PointTree: Transformation-Robust Point Cloud Encoder with Relaxed K-D Trees Supplementary Material

Jun-Kun Chen and Yu-Xiong Wang

University of Illinois at Urbana-Champaign
{junkun3, yxw}@illinois.edu

This document contains additional descriptions (e.g., formal or detailed definition, theoretical proofs, implementation details, etc.) and extra experiments (e.g., segmentation task under projective transformation, overtime accuracy, stability test, etc.). The content of this document is as below:

A Formal Definitions of K-D Trees

This section extends **Section 3.1 Point Cloud Encoder Based on Relaxed K-D Trees** in the main paper with formal and detailed definitions of K-D trees.

Formally, we define a **K-D tree** built on $n = 2^d$ 3D input points P as a full binary tree with $d+1$ **layers** $L_0, \cdots, L_d$ in a top-down order. There are a total of 2^i nodes in L_i. A **leaf node** $o \in L_d$ is corresponding to a unique input $p(o) = p_i$. A **non-leaf** node $o \in L_i$ where $0 \leq i < d$ has two exchangeable **children nodes** $o_l, o_r \in L_{i+1}$, and both of them have a unique **parent node** $\text{par}(o_l) = \text{par}(o_r) = o$. The structure of a K-D tree can be described with a triple $T = (\{L_i\}, \{(o, \{o_l, o_r\})\}, p(\cdot))$.

For each node o, we define its **sub-tree** $\text{sub}(o)$ as the set containing itself and all nodes in $\text{sub}(o_l) \cup \text{sub}(o_r)$ if o is non-leaf. By induction, we know that if $o \in L_i$, then $|\text{sub}(o)| = 2^{d-i}$.

We define the linear **criterion** $D_o(p) = 1_{W_o \cdot p + b_o p > 0}$ for node o, where W_o is a 1×3 matrix and b_o is a scalar. The criterion D_o acts as the criterion in a decision tree's node, and holds $\forall o' \in L_d \cup \text{sub}(o_l), D_o(p(o')) = 0$, and

Algorithm A Build a K-D tree

```
1: function BUILD(Point cloud P)
2:     if |P| = 1 then
3:         return make-leaf(P)
4:     end if
5:     o ← new-node()
6:     W_o ← choose-division-plane()
7:     b_o ← medium of {W_o · p | p ∈ P}
8:     o_l ← Build({p | W_o · p < b_0, p ∈ P})
9:     o_r ← Build({p | W_o · p ≥ b_0, p ∈ P})
10:    return K-D tree rooted at o
11: end function
```

$\forall o' \in L_d \cup \mathrm{sub}(o_r), D_o(p(o')) = 1$. We call the plane $\{W_o p + b_o = 0 \mid p\}$ the **division plane** between the left and right children. In the original definition of K-D trees, we limit the division plane to be parallel to an axis plane, or $W_o \in \{(0,0,1),(0,1,0),(1,0,0)\}$.

A node o is corresponding to a continuous 3D **sub-space** $S(o)$ being the intersection of the criterions of all its ancestor nodes. In the original definition of K-D trees, all $S(o)$'s are 3D rectangulars. $S(o)$ characterizes the sub-tree of node o, so that $S(o) \cap P = \{p(o') \mid o' \in \mathrm{sub}(o) \cup L_d\}$.

As a K-D tree of 2^d points is always a full binary tree, it can be determined by the mapping or arrangement $p(o)$ from leaf nodes to input points. The arrangement algorithm **arrange-points**$(P') \to o$ is a recursive algorithm, which takes a subset P' of input points with $|P'| = 2^{d'}$, and returns a depth-d' sub-tree with root node o, so that $\mathrm{sub}(o) = P'$. Intuitively, the arrangement algorithm recursively constructs each node of the K-D tree with given input points.

Our algorithm (Algorithm A) acts as below. For a leaf node, the algorithm immediately returns with a single leaf node o with $p(o) = p_i$ s.t. $P' = \{p_i\}$. Otherwise, the algorithm chooses the normal vector $W_o =$ **choose-division-plane**(P') of the division plane. Then, it finds proper bias b_o to divide P' into two equal-size parts P'_l and P'_r, calls **arrange-points** recursively on each of them, and uses the returned nodes as o's children nodes o_l and o_r.

