Homework 3

Instructions

This homework reviews the basics of CNNs, segmentation, and pose estimation. By the end of this homework, you will learn how to set up a simple training framework for image segmentation, and how to use the segmentation with the ICP algorithm to estimate object poses. The pose estimated for each object can be used for grasp planning in later homework.

For Problem 1, compile your answers in a PDF and title it `hw3.pdf`. For Problem 2, you will directly edit the Python files provided by us and provide some commentary in `hw3.pdf`. A detailed submission checklist can be found at the end of the document.

Submission: Compress the hw3.pdf, your code, and other submission checklist items into `SUNetID_hw3.zip`, and upload it to Canvas before the homework deadline.

Problem 1 (10 pts):

1.1 Convolution – 10 points

Consider a 4×4 grayscale image represented by the following matrix:

$$\begin{bmatrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 2 & 3 \\ 3 & 0 & 2 & 1 \\ 1 & 2 & 0 & 3 \end{bmatrix}$$

Define a 2×2 kernel matrix for a convolution operation:

$$\begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix}$$

Perform a convolution operation on the given image using this kernel (stride=1, padding=0). Show the resulting feature map (output matrix).

Problem 2: Coding Assignments

Setup: Same as previous homework, install the relevant packages and start the environment. For this homework, we highly recommend Windows users run code with Windows Subsystem for Linux (WSL). A tutorial for setting up WSL is here.

If you have an Apple computer with Apple Silicon processor and want to use M chip GPU acceleration:

```
mamba env create -f environment_apple.yaml
```

If your computer has an NVIDIA GPU:

```
mamba env create -f environment_gpu.yaml
```

Else:

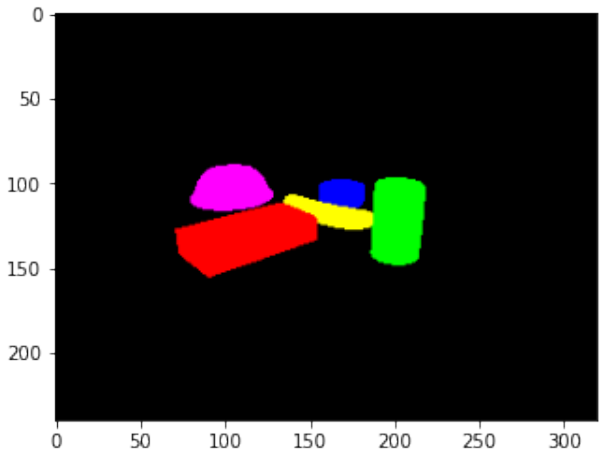
```
mamba env create -f environment_cpu.yaml
```

In the data provided, you can find the data in the following folder structure:

```
hw3/dataset
|-- train
| |# 30 training scenes * 10 samples each.
| |# Each sample consists of an RGB image and a ground truth mask,
| |# saved separately in /rgb and /gt folder.
| |# The naming rule is 'sampleID_suffix.png',
| |# where the correspondence between RGB images and masks
| |# is marked by the sampleID.
| |
| |-- rgb
| |-- gt
|
|-- val
| |# 5 provided validation scenes.
| |# Images in /rgb, /gt and /depth follow the same naming rule
| |# as in /train.
| |
| |-- rgb
| |-- gt
| |-- depth
| |-- gt_pose # the ground truth pose matrices of each object in
| | each scene, named 'sceneID_objectID.npy'
| |-- view_matrix # the view matrices of each scene,
| | named 'sceneID.npy'
|
|-- test
  |# 5 provided test scenes.
```

```
|# Similar to /val but without ground truth.
|
|-- rgb
|-- depth
|-- view_matrix
```

The images inside `/gt` folders can be read using `read_mask()` from `image.py` and visualized using `show_mask()` from `segmentation_helper.py` as below.



