Project Progress Report

10-02-2021

Semantic Segmentation of Robotic Arm using RGB-D Data

Onur Berk Töre otore19@ku.edu.tr
Farzin Negahbani fnegahbani19@ku.edu.tr
Buğra Can Sefercik bsefercik@ku.edu.tr

Project Overview

Most robots nowadays are equipped with high-quality depth cameras to understand their surroundings. At first glance, it seems like these sensors combined with current advanced deep learning approaches can solve vision problems. However, many challenges exist on the way to a robust environment perception for robotic arms, especially during interaction with unknown objects. One of crucial and complicated challenge is semantic segmentation or classifying pixels in a digital image into semantically meaningful ones. This problem becomes harder specifically during the interaction when robot arm touches the object of interest. To address this problem, employing a semantic segmentation method that segments the robotic arm in RGB-D frame and extracting the arm's points from the environment with low latency is suggested. This method can also help with the automatic extrinsic calibration of the robotic arm and RGB-D sensors that is a promising future direction of this project. To this end, we aim to gather a custom dataset and adapt a recent semantic segmentation method for our purposes.

1 Methods

In this section, we explain all methods and materials developed and employed in this project. First, we discuss our proposed dataset and how we gathered and labeled data. Then we explain our backbone code which is borrowed from PointGroup[1] and is written in PyTorch [2] following by our contributions and modifications to this baseline.

1.1 Dataset

The baseline code is mainly tested with ScanNet [3] dataset. ScanNet is one of the largest datasets available with more than 20 classes for 3D segmentation task. ScanNet data samples are categorized into different scenes, each scene containing multiple RGB-D samples and corresponding segmentation labels, high-quality reconstructed mesh, decimated reconstructed mesh, and aggregated(all images from a scene) labels.

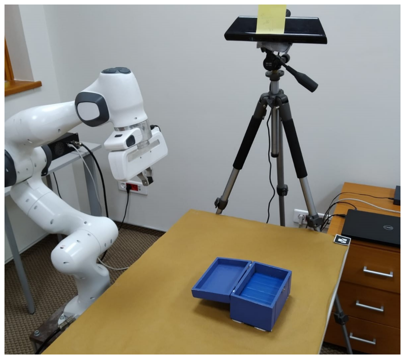


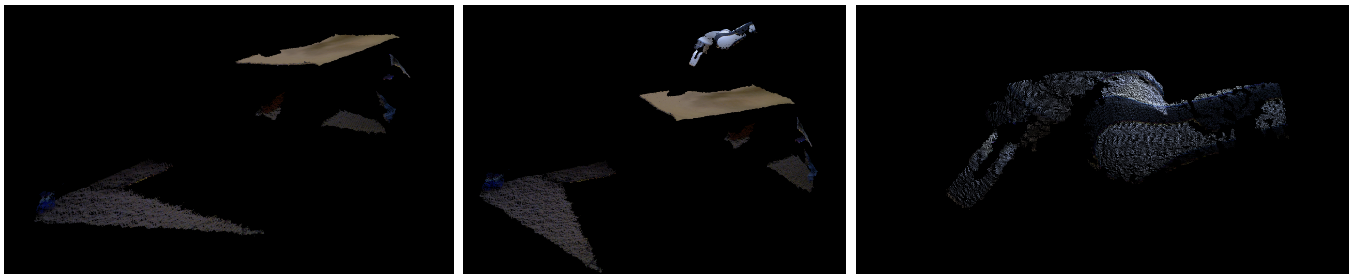
Figure 1: Experiment environment showing the robot, Kinect, table, and one object.

To make our dataset community-friendly, and move fast, we utilize a similar data format. Each scene differs in orientation and position of Kinect and illumination of the environment. Each sample of the scene differs in arm position. For each scene, we first extrinsically calibrate the Kinect with the robot arm to have all the transformations in case we need it. The procedure of collection for a scene starts from getting samples from the environment without the presence of the robot arm. Then with the presence of the robot in the Kinect's field of view, the arm starts executing a task and we collect consecutive frames. Table 1 shows more details about this dataset. Following collection of the data, a data annotator categorizes each point to background or arm point using the priors from the background sample and empirical settings from the user. Furthermore, it stores the labels in the desired format. Figure 2 shows a sample of scene, background, and labeled data.

