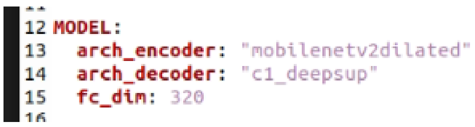
i. Code

1. Training and validating segmentation model

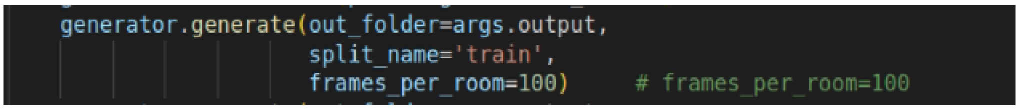
I use MobileNetV2dilated as the encoder and C1_deepsup as the decoder to constitute my semantic segmentation model.

Below is the MODEL part in ".yaml" file (the file specifying parameters used in training and validation process).



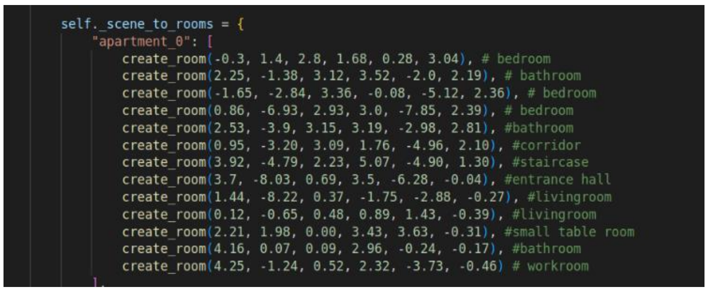
```
MODEL:
  arch_encoder: "mobilenetv2dilated"
  arch_decoder: "c1_deepsup"
  fc_dim: 320
```

For training data collection, I set the parameter "frames_per_room" as 100.



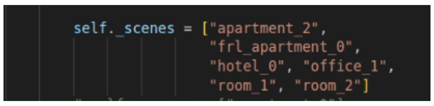
```
    generator.generate(out_folder=args.output,
                       split_name='train',
                       frames_per_room=100)      # frames_per_room=100
```

There are 13 rooms in apartment_0. I collected images from all of these room.

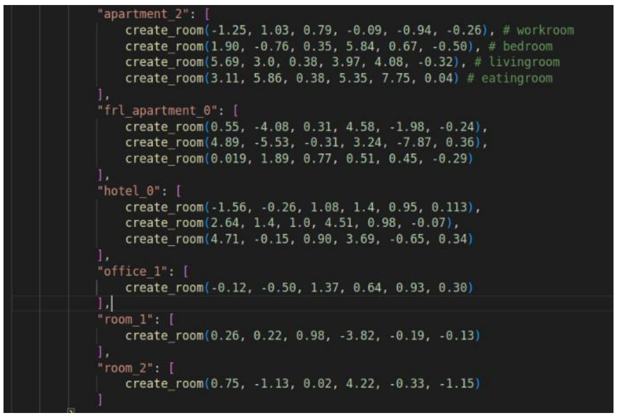


```
        self._scene_to_rooms = {
            "apartment_0": [
                create_room(-0.3, 1.4, 2.8, 1.68, 0.28, 3.04), # bedroom
                create_room(2.25, -1.38, 3.12, 3.52, -2.0, 2.19), # bathroom
                create_room(-1.65, -2.84, 3.36, -0.08, -5.12, 2.36), # bedroom
                create_room(0.86, -6.93, 2.93, 3.0, -7.85, 2.39), # bedroom
                create_room(2.53, -3.9, 3.15, 3.19, -2.98, 2.81), #bathroom
                create_room(0.95, -3.20, 3.09, 1.76, -4.96, 2.10), #corridor
                create_room(3.92, -4.79, 2.23, 5.07, -4.90, 1.30), #staircase
                create_room(3.7, -8.03, 0.69, 3.5, -6.28, -0.04), #entrance hall
                create_room(1.44, -8.22, 0.37, -1.75, -2.88, -0.27), #livingroom
                create_room(0.12, -0.65, 0.48, 0.89, 1.43, -0.39), #livingroom
                create_room(2.21, 1.98, 0.00, 3.43, 3.63, -0.31), #small table room
                create_room(4.16, 0.07, 0.09, 2.96, -0.24, -0.17), #bathroom
                create_room(4.25, -1.24, 0.52, 2.32, -3.73, -0.46) # workroom
```

For others scenes, I also chose scenes with total 13 rooms. These scenes are apartment_2, frl_apartment_0, hotel_0, office_1, room_1 and room_2.



```
        self._scenes = ["apartment_2",
                        "frl_apartment_0",
                        "hotel_0", "office_1",
                        "room_1", "room_2"]
```



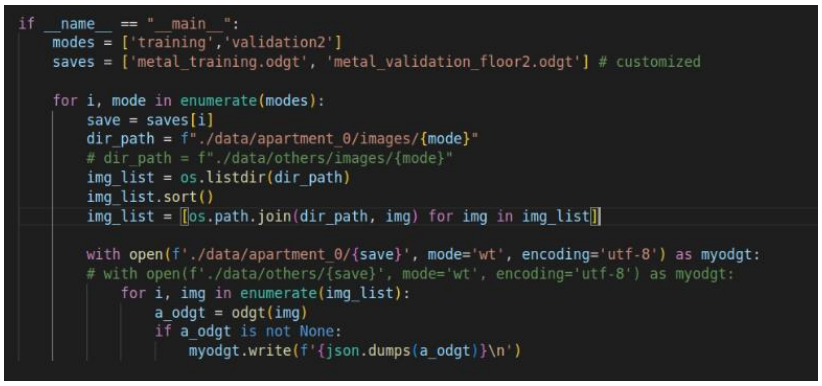
```
            "apartment_2": [
                create_room(-1.25, 1.03, 0.79, -0.09, -0.94, -0.26), # workroom
                create_room(1.90, -0.76, 0.35, 5.84, 0.67, -0.50), # bedroom
                create_room(5.69, 3.0, 0.38, 3.97, 4.08, -0.32), # livingroom
                create_room(3.11, 5.86, 0.38, 5.35, 7.75, 0.04) # eatingroom
            ],
            "frl_apartment_0": [
                create_room(0.55, -4.08, 0.31, 4.58, -1.98, -0.24),
                create_room(4.89, -5.53, -0.31, 3.24, -7.87, 0.36),
                create_room(0.019, 1.89, 0.77, 0.51, 0.45, -0.29)
            ],
            "hotel_0": [
                create_room(-1.56, -0.26, 1.08, 1.4, 0.95, 0.113),
                create_room(2.64, 1.4, 1.0, 4.51, 0.98, -0.07),
                create_room(4.71, -0.15, 0.90, 3.69, -0.65, 0.34)
            ],
            "office_1": [
                create_room(-0.12, -0.50, 1.37, 0.64, 0.93, 0.30)
            ],
            "room_1": [
                create_room(0.26, 0.22, 0.98, -3.82, -0.19, -0.13)
            ],
            "room_2": [
                create_room(0.75, -1.13, 0.02, 4.22, -0.33, -1.15)
            ]
```

Thus, total 1300 images were collected from apartment_0 and other scenes respectively.