In the mask, each pixel value is an integer standing for the ID of the object (also called the class label). In this homework, we have five objects and one background class. The visualization method maps each ID to a unique color:

ID	name	color
0	background	black
1	sugar box	red
2	tomato soup can	green
3	tuna fish can	blue
4	banana	yellow
5	bowl	purple

2.1 CNN for Segmentation *(40 points)*

We are going to use the dataset to train a neural network for instance segmentation. Depending on your device, training on the local computer without GPU can take about 30 minutes to 1 hour. If you take advantage of a GPU, the runtime can be reduced to 2-5 minutes. You are welcome to try Google Collab to access a GPU, but we have found that it's challenging to get the dependencies setup correctly and difficult to directly use as our assignment is not formatted as a notebook. Therefore, we suggest running everything locally. Please implement the following methods:

```
dataset.py
    class RGBDataset() -- 6 points
          __getitem__
model.py
    class MiniUNet() -- 14 points
          __init__()
          forward()
Written question -- 10 points (see 2.1.3)
Train and validate the model -- 10 points
```

You need to explore the PyTorch neural network package to complete this part. If you are new to PyTorch, we suggest you go through these tutorials before you start: Data Loading, Neural Network, Training a Classifier

2.1.1 Data loader

Before feeding the images into the model, we need to build a `Dataset` to serve as a mapping from integer indices to data samples, and a `DataLoader` to iterate over the dataset, batch the data, and shuffle the train set. Don't shuffle the validation set and test set.
In `dataset.py`, fill out the `RGBDataset` class to load and pair the input and target (desired output) i.e. the RGB image and its corresponding ground truth mask (if exists) as a sample.

You may use the two methods implemented in `segmentation_helper.py`, which are imported in `segmentation.py`, for sanity checks:

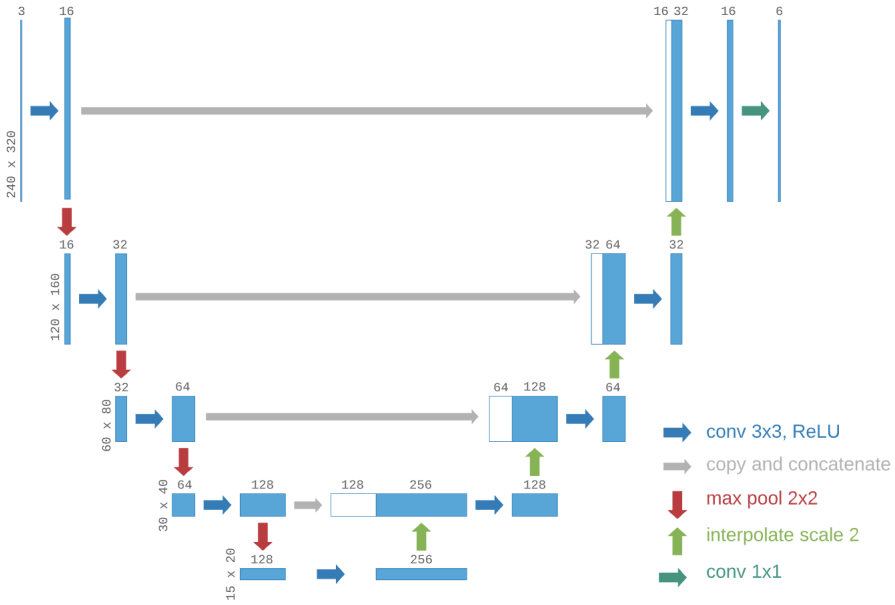
`check_dataset()` will visualize a random sample

`check_dataloader()` will visualize a batch

2.1.2 Construct the network

U-Net architecture is a popular architecture for image segmentation. As the figure shows, it looks like a U! The architecture has 3 highlights:

> It extracts features and down-samples them to go from the input resolution to a representation space that is information rich.
>
> It up-samples this compressed feature back to the resolution of the input to make pixel-wise predictions.
>
> It has skip connections, which allows information to flow from the down-sampling layers to the up-sampling layers directly.



Now you get to implement your own version (following the exact same architecture as in the diagram above)! In `model.py`, implement the `MiniUNet` class to create a simplified U-Net:

(1) The contracting path (left side) extracts a compact representation of the image. Use 3x3 convolutions followed by an activation function (ReLU) and down-sample using 2x2 max pooling to reduce the resolution of the feature maps by a factor of 2.

(2) The expansive path (right side) recovers the extracted image feature back to the original resolution. Up-sample by a scale of 2 using interpolation, concatenate

the up-sampled feature with its corresponding down-sampled feature of the same resolution, then apply 3x3 convolution + ReLU.

