Semantic Segmentation using Unet

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

mUnet architecture is a modified version of the U-Net architecture used for semantic segmentation of images which is based on the encoder-decoder architecture. It can be used for capturing both global context and fine details in images.

- The mUnet architecture is made up of two main parts: the downsampling part (DN) and the upsampling part (UP). These parts are connected through skip connections.
- The DN part is responsible for encoding the input image and extracting high-level features and capturing context, while the UP part is responsible for precise localization and decoding and generating the segmentation mask.
- Each part contains a series of convolutional layers and skip blocks connections, which help in preserving spatial information and learning hierarchical features.
- The mUnet class is initialized with parameters such as the depth of the network and whether skip connections are enabled. Skip blocks are initialized for both the down part (DN) and the up part (UP).
- In the forward pass, the input image is passed through the DN part to extract features, where the input is downsampled to reduce spatial dimensions and increase the receptive field.
- At each downsample step, the resolution is halved, and the number of channels is increased.
- After reaching the bottom of the U shape, the features are passed through the UP part for decoding, where the features are upsampled while passing

through the UP part to reconstruct the original spatial dimensions.

- At each upsample step, the spatial resolution is doubled, and the number of channels is decreased.
- Skip connections are used to concatenate features from the corresponding layers in the DN and UP parts, helping in preserving spatial information and improving segmentation accuracy.
- Finally, the output is passed through a convolutional layer to produce the predicted segmentation mask.

Dice Loss:

In semantic segmentation tasks, dice loss, which is also referred to as the Sørensen-Dice coefficient, is a popular similarity metric. It calculates the overlap between two sets, the ground truth mask and the predicted segmentation mask. The formula for Dice coefficient is twice the intersection of the two sets divided by the sum of their sizes. The dice loss is calculated as 1 minus the dice coefficient when there is a loss. Better segmentation performance is indicated by the fact that the loss decreases as the overlap between the predicted and ground truth masks increases. Because it equally penalizes false negatives and false positives, the dice loss is especially helpful for datasets that are having unbalanced classes or when there are small objects in the dataset images.

The Dice coefficient is calculated as:

$$Dice = \frac{2\,|A \cap B|}{|A| + |B|}$$

```
#calculation of Dice Loss code used in the code below:
numerator = torch.sum(output * mask_tensor)
denominator = torch.sum(output * output) + torch.sum(mask_ten
sor * mask_tensor)
```

```
dice_coefficient = 2 * numerator / (denominator + (1e-6))
segmentation_loss_dice = (1 - dice_coefficient)
```

The mUnet architecture used:

```
class mUnet(nn.Module):

    def __init__(self, skip_connections=True, depth=16):
        super().__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=skip_connections))
        self.skip64dsDN = SkipBlockDN(64, 64,   downsample=Tr
ue, skip_connections=skip_connections)
        self.skip64to128DN = SkipBlockDN(64, 128, skip_connec
tions=skip_connections )
        self.skip128DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128DN_arr.append(SkipBlockDN(128, 128, s
kip_connections=skip_connections))
        self.skip128dsDN = SkipBlockDN(128,128, downsample=Tr
ue, skip_connections=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=skip_connections))
        self.skip64usUP = SkipBlockUP(64, 64, upsample=True,
```

```
skip_connections=skip_connections)
        self.skip128to64UP = SkipBlockUP(128, 64, skip_connec
tions=skip_connections )
        self.skip128UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128UP_arr.append(SkipBlockUP(128, 128, s
kip_connections=skip_connections))
        self.skip128usUP = SkipBlockUP(128,128, upsample=Tru
e, skip_connections=skip_connections)
        self.conv_out = nn.ConvTranspose2d(64, 5, 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.dep
th//4]):
            x = skip64(x)

        num_channels_to_save1 = x.shape[1] // 2
        save_for_upside_1 = x[:,:num_channels_to_save1,:,:].c
lone()
        x = self.skip64dsDN(x)
        for i,skip64 in enumerate(self.skip64DN_arr[self.dept
h//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,:,:].c
lone()
        x = self.skip64to128DN(x)
        for i,skip128 in enumerate(self.skip128DN_arr[:self.d
epth//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,:,:].c
lone()
        for i,skip128 in enumerate(self.skip128DN_arr[self.de
pth//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.d
epth//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.d
epth//4]):
            x = skip128(x)
        x = self.skip128to64UP(x)
        for i,skip64 in enumerate(self.skip64UP_arr[self.dept
h//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.dep
th//4]):
            x = skip64(x)
        x[:,:num_channels_to_save1,:,:] =  save_for_upside_1
        x = self.conv_out(x)
        return x



class SkipBlockDN(nn.Module):
```

```
    def __init__(self, in_ch, out_ch, downsample=False, skip_
connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, st
ride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + torch.cat((identity, identity), d
im=1)
        return out
```

```
class SkipBlockUP(nn.Module):

    def __init__(self, in_ch, out_ch, upsample=False, skip_co
nnections=True):
        super().__init__()
        self.upsample = upsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if upsample:
            self.upsampler = nn.ConvTranspose2d(in_ch, out_c
h, 1, stride=2, dilation=2, output_padding=1, padding=0)
    def forward(self, x):
        identity = x
        out = self.convoT1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out  =  nn.ReLU(inplace=False)(out)
        if self.in_ch == self.out_ch:
            out = self.convoT2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.upsample:
            out = self.upsampler(out)
            identity = self.upsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
```

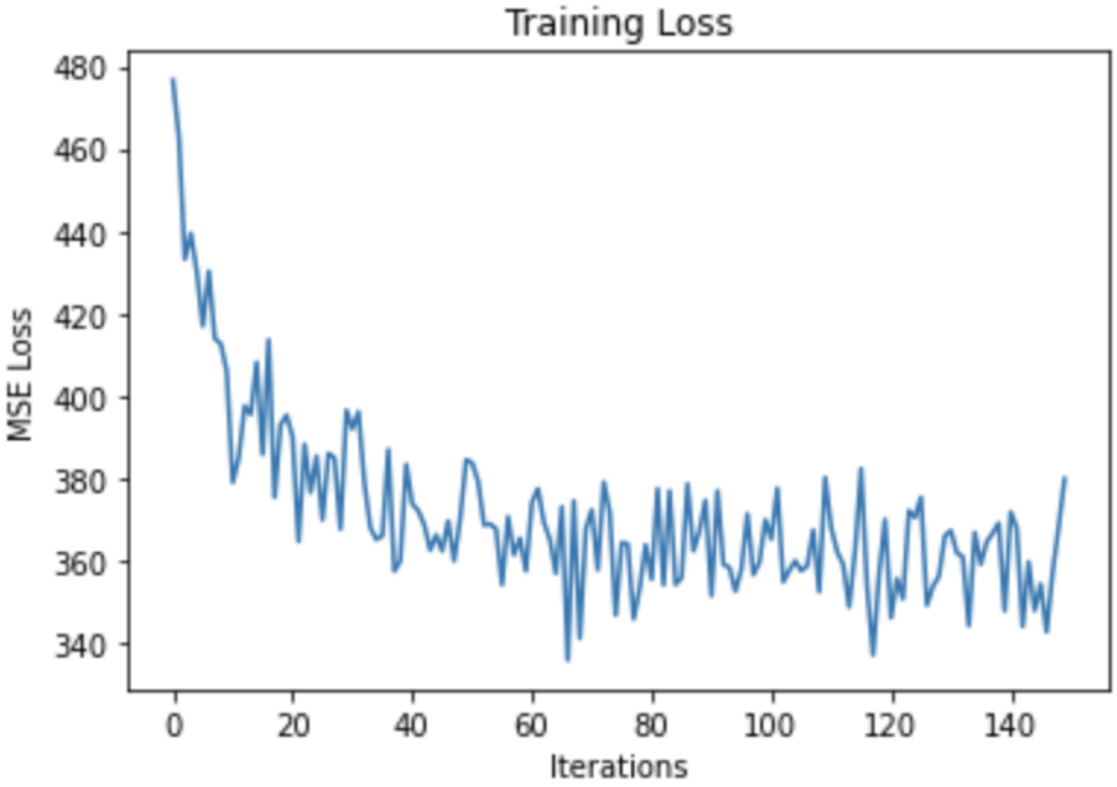
```
                out = out + identity[:,self.out_ch:,:,:]
        return out
```

