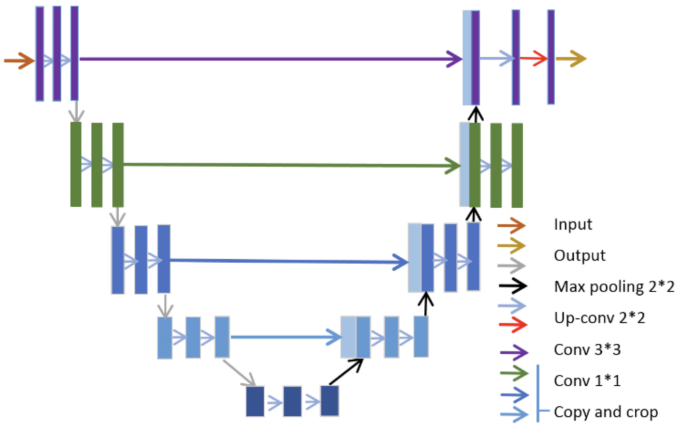
ECE64106/BME646 Homework 7

mUNet Explanation:

- In this homework we were asked to work on semantic segmentation using the Unet network which works on an encoder decoder architecture. The use case for this network is very similar to YOLO(which we implemented previously).
- The difference arises from how the identification and differentiation of each class in the image is done, while in YOLO architecture, we segment the image and create anchor boxes which differentiate the blob of pixels which we want to classify i.e the classification happens on the level of segments and anchor boxes that we create, in semantic segmentation, this differentiation is done on the pixel level.
- To achieve this differentiation on a pixel level we first need to break down the image into the basic features(feature map) similar to how we do it for the YOLO approach, using convolutions and from this broken down feature map we reconstruct the actual image in the same dimensions as the input image, by upsampling the feature maps using transpose convolutions.
- For the downsampling/creation of feature maps, we use a normal convolutional network which we then pass through an Atrous Sample Pyramid Pooling(ASPP) to generate the segmentation map. This is done as Atrous convolutions allow for conservation of space invariance, which is an important factor, as this type of network requires a very large number of layers and the space invariant nature of the Atrous Convolutions allow us to keep the number of learnable parameters to increase exponentially as the number of layers increase.
- In this first leg of the network we downsample the image using a convolutional network and ASPP to generate a segmentation map.
- Once the segmentation map is generated we plug it into the next leg of the network for upsampling, to generate the estimated image. The upsampling is done using transpose convolutional layers similar to normal convolutions were used for downsampling the image to its basic features.
- Both the Downsampling and Upsampling are done using skip blocks similar to what we used in the previous networks, which are a combination of convolutional layer and batch normalization layer with the option to skip the intermediate layers if the option to skip has been enabled.
- The downsampling skip block is similar to the skip blocks that we used previously. Using stride to downsample the images to generate the segmentation map.
- The upsampling skip block is also quite similar to the the downsampling skip blocks with the difference that it uses transpose convolutions and dilation to implement the ASPP and the upsampling of images, using the dilation and stride parameters to generate the approximation of the original image from the segmentation map.
- For our case, the inputs to the network also include the pixel level masks for the different classes in the image, which is also passed as an features to the network. So when the image is reconstructed by the network, we also obtain the estimated masks for the classes contained in the image, which we can then compare to the masks from the original image to understand how well our network is working.



U Net Architecture(https://www.researchgate.net/figure/The-architecture-of-Unet_fig2_334287825)

Results from *semantic_segmentation.py script:*

MSE loss

Loss graph:

Best Loss: `333.9312496948242`

Worst Loss: `461.703271`

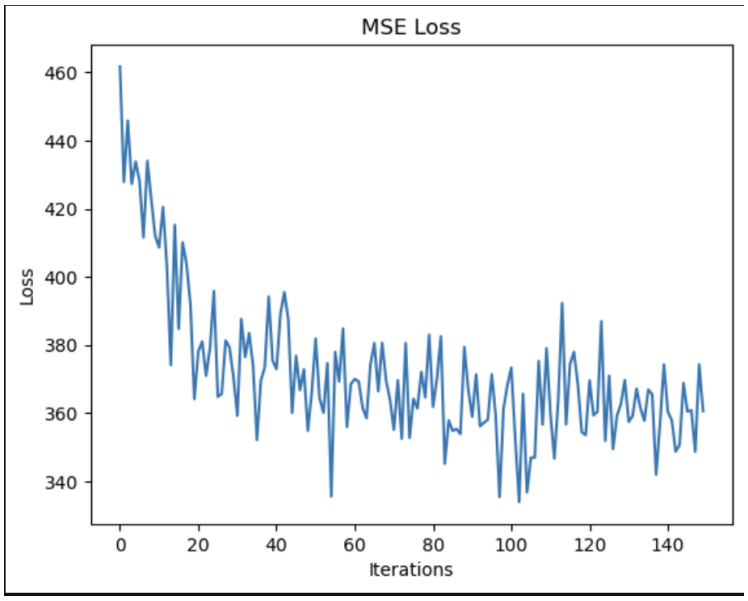


Fig 1

Test Results for batch 201 for MSE :

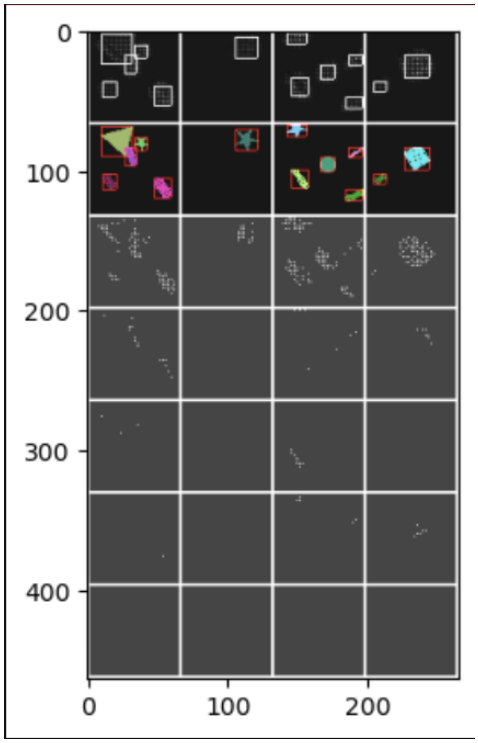


Fig 2

From the test results results we can see that for the rows for the different shapes, we are not able to see the most of the shapes in a very identifiable manner when using the MSE Loss.