The method **choose-division-plane**(P') is the key of the whole arrangement algorithm. An implementation of the original K-D tree defines such a method to choose the best axis as the normal vector, according to some metric. In PointTree, we use the principle component of P' (obtained by a PCA algorithm) as the normal vector.

B Transformation Sampling and Dataset Generation

This section covers more details about the transformed dataset construction mentioned in **Section 4.1 Transformations** in experiments of the main paper.

We use Algorithm B to generate our transformed dataset from an existing dataset D with transformation distribution T. In this algorithm, each single data

Algorithm B Construct transformed dataset

```
function TransData(Dataset D, Transformation distribution T, Augment time a)
    D_t ← ∅
    for all P ∈ D do
        repeat
            Sample transformation t ~ T
            D_t ← D_t ∩ {t(P)}
        until repeated a times
    end for
    return D_t
end function
```

i.e., point cloud in D will be applied with a (augment time) *different* transformations in T and included in the transformed dataset.

We define 3 transformation distributions: T_{affine}, $T_{\text{affine\_agg}}$, and $T_{\text{projective}}$. T_{affine} generates a random affine transformation with `torch.nn.Linear(3, 3, bias=False)` in PyTorch, i.e., generating a random 3×3 affine matrix A where $A_{i,j} \sim U\left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$. $T_{\text{affine\_agg}}$ is the distribution of the affine transformation with maximum EAD (a metric for measuring transformation intensity, defined in **Section 3.2 Robustness Against Transformations** in the main paper) among 5,000 different samples from T_{affine}, which represents more aggressive affine transformations with larger EAD.

$T_{\text{projective}}$ is defined with a special generating algorithm to guarantee that there is no numerical issue. The point cloud is unitilized to $[-1, 1]^3$, and then applied with a random affine transformation sampled from T_{affine}. Then, we randomly select 4 points from $[-2, 2]^3$, and use three of them as vanishing points V_x, V_y, V_z for each axis and the remaining one as the point O_p which the original point O is projected to. Then, we randomly generate scalor arguments a, b, c and decide d to construct the projective matrix as below

$$\begin{pmatrix} aV_x & a \\ bV_y & b \\ cV_z & c \\ dO_p & d \end{pmatrix}. \tag{1}$$

For the argument d, we randomly select it in the range that makes sure that no points will be projected to a point with infinity coordinates.

C Robustness Against Similarity Transformations

In **Section 3.2 Robustness Against Transformations** in the main paper, we analyzed the robustness against similarity transformations. We explain more details and prove the lemma in this section.

A **similarity transformation** is a transformation that preserves the shape (degree of all angles) of a point cloud. It includes rotations, flips, scales, and shifts.

PointTree uses relaxed K-D trees as the base tree, which holds lemma below:

Lemma. If **choose-division-plane**(P) is equivariant to a similarity transformation σ, or

$$\textbf{choose-division-plane}(\sigma(P)) = \sigma(\textbf{choose-division-plane}(P)), \tag{2}$$

then the whole arrangement algorithm **arrange-points**(P) is equivariant to such similarity transformation. Also, each leaf's corresponding point $p(o)$ is invariant to similarity transformation σ.

Proof.

The normal vectors of division planes returned by **choose-division-plane**(P) is agnostic to shift and global scales, since both transformations doesn't change the direction of such planes indicated by normal vectors. As a result, function **choose-division-plane**(P) is natively invariant to shift and scaling. Equivalently, we can assume the input point cloud P is already re-centered (shift to make the center of mass locates at O) and unitilized (scale to make all points locate in a unit sphere and one point is on the sphere). In this way, we don't need to consider the shift and scaling components in σ, and σ can therefore be regarded as a orthogonal 3×3 affine matrix.