After having these images, I ran my data_to_odgt.py to convert these images in to an image dataset that can be fed into the model.

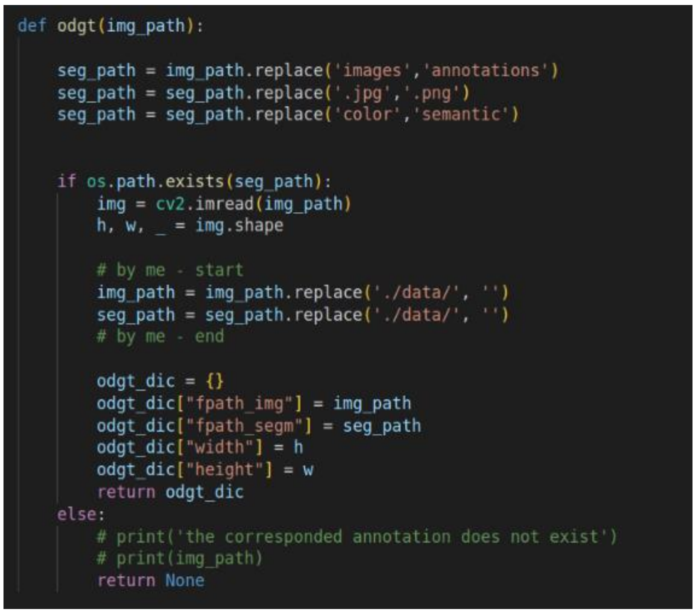
The following code would read all the image files in a folder and then write the

necessary information into “.odgt” file, which is the file of the image dataset.
Reference: https://hackmd.io/wNGlmMq2RC-lY3l8JhO4SA?view



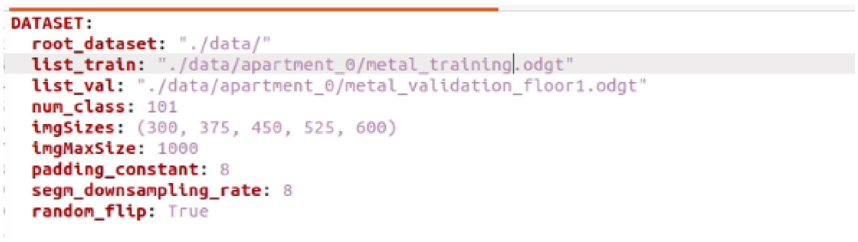
```python
if __name__ == "__main__":
    modes = ['training','validation2']
    saves = ['metal_training.odgt', 'metal_validation_floor2.odgt'] # customized

    for i, mode in enumerate(modes):
        save = saves[i]
        dir_path = f"./data/apartment_0/images/{mode}"
        # dir_path = f"./data/others/images/{mode}"
        img_list = os.listdir(dir_path)
        img_list.sort()
        img_list = [os.path.join(dir_path, img) for img in img_list]

        with open(f'./data/apartment_0/{save}', mode='wt', encoding='utf-8') as myodgt:
        # with open(f'./data/others/{save}', mode='wt', encoding='utf-8') as myodgt:
            for i, img in enumerate(img_list):
                a_odgt = odgt(img)
                if a_odgt is not None:
                    myodgt.write(f'{json.dumps(a_odgt)}\n')
```

Function “odgt” would return the line to be written into the “.odgt” file.



```python
def odgt(img_path):

    seg_path = img_path.replace('images','annotations')
    seg_path = seg_path.replace('.jpg','.png')
    seg_path = seg_path.replace('color','semantic')


    if os.path.exists(seg_path):
        img = cv2.imread(img_path)
        h, w, _ = img.shape

        # by me - start
        img_path = img_path.replace('./data/', '')
        seg_path = seg_path.replace('./data/', '')
        # by me - end

        odgt_dic = {}
        odgt_dic["fpath_img"] = img_path
        odgt_dic["fpath_segm"] = seg_path
        odgt_dic["width"] = h
        odgt_dic["height"] = w
        return odgt_dic
    else:
        # print('the corresponded annotation does not exist')
        # print(img_path)
        return None
```

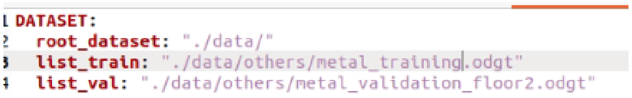
The following is DATASET part in “.yaml” file, which is the file specifying parameters used in training and validation process.
This is the “.yaml” file using images collected from apartment_0 as the training data and using images collected from the first floor of apartment_0 as the validation data. Actually, it should be fine to validate the model using both the images of the first floor and the second floor. Considering that reconstruction of the first floor and the second floor would be done respectively, for my convenience, I just validated them one by one so that predicted semantic images of these two floors won’t be mixed together.



```yaml
DATASET:
  root_dataset: "./data/"
  list_train: "./data/apartment_0/metal_training.odgt"
  list_val: "./data/apartment_0/metal_validation_floor1.odgt"
  num_class: 101
  imgSizes: (300, 375, 450, 525, 600)
  imgMaxSize: 1000
  padding_constant: 8
  segm_downsampling_rate: 8
  random_flip: True
```

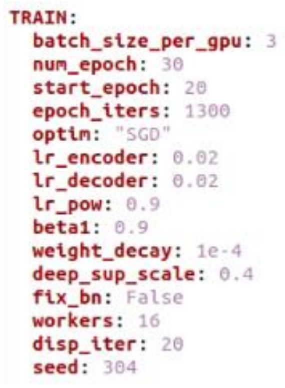
The parameter “num_class” was modified to 101 (as showed above).

Similarly, the “.yaml” file using images collected from other scenes as the training data and using images collected from the second floor of apartment_0 as the validation data is showed in the following picture.