Dice Loss and Combined Loss(Dice+MSE) Implementation:

Dice Loss:

The degree of similarity between two masks, A and B, is measured by the dice loss. In particular, the Dice coefficient, D(A, B), functions as an accuracy metric for the prediction where A stands for the ground truth mask and B for the anticipated mask, allowing us to understand how similar the predicted mask is to the actual mask.

$$L = 1 - D(A, B)$$

$$D(A,\ B) = \frac{2|A \cap B|}{|A| + |B|}$$

1. **Only Dice Loss:**For the first case I have implemented just plain dice loss and the performance for the same is not great, please find the code and results from the same below::

```
def dice_coefficient(y_true, y_pred, epsilon=1e-3):
    intersection = torch.sum(y_true * y_pred)
    union = torch.sum(y_true) + torch.sum(y_pred)
    dice_coeff = (2. * intersection + epsilon) / (union + epsilon)
    return dice_coeff

def dice_loss(y_true, y_pred):
```

```
    return 1 - dice_coefficient(y_true, y_pred)
```

Best Loss: `33.18495580673218`

Worst Loss: `49.096791`

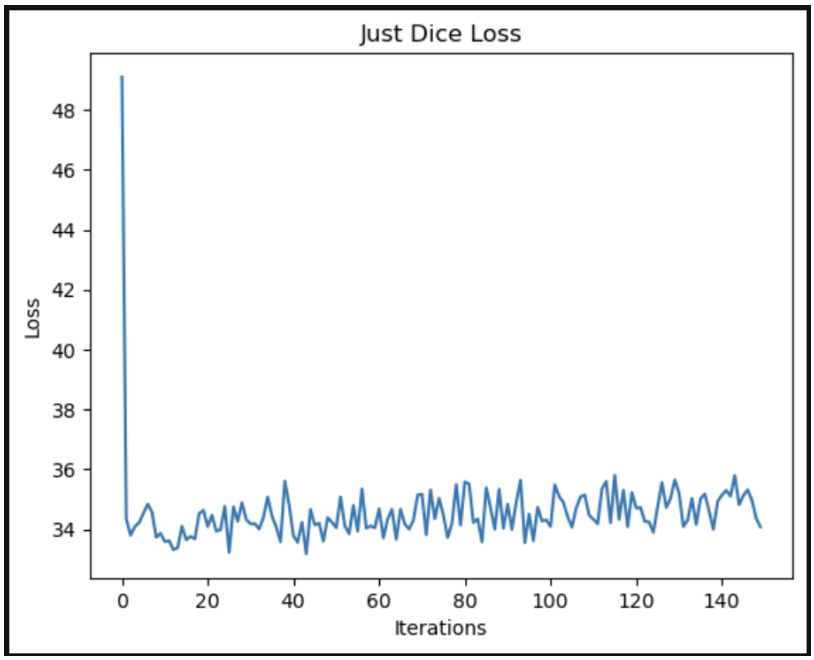


Fig 3

Test Results for batch 201 for only Dice loss :

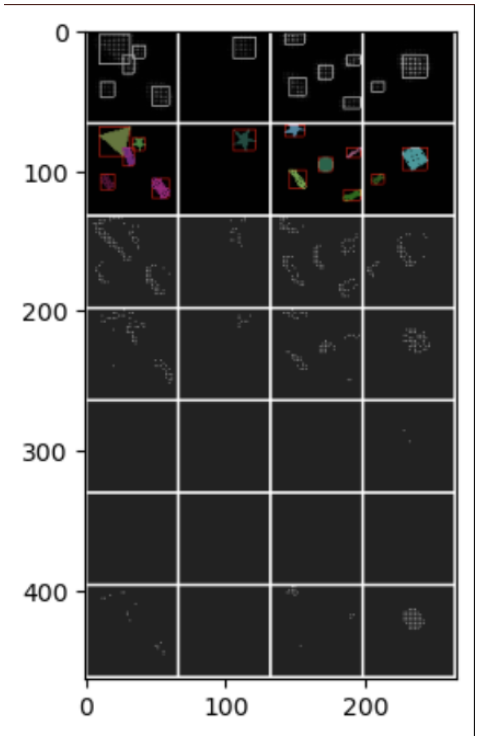


Fig 4

From the test results results on unseen data we can see that for the rows for the different shapes, we are not able to see the most of the shapes in a very identifiable manner, but the overall effect of differentiating boundaries for the different shapes seems to be better than when using the MSE Loss.