Scenes	Illuminations	Train	Test	Avg. points
5	3	1028	277	200 K

Table 1: **Dataset Statistics.** Each sample contains point cloud data, end-effector pose, joint states and semantic labels. All other data such as reconstructed surface, decimated or filtered mesh can be automatically generated from sample.

In addition to our point cloud and RGB data, we built high quality and decimated reconstructed surface



(a) Background sample (b) Data sample (c) Extracted arm point cloud

Figure 2: Data collection and labeling process of the customized dataset.

meshes that may provide more options and data for those who wish to use this dataset in the future. Also, our code base can apply average filter on point cloud data with a desired number of iteration (the more iterations, the smoother the resulted surface). Samples of filtered data is depicted in Fig. 3.

1.2 Baseline Model for Robot Segmentation

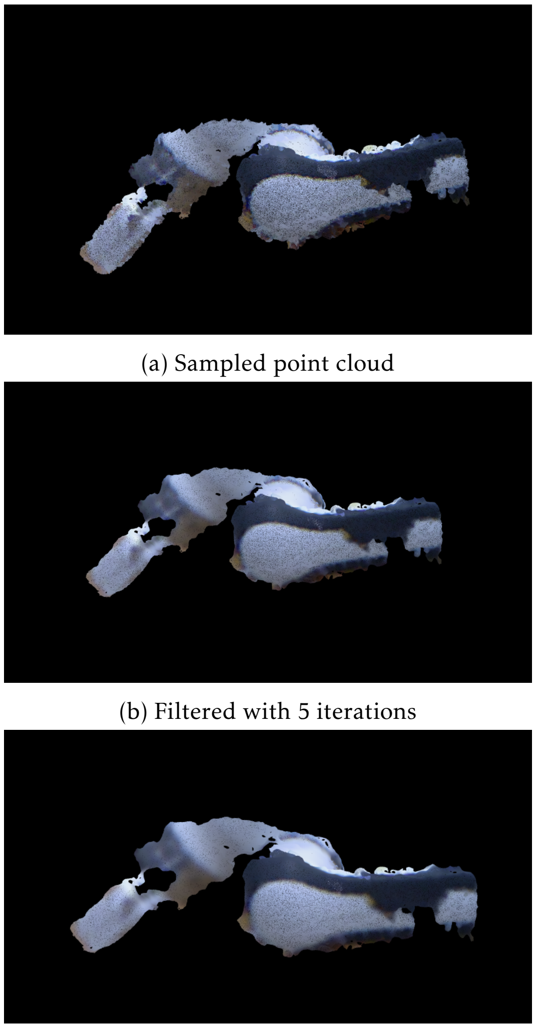
In this project, we decided to start with PointGroup from Li et al.[1] (CVPR2020) as our baseline model. PointGroup come second on the mAP_{50} in ScanNetV2 and 3DSIS datasets.

As the first step, Pointgroup [1] code adapted to work with our own dataset. The first step of adaptation consists of setting the parameters, implementing data loaders, and tweaking the training and testing code of the baseline. Moreover, the PointGroup [1] method trained and tested on the toy dataset providing a rich feedback on the baseline.

Fig. 6 shows the train and validation loss of the baseline model while training on our own dataset. The toy dataset consists of one scene with 76 training and 22 validation samples.

Class	mAP$_{50}$	mAP$_{25}$
Arm	0.909	0.977
Background	1.00	1.00
Average	0.955	0.989

Table 2: **Cropped Pointgroup Results on the validation set.** This table shows the cropped Pointgroup performance on the validation set of our toy dataset. The results reported in terms of average precision and IoU with 25% and 50% intersection over union metrics.



