HOMEWORK 7

Alexandre Olive Pellicer

3. Programming tasks

3.1. Execute the semantic segmentation.py script and evaluate both the training loss and the test results. Provide a brief write-up of your understanding of mUnet and how it carries out semantic segmentation of an image. By "evaluate" we mean just record the running losses during training. One of the most commonly used tools for evaluating a semantic segmentation network is through the IoU loss. If you wish, you can write that code yourself. But that is not required for this homework.

Training loss:

The following plot contains the "running_loss_segmentation" every 500 iterations. The train has been done during 6 epochs and a batch_size of 4 as done in the original code.

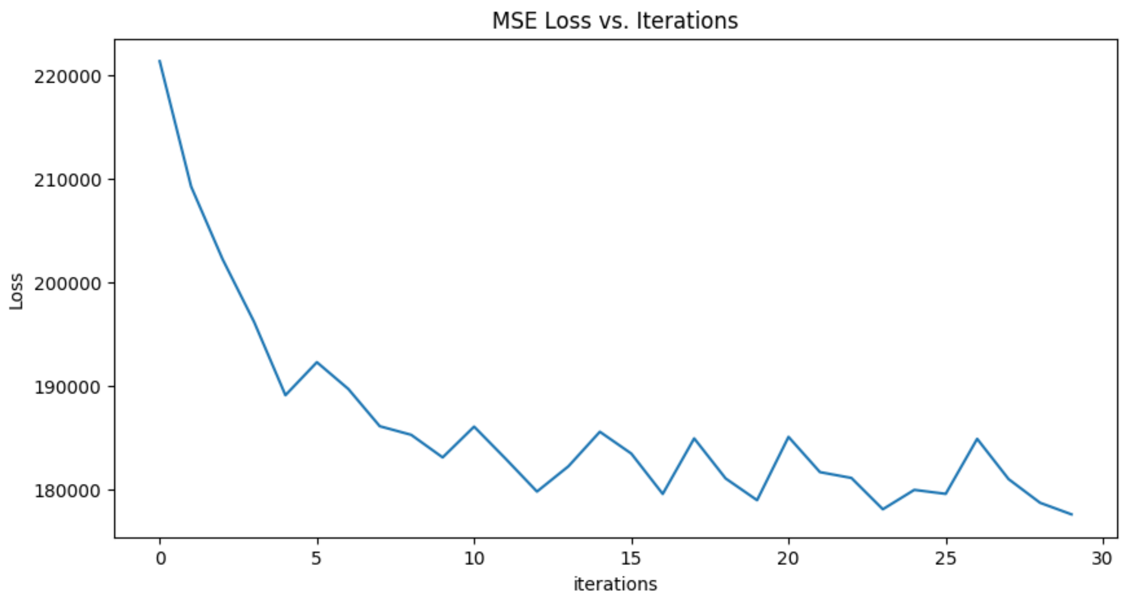
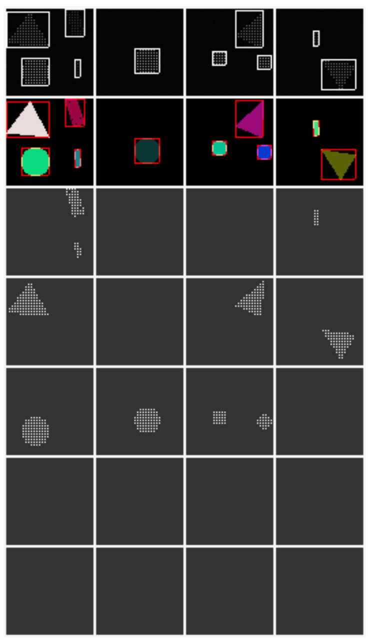


Fig 1: Plot of the MSE Loss of the PurdueShapes5MultiObject dataset

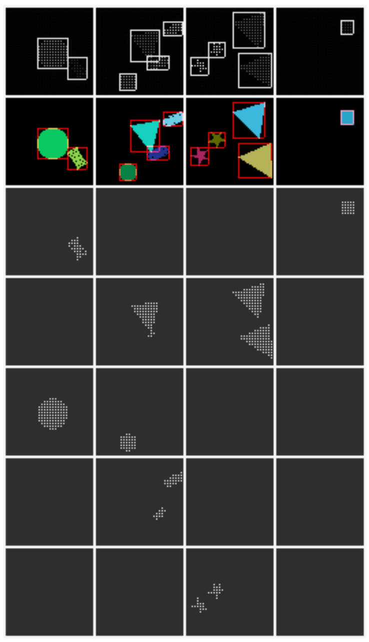
We can clearly see how the loss decreases converging towards a minimum although it does some fluctuations. It looks like by increasing the number of epochs the loss wouldn't be reduced much more since it looks like it is converging to a minimum.

Test results:

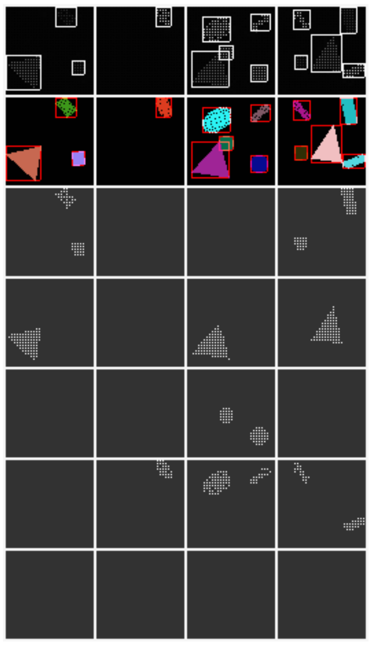
In order to do a qualitative analysis of the obtained results, we have plotted the testing images from the batch number 1, 50, 100, 150 and 200 with their predicted mask. These are the obtained results:



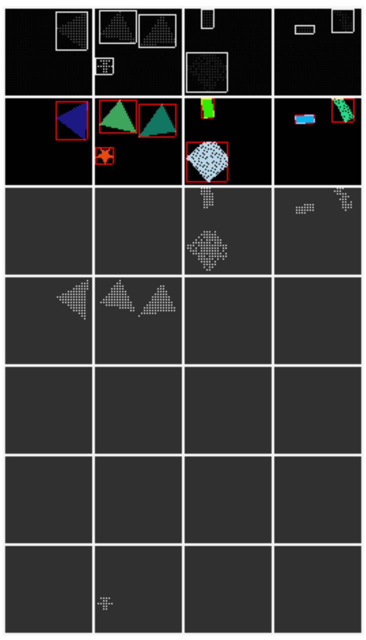
Batch 1



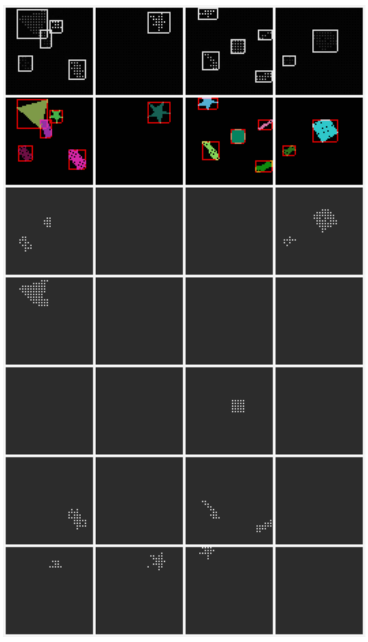
Batch 50



Batch 100



Batch 150



Batch 200

Fig 2: Predicted masks obtained using the PurdueShapes5MultiObject testing dataset

In general, the performance is good although we can see that there isn't a lot of precision when it comes to specify correctly the boundaries. This also generates some confusion when it comes to classify the detected objects, for example, between ovals and rectangles. Looking at each batch we can say:

- Batch 1: we can see that the ovals from the first and fourth images (from left to right) aren't detected correctly since they are classified as rectangles. Their mask doesn't correctly fit with their real shape.
- Batch 50: in the second image we can see that we are misclassifying a rectangle as oval and that the boundaries of the mask corresponding to the triangle are not accurate.
- Batch 100: in the third image we can see that the boundaries of the mask of the small oval are not accurate.
- Batch 150: in this case we can see that the performance in the 4 images is good. There are no relevant mistakes.
- Batch 200: in this case we can see how in the third image we are not being able to detect the rectangle from the top right of the image. There is no mask associated to it.

Brief write-up of your understanding of mUnet and how it carries out semantic segmentation of an image:

Since the output of the mUnet is compared with the ground truth mask to compute the loss, we are forcing the network to predict the mask of the input images. Masks are composed of 5 channels each of them containing the "mask/shape" of the instances contained in the image corresponding to each of the 5 classes "rectangle, triangle, disk, oval, star".

About the structure of the mUnet, we first have a convolutional layer that increases the number of the input image from 3 to 64. By doing this we are creating an embedding vector for each pixel containing the features that will be learnt during training. Afterwards a concatenation of several instances of "SkipBlockDN" are called. By doing this the image size is down sampled from 64x64 (original input size) to 16x16 (size of the feature map in the bottle neck of the mUnet) and the number of channels is increased from 64 to 128.

Furthermore, there are 3 skip connections that will be used to pass the detailed spatial information from the encoder to the decoder. These skip connections take half of the channels at 3 different levels (feature maps) of the encoder and add them (in terms of substitution) at the corresponding 3 levels (feature maps) of the decoder.

The decoder is composed of a concatenation of several instances of "SkipBlockUP" with which the feature map from the bottle neck goes from 128 channels and a size of 16x16 to 64 channels and a size of 64x64. A final convolution reduces the number of channels to 5 which will be the 5 channels that characterize each mask as mentioned above.

To sum up, for segmentation, the encoder-decoder structure is used so that the encoder gradually reduces the spatial dimensions of the input image to capture high-level features, while the decoder gradually up samples the feature maps to produce a segmentation map of the same size as the input image. Skip connections are used to pass the detailed spatial information from the encoder to the decoder.

3.2. The run_code_for_training_for_semantic_segmentation function of the SemanticSegmentation class in DLStudio uses just the MSE loss. MSE loss may not adequately capture the subtleties of segmentation boundaries. To this end, we will implement our own Dice loss and augment it with MSE loss and compare it against vanilla MSE.

3.3. What follows is a code snippet to help you create your own implemenation for Dice Loss. Make sure you set required_grad=True wherever necessary to ensure backpropagation, therefore, enabling model learning.

Find the implementation of the dice loss in the code section at the bottom of the document.

3.4. Plot the best- and the worst-case training-loss vs. iterations using just the MSE loss, just the Dice Loss and a combination of the two. Provide insights into potential factors contributing to the observed variations in performance.

The following plots contain the "running_loss_segmentation" every 500 iterations for the MSE loss, the Dice loss and the combination of the MSE and Dice loss. In all cases the train has been done during 6 epochs and a batch_size of 4 as done in the original code.