(3) Finally, apply an 1x1 convolution to get to the desired number of output channels.

Note: The input and output channel numbers of convolutional layers are shown in the figure. For 3x3 convolutions, pad the image with a 1-pixel border so that the resolution won't change.

After you implement the network, run this command as a sanity check:

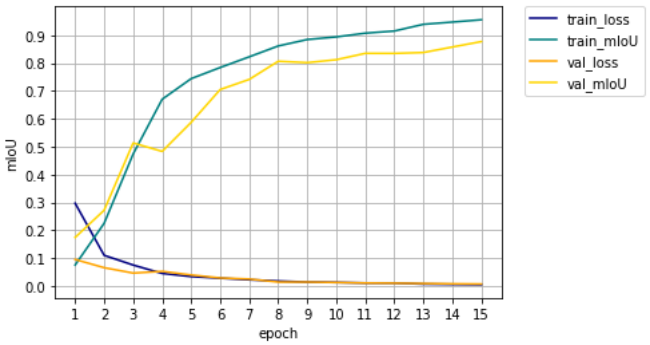
```
python model.py
```

2.1.3 Written question

1) We know the input has 3 channels because the network takes in RGB images. But why does the output have 6 channels? What does each channel stand for? (< 50 words) 2) Which functions do you need to change and how to make this network perform pixel-wise depth estimation? (< 50 words)

2.1.4 Train and validate the model

In `segmentation.py`, take a look at the train-validate loop to see what's happening there. The evaluation metric for this task is mean Intersection over Union (mIoU). An `iou()` method to compute mIoU on each sample is provided. A learning curve plot recording loss and mIoU on training and validation set in each epoch will be saved as `hw3/learning_curve.png`. **Remember to include it in `hw3_report.pdf`.** Here is an example:



To achieve reasonable prediction, you should train a model to reach about 85+ percent mIoU on the validation set. We have implemented code to save and load the trained model with the highest mIoU using `save_chkpt()` and `load_chkpt()` and to save the predicted masks of validation and test set as `sceneID_pred.png` (and the RGB visualizations as `sceneID_pred_rgb.png`) as well as visualize them in the GUI using `save_prediction()`. Predicted masks will be used in the next problem.

You can train your network and generate the output training curve and prediction masks by running:

```
python segmentation.py
```

Since training the network takes some time, we save a model checkpoint each epoch in which the performance of the model on the validation set exceeds the best performance so far. If you need to stop training you can quit the `segmentation.py` script and then later resume from a saved checkpoint at the epoch you left off by running:

```
python segmentation.py --checkpoint checkpoint.pth.tar
```

During training, we recommend monitoring the `learning_curve.png` that is logged. If your learning curve is not roughly similar to the learning curve shown above, we recommend cancelling training as it's likely that your code has a bug as we do not expect you to need to tune any of the learning parameters like learning rate.

If you complete this part on the cloud (e.g., Google Collab), download the files listed below and put them to the same directory on your local machine to continue:

completed scripts

the best `checkpoint.pth.tar` and corresponding `learning_curve.png` in `/hw3/`

predicted masks in `/dataset/val/pred/` and `/dataset/test/pred/`

2.2 Pose Estimation *(50 points)*

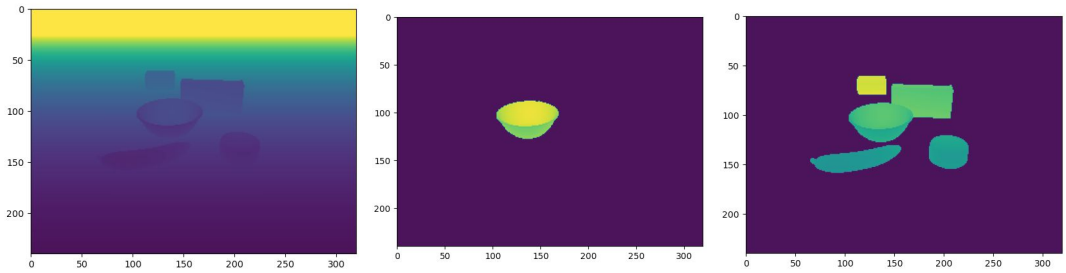
Here is the list of functions you will implement in problem 2.2.

```
icp.py
    gen_obj_depth() -- 10 points
    obj_depth2pts() -- 10 points
    align_pts() -- 10 points
    estimate_pose() -- 10 points
result -- 10 points
```