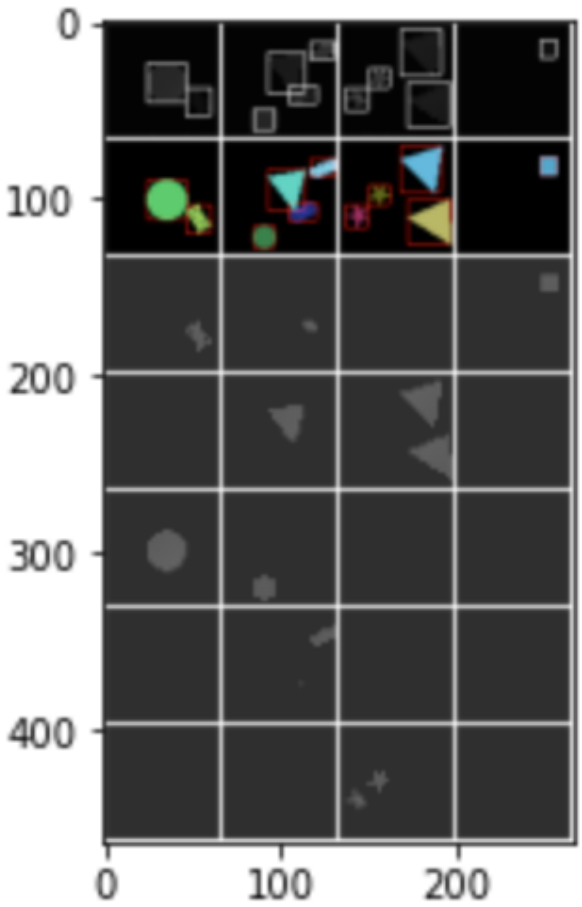
```
The number of learnable parameters in the model: 6999109
The number of layers in the model: 324
```

MSE Loss curve:

min loss : 335

max loss : 476





Dice Loss:

Code for Dice Loss Training routine used with the code for Dice Loss calculation:

```
def run_code_for_training_for_semantic_segmentation(self, ne
t):
        filename_for_out1 = "performance_numbers_" + str(sel
f.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(),
```

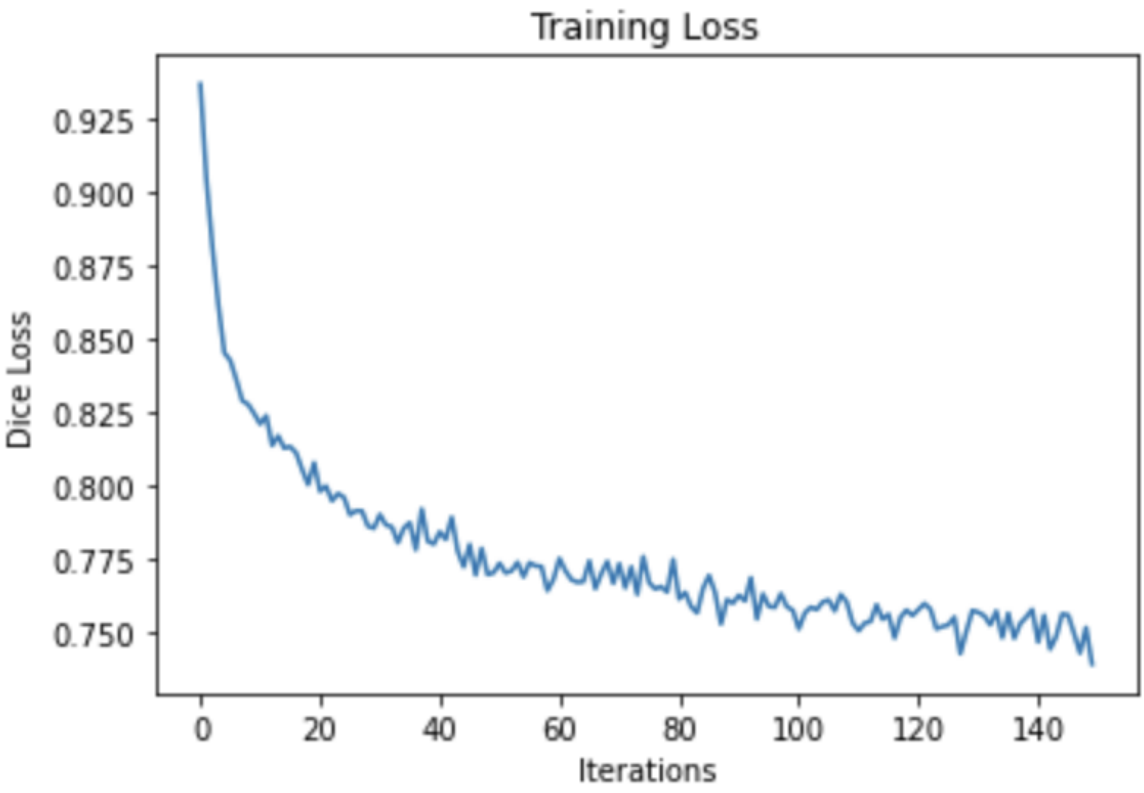
```
                    lr=self.dl_studio.learning_rate, momentu
m=self.dl_studio.momentum)
        start_time = time.perf_counter()

        loss_values = []
        for epoch in range(self.dl_studio.epochs):
            print("")
            running_loss_segmentation_dice = 0.0
            for i, data in enumerate(self.train_dataloader):
                im_tensor,mask_tensor,bbox_tensor =data['imag
e'],data['mask_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.dev
ice)
                mask_tensor = mask_tensor.type(torch.FloatTen
sor)
                mask_tensor = mask_tensor.to(self.dl_studio.d
evice)
                bbox_tensor = bbox_tensor.to(self.dl_studio.d
evice)
                optimizer.zero_grad()
                output = net(im_tensor)

                #calculation of Dice Loss:
                numerator = torch.sum(output * mask_tensor)
                denominator = torch.sum(output * output) + to
rch.sum(mask_tensor * mask_tensor)
                dice_coefficient = 2 * numerator / (denominat
or + (1e-6))
                segmentation_loss_dice = (1 - dice_coefficien
t)



                segmentation_loss_dice.backward()
                optimizer.step()
                running_loss_segmentation_dice += segmentatio
n_loss_dice.item()
```

```
                if i%100==99:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segm
entation_dice / float(100)
                    print("[epoch=%d/%d, iter=%4d  elapsed_ti
me=%3d secs]   Dice loss: %.3f" % (epoch+1, self.dl_studio.ep
ochs, i+1, elapsed_time, avg_loss_segmentation))
                    FILE1.write("%.3f\n" % avg_loss_segmentat
ion)
                    FILE1.flush()
                    running_loss_segmentation_dice = 0.0

                    loss_values.append(avg_loss_segmentation)

        print("\nFinished Training\n")
        self.save_model(net)
            # Plot loss values versus iterations
        import matplotlib.pyplot as plt
        plt.plot(range(len(loss_values)), loss_values)
        plt.xlabel('Iterations')
        plt.ylabel('Dice Loss')
        plt.title('Training Loss')
        plt.show()
        return loss_values
```