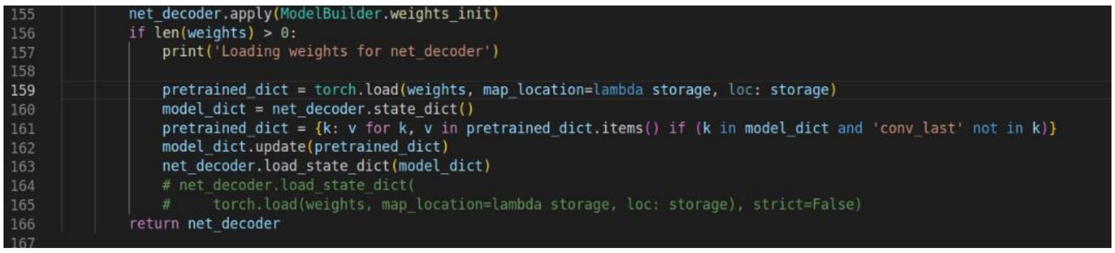
```
DATASET:
  root_dataset: "./data/"
  list_train: "./data/others/metal_training.odgt"
  list_val: "./data/others/metal_validation_floor2.odgt"
```

The following are other parameters used in training process. I changed “epoch_iters” to 1300 so that each image would be selected three times in one training epoch (because “batch_size_per_gpu” is three). Also, I changed “start_epoch” to 20, which means that the model would be finetuned from a pretrained model which was trained until the 20th epoch. By the way, although “num_epoch” is set to 30, I didn’t train the model until the 30th epoch. I stopped training at the beginning of the 26th epoch (because the training accuracy has exceeded 96%).



```
TRAIN:
  batch_size_per_gpu: 3
  num_epoch: 30
  start_epoch: 20
  epoch_iters: 1300
  optim: "SGD"
  lr_encoder: 0.02
  lr_decoder: 0.02
  lr_pow: 0.9
  beta1: 0.9
  weight_decay: 1e-4
  deep_sup_scale: 0.4
  fix_bn: False
  workers: 16
  disp_iter: 20
  seed: 304
```

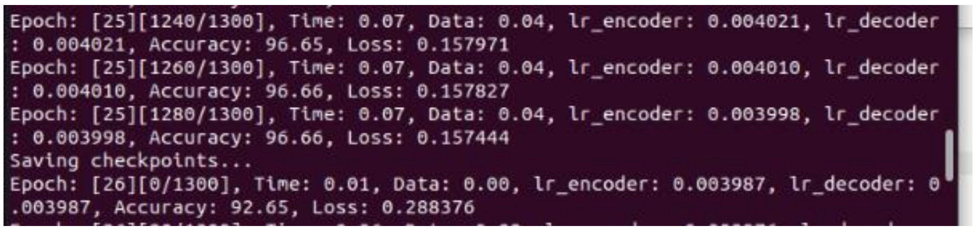
Because the pretrained model was trained on another dataset whose images can be classified to 150 classes, which is different from ours (which is 101), the dimension of some parts of the pretrained weights would be different from that of our weights. We can fix this issue by only loading the pretrained weights whose dimension match with ours. I modified the following part in “model.py”. Since the dimension mismatch happens in “conv_last” layer, I load pretrained weights other than those of “conv_last” layer.



```
        net_decoder.apply(ModelBuilder.weights_init)
        if len(weights) > 0:
            print('Loading weights for net_decoder')

            pretrained_dict = torch.load(weights, map_location=lambda storage, loc: storage)
            model_dict = net_decoder.state_dict()
            pretrained_dict = {k: v for k, v in pretrained_dict.items() if (k in model_dict and 'conv_last' not in k)}
            model_dict.update(pretrained_dict)
            net_decoder.load_state_dict(model_dict)
            # net_decoder.load_state_dict(
            #     torch.load(weights, map_location=lambda storage, loc: storage), strict=False)
        return net_decoder
```

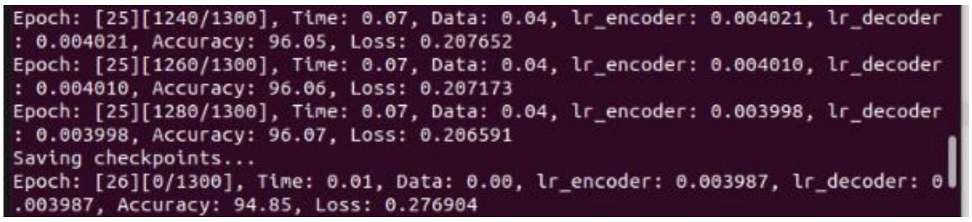
The following two pictures show the training result, including final step accuracy and loss, at the end of the 25th epoch.

Trained on images from apartment_0:



```
Epoch: [25][1240/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.004021, lr_decoder
: 0.004021, Accuracy: 96.65, Loss: 0.157971
Epoch: [25][1260/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.004010, lr_decoder
: 0.004010, Accuracy: 96.66, Loss: 0.157827
Epoch: [25][1280/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.003998, lr_decoder
: 0.003998, Accuracy: 96.66, Loss: 0.157444
Saving checkpoints...
Epoch: [26][0/1300], Time: 0.01, Data: 0.00, lr_encoder: 0.003987, lr_decoder: 0
.003987, Accuracy: 92.65, Loss: 0.288376
```

Trained on images from other scenes:



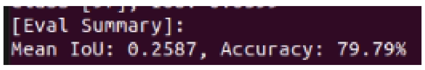
```
Epoch: [25][1240/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.004021, lr_decoder
: 0.004021, Accuracy: 96.05, Loss: 0.207652
Epoch: [25][1260/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.004010, lr_decoder
: 0.004010, Accuracy: 96.06, Loss: 0.207173
Epoch: [25][1280/1300], Time: 0.07, Data: 0.04, lr_encoder: 0.003998, lr_decoder
: 0.003998, Accuracy: 96.07, Loss: 0.206591
Saving checkpoints...
Epoch: [26][0/1300], Time: 0.01, Data: 0.00, lr_encoder: 0.003987, lr_decoder: 0
.003987, Accuracy: 94.85, Loss: 0.276904
```

Before validating the model, we should modify the calculation of mIOU in eval_miltipro.py because there are only 49 categories in apartment_0 (although the number of the total categories is 101).

Now, we can validate the model using images collecting from hw1 (actually I still re-collect them again for reasons that I will explain later). As I mentioned above, I validated the model by splitting images of the first floor and the second floor.

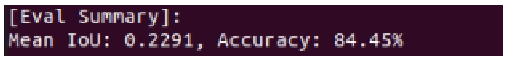
The following four pictures show the validation result.

