Perception and Decision Making

in Intelligent Systems Homework 2 Report

Student ID:312512061

Name: 劉郁昇

1. Implementation

Task 1:

a. Code

- Data collection

  We are using the code “data_gernerator.py” to collect data.

  Dataset1: collecting data from apartment_1, apartment_2, frl_apartment_0, frl_apartment_1, room_0, room_1 , room_2, hotel_0, office_0, office_1.

  Dataset2: Collect data from apartment_0

  We can modify figure1 to choose the data

```
self._scenes = ["apartment_1", "apartment_2",
                "frl_apartment_0", "frl_apartment_1",
                "hotel_0", "office_0", "office_1",
                "room_0", "room_1", "room_2"]
# self._scenes = ["apartment_0"]

self._height = 512
self._width = 512
```

  Fig1. data_gernerator.py

  Shown in Table1, it is the data collections for training.

  Table1. image collection

	Training	validation
Dataset1	2000 pictures	400 pictures
Dataset2	1300 pictures	260 pictures

- **Generating odgt file**

  Then, we have to customize our dataset and create .odgt files for training.

  The file data structure should look as follow:
  #under ‘data’ directory
  -dataset_1
  -annotations
  -training
  -apartment_000000.png
  …
  -validation
  -apartment_000000.png
  …
  -images
  -training
  -apartment_000000.png
  …
  -validation
  -apartment_000000.png
  …

  We can use the code in figure2 and figure 3 to create .odgt file.

```python
import os
import cv2
import json

def odgt(img_path):
    seg_path = img_path.replace('images','annotations')
    seg_path = seg_path.replace('.jpg','.png')

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

        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
```

Figure2. creat_odgt.py part1

```python
if __name__ == "__main__":

    modes = ['training','validation']
    saves = ['dataset_1_training.odgt', 'dataset_1_validation.odgt'] # customized

    for i, mode in enumerate(modes):
        save = saves[i]
        dir_path = f"data/dataset_1/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/{save}', mode='wt', encoding='utf-8') as myodgt:
            for i, img in enumerate(img_list):
                print(img)
                a_odgt = odgt(img)
                if a_odgt is not None:
                    myodgt.write(f'{json.dumps(a_odgt)}\n')
```

Figure3. creat_odgt.py part2

- Model Training

  Shown in figure4, we have to customize our own configuration file.

  We only need to change the “list_train” and “list_val” to match different dataset.

  I would like to explain the meaning of some parameters.

  - Since this work is using color101.mat, so the “num_class” is 101.
  - The image is 512X512, so we set the “imgMaxsize” as 525
  - After training many times, “epoch” set 20 can avoid overfitting and save training times.
  - The batch size for each GPU is 4. Since we have 2000 pictures, each “epoch_iters” should be 500.
  - We choose **“ade20k-hrnetv2-c1”** as our pretrain model

```yaml
DATASET:
  root_dataset: ""
  list_train: ./data/dataset_1_training.odgt
  list_val: ./data/dataset_1_validation.odgt
  imgMaxSize: 525
  imgSizes: (300, 375, 450, 525, 600)
  num_class: 101
  padding_constant: 32
  random_flip: True
  segm_downsampling_rate: 4
DIR: ckpt2/model_dataset1
MODEL:
  arch_decoder: c1
  arch_encoder: hrnetv2
  fc_dim: 720

TEST:
  batch_size: 1
  checkpoint: epoch_20.pth
  result: ./
TRAIN:
  batch_size_per_gpu: 4
  num_epoch: 20
  start_epoch: 0
  epoch_iters: 400
  optim: SGD
  lr_decoder: 0.02
  lr_encoder: 0.02
  lr_pow: 0.9
  beta1: 0.9
  weight_decay: 0.0001
  deep_sup_scale: 0.4
  disp_iter: 20
  fix_bn: false
  seed: 304
  workers: 16
VAL:
  batch_size: 1
  checkpoint: epoch_20.pth
  visualize: True
```

Figure4. .yaml

Then, we do to the **training** part. We have to train two datasets.
Environment and Hardware Device

