Homework 2

Problem 1

```
camera.py
    compute_camera_matrix()
    Input - nothing
    Output - two matrix,
    1. intrinsic camera matrix representing camera parameters and
    2. a projection matrix.
```

The camera intrinsic matrix is calculated as -

$$Q = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

Here,

$$o_x = \frac{width}{2} \tag{1}$$

$$o_y = \frac{height}{2} \tag{2}$$

Assuming origin to be at the top left of the image.

The projection matrix is calculated using pybullets method, computeProjectionMatrixFOV which takes field of view height, aspect and near and far values as the input as,

$$aspect = width/height \tag{3}$$

$$projection\, matrix = \begin{bmatrix} \frac{1}{\tan fov/2} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan fov/2} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & -1 \\ 0 & 0 & \frac{2nearfar}{near-far} & 0 \end{bmatrix}$$

$$FOV = 2 \tanh \frac{h}{2f} \tag{4}$$

Problem 2

Part1

1. Describe the instance segmentation problem. What are the inputs and outputs of the task? What supervision do we need?
   Solution
   Instance segmentation is a form of image segmentation which deals with detecting and delineating each distinct instance of an object appearing in an image. It separates for example, every person from a scene into a different class.

   Input to the task is a scene/image and the output is a mask representing each object instance in the image, helping to identify which pixels belong to which object, could also have a label corresponding to a class.

   To train the model, we need ground truth information about the location and class of object instances in the image. Mostly done by manually annotating the scene, various object instances and their corresponding labels.

Part 2

Data loader

```
dataset.py
class RGBDataset(Dataset):
    1. initialize, and apply appropriate transform as
       transforms.Compose([transforms.ToTensor(),
       transforms.Normalize(mean_rgb, std_rgb),])
    2. Find thel length of the dataset, number of images
       self.dataset_length = len(list images in the given path)
    3. Iterate, and pair transformed rgb images with truth mask if any
       sample = {}
       if self.has_gt is False:
           rgb_img = image.read_rgb(path)
           rgb_img = self.transform(rgb_img)
           sample = {'input': rgb_img}
       else:
           rgb_img = image.read_rgb(path)
           rgb_img = self.transform(rgb_img)
           gt_mask = torch.LongTensor(image.read_mask(path)
           sample = {'input': rgb_img, 'target': gt_mask}
```

```
        return sample
segmentation.py
main()
    create instances of RGBDataset for training, ground truth mask
    is true for training and validation, and false for test dataset.
    train_dataset = RGBDataset(train_dir, True)
    val_dataset = RGBDataset(val_dir, True)
    test_dataset = RGBDataset(test_dir, False)
    Similarly, create instances of DataLoader and pass batch size,
    datasets read using RGBDataset and shuffle as true only
    in the case of training.
    train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size, shuffle=False)
```

Part 3

Construct the network

```
model.py
class convolutionBlock
     apply a 3 x 3 convolution, followed by ReLU
class encoderBlock
    apply convolution block and use pooling to downsample.
class decoderBlock
    upsample the image using ConvTranspose2d and concatenate with skip
    connections which has the same dimensions as the upsampled image.

class MiniUNet(nn.Module):
    def __init__(self):
        e1 -> encoderBlock(3, 16)
        e2 -> encoderBlock(16, 32)
        e3 -> encoderBlock(32, 64)
        e4 -> encoderBlock(64, 128)
        b -> convolutionBlock(128, 256)
        d1 -> decoderBlock(256, 128)
        d2 -> decoderBlock(128, 64)
        d3 -> decoderBlock(64, 32)
        d4 -> decoderBlock(32, 16)
        out -> 1 x 1 convolution

    def forward(self, x):
```

```
In:x: Tensor, channel=3 for rgb input
Out:output: Tensor, class=number of objects + 1 for background
s1, p1 = e1(x)
s2, p2 = e2(p1)
s3, p3 = e3(p2)
s4, p4 = e4(p3)
b = self.b(p4)
here decoder block is fed with the skip connections which have
the same dimensions as the new upsampled values
d1 = d1(b, s4)
d2 = self.d2(d1, s3)
d3 = self.d3(d2, s2)
d4 = self.d4(d3, s1)
output = self.output(d4)
return output
```

Part 4

Rethink the network before you start training it. 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)
   Solution
   A 3*3 kernel is used in the convolutional layers to capture the spatial information in the input image, extract a rich set of features to capture local and global information about the image, and helps network to learn more complex representation. Essentially they are small enough to capture fine details, and large enough to capture spatial relationships between pixels.

   1*1 convolution are used at the output of a convolutional neural network for dimensionality reduction. 1*1 convolutions reduces the dimensionality of the output from 16 to 6 channels, which helps to reduce the computational cost of the network and prevent overfitting by reducing the number of parameters in the model. Essentially, the information is compressed to a lower dimensional space, and preserving the relevant features. In general, 1*1 convolutions could be used to combine learned deatures from different channels more accurately, to help focus the network on important channels by caculating a weighted sum of the channels which would result in ignoring of the redundant channels.

