Deep Hough Voting for Object Detection in 3D Point Clouds
Final Project Report

Paul Jacob
paul.jacob@polytechnique.edu

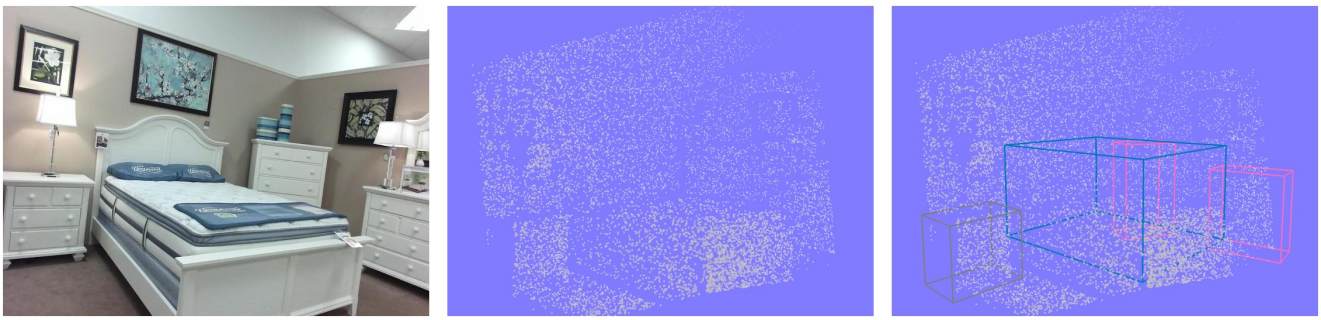

Figure 1. Example of a 3D object detection result in a point cloud. From left to right: photo of the room, input point cloud, and detected objects with the corresponding labels. See table 1 for the correspondence between colors and labels.

Abstract

The paper studied in this project presents an object detection method in point clouds published at ICCV in 2019 [6]. At the time of publication, this method did outperform other state-of-the-art methods in object detection in point clouds when measuring average precision in object detection. This result is achieved by leveraging deep learning architectures especially designed for point clouds, as well as a voting mechanism inspired from classical Hough voting. For this project, I studied the paper, and I did experiment with the method by digging into the official implementation and rewriting some core elements of the method to better understand it. I tested the performance of the method on the SUN RGB-D dataset, and obtained similar performances to the ones reported in the original article. I provide both quantitative and qualitative visualization results, and I also briefly discuss some ideas to improve the performances (in view of similar publications).

1. Introduction

1.1. Deep Hough Voting: the paper

Deep Hough Voting for 3D Object Detection in Point Clouds (Qi et al. 2019 [6]) is an object detection article published in 2019 at ICCV. The general idea of this paper is to present an end-to-end deep learning method to automatically detect object in point clouds. This method is inspired by the classical Hough voting algorithm and leverages recent improvements in deep learning architectures applied to 3D point clouds [8, 9, 11]. The main idea of the method is to retrieve a set of point seeds that vote for the objects centers, which then give oriented bounding box proposals with semantic classification labels. By proposing this voting mechanism, the idea of the authors is to be able to retrieve object centers that are not necessarily close to other points on the cloud (which may only lie on the surface of the objects).

1.2. Applications

Even if the authors do not argue much on applications of their work, we can notice that lots of applications require 3D scene understanding. They may include autonomous driving, civil engineering, video game, entertainment... Being able to detect 3D objects can be key for better software designs (e.g. to be able to modify scenes at a semantic level on 3D modelling softwares), but also for lots of safety applications (e.g. to detect pedestrians in the context of autonomous driving). While the Deep Hough Voting method has mostly been evaluated on indoor scan datasets (namely the SUN RGB-D and ScanNet datasets), one can imagine generalizing this method to different types of objects in the context of 3D scene understanding. Plus, if applicable in real time, such object detection methods could assist a large variety of decision making process in 3D environments.