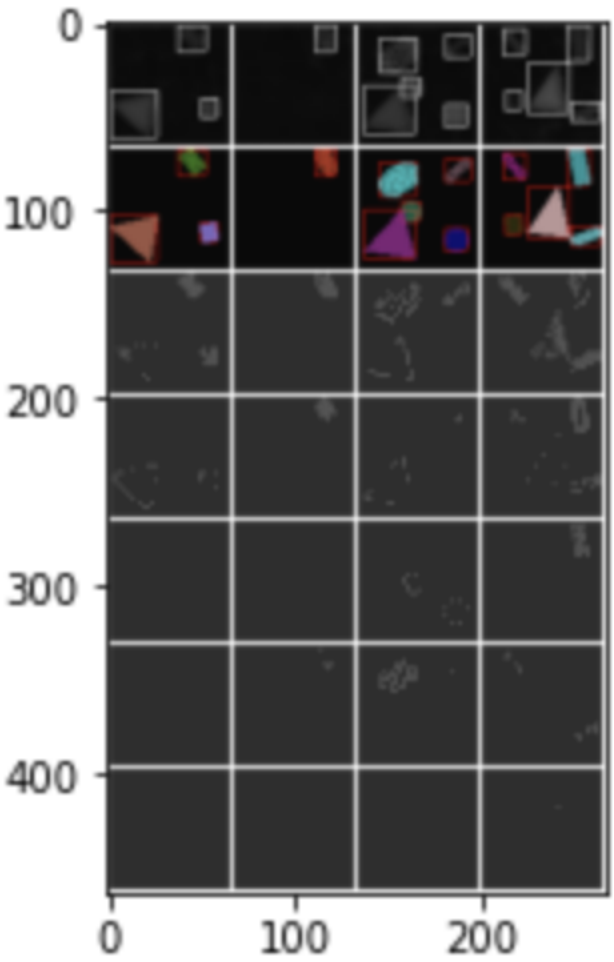
Dice loss curve:

min loss value: 0.739

max loss value: 0.936


Training Loss
Dice Loss
0.925
0.900
0.875
0.850
0.825
0.800
0.775
0.750
0
20
40
60
80
100
120
140
Iterations

test output



MSE + Dice Loss:

Code used for training loop for Combined loss:

```
def run_code_for_training_for_semantic_segmentation(self, ne
t):
        filename_for_out1 = "performance_numbers_" + str(sel
f.dl_studio.epochs) + ".txt"
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)
        criterion1 = nn.MSELoss()
```

```
        optimizer = optim.SGD(net.parameters(),
                     lr=self.dl_studio.learning_rate, momentu
m=self.dl_studio.momentum)
        start_time = time.perf_counter()

        loss_values = []
        for epoch in range(self.dl_studio.epochs):
            print("")

            running_loss_segmentation_mse = 0.0
            running_loss_segmentation_dice = 0.0
            running_loss_segmentation_combined = 0.0

            for i, data in enumerate(self.train_dataloader):
                im_tensor,mask_tensor,bbox_tensor =data['imag
e'],data['mask_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.dev
ice)
                mask_tensor = mask_tensor.type(torch.FloatTen
sor)
                mask_tensor = mask_tensor.to(self.dl_studio.d
evice)
                bbox_tensor = bbox_tensor.to(self.dl_studio.d
evice)
                optimizer.zero_grad()
                output = net(im_tensor)


                 # Calculate MSE loss

                segmentation_loss_mse = criterion1(output, ma
sk_tensor)
                running_loss_segmentation_mse += segmentation
_loss_mse.item()

                # Calculate Dice loss
```

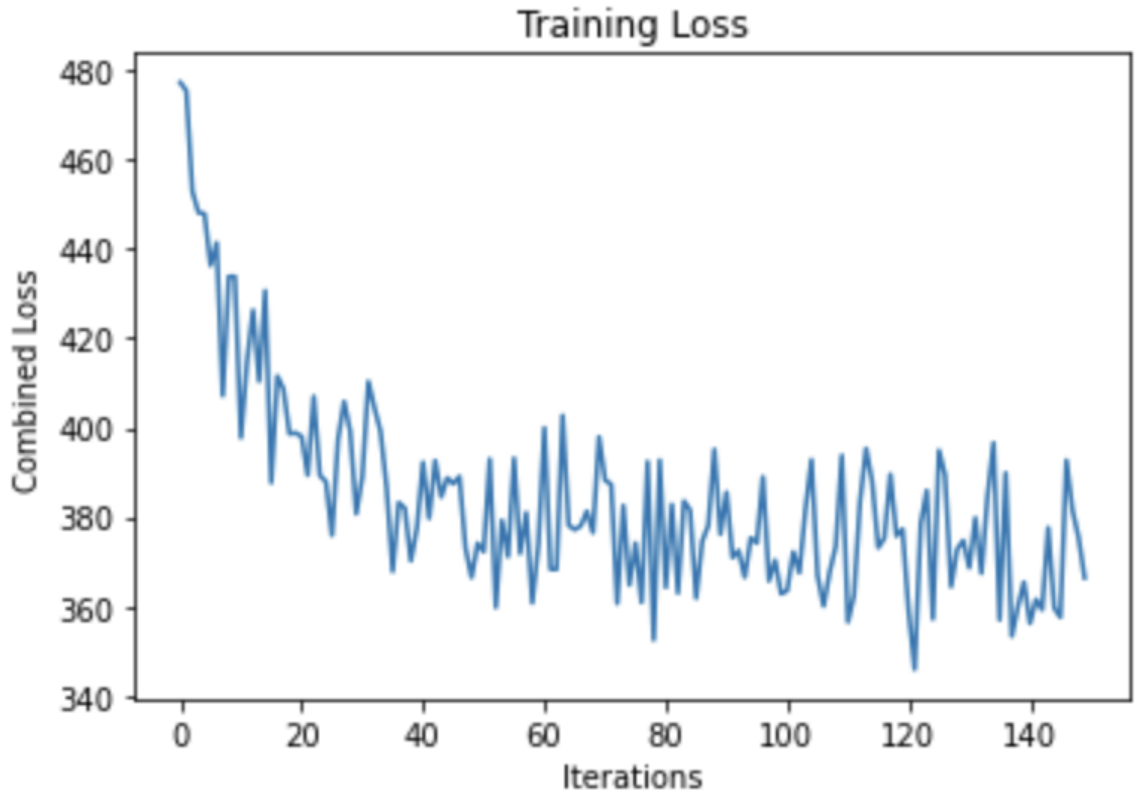
```
                numerator = torch.sum(output * mask_tensor)
                denominator = torch.sum(output * output) + to
rch.sum(mask_tensor * mask_tensor)
                dice_coefficient = 2 * numerator / (denominat
or + (1e-6))
                segmentation_loss_dice = 1 - dice_coefficient
                running_loss_segmentation_dice += segmentatio
n_loss_dice.item()

                # Combine MSE and Dice losses with weights
                combined_loss =  segmentation_loss_mse  + 20
* segmentation_loss_dice

                #combined_loss.backward()
                combined_loss.backward()
                optimizer.step()

                running_loss_segmentation_combined += combine
d_loss.item()

                if i%100==99:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segm
entation_combined / float(100)
                    print("[epoch=%d/%d, iter=%4d  elapsed_ti
me=%3d secs]   Combined loss: %.3f" % (epoch+1, self.dl_studi
o.epochs, i+1, elapsed_time, avg_loss_segmentation))
                    FILE1.write("%.3f\n" % avg_loss_segmentat
ion)
                    FILE1.flush()

                    running_loss_segmentation_mse = 0.0
                    running_loss_segmentation_dice = 0.0
                    running_loss_segmentation_combined = 0.0
```