2. **Combined MSE with Dice Loss:** For this case, I have implemented the combination of dice loss and mse loss to overcome the disadvantages of pure mse loss, with alpha as the scale value. The idea of using alpha in this manner was picked up from this paper(https://arxiv.org/pdf/2006.14822.pdf), where they have used MSE loss combined with a type of Cross entropy loss. For the same I implemented the below code:

```
class MSEPlusDiceLoss(nn.Module):
                def __init__(self, batch_size, alpha=0.5, epsilon= 1e-3):
                    super(SemanticSegmentation.MSEPlusDiceLoss, self).__init__()
                    self.batch_size = batch_size
                    self.alpha = alpha
                    self.epsilon = epsilon

                def forward(self, output, mask_tensor):
                    composite_loss = torch.zeros(1, self.batch_size, requires_grad=True)
                    dice_loss = torch.zeros(1, self.batch_size, requires_grad=True)
                    mse_loss = torch.zeros(1, self.batch_size, requires_grad=True)


                    for idx in range(self.batch_size):
                        for mask_layer_idx in range(mask_tensor.shape[0]):
                            mask = mask_tensor[idx, mask_layer_idx, :, :]
                            output_mask = output[idx, mask_layer_idx, :, :]
```

```
                            if torch.sum(mask) + torch.sum(output_mask) > 0:
                                intersection = torch.sum(output[idx, mask_layer_idx, :,
:] * mask)
                                dice_coefficient = (2. * intersection + self.epsilon ) /
(torch.sum(output[idx, mask_layer_idx, :, :]) + torch.sum(mask) + self.epsilon)
                                dice_loss_copy = dice_loss.clone()
                                dice_loss_copy[0, mask_layer_idx] = 1 - dice_coefficient
                                dice_loss = dice_loss_copy



                        if torch.sum(mask_tensor[idx])>0 and torch.sum(output[idx])>0  :
                            mse = nn.MSELoss()
                            mse_loss_copy = mse_loss.clone()
                            mse_loss_copy[0, idx] = mse(output[idx], mask_tensor[idx])
                            mse_loss = mse_loss_copy


                        if torch.sum(mse_loss)>0 and torch.sum(dice_loss)>0:
                            composite_loss_copy = composite_loss.clone()
                            composite_loss_copy[0, idx]= self.alpha*torch.sum(mse_loss) +
(1-self.alpha)*torch.sum(dice_loss)
                            composite_loss = composite_loss_copy


                    return  torch.sum(composite_loss) / self.batch_size
```

This allowed for a significant improvement to plain MSE loss, the loss graph for the same and the same compared to MSE and plain dice loss can be found below:

For best performing scale variable alpha = 0.5

Best Loss: `2.968087463378906`

Worst Loss: `124.22162278413772`

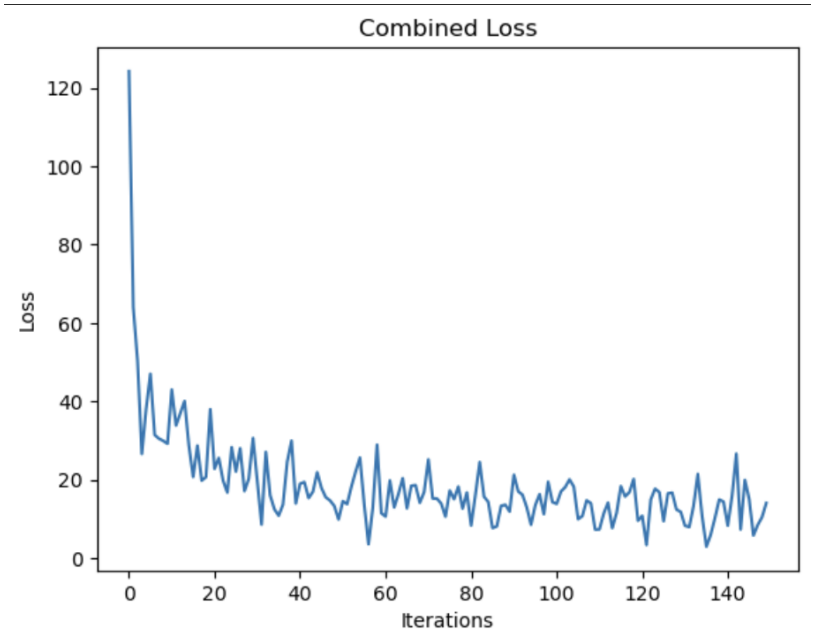


Fig 5

Test Results for batch 201 for Combined loss with best alpha:

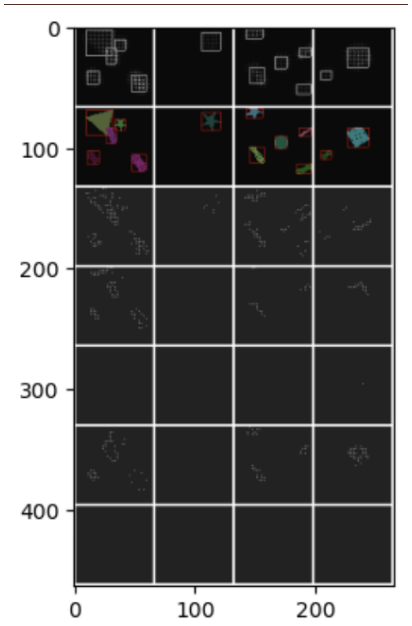


Fig 6

From the test results results on unseen data we can see that for the rows for the different shapes, we are able to see the most of the shapes in a somewhat identifiable manner with somewhat differentiable boundaries for the different shapes which seems to provide a better performance as compared to the other two loss functions.

