Homework 2

Out: Monday, Feb 21

Due: Thursday, March 10, @ 5:00 pm EST

In this homework, you will learn how to train a neural network to perform segmentation using RGB images and corresponding ground truth instance segmentation masks obtained from a simulated camera. You will then use the model and RGB-D images to estimate the pose of objects.

This homework is one of the four assignments. It is worth 100 points.

There are requested written answers or images in Problem 2.1, 2.4, 2.5, and 3.2. For these problems, compile your answers in `hw2_report.pdf` using the LaTeX template provided on the course website. We recommend Overleaf for this purpose.

Output files are requested for Problem 2 and 3. You can find details in those sections.

For most of the problems, you will directly be editing python files provided by the TAs. **In `hw2_report.pdf`, for each method you implement, write a brief description of how it works, parameters you use and special features in your design (if any).** You can include more images of your result if you feel that would help to illustrate your work.
Please do not introduce any other dependencies/packages without taking prior permission from TAs.
You are free to change parameters in provided methods, define more methods and add more code files. But make sure we are able to run your code and generate and evaluate the output files you submit using these commands:

```
python gen_dataset.py
```

```
python segmentation.py
```

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

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

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

Setting up the environment

Students on Windows need to install visual-cpp-build-tools, and the command to install PyTorch might be different from what is in the requirements.txt.

Make sure your default python interpreter is python3. If not, use the command `python3` and `pip3` instead of `python` and `pip`. Dependencies for hw2 can be installed using the following commands where we first create a python virtual environment:

```
python -m venv venv_hw2
```

To activate your environment run:

```
source venv_hw2/bin/activate // Linux or OSX
venv_hw2\Scripts\activate.bat // Windows
```

To install dependencies (can take 5-20 min), once you have activated your environment, run:

```
pip install -r requirements.txt
```

To exit the environment, when you are not working on this project, run:

```
deactivate // Linux of OSX
venv_hw2\Scripts\deactivate.bat // Windows
```

You can read more about python virtual environments here.

Problem 1: Generate training set *(5 points)*

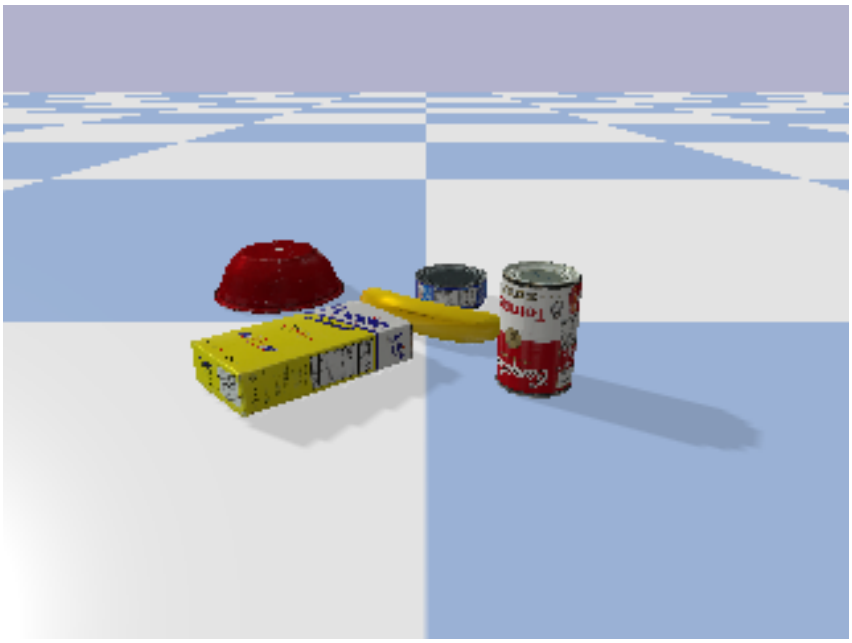
TODO list:

```
camera.py
    compute_camera_matrix() -- 5 points
```

The simulated camera is defined in `camera.py`. Take a look at the arguments for the `Camera` class. You will need them to **implement the `compute_camera_matrix()` method, which computes the intrinsic matrix and projection matrix.** For the projection matrix, use the `computeProjectionMatrixFOV()` method in PyBullet. You can check out OpenGL documentation for the meaning of the projection matrix and its parameters.
After you implement this method, under the directory of homework 2, execute the command:

```
python gen_dataset.py
```

Depending on your device, it will take 1-20 minutes to generate the training set. As the script is running, you will be able to see the scene we are going to deal with.