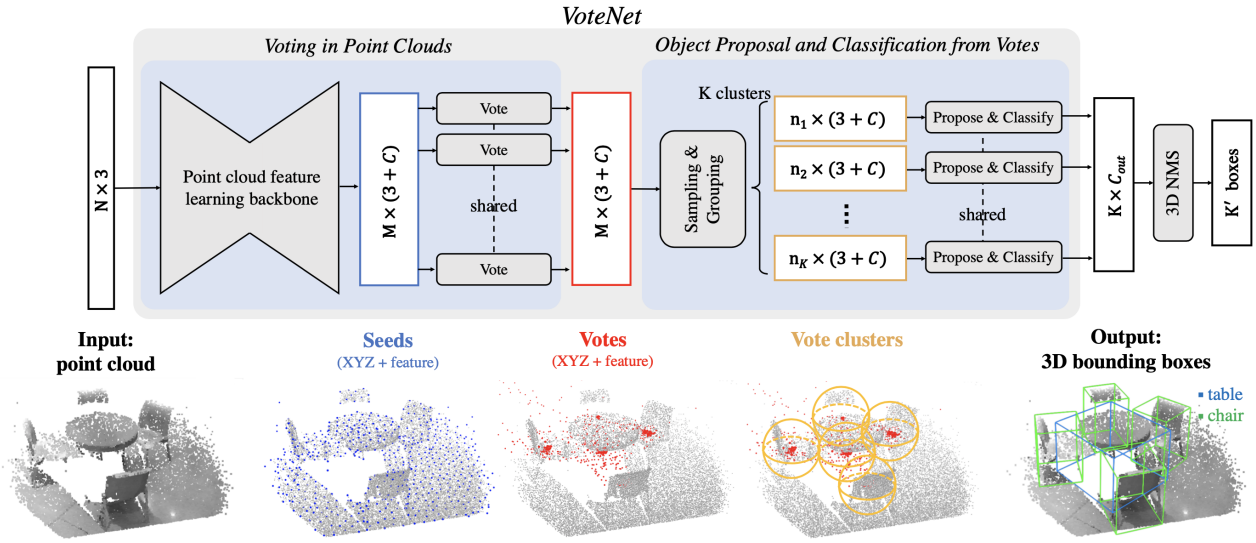

Figure 2. The architecture of the model

2. VoteNet: the model

2.1. Architecture

The method described in the paper relies on a deep learning architecture called *VoteNet*. It can be summed up as follows (figure 2):

- **Backbone module:** First, a backbone 3D point cloud deep learning architecture (namely, a *PointNet++* network) processes the $3 \times N$ point cloud, to retrieve a small subset of these points with C additional features each: that is, a $(3 + C) \times M$ point cloud, with $M << N$. These output points are called the *seeds*, or *interest points*.
- **Voting module:** Then, these *seeds* are processed independently by MLPs with shared weights, and each seed produce another point in $\mathbb{R}^{3+C}$ (3 coordinates + C features). These new points, which have new spatial coordinates, are called the *votes*, and they should lie close to object centers.
- **Proposal module:** Finally, the *votes* are grouped into clusters based on their proximity, and are processed to generate K *object proposals* (3D oriented bounding box + classification scores). After a NMS (non-max suppression) and thresholding step, one can retrieve the final 3D bounding box outputs.

This architecture is entirely end-to-end trainable. As explained in the original paper, this idea of *voting* for the object center comes from the fact that contrary to images or dense representations (e.g. voxels), the centers of objects in point clouds can lie far away from any point. Indeed, depth censors are only able to capture the surface of objects. By introducing a voting module, the authors intend to bypass this issue by automatically generating artificial points that lie close to the object centers. I discuss more in detail the three modules of the architecture in the following paragraphs.

2.1.1 Backbone module

The backbone module is the first part of the model. It takes as input the whole point cloud as a $3 \times N$ matrix, or optionally a $(3 + f) \times N$ matrix where f denotes some hand-crafted additional features. For instance, such additional features can be the RGB color of the points, or the height ($z - z_f$ where z_f is the 1% percentile of all points' z value). Its main goal is to abstract the point cloud into a small subset of points, namely the *seeds*, which each contain more information and context.

Starting from this point cloud and following the principles of *PointNet++* architectures, the backbone module first applies four set abstraction layers (SA), followed by two feature propagation layers (FP). Here I detail quickly these two types of layers, and more details can be found in the original *PointNet++* paper [9].

The set abstraction layers (SA) process the set of points and abstract it to produce a new set with fewer points, but with more features. It is made of three key layers (Sampling layer, Grouping layer and PointNet layer), which respectively selects the set of points to keep, gather the points in local regions, and abstract these regions into feature vectors.