Validate model trained on apartment_0 images using the first-floor images:



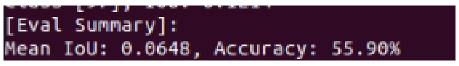
```
[Eval Summary]:
Mean IoU: 0.2587, Accuracy: 79.79%
```

Validate model trained on apartment_0 images using the second-floor images:



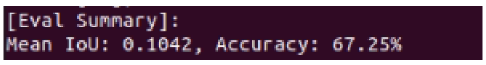
```
[Eval Summary]:
Mean IoU: 0.2291, Accuracy: 84.45%
```

Validate model trained on other scenes images using the first-floor images:



```
[Eval Summary]:
Mean IoU: 0.0648, Accuracy: 55.90%
```

Validate model trained on other scenes images using the second-floor images:

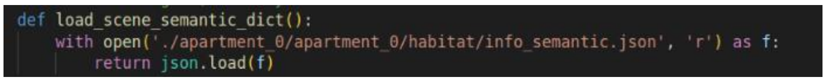


```
[Eval Summary]:
Mean IoU: 0.1042, Accuracy: 67.25%
```

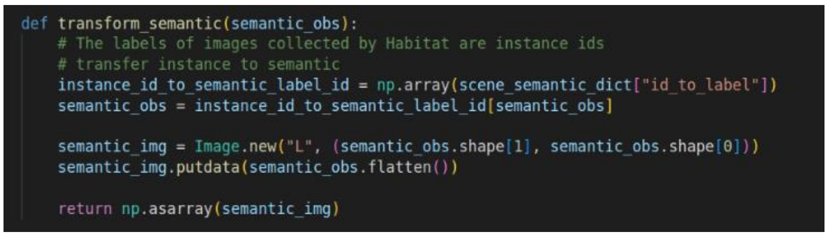
The reason why I re-collected images of hw1 as the validation data:

The semantic image that we originally collected in hw1 is like the following. The different colors represent different instance ids, instead of sematic labels which is used in training and validation process.

Thus, I modified the function “transform_sematic” in load.py provided in hw1. It uses “info_semantic.json” as the dictionary to map the instance ids to semantic labels. After that, I re-collected the image data.

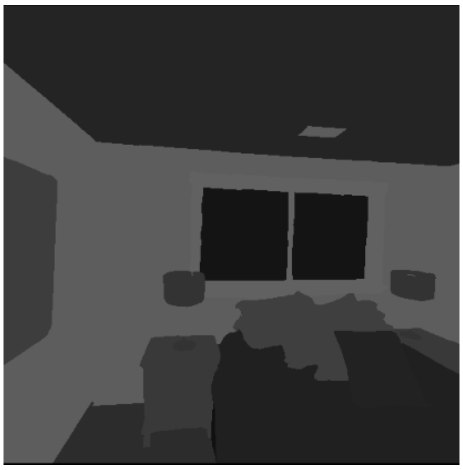


```
def load_scene_semantic_dict():
    with open('./apartment_0/apartment_0/habitat/info_semantic.json', 'r') as f:
        return json.load(f)
```



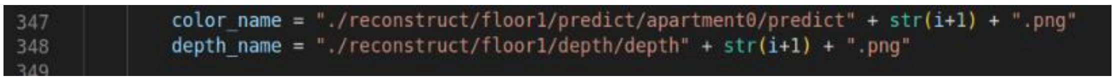
```
def transform_semantic(semantic_obs):
    # The labels of images collected by Habitat are instance ids
    # transfer instance to semantic
    instance_id_to_semantic_label_id = np.array(scene_semantic_dict["id_to_label"])
    semantic_obs = instance_id_to_semantic_label_id[semantic_obs]

    semantic_img = Image.new("L", (semantic_obs.shape[1], semantic_obs.shape[0]))
    semantic_img.putdata(semantic_obs.flatten())

    return np.asarray(semantic_img)
```

The re-collected semantic image is like the following.



2. 3D semantic map reconstruction

Every time before I run the program, I need to change the file path manually.



```
        color_name = "./reconstruct/floor1/predict/apartment0/predict" + str(i+1) + ".png"
        depth_name = "./reconstruct/floor1/depth/depth" + str(i+1) + ".png"
```

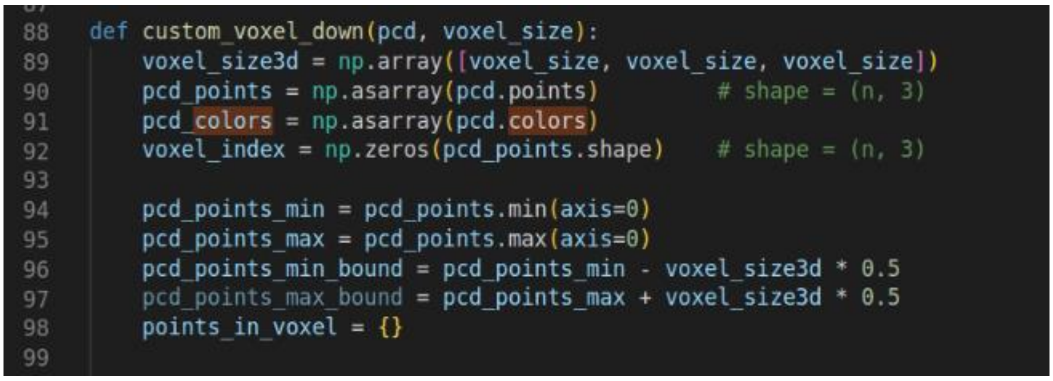
Then, I just enter “python 3d_semantic_map.py” to run my program.

```
python 3d_semantic_map.py
```

Before I started to write my function “custom_down_sample”, I peeked how Open3d implemented its function “VoxelDownSample”. Thus, my implementation is totally based on the implementation of Open3d, except that the color of a down-sampled voxel is decided by the major color of those points in that voxel, instead of the average color.

First, I calculate the minimal x, y and z values of all points in the point cloud and store these three values into a variable “pcd_points_min”. It’s the center of the voxel which has the minimal coordinate value in all x, y, and z direction (It may not be a point in the point cloud). The variable “pcd_points_min_bound” is a corner of the bounding box which surrounds that min-value voxel. No points in the point cloud goes beyond this “pcd_points_min_bound”. Same for “pcd_points_max_bound” (actually this variable isn’t used).