- python 3.8.10
- Pytorch 2.0.0 +cu117
- CPU AMD Ryzen Threadipper 2950 X 16 core
- GPU two NVDIA RTX 2080Ti

- Evaluate mIOU and accuracy
  Shown in figure5 , this code snippet is used to compute and print the class-wise Intersection over Union (IoU) metrics for a semantic segmentation task and then calculate and print the mean IoU along with accuracy as an evaluation summary.

```
# specify
for i, _iou in enumerate(iou):
    print('class [{}], IoU: {:.4f}'.format(i, _iou))

# mean iou
new_iou = []
wb = openpyxl.load_workbook('apartment0_classes.xlsx', data_only=True)
s1 = wb['Sheet1']
for i in range(2,51):
    label = s1.cell(i,1).value
    print("add  " + str(int(label)) + "  " +  str(iou[int(label)]))
    new_iou.append(iou[int(label)])
wb.save('apartment0_classes.xlsx')

iou_count = 0
iou_sum = 0
for iou in new_iou:
    if iou != 0:
        iou_count += 1
        iou_sum += iou

print('[Eval Summary]:')
print('Mean IoU: {:.4f}, Accuracy: {:.2f}%'
        .format(iou_sum/iou_count, acc_meter.average()*100))

print('Evaluation Done!')
```

Figure5. eval_multipro.py

b. Result and Discussion
   i. Result
      - Dataset1 / floor1

```
add  80  0.0
add  84  0.0
add  91  0.05114623133032258
add  92  0.487708075982286
add  93  0.04784920510562132
add  94  0.013670431753404341
add  95  0.0
add  97  0.25856231609144015
add  98  0.0
[Eval Summary]:
Mean IoU: 0.2159, Accuracy: 62.16%
Evaluation Done!
100%|
```

Figure6. IoU and Accuracy

rgb ground truth prediction

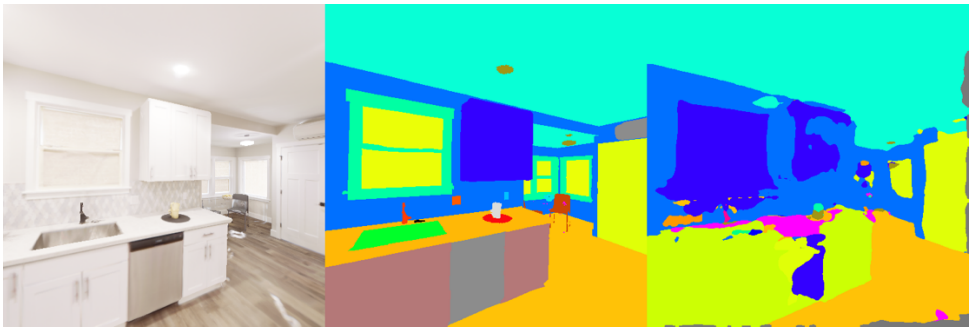

Figure7. Outputs of Segmentation Model

■ Dataset1 / floor2

```
add  76  0.0
add  77  0.0
add  78  0.0003656739088853132
add  79  0.005118120841526815
add  80  0.0
add  84  0.0
add  91  0.625715131189583
add  92  0.5431247089122578
add  93  0.0
add  94  0.030654115375331702
add  95  0.0
add  97  0.0
add  98  0.0
[Eval Summary]:
Mean IoU: 0.3699, Accuracy: 63.94%
Evaluation Done!
100%|
```

