Supplementary Material

7. Limitation and Future Work

We further discuss the limitations of our work, which will be the focus in future. Firstly, although we find that explicit position embedding will promote performance significantly compared with other manners of position embedding, the rationale still remains unexplored. One possible direction is to look for theoretical explanations from gradient flows. Secondly, for the neighbor aggregation functions, we categorize them into learnable and non-learnable aggregation and find that the max pooling is a strong parameter-free method that is comparable to the learnable aggregation. Therefore, the next step is to do a finer taxonomy and richer ablation studies to find the applicable aggregation functions in different scenarios.

8. More Instantiation Examples

DGCNN [29]. The neighbor update is described as:

$$\begin{aligned} \boldsymbol{f}'_{\mathcal{N}(i)} &= \text{Group}(f_i), \\ f_j &= \text{MLP}_1(\text{Concat}(f'_j - f_i, f_i)). \end{aligned}$$

Different from PointNet++ [16] that groups the neighbors in coordinate space, the neighbors grouping in DGCNN is conducted in the feature spaces. Similar to PointNet++, DGCNN uses max pooling as the neighbor aggregation function. The point update is absent in the DGCNN block. Additionally, there exists no position embedding function ϕ_e neither explicitly nor implicitly. In another word, there is no relative position information encoded in the output features.

PointCNN [10]. The neighbor update is described as:

$$\begin{aligned} \boldsymbol{f}'_{\mathcal{N}(i)}, \boldsymbol{p}_{\mathcal{N}(i)} &= \text{Group}(f_i, p_i), \\ e_j &= \text{MLP}_1(p_j - p_i), \\ f_j &= \text{Concat}(f'_j, e_j). \end{aligned}$$

The neighbor aggregation is defined as:

$$\begin{aligned} \mathcal{X} &= \text{MLP}_2(\boldsymbol{p}_{\mathcal{N}(i)} - p_i), \\ f_i^{(1)} &= \text{Conv}(\boldsymbol{K}, \mathcal{X} \times \boldsymbol{f}_{\mathcal{N}(i)}), \end{aligned}$$

where $\times$ denotes matrix multiplication and $\boldsymbol{K}$ denotes the trainable convolution kernels of the convolution layer Conv. Two position embedding functions exist in this block. The first one ϕ_1^e is MLP$_1$ used in the neighbor update, while the second one ϕ_2^e is implicitly implemented by Conv$(\boldsymbol{K}, \cdot)$ used in the neighbor aggregation function. Implementing position embedding by convolution layers is a common practice in the ViT family [3, 32].

RandLA-Net [7]. The neighbor update is described as follows:

$$\begin{aligned} \boldsymbol{f}'_{\mathcal{N}(i)}, \boldsymbol{p}_{\mathcal{N}(i)} &= \text{Group}(f_i, p_i), \\ f_j &= \text{MLP}_1(\text{Concat}(f'_j, p_j)). \end{aligned}$$

The position embedding is described as:

$$e^j = \text{MLP}_2(\text{Concat}(p_j - p_i, |p_j - p_i|, p_j, p_i)).$$

The neighbor aggregation is implemented using attentive pooling that is similar to Point Transformer [35]. The neighbor aggregation is as follows:

$$\begin{aligned} M_j &= \text{Softmax}(\text{MLP}_3(\text{Concat}(f_j, e_j))), \\ f_i^{(1)} &= \sum_{j \in \mathcal{N}(i)} (M_j \odot \text{Concat}(f_j, e_j)), \end{aligned}$$

where $M_j \in \mathbb{R}^{2d}$ denotes the attention weights for the neighbor j, $\odot$ denotes the Hadamard product. Finally, the point feature is updated with an MLP layer and a shortcut, then we have

$$f_i^{(2)} = \text{MLP}_4(f_i^{(1)}) + f_i.$$

PointConv [31]. The neighbor update and point update of PointConv is same with PointNet++ [16]. The original neighbor aggregation function of PointConv is defined as follow:

$$\begin{aligned} g_j &= \text{MLP}_1(p_j - p_i), \\ f_i &= \sum_{j \in \mathcal{N}(i)} g_j \times f_j, \end{aligned}$$

