Homework 0

Problem 1

1. **compute_camera_matrix() description:**

   I first used the class variables and hints from the lecture that reviewed HW2 to calculate the intrinsic matrix parameters of the camera, saving them in the variable intrinsic_matrix. After this, I use the computeProjectionMatrixFOV function and the class variables to determine the projection matrix. Finally, I return a tuple of these matrices.

Problem 2

1. **Instance segmentation problem:**

   The task of segmentation is similar to image classification in that you are trying to infer the object categories in an image, but the difference is that you are trying to label every pixel with a semantic category instead of just giving one label for the whole image. The goal of this problem is as follows: given an input RGB image, output an image (no longer just a vector or one category) that has the same size as the input image. Within this image, each pixel will be labeled by an index (integer). This index corresponds to the category that this pixel belongs to. This task of segmentation is a lot more useful than classification because not only does it tell you the semantic category of the image and what objects are in the image, but also gives you the location of each of the objects in the image. In order to train the network we need to provide ground truth labels, the ground truth label in this particular task (pixel-wise label) is very expensive to get. Based on this, in this case the supervision we need is supervised learning as the algorithm uses prior knowledge involving the ground truth of a training set of images to train.

2. **Data Loader Code:**

   For the dataset.py file the following additions were made to the following methods:

   To the RGBDataset class __init__() function I first composed transforms.ToTensor() and transforms.Normalize() using transforms.Compose(). Next, to determine the number of samples in the dataset I entered the directory corresponding to the input value for dataset_dir. I then entered the '/rgb' subdirectory and counted all the .png files within it.

   Next, in the RGBDataset class __getitem__() function, to determine the path of the desired image I concatenate the instance's dataset_dir, /rgb, idx, and _rgb.png. I use this path to calculate rgb_img as the output of the transform applied to the read image. Next, if has_gt is False I return sample as just the input. Otherwise, I determine the mask name in a similar fashion to determining the rgb image name except I swapped rgb for gt. Then I use this mask name to read the mask, then perform the torch.LongTensor transformation to determine the target associated with the input rgb_img, then I return sample as the dictionary containing rgb_img and gt_mask.

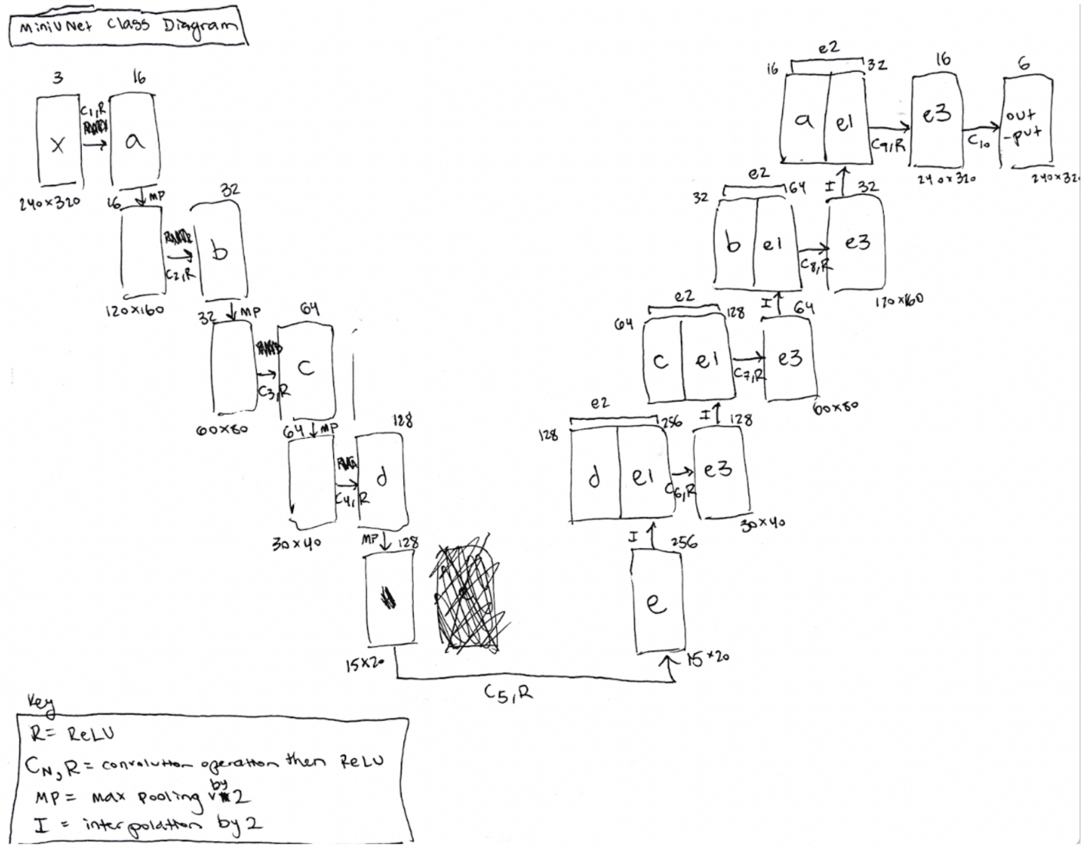
In the Segmentation.py file I made the following additions to the following methods:

Within the main() I created three RGBDataset class instances, one for each of the directory paths created a few lines above. For train and val I set the input value has_gt to True and for test I set it to False. Next, I created three DataLoaders using the three aforementioned class instances using a batch size of 4 for all three and setting shuffle to be true only for the train dataset.

In addition to these files, I also changed the line ‘sample = dataiter.next()’ in segentation_helper.py’s check_dataloader function to ‘try: sample = dataiter.next() except AttributeError: sample = next(dataiter)’ because I was getting the following error “”AttributeError: ’_MultiProcessingDataLoaderIter’ object has no attribute ’next”” and a post on EdStem suggested to make this changed and it worked.

3. **Constructing the network:**

   For this part I implemented my own version of MiniUNet in the model.py file. To do so I followed the 3 steps outlined in the instructions and reviewed the Neural Network and Training a Classifier links also from the instructions. The main thing that helped me create this class was the following sketch I derived from the given outline of a MiniUNet. Overall, it helped me plan the order of my functions and the many print statements I used in the forward function helped me debug by giving me the sizes at each step of the process. In the __init__() function of the class I created class variables c1 through c10 which were the convolution operations shown in the sketch below. As instructed I used kernel_size of 3 and a padding of 1 for all the convolution operations except for the last one. The last convolution layer only required a 1x1 convolution filter with padding = 0. In __init__() I also created the function variable pool to simplify the max pooling operation in the forward function. Within the forward function I performed the operations of MiniUNet detailed in the instructions in the order depicted in the sketch below using the operations I initialized in __init__().



4. **Rethinking network before training:**

1.

A convolution operation with dimensions nxn where n is not equal to 1 result in a change in the filter / image dimension. A change associated with 3x3 convolution is important for the downsampling and upsampling steps of MiniUNet, however, for the last step of the algorithm we are only interested in changing the image such that it is represented by a 6-dimensional vector instead of a 16-dimensional vector. Since we don't want to change the image size, a convolution operation of 1x1 is used to perform this change in dimensions since it is a filter made up of one number instead of a matrix so the size won't change.

2.

The output has 6 channels because the goal of segmentation is to label each pixel with a class that is associated with it. Every entry in this 6-dimensional state/vector/channels corresponds to a class (0 for the background and 1-5 for the objects). Since, in the 1-channel ground truth mask, each pixel stores an integer value standing for the id of the object that is makes up (aka the class label), we compare the 6 dimensional vector to it by comparing the index of the channel with the highest value to the value stored in the 1D mask vector. If these values are the same, our algorithm was successful.

5. **Training and validating the model:**

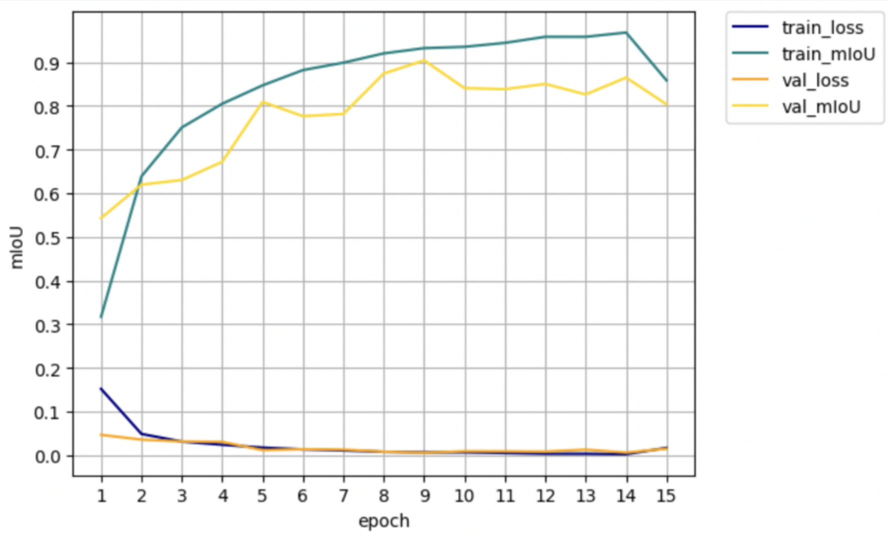
   In segmentation.py, I implemented the train() and val() functions and then defined criterion and optimizer.