The feature propagation layers (FP) upsample back the reduced point cloud after the set abstraction layers, to obtain features for more points. It uses skip connections ([2])

to propagate the coordinates of previous SA layers, and interpolate the features of the points of previous layers before passing then through a shared MLP.

After the four SA and two FP layers, the result of the backbone module is a $(3+C) \times M$ point cloud, where M is the number of resulting *seed* points and C is the number of additional features per seed. In practice, the authors choose $M = 1024$ and $C = 256$.

2.1.2 Voting module

The voting module processes the *seeds* that were returned by the backbone module, to produce *votes* which will be later fed to the proposal module. It is relatively straightforward: indeed, it is in fact a shared MLP across the features of different points (in $\mathbb{R}^C$), which returns a $\mathbb{R}^{3+C}$ vector corresponding to a spatial coordinates offset in $\mathbb{R}^3$, and a features offset in $\mathbb{R}^C$. These offsets are then added to the original *seeds* coordinates and features (residual connection, as in [2]), to produce the *votes*. At the end, there are as many votes as there are seeds.

2.1.3 Proposal module

The role of the proposal module is to process the *votes* that were given by the voting module, to produce the bounding proposals. It proceeds as follows: first, the votes are clustered into K clusters by farthest point sampling (based on their 3D coordinates, not on their entire features), and then their are grouped by spherical neighbourhoods (i.e. within a radius r) and aggregated by a small *PointNet*-like module to form a single vector per cluster. Essentially, this is the same procedure as a SA layer in a *PointNet++*, so it is implemented as such in practice.

After that, the outputs of this SA layer are processed by a shared MLP to form the bounding box proposals (i.e. the same MLP for all K vote clusters to form K box proposals). These box proposals are vectors that contains both spatial information (the center, the size and the orientation of the proposed object), and semantic information (the classification scores, and objectness confidence scores).

In practice, the authors choose not to output raw regression scores for the bounding box's spatial information (center, size and orientation), but they parameterize this information according to previous papers (see [7], last paragraph of subsection 4.3). For the center of the bounding box, the output is a residual regression score relatively to the center of the vote cluster. For its size, the output is a size classification score based on a set of template sizes, and a size scale residual regression score. For the orientation, the output is an orientation classification score based on sub-intervals of $[0, 2\pi]$, and an orientation residual score.

2.1.4 Post-processing

The design of *VoteNet*'s architecture leads to always having a fixed number of K object proposals. However, different scenes main contain different numbers of detectable objects and one does not want to be constrained to always predict the same number of objects. Plus, the predicted bounding boxes may be redundant, and encode the same object several times.

To overcome the above mentionned issues, the number of box proposals K is set to a pretty large value ($K = 128$ for instance), and the output of the network is post-processed using a non-maximum suppression (NMS) step, with a criterion of 0.25 intersection over union (IoU) overlap. Then, optionally, one can use the objectness confidence scores that were output by the network to threshold the remaining bounding boxes, to keep only the ones which should be considered as objects.

2.2. Loss function

The loss function used to train the *VoteNet* architecture is a weighted combination of several criterions, which all play a different role for the training. It can be summed up as:

$$L_{final} = L_{vote} + \alpha L_{objectness} + \beta L_{box} + \gamma L_{semantic}$$

The different components are, respectively:

- L_{vote}: A supervised regression loss over the vote offsets.
- $L_{objectness}$: A binary classification score over the aggregated votes (i.e. predicted centers), to tell whether they correspond to real object centers.
- L_{box}: A loss which represents the error in 3D bounding box estimation. It has several components, which we will detail in the corresponding paragraph.
- $L_{semantic}$: A semantic classification loss for the objects category.

In practice the values chosen by the authors are $\alpha = 0.5$, $\beta = 1$ and $\gamma = 0.1$. Now describe these four quantities.

2.2.1 Vote regression loss

The vote regression loss is computed as follows:

$$L_{vote} = \frac{1}{|S_{obj}|} \sum_{i \in S_{obj}} ||\Delta x_i - \Delta x_i^\star||$$

where $S_{obj} = \{i \in [M], s_i \text{ on object}\}$ is the set of seeds that lie on objects (in practice, within ground truth bounding boxes), Δx_i is the predicted vote offset and $\Delta x_i^\star$ is the ground truth offset between the seed and the object center.