For worst performing scale variable alpha = 0.80

Best Loss: `48.92407157897949`

Worst Loss: `169.01005604743958`

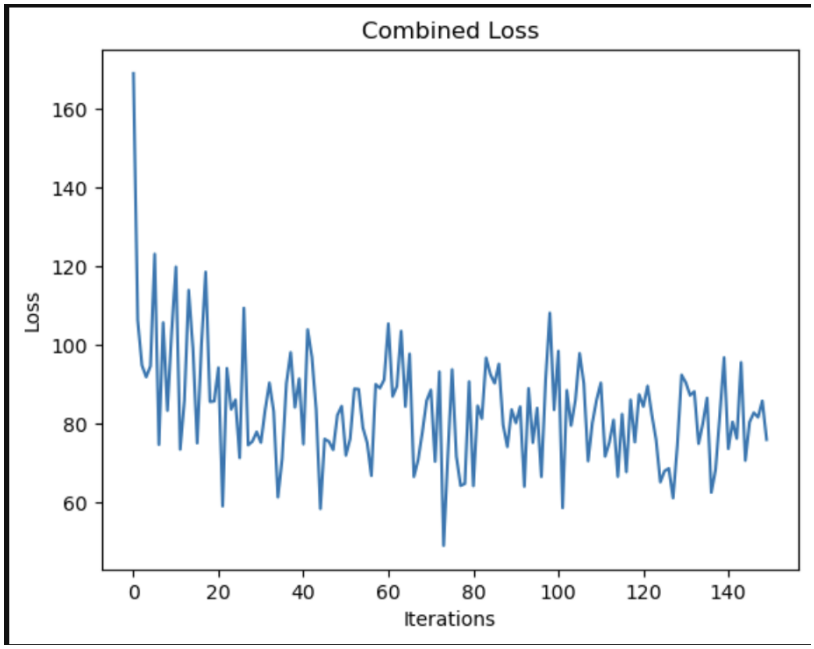


Fig 7

Test Results for batch 201 for Combined loss with worst alpha:

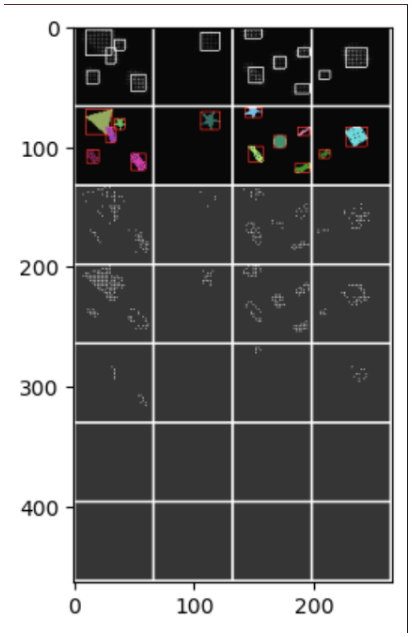


Fig 8

The training loss when the alpha value is set to 1 does go down a bit but does so very erratically, which is very similar to the behavior when training is done with plain MSE(Fig 1). As the higher alpha value leads to a higher contribution from the MSE loss than the dice loss.

3. **Comparison of the losses for the different loss functions(here the combined loss with the best apha is plotted):**

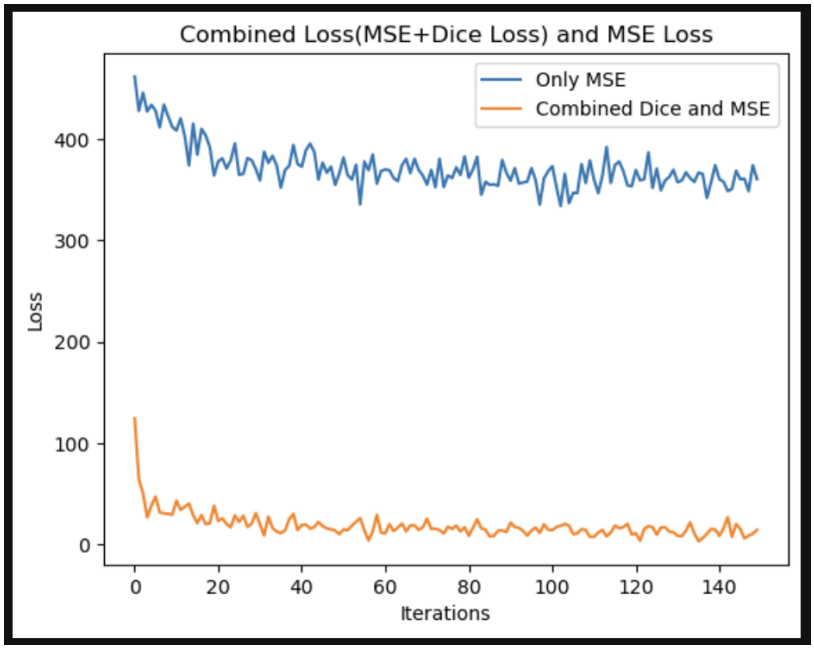


Fig 9

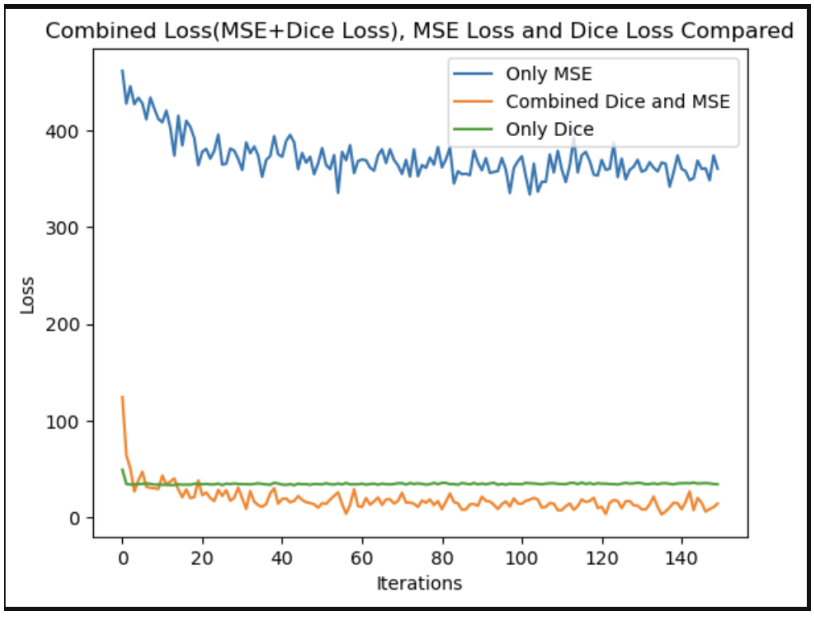


Fig 10

Potential factors contributing to the observed variations in performance for the three losses:

1. MSE Loss:
    a. From the training results, it is observable that the loss drops a bit but still stays pretty high overall, this might be due to the fact that MSE is not able to handle the issue of class imbalance as well as the added factor that MSE gets affected by outliers inordinately.
    b. From the test results results we can see that for the rows for the different shapes, we are not able to see the most of the shapes in a very identifiable manner when using the MSE Loss.