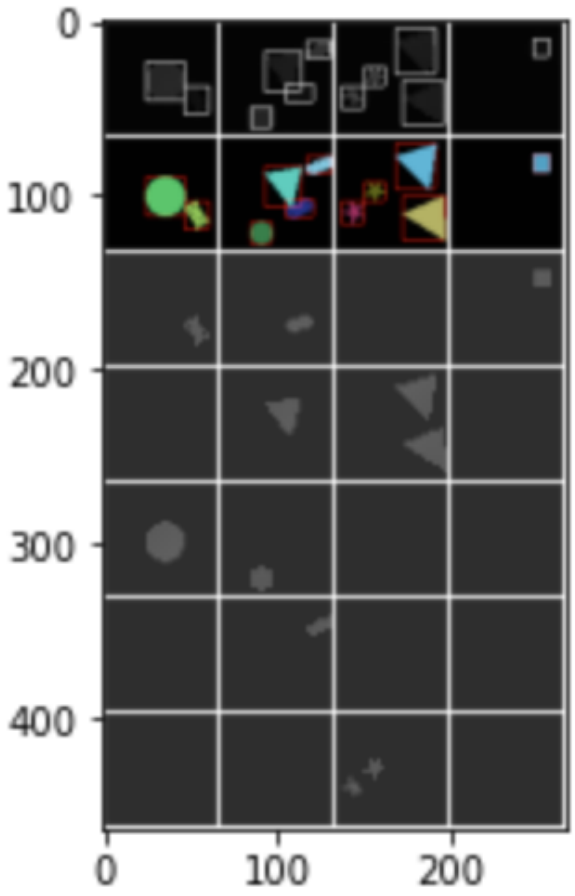
```
                    loss_values.append(avg_loss_segmentation)

        print("\nFinished Training\n")
        self.save_model(net)
            # Plot loss values versus iterations

        plt.plot(range(len(loss_values)), loss_values)
        plt.xlabel('Iterations')
        plt.ylabel('Combined Loss')
        plt.title('Training Loss')
        plt.show()
        return loss_values
```

MSE+ Dice combined loss curve:

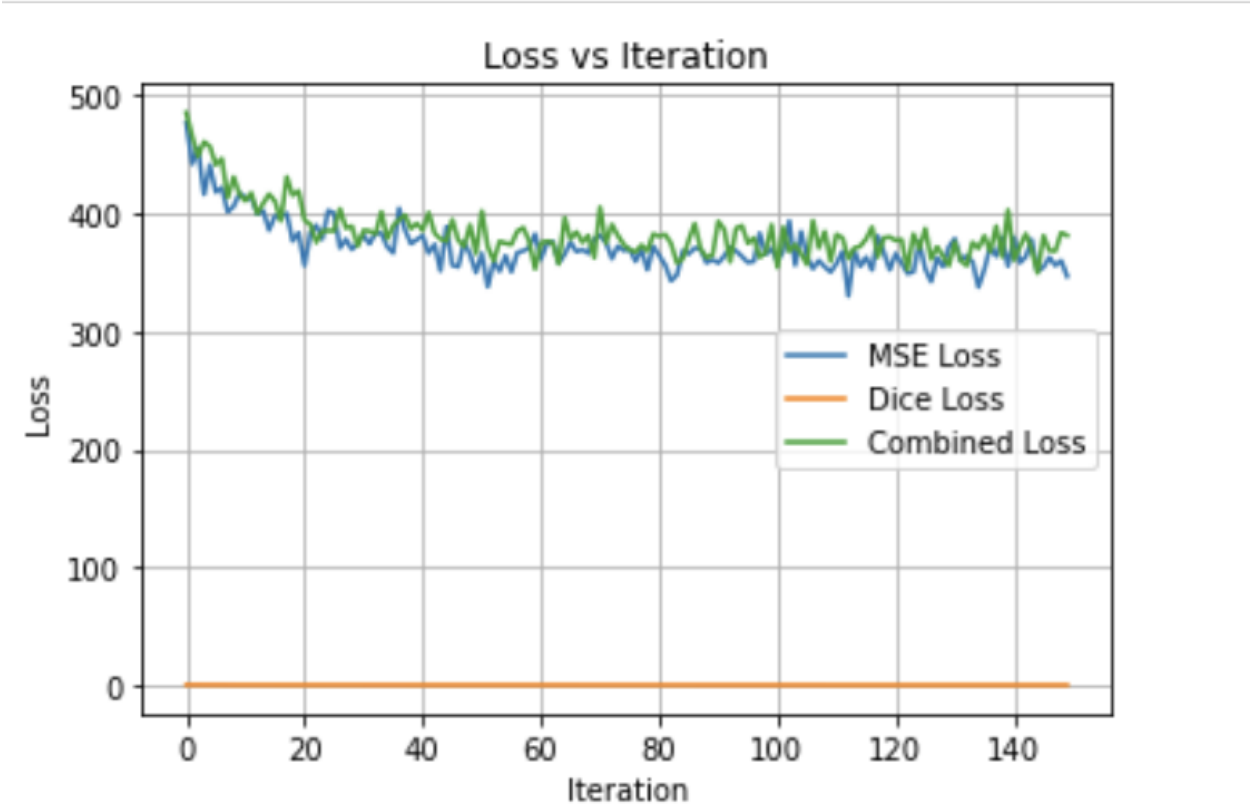
min loss value :345

max loss value : 477





Loss curve of all the 3 loss functions in one plot:



Factors Contributing to Performance Variations:

- **Dataset**: The choice of loss function can be influenced by the characteristics of the dataset, such as class imbalance, spatial coherence, and intensity variations. Datasets with significant class imbalance may benefit more from Dice loss, while a dataset with diverse intensity ranges may require MSE loss. Since MSE loss treats each pixel independently and does not consider spatial coherence or class imbalance. Dice loss does not explicitly penalize intensity differences between predicted and ground truth masks as it measures the overlap between them.
- **Training**: parameters such as learning rate, optimization algorithm and momentum values, and data transformation techniques also impact the training dynamics and convergence behavior when using different loss functions.

- **Combination of MSE + Dice Loss**: Combining MSE and Dice loss aims to leverage the strengths of both loss functions. MSE loss can help in preserving fine details and intensity information, while Dice loss can ensure better spatial overlap and handle class imbalance. However, the effectiveness of the combination depends on the scaling factors assigned to each loss term.
- If the evaluation metric focuses on pixel-wise accuracy, MSE loss may be more suitable. However, if the evaluation metric considers spatial overlap, Dice loss would be preferred, and accordingly weightage should be assigned to the proportion of loss term used if calculating the MSE + Dice combined loss.

