ARKit LabelMaker: A New Scale for Indoor 3D Scene Understanding

Supplementary Material

A. Dataset Class Statistics

Dataset Statistics of ARKit LabelMaker. In Figure A1, we present the point count for each class in the LabelMaker WordNet label space. Our dataset maintains a substantial data distribution even across tail classes.

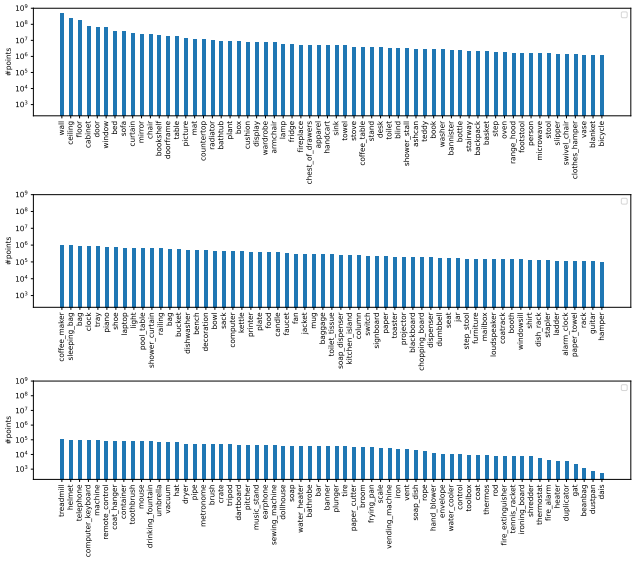

Figure A1. **Number of points for each ARKitLabelMaker class**.

B. PTv3 Results on ScanNet++

We also report the training and evaluation results of PTv3 on ScanNet++ in Table B1. Unfortunately, the numbers are not fully comparable, because we were so far unable to reporduce the validation results of PTv3. When the authors of [2] released PTv3's performance on ScanNet++, they expanded Structured3D's training set from 6,519 to 18,348 samples, which we refer to as Structured3D v2. Due to limited computational resources, we could not train with this updated version of Structured3D yet. We will update ScanNet++ results once the new result is available. Our pre-training and joint-training (PPT) experiments show performance gains over vanilla PTv3, with PTv3-PPT achieving similar improvements to the original PTv3-PPT but with significantly less training data.

C. Tail Classes Performance of PTv3

We give a detailed plot of the number of correctly predicted tail class points on ScanNet200 validation set in Figure C2.

PTv3 Variant	Training Data	#Data	val mIoU
vanilla	ScanNet++	713	41.8
fine-tune (Ours)	ARKit LabelMaker^{SN200} $\rightarrow$ ScanNet++	4471 $\rightarrow$ 713	42.5
PPT [2]	ScanNet200 + ScanNet++ + Structure3Dv2	45868	**45.3**†
PPT (Ours)	ScanNet200 + ScanNet++ + ARKit LabelMaker	11168	44.5
PPT (Ours)	ScanNet+ ScanNet200 + ScanNet++ + Structure3D + ARKit LabelMaker	30386	44.6

Table B1. **Semantic Segmentation Scores on ScanNet++ [3].** We evaluated the efficacy of our ARKit LabelMaker dataset on the ScanNet++ benchmark using both pre-training and joint training methods. †: this number comes from Wu et al..

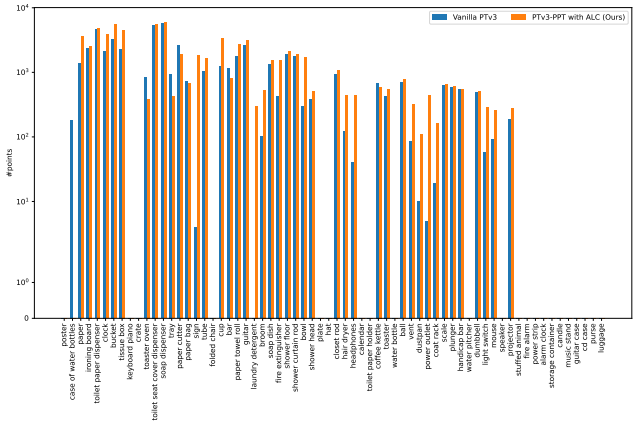

Figure C2. **Correctly predicted tail class points on ScanNet200 validation set.** We compare the number of correctly predicted points of tail class in ScanNet200 validation sets between PTv3 trained from scratch and the PTv3-PPT trained with our datasets. With our dataset, Point Transformer gains more ability to detect rase classes.