2. Dice Loss:
    a. From the training results, we can see that the loss is brought down quite a bit as compared to MSE, owing to the fact that Dice Loss is much better at differentiation from MSE and is able to segment the required classes much more efficiently. Additionally, not greatly affected by outliers as compared to MSE.
    b. From the test results results on unseen data we can see that for the rows for the different shapes, we are not able to see the most of the shapes in a very identifiable manner, but the overall effect of differentiating boundaries for the different shapes seems to be better than when using the MSE Loss.
3. Combined(MSE+Dice) Loss:
    a. From the training resuls, we can see that the combined loss function performs much better than both the loss functions, as it combined the aspect of differentiation and localization from dice loss with the pixel level accuracy provided by MSE.
    b. From the test results results on unseen data we can see that for the rows for the different shapes, we are able to see the most of the shapes in a somewhat identifiable manner with somewhat differentiable boundaries for the different shapes which seems to provide a better performance as compared to the other two loss functions.

Qualitative Observations leading to better performance by Combined Loss over plain MSE:

1. Firstly, addition of Dice Loss to the MSE Loss, helps with the issue of class imbalance, where certain classes are much smaller or less in number than the other classes. Dice loss contributes to this, by penalizing false positive and false negatives separately. While the MSE loss part add to the better performance in ensuring the pixel level accuracy. If it would have been just the MSE loss, it would have focused just on the pixel level accuracy.
2. Addition of Dice loss aligns with evaluation metrics commonly used in diffrentiation and localization tasks, such as Intersection over Union (IoU). By using the metric with Dice loss, we are directly optimizing the models ability to segment or classify the different classes present in an image.
3. Addition of dice loss to the mse loss also helps to stabilize the loss, as plain mse tends to be more sensitive to outliers and noise in general, addition of dice loss helps to decrease this sensitivity and allows the model to focus on data points much more relevant to the semantic segmentation leading to higher stability.

Extra Credit:

Extracting Image and Mask Tensor:

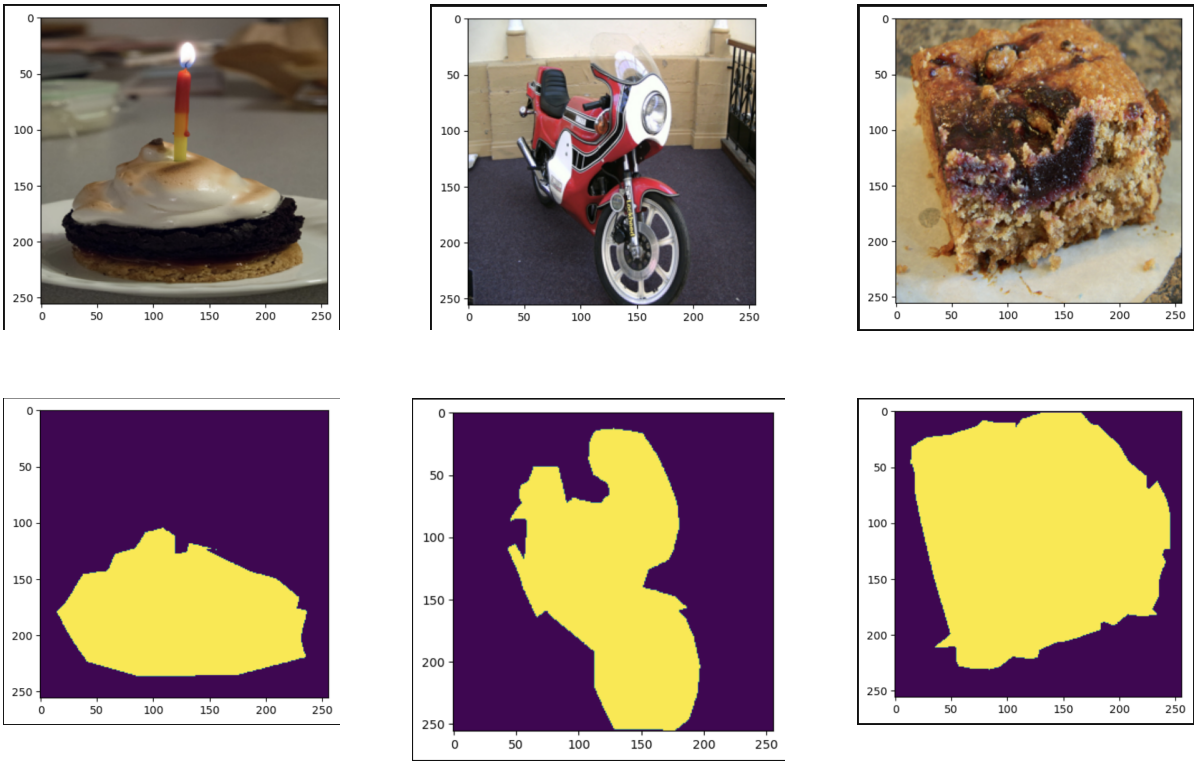
Using the pycocotools library, we will be extracting the image tensors and mask tensors for single object image instances with classes cake, dog and motorcycle which have at least 200 X 200 bounding box area. The image tensor will be be extracted in the same manner that it was in the previous assignments, while for extracting the mask tensor we will be using the annToMask function to extract the image mask from the annotations. We will also be resizing the image to 256 × 256 and ensuring that the mask tensor is also resized match the image size. A total of 1976 train images were extracted and 511 test images were extracted. This has been implemented using the below code, which contains the function, dataset and dataloader for the same:

```
root_directory_train = '/Users/avnishkanungo/Desktop/coco-dataset/train2017/train2017/tra
in2017'
root_directory_test = '/Users/avnishkanungo/Desktop/coco-dataset/train2017/train2017/val2
017'
classes = ['cake', 'dog', 'motorcycle']
```

```
def getImgAndMask(coco, path, classes):
    l = coco.getImgIds() #list(coco.imgs.keys())
    masks = []
    img = []
    class_ids = [coco.getCatIds(catNms=[class_name])[0] for class_name in classes]
    for i in l:
        x = coco.getAnnIds(i)
        y = coco.loadAnns(x)
        if y:
            if y[0]['category_id'] in class_ids:
                image_info = coco.loadImgs(i)[0]
                mask = np.zeros((image_info["height"], image_info["width"]), dtype=np.uin
t8)

                if len(y) == 1:
                        bbox = y[0]['bbox']
                        bbox_size = max(bbox[2], bbox[3])

                        if bbox_size >= 200:
                            image_path = f"{path}/{image_info['file_name']}"
                            image = Image.open(image_path).convert("RGB")
                            image = image.resize((256,256))
                            img.append(image)

                            for ann in y:
                                mask += coco.annToMask(ann)
                                mask = zoom(mask, (256/image_info["height"], 256/image_in
fo["width"]), order=1)
                                masks.append(mask)

    return masks, img


class CustomCocoDataset(Dataset):
    def __init__(self, masks, images, transform=None):
        self.masks = masks
        self.images = images
        self.transform = transform

    def __len__(self):
        return len(self.masks)

    def __getitem__(self, idx):
        mask = self.masks[idx]
        image = self.images[idx]

        if self.transform:

            mask = torch.tensor(mask, dtype=torch.float32)
            image, transformed_mask = self.transform(image, mask)
```

```
        return transformed_mask, image

class CustomTransform:
    def __init__(self, image_transforms):
        self.image_transforms = image_transforms

    def __call__(self, image, mask):
        transformed_image = self.image_transforms(image)
        transformed_mask = torch.unsqueeze(mask, 0)  # Add channel dimension
        return transformed_image, transformed_mask

image_transform = tvt.Compose([
    tvt.ToTensor(),
    tvt.Resize(size=(256,256))
])

transform = CustomTransform(image_transform)

im_masks_test, im_img_test = getImgAndMask(test_ann, root_directory_test, classes)
im_masks_train, im_img_train = getImgAndMask(train_ann, root_directory_train, classes)

test_dataset = CustomCocoDataset(im_masks_test, im_img_test, transform=transform)
train_dataset = CustomCocoDataset(im_masks_train, im_img_train, transform=transform)
```