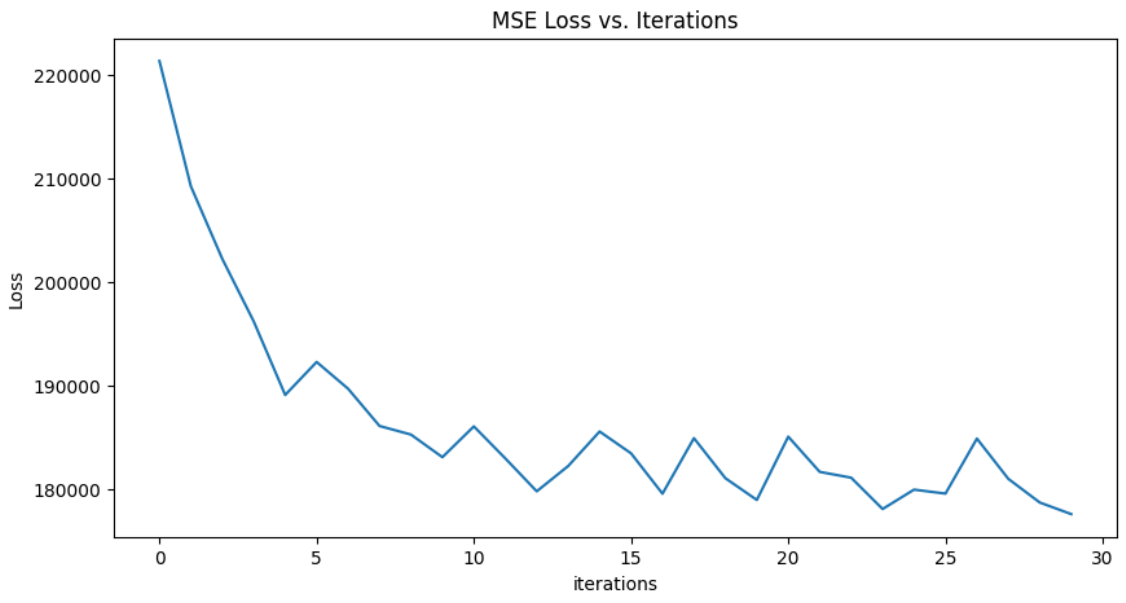


Fig 3: Plot of the MSE Loss of the PurdueShapes5MultiObject dataset

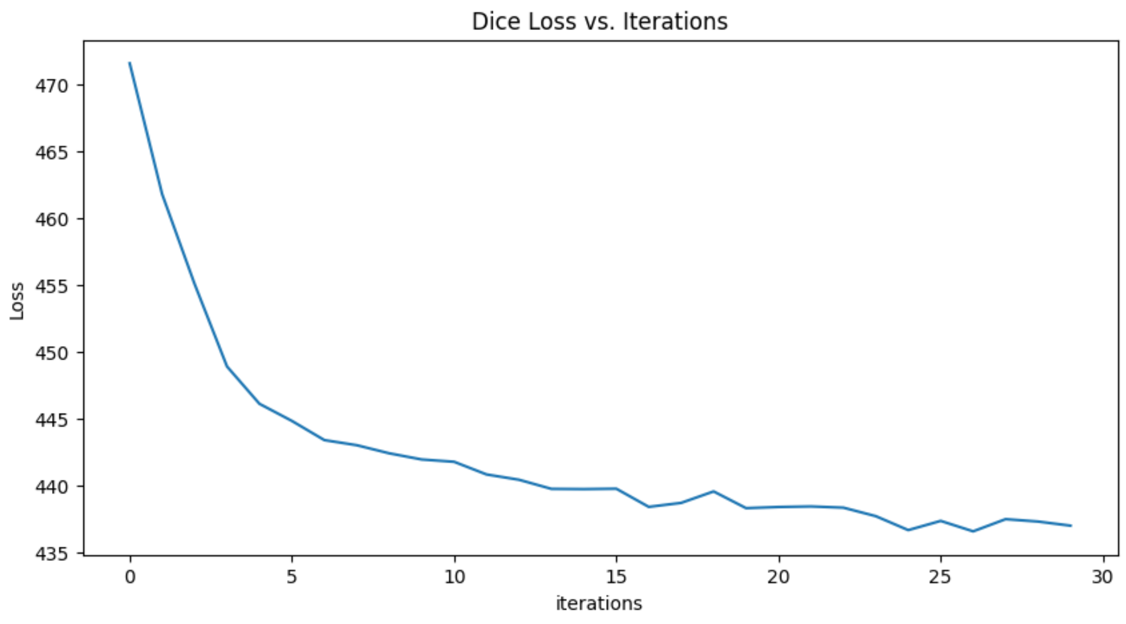
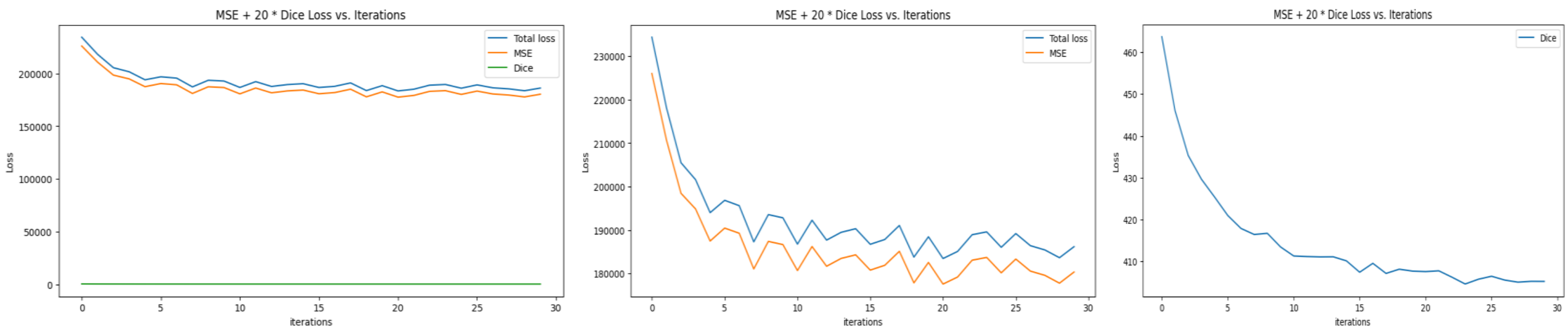


Fig 4: Plot of the Dice Loss of the PurdueShapes5MultiObject dataset

As can be seen, the values of the MSE loss are much bigger than the values of the Dice loss. Thus, to give to both of them relevance and following the advice from the TA Akshita Kamsali in Piazza we have given extra weight to the Dice loss with values 20, 30 and 40 so that the total loss has been computed 3 different times as:

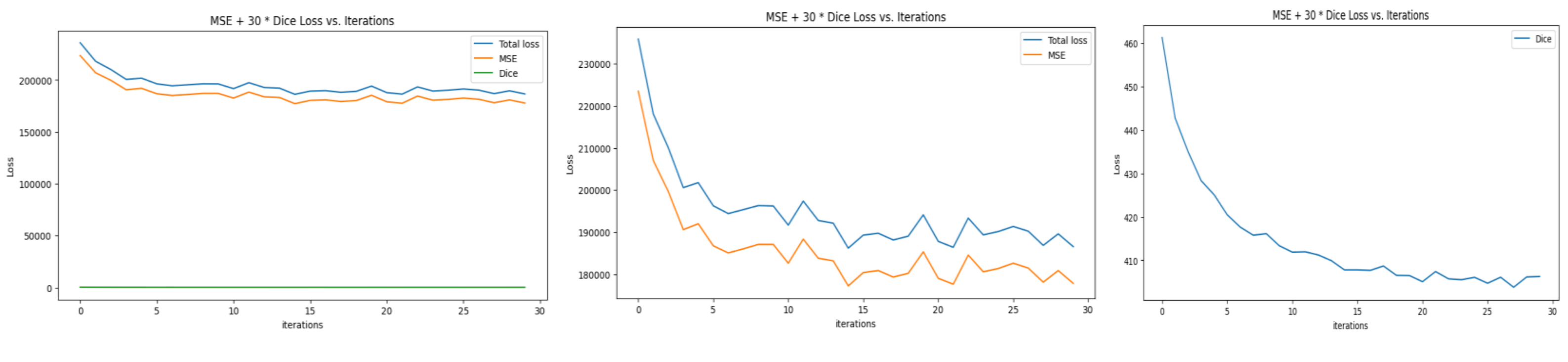
Total loss = MSE loss + {20, 30, 40} * Dice loss

These are the obtained plots:



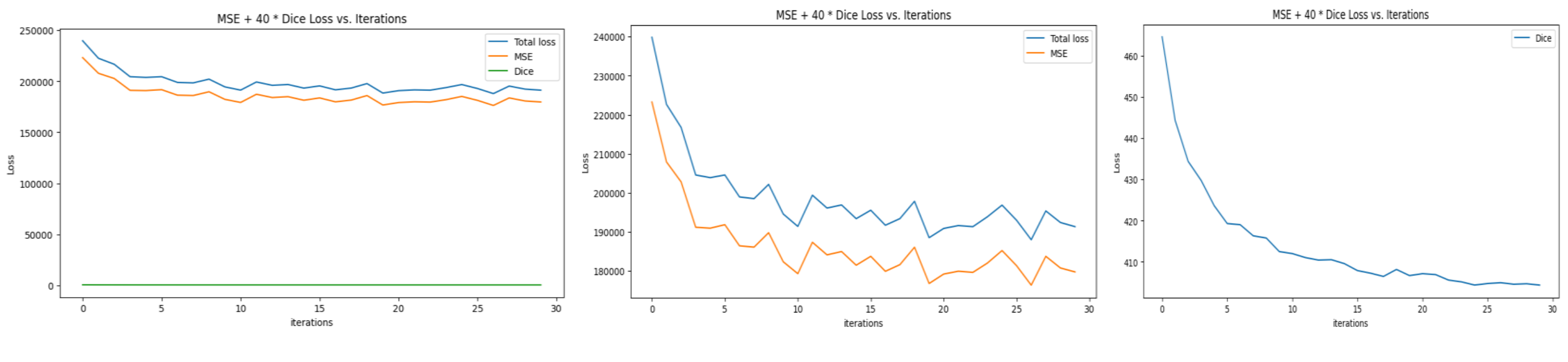
(a) (b) (c)

Fig 5: (a) Plot of the Total loss = (MSE loss + 20 * Dice loss) of the PurdueShapes5MultiObject dataset. (b) and (c) represent the same data as in (a) but in 2 different plots so that the loss curve can be seen clearly.



(a) (b) (c)

Fig 6: (a) Plot of the Total loss = (MSE loss + 30 * Dice loss) of the PurdueShapes5MultiObject dataset. (b) and (c) represent the same data as in (a) but in 2 different plots so that the loss curve can be seen clearly.



(a) (b) (c)

Fig 7: (a) Plot of the Total loss = (MSE loss + 40 * Dice loss) of the PurdueShapes5MultiObject dataset. (b) and (c) represent the same data as in (a) but in 2 different plots so that the loss curve can be seen clearly.

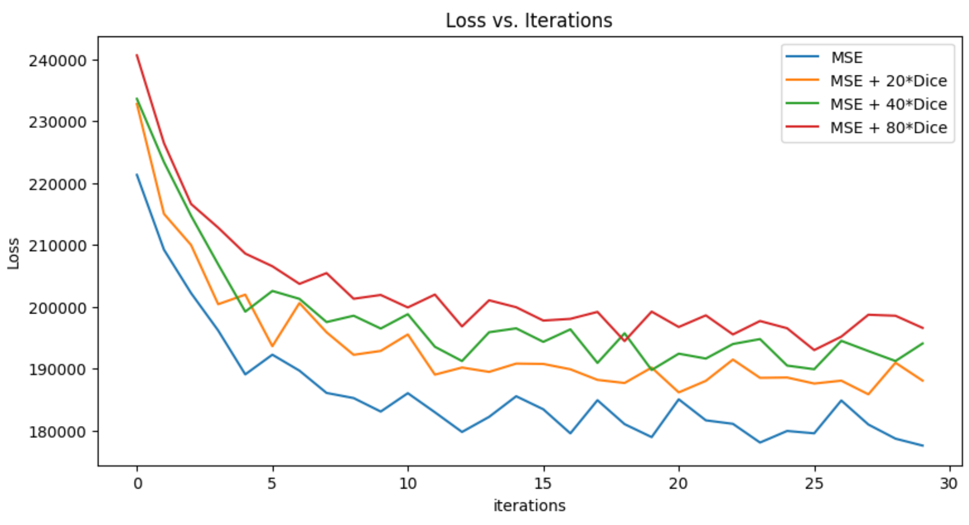


Fig 8: Plot containing the loss curves for MSE alone, MSE + 20* Dice, MSE + 30*Dice and MSE + 40* Dice