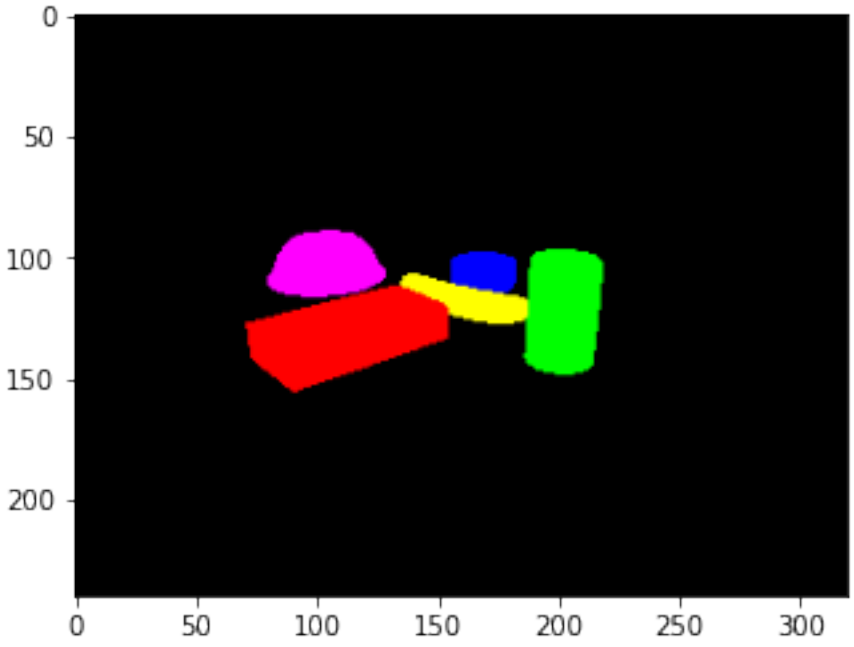


For each batch of rendering, five objects are dropped onto the floor from random starting positions. The camera moves in a circular orbit to make observations and will gradually move closer to the objects and look from a higher position to add further variations to the data.

After the generating procedure finished, the folder structure should be as follows:

```
hw2/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

Problem 2: CNN for Segmentation *(50 points)*

TODO list:

```
written question -- 5 points
dataset.py
    class RGBDataset() -- 10 points
model.py
    class MiniUNet() -- 20 points
segmentation.py
    train() -- 5 points
    val() -- 5 points
    main() -- 5 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

Now that we have our dataset, we are going to use it to train a neural network for instance segmentation. Depending on your device, training on the local computer can take about 30 minutes to hours. If you take advantage of GPU, the runtime can be reduced to 2-5 minutes on Google Colab or Google Cloud. See this for more detailed instructions.

1. Describe the instance segmentation problem. What are the inputs and outputs of the task? What supervision do we need? *(1 point)*

2. Data loader *(10 points)*

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.
Then, in `main()` of `segmentation.py`, create RGBDataset instances and

DataLoaders for train, validation, and test dataset respectively.
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

