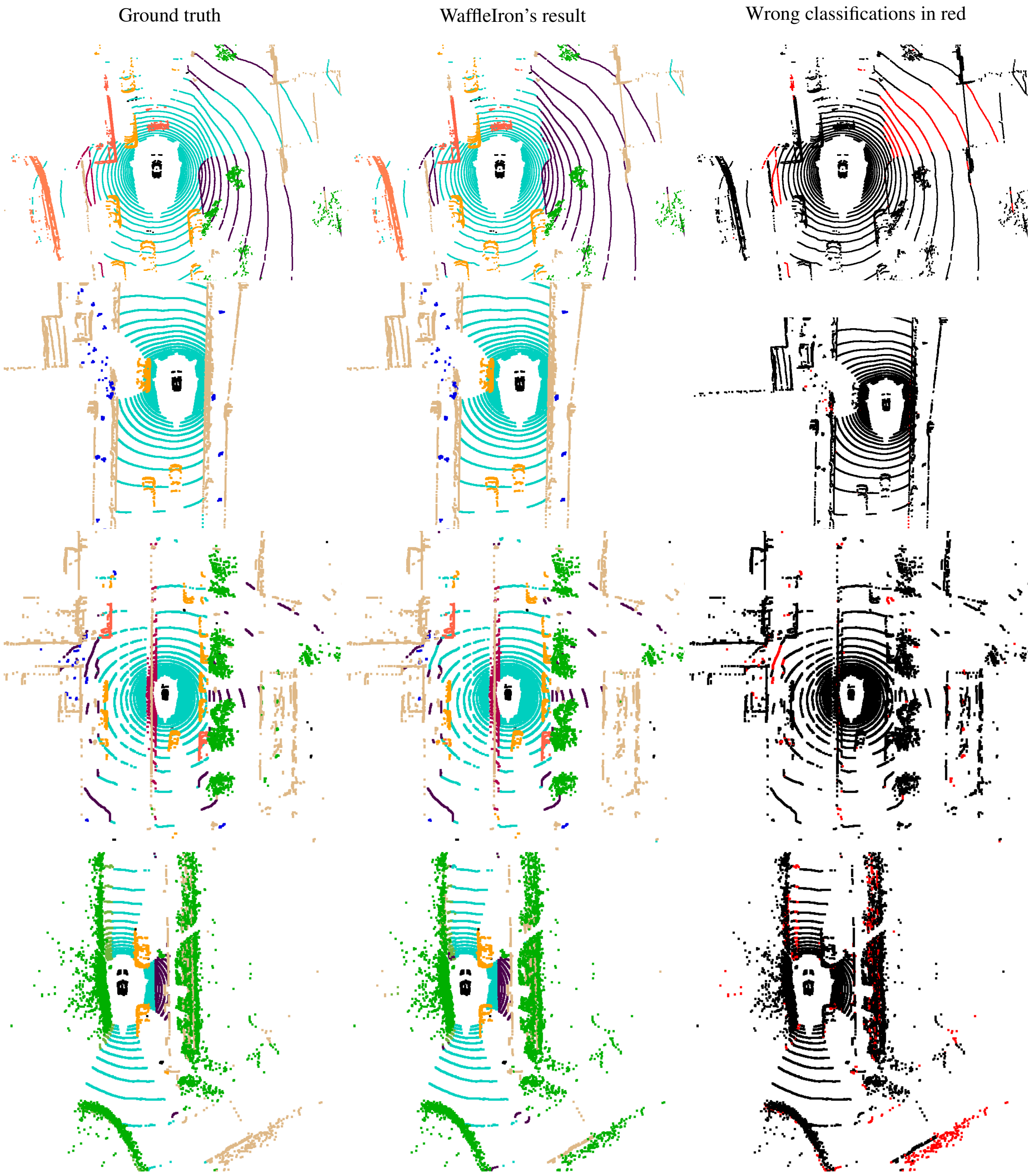

Figure 5. Visualization of semantic segmentation results on the validation set of nuScenes obtained with WaffleIron.