(a) Sampled point cloud

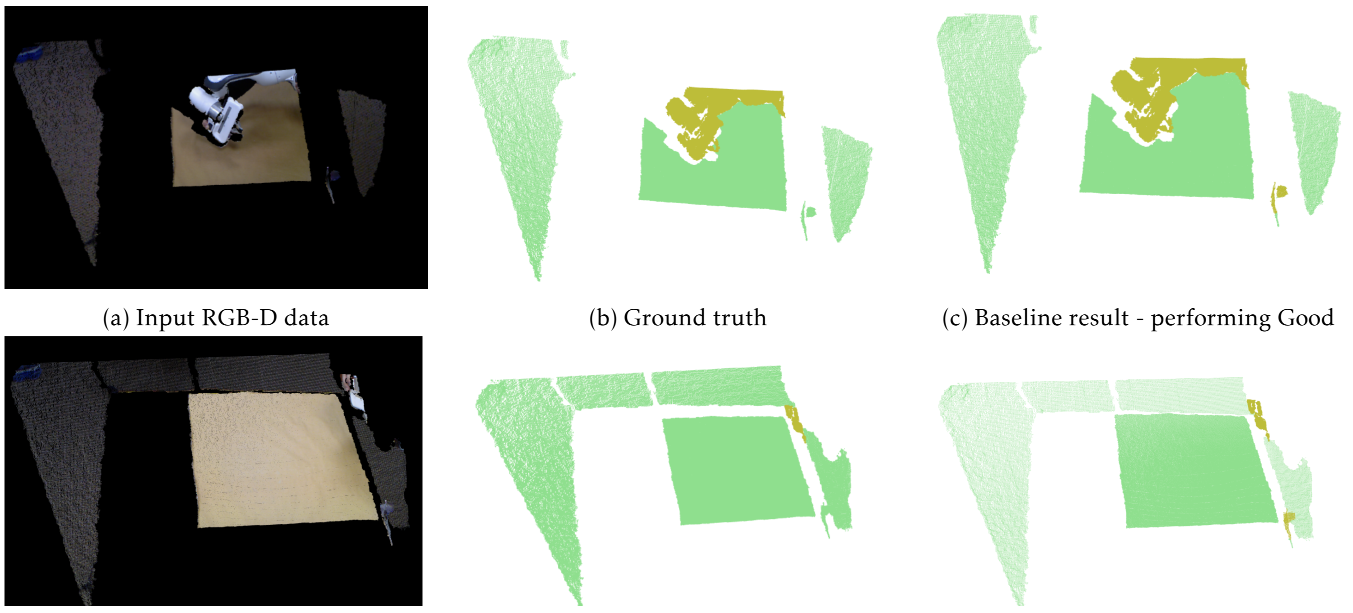
(b) Filtered with 5 iterations

(c) Filtered with 10 iterations

Figure 3: **Filtered surfaces.** Reconstructed surfaces are represented as mesh files in our dataset. To generate point cloud data, after sampling the mesh (sampled from mesh point cloud (a)), an average filtering can be applied to further smooth the surface. Two samples of filtered point cloud with 5 and 10 iterations illustrated in (b) and (c).

1.3 Increasing Model Throughput

The Pointgroup model has a second part after the semantic segmentation which estimates the clusters of multiple

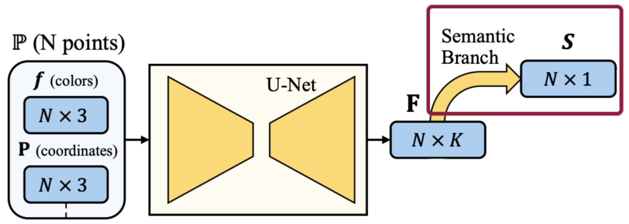


(a) Input RGB-D data (b) Ground truth (c) Baseline result - performing Good

(d) Input RGB-D data (e) Ground truth (f) Baseline result - performing poorly

Figure 4: **Qualitative samples.** Predicted from the validation set by the baseline model trained with our toy dataset. RGB-D input data (a) visualized besides the ground-truth (b) and a well performing prediction (c). In the second row, a sample visualized where the baseline performed low (f), plus, the GT (e) and the sample data (d). Colors in the figures (b), (c), (e), and (f) represent the semantic class of the points. The RGB color of the point omitted for the visualization purposes.

objects to further increase the accuracy of the semantic and instance segmentation. After seeing the qualitative results of the first segmentation part, we decided not to use these part of the model. The resulting cropped model can be seen in Fig.5. The qualitive results on validation set can be seen in the Fig. 4 and the accuracy on the validation set in terms of mean average precision and IoU of our cropped model can be seen in Table.2



(a) Backbone Network

Figure 5: **Customized PointGroup Overview.** Red boxed area shows the semantic branch that we are interested in.