Figure8. IoU and Accuracy

rgb ground truth prediction

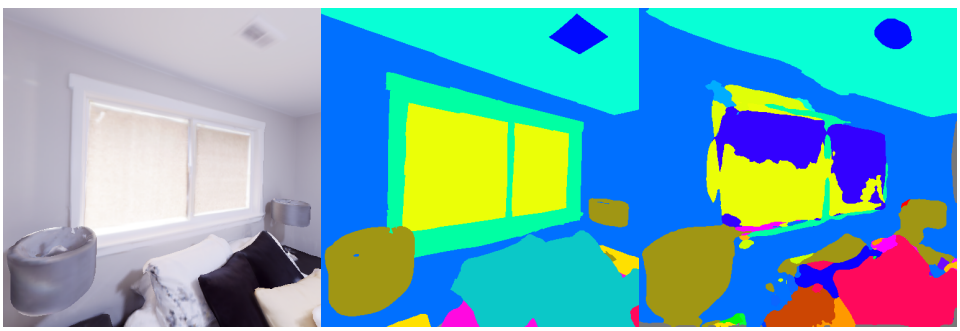

Figure9. Outputs of Segmentation Model

■ Dataset2 / floor1

```
add  74  0.0
add  76  0.10930865541643865
add  77  0.7687048251611621
add  78  0.5780626538755363
add  79  0.47237145889639
add  80  0.0
add  84  0.0
add  91  0.268256333830091
add  92  0.7802775527605342
add  93  0.7239207225643156
add  94  0.3796794105933872
add  95  0.0
add  97  0.7771546834030831
add  98  0.0
[Eval Summary]:
Mean IoU: 0.6053, Accuracy: 90.75%
Evaluation Done!
100%|
```

Figure10. IoU and Accuracy

rgb ground truth prediction

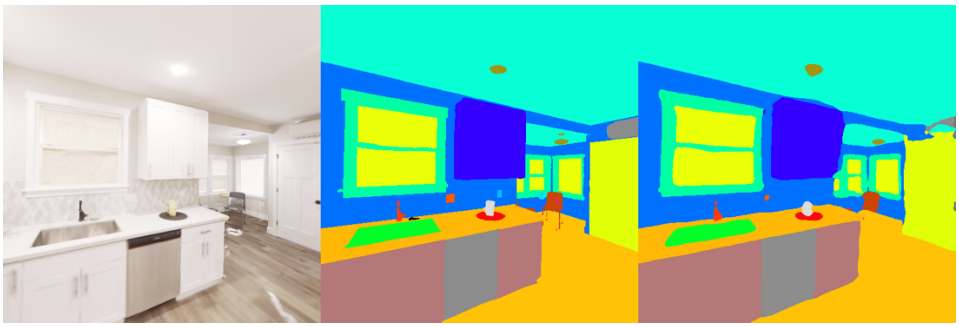

Figure11. Outputs of Segmentation Model

■ Dataset2 / floor2

```
add  78  0.5217041800643076
add  79  0.09527064970578876
add  80  0.0
add  84  0.0
add  91  0.6945786849005183
add  92  0.8632030345072803
add  93  0.0
add  94  0.5772490221642733
add  95  0.0
add  97  0.0
add  98  0.0
[Eval Summary]:
Mean IoU: 0.6500, Accuracy: 92.95%
Evaluation Done!
100%|
matt@matt:~/hw2$
```

Figure12. IoU and Accuracy

rgb ground truth prediction

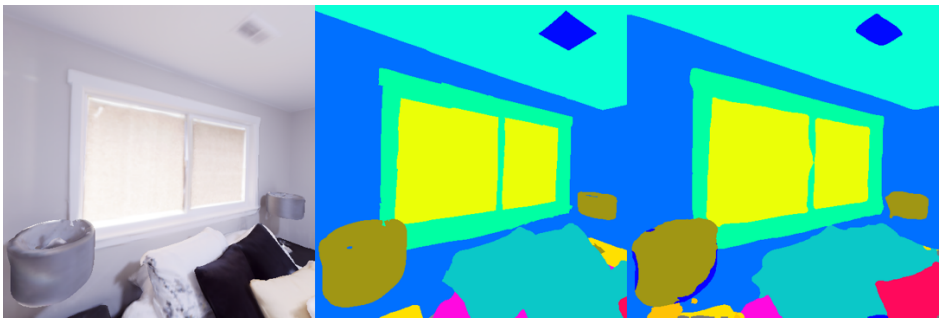

Figure13. Outputs of Segmentation Model

Table2. Iou and Accuracy

	Floor1	Floor2
Dataset1 Mean Iou	0.2159	0.3699
Dataset1 Accuracy	62.16%	63.94%
Dataset2 Mean Iou	0.6035	0.65
Dataset2 Accuracy	90.75%	92.95%

ii. Discussion

From Table2 and the outputs of each segmentation model. We can obviously observe that the accuracy of dataset2 is much higher than dataset1.