Qualitative observations on the model test results for MSE loss vs. Dice+MSE loss

- The model trained with MSE loss struggles to preserve fine details and edges of the objects in dataset. Due to the nature of MSE loss, which penalizes pixel-wise differences heavily and the predicted masks appear blurred.
- The predicted masks exhibit better consistency in pixel intensities across the object when MSE loss is used but may lacks spatial accuracy and quality.
- Incorporating Dice loss alongside MSE loss improves the spatial accuracy of the predicted masks. Since Dice loss emphasizes spatial overlap between predicted and ground truth masks, better object shapes are observed in case of MSE + Dice combined loss.
- Dice loss mitigates the effects of class imbalance by providing a more balanced measure of segmentation performance. This results in more accurate segmentation of minority classes or instances, in case of combined loss function usage.
- Overall, the model trained with Dice+MSE loss visibly produce slightly better segmentation results with sharper object boundaries, improved spatial accuracy, and better handling of class imbalance compared to the model trained solely with MSE loss.

Source Code:

```
#imports
import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt
import skimage
import skimage.io as io
import cv2
from tqdm import tqdm
from PIL import Image
from pycocotools.coco import COCO
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as tvt
from torchvision.ops import box_iou, distance_box_iou, comple
te_box_iou



#Segmentation class inherited from Dr. Kak's DL studio code
class SemanticSegmentationMSE(DLStudio):

    def __init__(self, dl_studio, max_num_objects, dataserver
_train=None, dataserver_test=None, dataset_file_train=None, d
ataset_file_test=None):
        super().__init__()
        self.dl_studio = dl_studio
        self.max_num_objects = max_num_objects
        self.dataserver_train = dataserver_train
        self.dataserver_test = dataserver_test
```

```
    class SkipBlockDN(nn.Module):
        """
        This class for the skip connections in the downward l
eg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockDN
        """
        def __init__(self, in_ch, out_ch, downsample=False, s
kip_connections=True):
            super().__init__()
            self.downsample = downsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
            self.bn2 = nn.BatchNorm2d(out_ch)
            if downsample:
                self.downsampler = nn.Conv2d(in_ch, out_ch,
1, stride=2)
        def forward(self, x):
            identity = x
            out = self.convo1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
            if self.in_ch == self.out_ch:
                out = self.convo2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.downsample:
                out = self.downsampler(out)
```

```
                identity = self.downsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
                    out = out + identity
                else:
                    out = out + torch.cat((identity, identit
y), dim=1)
            return out


    class SkipBlockUP(nn.Module):
        """
        This class is for the skip connections in the upward
leg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockUP
        """
        def __init__(self, in_ch, out_ch, upsample=False, ski
p_connections=True):
            super().__init__()
            self.upsample = upsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
            self.bn2 = nn.BatchNorm2d(out_ch)
            if upsample:
                self.upsampler = nn.ConvTranspose2d(in_ch, ou
t_ch, 1, stride=2, dilation=2, output_padding=1, padding=0)
        def forward(self, x):
            identity = x
```

```
            out = self.convoT1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
            out  =  nn.ReLU(inplace=False)(out)
            if self.in_ch == self.out_ch:
                out = self.convoT2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.upsample:
                out = self.upsampler(out)
                identity = self.upsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
                    out = out + identity
                else:
                    out = out + identity[:,self.out_ch:,:,:]
            return out


    class mUnet(nn.Module):


        def __init__(self, skip_connections=True, depth=16):
            super().__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(DLStudio.SemanticSeg
mentation.SkipBlockDN(64, 64, skip_connections=skip_connectio
ns))
            self.skip64dsDN = DLStudio.SemanticSegmentation.S
kipBlockDN(64, 64,   downsample=True, skip_connections=skip_c
```

```
onnections)
            self.skip64to128DN = DLStudio.SemanticSegmentatio
n.SkipBlockDN(64, 128, skip_connections=skip_connections )
            self.skip128DN_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128DN_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockDN(128, 128, skip_connections=skip_connec
tions))
            self.skip128dsDN = DLStudio.SemanticSegmentation.
SkipBlockDN(128,128, downsample=True, skip_connections=skip_c
onnections)
            ##  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(DLStudio.SemanticSeg
mentation.SkipBlockUP(64, 64, skip_connections=skip_connectio
ns))
            self.skip64usUP = DLStudio.SemanticSegmentation.S
kipBlockUP(64, 64, upsample=True, skip_connections=skip_conne
ctions)
            self.skip128to64UP = DLStudio.SemanticSegmentatio
n.SkipBlockUP(128, 64, skip_connections=skip_connections )
            self.skip128UP_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128UP_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockUP(128, 128, skip_connections=skip_connec
tions))
            self.skip128usUP = DLStudio.SemanticSegmentation.
SkipBlockUP(128,128, upsample=True, skip_connections=skip_con
nections)
            self.conv_out = nn.ConvTranspose2d(64, 5, 3, stri
de=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.con
v_in(x)))
            for i,skip64 in enumerate(self.skip64DN_arr[:sel
f.depth//4]):
                x = skip64(x)

            num_channels_to_save1 = x.shape[1] // 2
            save_for_upside_1 = x[:,:num_channels_to_save
1,:,:].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_save
2,:,:].clone()
            x = self.skip64to128DN(x)
            for i,skip128 in enumerate(self.skip128DN_arr[:se
lf.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_save
3,:,:].clone()
            for i,skip128 in enumerate(self.skip128DN_arr[sel
f.depth//4:]):
                x = skip128(x)
            x = self.skip128dsDN(x)
            ## Coming up from the bottom of U on the other si
de:
            x = self.skip128usUP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.depth//4]):
```

```
                x = skip128(x)
            x[:,:num_channels_to_save3,:,:] =  save_for_upsid
e_3
            x = self.bn1UP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.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_upsid
e_2
            x = self.bn2UP(x)
            x = self.skip64usUP(x)
            for i,skip64 in enumerate(self.skip64UP_arr[:sel
f.depth//4]):
                x = skip64(x)
            x[:,:num_channels_to_save1,:,:] =  save_for_upsid
e_1
            x = self.conv_out(x)
            return x
    def load_PurdueShapes5MultiObject_dataset(self, dataserve
r_train, dataserver_test ):
        self.train_dataloader = torch.utils.data.DataLoader(d
ataserver_train,
                     batch_size=self.dl_studio.batch_size,shuf
fle=True, num_workers=4)
        self.test_dataloader = torch.utils.data.DataLoader(da
taserver_test,
                              batch_size=self.dl_studio.batch_si
ze,shuffle=False, num_workers=4)

    class SegmentationLoss(nn.Module):
        """
        I wrote this class before I switched to MSE loss.  I
```

```
am leaving it here
        in case I need to get back to it in the future.

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SegmentationLoss
        """
        def __init__(self, batch_size):
            super().__init__()
            self.batch_size = batch_size
        def forward(self, output, mask_tensor):
            composite_loss = torch.zeros(1,self.batch_size)
            mask_based_loss = torch.zeros(1,5)
            for idx in range(self.batch_size):
                outputh = output[idx,0,:,:]
                for mask_layer_idx in range(mask_tensor.shape
[0]):
                    mask = mask_tensor[idx,mask_layer_id
x,:,:]
                    element_wise = (outputh - mask)**2