   For the train() method I followed the pseudocode provided in the instructions. Some differences I made to this outline that likely increased the success of the program was I performed the forward + backward + optimize not only on every batch but on every image tensor as well. After finding a way to iterate through the batches of train_loader using an iterator type, I split the inputs and targets from the batch dictionary into two lists, saving batch_size as the length of these lists. Then I iterated through each tensor in the current batch and passed the input into the model after converting it from a numpy array to a tensor and adding a dimension to the front of it in order for it to be the correct size for the model class. I calculated loss and miou using the output of model as well as the target associated with the input I passed into model and added each of these values to a running total for each. Like I mentioned, this loop also contained the step for zeroing the parameter gradients of the optimizer and the backward and optimize steps as well. After iterating through each batch, I divided my miou and loss running totals by the length of the entire dataset and returned these resulting values. Throughout the method I made sure to set the device type of the tensor to the device input passed into the function so it could perform as accurately as possible.

   For the val() method I performed almost the exact same steps as for train(), except I removed the steps involving the optimizer. Additionally, a difference between train() and val() is that val() specifies the use of no gradient with the line 'with torch.no_grad()' enclosing the entirety of the function.

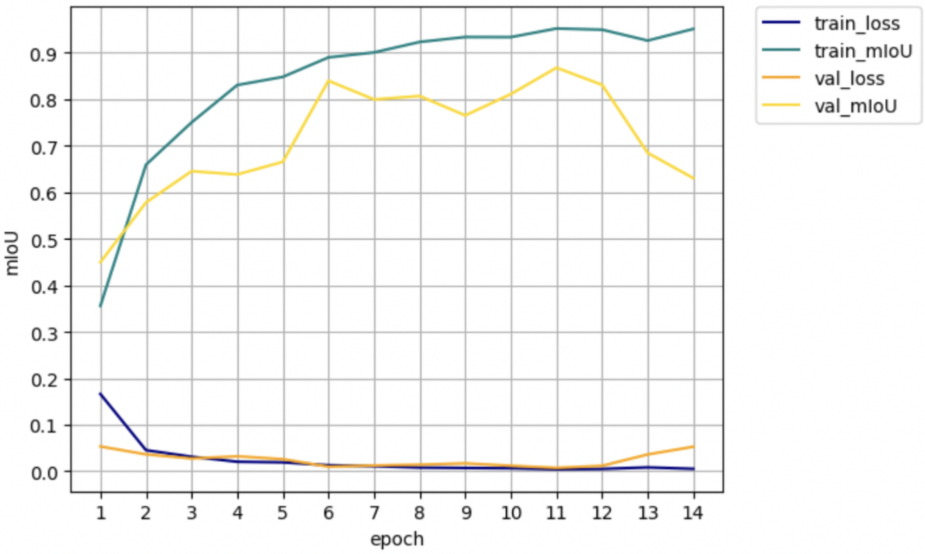
   main(): Since the main loss function associated with image segmentation is cross entropy loss, I defined the criterion to be this function. Additionally, based on the suggestion of an EdStem post, I made the optimizer an Adam optimizer, starting with a learning rate of 0.001. After changed the batch size input value to the Dataloader initializations from 4 to 8 based off of an Ed suggestion, I set the number of epochs to 15 and ran the file, which resulted in the following loss and mlou graph:

   Loss and MLoU Graph for Adam and 15 Epochs:



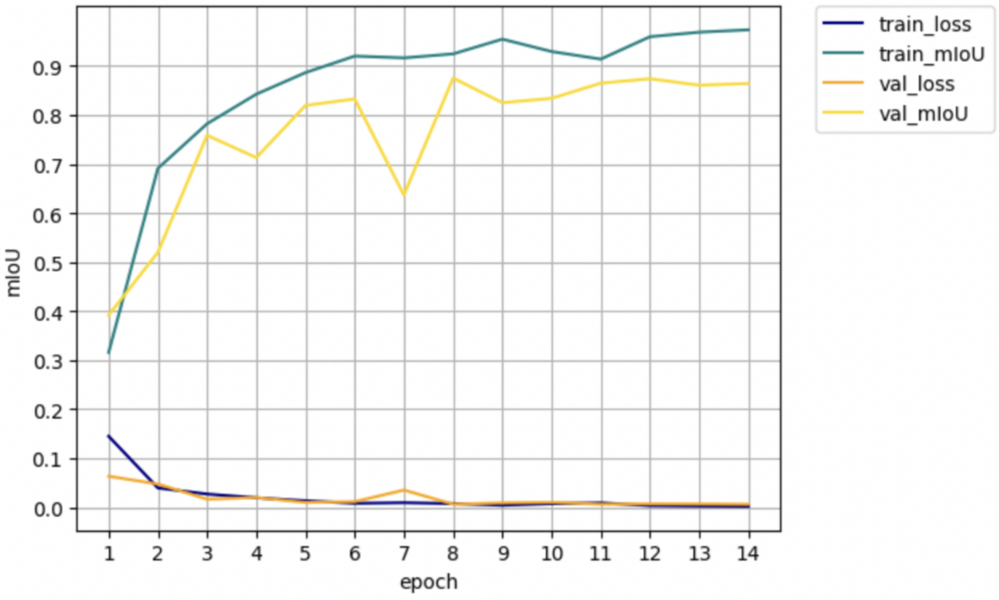
I thought I could improve the smoothness of the MLoU functions so I changed the optimizer to AdamW and reduced the total number of epochs to 14 to prevent the dipping down of the MLoU values seen in epoch 15. This resulted in the following graph.