During the training, this loss explicitly incites the network to predict votes that are close to the object centers, when it makes sense for the seeds. As the votes will be grouped based on their location, and the center of these vote clusters will be the centers of the predicted bounding box, it is crucial that they lie near the object centers.

2.2.2 Objectness classification loss

The objectness loss is a weighted cross-entropy on two *vote* classes, respectively the negative and positive objectness. Given an aggregated vote (or predicted center, equivalently) along with its objectness scores, and two distance thresholds $0 < d_0 < d_1$ there are three possibilities:

- The predicted center can be close to a real object center, at a distance lower than d_0. In this case, the objectness label is positive.
- The predicted center can be far from any real object center, at distances higher than d_1. In this case, the objectness label is negative.
- The predicted center is in a "gray zone", where the nearest real center is at a distance between d_0 and d_1. In this case, this prediction is ignored in the loss computation and there is no penalization.

This loss incites the network to give high positive objectness scores when the predicted center is near a ground truth center. Conversely, it incites false objectness when the predicted center is too far from a real object center.

In practice, the authors choose the value $d_0 = 0.3$ and $d_1 = 0.6$. They also decide to put a larger weight on positive objectness, with a weight distribution of $(0.2, 0.8)$. I believe that playing with these weight values can either incite our model to have a good precision or a good recall.

2.2.3 Bounding box loss

The bounding box loss is a combination of several spatial error quantification metrics. It is important to notice that it is computed only on the boxes which correspond to positive proposals (and normalized accordingly), according to the labelling of previous paragraph 2.2.2 (otherwise it would not make sense since there is no ground truth bounding box to compare with!).

For the positive proposals, the supervision is done according to the closest ground truth bounding box (in terms of object centers proximity). This loss can be summed up as:

$$L_{box} = L_{center} + L_{angle} + L_{size}$$

where:

- L_{center} is a Chamfer loss between predicted centers and ground truth centers. Choosing a Chamfer loss implies that each ground truth object center should be near a predicted object center, and vice-versa.
- $L_{angle} = 0.1L_{angle-cls} + L_{angle-reg}$ is a combination of angle classification according to subintervals of $[0, 2\pi]$, and a residual regression. It corresponds to the parametrization as explained in paragraph 2.1.3.
- $L_{size} = 0.1L_{size-cls} + L_{size-reg}$ is a combination of size classification according to a set of template sizes, and a residual scale regression. Similarely, it corresponds to the parametrization as explained in paragraph 2.1.3.

The regression losses are Huber losses, and the classification losses are Cross Entropy losses. In the end, this bounding box estimation loss is pretty sophisticated. To sum up briefly, it incites positive bounding boxes (i.e. corresponding to real objects) to be adequately placed in the 3D space, both in terms of position, size and orientation.

2.2.4 Semantic classification loss

Similarely to the bounding box loss, the semantic classification loss is computed only on the semantic classification scores corresponding to positive proposals, and the supervision is done according to the label of the closest ground truth object (i.e. we choose the closest object category as the ground truth label). The chosen criterion is a classic Cross Entropy loss.

2.3. Training details

In the original paper, the authors perform a training over 180 epochs, using the optimizer Adam [4] which has proven to be very efficient over a large variety of neural networks trainings. They use a batch size of 8, and an initial learning rate of 0.001 which is divided by 10 after 80 epochs, and once again after 120 epochs. They also use batch normalization [3] with a momentum that goes from 0.5 to 0.999, using the formula $BN_{momentum} = 1 - max(0.001, 0.5^{P+1})$ where P is 0 at the beginning and is increased every 20 epochs.

Each point cloud that is fed to the network during the training is obtained by randomly sub-sampling $20k$ points from the original cloud, and performing random data augmentations such as horizontal flips, rotations around the vertical axis with an angle in $[-30°, 30°]$, and scaling by a factor in $[0.85, 1.15]$. In addition to the three spatial coordinates, the authors add a fourth feature (the height), which I defined in paragraph 2.1.1. They choose not to use the RGB color features for the final training.

3. Implementation & Tests

In this part I give some insights about my experiments and tests, with more details on the code and on the choices that I made.

3.1. Implementation

3.1.1 Overview

My implementation is broadly inspired from the original implementation of the method[1]. I found the original code very well organized. As there are a lot of elements (votes, box centers, box size and orientation scores, objectness scores, semantic classification scores...) that are important to keep for the loss computation, I found very ingenious how they decided to use a dictionary to encode all of the different information that is obtained during the network's successive forward passes.