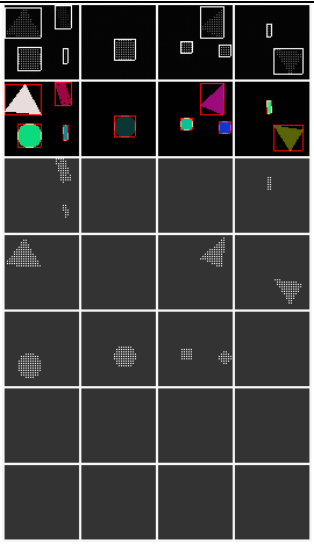
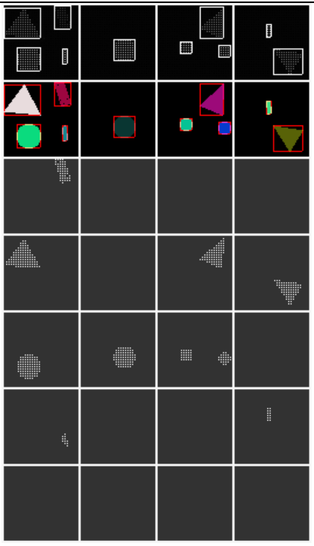
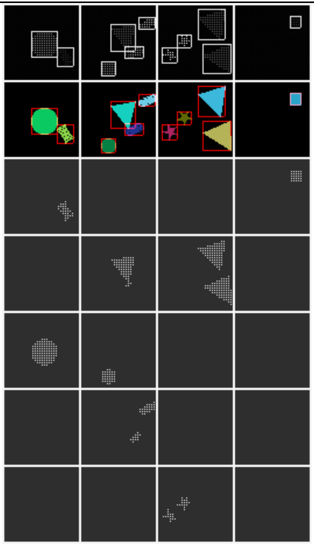
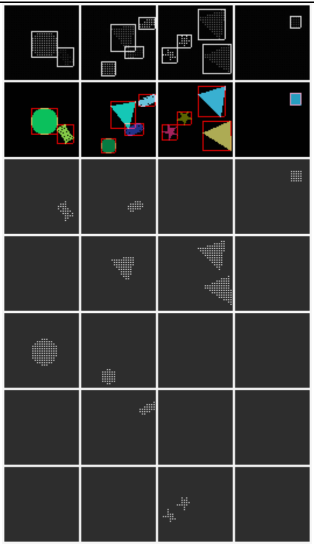
Since the MSE loss and the Dice loss are at different orders of magnitude, it is difficult to obtain clear insights just by looking at the plots of the losses. For this reason, it is better to evaluate qualitatively the obtained masks as it is done in the following section.

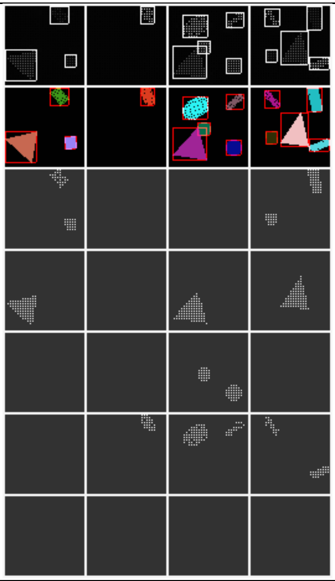
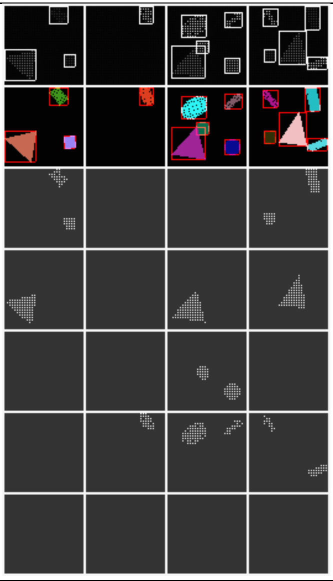
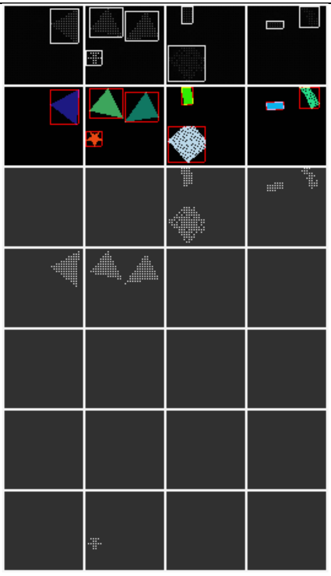
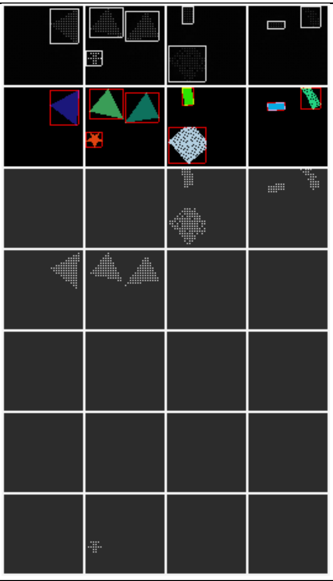
Nevertheless, we can also obtain some conclusions from the showed plots. For example, we can see that the Dice loss converges towards a lower value when using it combined with the MSE loss than when using it alone. This behavior happens when using scale_factor = 20, 30, 40. This is because the MSE loss helps to get a general shape of the mask while the dice loss is useful to define accurate boundaries of the mask. When using the Dice loss alone without the MSE loss it is more difficult to reach smaller values.

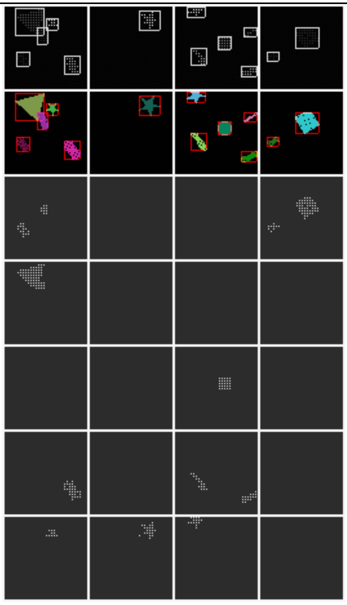
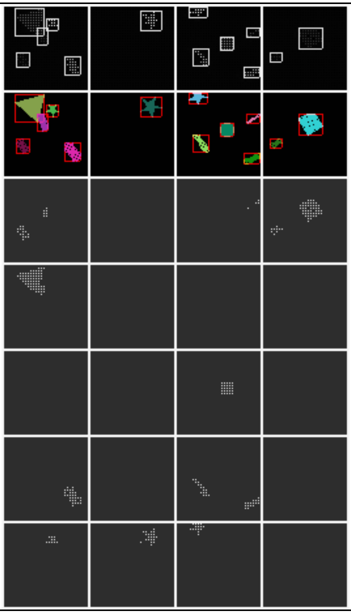
We can also get some insights from the plot shown in Figure 8. We see that the MSE loss alone reaches a smaller value compared to the combined loss. This does not give us much information since the combined losses contain information about the overlap of the predicted and ground truth masks so that a higher value does not mean a worst performance. In some way, by comparing the MSE loss with the combined losses we could say that we are comparing pears with apples. Nevertheless, comparing the 3 combined losses we see that the minimum value is achieved when using a scale factor of 20.

3.5. State your qualitative observations on the model test results for MSE loss vs. Dice+MSE loss.

Comparing the results obtained by scaling the Dice loss by a factor of 20, 30 and 40 we have seen that the best result is achieved when we use scale_factor = 20. For this reason, we directly compare the results obtained when using the MSE loss alone and the results obtained when using the combined loss MSE + 20*Dice loss.

Batch 1		Batch 50	
MSE loss	MSE loss + 20 * Dice loss	MSE loss	MSE loss + 20 * Dice loss
			
Comments: We can clearly see here how thanks to the Dice loss the ovals from the firs and fourth image (from left to right) are correctly classified as "ovals" when using the MSE + Dice loss and not as rectangles as was happening when using only the MSE loss		Comments: We can clearly see how in the second image, when using the MSE + Dice loss we are correctly classifying a rectangle as rectangle and not as oval. Furthermore the mask of the triangle in this same image is more accurate	

Batch 100		Batch 150	
MSE loss	MSE loss + 20 * Dice loss	MSE loss	MSE loss + 20 * Dice loss
			
Comments: We can see that with the MSE + Dice loss we are achieving a more accurate mask for one of the ovals from the third image		Comments: In this particular case we can see that for the 4 images the performance is almost the same	

Batch 200	
MSE loss	MSE loss + 20 * Dice loss
	
Comments: In this case we can see how when using the MSE + Dice loss we are able to get partially part of the mask of one of the rectangles from the third image that when using only the MSE loss was not detected	

In general, we can see that when using the MSE + Dice loss the performance has been improved in terms of being more accurate with the boundaries of the masks. Furthermore, we see that some thin objects that weren't detected when using the MSE loss alone, are detected and created a mask for them when using the MSE + Dice loss.

4. Extra credit:

Creating the dataset:

We create a training and testing dataset using the COCO dataset and its API following the strategy like the strategy used in previous homeworks. In this case, we pick the images containing instances of “motorcycle”, “dog” or “cake” that accomplished the following condition:

- Only contain a single object instance of at least 200 × 200 bounding box.

In order to create a dataset with the same structure as the PurdueShapes5MultiObject dataset, we created a dictionary with key “index of image” and value a dictionary containing the following information:

```
dataset[idx] = {
            0: r,
            1: g,
            2: b,
            3: final_mask,
            4: bbox_dict
        }
```

Where r, g and b contain a column array with the value of the pixels of the images for each of the RGB components respectively. Final_mask is a tensor with the same size as the image and with 3 channels. The first channel contains the mask of a motorcycle in case it exists with value 50 to identify the mask. The second channel contains the mask of a dog in case it exists with value 100 to identify the mask. The third channel contains the mask of a cake in case it exists with value 150 to identify the mask. Bbox_dict contains the bounding boxes of the instances in the image with a vector [x1, y1, x2, y2] where “1” indicates top left and “2” indicates bottom right. It must be said that, as has been done with the images, masks and bounding boxes have been scaled to 256x256. The code used to create the dataset can be found at the end of the document.

The obtained dataset has the following characteristics:

- Length training dataset: 2606
- Length testing dataset: 119

Repeating the steps from section 3 for our portion of the COCO dataset:

4.1. Execute the semantic segmentation.py script and evaluate both the training loss and the test results. Provide a brief write-up of your understanding of mUnet and how it carries out semantic segmentation of an image. By “evaluate” we mean just record the running losses during training. One of the most commonly used tools for evaluating a semantic segmentation network is through the IoU loss. If you wish, you can write that code yourself. But that is not required for this homework.

In this case we have used a batch_size of 4 and trained for 50 epochs. In the loss plots from this section, we have plot the “running loss” every 50 iterations (i.e. the x axis represent 50 iterations).

Training loss:

This is the training Loss using only MSE loss:

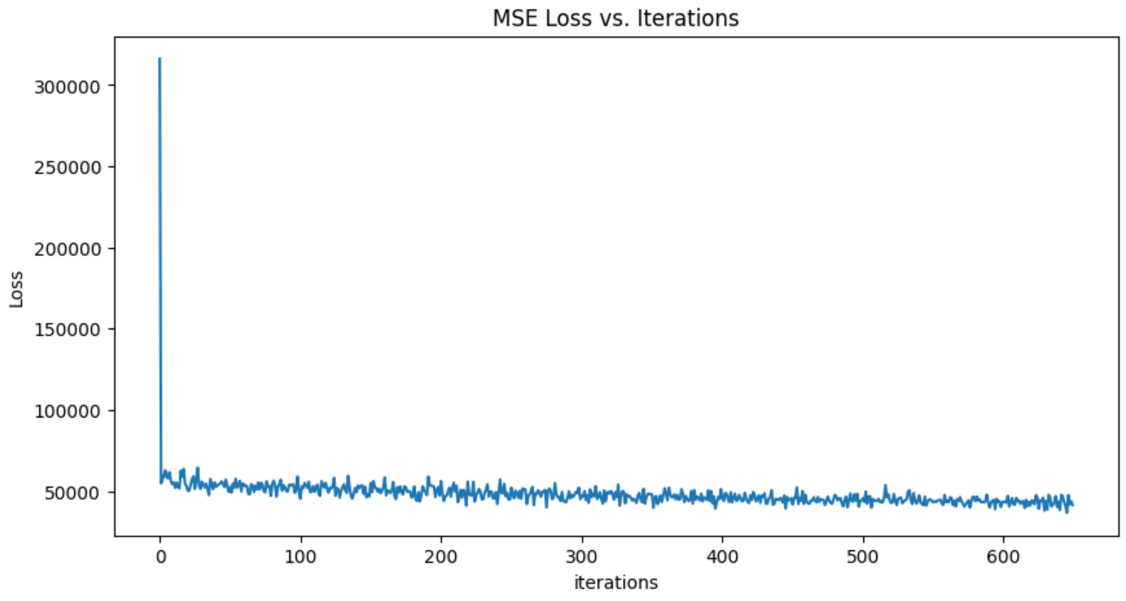
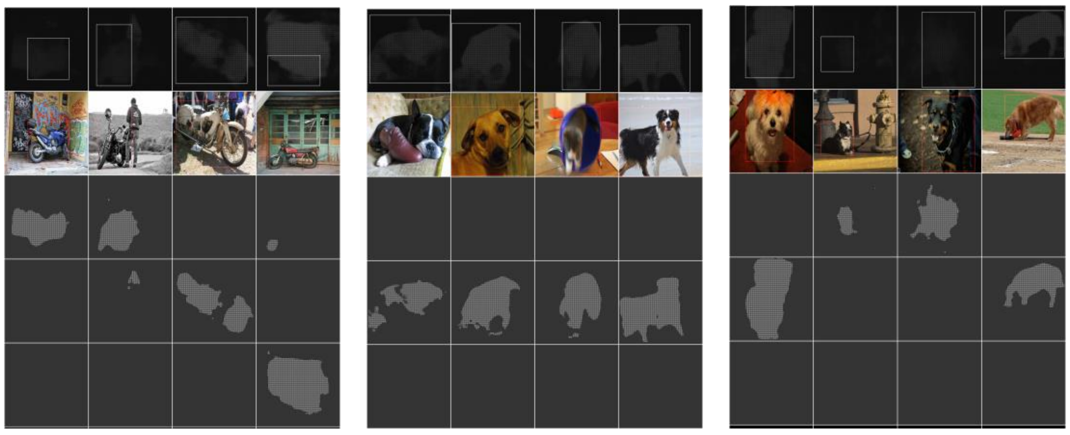


Fig 9: Plot of the MSE Loss using our portion of the COCO dataset

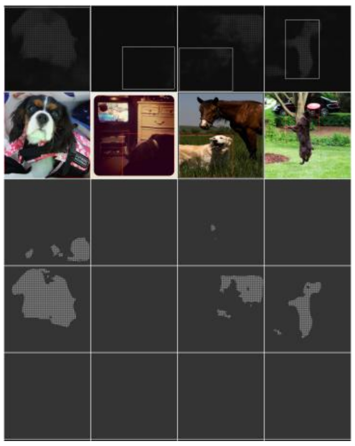
We can clearly see how the loss decreases converging towards a minimum although it does some fluctuations. It looks like by increasing the number of epochs the loss wouldn't be reduced much more since it looks like it is converging to a minimum.

Testing results:

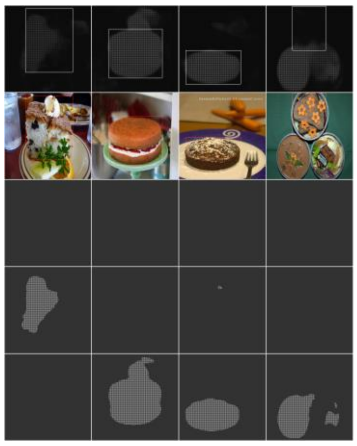
These are some results obtained using the MSE loss:



Group 1 Group 2 Group 3



Group 4



Group 5

Fig 10: Predicted masks obtained using our portion of the COCO testing dataset

First of all, it must be said that the first channel of the mask contains the mask of motorcycles, the second channel contains the mask of dogs and the third channel contains the mask of cakes. From the obtained results we can see that there are some mistakes in the performance. On the one hand, some instances are misclassified (i.e. for example, a motorcycle is classified as a cake). On the other hand, the predicted masks are not very accurate. It must be stated that in this case we are working with real world images that contain several objects and textures. Thus, the images from the COCO dataset are more complex than the images from the PurdueShapes5MultiObject and a decrease in the performance can be expected. Looking at each of the groups we can obtain the following observations:

- Group 1: some masks are misclassified, and the shapes are not very accurate.
- Group 2: all the masks are correctly classified as dogs. From left to right, we can see that the first and third masks are not very accurate.
- Group 3: we can see that the performance in the first and last image is acceptable. The boundaries of the fourth mask aren't very accurate. The network is not capable of correctly identifying the masks for the second and third image.
- Group 4: for the first and fourth image the mask is built although the boundaries aren't very accurate. For the second image the dog is not detected and for the third image it looks like the network has confused the dog with the horse. This is something possible since instances of "dog" and "horse" share several features.
- Group 5: the second and third image create a mask of the cakes although the boundaries are not very accurate. The first and fourth images deserve some room for improvement.

New U-Net architecture:

Now we are working with images of size 256x256 that contain 1 instance of 3 different classes_ "dog", "cake" or "motorcycle". Since the mUnet given in the DLStudio is used to predict the masks for images of size 64x64 containing instances of 5 different classes, we need to do some modifications so that it can work with our new requirements.

The first change that is made is the number of channels that the output of the network will have. We changed it from 5 to 3 since now we are dealing with only 3 images.

Another change that we make is increasing the depth of the network by incrementing the number of down sample and up sample layers following the original structure from mUnet. One of the characteristics from a U-Net architecture is reducing the amount of information in the bottleneck so that only the relevant information is kept and the noisy information is removed so that the relation between the pixels of the input image is learnt as desired. In the original mUnet the input feature map is down sampled and increased the number of channels from 64x32x32 to 128x16x16. If we didn't increase the depth of the network, we would go from a 64x256x256 input to a 128x64x64 feature map in the bottleneck. The size of the feature map of the bottleneck might not be small enough to remove the noise. Thus, we add 2 more down sample and up sample layers with their respective skip connections so that the feature map in the bottleneck is of size 512x16x16.

4.2. The run_code_for_training_for_semantic_segmentation function of the SemanticSegmentation class in DLStudio uses just the MSE loss. MSE loss may not adequately capture the subtleties of segmentation boundaries. To this end, we will implement our own Dice loss and augment it with MSE loss and compare it against vanilla MSE.

4.3. What follows is a code snippet to help you create your own implemenation for Dice Loss. Make sure you set required_grad=True wherever necessary to ensure backpropagation, therefore, enabling model learning.

The dice loss used in this section is the same as presented in Section 3

4.4. Plot the best- and the worst-case training-loss vs. iterations using just the MSE loss, just the Dice Loss and a combination of the two. Provide insights into potential factors contributing to the observed variations in performance.

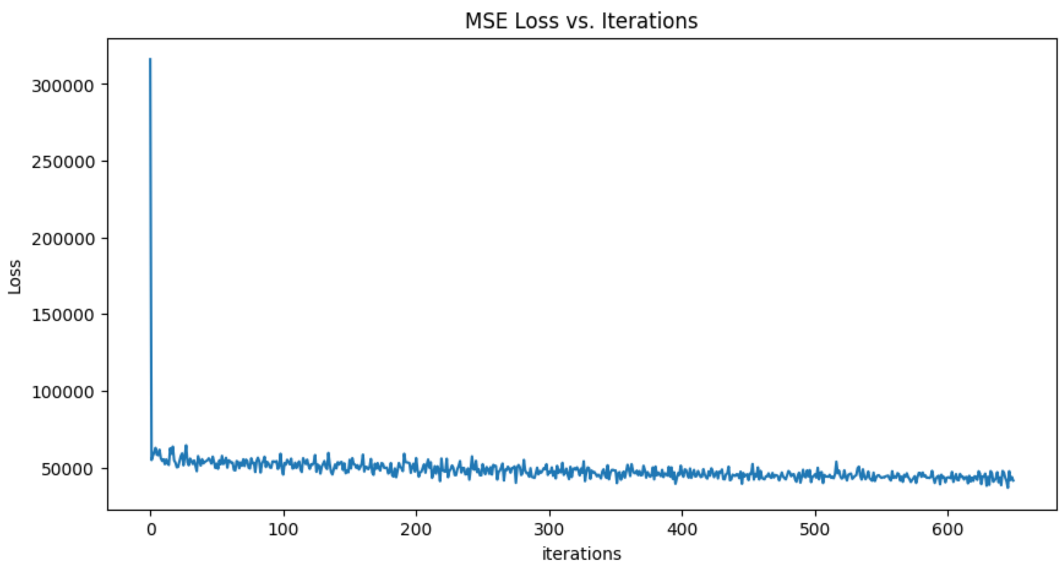


Fig 11: Plot of the MSE Loss using our portion of the COCO dataset

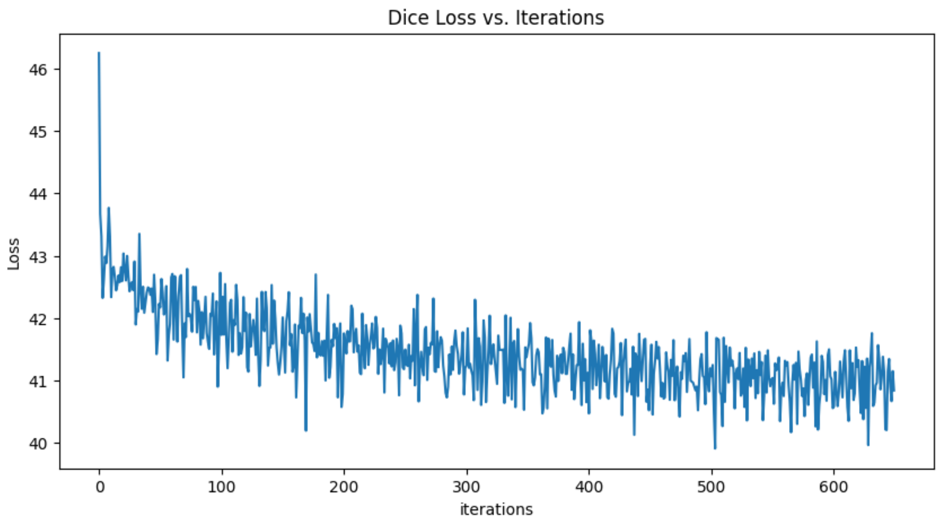


Fig 12: Plot of the Dice Loss using our portion of the COCO dataset

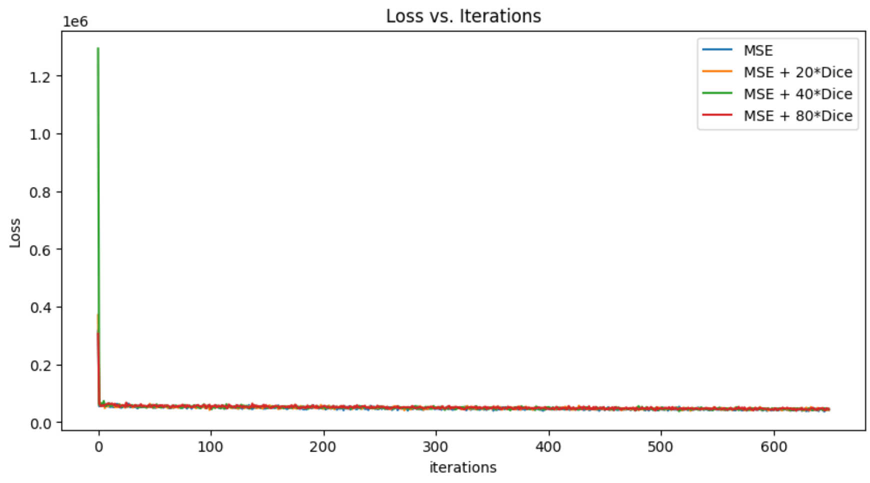


Fig 13: Plot containing the loss curves for MSE alone, MSE + 20* Dice, MSE + 40*Dice and MSE + 80* Dice