where $g_j \in \mathbb{R}^{d_{out} \times d_{in}}$ is the weight matrices that map features from dimension d_{in} to d_{out}, $\times$ denotes matrix multiplication, L denotes the number of the kernel points, $\{p_l | l < L\}$ denotes the coordinates of the kernel points, and $\{W_l | l < L\}$ denotes the associated weights matrices. However, different with the convolution for 2D images, the weight matrices in each local neighborhood for 3D is unique, which results in a huge memory cost. To address this challenge, the authors propose an efficient version, which decouples the weight matrix for each local neighborhood into two parts: a dynamic weight matrix $M_j \in \mathbb{R}^{d_{mid}}$ and a static weight matrix $H \in \mathbb{R}^{d_{out} \times (d_{in} \times d_{mid})}$. In this way, the memory consumption of the generated weights reduces to $\frac{d_{mid}}{N d_{out}}$ of the original version. The efficient version of neighbor aggregation is described as follows:

$$\begin{aligned} M_j &= \text{MLP}_1(p_j - p_i), \\ f'_i &= \sum_{j \in \mathcal{N}(i)} M_j \times f_j^{\top}, \end{aligned} \tag{5}$$

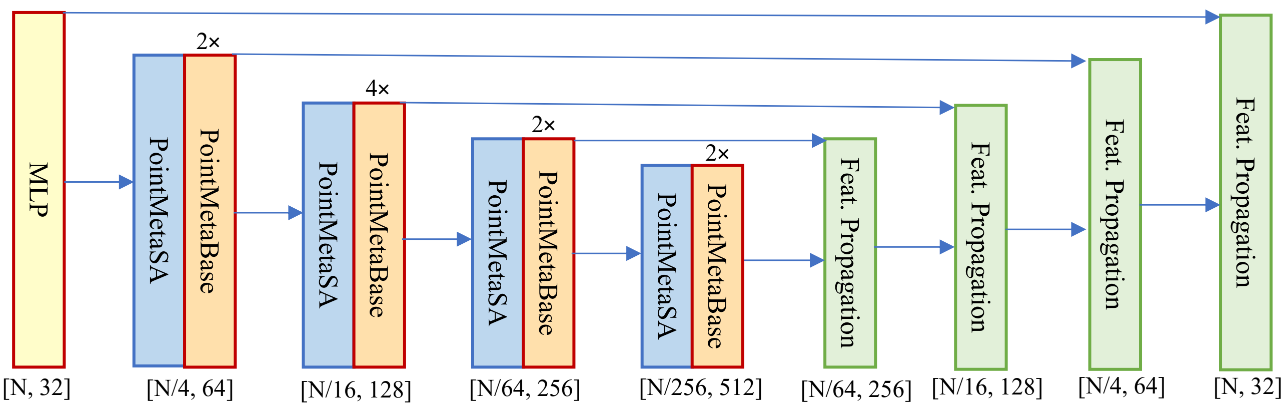

Figure 3. Macro-architecture of PointMetaBase-L. Applying explicit position embedding and the MLP-before-Group order, we tweak the set abstraction module [16, 18] as the reduction block, termed PointMetaSA. We adopt the same scaling strategies and decoder with PointNeXt [18] to construct our PointMetaBase family.

$$f_i = H \times \text{Vec}(f_i'). \tag{6}$$

In Eq. 5, the neighbor features $\{f_j | j \in \mathcal{N}(i)\}$ is transfomed into $f_i' \in \mathbb{R}^{d_{mid} \times d_{in}}$. Then, in Eq. 6, f_i' is turned into a vector by $\text{Vec}(\cdot)$ and then multiply with matrix H.

KPConv [24]. The neighbor update is described as follows:

$$f_j, p_j = \text{Group}(f_i, p_i).$$

The neighbor aggregation is as follows:

$$g_j = \sum_{l=1}^{L} \max(0, 1 - \frac{||p_j - p_l||}{\sigma}) W_l, f_i = \sum_{j \in \mathcal{N}(i)} g_j \times f_j,$$

where $\times$ denotes matrix multiplication, L denotes the number of the kernel points, $\{p_l | l < L\}$ denotes the coordinates of the kernel points, and $\{W_l | l < L\}$ denotes the associated weights matrices. Similar with the original version of PointConv [31], the dynamic weights matrices $\{g_j | j \in \mathcal{N}(i)\}$ across all local neighborhoods will incur huge memory comsumption. Thus in the code implementation KPConv adopt similar strategy that first transforms the neighbor features with dynamic weights and then update point feature with static weights.