The reason for this is that we initially trained and tested our models using the apartment0 dataset. As a result, the model trained on dataset2 became highly specialized for the apartment environment and struggled to adapt to new data outside of it. On the other hand, the model trained on dataset1, despite not performing well on floor1 and floor2, had the advantage of being exposed to a wider range of data. When we tested both models in a non-apartment0 environment, the dataset1 model performed better than the dataset2 model.

iii. References

https://github.com/CSAILVision/semantic-segmentation-pytorch
https://github.com/open-mmlab/mmsegmentation
https://hackmd.io/wNGlmMq2RC-lY3l8JhO4SA?view
https://stackoverflow.com/questions/62461379/multiclass-semantic-segmentation-model-evaluation

Task 2:

a. Code

- **custom_voxel_down**

In this task, if we average all points into one point in each voxel, it will influence the performance of the 3D semantic map because there are two or more labels within a voxel. To address the issue, we need to implement a **custom_voxel_down()** function to determine a label of each voxel.

- Convert Point Cloud to NumPy Arrays:

  The function first converts the point cloud pcd into NumPy

arrays. It extracts the 3D point coordinates (xyz_points) and colors (xyz_colors) from the Open3D point cloud object.

- Initialize Voxel Dictionaries:
  It creates two empty dictionaries, voxel_point_cloud and voxel_colors, to hold the voxelized points and colors
- Calculate Voxel Indices:
  The voxel indices for each point are calculated based on the point's 3D coordinates. This calculation involves dividing the xyz points by the specified voxel_size and shifting them so that they are relative to the minimum xyz coordinates.
- Voxelization with Dictionaries:

The function then iterates through the points and populates the voxel_point_cloud and voxel_colors dictionaries. It groups points into the voxel they belong to based on their calculated voxel indices. If the voxel index is not in the dictionary, a new entry is created.

- Create Lists for Downsampled Points and Colors:

After populating the dictionaries, the function initializes lists to store the downsampled points (downsampled_points) and their associated colors (downsampled_colors).

- Compute Downsampled Points and Colors:

The function iterates over the voxelized points stored in the dictionaries. For each voxel, it calculates the most frequent color (major_color) among the points in the voxel and computes the mean of the voxel's points to obtain the downsampled point.

- Create a Downsampled Point Cloud:

Finally, the downsampled points and colors are used to create a new Open3D point cloud object (downsampled_pcd). This new point cloud represents the downsampled version of the input point cloud.

- Return the Downsampled Point Cloud:
  The downsampled point cloud (downsampled_pcd) is returned as the output of the function.

```python
def custom_voxel_down(pcd, voxel_size):
    xyz_points = np.asarray(pcd.points)
    xyz_colors = np.asarray(pcd.colors)

    # Create an empty dictionary to hold voxelized points
    voxel_point_cloud = {}
    voxel_colors = {}

    # Calculate voxel indices for each point
    voxel_indices = ((xyz_points / voxel_size) - (xyz_points.min(0) / voxel_size)).astype(int)

    # Iterate over points and populate the voxel dictionaries
    for i, voxel_index in enumerate(voxel_indices):
        voxel_index = tuple(voxel_index)
        if voxel_index not in voxel_point_cloud:
            voxel_point_cloud[voxel_index] = []
            voxel_colors[voxel_index] = []
        voxel_point_cloud[voxel_in(function) append: Any)
        voxel_colors[voxel_index].append(tuple(xyz_colors[i]))

    # Create lists for downsampled points and colors
    downsampled_points = []
    downsampled_colors = []

    # Iterate over the voxelized points to compute downsampled points
    for voxel_index, points in voxel_point_cloud.items():
        colors = voxel_colors[voxel_index]
        major_color = max(set(colors), key=colors.count)
        downsampled_points.append(np.mean(points, axis=0))
        downsampled_colors.append(np.array(major_color))

    downsampled_pcd = o3d.geometry.PointCloud()
    downsampled_pcd.points = o3d.utility.Vector3dVector(downsampled_points)
    downsampled_pcd.colors = o3d.utility.Vector3dVector(downsampled_colors)

    return downsampled_pcd
```