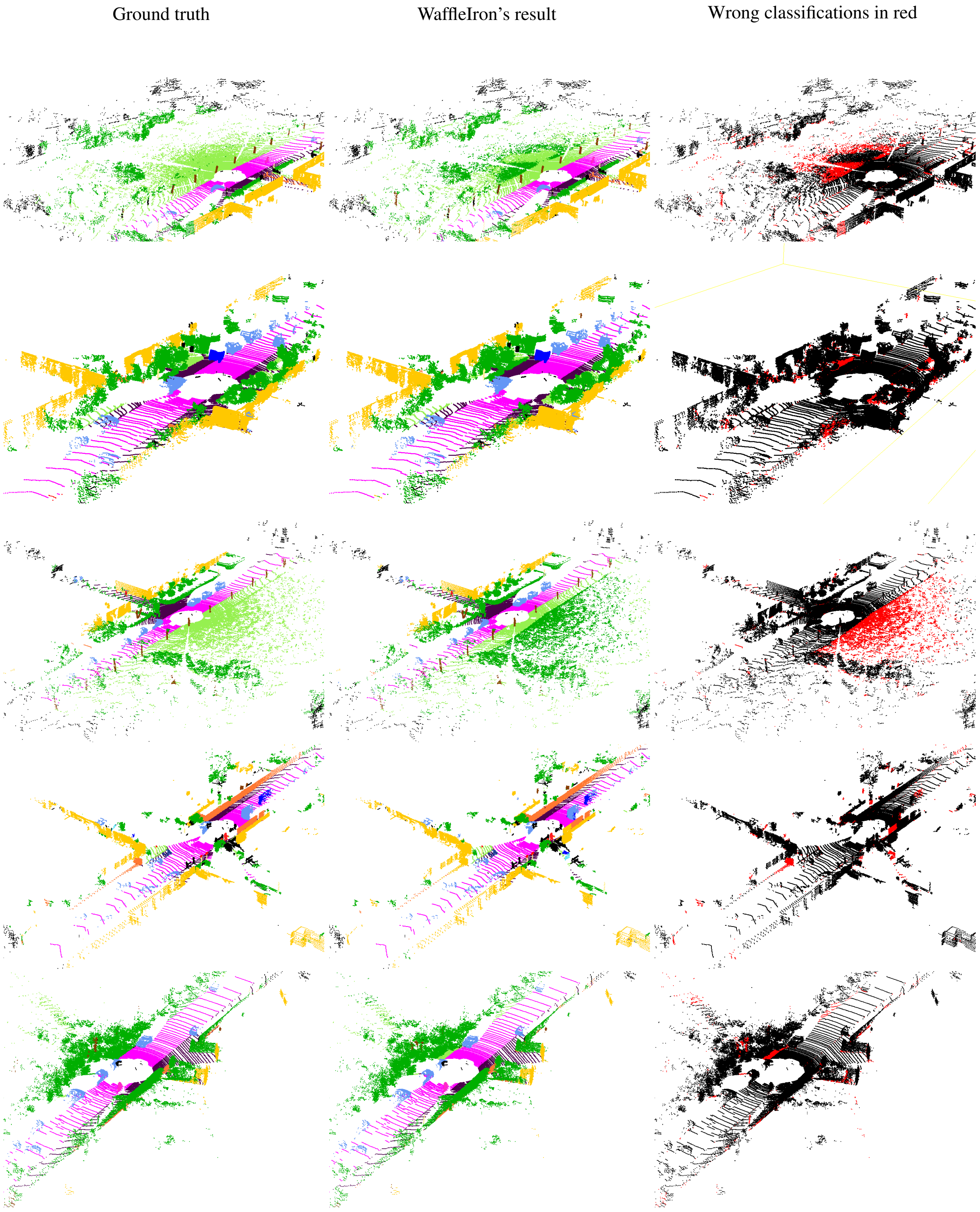

Figure 6. Visualization of semantic segmentation results on the validation set of SemanticKITTI obtained with WaffleIron.