The project aims is to increase inference speed of the model. The inference speed best measured with Frames Per Second (FPS). While 10 FPS may be enough for our real-time measurements, with the overhead of underlying robotics system, we aim to achieve around 25-30 FPS

Method	**Inference Time**	**FPS**	**# Params**
PointGroup	2.59 (Seconds)	0.38	7714710
Semantic PointGroup	0.06 (Seconds)	16.6	7532162

Table 3: **Inference Speed.** PointGroup is the baseline model without any modification. The semantic PointGroup is the version that uses only the semantic prediction part of the PointGroup. Tests are done using Nvidia GTX1080Ti.

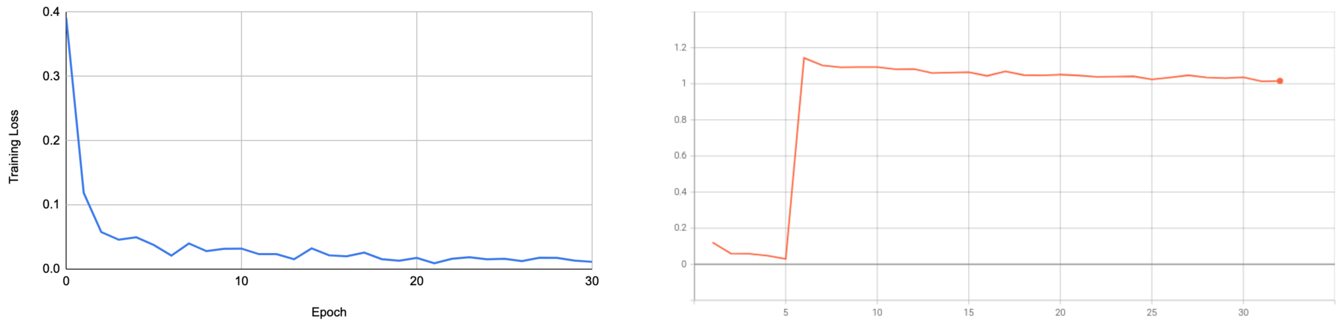
in practice.

The test (Table 3) reveals that while PointGroup model achieves around 0.38 FPS speed on our environment, the cropped Pointgroup achieves around 17 FPS.

1.4 Overall Method

As can be observed in Figure 8, our pipeline takes an 3D image as input and outputs a 3D position and orientation of the robot's end-effector. The two major learnable modules between these input and outputs are, customized version of PointGroup architecture for robot segmentation and a multi-layer perceptron to predict the 3D position of robot's end-effector.

In the previous sections, we investigated the customization steps and accuracy of segmentation module.



(a) Semantic Train Loss

(b) MLP Train Loss

Figure 6: **Traning Losses for Segmentation and Multi-Layer Perceptron.**

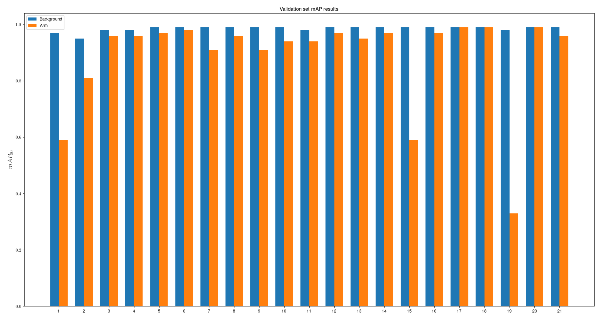


Figure 7: **Prediction Accuracy.** Shows the prediction accuracy on each sample of the validation set. Bars in orange represent the background and blue bars shows arm prediction accuracy.