If the division plane W_o returned by **choose-division-plane**(P) is equivariant to the transformation σ, the division plane in the same function call in the K-D tree construction of $\sigma(P)$ will choose the division plane $\sigma(W_o)$. For each point $p \in P$, the criterion for division into $P_l^o = \{p \mid W_o \cdot p < b_0, p \in P\}$ or $P_r^o = \{p \mid W_o \cdot p \geq b_0, p \in P\}$ is the value of $W_o \cdot p$; and for $\sigma(p) \in \sigma(P)$, the criterion is (here we regard p and W_o as column vectors)

$$\sigma(W_o) \cdot \sigma(p) = W_o^\top \sigma^\top \sigma p = W_o^\top p = W_o \cdot p, \tag{3}$$

which is the same as that of the construction of the original point cloud P. So, the divided point sets P_l^o and P_r^o will be equivariant to transformation σ, i.e., if p is divided into P_l^o in construction of P, then it will also be divided into P_l^o in construction of $\sigma(P)$, and vice versa for P_r^o. They will be passed to the recursive call of **Build(·)** at line 8 and 9 in Algorithm A. By induction, each function call of **Build(·)** will be exactly the same for P and $\sigma(P)$, so that the division plane chosen by each function call will be equivariant to σ, and the structure of the K-D tree $T = (\{L_i\}, \{(o, \{o_l, o_r\})\}, p(\cdot))$ will be the invariant.

PointTree uses PCA to implement **choose-division-plane**(P), which is equivariant under any similarity transformation. As a result, the structure of the relaxed K-D tree $T = (\{L_i\}, \{(o, \{o_l, o_r\})\}, p(\cdot))$ remains identical after applying the similarity transformation σ, and the bottom-up flow also works in an identical way. This makes our model's working flow invariant to similarity transformations.

Table A: Our pre-alignment ('PA') method achieves a mean EAD that is very close to zero in affine transformed point clouds, and highly reduces the EAD of projective transformed point clouds by 50%. For each number in the table, we randomly sample $3,000$ transformations and take the mean of the EAD between pre-aligned original and pre-aligned transformed point clouds. The point clouds with a label (e.g., 'Laptop') are taken from ModelNet40 train data, and we also provide their index in the training dataset like #0. For random point clouds, the point sets are uniformly sampled from $[-1, 1]^{2048\times 3}$

Point Cloud	Affine	Affine Aggressive	Projective	Projective w/o PA
Laptop (#0)	8×10^{-7}	5×10^{-6}	0.3073	0.9641
Wardrobe (#1)	9×10^{-7}	2×10^{-6}	0.3377	0.9191
Table (#5)	1×10^{-6}	3×10^{-6}	0.2979	0.8996
Airplane (#11)	7×10^{-7}	8×10^{-6}	0.3254	0.9806
Random	4×10^{-7}	2×10^{-6}	0.6745	1.2173

D Robustness Against Affine Transformation

Our pre-alignment method and the function **choose-division-plane**() in K-D tree construction use PCA, which is invariant to similarity transformations, as discussed in the previous section and in **Section 3.2 Robustness Against Transformations** in the main paper. In this section, we discuss more about the robustness against affine transformations.

There is some work [3,6] that uses PCA to design affine-invariant descriptors or functions for 2D and 3D point sets. It has been theoretically proved that some features obtained by PCA (e.g., based on eigenvalues and eigenvectors) are affine-invariant. Such an observation shows the potential of PCA in extracting affine-invariant/robust information – we hypothesize that this is the underlying reason that our pre-alignment and PointTree with PCA-based relaxed K-D tree construction are robust against affine transformations. We leave a rigid proof as interesting future work.

We also provide empirical analysis for pre-alignment. We calculate EAD (a metric for measuring transformation intensity, defined in **Section 3.2 Robustness Against Transformations** in the main paper) on pre-aligned original point clouds and pre-aligned affine-transformed point clouds, and the results are shown in Table A. We can find that the mean EAD is smaller than 10^{-4} in all affine cases, which means that the shape after pre-alignment is invariant to affine transformations with high precision. The experiments also show that our pre-alignment can highly reduce the EAD for projective-transformed point clouds. These results empirically shows that our pre-alignment is highly invariant to affine transformations and can also work on projective transformations, and thus it is a simple yet powerful and general approach for pre-processing of point clouds.