In order to better understand the model's architecture, the training mechanisms and the loss used by the authors, I decided to partially rewrite the architecture and loss computation, with different code organization and some additional comments. The training and evaluation scripts are also adapted from the original code.

I made some small additional changes (added support for a progress bar, modified the results export to color the bounding boxes with their labels), but the general structure remains the same.

3.1.2 Code structure

The code is structured as follows:

`main_train.py` contains the training script. It performs regular checkpoints of the model and optimizer state, and stores logs about the evolution of the different metrics.

`main_eval.py` contains the evaluation script. It also stores some visual results, that can be loaded using *MeshLab*.

`models` contains the different classes that compose the architecture.

`losses` contains the different losses.

`pointnet2` contains a bank of useful functions and tools to install and use *PointNet++* layers.

`sunrgbd` contains tools to transform, load and compute informations on the SUN RGB-D dataset.

`utils` contains all sort of useful functions.

[1] https://github.com/facebookresearch/votenet

3.2. Tests

I decided to evaluate the performance of *VoteNet* on the SUN RGB-D Dataset [10]. My tests and experiments were mostly training the network, playing with the hyperparameters and the dataset preprocessing, and evaluating the results of my different trainings, while comparing to the ones of the original article.

First, I downloaded the SUN RGB-D Dataset and performed preparation of the data according to the official instructions. It allowed to retrieve 3D point clouds with 3D bounding boxes for the 10 most present object category, starting from the RGB-D images. I decided to create three different datasets of different sizes: with 10k,20k and 50k points per point clouds each. Hence, I was able to balance the training time and the performance.

Most of the time, I used a batch size that was higher than the paper (16 rather than 8), because it led to faster trainings and the results were comparable. The paper generates a dataset with 50k points per cloud and subsample 20k points among them at each batch iteration. On my side, I settled on using the dataset with 20k points per cloud, and subsampling only 10k points at each iteration, once again to speed up the training process.

To have access to a GPU, I used Google Colab for all my trainings. Depending on the configuration, the training time could go from approximately 6 hours, to 20 hours at most. Because I had to wait a long time for each training, and Google Colab has GPU time limits, performing a real parameter optimization study could be a bit cumbersome. Still, my best trainings led to similar results to the original paper, and I was able to draw some conclusions about the results that I will detail in part 4.

4. Results

4.1. Qualitative Results

The results that I present here are obtained using the training configuration suggested by the paper, but with 10k points subsampled on each cloud rather than 20k, to speed up the training. Also, the batch size was set to 16.

Figures 1, 3 and 4 present some visual results of 3D object detection at inference time. The bounding boxes that are shown are the ones that remain after a NMS Step with 0.25 IoU criterion, and a confidence thresholding of 0.5 objectness probability. Table 1 gives the color palette that defines the correspondence between the colors of the bounding boxes, and the category labels. One can see that most of the time, the bounding boxes are accurate, both in terms of spatial information (position, orientation, size), and semantic information.

In some cases, the prediction is wrong. For instance, when the object has a difficult shape (e.g. the two curved desks in figure 3), the model sometimes predicts two bound-

ing boxes as if there were two separate objects. The same phenomenon can appear when the NMS step fails to remove all redundant bounding boxes (e.g. the toilet in figure 4).

Also, some object categories are similar, in a sense that objects from both categories can look like each other (e.g. desks and tables). Hence, the predicted object category may not be the ground truth one, but still valid to some extent (e.g. the second and fourth rows in figure 3, where the desk is predicted as a table). Finally, there are some cases where the model is obviously wrong (e.g. the third row in figure 3 where two trashcans are considered as chairs).

Color	Label
Blue	Bed
Orange	Table
Green	Sofa
Red	Chair
Purple	Toilet
Brown	Desk
Pink	Dresser
Gray	Nightstand
Olive	Bookshelf
Cyan	Bathtub

Table 1. Correspondences between colors and labels

4.2. Quantitative results

To evaluate the performance of the model, the original paper uses the average precision (mAP) with a 3D IoU threshold of 0.25, which was the metric proposed by [10].