                    mask_based_loss[0,mask_layer_idx] = torc
h.mean(element_wise)
                composite_loss[0,idx] = torch.sum(mask_based_
loss)
            return torch.sum(composite_loss) / self.batch_siz
e


    def run_code_for_training_for_semantic_segmentation(self,
net):
        filename_for_out1 = "performance_numbers_" + str(sel
f.dl_studio.epochs) + ".txt"
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)
        criterion1 = nn.MSELoss()
        optimizer = optim.SGD(net.parameters(),
```

```
                      lr=self.dl_studio.learning_rate, momentu
m=self.dl_studio.momentum)
        start_time = time.perf_counter()

        loss_values = []
        for epoch in range(self.dl_studio.epochs):
            print("")
            running_loss_segmentation = 0.0
            for i, data in enumerate(self.train_dataloader):
                im_tensor,mask_tensor,bbox_tensor =data['imag
e'],data['mask_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.dev
ice)
                mask_tensor = mask_tensor.type(torch.FloatTen
sor)
                mask_tensor = mask_tensor.to(self.dl_studio.d
evice)
                bbox_tensor = bbox_tensor.to(self.dl_studio.d
evice)
                optimizer.zero_grad()
                output = net(im_tensor)
                segmentation_loss = criterion1(output, mask_t
ensor)
                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_los
s.item()
                if i%100==99:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segm
entation / float(100)
                    print("[epoch=%d/%d, iter=%4d  elapsed_ti
me=%3d secs]   MSE loss: %.3f" % (epoch+1, self.dl_studio.epo
chs, i+1, elapsed_time, avg_loss_segmentation))
                    FILE1.write("%.3f\n" % avg_loss_segmentat
```

```
ion)
                    FILE1.flush()
                    running_loss_segmentation = 0.0

                    loss_values.append(avg_loss_segmentation)

        print("\nFinished Training\n")
        self.save_model(net)
            # Plot loss values versus iterations
        import matplotlib.pyplot as plt
        plt.plot(range(len(loss_values)), loss_values)
        plt.xlabel('Iterations')
        plt.ylabel('MSE Loss')
        plt.title('Training Loss')
        plt.show()
        return loss_values

    def save_model(self, model):
        '''
        Save the trained model to a disk file
        '''
        torch.save(model.state_dict(), self.dl_studio.path_sa
ved_model)
    def run_code_for_testing_semantic_segmentation(self, ne
t):
        net.load_state_dict(torch.load(self.dl_studio.path_sa
ved_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['imag
e'],data['mask_tensor'],data['bbox_tensor']
                if i % 50 == 0:
                    print("\n\n\n\nShowing output for test ba
```

```
tch %d: " % (i+1))
                    outputs = net(im_tensor)
                    ## In the statement below: 1st arg for ba
tch items, 2nd for channels, 3rd and 4th for image size
                    output_bw_tensor = torch.zeros(batch_siz
e,1,image_size[0],image_size[1], dtype=float)
                    for image_idx in range(batch_size):
                        for layer_idx in range(max_num_object
s):
                            for m in range(image_size[0]):
                                for n in range(image_size
[1]):
                                    output_bw_tensor[image_id
x,0,m,n]  =  torch.max( outputs[image_idx,:,m,n] )
                    display_tensor = torch.zeros(7 * batch_si
ze,3,image_size[0],image_size[1], dtype=float)
                    for idx in range(batch_size):
                        for bbox_idx in range(max_num_object
s):
                            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] = 2
55
```

```
                                im_tensor[idx,0,i1:i2,j2] = 2
55
                                im_tensor[idx,0,i1,j1:j2] = 2
55
                                im_tensor[idx,0,i2,j1:j2] = 2
55
                    display_tensor[:batch_size,:,:,:] = outpu
t_bw_tensor
                    display_tensor[batch_size:2*batch_siz
e,:,:,:] = im_tensor

                    for batch_im_idx in range(batch_size):
                        for mask_layer_idx in range(max_num_o
bjects):
                            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[batc
h_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[batc
h_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_i
dx,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_id
x,mask_layer_idx,:,:]
                        self.dl_studio.display_tensor_as_image(
                           torchvision.utils.make_grid(display_te
nsor, nrow=batch_size, normalize=True, padding=2, pad_value=1
0))
```

```
##Code inherited from Dr. Kak's DL studio code

##  semantic_segmentation.py

"""
This script should be your starting point if you wish to lear
n how to use the
mUnet neural network for semantic segmentation of images.  As
mentioned elsewhere in
the main documentation page, mUnet assigns an output channel
to each different type of
object that you wish to segment out from an image. So, given
a test image at the
input to the network, all you have to do is to examine each c
hannel at the output for
segmenting out the objects that correspond to that output cha
nnel.
"""



"""
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
"""


##  watch -d -n 0.5 nvidia-smi

from DLStudio import *
```

```
dls = DLStudio(

                  dataroot = "/home/skose/dataset/",
                  image_size = [64,64],
                  path_saved_model = "./saved_model_mse",
                  momentum = 0.9,
                  learning_rate = 1e-4,
                  epochs = 6,
                  batch_size = 4,
                  classes = ('rectangle','triangle','disk','o
val','star'),
                  use_gpu = True,
              )

segmenter = SemanticSegmentationMSE(
                  dl_studio = dls,
                  max_num_objects = 5,
              )

dataserver_train = DLStudio.SemanticSegmentation.PurdueShapes
5MultiObjectDataset(
                          train_or_test = 'train',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiO
bject-10000-train.gz",
                        )
dataserver_test = DLStudio.SemanticSegmentation.PurdueShapes5
MultiObjectDataset(
                          train_or_test = 'test',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiO
bject-1000-test.gz"
                        )
```

```
segmenter.dataserver_train = dataserver_train
segmenter.dataserver_test = dataserver_test

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_tr
ain, 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.par
ameters() 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_lay
ers)


mse_loss = segmenter.run_code_for_training_for_semantic_segme
ntation(model)


# In[56]:


segmenter.run_code_for_testing_semantic_segmentation(model)
```

```
#Segmentation class inherited from Dr. Kak's DL studio code
class SemanticSegmentationDice(DLStudio):

    def __init__(self, dl_studio, max_num_objects, dataserver
_train=None, dataserver_test=None, dataset_file_train=None, d
ataset_file_test=None):
        super().__init__()
        self.dl_studio = dl_studio
        self.max_num_objects = max_num_objects
        self.dataserver_train = dataserver_train
        self.dataserver_test = dataserver_test



    class SkipBlockDN(nn.Module):
        """
        This class for the skip connections in the downward l
eg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockDN
        """
        def __init__(self, in_ch, out_ch, downsample=False, s
kip_connections=True):
            super().__init__()
            self.downsample = downsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
```

```
            self.bn2 = nn.BatchNorm2d(out_ch)
            if downsample:
                self.downsampler = nn.Conv2d(in_ch, out_ch,
1, stride=2)
        def forward(self, x):
            identity = x
            out = self.convo1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
            if self.in_ch == self.out_ch:
                out = self.convo2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.downsample:
                out = self.downsampler(out)
                identity = self.downsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
                    out = out + identity
                else:
                    out = out + torch.cat((identity, identit
y), dim=1)
            return out


    class SkipBlockUP(nn.Module):
        """
        This class is for the skip connections in the upward