Figure14. Custom_voxel_down

```python
def preprocess_point_cloud(pcd, voxel_size, counter):
    #pcd_down = pcd.voxel_down_sample(voxel_size)
    pcd_down = custom_voxel_down(pcd,voxel_size)
    radius_normal = voxel_size * 2

    pcd_down.estimate_normals(
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))

    radius_feature = voxel_size * 5

    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd_down,
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100))

    print(f"Processed Point{counter+1} Done")
    return pcd_down, pcd_fpfh
```

Figure15. preprocess_point_cloud

- Reconstruct 3D semantic maps

Reconstruct 3D semantic maps of the first and second floor is the same as the function “reconstruct” in hw1. However in this task, we don’t need to calculate L2 , camera pose and ground truth.

b. Result and Discussion

i. Result

- First floor

Figure16. Semantic map floor1 ground truth

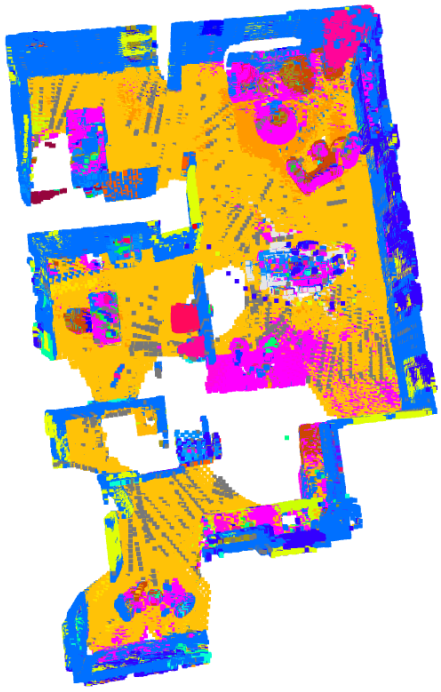

Figure17. Semantic map floor1 (trained on other scenes)

Figure18. Semantic map floor1 (trained on apartment_0)

- Second floor

Figure19. Semantic map floor2 ground truth

Figure20. Semantic map floor2 (trained on other scenes)

Figure21. Semantic map floor2 (trained on apartment_0)

ii. Discussion

When creating maps, we can observe that maps built under the "apartment0" training scenario outperform maps created for other scenes. The reason for this superiority is consistent with what was

discussed in Task 1.
Furthermore, the construction of 'floor2' reveals a substantial influence from different datasets on the results. This may be attributed to limitations in the reconstruction algorithm. Fortunately, after several rounds of data collection, we managed to successfully build the map.

iii. Reference
https://github.com/CSAILVision/semantic-segmentation-pytorch

2. Questions

1. From the results, we can clearly see the impact of domain shift. In your opinion, besides collecting training data directly under the same scene, are there any other methods to improve performance?

   I have survey several methods that can be employed to improve model performance:

1. **Domain Adaptation Techniques:** Domain adaptation methods aim to reduce the distribution discrepancy between the source domain (training data) and the target domain (data you want to make predictions on). Common techniques include Domain Adversarial Neural Networks (DANN), Maximum Mean Discrepancy (MMD) minimization, and more.
2. **Data Augmentation:** Augmenting your dataset with variations of the existing data can make your model more robust to domain shifts. Techniques such as rotation, translation, scaling, and adding noise can help.
3. **Transfer Learning:** Pre-trained models on a large and diverse dataset can be fine-tuned on your specific dataset. This leverages the knowledge acquired during pre-training and can be especially effective when you have limited data.
4. **Ensemble Learning:** Combining the predictions of multiple models trained on different domains or data subsets can improve overall performance. Ensemble techniques like bagging and boosting are commonly used.
5. **Domain-Specific Regularization:** Adding regularization terms to the loss function that encourage the model to be less sensitive to domain-specific features can be helpful.