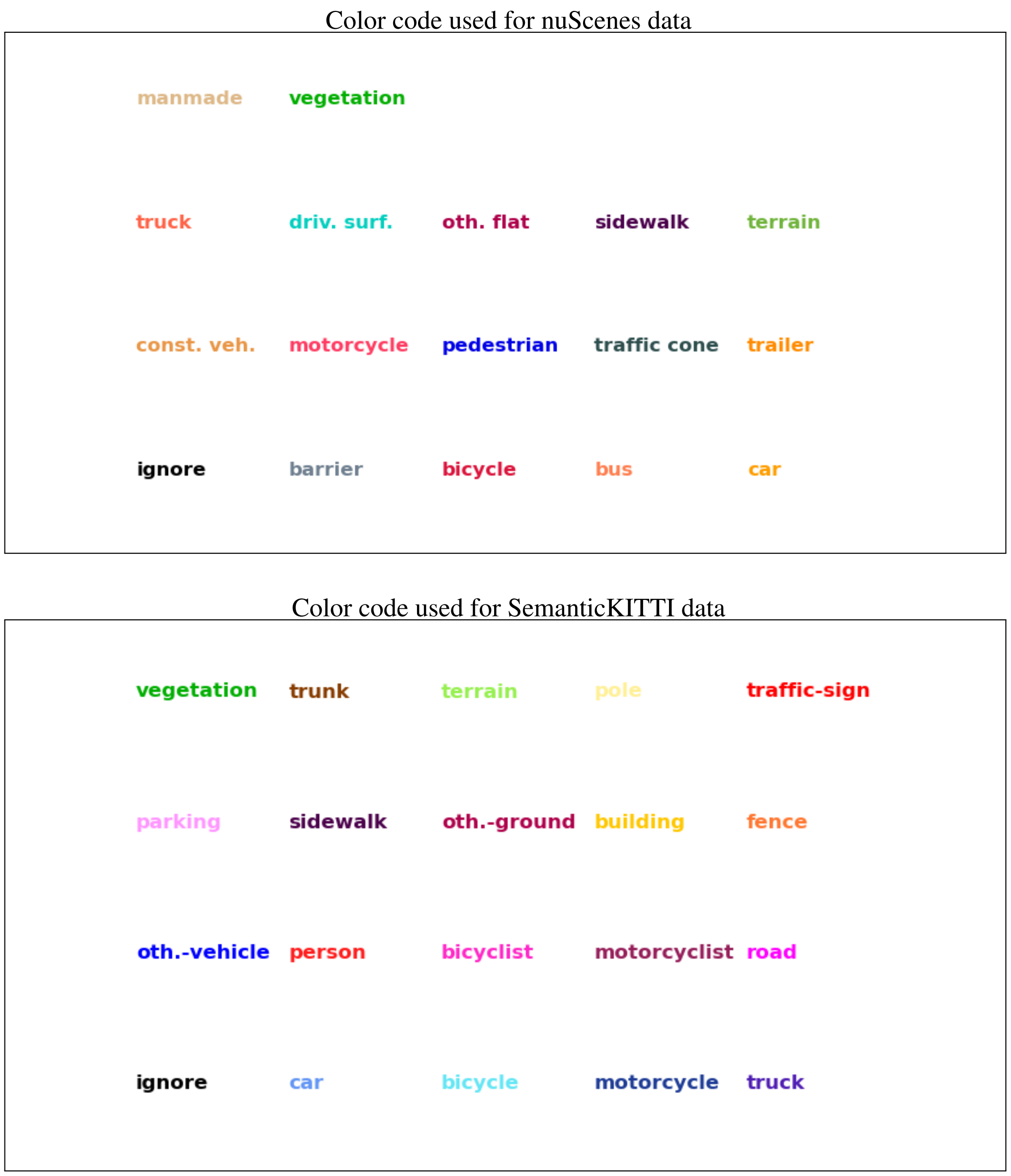

Figure 7. Color code used to represent each class on nuScenes (top) and SemanticKIITI (bottom).