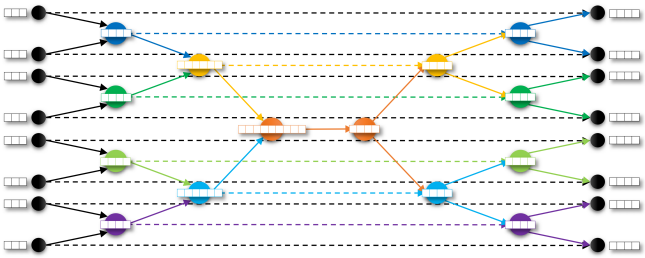

Fig. A: The segmentation model. The two symmetric K-D trees represent two stages: a bottom-up information flow (**left**) and a top-down information flow (**right**), on the same K-D tree

E Details About Segmentation Component

This section covers more details about the general segmentation component in **Section 3.3 Downstream Components** in the main paper.

First, we introduce the top-down information flow, which is opposite to the bottom-up flow we used in the encoder. Each node has an information self(o) of itself, and we compute the carried information carry(o) that is downloaded from its ancestors.

The input of such an information flow is $\{\text{self}(o)\}$ and $\text{carry}(R) = \text{self}(R)$ for the root node R. Then, for each node o, its carry(o) is obtained by aggregating self(o) and its parent's carried information carry(par(o)), as the formula below:

$$\text{carry}(o) = \textbf{merge-carry}(\text{self}(o), \text{carry}(\text{par}(o))). \tag{4}$$

The carried information contains all information of its ancestor node. For example, if we want to compute the number of ancestors for each node, we can set $\text{self}(o) = 1$ for all nodes, and

$$\textbf{merge-carry}(a, b) = a + b. \tag{5}$$

After running the top-down information flow, carry(o) will be the number of ancestors (including itself) for each node o.

For the segmentation task, we maintain both a bottom-up flow and a top-down flow on the same K-D tree (as shown in Figure A) and define $\text{self}(o) = \text{info}(o)$ with a skip connection from the symmetric node in the bottom-up information flow. As $\text{carry}(R) = \text{self}(R) = \text{info}(R)$ is the vector of global features of the point cloud, intuitively, in each **merge-carry** step, we are obtaining a connection between the local information and the global information. For each leaf node o, carry(o) can be regarded as a connection between the point $p(o)$ and the whole point cloud . So we can regard such carry(o) as the point feature of point $p(o)$. We use an MLP to classify each leaf node's point feature, and output the log-likelihood score for each segment class candidate.

Table B: On projectived transformed, pre-aligned ShapeNetPart [2], PointTree significantly outperforms the baselines in class-level mIoU(%), and achieves the state-of-the-art mean IoU in 75% of all classes

Method	airplane	bag	cap	car	chair	earphone	guitar	knife	lamp	laptop	motorbike	mug	pistol	rocket	skateboard	table	mIoU
DGCNN [7]	**69.8**	46.5	**92.4**	29.3	36.8	**92.0**	83.7	**75.6**	65.9	**80.6**	14.5	45.7	10.4	39.1	29.8	66.8	55.0
CurveNet [9]	22.4	44.8	36.6	5.7	30.7	20.5	22.7	25.2	45.9	27.2	16.9	48.6	23.7	22.7	43.7	45.5	30.2
PointTree Def	67.4	**63.0**	76.4	**49.0**	**75.2**	57.1	**84.2**	70.7	**67.7**	59.9	**39.9**	**80.8**	**60.3**	**52.5**	**57.8**	**72.2**	**64.6**

F Part Segmentation on Projective Transformed Dataset

The segmentation results on projective transformed ShapeNetPart [2] is in Table B. Our PointTree achieves a top mIoU over all the baselines with an increment of nearly 10%, and achieves the state-of-the-art mean IoU in 75% of all classes. It shows that PointTree outperforms other baselines with a larger gap under more challenging projective transformation.