6. **Collecting More Diverse Data:** Expanding your dataset to be more representative of the target domain can mitigate domain shift. It's essential to ensure that your training data covers various scenarios and conditions.
7. **Fine-Tuning Hyperparameters:** Carefully fine-tuning hyperparameters, such as learning rates, batch sizes, and optimizer choices, can lead to better adaptation to domain shifts.
8. **Domain-Invariant Features:** Learning features that are domain-invariant can help the model focus on domain-independent patterns. Techniques like adversarial feature learning can be useful for this purpose.
9. **Robust Loss Functions:** Using loss functions that are less sensitive to outliers or domain-specific variations can lead to more stable training.
10. **Regular Monitoring and Retraining:** Continuously monitoring model performance on the target domain and periodically retraining the model with the latest data can help adapt to evolving domain shifts.

2. List and explain all the tricks you have tried to speed up the voxelization, and compare the results.

Following points is the tricks I have used in this task.

Shown in Table3 , we can compare the results. The code is also show in Figure22.

1. **Numpy**

   - The calculation of voxel indices and the manipulation of these indices are done in a vectorized manner using NumPy. This is much faster than using Python loops for each individual point.
   - The computation of the downsampled points and colors is also performed efficiently by taking advantage of NumPy's array operations

2. **Data Structures for Voxelization:**
   - Use dictionaries (voxel_point_cloud and voxel_colors) to organize the voxelized points and their colors.
   - Dictionaries allows efficiently group points by their voxel indices and store associated colors. This organization is essential for voxel downsampling.
3. **Voxelization Algorithm**:

- Calculate voxel indices for each point to determine their membership.
- The voxel indices are determined based on the division of each point's coordinates by the voxel_size. This step is the core of voxelization, as it maps points to specific voxels in 3D space.

4. **Downsampling with Majority Color:**
   - Downsample the voxelized points by selecting the majority color within each voxel.
   - By choosing the most frequent color within each voxel, ensure that the downsampled point cloud retains representative colors, which can be useful for visualization or other downstream tasks.

Shown in table3. We can compare the results. The time is the whole process of preprocess (See figure 14 and figure15)

```python
def custom_voxel_down(pcd, voxel_size):
    points = pcd.points
    colors = pcd.colors

    # Calculate voxel indices for each point
    voxel_indices = []
    for point in points:
        voxel_index = (
            int((point[0] / voxel_size) - (points[0][0] / voxel_size)),
            int((point[1] / voxel_size) - (points[0][1] / voxel_size)),
            int((point[2] / voxel_size) - (points[0][2] / voxel_size))
        )
        voxel_indices.append(voxel_index)

    # Create lists for downsampled points and colors
    downsampled_points = []
    downsampled_colors = []

    previous_voxel_index = None

    for i in range(len(points)):
        current_voxel_index = voxel_indices[i]

        if i == 0 or current_voxel_index != previous_voxel_index:
            # This is a new voxel, add it to the downsampled point cloud
            downsampled_points.append(points[i])
            downsampled_colors.append(colors[i])
        else:
            # This point belongs to the same voxel as the previous one,
            # so update the downsampled point with the mean of all points in the voxel
            downsampled_points[-1] = [(x + y) / 2 for x, y in zip(downsampled_points[-1], points[i])]
            downsampled_colors[-1] = [(x + y) / 2 for x, y in zip(downsampled_colors[-1], colors[i])]

        previous_voxel_index = current_voxel_index

    downsampled_pcd = o3d.geometry.PointCloud()
    downsampled_pcd.points = o3d.utility.Vector3dVector(downsampled_points)
    downsampled_pcd.colors = o3d.utility.Vector3dVector(downsampled_colors)

    return downsampled_pcd
```

Figure22. voxelization with no. speed up

Table3. compare result

	With Speed	Without Speed
Apartment0 Floor1	93.70 sec	302.9 sec
Apartment0 Floor2	112.32 sec	370.82 sec
Others floor1	95.72 sec	307.16 sec
Others floor2	115.39 sec	380.23 sec

The link of shared ckpt:
https://drive.google.com/drive/folders/1LOlHpI4yn2hna21tYqCaTEJ4wOKhp440?usp=share_link