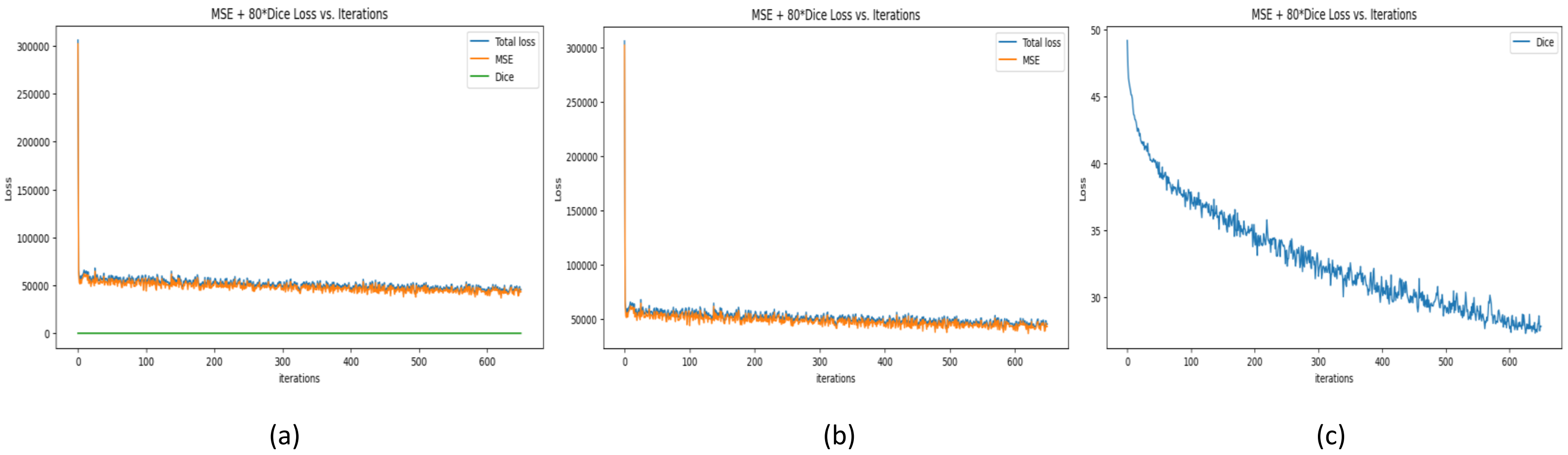


Fig 14: (a) Plot of the Total loss = (MSE loss + 80 * Dice loss) of our portion of the COCO dataset. (b) and (c) represent the same data as in (a) but in 2 different plots so that the loss curve can be seen clearly.

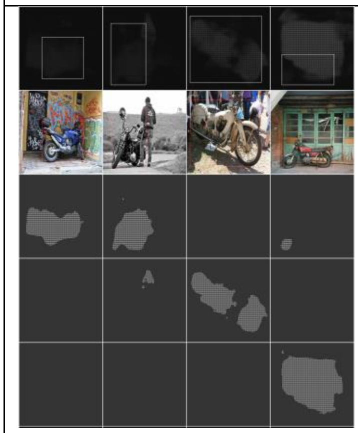
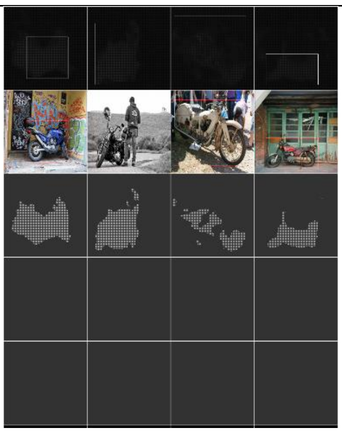
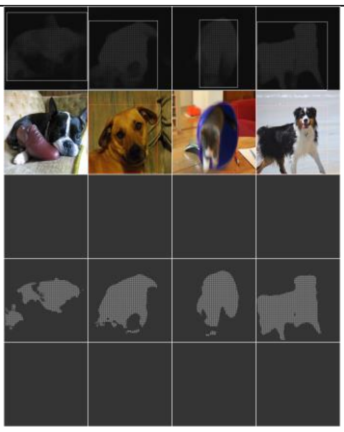
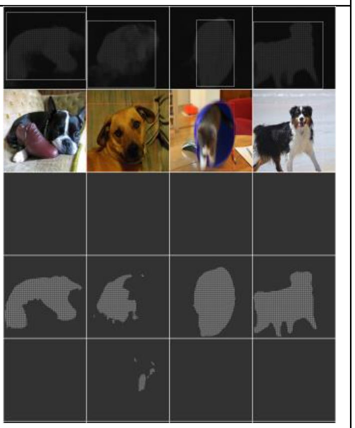
Since the MSE loss and the Dice loss are at different orders of magnitude, it is difficult to obtain clear insights just by looking at the plots of the losses. For this reason, it is better to evaluate qualitatively the obtained masks as it is done in the following section.

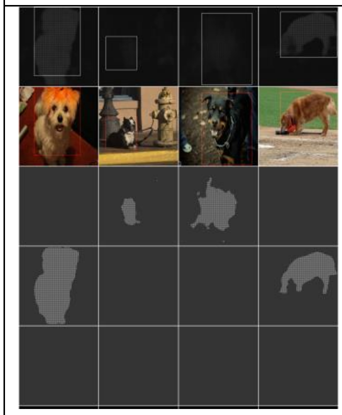
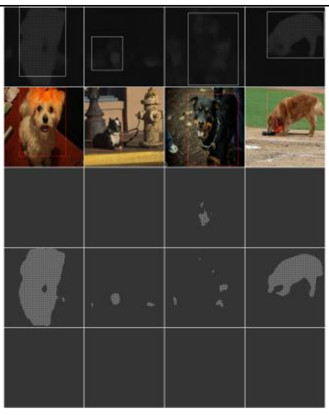
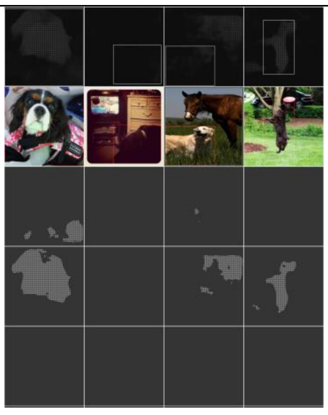
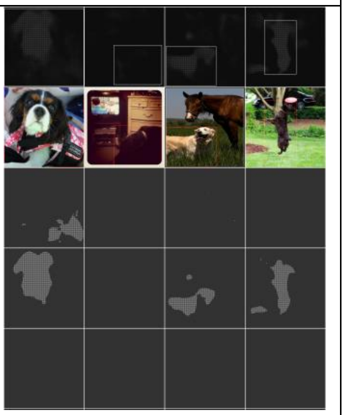
Nevertheless, we can also obtain some conclusions from the showed plots. For example, we can see that the Dice loss converges towards a lower value when using it combined with the MSE loss than when using it alone. This is because the MSE loss helps to get a general shape of the mask while the dice loss is useful to define accurate boundaries of the mask. When using the Dice loss alone without the MSE loss it is more difficult to reach smaller values.

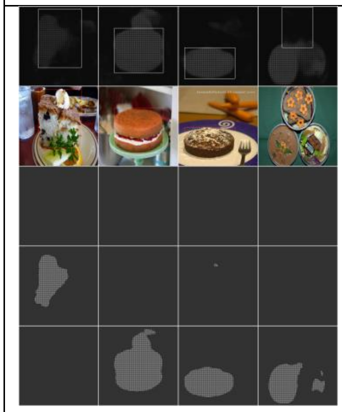
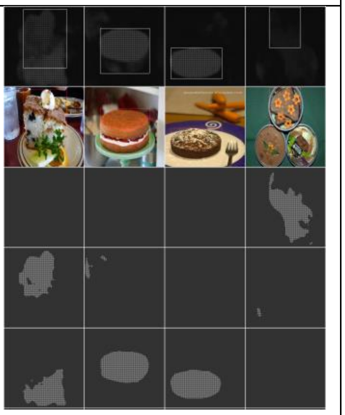
We can also get some insights from the plot shown in Figure 13. It can be seen that the performance when using a scale factor of 20, 40 or 80 is very similar. By doing a qualitative evaluation for each of these 3 different values of the scale factor, we saw that the best results were obtained when scale_factor = 80. That is the reason why we add Figure 14 with the plots of the loss when using a scale_factor=80 and why in the next section we also present the results for scale_factor=80.

4.5. State your qualitative observations on the model test results for MSE loss vs. Dice+MSE loss

Comparing the results obtained by scaling the Dice loss by a factor of 20, 40 and 80 we have seen that the best result is achieved when we use scale_factor = 80. For this reason, we directly compare the results obtained when using the MSE loss alone and the results obtained when using the combined loss MSE + 80*Dice loss.

Group 1		Group 2	
MSE loss	MSE loss + 80 * Dice loss	MSE loss	MSE loss + 80 * Dice loss
			
Comments: In this case we can clearly see an improvement when using the MSE + Dice loss. All the masks are classified as motorcycles and the shapes are more accurate although they are not perfect yet.		Comments: We can see that the performance has improved when using the MSE+Dice loss for the boundaries of the masks for the first, third and fourth images. Nevertheless, we can note that the mask of the second image is worst compared to the one obtained when using only the MSE loss.	

Group 3		Group 4	
MSE loss	MSE loss + 80 * Dice loss	MSE loss	MSE loss + 80 * Dice loss
			
Comments: We can see a slightly performance in the boundaries of the mask of the first and third image when using the MSE + Dice loss. For the second and third image we are not reconstructing any mask.		Comments: Using the MSE + Dice loss we are still not being able to predict a mask for the dog from the second image. The performance for the first image hasn't changed a lot. In the third image we can see that with the MSE + Dice loss we are reconstructing the mask from the dog and not from the horse as it was happening when using the MSE loss alone.	

Group 5	
MSE loss	MSE loss + 80 * Dice loss
	
Comments: In this case we can see that using the MSE + Dice loss we are able to predict the mask of the second image with more accurate boundaries. For the fourth image we see that the network is confusing a plate of soup with a cake due to their similarity. The performance for the first image is almost the same using the MSE loss alone and using the MSE+Dice loss	

In general, we can see that when using the combination of MSE +80* Dice loss the results are better than using the MSE alone. Nevertheless, we can find some singular cases where this is not the exact behavior. We can also state that, although there is a general improvement when using the MSE + 80*Dice loss, in some cases the masks are not accurate yet. In the

PurdueShapes5MultiObject dataset all the images had a solid black background and the different instances were also from a solid color (for example, we could have a pink star over a black background). The simplicity of the input images helps to predict more accurate masks. The COCO dataset contains real world images (i.e. images with several objects and several textures, thus several edges and boundaries) which are complex due to their naturality. Thus, it is expected that the masks predicted by the network are not as accurate as the ones obtained when using the PurdueShapes5MultiObject dataset.