G Architectures of PointTree Variants

According to **Section 4.2 Baselines and PointTree Variants** in the main paper, we have 3 variants of PointTree: Def, KA, and RNS. Their architectures are shown in Figure B.

For PointTree KA, we design this variant by introducing a stronger alignment network based on PointTree Def encoder. As we mentioned in "Alignment Network" in Section **??**, the alignment network supports any encoder that outputs a point cloud feature. Different from the default encoder that uses T-Net in PointNet [5] as the alignment network, we build the alignment network as another PointTree Def encoder in this version.

For PointTree RNS, we can stack multiple ResNet blocks to get a larger model. We only use the single ResNet block version in all our experiments.

H Additional Implementation Details

In this section, we include additional implementation details when conducting experiments in **Section 4 Experiments** in the main paper.

Hyperparameters. The architectures of the model variants we use are shown in Figure B. We use 2,048 points in each point cloud, and build a 12-level tree. In the bottom-up flow, the dimension of node features from bottom to top is 32, 64, 128, 256, 512, 512, 1024, 1024, 2048, 2048, 4096, and 4096. In the top-down flow, the carried information's dimension is 512. The batch size is set to 32 for all segmentation tasks and for PointTree RNS on classification tasks, and 64 for all the other experiments.

Implementation. We implement the whole PointTree with PyTorch. The PCA in pre-alignment and K-D tree construction are implemented with a library function `torch.pca_lowrank`. For the dimension-increasing MLP at each

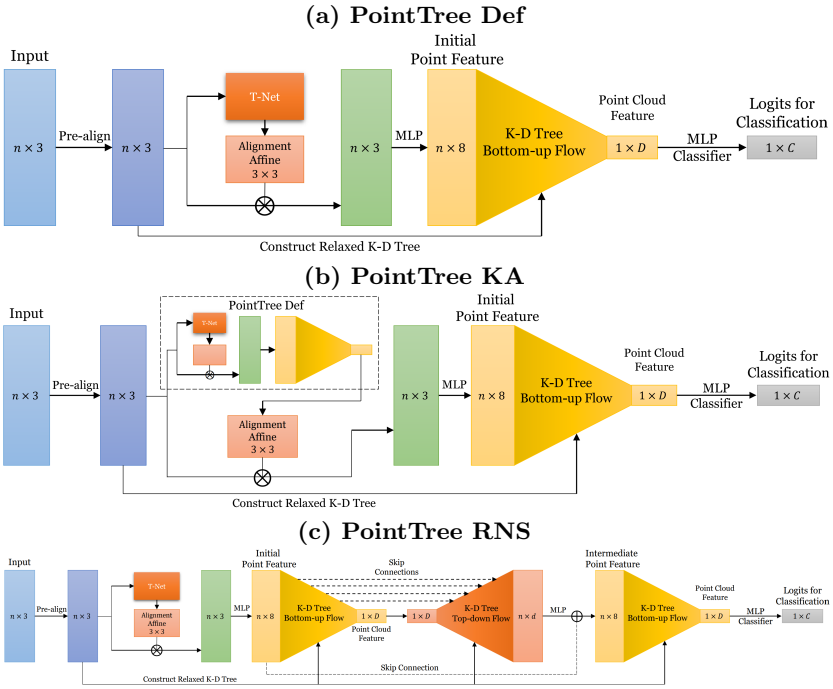

Fig. B: The architectures of 3 PointTree Variants: (a) Def, (b) KA, and (c) RNS. (a) PointTree Def is the simplest model that uses the T-Net in PointNet [5] as the alignment network. (b) PointTree KA replaces the T-Net with another PointTree Def, and uses its output, the point cloud feature, to generate the 3×3 alignment affine matrix. (c) PointTree RNS connects a PointTree Def with a segmentation component, and uses the segmentation component's output plus the skip connection from the initial point feature as the input features for another K-D tree bottom-up flow

layer, we use one `torch.nn.Linear`. For the MLP classifier for both classification and segmentation, we use a 3-layer MLP with ReLU activation and batch normalization for hidden layers.