We use this module at the first stage of our pipeline. In the second stage, first we voxelize the output of UNet in order to generate constant sized inputs for the perceptron. Next, as the end of the perceptron we predict xyz and quaternion coordinates of the robotic arm's end-effector.

In training time, we first train the segmentation module alone. Following the convergence of the first module, we start to train the multi-layer perceptron using minimum squared error as our objective function. We experimented with training the second alone and with the second stage utilizing weighted loss approach. Our experiments showed us that it is better to train modules one at a time because updating the custom unet module alongside the perceptron module decreases the first module's accuracy. We believe that is caused by disparate natures of the problems that we are attempting to solve.

Moreover, we experimented with different number of hidden units with various sizes. At end of exhaustive number of trials, we deduced that two hidden layers both with 4096 units suffice for our needs.

This version of the model is able to overfit small amount of data very well. However, we are not able to share our results with the real data because we have had errors while collecting the data. To tackle this problem, we need to work on our robot to which we have extremely limited access to laboratories due COVID19 precautions in Koc University.

2 Code

This section will summarize our baseline code and the changes that are done. Interested readers can look to section A for a more detailed explanation of each code file. The codebase is separated into three parts, first the data gathering part, second the PointGroup part, and the final RobotNet part.

The data collection part starts by recording the movement of the robot and the point cloud data. We wrote and tested necessary scripts to ensure the collected data do not have problems. After these tests the point cloud data separated into two parts as background and foreground. Sequential background point clouds are combined to generate one unified background information. This background is extracted from each foreground data to use as semi-supervised data. Next, the joint state information from the moving robot converted to the end effector to the base with basic forward kinematics. These two-part of the data then combined, and converted to a point group ready format using our Annotater class.

We changed the PointGroup code for newly generated data and extended our model for regression with RobotNet code. We made necessary changes in the data loader to load our data (PointGroup/data/alive1_inst.py), and config file. The config (PointGroup/config/pointgroup_alive) file changed in order to modify model properties such as epoch numbers, and the number of classes. Next, we changed the train (PointGroup/train) and test code (PointGroup/test_alive) to consider our dataset. To visualize and print results we changed the visualization (PointGroup/util/visualize_alive) and evaluation codes (PointGroup/util/eval_alive.py).

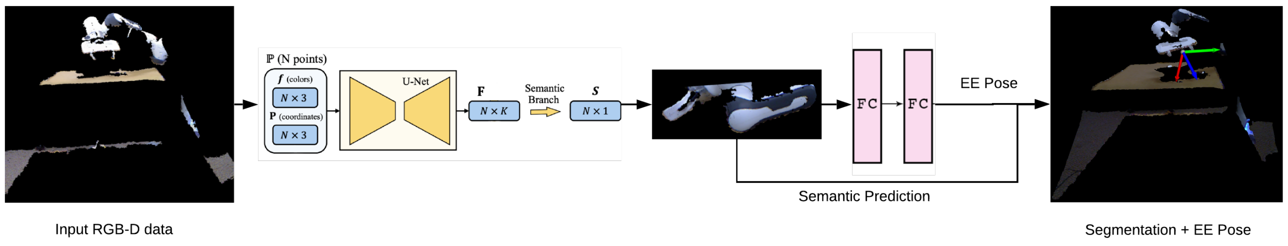


Figure 8: **Method Overview.** Method process voxelized RGB-D data with a sub-manifold U-Net structure to estimate the semantic predictions. After having semantic estimations, then MLP layers regress the 7D EE pose values (position and Quaternions).

After the segmentation part was tested, we moved to regress the end effector. The newly generated RobotNet directory includes our new model (RobotNet/model/robotNet/robotNet) and necessary changes done in affected parts due to loading the end-effector pose for training. In this part, implementations voxelization and multi-layer perceptron with both training and testing samples can be found.

We used several libraries and programs to check our implementation. The notable ones are the Open3d [4] to 3d visualizations and RViz [5] to frame and robot visualizations. Please see Figure 9 for visualizations.

We organize and develop our code using the git. The latest project code can be accessed via Github . (https://github.com/bcsefercik/comp551-project) We also spend time to generating a Readme for the non-code related parts such as data collection. Readmes inside the PointGroup and RobotNet were further modified to show how to run them.