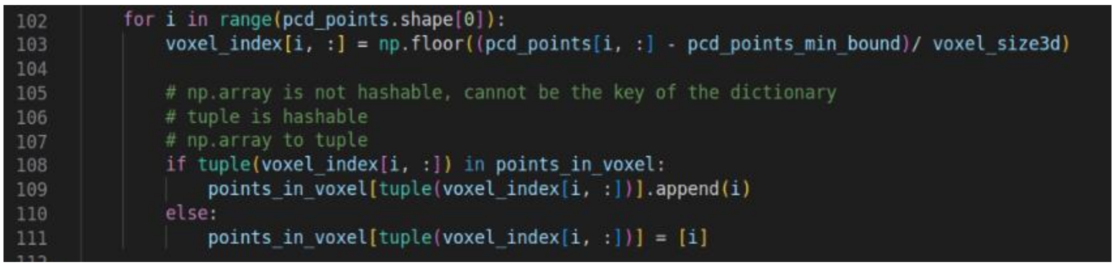
Dictionary “points_in_voxel” would be used to store how many (or which) points belong to a down-sampled voxel. Its “key” is the voxel id (represented as a Tuple); Its “value” is a List storing ids of those points belonging to this voxel.



```python
def custom_voxel_down(pcd, voxel_size):
    voxel_size3d = np.array([voxel_size, voxel_size, voxel_size])
    pcd_points = np.asarray(pcd.points)         # shape = (n, 3)
    pcd_colors = np.asarray(pcd.colors)
    voxel_index = np.zeros(pcd_points.shape)    # shape = (n, 3)

    pcd_points_min = pcd_points.min(axis=0)
    pcd_points_max = pcd_points.max(axis=0)
    pcd_points_min_bound = pcd_points_min - voxel_size3d * 0.5
    pcd_points_max_bound = pcd_points_max + voxel_size3d * 0.5
    points_in_voxel = {}

```

Then, I calculate id of their belonging voxel for every point in the point cloud. Voxel id can be acquired by subtracting “pcd_points_max_bound” from the coordinate of a point, and then dividing it by the voxel size.

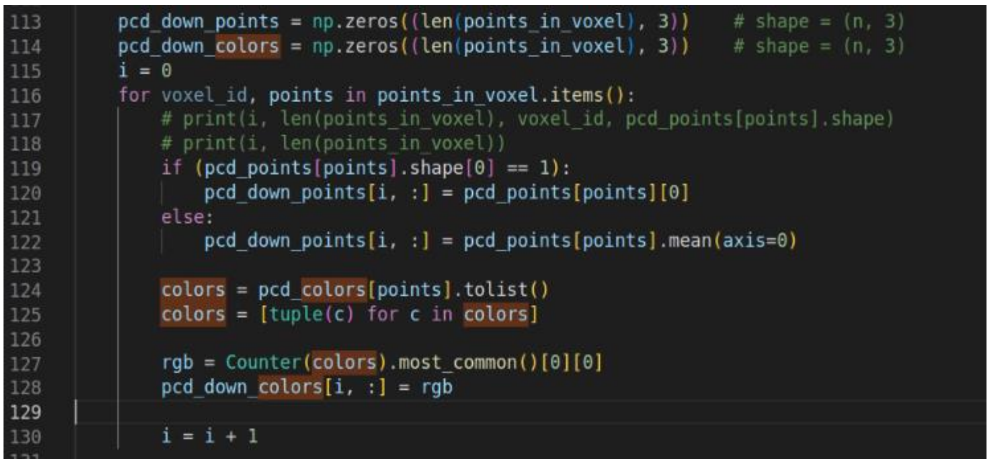
After calculating the voxel id, add the id of this point into the point List of this voxel, which stores all the id of points belonging to this voxel.



```python
    for i in range(pcd_points.shape[0]):
        voxel_index[i, :] = np.floor((pcd_points[i, :] - pcd_points_min_bound)/ voxel_size3d)

        # np.array is not hashable, cannot be the key of the dictionary
        # tuple is hashable
        # np.array to tuple
        if tuple(voxel_index[i, :]) in points_in_voxel:
            points_in_voxel[tuple(voxel_index[i, :])].append(i)
        else:
            points_in_voxel[tuple(voxel_index[i, :])] = [i]
```

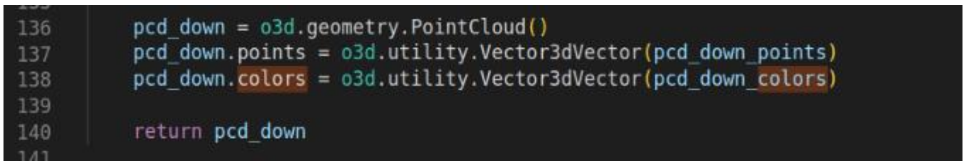
Then I calculate the coordinate and the color of all the down-sampled voxel. The coordinate of a down-sampled voxel is the mean coordinate of all points belonging to the voxel. (It is the implementation method used by Open3d. I just follow it. Actually, I think it can also be implemented by multiplying voxel id by the voxel size, and adding it to "pcd_points_min".)

As for the color of a down-sampled voxel, it's the major color of those points in the voxel.



```
    pcd_down_points = np.zeros((len(points_in_voxel), 3))    # shape = (n, 3)
    pcd_down_colors = np.zeros((len(points_in_voxel), 3))    # shape = (n, 3)
    i = 0
    for voxel_id, points in points_in_voxel.items():
        # print(i, len(points_in_voxel), voxel_id, pcd_points[points].shape)
        # print(i, len(points_in_voxel))
        if (pcd_points[points].shape[0] == 1):
            pcd_down_points[i, :] = pcd_points[points][0]
        else:
            pcd_down_points[i, :] = pcd_points[points].mean(axis=0)

        colors = pcd_colors[points].tolist()
        colors = [tuple(c) for c in colors]

        rgb = Counter(colors).most_common()[0][0]
        pcd_down_colors[i, :] = rgb

        i = i + 1
```

Last, assign these down-sampled voxels to a new point cloud.