Loss and MLoU Graph for AdamW and 14 Epochs:



I was not satisfied with this result so I went back to Adam for the optimizer. My final graph for this step of the assignment is shown below. This is a reasonable prediction as the model is trained to reach almost 90 percept MLoU on the validation set.

Loss and MLoU Graph for Adam and 14 Epochs:



Problem 3

1. **Prepare point clouds:**

   I implemented the gen_obj_depth() method by using the instance segmentation mask to create a depth image that only contains pixels of the specific objects. To accomplish this I created a copy np array of depth and if the input object id was not -1 I set all of the elements of this new 2D np array to zero if their mask value at this location was not equal to the object id. If object id was -1, I set the elements in which their corresponding mask value was 0 (indicating background) equal to zero.

   In order to implement obj_depth2pts() to create a point cloud projected from the depth image I used depth_to_point_cloud() from transforms.py as well as the helper functions cam_view2pose() and transform_point3s(). First I accessed the camera's intrinsic matrix from the camera class and determined the value of the camera pose matrix using cam_view2pose() with view_matrix as the input. I then used gen_obj_depth() to generate a depth image for the given object and inserted this depth image and the intrinsic matrix into depth_to_point_cloud() to get the points in the camera coordinate. Finally, I found the world coordinates by transforming the camera coordinate points using the camera pose matrix.

2. **Iterative Closest Point:**

   (1) I implemented the align_points() methods using the trimesh.registration.procrustes() and trimesh.registration.icp() functions. The procrustes function gave me the

'initial' input to the icp function. The try and except block is included in this function because the predicted mask might not have enough points to solve the transformation matrix.

(2) The initial parameters I used for the icp() method were: max_iterations=20 and threshold=1e-05. When I tuned the max_iterations to 100 I noticed _. From the original setting, when I tuned the threshold to 1e-20 I noticed _. Finally, from the original settings, when I changed the initial matrix from the output of the procrustes matrix to a completely empty matrix of the same shape I noticed _.

3. **Object Pose Estimation:**

In this part I implemented the estimate_pose() method which performs pose estimation on each object in the provided image. To do so I iterated through each of the object in LIST_OBJ_FOLDERNAME and for each object I determined the depth points using obj_depth2pts, then used the length of this value to determine the mesh points using obj_mesh2pts. Lastly, I called align_pts using the depth and mesh points as inputs as well as a max_iterations value of 20 and threshold of 1e-05, appending the output to an array that is returned at the end of the method.

In main(), for each scene_id I loaded the scene view matrix into view_matrix, the depth image into depth, the predicted mask into predmask, and the list of object poses associated with the predicted mask into list_obj_pose_pred. If the option selected when icp.py was called was val I then loaded the gtmask of the scene into gtmask and exported the gt_ply. Next, I calculated the list of object poses associated with this gtmask instead of the predmask and saved it to list_obj_pose_gt. I then exported the prediction image associated with the ground truth mask transformed and saved the poses in list_obj_pose_gt to gtmask within /dataset/val/pred_pose/gtmask.

One problem I had when calling the evaluate_icp.py script was that gt_pose was empty so the script would never run. The save_pose method provided only saves to pred_pose and not to gt_pose so I decided I would save the objects in list_obj_pose_gt into gt_pose in order to get the script to run. I recognize that this is not the correct contents of the gt_pose directory but I was unsure what the correct contents would be. After doing this the evaluate script worked. Despite this, the overall scores seem to be far from what they should be and this is most likely due to the contents of gt_pose.

Next, in main(), since both test and val require the exporting the prediction image associated with prediction mask transformed and saving the predicted poses from list_obj_pose_pred to /dataset/val/pred_pose/predmask, I did that at the end of the loop.