CODE SECTION 3

Semantic_segmentation_combined.py

```
#!/usr/bin/env python

##  semantic_segmentation.py

import random
import numpy
import torch
import os, sys

import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torch.optim as optim
import numpy as np
from PIL import ImageFilter
import numbers
import re
import math
import random
import copy
import matplotlib.pyplot as plt
import gzip
import pickle
import pymsgbox
import time
import logging
import torchvision.transforms.functional as TF


from DLStudio import *

# WE CREATE SUPERCLASSES TO OVERWRITE THE FOLLOWING FUNCTIONS FROM THE
DLSTUDIO LIBRARY:
```

```
# run_code_for_training_for_semantic_segmentation(self, net)
#     WE IMPLEMENT THE COMBINED LOSS AND SAVE THE RUNNING LOSS VALUES TO
DISPLAY THEM LATER IN PLOTS
# run_code_for_testing_semantic_segmentation(self, net)
#     WE MODIFY THE CODE IN ORDER TO SAVE THE TESTING IMAGES WITH THEIR
MASKS
# WE ADD THE METHOD dice_loss IMPLEMENTED FOLLOWING THE INSTRUCTIONS FROM
SECTION 3.3
# THE REST OF THE CODE IS TAKEN FROM THE DLSTUDIO LIBRARY

class Prova1(DLStudio):
    class Prova2(DLStudio.SemanticSegmentation):
        # WE ADD THE METHOD dice_loss IMPLEMENTED FOLLOWING THE
INSTRUCTIONS FROM SECTION 3.3
        def dice_loss (self, preds : torch.Tensor, ground_truth :
torch.Tensor, epsilon =1e-6 ):
            """
            inputs :
                preds : predicted mask
                ground_truth : ground truth mask
                epsilon ( float ): prevents division by zero
            returns :
                dice_loss
            """

            # Step 1: Compute Dice Coefficient.
            numerator = torch.sum(preds * ground_truth, dim=(2, 3))
            denominator = torch.sum(preds * preds, dim=(2, 3)) +
torch.sum(ground_truth * ground_truth, dim=(2, 3))

            # Step 2: Compute dice_coefficient
            dice_coefficient = (2 * numerator) / (denominator + epsilon)

            # Step 3: Compute dice_loss
            dice_loss = 1 - dice_coefficient

            return dice_loss.mean()

        def run_code_for_training_for_semantic_segmentation(self, net):

            filename_for_out1 = "performance_numbers_" +
str(self.dl_studio.epochs) + ".txt"
            FILE1 = open(filename_for_out1, 'w')
            net = copy.deepcopy(net)
            net = net.to(self.dl_studio.device)
            optimizer = optim.SGD(net.parameters(),
                                lr=self.dl_studio.learning_rate,
momentum=self.dl_studio.momentum)
            start_time = time.perf_counter()
            criterion1_loss = []
            criterion2_loss = []
            criterion3_loss = []
            criterion1 = nn.MSELoss()


            for epoch in range(self.dl_studio.epochs):
                print("")
                running_loss = 0.0
                running_mse_loss = 0.0
                running_dice_loss = 0.0
                for i, data in enumerate(self.train_dataloader):
                    im_tensor,mask_tensor,bbox_tensor
=data['image'],data['mask_tensor'],data['bbox_tensor']
                    im_tensor   = im_tensor.to(self.dl_studio.device)
                    mask_tensor = mask_tensor.type(torch.FloatTensor)
                    mask_tensor = mask_tensor.to(self.dl_studio.device)

                    bbox_tensor = bbox_tensor.to(self.dl_studio.device)

                    optimizer.zero_grad()
                    output = net(im_tensor)

                    #WE IMPLEMENT THE COMBINED LOSS. WE CREATE A LOSS
VECTOR AND SET required_grad=True TO ENSURE BACKPROPAGATION
                    loss = torch.tensor(0.0,
requires_grad=True).float().to(self.dl_studio.device)

                    mse_loss = criterion1(output, mask_tensor)
                    dice_loss = self.dice_loss(preds=output,
ground_truth=mask_tensor)
                    loss = mse_loss + 40 * dice_loss
                    loss.backward()

                    optimizer.step()

                    running_loss += loss.item()
                    running_mse_loss += mse_loss.item()
                    running_dice_loss += dice_loss.item()

                    if i%500==499:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time

                        avg_loss = running_loss / float(500)
                        avg_mse_loss = running_mse_loss / float(500)
                        avg_dice_loss = running_dice_loss / float(500)
                        #WE SAVE THE RUNNING LOSS VALUES TO DISPLAY THEM
LATER IN PLOTS
                        criterion1_loss.append(running_loss)
                        criterion2_loss.append(running_mse_loss)
                        criterion3_loss.append(running_dice_loss)

                        print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d
secs]   loss: %.3f, MSE loss: %.3f, Dice loss: %.3f" % (epoch+1,
self.dl_studio.epochs, i+1, elapsed_time, avg_loss, avg_mse_loss,
avg_dice_loss))
                        FILE1.write("%.3f\n" % avg_loss)
                        FILE1.flush()

                        running_loss = 0.0
                        running_mse_loss = 0.0
                        running_dice_loss = 0.0

            print("\nFinished Training\n")
            self.save_model(net)

            dictionary_losses = {}

            nombre_imagen = 'yes'
            dictionary_losses[nombre_imagen] = {
                'criterion1': criterion1_loss,
                'criterion2': criterion2_loss,
                'criterion3': criterion3_loss,
            }
            with open('/home/aolivepe/ECE60146/HW7/DLStudio-
2.3.6/Examples/dictionary_Combined_scaleDice_40.pkl', 'wb') as archivo:
                pickle.dump(dictionary_losses, archivo)


        def save_model(self, model):
            '''
            Save the trained model to a disk file
            '''
            torch.save(model.state_dict(),
self.dl_studio.path_saved_model)


        def run_code_for_testing_semantic_segmentation(self, net):

net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
            batch_size = self.dl_studio.batch_size
            image_size = self.dl_studio.image_size
            max_num_objects = self.max_num_objects
            with torch.no_grad():
                for i, data in enumerate(self.test_dataloader):
                    im_tensor,mask_tensor,bbox_tensor
=data['image'],data['mask_tensor'],data['bbox_tensor']
                    if i % 50 == 0:
                        aa= i+1
                        print("\n\n\n\nShowing output for test batch %d:
" % (aa))
                        outputs = net(im_tensor)
                        ## In the statement below: 1st arg for batch
items, 2nd for channels, 3rd and 4th for image size
                        output_bw_tensor =
torch.zeros(batch_size,1,image_size[0],image_size[1], dtype=float)
                        for image_idx in range(batch_size):
                            for layer_idx in range(max_num_objects):
                                for m in range(image_size[0]):
                                    for n in range(image_size[1]):
                                        output_bw_tensor[image_idx,0,m,n]
 =  torch.max( outputs[image_idx,:,m,n] )
                        display_tensor = torch.zeros(7 *
batch_size,3,image_size[0],image_size[1], dtype=float)
                        for idx in range(batch_size):
                            for bbox_idx in range(max_num_objects):
                                bb_tensor = bbox_tensor[idx,bbox_idx]
                                for k in range(max_num_objects):
                                    i1 = int(bb_tensor[k][1])
                                    i2 = int(bb_tensor[k][3])
                                    j1 = int(bb_tensor[k][0])
                                    j2 = int(bb_tensor[k][2])
                                    output_bw_tensor[idx,0,i1:i2,j1] =
255
                                    output_bw_tensor[idx,0,i1:i2,j2] =
255
                                    output_bw_tensor[idx,0,i1,j1:j2] =
255
                                    output_bw_tensor[idx,0,i2,j1:j2] =
255
                                    im_tensor[idx,0,i1:i2,j1] = 255
                                    im_tensor[idx,0,i1:i2,j2] = 255
                                    im_tensor[idx,0,i1,j1:j2] = 255
                                    im_tensor[idx,0,i2,j1:j2] = 255
                        display_tensor[:batch_size,:,:,:] =
output_bw_tensor
                        display_tensor[batch_size:2*batch_size,:,:,:] =
im_tensor

                        for batch_im_idx in range(batch_size):
                            for mask_layer_idx in range(max_num_objects):
                                for i in range(image_size[0]):
                                    for j in range(image_size[1]):
                                        if mask_layer_idx == 0:
                                            if 25 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 85:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 1:
                                            if 65 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 135:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 2:
                                            if 115 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 185:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 3:
                                            if 165 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 230:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 4:
                                            if
outputs[batch_im_idx,mask_layer_idx,i,j] > 210:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50


display_tensor[2*batch_size+batch_size*mask_layer_idx+batch_im_idx,:,:,:]
= outputs[batch_im_idx,mask_layer_idx,:,:]

                        #WE MODIFY THE CODE IN ORDER TO SAVE THE TESTING
IMAGES WITH THEIR MASKS
                        # self.dl_studio.display_tensor_as_image(
                        #    torchvision.utils.make_grid(display_tensor,
nrow=batch_size, normalize=True, padding=2, pad_value=10))
```

```
                        image =
TF.to_pil_image(torchvision.utils.make_grid(display_tensor,
nrow=batch_size, normalize=True, padding=2, pad_value=10))

image.save(f"/home/aolivepe/ECE60146/HW7/DLStudio-
2.3.6/Examples/testing_imgs_comb_40/{aa}.png")


# WE USE THE NAME OF THE SUPERCLASSES THAT WE HAVE CREATED
dls = Prova1(
                  dataroot = "./../../data/",
                  image_size = [64,64],
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate = 1e-4,
                  epochs = 6,
                  batch_size = 4,
                  classes =
('rectangle','triangle','disk','oval','star'),
                  use_gpu = True,
              )

segmenter = Prova1.Prova2(
                  dl_studio = dls,
                  max_num_objects = 5,
              )

dataserver_train = Prova1.Prova2.PurdueShapes5MultiObjectDataset(
                          train_or_test = 'train',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiObject-10000-
train.gz",
                        )
dataserver_test = Prova1.Prova2.PurdueShapes5MultiObjectDataset(
                          train_or_test = 'test',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiObject-1000-
test.gz"
                        )
segmenter.dataserver_train = dataserver_train
segmenter.dataserver_test = dataserver_test

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train,
dataserver_test)

model = segmenter.mUnet(skip_connections=True, depth=16)
#model = segmenter.mUnet(skip_connections=False, depth=4)
```

```
number_of_learnable_params = sum(p.numel() for p in model.parameters() if
p.requires_grad)
print("\n\nThe number of learnable parameters in the model: %d\n" %
number_of_learnable_params)

num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model: %d\n\n" % num_layers)


segmenter.run_code_for_training_for_semantic_segmentation(model)

# import pymsgbox
# response = pymsgbox.confirm("Finished training.  Start testing on
unseen data?")
# if response == "OK":
segmenter.run_code_for_testing_semantic_segmentation(model)
```

CODE SECTION 4

Creation of the dataset

```
import torchvision.transforms as tvt
import cv2
from PIL import Image
import skimage.io as io
from pycocotools.coco import COCO
import os
import random
from torch.utils.data import DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
import skimage
import pickle
from skimage.transform import resize

#Data to use for either the training or validation dataset
full_COCO_training_path = "../../../../Downloads/train2017/train2017"
full_COCO_validation_path = "../../../../Downloads/val2017/val2017"

our_COCO_training_path = "./HW7_TRAINING_DATASET"
our_COCO_validation_path = "./HW7_VALIDATION_DATASET"
```

```
annotations_training_path =
"./../HW6/annotations/instances_train2017.json"
annotations_validation_path =
"./../HW6/annotations/instances_val2017.json"

coco=COCO(annotations_validation_path)

# Mapping from COCO label to Class indices
classes = ["dog", "cake", "motorcycle"]
catIds = coco.getCatIds(catNms=classes)
categories = coco.loadCats(catIds)
categories.sort(key=lambda x: x['id'])
coco_labels_inverse = {}
for idx, in_class in enumerate(classes):
    for c in categories:
        if c['name'] == in_class:
            coco_labels_inverse[c['id']] = idx

#4: motorcycle, 18: dog, 61: cake
# Save in an array the image ids of the images containing instance of 2
or more of the targeted classes
imgIds_4_18 = coco.getImgIds(catIds=[4, 18])
imgIds_4_61 = coco.getImgIds(catIds=[4, 61])
imgIds_61_18 = coco.getImgIds(catIds=[61, 18])
imgIds_4_61_18 = coco.getImgIds(catIds=[4, 61, 18])

total_ids = imgIds_4_18 + imgIds_4_61 + imgIds_61_18 + imgIds_4_61_18
print(len(total_ids))

dataset = {}
a = 0
# Iterate 3 times each over the images containing instances of ["dog",
"cake", "motorcycle"] and if accomplish the requirements, add it to the
dataset
for i, cat_id in enumerate(catIds):
    imgIds = coco.getImgIds(catIds=cat_id)

    # Remove the images containing instance of 2 or more of the targeted
classes
    filtered_array = [x for x in imgIds if x not in total_ids]

    for j, img_id in enumerate(filtered_array):
        annIds = coco.getAnnIds(imgIds=img_id, catIds=cat_id,
iscrowd=False)
        anns = coco.loadAnns(annIds)

        # If there is more than one instance of one class in the image we
skip it
```

```
        if len(anns) != 1:
            continue

        # Get the r, g, b arrays to save in the final dictionary resized
to 256x256
        img = coco.loadImgs(img_id)[0]
        I = io.imread(os.path.join(full_COCO_validation_path,
img['file_name']))
        if len(I.shape) == 2:
            I = skimage.color.gray2rgb(I)
        img_h, img_w = I.shape[0], I.shape[1]
        I = resize(I, (256, 256), anti_aliasing=True,
preserve_range=True)
        image = np.uint8(I)

        r = image[:,:,0]
        g = image[:,:,1]
        b = image[:,:,2]

        # Reshaping each channel into a 1D array
        r = r.reshape(-1, 1)  # Reshape to (256*256, 1)
        g = g.reshape(-1, 1)
        b = b.reshape(-1, 1)

        area = True
        for i, ann in enumerate(anns):
            # If the area of the bounding box is lower than 200*200 we
skip it
            if ann['area']<200*200:
                area=False
                continue

            # Get the bounding box and mask resized to 256x256
            [x, y, w, h] = ann['bbox']
            bbox = [x*(256/img_w), y*(256/img_h), w*(256/img_w),
h*(256/img_h)]
            mask = coco.annToMask(ann)
            mask = resize(mask, (256, 256), anti_aliasing=True,
preserve_range=True)

            threshold = 0.5

            # Binarize the array
            mask = (mask > threshold).astype(int)

            # Save the mask and the bounding box following the
characteristics mentioned at the beginning of section 4
            final_mask = torch.zeros(3, 256, 256)
            if cat_id == 4:
                final_mask[0, :, :] = torch.from_numpy(mask)*50
                bbox_dict = {
                    0:[[x*(256/img_w), y*(256/img_h), x*(256/img_w) + 
w*(256/img_w), y*(256/img_h) + h*(256/img_h)]],
                    1:[],
                    2:[]
                }
            if cat_id == 18:
                final_mask[1, :, :] = torch.from_numpy(mask)*100
                bbox_dict = {
                    0:[],
                    1:[[x*(256/img_w), y*(256/img_h), x*(256/img_w) + 
w*(256/img_w), y*(256/img_h) + h*(256/img_h)]],
                    2:[]
                }
            if cat_id == 61:
                final_mask[2, :, :] = torch.from_numpy(mask)*150
                bbox_dict = {
                    0:[],
                    1:[],
                    2:[[x*(256/img_w), y*(256/img_h), x*(256/img_w) + 
w*(256/img_w), y*(256/img_h) + h*(256/img_h)]]
                }

        if area == False:
            continue

        # We create a dictionary called dataset where for each image with 
index "a" we save the following information
        dataset[a] = {
            0: r,
            1: g,
            2: b,
            3: final_mask,
            4: bbox_dict
        }

        a = a+1
    print(f"**Acumulated** Num of images labeled with cat_id {cat_id}: 
{a}")

#Save the dictionary in a "pkl" file
with open('validation_dataset.pkl', 'wb') as file:
    pickle.dump(dataset, file)
```