leg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockUP
        """
        def __init__(self, in_ch, out_ch, upsample=False, ski
p_connections=True):
            super().__init__()
```

```
            self.upsample = upsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
            self.bn2 = nn.BatchNorm2d(out_ch)
            if upsample:
                self.upsampler = nn.ConvTranspose2d(in_ch, ou
t_ch, 1, stride=2, dilation=2, output_padding=1, padding=0)
        def forward(self, x):
            identity = x
            out = self.convoT1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
            out  =  nn.ReLU(inplace=False)(out)
            if self.in_ch == self.out_ch:
                out = self.convoT2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.upsample:
                out = self.upsampler(out)
                identity = self.upsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
                    out = out + identity
                else:
                    out = out + identity[:,self.out_ch:,:,:]
            return out


    class mUnet(nn.Module):
```

```
        def __init__(self, skip_connections=True, depth=16):
            super().__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(DLStudio.SemanticSeg
mentation.SkipBlockDN(64, 64, skip_connections=skip_connectio
ns))
            self.skip64dsDN = DLStudio.SemanticSegmentation.S
kipBlockDN(64, 64,   downsample=True, skip_connections=skip_c
onnections)
            self.skip64to128DN = DLStudio.SemanticSegmentatio
n.SkipBlockDN(64, 128, skip_connections=skip_connections )
            self.skip128DN_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128DN_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockDN(128, 128, skip_connections=skip_connec
tions))
            self.skip128dsDN = DLStudio.SemanticSegmentation.
SkipBlockDN(128,128, downsample=True, skip_connections=skip_c
onnections)
            ##  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(DLStudio.SemanticSeg
mentation.SkipBlockUP(64, 64, skip_connections=skip_connectio
ns))
            self.skip64usUP = DLStudio.SemanticSegmentation.S
kipBlockUP(64, 64, upsample=True, skip_connections=skip_conne
```

```
ctions)
            self.skip128to64UP = DLStudio.SemanticSegmentatio
n.SkipBlockUP(128, 64, skip_connections=skip_connections )
            self.skip128UP_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128UP_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockUP(128, 128, skip_connections=skip_connec
tions))
            self.skip128usUP = DLStudio.SemanticSegmentation.
SkipBlockUP(128,128, upsample=True, skip_connections=skip_con
nections)
            self.conv_out = nn.ConvTranspose2d(64, 5, 3, stri
de=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.con
v_in(x)))
            for i,skip64 in enumerate(self.skip64DN_arr[:sel
f.depth//4]):
                x = skip64(x)

            num_channels_to_save1 = x.shape[1] // 2
            save_for_upside_1 = x[:,:num_channels_to_save
1,:,:].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_save
2,:,:].clone()
            x = self.skip64to128DN(x)
            for i,skip128 in enumerate(self.skip128DN_arr[:se
lf.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_save
3,:,:].clone()
            for i,skip128 in enumerate(self.skip128DN_arr[sel
f.depth//4:]):
                x = skip128(x)
            x = self.skip128dsDN(x)
            ## Coming up from the bottom of U on the other si
de:
            x = self.skip128usUP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.depth//4]):
                x = skip128(x)
            x[:,:num_channels_to_save3,:,:] =  save_for_upsid
e_3
            x = self.bn1UP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.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_upsid
e_2
            x = self.bn2UP(x)
            x = self.skip64usUP(x)
            for i,skip64 in enumerate(self.skip64UP_arr[:sel
f.depth//4]):
                x = skip64(x)
            x[:,:num_channels_to_save1,:,:] =  save_for_upsid
e_1
            x = self.conv_out(x)
```

```
            return x
    def load_PurdueShapes5MultiObject_dataset(self, dataserve
r_train, dataserver_test ):
        self.train_dataloader = torch.utils.data.DataLoader(d
ataserver_train,
                     batch_size=self.dl_studio.batch_size,shuf
fle=True, num_workers=4)
        self.test_dataloader = torch.utils.data.DataLoader(da
taserver_test,
                             batch_size=self.dl_studio.batch_si
ze,shuffle=False, num_workers=4)

    class SegmentationLoss(nn.Module):
        """
        I wrote this class before I switched to MSE loss.  I
am leaving it here
        in case I need to get back to it in the future.

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SegmentationLoss
        """
        def __init__(self, batch_size):
            super().__init__()
            self.batch_size = batch_size
        def forward(self, output, mask_tensor):
            composite_loss = torch.zeros(1,self.batch_size)
            mask_based_loss = torch.zeros(1,5)
            for idx in range(self.batch_size):
                outputh = output[idx,0,:,:]
                for mask_layer_idx in range(mask_tensor.shape
[0]):
                    mask = mask_tensor[idx,mask_layer_id
x,:,:]
                    element_wise = (outputh - mask)**2
                    mask_based_loss[0,mask_layer_idx] = torc
h.mean(element_wise)
```

```
                composite_loss[0,idx] = torch.sum(mask_based_
loss)
            return torch.sum(composite_loss) / self.batch_siz
e


    def run_code_for_training_for_semantic_segmentation(self,
net):
        filename_for_out1 = "performance_numbers_" + str(sel
f.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, momentu
m=self.dl_studio.momentum)
        start_time = time.perf_counter()

        loss_values = []
        for epoch in range(self.dl_studio.epochs):
            print("")
            running_loss_segmentation_dice = 0.0
            for i, data in enumerate(self.train_dataloader):
                im_tensor,mask_tensor,bbox_tensor =data['imag
e'],data['mask_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.dev
ice)
                mask_tensor = mask_tensor.type(torch.FloatTen
sor)
                mask_tensor = mask_tensor.to(self.dl_studio.d
evice)
                bbox_tensor = bbox_tensor.to(self.dl_studio.d
evice)
                optimizer.zero_grad()
                output = net(im_tensor)
```

```
                #calculation of Dice Loss:
                numerator = torch.sum(output * mask_tensor)
                denominator = torch.sum(output * output) + to
rch.sum(mask_tensor * mask_tensor)
                dice_coefficient = 2 * numerator / (denominat
or + (1e-6))
                segmentation_loss_dice = (1 - dice_coefficien
t)


                segmentation_loss_dice.backward()
                optimizer.step()
                running_loss_segmentation_dice += segmentatio
n_loss_dice.item()

                if i%100==99:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segm
entation_dice / float(100)
                    print("[epoch=%d/%d, iter=%4d  elapsed_ti
me=%3d secs]   Dice loss: %.3f" % (epoch+1, self.dl_studio.ep
ochs, i+1, elapsed_time, avg_loss_segmentation))
                    FILE1.write("%.3f\n" % avg_loss_segmentat
ion)
                    FILE1.flush()
                    running_loss_segmentation_dice = 0.0

                    loss_values.append(avg_loss_segmentation)

        print("\nFinished Training\n")
        self.save_model(net)
            # Plot loss values versus iterations
        import matplotlib.pyplot as plt
        plt.plot(range(len(loss_values)), loss_values)
```

```
        plt.xlabel('Iterations')
        plt.ylabel('Dice Loss')
        plt.title('Training Loss')
        plt.show()
        return loss_values

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

    def run_code_for_testing_semantic_segmentation(self, ne
t):
        net.load_state_dict(torch.load(self.dl_studio.path_sa
ved_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['imag
e'],data['mask_tensor'],data['bbox_tensor']
                if i % 50 == 0:
                    print("\n\n\n\nShowing output for test ba
tch %d: " % (i+1))
                    outputs = net(im_tensor)
                    ## In the statement below: 1st arg for ba
tch items, 2nd for channels, 3rd and 4th for image size
                    output_bw_tensor = torch.zeros(batch_siz
e,1,image_size[0],image_size[1], dtype=float)
                    for image_idx in range(batch_size):