Output of the Image Tensors and the Mask tensor:



I have used the same network i.e mUnet from the DLStudio Module, with a small change that the output channel in the final layer will be 1, as we will be comparing the output to the image mask, which has just one channel.

```
# the coed for the network was picked from the DL Studio Library by Prof. Kak.

class mUnet(nn.Module):

    def __init__(self, skip_connections=True, depth=16):
                    super(mUnet, self).__init__()
                    self.depth = depth // 2
                    self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
                    ##  For the DN arm of the U:
                    self.bn1DN  = nn.BatchNorm2d(64)
                    self.bn2DN  = nn.BatchNorm2d(128)
                    self.skip64DN_arr = nn.ModuleList()
                    for i in range(self.depth):
                        self.skip64DN_arr.append(SkipBlockDN(64, 64, skip_connections=ski
p_connections))
                    self.skip64dsDN = SkipBlockDN(64, 64,   downsample=True, skip_connect
ions=skip_connections)
                    self.skip64to128DN = SkipBlockDN(64, 128, skip_connections=skip_conne
ctions )
                    self.skip128DN_arr = nn.ModuleList()
                    for i in range(self.depth):
                        self.skip128DN_arr.append(SkipBlockDN(128, 128, skip_connections=
skip_connections))
                    self.skip128dsDN = SkipBlockDN(128,128, downsample=True, skip_connect
ions=skip_connections)
                    ##  For the UP arm of the U:
                    self.bn1UP  = nn.BatchNorm2d(128)
                    self.bn2UP  = nn.BatchNorm2d(64)
                    self.skip64UP_arr = nn.ModuleList()
                    for i in range(self.depth):
                        self.skip64UP_arr.append(SkipBlockUP(64, 64, skip_connections=ski
p_connections))
                    self.skip64usUP = SkipBlockUP(64, 64, upsample=True, skip_connections
=skip_connections)
                    self.skip128to64UP = SkipBlockUP(128, 64, skip_connections=skip_conne
ctions )
                    self.skip128UP_arr = nn.ModuleList()
                    for i in range(self.depth):
                        self.skip128UP_arr.append(SkipBlockUP(128, 128, skip_connections=
skip_connections))
                    self.skip128usUP = SkipBlockUP(128,128, upsample=True, skip_connectio
ns=skip_connections)
                    self.conv_out = nn.ConvTranspose2d(64, 1, 3, stride=2,dilation=2,outp
ut_padding=1,padding=2) #remember to change the output channel to 3 later

    def forward(self, x):
                    ##  Going down to the bottom of the U:
```

```
                    x = nn.MaxPool2d(2,2)(nn.functional.relu(self.conv_in(x)))
                    for i,skip64 in enumerate(self.skip64DN_arr[:self.depth//4]):
                        x = skip64(x)

                    num_channels_to_save1 = x.shape[1] // 2
                    save_for_upside_1 = x[:,:num_channels_to_save1,:,:].clone()
                    x = self.skip64dsDN(x)
                    for i,skip64 in enumerate(self.skip64DN_arr[self.depth//4:]):
                        x = skip64(x)
                    x = self.bn1DN(x)
                    num_channels_to_save2 = x.shape[1] // 2
                    save_for_upside_2 = x[:,:num_channels_to_save2,:,:].clone()
                    x = self.skip64to128DN(x)
                    for i,skip128 in enumerate(self.skip128DN_arr[:self.depth//4]):
                        x = skip128(x)

                    x = self.bn2DN(x)
                    num_channels_to_save3 = x.shape[1] // 2
                    save_for_upside_3 = x[:,:num_channels_to_save3,:,:].clone()
                    for i,skip128 in enumerate(self.skip128DN_arr[self.depth//4:]):
                        x = skip128(x)
                    x = self.skip128dsDN(x)
                    ## Coming up from the bottom of U on the other side:
                    x = self.skip128usUP(x)
                    for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
                        x = skip128(x)
                    x[:,:num_channels_to_save3,:,:] =  save_for_upside_3
                    x = self.bn1UP(x)
                    for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
                        x = skip128(x)
                    x = self.skip128to64UP(x)
                    for i,skip64 in enumerate(self.skip64UP_arr[self.depth//4:]):
                        x = skip64(x)
                    x[:,:num_channels_to_save2,:,:] =  save_for_upside_2
                    x = self.bn2UP(x)
                    x = self.skip64usUP(x)
                    for i,skip64 in enumerate(self.skip64UP_arr[:self.depth//4]):
                        x = skip64(x)
                    x[:,:num_channels_to_save1,:,:] =  save_for_upside_1
                    x = self.conv_out(x)
                    return x
```