semantic_segmentation_combined_COCO.py

```
#!/usr/bin/env python
```

```
##  semantic_segmentation.py

import random
import numpy
import torch
import os, sys

import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torch.optim as optim
import numpy as np
from PIL import ImageFilter
import numbers
import re
import math
import random
import copy
import matplotlib.pyplot as plt
import gzip
import pickle
import pymsgbox
import time
import logging
import torchvision.transforms.functional as TF


from DLStudio import *

# WE CREATE SUPERCLASSES TO OVERWRITE THE FOLLOWING FUNCTIONS FROM THE
DLSTUDIO LIBRARY:

# run_code_for_training_for_semantic_segmentation(self, net)
#     WE IMPLEMENT THE COMBINED LOSS AND SAVE THE RUNNING LOSS VALUES TO
DISPLAY THEM LATER IN PLOTS

# run_code_for_testing_semantic_segmentation(self, net)
#     WE MODIFY THE CODE IN ORDER TO SAVE THE TESTING IMAGES WITH THEIR
MASKS

# WE ADD THE METHOD dice_loss IMPLEMENTED FOLLOWING THE INSTRUCTIONS FROM
SECTION 3.3

# WE CREATE A SUPERCLASS OF THE DATASET CLASS TO OVERWRITE ITS METHODS
# WE CREATE A SUPERCLASS OF THE mUnet CLASS TO OVERWRITE ITS METHODS

# THE REST OF THE CODE IS TAKEN FROM THE DLSTUDIO LIBRARY

class Prova1(DLStudio):
    class Prova2(DLStudio.SemanticSegmentation):
        class MyDataset(torch.utils.data.Dataset):
            def __init__(self, dl_studio, segmenter, train_or_test,
dataset_file):
                super(Prova1.Prova2.MyDataset, self).__init__()
                max_num_objects = segmenter.max_num_objects
                if train_or_test == 'train':
                    print("\nLoading training data from torch saved
file")

                    # Load dictionary with training dataset
                    with
open('/home/aolivepe/ECE60146/HW7/training_dataset.pkl', 'rb') as file:
                        self.dataset = pickle.load(file)

                    self.label_map = {
                        'motorcycle': 50,
                        'dog': 100,
                        'cake': 150
                    }

                    self.num_shapes = len(self.label_map)
                    self.image_size = dl_studio.image_size
                else:
                    # Load dictionary with testing dataset
                    with
open('/home/aolivepe/ECE60146/HW7/validation_dataset.pkl', 'rb') as file:
                        self.dataset = pickle.load(file)

                    self.label_map = {
                        'motorcycle': 50,
                        'dog': 100,
                        'cake': 150
                    }

                    # reverse the key-value pairs in the label
dictionary:
                    self.class_labels = {
                        50: 'motorcycle',
                        100: 'dog',
                        150: 'cake'
                    }
                    self.num_shapes = len(self.class_labels)
                    self.image_size = dl_studio.image_size
            def __len__(self):
                return len(self.dataset)

            def __getitem__(self, idx):

                # Creating image
                image_size = self.image_size
                r = np.array( self.dataset[idx][0] )
                g = np.array( self.dataset[idx][1] )
                b = np.array( self.dataset[idx][2] )
                R,G,B = r.reshape(image_size[0],image_size[1]),
g.reshape(image_size[0],image_size[1]),
b.reshape(image_size[0],image_size[1])
                im_tensor = torch.zeros(3,image_size[0],image_size[1],
dtype=torch.float)
                im_tensor[0,:,:] = torch.from_numpy(R)
                im_tensor[1,:,:] = torch.from_numpy(G)
                im_tensor[2,:,:] = torch.from_numpy(B)

                # Getting mask
                mask_array = np.array(self.dataset[idx][3])
                max_num_objects = len( mask_array[0] )
                mask_tensor = torch.from_numpy(mask_array)


                mask_val_to_bbox_map =  self.dataset[idx][4]
                max_bboxes_per_entry_in_map = max([
len(mask_val_to_bbox_map[key]) for key in mask_val_to_bbox_map ])
                ##  The first arg 5 is for the number of bboxes we are
going to need. If all the
                ##  shapes are exactly the same, you are going to need
five different bbox'es.
                ##  The second arg is the index reserved for each shape
in a single bbox
                bbox_tensor =
torch.zeros(max_num_objects,self.num_shapes,4, dtype=torch.float)
                for bbox_idx in range(max_bboxes_per_entry_in_map):
                    for key in mask_val_to_bbox_map:
                        if len(mask_val_to_bbox_map[key]) == 1:
                            if bbox_idx == 0:
                                bbox_tensor[bbox_idx,key,:] =
torch.from_numpy(np.array(mask_val_to_bbox_map[key][bbox_idx]))
                        elif len(mask_val_to_bbox_map[key]) > 1 and
bbox_idx < len(mask_val_to_bbox_map[key]):
                            bbox_tensor[bbox_idx,key,:] =
torch.from_numpy(np.array(mask_val_to_bbox_map[key][bbox_idx]))
                sample = {'image'         : im_tensor,
                          'mask_tensor'   : mask_tensor,
                          'bbox_tensor'   : bbox_tensor }
                return sample

        class MymUnet(nn.Module):
            # EXTENSION OF THE mUnet PROVIDED IN DLSTUDIO BUILT FOLLOWING
THE EXPLANATIONS FROM SECTION 4.1.
            def __init__(self, skip_connections=True, depth=16):
                super(Prova1.Prova2.MymUnet, 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.bn3DN  = nn.BatchNorm2d(256)
                self.bn4DN  = nn.BatchNorm2d(512)

                self.skip64DN_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip64DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(64,
64, skip_connections=skip_connections))
                self.skip64dsDN =
DLStudio.SemanticSegmentation.SkipBlockDN(64, 64,   downsample=True,
skip_connections=skip_connections)

                self.skip64to128DN =
DLStudio.SemanticSegmentation.SkipBlockDN(64, 128,
skip_connections=skip_connections )

                self.skip128DN_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip128DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(128,
128, skip_connections=skip_connections))
                self.skip128dsDN =
DLStudio.SemanticSegmentation.SkipBlockDN(128,128, downsample=True,
skip_connections=skip_connections)

                self.skip128to256DN =
DLStudio.SemanticSegmentation.SkipBlockDN(128, 256,
skip_connections=skip_connections )

                self.skip256DN_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip256DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(256,
256, skip_connections=skip_connections))
```

```
                self.skip256dsDN =
DLStudio.SemanticSegmentation.SkipBlockDN(256,256, downsample=True,
skip_connections=skip_connections)

                self.skip256to512DN =
DLStudio.SemanticSegmentation.SkipBlockDN(256, 512,
skip_connections=skip_connections )

                self.skip512DN_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip512DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(512,
512, skip_connections=skip_connections))
                self.skip512dsDN =
DLStudio.SemanticSegmentation.SkipBlockDN(512,512, downsample=True,
skip_connections=skip_connections)


                ##  For the UP arm of the U:
                self.bn1UP  = nn.BatchNorm2d(512)
                self.bn2UP  = nn.BatchNorm2d(256)
                self.bn3UP  = nn.BatchNorm2d(128)
                self.bn4UP  = nn.BatchNorm2d(64)

                self.skip64UP_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip64UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(64,
64, skip_connections=skip_connections))
                self.skip64usUP =
DLStudio.SemanticSegmentation.SkipBlockUP(64, 64, upsample=True,
skip_connections=skip_connections)

                self.skip128to64UP =
DLStudio.SemanticSegmentation.SkipBlockUP(128, 64,
skip_connections=skip_connections )

                self.skip128UP_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip128UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(128,
128, skip_connections=skip_connections))
                self.skip128usUP =
DLStudio.SemanticSegmentation.SkipBlockUP(128,128, upsample=True,
skip_connections=skip_connections)

                self.skip256to128UP =
DLStudio.SemanticSegmentation.SkipBlockUP(256, 128,
skip_connections=skip_connections )
```

```
                self.skip256UP_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip256UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(256,
256, skip_connections=skip_connections))
                self.skip256usUP =
DLStudio.SemanticSegmentation.SkipBlockUP(256,256, upsample=True,
skip_connections=skip_connections)

                self.skip512to256UP =
DLStudio.SemanticSegmentation.SkipBlockUP(512, 256,
skip_connections=skip_connections )

                self.skip512UP_arr = nn.ModuleList()
                for i in range(self.depth):

self.skip512UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(512,
512, skip_connections=skip_connections))
                self.skip512usUP =
DLStudio.SemanticSegmentation.SkipBlockUP(512,512, upsample=True,
skip_connections=skip_connections)

                self.conv_out = nn.ConvTranspose2d(64, 3, 3,
stride=2,dilation=2,output_padding=1,padding=2)

            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)
                num_channels_to_save3 = x.shape[1] // 2
                save_for_upside_3 =
x[:,:num_channels_to_save3,:,:].clone()
                x = self.skip128dsDN(x)
                for i,skip128 in
enumerate(self.skip128DN_arr[self.depth//4:]):
                    x = skip128(x)
                x = self.bn2DN(x)
                num_channels_to_save4 = x.shape[1] // 2
                save_for_upside_4 =
x[:,:num_channels_to_save4,:,:].clone()

                x = self.skip128to256DN(x)

                for i,skip256 in
enumerate(self.skip256DN_arr[:self.depth//4]):
                    x = skip256(x)
                num_channels_to_save5 = x.shape[1] // 2
                save_for_upside_5 =
x[:,:num_channels_to_save5,:,:].clone()
                x = self.skip256dsDN(x)
                for i,skip256 in
enumerate(self.skip256DN_arr[self.depth//4:]):
                    x = skip256(x)
                x = self.bn3DN(x)
                num_channels_to_save6 = x.shape[1] // 2
                save_for_upside_6 =
x[:,:num_channels_to_save6,:,:].clone()

                x = self.skip256to512DN(x)

                for i,skip512 in
enumerate(self.skip512DN_arr[:self.depth//4]):
                    x = skip512(x)
                x = self.bn4DN(x)
                num_channels_to_save7 = x.shape[1] // 2
                save_for_upside_7 =
x[:,:num_channels_to_save7,:,:].clone()
                for i,skip512 in
enumerate(self.skip512DN_arr[self.depth//4:]):
                    x = skip512(x)
                x = self.skip512dsDN(x)


                ## Coming up from the bottom of U on the other side:
                x = self.skip512usUP(x)
                for i,skip512 in
enumerate(self.skip512UP_arr[:self.depth//4]):
                    x = skip512(x)
                x[:,:num_channels_to_save7,:,:] =  save_for_upside_7
                x = self.bn1UP(x)
                for i,skip512 in
enumerate(self.skip512UP_arr[:self.depth//4]):
                    x = skip512(x)

                x = self.skip512to256UP(x)

                for i,skip256 in
enumerate(self.skip256UP_arr[self.depth//4:]):
                    x = skip256(x)
                x[:,:num_channels_to_save6,:,:] =  save_for_upside_6
                x = self.bn2UP(x)
                x = self.skip256usUP(x)
                for i,skip256 in
enumerate(self.skip256UP_arr[:self.depth//4]):
                    x = skip256(x)
                x[:,:num_channels_to_save5,:,:] =  save_for_upside_5

                x = self.skip256to128UP(x)

                for i,skip128 in
enumerate(self.skip128UP_arr[self.depth//4:]):
                    x = skip128(x)