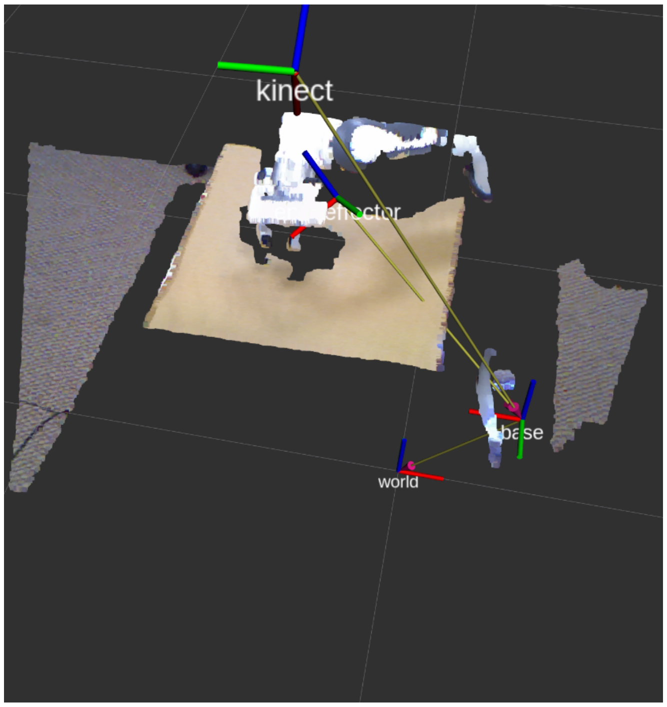


Figure 9: **End Effector Pose Estimation.** To investigate the EE pose estimation method, Rviz is leveraged to visualize frames and point cloud data on the Train set. As the visualizations suggest, issues exist in synchronizing EE pose and point cloud data.

3 Results & Discussion

We described the details and workings of our pipeline and the modules in the pipeline in the previous sections. In Figure 6, you can see changes in our respective loss values as training goes on. It is safe to say that our segmentation modules works accurately based on loss plot and visual outputs.

In plot b, we move onto the perceptron training after epoch 5. As you can see, the loss value converges just above 1. This is version is not able to fit dataset requirements and gives rather large errors. After certain amount of investigation we concluded that there is an error in the data which could not be fixed in time due the reason we have stated above.

In our future work, we will work on the fix of the error in data and further experiment with the second module in order to solve pose estimation problem.

References

[1] Li Jiang, Hengshuang Zhao, Shaoshuai Shi, Shu Liu, Chi-Wing Fu, and Jiaya Jia. Pointgroup: Dual-set point grouping for 3d instance segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4867–4876, 2020.

[2] 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 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[3] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5828–5839, 2017.

[4] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

[5] Hyeong Ryeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. Rviz: A toolkit for real domain data visualization. *Telecommun. Syst.*, 60(2):337–345, October 2015.

A Detailed Code Summary

This appendix explains the code snippets we wrote for data collection and visualization. We did not include the modifications done to the PointGroup architecture.
annotate_p3: Annotates the collected data and converts the data format to the pointGroup's format.
apply_tf_to_pcl: Transforms the point cloud data from with respect to kinect to with respect to base.
apply_transformation: Transforms the end effector with respect to base to with respect to kinect.
depth_registered_joint_state_recorder: Records the live point cloud data and the robot's joint states.
depth_registered_recorder: Records the live point cloud data.
depth_registered_to_pcl: Convert pointcloud pickle data to pointcloud pcl data. **get_ee_pose_from_bag:** Converts joint states information inside the bag file to the end effector frame with respect to base.
get_ee_pose_: Converts joint states information inside the recorded pickle file to the end effector frame with respect to base.
joint_states_recorder: Records the joint states of the robot.
pcl_to_np: Converts point cloud data to numpy array.
pickle_converter: Converts python3 pickle to python2 compatible pickle.
replay_recording: Runs the recorded joint states on the robot.
unifier_p3: Combines multiple pointcloud into one.
visualize_pcl: Visualizes the pointcloud and end effector information together.