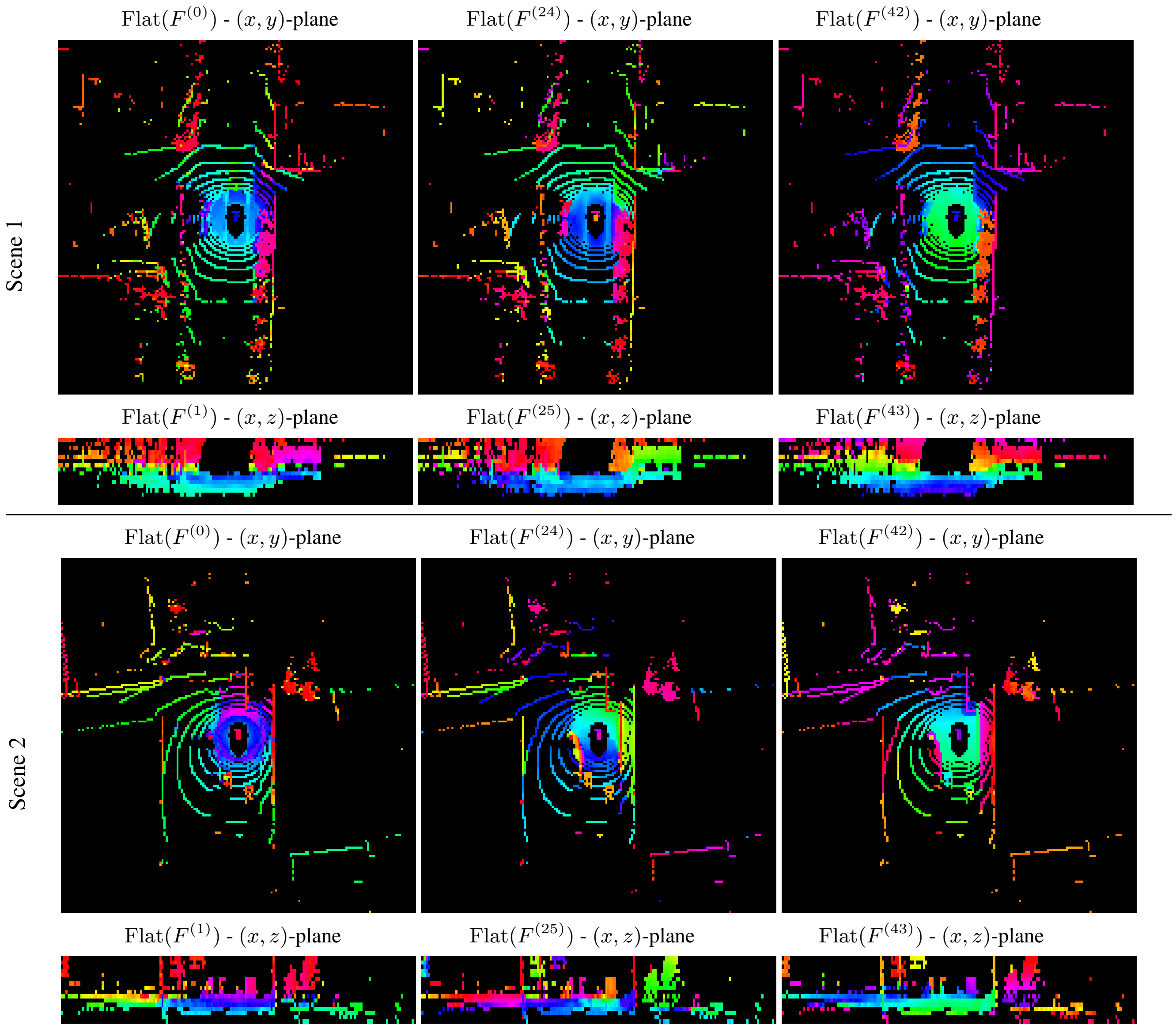

Figure 8. Visualization of 2D features maps obtained after the Flatten step at different layers ℓ of WaffleIron on two scenes of the validation set of nuScenes. The feature maps are colored by reducing the F-dimensional features to a 3-dimenional space using t-SNE.