With the configuration given in the original paper, the authors obtain a mAP of 57.7. With the same configuration, and the small changes that I detailed in subsection 3.2 to speed up the training, I obtain a mAP of 57.4. I would argue that the performances are pretty much similar.

I have tried to tweak the training hyper-parameters (batch size, learning rate...) in order to see if I was able to improve the evaluation results. Most of the time, the results that I obtained were on the same order of magnitude (between 55 and 57), except for large changes which prevented the network to train correctly. I also did a training with the RGB color values in addition to the coordinates, but it did not improve the performances (I obtained a mAP of 56.3).

I believe that the authors already selected very decent hyper-parameters for this model, and that in order to drastically improve the performances, one would have to bring new ideas to overcome this current method. I expose some ideas in the next section.

Original results	My results	My results (+RGB)
57.7	57.4	56.3

Table 2. mAP@0.25 with VoteNet on SUN RGB-D validation

5. Suggestions for improvement

One main bottleneck in the method studied here (that is in fact common to a lot of data-driven approaches) comes from the data itself, and in particular the annotations. By exploring the data, I have noticed that some objects are visible in the RGB-D images, but the corresponding point clouds are not annotated with bounding boxes. That is, if the network predicts an object that is indeed here, but not in the list of ground truth bounding boxes, it could incur a wrong loss. As always, using data with more annotations and more details should feed the network with more accurate examples, and thus, improve the performances at test time.

Without having to label more data by hand, one could try to artificially enrich the dataset by performing more data augmentation, or more data preprocessing. More data augmentation could be a way to artificially create new training examples. In the original implementation, the authors already perform some data augmentation as said in paragraph 2.3. One could imagine adding more augmentation pipelines, such as adding Gaussian noise to the points, or cropping the input point cloud.

In the paper, the authors already add a new feature in addition to the (x, y, z) coordinates, namely the height z_f. If the point colors are also available, they might improve the detection results in some cases. During the NPM3D course, we have seen examples of point descriptors that can be computed using local neighbourhoods, such as verticality, linearity, planarity and sphericality. These could be also given to the network, and potentially induce him to learn better (for instance, a rectangle composed of vertical points could be a dresser...).

Apart from improvements on the data, one can imagine changes on the architecture itself that could improve the performances in object detection. After the paper introducing *VoteNet*, new methods for 3D object detection in point clouds have been published, and were able to give better performances on SUN RGB-D and ScanNet benchmarks [1, 12, 5].

[12] extends the idea of *VoteNet* and predicts hybrid geometric primitives (bounding box centers, bounding box face centers, and bounding box edges centers), before converting then to object proposals. [1] proposes a new type of hierarchical graph network which better captures shape information of objects. Finally, [5] adds a 2D image as an additional information to the point cloud, and combines votes from both a 2D object detector and a *VoteNet*-like architecture. Due to the additional data, this method's results are not directly comparable to *VoteNet*, but it remains interesting to note that adding this additional information helps improving the object detection. Finally, these three methods lead to respectively 60.1, 61.6, and 63.4 mAP on the SUN RGB-D dataset.

6. Conclusion

As a conclusion, my work for this project has been to study, partially rewrite, train and test the deep learning method presented in the article by Qi et al. (2019) [6]. Although my computational power was limited, I have managed to perform decent trainings and reach performances similar to the ones presented in the original paper.

More generally, this project was the occasion for me to dive into recent methods for detection of 3D objects in point clouds, and point cloud deep learning in general. It allowed me to get a better understanding of some recent challenges in 3D deep learning. With more material means and probably more time, I would have enjoyed experimenting and digging more into the architecture to perhaps find ways to enhance the results.

References

[1] J. Chen, B. Lei, Q. Song, H. Ying, D. Z. Chen, and J. Wu. A hierarchical graph network for 3d object detection on point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. 6

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. 2, 3

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015. 4

[4] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017. 4

[5] C. R. Qi, X. Chen, O. Litany, and L. J. Guibas. Imvotenet: Boosting 3d object detection in point clouds with image votes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 6

[6] C. R. Qi, O. Litany, K. He, and L. J. Guibas. Deep hough voting for 3d object detection in point clouds. In *Proceedings of the IEEE International Conference on Computer Vision*, 2019. 1, 7

[7] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas. Frustum pointnets for 3d object detection from rgb-d data. *arXiv preprint arXiv:1711.08488*, 2017. 3