D. Detailed process of applying LabelMaker to ARKitScenes

ARKitScenes is one of the largest indoor 3D scenes dataset. It consists of 5047 parsable scenes of various size. We consider a scene parsable if the RGB-D trajecotry comes with associated pose data. Processing these scenes with our improved LabelMaker pipeline requires deliberate engineering with the following goals: a) Bring the data in to the format required by LabelMaker [1] b) Robust processing to not waste compute on failures, c) Improved parallelization to speed up processing. d) Accurate resource estimation to prevent waste of compute resources and longer job waiting time. e) Fast failure identification and results inspection.

Transforming the data LabelMaker [1] requires a specific data format to be able to reliably process all data. All trajectories require posed RGB-D data and a denoised 3D model that is used by Mask3D. ARKitScenes comprises data of varying resolutions and sampling rates. Depth data is captured at 256×192 and 60 FPS, while the RGB frames

are recorded at 640×480 and 30 FPS. Therefore, synchronization is required to process the data with LabelMaker. To this end, we match each RGB frame with the closest depth frame in time and we resize the depth frame to RGB frame's resolution. Pose data, originally at 10 FPS, is interpolated using rotation splines to synchronize with each RGB frame. To obtain a 3D mesh of each scene that can be processed by Mask3D, we reconstruct the 3D model by fusing the synchornized posed RGB-D data using TSDF fusion and then extract mesh with marching cube algorithm. We empirically chose a voxel size of 8mm and a truncation distance of 4cm for fusion.

Building a scalable pipeline LabelMaker [1] can be decomposed into individual modules such as the individual base models, the consensus computation, and the 3D lifting. This modular nature allows to build a scalable pipeline using popular high-performance computing toolboxes. As the different base models have different runtimes, we had to leverage dependency management system that can handle different dependencies of the pipeline steps. This architecture allows us to effectively leverage large-scale computing and distribute the processing across many different nodes.

In more detail, we decompose the pipeline into several jobs (ordered by dependency) for each scene:

1. Preprocessing: Downloading the original scene data, transforming it into LabelMaker format, and run TSDF fusion to get the 3D mesh of the scene.
2. Forwarding 2D images or 3D meshes to each base models: Grounded-SAM, Mask3D, OVSeg, CMX, InternImage. (all jobs depends on step 1.)
3. Getting the consensus label from base models' labels. (depends on all elementary jobs in step 2.)
4. Lifting the 2D consensus label into 3D. (depends on step 3.)
5. Rendering the outputs of base models or consensus into videos for visualization. (depends on steps 2. or 3.)
6. Post-processing, including removing temporary files and get statistics of each tasks. (depends on all steps above)

Optimizing compute resource scheduling. ARKitScenes contains scenes of a wide range of sizes, spanning from a minimum of 65 frames to a maximum of 13796, and different parts of the pipeline scale differently with increasing scene size. To figure out the minimum resources requirements, we select 11 scenes of varied sizes uniformly distributed within the minimum and maximum range and record their resources usage. While most jobs are not sensitive to scene size and can suffice with a fixed resource allocation, the base models exhibit greater sensitivity to scene size. We interpolate resource needs with respect to scene size and summarize the empirical numbers into Table D2. Through this, we ensure that we request minimal-required resources, so that we have lowest job waiting time and less idle compute power.

Task	#CPU	RAM	Time	GPU
Download & Prepossessing	2	24G	4h	-
Video Rendering	8	32G	30min	-
Grounded-SAM	2	12G	6h	3090 $\times$1
OVSeg	2	8G	8h	3090 $\times$1
InternImage	2	10G	8h	3090 $\times$1
Mask3D	8	16G	1h 30min	3090 $\times$1
OmniData	8	8G	2h	3090 $\times$1
HHA	18	9G	2h	-
CMX	2	8G	3h	3090 $\times$1
Consensus	16	16G	2h	-
Point Lifting	2	72G	4h	-

Table D2. **Requested resources for each task.** We report the average resources required by the individual steps of the LabelMakerv2 pipeline. The required cores, RAM, and GPU time varies across the different jobs. Through our job scheduling mechanism, we ensure that the required compute is optimially distributed across all jobs.