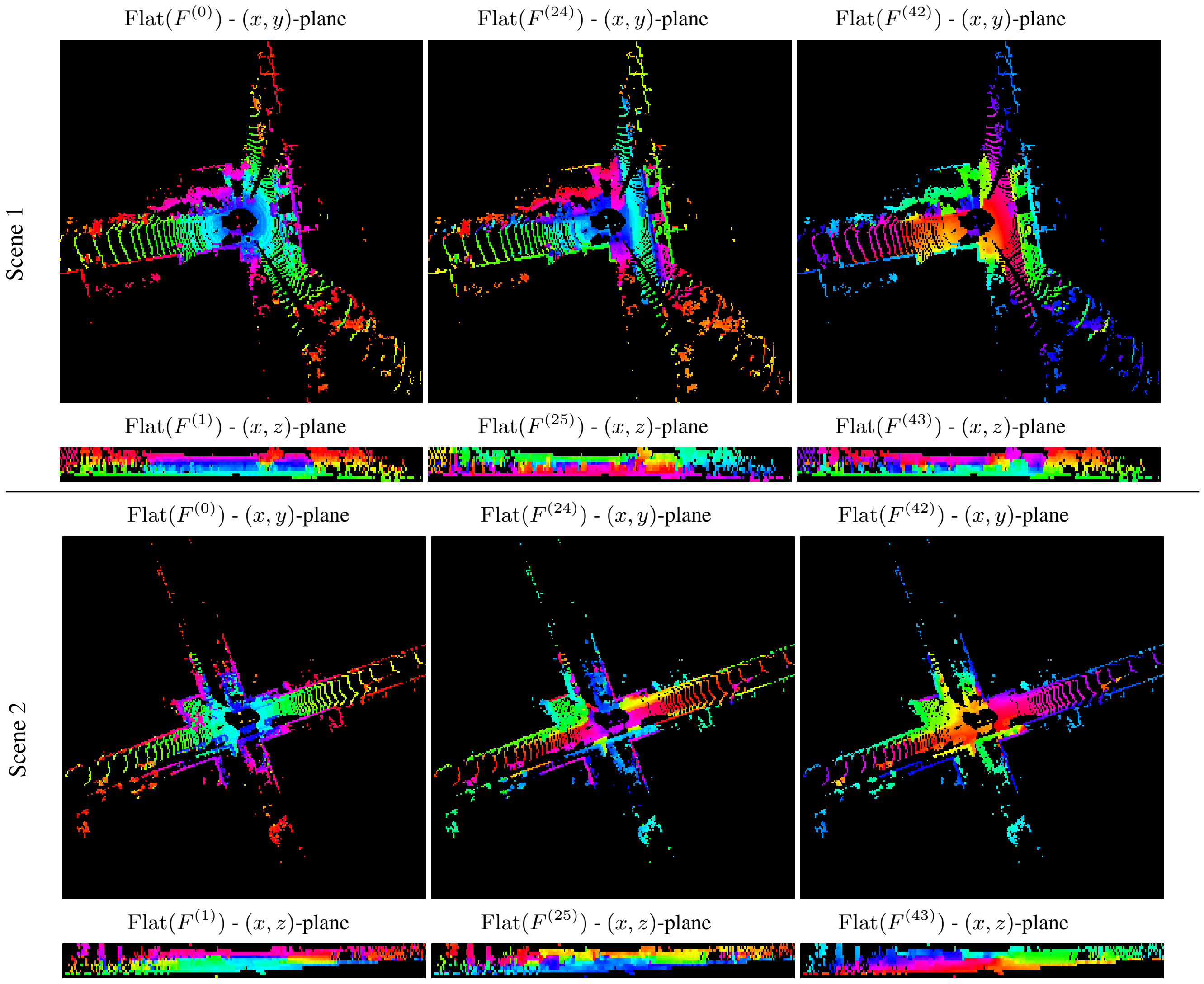

Figure 9. Visualization of 2D features maps obtained after projection at different layers ℓ of WaffleIron on two scenes of the validation set of SemanticKITTI. The feature maps are colored by reducing the F-dimensional features to a 3-dimensional space using t-SNE.

References

[1] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *ICCV*, 2019. 1

[2] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A multimodal dataset for autonomous driving. In *CVPR*, 2020. 1

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, 2019. 1

[4] Aoran Xiao, Jiaxing Huang, Dayan Guan, Kaiwen Cui, Shijian Lu, and Ling Shao. PolarMix: A General Data Augmentation Technique for LiDAR Point Clouds. In *NeurIPS*, 2022. 1

[5] Jianyun Xu, Ruixiang Zhang, Jian Dou, Yushi Zhu, Jie Sun, and Shiliang Pu. RPVNet: A Deep and Efficient Range-Point-Voxel Fusion Network for LiDAR Point Cloud Segmentation. In *ICCV*, 2021. 1

[6] Xu Yan, Jiantao Gao, Chaoda Zheng, Chao Zheng, Ruimao Zhang, Shuguang Cui, and Zhen Li. 2DPASS: 2D Priors Assisted Semantic Segmentation on LiDAR Point Clouds. In *ECCV*, 2022. 1

[7] Xinge Zhu, Hui Zhou, Tai Wang, Fangzhou Hong, Yuexin Ma, Wei Li, Hongsheng Li, and Dahua Lin. Cylindrical and Asymmetrical 3D Convolution Networks for LiDAR Segmentation. In *CVPR*, 2021. 2