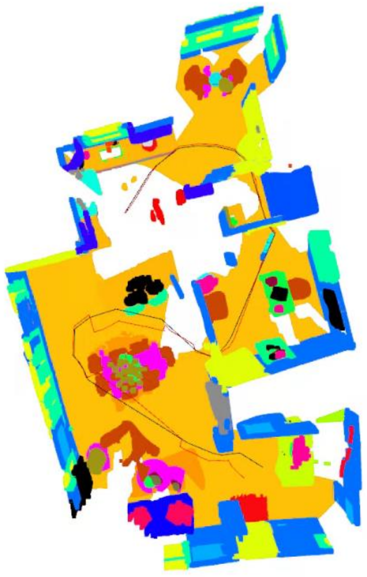
```
    pcd_down = o3d.geometry.PointCloud()
    pcd_down.points = o3d.utility.Vector3dVector(pcd_down_points)
    pcd_down.colors = o3d.utility.Vector3dVector(pcd_down_colors)

    return pcd_down
```

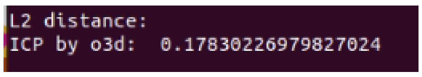
Reference: https://github.com/isl-org/Open3D/blob/master/cpp/open3d/geometry/PointCloud.cpp

ii. Result and Discussion

Using my "custom_down_sample"

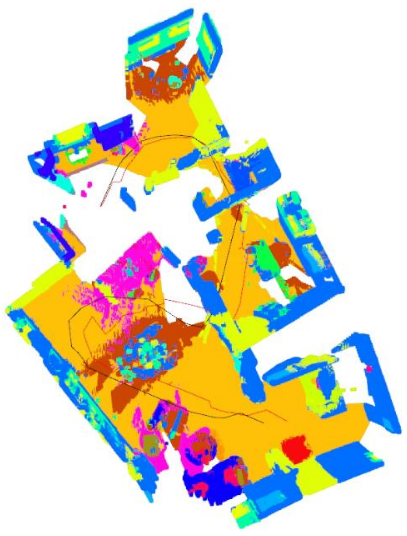
Floor 1 reconstruction (ground truth):

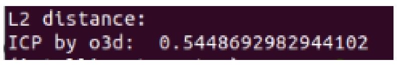




```
L2 distance:
ICP by o3d:   0.17830226979827024
```

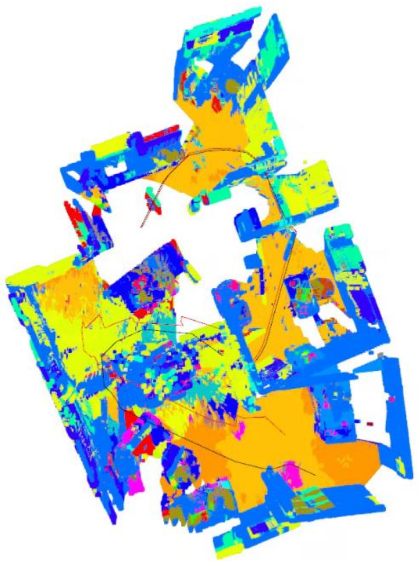
Floor 1 reconstruction (predicted by model trained on apartment 0):

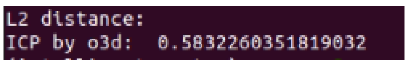




```
L2 distance:
ICP by o3d:  0.5448692982944102
```

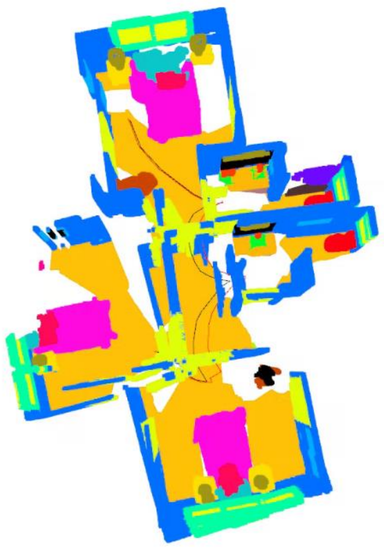
Floor 1 reconstruction (predicted by model trained on other scenes):

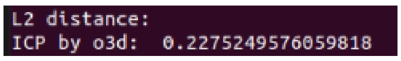




```
L2 distance:
ICP by o3d:  0.5832260351819032
```

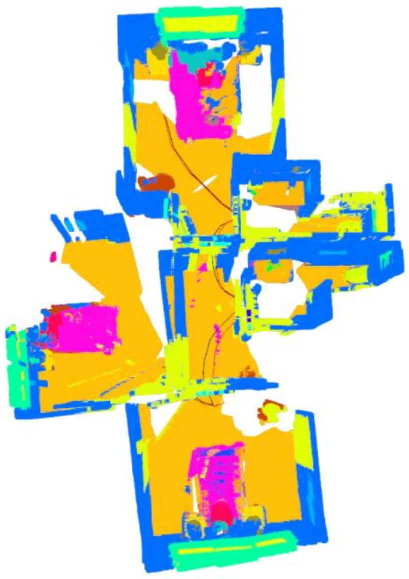
Floor 2 reconstruction (ground truth):

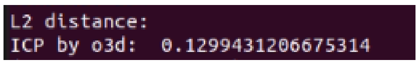




```
L2 distance:
ICP by o3d:  0.2275249576059818
```

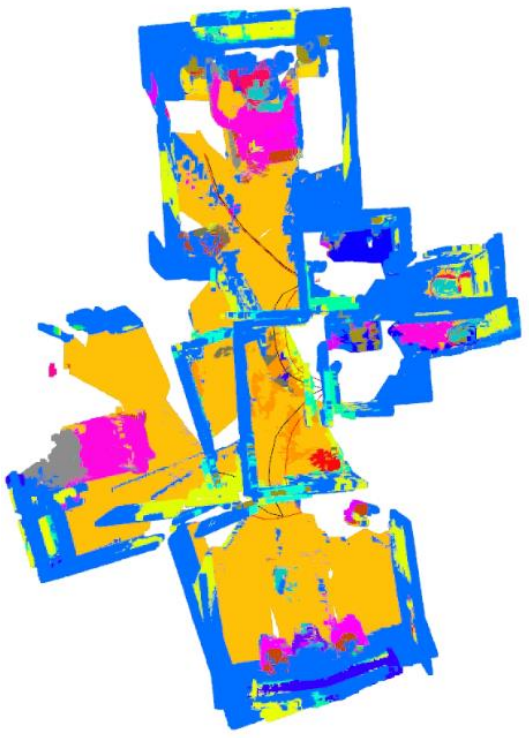
Floor 2 reconstruction (predicted by model trained on apartment 0):

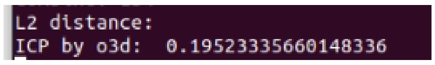




```
L2 distance:
ICP by o3d:  0.1299431206675314
```

Floor 2 reconstruction (predicted by model trained on other scenes):