Assuring the quality of the individual processings. In order to assure high-quality labels produced by our improved pipeline, we have built tooling to efficiently check for failures of the processed scenes. To this end, we store the logs and statistics of each job and built visualization tools for this data as well as the intermediate predictions. This allows us to conveniently browse at scale through the predictions and identify individual failures.

Failure handling and compute resource optimization. When doing large-scale processing on a high-performance compute cluster, a common issue is the failure of jobs. This can happen for several reasons such as node crashing, unexpected memory usage, and many more. Therefore, the processing pipeline has to be robust to these failures. Additionally, compute should not be wasted when recovering from these failures. Therefore, we designed a simple yet effective strategy for efficiently recovering from job failures. Before every restart is triggered for a scene, we analyze both the logs and file system to identify the jobs that have not finished for this scene. Once these jobs have been identified, we only resubmit these jobs. This ensures that no compute resource is used in rerunning completed tasks.

E. Log-Linear Performance Relation in Data Scaling

E.1. Implementation Details

We optimize the code to deploy each individual piece of the pipeline of LabelMakerV2 as individual jobs to a GPU cluster, with SLURM as a dependency manager between the pipeline pieces. To optimize the overall execution time, it is therefore important to be able to estimate the processing time of each piece of the pipeline at the point of job submission. ARKitScenes contains scenes of a wide range of sizes, spanning from a minimum of 65 frames to a maximum of

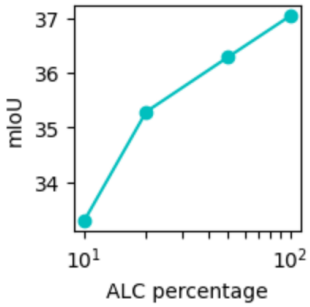

Figure E3. **Relation of Validation mIoU against training data percentage of ARKit LabelMaker**. This figure shows the validation mIoU on ScanNet200 after fine-tuning with respect to the percentage of ARKit LabelMaker data used in pre-training. This figure shows a log-linear relationship.

13796, and different parts of the pipeline scale differently with increasing scene size. To figure out the minimum resources requirements, we select 11 scenes of varied sizes uniformly distributed within the minimum and maximum range and record their resources usage. While most jobs are not sensitive to scene size and can suffice with a fixed resource allocation, the base models exhibit greater sensitivity to scene size. We interpolate resource needs with respect to scene size and summarize the empirical numbers in the Appendix. Through this, we ensure that we request minimal-required resources, so that we have lowest job waiting time and less idle compute power.

We use a CentOS 7 based SLURM cluster to process all the data, which is capable of handling task dependencies and parallel processing. Before submitting jobs for a single scene, we first check the progress of each job and generate a SLURM script to submit only those unfinished jobs. This ensures that no compute resource is used in rerunning completed tasks.

We employ test time augmentation by forwarding all models twice, with Mask3D using two different random seeds and other models being mirror flipped. Following the practice of LabelMaker [1], we assign equal weights to each model when calculating the consensus, although these weights are configurable in our pipeline code. Since we are primarily interested in the 3D labels that can be used for pre-training 3D semantic segmentation models, SDFStudio training and rendering are omitted due to their lengthy processing times. Further, we enhance the pipeline by storing both the most and second most voted predictions alongside their respective vote counts. This information is useful when we want to investigate on the uncertainty across the base models. We leave the exploitation of this information as potential future direction of research.

References

[1] Silvan Weder, Hermann Blum, Francis Engelmann, and Marc Pollefeys. LabelMaker: Automatic Semantic Label Generation from RGB-D Trajectories. In *International Conference on 3d Vision (3dV)*, 2024. 1, 2, 3

[2] Xiaoyang Wu, Li Jiang, Peng-Shuai Wang, Zhijian Liu, Xihui Liu, Yu Qiao, Wanli Ouyang, Tong He, and Hengshuang Zhao. Point Transformer V3: Simpler, Faster, Stronger. In *International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 1

[3] Chandan Yeshwanth, Yueh-Cheng Liu, Matthias Nießner, and Angela Dai. ScanNet++: A High-Fidelity Dataset of 3D Indoor Scenes. In *International Conference on Computer Vision (ICCV)*, 2023. 1