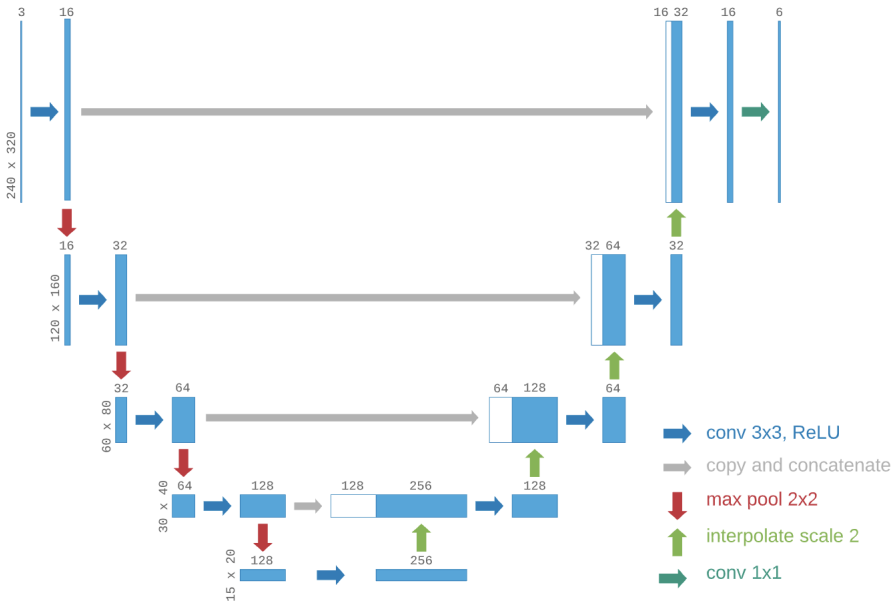
3. Construct the network *(20 points)*

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! **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
```

Then, in `main()` of `segmentation.py`, create a MiniUNet instance.

4. Rethink the network before you start training it *(4 points)*

We've explained the purpose of most of the modules, like pooling for down-sampling, interpolate for up-sampling, and concatenation for skip connections. Now is your turn to think about:

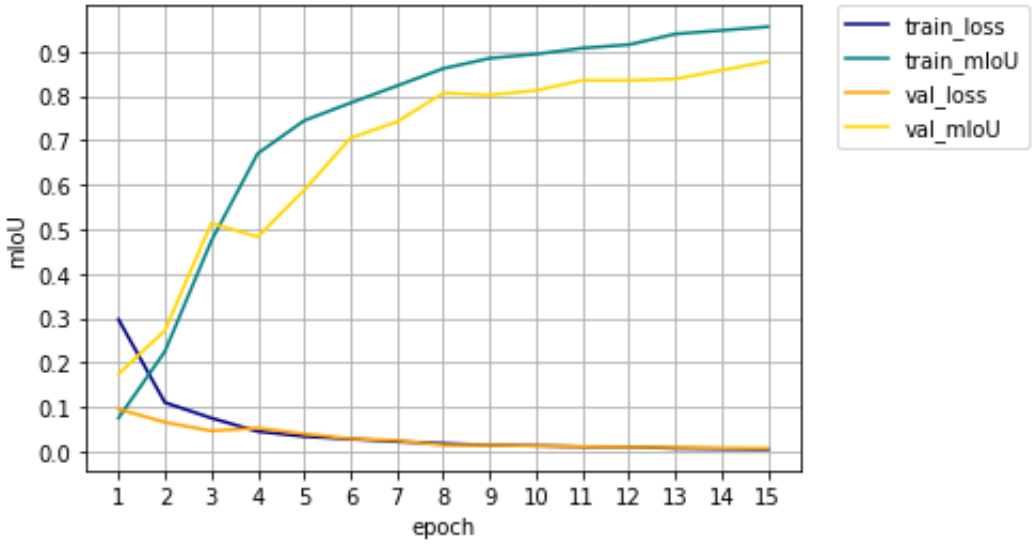
(1) Why are 3x3 kernels used in most of the convolutional layers, but 1x1 kernels are used right before the output? *(1 point)*

(2) 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? How to compare it with the 1 channel ground truth mask? (Hint: Think about the information stored in the mask, and the classification task introduced in the lecture.) *(3 points)*

5. Train and validate the model *(15 points)*

In `segmentation.py`, implement the `train()` and `val()` methods. Then, define criterion and optimizer. Finally, take a look at the train-validate loop to see what's happening there. You might want to make changes like to continue training a loaded checkpoint.

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 `/hw2/learning_curve.png`. **Remember to include it in `hw2_report.pdf`.** Here is an example:




```
# pseudo-code of train()
# Pass all the samples in the training set through the model once.
# For each batch, update the parameters of the model.
for batch in dataloader:
    feed a batch of input into the model to get the output
    compute average loss of the batch using criterion()
    compute mIoU of the batch using iou()
    store total loss and mIoU of the batch for computing statistics
    zero the parameter gradients using optimizer.zero_grad()
    compute the gradients using loss.backward()
    updates the parameters of the model using optimizer.step()
compute the average loss and mIoU of the dataset to return
```

To achieve reasonable prediction, you would better train a model to reach about 90 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 save the predicted masks of validation and test set as `sceneID_pred.png` as well as visualize them using `save_prediction()`. Predicted masks will be used in the next problem. **If you complete this part on the cloud, 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 `/hw2`

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

Make sure the requested files can be generated by running:

```
python segmentation.py
```

Problem 3: Pose Estimation *(45 points)*

TODO list:

```
written question -- 5 points
icp.py
    gen_obj_depth() -- 4 points
    obj_depth2pts() -- 6 points
    align_pts() -- 5 points
    estimate_pose() -- 5 points
    main() -- 5 points
result -- 15 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.