Data Augmentation. When training PointTree, we do data augmentation by applying random axis flipping and axis permutations. And, for pre-aligned experiments, we also augment the data by applying random affine transformations on the point cloud then pre-align again. According to Figure B, these augmentations are inserted after pre-alignment and only affect the coordinate inputs of PointTree. They do *not* affect the construction of K-D trees. The K-D tree for each point cloud is constructed only on the transformed coordinates (with or without pre-alignment in different settings).

Training. We run all experiments of PointTree with a PC on NVIDIA RTX 3060 GPU. We train the model with mini-batch training and use Adam as the optimizer. It took 20 hours for full convergence, but according to "Overtime

Algorithm C Iterative Pre-alignment

function IterativePreAlign(Point cloud P, Maximum Iteration m)
  while # Iterations $< m$ **do**
    $U, \Sigma, V \leftarrow \mathbf{PCA}(P)$
    if Vectors in $\{V\}$ are x,y,z-axes **then**
      (*The axes suggested by PCA converges.*)
      break
    end if
    (*Align the point cloud with PCA.*)
    $P \leftarrow U$
    (*Apply a designed scaling scheme.*)
    for $\mathbf{d} \in \mathbf{x}, \mathbf{y}, \mathbf{z}$ **do**
      (*For each axis, unitilize the coordinates to make the average abs. 1.*)
      $f \leftarrow \mathbf{Mean}_{p \in P} |\mathbf{d}_p|$
      $\mathbf{d}_p \leftarrow \mathbf{d}_p / f, \forall p \in P$
    end for
  end while
  return P
end function

Accuracy" in **Section 4.3 Classification on ModelNet40** in the main paper, it can achieve a comparable result within 2.5 hours. For ModelNet40 [8], it only includes train and test splits. We further split the original training dataset into the new training dataset and the validation dataset. We train PointTree only on the new train dataset and tune hyperparameters only on the validation dataset. For ShapeNetPart [2], it already have the splits of training, validation, and test, so we train PointTree using these splits in the common way.

I Iterative Pre-alignment

There is still some issues in our PCA-based pre-alignment. The solution of PCA is not unique (e.g., for a sphere-shaped point cloud, any vector is the principal component). Though our experiments shows that the EAD between differently affine transformed point cloud is very small, i.e., the they have same shapes, there may still be a similarity transformation between them. Also, the scaling factor Σ provided by PCA may be not optimal for our model to obtain best results.

To obtain better results, we further propose an iterative pre-alignment scheme, to iterative align the point cloud with PCA and apply a designed scaling factor until convergence. The algorithm is shown in Algorithm C.

The iterative pre-alignment can stabilize and improve the performance of PointTree.

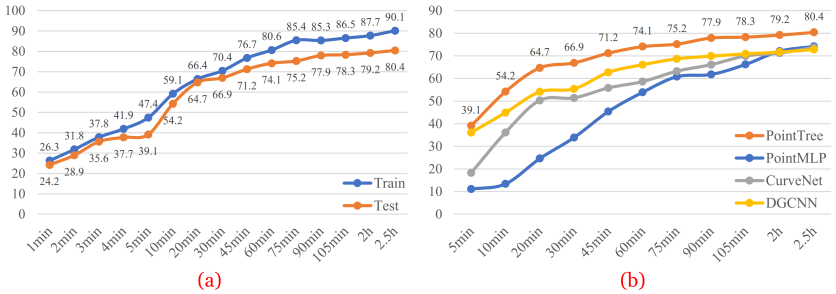

Fig. C: PointTree converges very fast during training time and leads the test accuracy among all baselines. It highly adapts the transformed data so that it can achieve a reasonable accuracy within 5 minutes, and reaches 95 percent of its best accuracy (%) in a very limited time of 2.5 hours. (a) The overtime train and test accuracy (%) of our most powerful model, PointTree RNS. (b) The overtime test accuracy of PointTree and top baselines

J Overtime Accuracy of Classification Task

Figure C-a shows the train and test accuracy of our most powerful model, PointTree RNS, on pre-aligned ModelNet40 with affine transformation. Even with very limited time of 2.5 hours, our model can still achieve 95 percent of its best accuracy (80.4% out of 84.1%), which already outperforms many baseline models. PointTree can even achieve a reasonable accuracy of 24.2% within 1 minute. These facts show that PointTree is highly adaptive on transformed data. Figure C-b shows overtime test accuracy of PointTree and top baselines. Among these models, PointTree is always leading the test accuracy, and has a gap that is more than 5% higher than all these baselines at most of the time.