3D object pose can be described using a SE(3) transformation with respect to a predefined reference frame of the object. Iterative Closest Point (ICP) is an algorithm that takes two point clouds as input and outputs an estimated transformation to align them. The algorithm iteratively (1) associates points in one point cloud with those in another using some notion of closeness, (2) estimates a transformation to further align point clouds by minimizing some objective (for example, sum of squared distances), (3) applies the transform. In this problem, we will try to estimate the pose of the objects in the validation and test set. The high level idea is to take points sampled from a known object mesh and align them with the segmented point cloud from our scene (the segmentation model we just trained tells us which regions of the depth image contain the object of interest). This will allow us to figure out the pose of the object in the scene and, in future homework assignments, allow us to figure out where we should place our gripper in order to grasp the object.

2.2.1 Prepare point clouds: `gen_obj_depth()`, `obj_depth2pts()`

Implement the `gen_obj_depth()` method, using the instance segmentation mask (ground truth mask for now) to create a depth image that only contains pixels of the specific object(s). The Figure below shows the original depth image, object depth image (obj_id =4), and foreground object depth (obj_id =-1)



Then implement `obj_depth2pts()` to create the point cloud projected from the depth image. You can use `depth_to_point_cloud()` from `transforms.py`. Note that this

method returns coordinates in the camera reference frame, so don't forget to convert to the world reference frame using camera pose corresponding to this scene.

The view matrices are provided in the `val/view_matrix` and `test/view_matrix` folder as `sceneID.npy` and have already been loaded for you. The method `cam_view2pose()` from `camera.py` is provided to convert the camera view matrix to the pose matrix.

2.2.2 Iterative Closest Point (ICP): `align_pts()`

(1) **Implement `align_pts()` method, using the ICP method in trimesh to align two point clouds.** We assume we are given a 3D mesh of the desired object. We first randomly sample a number of points on this mesh equal to the number of points in our segmented point cloud. Next, our goal is to figure out the transform that aligns these sampled mesh points with the segmented point cloud points. Be careful: we want the output to be the transformation sending the sampled point cloud from the object mesh (`pts_a`) to the projected one computed from the scene (`pts_b`).

When calling the `icp` method, make sure to pass `reflection=False, translation=True, scale=False`. These are part of the `kwargs` in the `icp` documentation that are internally passed to calls that `icp` makes to `procrustes`. The idea is that we know the object scale from the object mesh, so we don't want ICP to try and optimize the object scale.

(2) **Tune the parameters of the ICP method to see how the parameters affect the result.** There are three tunable parameters:

minimum change in cost (threshold)

maximum number of iterations

initial transformation

You can see the differences from tuning these parameters by running the `icp.py` script in the next subpart and looking at the qualitative results from visualizing the .ply files. Intuitively, the lower the threshold and the more iterations to run, the better the alignment. But this can result in wasting computation on very tiny or even no improvement. You can experiment with the parameters and try to improve the result on harder cases. **We don't require you to submit any written response for this subpart; this is just for your understanding and to improve the qualitative results of the images you submit.**

To compute the initial transformation, we use Procrustes' analysis. To use it, you will need to sample the same number of points from the .obj file as the projected point cloud and set the parameters `reflection=False, scale=False`.

2.2.3 Object pose estimation: `estimate_pose()`

Finally, let's wrap up what we have done and collect the result! Complete the `estimate_pose()` method that takes depth and mask images of a scene as input to perform pose estimation on each object. For the validation set, use both ground truth and predicted masks. For the test set, use predicted masks. Read `main()` to see how this function is called.