                x[:,:num_channels_to_save4,:,:] =  save_for_upside_4
                x = self.bn3UP(x)
                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.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.bn4UP(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

        # WE ADD THE METHOD dice_loss IMPLEMENTED FOLLOWING THE
INSTRUCTIONS FROM SECTION 3.3
        def dice_loss (self, preds : torch.Tensor, ground_truth :
torch.Tensor, epsilon =1e-6 ):
            """
            inputs :
                preds : predicted mask
                ground_truth : ground truth mask
                epsilon ( float ): prevents division by zero
            returns :
                dice_loss
            """

            # Step 1: Compute Dice Coefficient.
            numerator = torch.sum(preds * ground_truth, dim=(2, 3))
            denominator = torch.sum(preds * preds, dim=(2, 3)) +
torch.sum(ground_truth * ground_truth, dim=(2, 3))

            # Step 2: Compute dice_coefficient
            dice_coefficient = (2 * numerator) / (denominator + epsilon)

            # Step 3: Compute dice_loss
            dice_loss = 1 - dice_coefficient

            return dice_loss.mean()

        def run_code_for_training_for_semantic_segmentation(self, net):

            filename_for_out1 = "performance_numbers_" +
str(self.dl_studio.epochs) + ".txt"
            FILE1 = open(filename_for_out1, 'w')
            net = copy.deepcopy(net)
            net = net.to(self.dl_studio.device)
            optimizer = optim.SGD(net.parameters(),
                                lr=self.dl_studio.learning_rate,
momentum=self.dl_studio.momentum)
            start_time = time.perf_counter()

            criterion1_loss = []
            criterion2_loss = []
            criterion3_loss = []
            criterion1 = nn.MSELoss()


            for epoch in range(self.dl_studio.epochs):
                print("")
                running_loss = 0.0
                running_mse_loss = 0.0
                running_dice_loss = 0.0
                for i, data in enumerate(self.train_dataloader):
                    im_tensor,mask_tensor,bbox_tensor
=data['image'],data['mask_tensor'],data['bbox_tensor']
                    im_tensor   = im_tensor.to(self.dl_studio.device)
                    mask_tensor = mask_tensor.type(torch.FloatTensor)
                    mask_tensor = mask_tensor.to(self.dl_studio.device)

                    bbox_tensor = bbox_tensor.to(self.dl_studio.device)

                    optimizer.zero_grad()
                    output = net(im_tensor)

                    # print("output: ", output.shape, torch.max(output),
torch.min(output))
                    # print("mask_tensor: ", mask_tensor.shape,
torch.max(mask_tensor), torch.min(mask_tensor))

                    #WE IMPLEMENT THE COMBINED LOSS. WE CREATE A LOSS
VECTOR AND SET required_grad=True TO ENSURE BACKPROPAGATION
                    loss = torch.tensor(0.0,
requires_grad=True).float().to(self.dl_studio.device)

                    mse_loss = criterion1(output, mask_tensor)
                    dice_loss = self.dice_loss(preds=output,
ground_truth=mask_tensor)
                    loss = mse_loss + 80 * dice_loss
                    loss.backward()

                    optimizer.step()

                    running_loss += loss.item()
                    running_mse_loss += mse_loss.item()
                    running_dice_loss += dice_loss.item()

                    if i%50==49:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time

                        avg_loss = running_loss / float(50)
                        avg_mse_loss = running_mse_loss / float(50)
                        avg_dice_loss = running_dice_loss / float(50)

                        #WE SAVE THE RUNNING LOSS VALUES TO DISPLAY THEM
LATER IN PLOTS
                        criterion1_loss.append(running_loss)
                        criterion2_loss.append(running_mse_loss)
                        criterion3_loss.append(running_dice_loss)

                        print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d 
secs]   loss: %.3f, MSE loss: %.3f, Dice loss: %.3f" % (epoch+1, 
self.dl_studio.epochs, i+1, elapsed_time, avg_loss, avg_mse_loss, 
avg_dice_loss))
                        FILE1.write("%.3f\n" % avg_loss)
                        FILE1.flush()

                        running_loss = 0.0
                        running_mse_loss = 0.0
                        running_dice_loss = 0.0

            print("\nFinished Training\n")
            self.save_model(net)

            dictionary_losses = {}

            nombre_imagen = 'yes'
            dictionary_losses[nombre_imagen] = {
                'criterion1': criterion1_loss,
                'criterion2': criterion2_loss,
                'criterion3': criterion3_loss,
            }

            with open('/home/aolivepe/ECE60146/HW7/DLStudio-
2.3.6/Examples/dictionary_Combined_scaleDice_80_COCO.pkl', 'wb') as 
archivo:
                pickle.dump(dictionary_losses, archivo)


        def save_model(self, model):
            '''
            Save the trained model to a disk file
            '''
            torch.save(model.state_dict(), 
self.dl_studio.path_saved_model)


        def run_code_for_testing_semantic_segmentation(self, net):
            
net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
            batch_size = self.dl_studio.batch_size
            image_size = self.dl_studio.image_size
            max_num_objects = self.max_num_objects
            with torch.no_grad():
                for i, data in enumerate(self.test_dataloader):
                    im_tensor,mask_tensor,bbox_tensor 
=data['image'],data['mask_tensor'],data['bbox_tensor']
```

```
                    if i % 3 == 0:
                        aa= i+1
                        print("\n\n\n\nShowing output for test batch %d:
" % (aa))
                        outputs = net(im_tensor)
                        print("output testing: ", outputs.shape)

                        ## In the statement below: 1st arg for batch
items, 2nd for channels, 3rd and 4th for image size
                        output_bw_tensor =
torch.zeros(batch_size,1,image_size[0],image_size[1], dtype=float)
                        for image_idx in range(batch_size):
                            for layer_idx in range(max_num_objects):
                                for m in range(image_size[0]):
                                    for n in range(image_size[1]):
                                        output_bw_tensor[image_idx,0,m,n]
 =  torch.max( outputs[image_idx,:,m,n] )
                        display_tensor = torch.zeros(7 *
batch_size,3,image_size[0],image_size[1], dtype=float)
                        for idx in range(batch_size):
                            for bbox_idx in range(max_num_objects):
                                bb_tensor = bbox_tensor[idx,bbox_idx]
                                for k in range(max_num_objects):
                                    i1 = int(bb_tensor[k][1])
                                    i2 = int(bb_tensor[k][3])
                                    j1 = int(bb_tensor[k][0])
                                    j2 = int(bb_tensor[k][2])

                                    # I CAN PROBABLY REMOVE THIS
                                    if i1 > 255:
                                        i1 = 255
                                    if i2 > 255:
                                        i2 = 255
                                    if j1 > 255:
                                        j1 = 255
                                    if j2 > 255:
                                        j2 = 255

                                    output_bw_tensor[idx,0,i1:i2,j1] =
255
                                    output_bw_tensor[idx,0,i1:i2,j2] =
255
                                    output_bw_tensor[idx,0,i1,j1:j2] =
255
                                    output_bw_tensor[idx,0,i2,j1:j2] =
255
                                    im_tensor[idx,0,i1:i2,j1] = 255
                                    im_tensor[idx,0,i1:i2,j2] = 255
                                    im_tensor[idx,0,i1,j1:j2] = 255
                                    im_tensor[idx,0,i2,j1:j2] = 255
                        display_tensor[:batch_size,:,:,:] =
output_bw_tensor
                        display_tensor[batch_size:2*batch_size,:,:,:] =
im_tensor

                        for batch_im_idx in range(batch_size):
                            for mask_layer_idx in range(max_num_objects):
                                for i in range(image_size[0]):
                                    for j in range(image_size[1]):
                                        if mask_layer_idx == 0:
                                            #SINCE WE ARE WORKING ONLY
WITH 3 CLASSES, WE REMOVED THE OTHER 2 LEVELS
                                            if 25 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 85:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 1:
                                            if 65 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 135:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50
                                        elif mask_layer_idx == 2:
                                            if 115 <
outputs[batch_im_idx,mask_layer_idx,i,j] < 185:

outputs[batch_im_idx,mask_layer_idx,i,j] = 255
                                            else:

outputs[batch_im_idx,mask_layer_idx,i,j] = 50


display_tensor[2*batch_size+batch_size*mask_layer_idx+batch_im_idx,:,:,:]
= outputs[batch_im_idx,mask_layer_idx,:,:]
                        #WE MODIFY THE CODE IN ORDER TO SAVE THE TESTING
IMAGES WITH THEIR MASKS
                        # self.dl_studio.display_tensor_as_image(
                        #    torchvision.utils.make_grid(display_tensor,
nrow=batch_size, normalize=True, padding=2, pad_value=10))
                        image =
TF.to_pil_image(torchvision.utils.make_grid(display_tensor,
nrow=batch_size, normalize=True, padding=2, pad_value=10))
```

```
image.save(f"/home/aolivepe/ECE60146/HW7/DLStudio-
2.3.6/Examples/testing_imgs_comb_300_COCO/{aa}.png")

# WE USE THE NAME OF THE SUPERCLASSES THAT WE HAVE CREATED
dls = Prova1(
                  dataroot = "./../../data/",
                  image_size = [256,256],
                  path_saved_model = "./saved_model_Dice_COCO_300",
                  momentum = 0.9,
                  learning_rate = 1e-4,
                  epochs = 50,
                  batch_size = 4,
                  classes = ('motorcycle','dog','cake'),
                  use_gpu = True,
              )

segmenter = Prova1.Prova2(
                  dl_studio = dls,
                  max_num_objects = 3,
              )

dataserver_train = Prova1.Prova2.MyDataset(
                          train_or_test = 'train',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiObject-10000-
train.gz",
                        )
dataserver_test = Prova1.Prova2.MyDataset(
                          train_or_test = 'test',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiObject-1000-
test.gz"
                        )
segmenter.dataserver_train = dataserver_train
segmenter.dataserver_test = dataserver_test

#Create dataloaders
segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train,
dataserver_test)

model = segmenter.MymUnet(skip_connections=True, depth=16)
#model = segmenter.mUnet(skip_connections=False, depth=4)

number_of_learnable_params = sum(p.numel() for p in model.parameters() if
p.requires_grad)
```

```
print("\n\nThe number of learnable parameters in the model: %d\n" %
number_of_learnable_params)

num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model: %d\n\n" % num_layers)


segmenter.run_code_for_training_for_semantic_segmentation(model)
# model.load_state_dict(torch.load("/home/aolivepe/ECE60146/HW7/DLStudio-
2.3.6/Examples/saved_model"))

print("Start Testing")
# import pymsgbox
# response = pymsgbox.confirm("Finished training.  Start testing on
unseen data?")
# if response == "OK":
segmenter.run_code_for_testing_semantic_segmentation(model)
```