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)

   Solution

   The network takes RGB images as a input for feeding into the UNet model and than output is a set of channels, corresponding to one channel each for the object classes model is trained to detect, and one channel for the background class. Since our model is trained to detect 5 objects - tomato soup can, tuna fish can, banana, bowl, and sugarbox, we have 6 channels, where one channel corresponds to the background, and remaining five correspond to the five object classes.

   Each channels outputs the probability of a pixel to belong to a specific class. Network learns to assign higher probabilities to pixel more likely to belong to an object of a corresponding class, which is than applied to go thorugh a threshold, creating a binary mask for each object indicating if a pixel belongs to the object.

   To compare the output of the UNet model with ground truth mask, we can use metric such as mean intersection over union, mIoU. IoU is essentially a measure of the overlap between the predicted pixels and the ground truth pixels. It is defined as the area of intersection between the two sets divided by the area of their union.

Part 5

Train and validate the model

Training and validation, parameters -
Loss - cross-entropy loss
Optimizer - Adam
Learning rate - 1e-3
Epochs - 30

```
segmentation.py
def train(model, device, train_loader, criterion, optimizer):
    Loop over each sample in the dataloader -
        forward pass calculate mean intersection over union and loss
        backward pass and update and optimize the weights
    calculate the average loss and miou
```

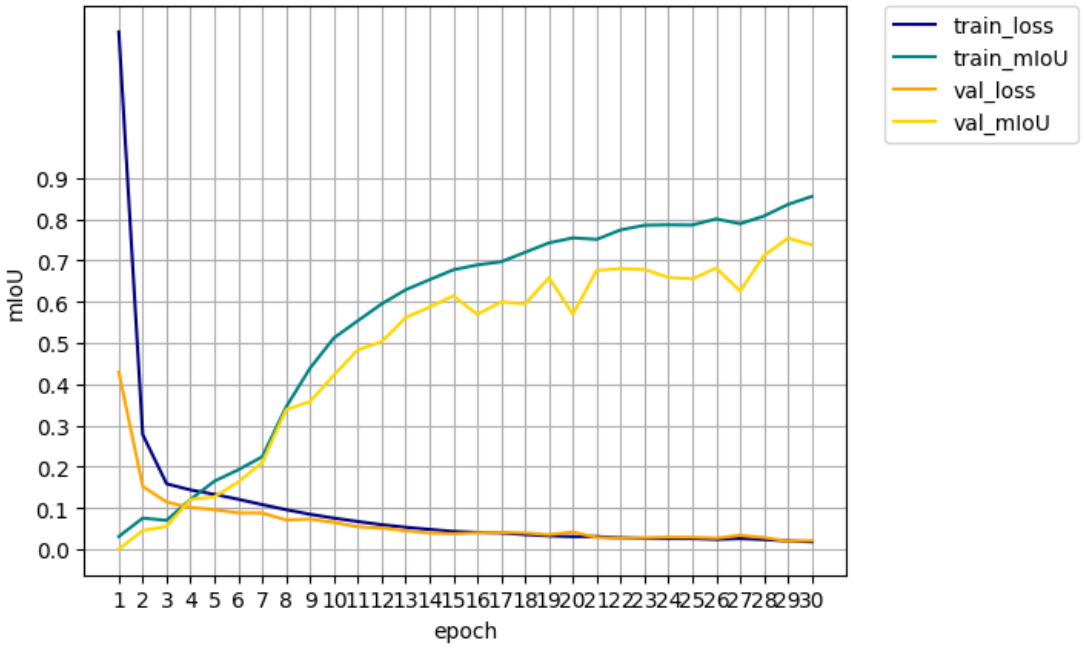

Figure 1: Learning curve

```
segmentation.py
    def val(model, device, val_loader, criterion):
    Similar to train(), but no need to backward and optimize.
    with torch.no_grad():
       Loop over each of the batch in dataloader
            and do a forward pass,
            and calculate loss and mean intersection over union
    calculate the average loss and miou
```

The output of the trained model is shown in 1

Problem 3

Part1

Prepare point clouds

Here two point clouds are prepared, one is down-sampled from the .obj file and another is projected from the depth image of the object. In icp.py gen obj depth() method, generates depth corresponding to a specific object based on the object id,

in case of object id being -1, depth corresponding to all the objects is generated. Also, objdepth2pts is used to project generate point cloud from depth of the specific object(s) into the world frame of reference.

```
icp.py
def (obj_id, depth, mask, camera, view_matrix):
    obj_depth = gen_obj_depth(obj_id, depth, mask)
    point_cloud = depth_to_point_cloud(camera.intrinsic_matrix,
    obj_depth)
    world_pose_matrix = np.linalg.inv(cam_view2pose(view_matrix))
    world_pts = transform_point3s(world_pose_matrix, point_cloud)
```

Part2

Iterative Closest Point (ICP)

1. Implement align pts() method

   Here iterative closest point algorithm to estimate a transformation that aligns the point cloud is used, first trimesh's method procrustes is used to find a initial tranformation which is fed to the icp method.