2.2.4 Qualitative result:

After implementing all functions, we can generate the result by running:

```
python icp.py --val
```

```
python icp.py --test
```

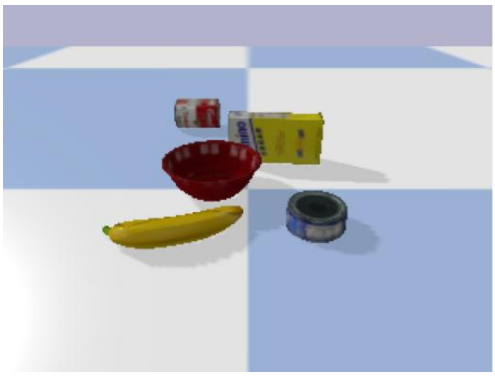
The provided `export_gt_ply()` and `export_pred_ply()` are used to export the point cloud of a scene as `sceneID_suffix.ply`. These files will be saved in `/dataset/val/exported_ply/` and `/dataset/test/exported_ply/`.

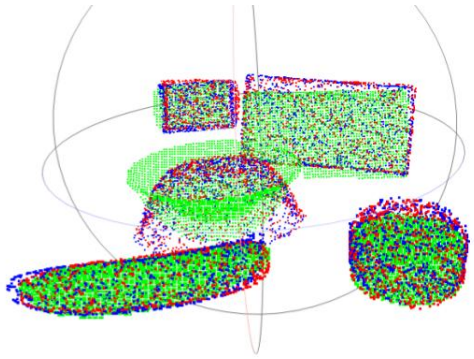
suffix	**point cloud**	**color**	**dataset**
gtmask	scene points projected using gt mask containing partial object point cloud from depth camera (what ICP is attempting to align to)	green	val
gtmask_transformed	estimated using ground truth mask (post ICP)	blue	val
predmask_transformed	estimated using predicted mask (post ICP)	red	val&test

You can then drag the .ply files into Meshlab to visualize and self-evaluate: first make sure the green point cloud correctly matches the RGB image of the scene, then compare the blue and red point cloud to the green one, as shown in the following image. Ideally, the position and orientation of the same object will be the same. It's OK to have errors on hard/occluded objects. Just try your best! For example, the pose of the bowl is upside down in the result below.

2.2.5 Quantitative result:

The Python commands run in the previous part use the provided `save_pose()` method to save the pose of each object in each scene as `sceneID_objectID.npy`. These files will be saved in `/dataset/val/pred_pose/` and `/dataset/test/pred_pose/`.





For the validation set, you can quantitatively compare the error between your predicted poses (for both ground truth and predicted masks) with the ground truth pose. Note that ground truth pose is only available in the validation set. To evaluate on the validation set you can run:

```
python evaluate_icp.py --gtmask --predmask
```

The evaluation metric is the Average Closest Point Distance, smaller is better. It's invariant under pose ambiguity caused by symmetries. For reasonable pose estimation results, the output should be on the order of 1e-4 but may vary depending on the object and the scene. We expect higher error for predicted masks compared to when we use the ground truth masks.

You do not need to report any of these values in your writeup. Looking at the qualitative results from the prior part will likely give you a better intuition about the quality of your results compared to these quantitative metrics.

Submission checklist

When you are done, prepare your hw3 folder for submission by making sure it contains the following (it's ok if it also includes other helper files, but it is not necessary):

1. `hw3_report.pdf`: that include:
   1) Written answer to Problem 1
   2) Answer to the written question in Problem 2.1.3
   3) `learning_curve.png`
   4) screenshot of the qualitative point cloud visualization result of the first validation scene with all three ply visualized in the same viewer (like in the example point cloud image in 2.2.4) using files from: `/dataset/val/exported_ply/0_*`

2. completed python code: `dataset.py, model.py, icp.py`
3. `checkpoint.pth.tar` of your best model
4. predicted masks as SCENEID_pred.png files in `/dataset/test/pred/`, 5 files
5. point clouds of the scene as .ply files in `/dataset/test/exported_ply/`, 5 scenes * 1 file each
6. estimated poses .npy files in

   `/dataset/val/pred_pose/gtmask/`, 5 scenes * 5 files each

   `/dataset/val/pred_pose/predmask/`, 5 scenes * <= 5 files each

   `/dataset/test/pred_pose/predmask/`, 5 scenes * <= 5 files each

Please rename your folder as SUNetID_hw3. Zip the folder into one single `SUNetID_hw3.zip` file and upload it to Canvas. We will review your report, evaluate your outputs against our grading scripts, and run your code!