PointNeXt [18]. PointNeXt conducts almost the same neighbor update with PointNet++ [16] except for the configuration of the MLP. PointNet++ applies a 3-layer MLP while PointNeXt only applies 1 layer on the neighbor feature f_j' to reduce computation. Besides, PointNeXt also adopts a simple max pooling operation. As for the point update, PointNeXt uses a 2-layer MLP with an inverted bottleneck together with a shortcut layer from the input point feature f_i, then we have:

$$f_i^{(2)} = \text{MLP}_{inv}(f_i^{(1)}) + f_i.$$

9. Macro-Architecture

Applying explicit position embedding and the MLP-before-Group order, we tweak the set abstraction module [16, 18] as the reduction block, termed PointMetaSA. The decoder is composed of a series of feature propagation blocks [16, 18]. As shown in Fig. 3, a 1-layer MLP is used as stem in the first stage. In the later stages, PointMetaSA is placed first and the following are several PointMetaBase blocks. We adopt the same scaling strategies with PointNeXt [18] to construct our PointMetaBase family. The configuration of our PointMetaBase family is summarized as follows:

- PointMetaBase-S : $C = 32, B = 0$
- PointMetaBase-L : $C = 32, B = (2, 4, 2, 2)$
- PointMetaBase-XL : $C = 32, B = (3, 6, 3, 3)$
- PointMetaBase-XXL : $C = 32, B = (4, 8, 4, 4)$

C denotes the channel size of the stem MLP and B denotes the number of the PointMetaBase block in each stage. Note that when $B = 0$, only one PointMetaSA block but no PointMetaBase blocks are used at each stage. Due to the excellent efficiency of PointMetaBase, we do not need to construct the network at the level ”B” as done in PointNeXt [18]. In contrast, we choose to scale the model to a larger level ”XXL”.

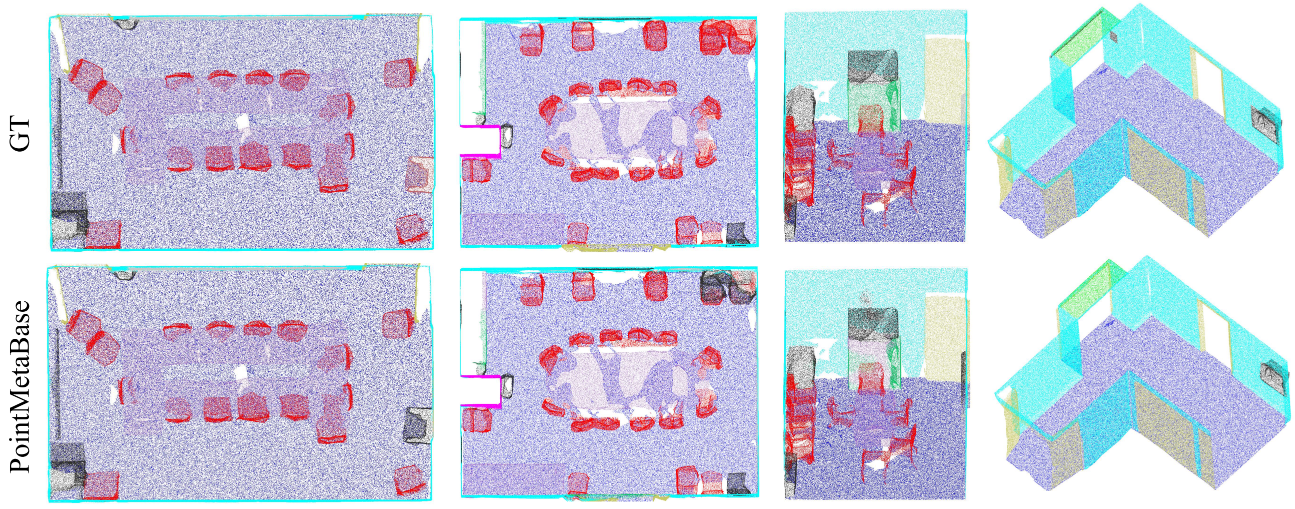

Figure 4. Semantic segmentation results on S3DIS [1]. The fist row is groundtruth, and the second one is predicted by PointMetaBase-XL. Best viewed in color.