   ```
   icp.py
   def align_pts(pts_a, pts_b, max_iterations=2000,
   threshold=1e-40):
          matrix_initial, _, _ = trimesh.registration.procrustes(
          pts_a, pts_b, reflection=False, scale=False)
         matrix, _, _ = trimesh.registration.icp(pts_a, pts_b,
         initial=matrix_initial, threshold, max_iterations,
         reflection=False, scale=False)
      return matrix
   ```

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.
   Observations
   1. Changing the cost (threshold) - a lower threshold value leads to better results, better alignment and more computational effort and more time to converge
   2. Increasing number of iterations leads to better alignment and more computational time

3. Initial transformation - a better initial estimate helps the algorithm quickly align since algorithm does not have to iterate a lot to find the correct alignment

Part3

Object pose estimation

Estimate pose method in icp.py is used to calculate the pose of each of the objects.

```
icp.py
def estimate_pose(depth, mask, camera, view_matrix):
    list_obj_pose = []
    for all object ids
        obj_pts_depth = obj_depth2pts(obj_id+1, depth, mask, camera,
        view_matrix)
        obj_pts = obj_mesh2pts(obj_id+1, point_num=len(obj_pts_depth))
        T = align_pts(obj_pts, obj_pts_depth)
        list_obj_pose.append(T)
    return list_obj_pose
```

In case of argument being for validation use both ground truth and predicted masks. For the test set, use predicted masks.

```
icp.py
main()
        if dataset_dir == "./dataset/val/":
            loop over objects
                depth = image.read_depth()
                view_matrix = np.load()

                gt_mask = image.read_mask()
                list_obj_pose_gt = estimate_pose()
                save_pose()
                export_gt_ply()
                export_pred_ply()

                pred_mask = image.read_mask()
                estimate_pose()
                save_pose()
                export_pred_ply()
        else:
            loop over objects
```

```
pred_mask = image.read_mask()
depth = image.read_depth()
view_matrix = np.load()
list_obj_pose_pred = estimate_pose()
save_pose()
export_pred_ply()
```

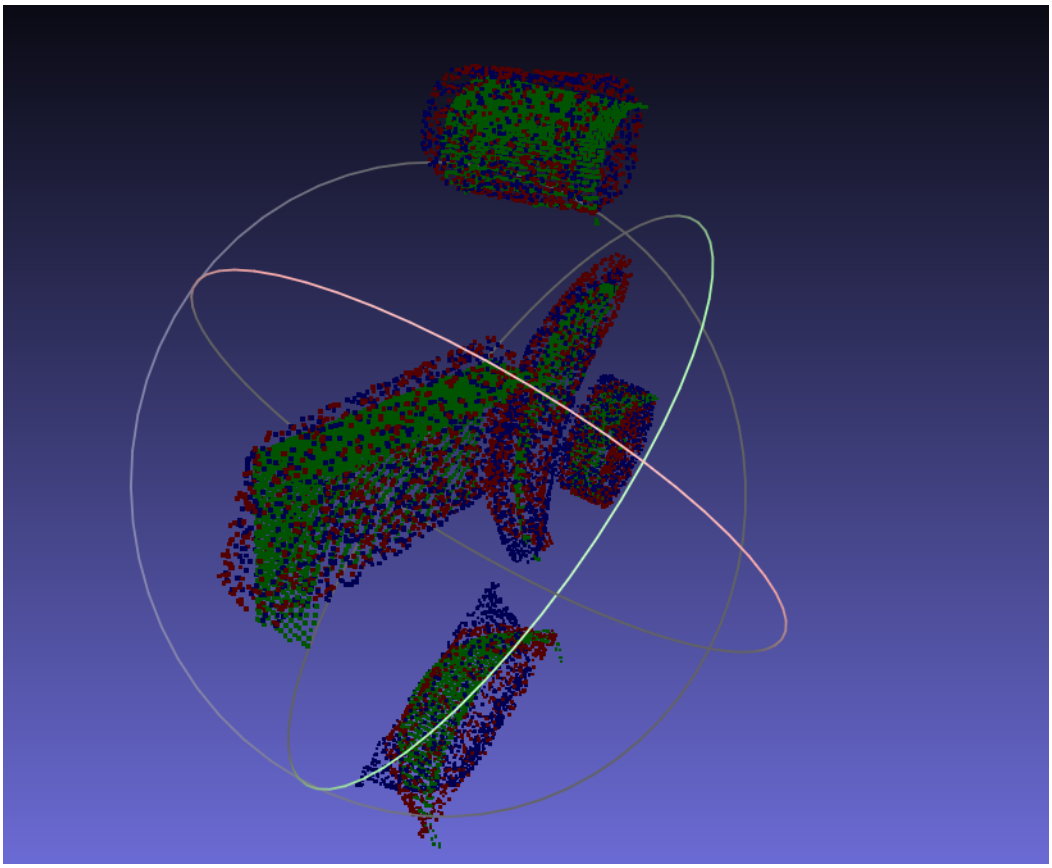

Figure 2: Resulting point clouds for validation datasets

The final results are as 2.