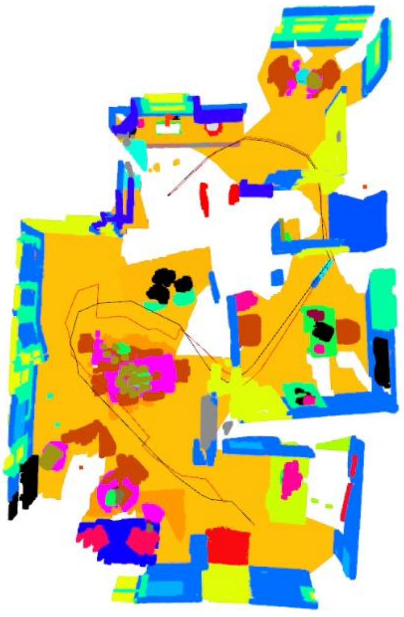


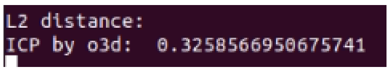


```
L2 distance:
ICP by o3d:  0.19523335660148336
```

Using Open3d "custom_down_sample"

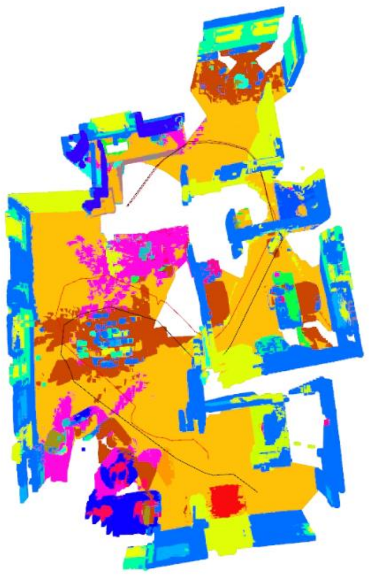
Floor 1 reconstruction (ground truth):

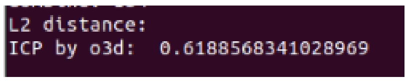




```
L2 distance:
ICP by o3d:  0.3258566950675741
```

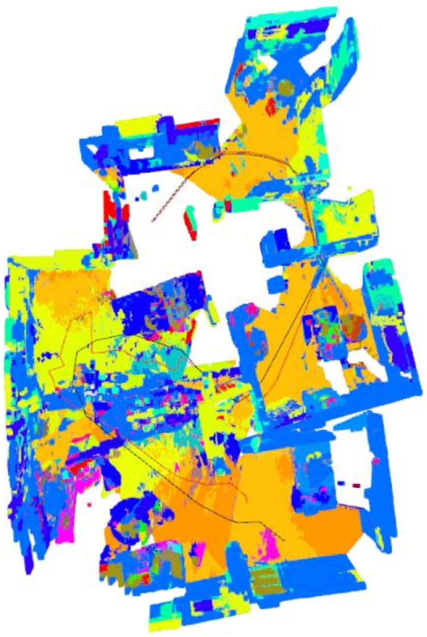
Floor 1 reconstruction (predicted by model trained on apartment 0):

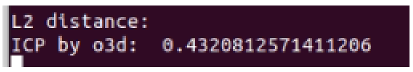




```
L2 distance:
ICP by o3d:  0.6188568341028969
```

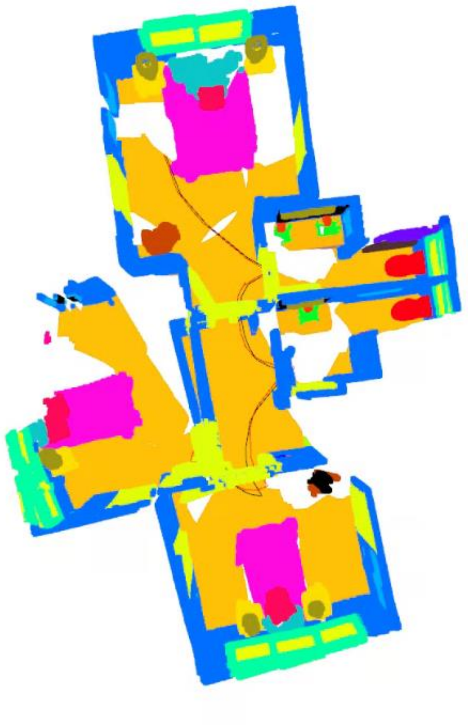
Floor 1 reconstruction (predicted by model trained on other scenes):

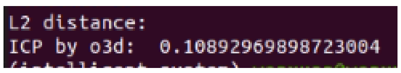




```
L2 distance:
ICP by o3d:  0.4320812571411206
```

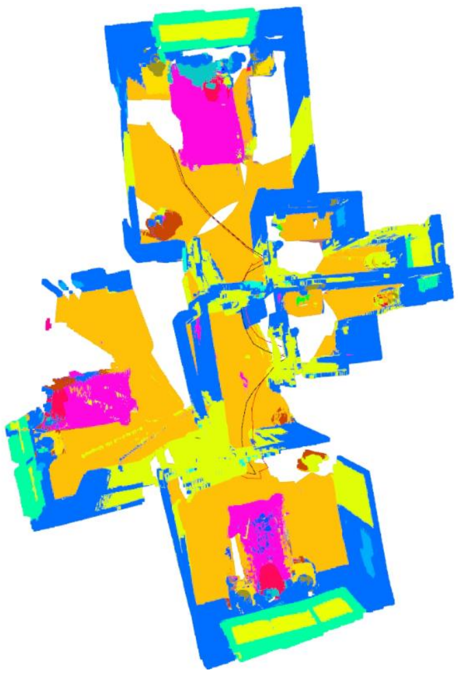
Floor 2 reconstruction (ground truth):

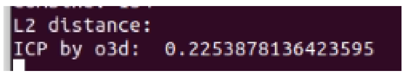




```
L2 distance:
ICP by o3d:  0.10892969898723004
```

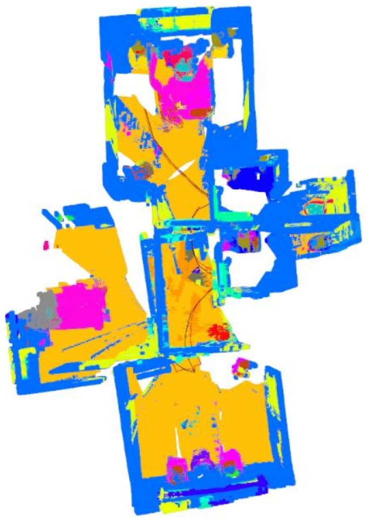
Floor 2 reconstruction (predicted by model trained on apartment 0):

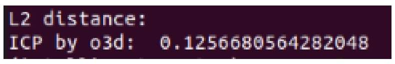




```
L2 distance:
ICP by o3d:  0.2253878136423595
```