K Stability Test of PointTree

Table C shows the stability test results. In this experiment, we sample multiple transformations for each point cloud in test data, and compute the standard variance to show the stability of our model on different transformations. Also, we add a special test: "affine (aggressive)", for which we sample affine transformation from another distribution that is expected to have higher EAD. The experiment results show that (1) PointTree is stable among different transformations with low standard variance and (2) PointTree's accuracy is higher than the baseline PointMLP [4] with statistical significance.

L Details About S3DIS

We evaluate PointTree for the point cloud semantic segmentation on S3DIS [1]. It contains 6 areas with 271 rooms. There is a point cloud for each room, containing

Table C: PointTree has stable performance with a low standard variance over different random transformations on each transformed ModelNet40. Its accuracy on all experiments is higher than the baseline PointMLP [4] with statistical significance. The second column is the mean EAD over all point clouds in the dataset (a more difficult task has a higher EAD). The accuracy is also instance-level accuracy (%), and the standard variance σ(%) are computed over ModelNet40's testing data with different random transformations

Transformation	Mean EAD	PointMLP [4]	PointTree RNS
Affine w/ PA	10^{-4}	82.3	$84.1^{\sigma=0.23}$
Affine w/o PA	0.387	63.1	$73.1^{\sigma=0.22}$
Affine (Aggressive) w/o PA	0.785	37.5	$63.7^{\sigma=0.38}$
Projective w/ PA	0.307	49.9	$62.1^{\sigma=0.67}$
Projective w/o PA	0.960	4.1	$31.8^{\sigma=0.76}$

all objects' surfaces in the room within 13 object categories, e.g., ceiling, floor, window, etc. In our evaluation, we use the PointNet [5] version of S3DIS, which "sample(s) rooms into blocks with area 1m by 1m", and "randomly sample(s) 4,096 points in each block". We use the processed data released on PointNet's Github repository for training and evaluation, to make it a fair comparison over all baselines.

References

1. Armeni, I., Sax, A., Zamir, A.R., Savarese, S.: Joint 2D-3D-semantic data for indoor scene understanding. arXiv **abs/1702.01105** (2017) 10
2. Chang, A.X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., Yu, F.: ShapeNet: An information-rich 3D model repository. arXiv **abs/1512.03012** (2015) 7, 9
3. El Oirrak, A.: Affine invariant descriptors using principal components analysis. Pattern Recognition and Image Analysis **18**(1) (2008) 5
4. Ma, X., Qin, C., You, H., Ran, H., Fu, Y.: Rethinking network design and local geometry in point cloud: A simple residual MLP framework. arXiv **abs/2202.07123** (2022) 10, 11
5. Qi, C.R., Su, H., Kaichun, M., Guibas, L.J.: PointNet: Deep learning on point sets for 3D classification and segmentation. In: CVPR (2017) 7, 8, 11
6. Tzimiropoulos, G., Mitianoudis, N., Stathaki, T.: An affine invariant function using pca bases with an application to within-class object recognition. In: ICASSP. vol. 1 (2007) 5
7. Wang, Y., Sun, Y., Liu, Z., Sarma, S.E., Bronstein, M.M., Solomon, J.M.: Dynamic graph CNN for learning on point clouds. ACM Transactions on Graphics **38**(5) (2019) 7
8. Wu, Z., Song, S., Khosla, A., Tang, X., Xiao, J.: 3D shapenets for 2.5D object recognition and next-best-view prediction. arXiv **abs/1406.5670** (2014) 9
9. Xiang, T., Zhang, C., Song, Y., Yu, J., Cai, W.: Walk in the cloud: Learning curves for point clouds shape analysis. In: ICCV (2021) 7