Below is the code that was used for Combined Loss, training and evaluation of the outputs:

```
class MSEPlusDiceLoss1(nn.Module):
    def __init__(self, alpha=0.5, epsilon=1e-2):
        super(MSEPlusDiceLoss1, self).__init__()
        self.alpha = alpha
        self.epsilon = epsilon
```

```
    def forward(self, output, mask_tensor):
        total_loss = 0
        composite_loss = torch.zeros(1, 4, requires_grad=True)
        dice_loss = torch.zeros(1, 4, requires_grad=True)
        mse_loss = torch.zeros(1, 4, requires_grad=True)

        for idx in range(min(output.shape[0], mask_tensor.shape[0])):
            mask = mask_tensor[idx, 0, :, :]
            output_mask = output[idx, 0, :, :]

            # Dice Loss
            if torch.sum(mask)+torch.sum(output_mask)>0:
                intersection = torch.sum(output_mask * mask)
                union = torch.sum(output_mask) + torch.sum(mask) + self.epsilon
                dice_coefficient = (2. * intersection + self.epsilon) / union
                dice_loss_copy = dice_loss.clone()
                dice_loss_copy[0,idx] = 1 - dice_coefficient
                dice_loss = dice_loss_copy

            if torch.sum(mask)>0 and torch.sum(output_mask)>0:
                # MSE Loss
                mse_loss_copy = mse_loss.clone()
                mse_loss_copy[0, idx] = F.mse_loss(output[idx], mask_tensor[idx])
                mse_loss = mse_loss_copy

            if torch.sum(mse_loss)>0 and torch.sum(dice_loss)>0:
                # Composite Loss
                composite_loss_copy = composite_loss.clone()
                composite_loss_copy[0, idx]= torch.sum(mse_loss) + 30*torch.sum(dice_los
s)
                composite_loss = composite_loss_copy

        average_loss = torch.sum(composite_loss) / min(output.shape[0], mask_tensor.shape
[0])

        return average_loss
```

This is the code that runs the training, implements the SGD optimizer and performs the backward propagation of the gradient for the network parameters:

```
# The training function is a slightly modified version of the function used in the DL Stu
dio library by Prof. Kak.

def run_code_for_training_for_semantic_segmentation(net, data_loader, save_path):
                net = copy.deepcopy(net)
                net = net.to(torch.device("cpu"))
                criterion1 = MSEPlusDiceLoss1(4, alpha=0.5, epsilon = 1e-3)
                optimizer = optim.SGD(net.parameters(),
                             lr=1e-4, momentum=0.9)
                start_time = time.perf_counter()
                running_loss = []
```

```
                for epoch in range(10):
                    print("")
                    running_loss_segmentation = 0.0
                    for i, data in enumerate(data_loader):
                        mask_tensor, im_tensor = data
                        # print(im_tensor.shape)
                        im_tensor   = im_tensor.to(torch.device("cpu"))
                        mask_tensor = mask_tensor.to(torch.device("cpu"))
                        mask_tensor = mask_tensor.type(torch.FloatTensor)
                        optimizer.zero_grad()
                        output = net(im_tensor)
                        segmentation_loss= criterion1(output, mask_tensor)
                        segmentation_loss.requires_grad
                        output.requires_grad
                        im_tensor.requires_grad = True
                        mask_tensor.requires_grad = True
                        segmentation_loss.backward()
                        optimizer.step()
                        running_loss_segmentation += segmentation_loss.item()
                        if i%100==99:
                            current_time = time.perf_counter()
                            elapsed_time = current_time - start_time
                            avg_loss_segmentation = running_loss_segmentation / float(10
0)
                            print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   Combi
ned loss: %f" % (epoch+1, 10, i+1, elapsed_time, avg_loss_segmentation ))
                            running_loss.append(avg_loss_segmentation)
                            running_loss_segmentation = 0.0
                print("\nFinished Training\n")
                return running_loss
                torch.save(net.state_dict(), '/Users/avnishkanungo/Desktop/coco-dataset/t
rain2017/hw7_model_save/save_model_loss_1')
```