[8] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *arXiv preprint arXiv:1612.00593*, 2016. 1

[9] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017. 1, 2

[10] S. Song, S. P. Lichtenberg, and J. Xiao. Sun rgb-d: A rgb-d scene understanding benchmark suite. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 567–576, 2015. 5, 6

[11] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 2019. 1

[12] Z. Zhang, B. Sun, H. Yang, and Q. Huang. H3dnet: 3d object detection using hybrid geometric primitives, 2020. 6

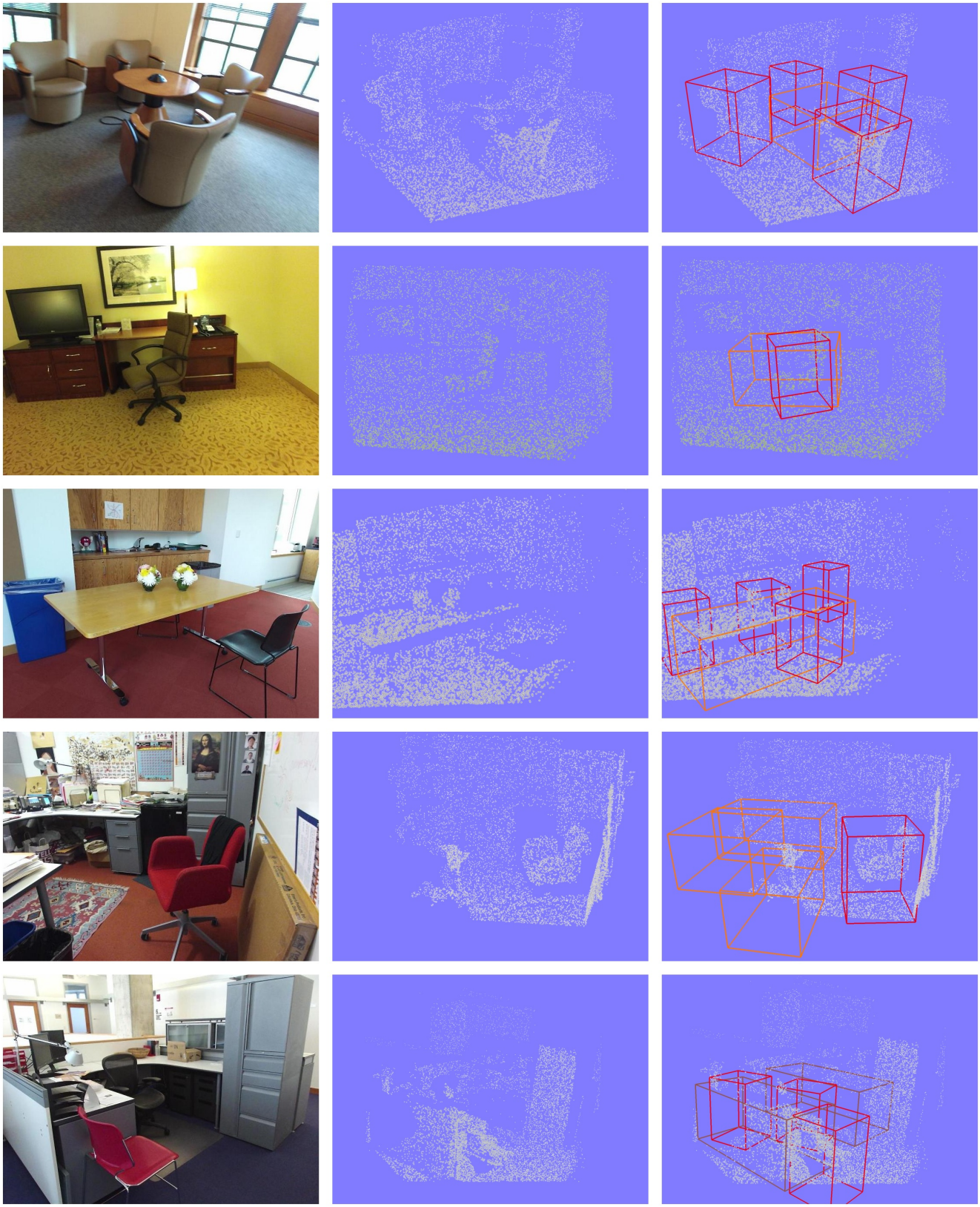

Figure 3. Examples of object detection results in a point cloud. From left to right: photo of the room, input point cloud, and detected objects with the corresponding labels. See table 1 for the correspondence between colors and labels.