1. Prepare point clouds *(10 points)*

For an object, we need to prepare two point clouds: one is down-sampled from the .obj file of the object, the other is projected from the depth image of the object.
In `icp.py`, the method `obj_mesh2pts()` is provided to create the first point cloud. **For the second point cloud, 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). 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 `/dataset/val/view_matrix` and `/dataset/test/view_matrix` folder as `sceneID.npy` and can be loaded using `np.load()`. The method `cam_view2pose()` from `camera.py` is provided to convert the camera view matrix to the pose matrix.

2. Iterative Closest Point (ICP) *(10 points)*

(1) **Implement `align_pts()` method, using ICP method in trimesh to align two point clouds.** Be careful: we want the output to be the transformation sending the sampled one (first) to the projected one (second).
(2) **Tune the parameters of the ICP method and describe what difference you observe.** There are three tunable parameters:

minimum change in cost (threshold)

maximum number of iterations

initial transformation

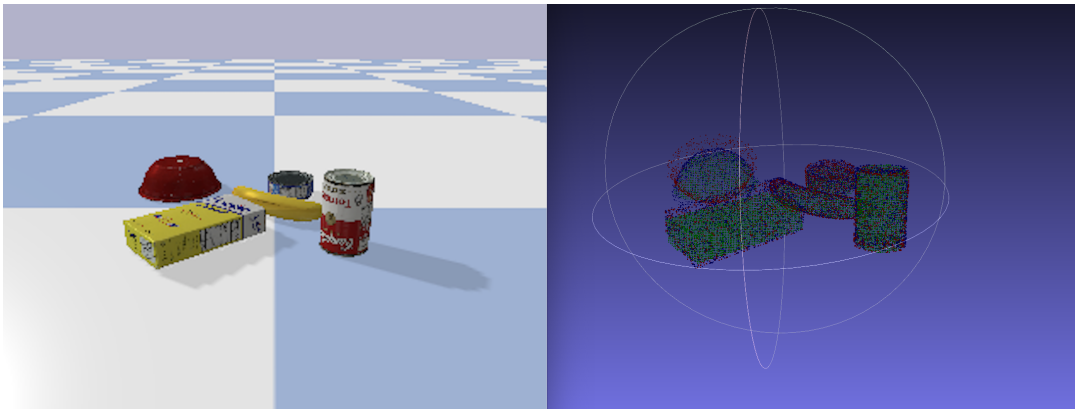
Intuitively, the lower threshold and the more iterations to run, the better alignment. But this can result in wasting computation on very tiny or even no improvement. You can experiment on the parameters and try to improve the result on harder cases. For Initial transformation, try Procrustes' analysis. **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.**

3. Object pose estimation (25 points)

Finally, let's wrap up what we have done and collect the result!
Implement the `estimate_pose()` method that takes depth and mask images of a scene as input to perform pose estimation on each object. Then, implement main() to perform pose estimation. For the validation set, use both ground truth and predicted masks. For the test set, use predicted masks.
(1) Qualitative result
The provided `export_gt_ply()` and `export_pred_ply()` can be 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	projected using ground truth mask	green	val
gtmask_transformed	estimated using ground truth mask	blue	val
predmask_transformed	estimated using predicted mask	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!



(2) Quantitative result
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 your result with the ground truth pose. Run:

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

The evaluation metric is Average Closest Point Distance, smaller is better. It's invariant under pose ambiguity caused by symmetries. In this homework, the number is about 1e-5 for ground truth pose. 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.
Make sure the requested files can be generated by running:

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

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

4. Further improvement (0 points)

generate extra training data or use more complicated segmentation model to reach higher mIoU

use multi-view images (three observations of the same scene are provided in `/hw2/hw2_multiview`)

try different/multiple initial transformations and optimize

denoise the predicted mask

denoise the projected point cloud when using predicted mask

encode more information per point than just the x,y,z location

propose your own way

Submission checklist

When you are done, make a new copy of your hw2 folder and double check your submission contains:

No `/venv_hw2` and `/YCB_subsubset`

`hw2_report.pdf`

completed code and any other files needed to reproduce your result

`learning_curve.png`

`checkpoint.pth.tar` of your best model

training images as .png files in

`/dataset/train/rgb/`, 300 files

`/dataset/train/gt/`, 300 files

predicted masks as .png files in

`/dataset/val/pred/`, 5 files

`/dataset/test/pred/`, 5 files

point clouds of the scene as .ply files in

`/dataset/val/exported_ply/`, 5 scenes * 3 files each

`/dataset/test/exported_ply/`, 5 scenes * 1 file each

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 hw2 folder as UNI_hw2. Zip the folder into one single `UNI_hw2.zip` file and upload to CourseWorks. We will review your report, evaluate your outputs against our grading scripts and run your code!