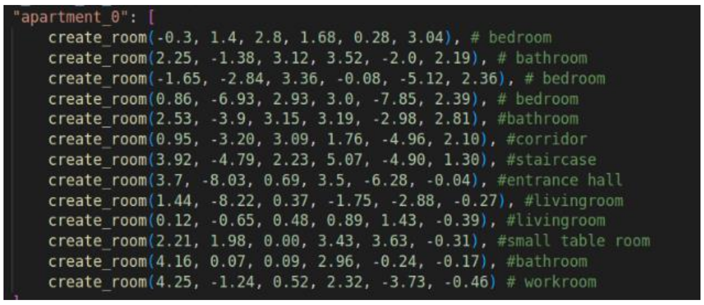
Floor 2 reconstruction (predicted by model trained on other scenes):





```
L2 distance:
ICP by o3d:  0.1256680564282048
```

1. From the reconstruction results and the validation accuracy (and mIOU), we can easily see that the performance of semantic segmentation: ground truth > prediction by model trained on apartment0 > prediction by model trained on others scenes. This is because the more the validation data, which is images all taken in apartment0, is close to the training data, the more the validation accuracy should be. The model can predict better when it has seen data that is similar to that it will see in the future.
2. Also, I observe that the performance of predicted semantic segmentation: floor 2 > floor 1. I think it might be because most of rooms at floor 2 are bedrooms and there are more bedroom images in the training data compared to other rooms like living rooms, workrooms, etc (I guess training data of other scenes also). Thus, it’s reasonable that the prediction performance on floor 2, which majorly consists of bedrooms, would be better than that on floor 1, which is composed by more various rooms.



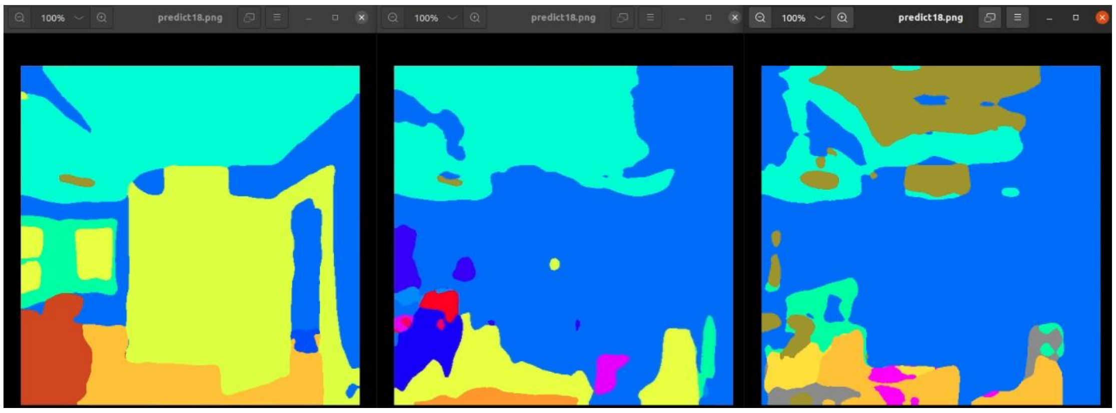
```
"apartment_0": [
    create_room(-0.3, 1.4, 2.8, 1.68, 0.28, 3.04), # bedroom
    create_room(2.25, -1.38, 3.12, 3.52, -2.0, 2.19), # bathroom
    create_room(-1.65, -2.84, 3.36, -0.08, -5.12, 2.36), # bedroom
    create_room(0.86, -6.93, 2.93, 3.0, -7.85, 2.39), # bedroom
    create_room(2.53, -3.9, 3.15, 3.19, -2.98, 2.81), #bathroom
    create_room(0.95, -3.20, 3.09, 1.76, -4.96, 2.10), #corridor
    create_room(3.92, -4.79, 2.23, 5.07, -4.90, 1.30), #staircase
    create_room(3.7, -8.03, 0.69, 3.5, -6.28, -0.04), #entrance hall
    create_room(1.44, -8.22, 0.37, -1.75, -2.88, -0.27), #livingroom
    create_room(0.12, -0.65, 0.48, 0.89, 1.43, -0.39), #livingroom
    create_room(2.21, 1.98, 0.00, 3.43, 3.63, -0.31), #small table room
    create_room(4.16, 0.07, 0.09, 2.96, -0.24, -0.17), #bathroom
    create_room(4.25, -1.24, 0.52, 2.32, -3.73, -0.46) # workroom
```

3. My reconstruction results seem not to be related to whether I use my own down-sample function or I use down-sample function that Open3d

provides. I down sample the point cloud before it is used to perform ICP. However, in ICP, it doesn't matter whether the color of the voxel is average color or the major color because ICP only use the position of the voxel to compute the transformation matrix. There might be some ICP implementations using color to compute transformation matrix, but at least in the implementation of Open3d ICP, it seems that the color information doesn't matter. After computing the transformation matrix, I use non-down-sampled point cloud, that it, the original point cloud which is built directly on the input image, to reconstruct the scene. Thus, down-sampling function doesn't influence my reconstruction results (no significant L2-distance difference between three semantic models might be also a proof showing that the color doesn't really matter in my reconstruction result). It only makes the program execution time longer (but it's still acceptable to me).

4. I also compared the predicted semantic images by different trained model. The left picture is the result from model trained on apartment_0 images until the 24th epoch. The middle one comes from the model trained until the 5th epoch (no pretrained model is used). The right one comes from the model trained on other scenes until the 21th epoch, which is the first epoch after pretrained on another image dataset. Although the middle and the right have nearly equal validation accuracy, the predicted sematic images have different characteristics. If we use the left one as the baseline (near-ground-truth image) to compare the differences between the middle one the right one, we can know the reason of their worse accuracy.
   Basically, the color distribution in the middle image is close to the right one. Its bad accuracy comes from that it cannot predict the boundary of the color well. I think it's because the model used to validation was trained not from a pre-trained model. It uses the data more similar to the validation data (compared to the images used in pre-trained model) to train the model, so its prediction distributes more similarly to the near-ground truth. The right prediction has some colors that seldom appear in the near-round-truth prediction (like the olive on the ceiling and the gray on the ground). It may be because the model was only trained for one epoch after being pre-trained on another image dataset. It has not yet known well about the validation data. It is its previous knowledge on another image dataset that gives it some abilities to predict the coming images (like it predicts better for the ground compared to the middle one). However, it also causes that it predicts a larger amount of color that seldom appears in the near-ground-

truth image.


100%
predict18.png
100%
predict18.png
100%
predict18.png