The below is the code to run the evaluation, the evaluation is just passing the images from the test data into the saved model to generate outputs which we can view and evaluate the overall performance of the saved model:

```
# The test function is modified version of the function used in the DL Studio library by
Prof. Kak.

def run_code_for_testing_semantic_segmentation( net, dataloader, model_path):
                net.load_state_dict(torch.load(path))
                batch_size = 4
                image_size = [256,256]
                # max_num_objects = self.max_num_objects
                with torch.no_grad():
                    for i, data in enumerate(dataloader):
                        mask_tensor,im_tensor = data
                        outputs = net(im_tensor)
                        fig = plt.figure(figsize=(10, 7))
                        rows = 2
                        columns = 2
```

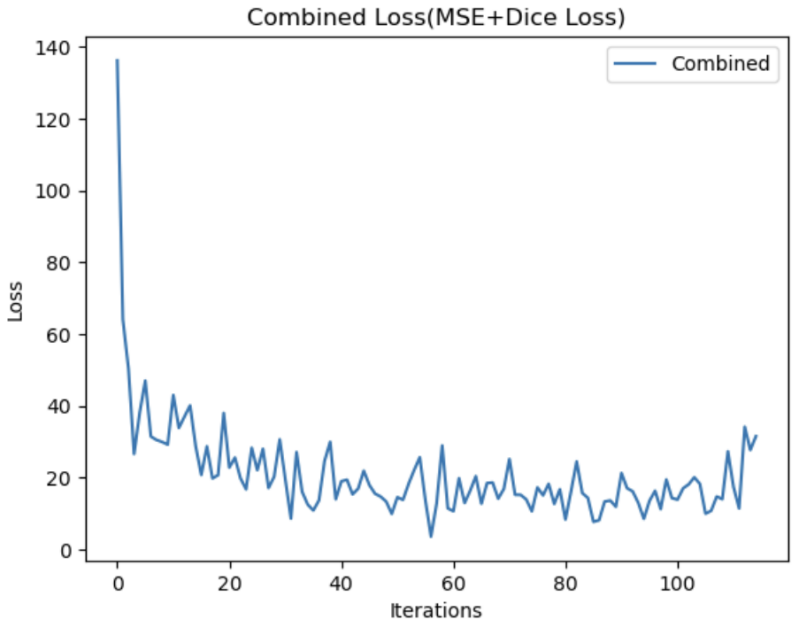
```
                        fig.add_subplot(rows, columns, 1)

                        # showing image
                        plt.imshow(outputs[i-1].permute(1,2,0))
                        plt.axis('off')
                        plt.title("Output Image")

                        # Adds a subplot at the 2nd position
                        fig.add_subplot(rows, columns, 2)

                        # showing image
                        plt.imshow(im_tensor[i-1].permute(1,2,0))
                        plt.axis('off')
                        plt.title("Input Image")

                        # Adds a subplot at the 3rd position
                        fig.add_subplot(rows, columns, 3)

                        # showing image
                        plt.imshow(mask_tensor[i-1].permute(1,2,0))
                        plt.axis('off')
                        plt.title("Input Mask")
```

Implementation of the above functions and classes:

```
model = mUnet(skip_connections=True, depth=16)
combined_loss = run_code_for_training_for_semantic_segmentation(model, train_dataloader,
'/Users/avnishkanungo/Desktop/coco-dataset/train2017/hw7_model_save/saved_model_combine
d')
run_code_for_testing_semantic_segmentation(model, test_dataloader, '/Users/avnishkanungo/
Desktop/coco-dataset/train2017/hw7_model_save/saved_model_combined')
```

Output from Training(apha=0.5):

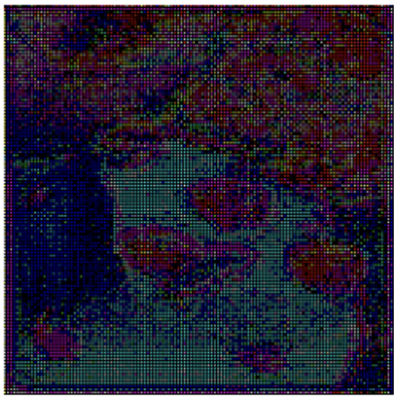
```
[epoch=1/10, iter= 100  elapsed_time=298 secs]   Combined loss: 136.794047
[epoch=2/10, iter= 100  elapsed_time=683 secs]   Combined loss: 47.828275
[epoch=3/10, iter= 100  elapsed_time=1100 secs]   Combined loss: 41.445489
[epoch=4/10, iter= 100  elapsed_time=1509 secs]   Combined loss: 66.523172
[epoch=5/10, iter= 100  elapsed_time=1927 secs]   Combined loss: 41.076811
[epoch=6/10, iter= 100  elapsed_time=2355 secs]   Combined loss: 40.720104
[epoch=7/10, iter= 100  elapsed_time=2803 secs]   Combined loss: 37.879594
[epoch=8/10, iter= 100  elapsed_time=3231 secs]   Combined loss: 34.369073
[epoch=9/10, iter= 100  elapsed_time=3622 secs]   Combined loss: 41.578545
[epoch=10/10, iter= 100  elapsed_time=3997 secs]   Combined loss: 31.905929
Finished Training
```



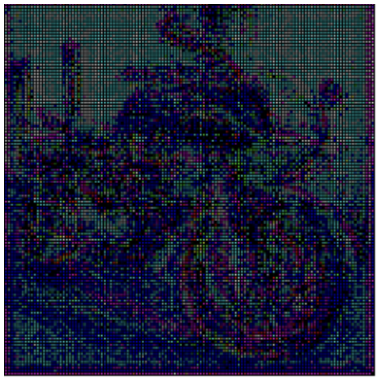
Output on 6 random images from the test data:










Conclusion:

From the above we can see that, the model is performing adequately and is able to recreate the input images using the mUNet and we can see that the pixel level segmentation mask created from the network, when upscaled and passed through transpose convolutions is able to recreate the images to some extent. The loss function used here is the Combined(MSE+Dice) Loss function which has an overall better performance as compared to just MSE or just Dice loss.