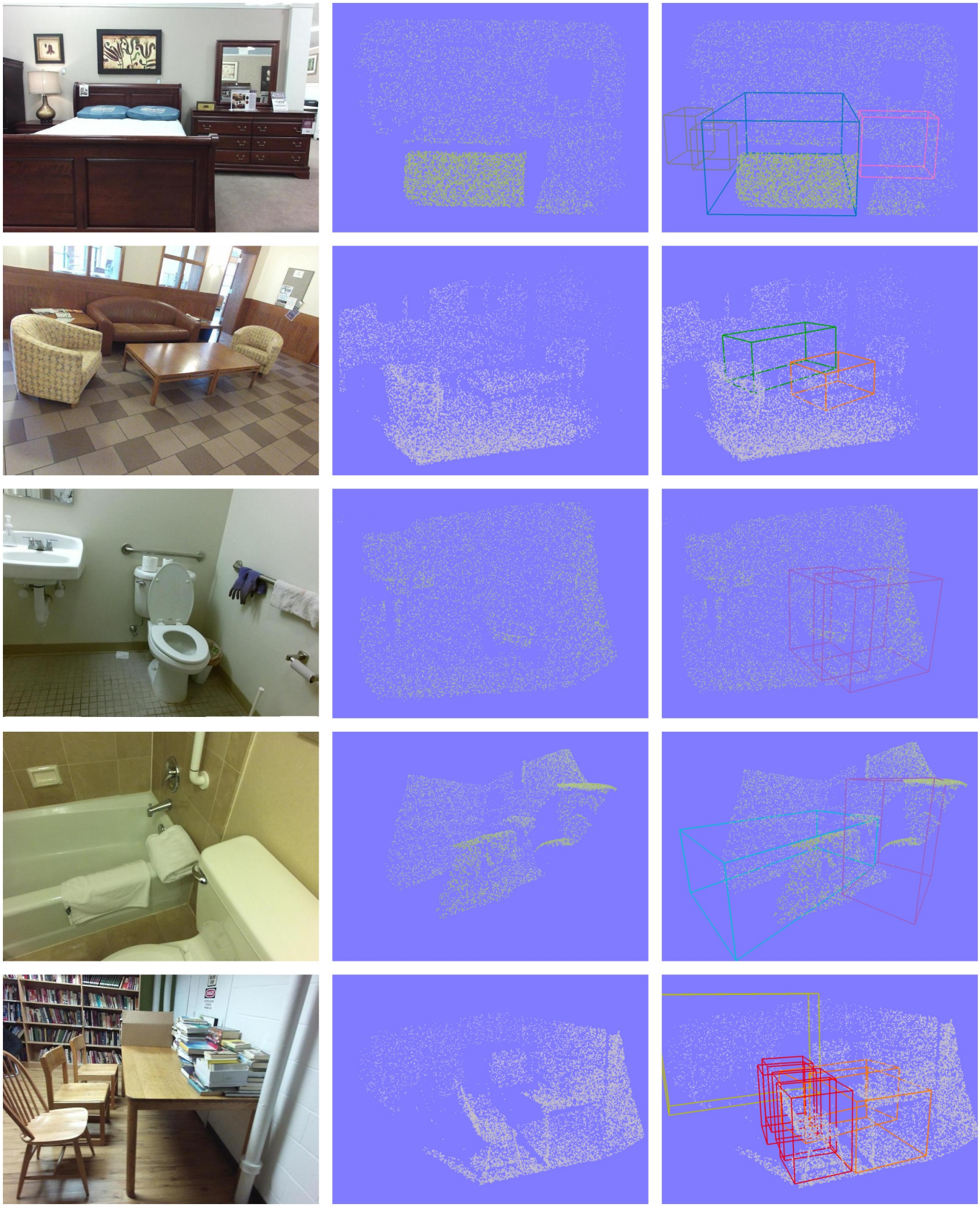

Figure 4. Examples of object detection results in a point cloud. From left to right: photo of the room, input point cloud, and detected objects with the corresponding labels. See table 1 for the correspondence between colors and labels.