                        for layer_idx in range(max_num_object
s):
                            for m in range(image_size[0]):
```

```
                                    for n in range(image_size
[1]):
                                        output_bw_tensor[image_id
x,0,m,n]  =  torch.max( outputs[image_idx,:,m,n] )
                    display_tensor = torch.zeros(7 * batch_si
ze,3,image_size[0],image_size[1], dtype=float)
                    for idx in range(batch_size):
                        for bbox_idx in range(max_num_object
s):
                            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] = 2
55
                                im_tensor[idx,0,i1:i2,j2] = 2
55
                                im_tensor[idx,0,i1,j1:j2] = 2
55
                                im_tensor[idx,0,i2,j1:j2] = 2
55
                    display_tensor[:batch_size,:,:,:] = outpu
t_bw_tensor
                    display_tensor[batch_size:2*batch_siz
e,:,:,:] = im_tensor
```

```
                    for batch_im_idx in range(batch_size):
                        for mask_layer_idx in range(max_num_o
bjects):
                            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[batc
h_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[batc
h_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_i
dx,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_id
x,mask_layer_idx,:,:]
                        self.dl_studio.display_tensor_as_image(
                           torchvision.utils.make_grid(display_te
nsor, nrow=batch_size, normalize=True, padding=2, pad_value=1
0))









##Code class inherited from Dr. Kak's DL studio code

##  semantic_segmentation.py

"""
This script should be your starting point if you wish to lear
n how to use the
mUnet neural network for semantic segmentation of images.  As
mentioned elsewhere in
```

```
the main documentation page, mUnet assigns an output channel
to each different type of
object that you wish to segment out from an image. So, given
a test image at the
input to the network, all you have to do is to examine each c
hannel at the output for
segmenting out the objects that correspond to that output cha
nnel.
"""

"""
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
"""


##  watch -d -n 0.5 nvidia-smi

from DLStudio import *

dls = DLStudio(

                  dataroot = "/home/skose/dataset/",
                  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','o
```

```
val','star'),
                  use_gpu = True,
              )

segmenter = SemanticSegmentationDice(
                  dl_studio = dls,
                  max_num_objects = 5,
              )

dataserver_train = DLStudio.SemanticSegmentation.PurdueShapes
5MultiObjectDataset(
                          train_or_test = 'train',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiO
bject-10000-train.gz",
                        )
dataserver_test = DLStudio.SemanticSegmentation.PurdueShapes5
MultiObjectDataset(
                          train_or_test = 'test',
                          dl_studio = dls,
                          segmenter = segmenter,
                          dataset_file = "PurdueShapes5MultiO
bject-1000-test.gz"
                        )
segmenter.dataserver_train = dataserver_train
segmenter.dataserver_test = dataserver_test

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_tr
ain, 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.par
ameters() 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_lay
ers)


dice_loss = segmenter.run_code_for_training_for_semantic_segm
entation(model)




segmenter.run_code_for_testing_semantic_segmentation(model)




#Segmentation class inherited from Dr. Kak's DL studio code
class SemanticSegmentationCombined(DLStudio):

    def __init__(self, dl_studio, max_num_objects, dataserver
_train=None, dataserver_test=None, dataset_file_train=None, d
ataset_file_test=None):
        super().__init__()
        self.dl_studio = dl_studio
        self.max_num_objects = max_num_objects
        self.dataserver_train = dataserver_train
        self.dataserver_test = dataserver_test



    class SkipBlockDN(nn.Module):
        """
```

```
        This class for the skip connections in the downward l
eg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockDN
        """
        def __init__(self, in_ch, out_ch, downsample=False, s
kip_connections=True):
            super().__init__()
            self.downsample = downsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=
1, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
            self.bn2 = nn.BatchNorm2d(out_ch)
            if downsample:
                self.downsampler = nn.Conv2d(in_ch, out_ch,
1, stride=2)
        def forward(self, x):
            identity = x
            out = self.convo1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
            if self.in_ch == self.out_ch:
                out = self.convo2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.downsample:
                out = self.downsampler(out)
                identity = self.downsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
```

```
                    out = out + identity
                else:
                    out = out + torch.cat((identity, identit
y), dim=1)
            return out


    class SkipBlockUP(nn.Module):
        """
        This class is for the skip connections in the upward
leg of the "U"

        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SkipBlockUP
        """
        def __init__(self, in_ch, out_ch, upsample=False, ski
p_connections=True):
            super().__init__()
            self.upsample = upsample
            self.skip_connections = skip_connections
            self.in_ch = in_ch
            self.out_ch = out_ch
            self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch,
3, padding=1)
            self.bn1 = nn.BatchNorm2d(out_ch)
            self.bn2 = nn.BatchNorm2d(out_ch)
            if upsample:
                self.upsampler = nn.ConvTranspose2d(in_ch, ou
t_ch, 1, stride=2, dilation=2, output_padding=1, padding=0)
        def forward(self, x):
            identity = x
            out = self.convoT1(x)
            out = self.bn1(out)
            out = nn.functional.relu(out)
```

```
            out  =  nn.ReLU(inplace=False)(out)
            if self.in_ch == self.out_ch:
                out = self.convoT2(out)
                out = self.bn2(out)
                out = nn.functional.relu(out)
            if self.upsample:
                out = self.upsampler(out)
                identity = self.upsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
                    out = out + identity
                else:
                    out = out + identity[:,self.out_ch:,:,:]
            return out


    class mUnet(nn.Module):


        def __init__(self, skip_connections=True, depth=16):
            super().__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(DLStudio.SemanticSeg
mentation.SkipBlockDN(64, 64, skip_connections=skip_connectio
ns))
            self.skip64dsDN = DLStudio.SemanticSegmentation.S
kipBlockDN(64, 64,   downsample=True, skip_connections=skip_c
onnections)
            self.skip64to128DN = DLStudio.SemanticSegmentatio
n.SkipBlockDN(64, 128, skip_connections=skip_connections )
```

```
            self.skip128DN_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128DN_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockDN(128, 128, skip_connections=skip_connec
tions))
            self.skip128dsDN = DLStudio.SemanticSegmentation.
SkipBlockDN(128,128, downsample=True, skip_connections=skip_c
onnections)
            ##  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(DLStudio.SemanticSeg
mentation.SkipBlockUP(64, 64, skip_connections=skip_connectio
ns))
            self.skip64usUP = DLStudio.SemanticSegmentation.S
kipBlockUP(64, 64, upsample=True, skip_connections=skip_conne
ctions)
            self.skip128to64UP = DLStudio.SemanticSegmentatio
n.SkipBlockUP(128, 64, skip_connections=skip_connections )
            self.skip128UP_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128UP_arr.append(DLStudio.SemanticSe
gmentation.SkipBlockUP(128, 128, skip_connections=skip_connec
tions))
            self.skip128usUP = DLStudio.SemanticSegmentation.
SkipBlockUP(128,128, upsample=True, skip_connections=skip_con
nections)
            self.conv_out = nn.ConvTranspose2d(64, 5, 3, stri
de=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.con
v_in(x)))
```

```
            for i,skip64 in enumerate(self.skip64DN_arr[:sel
f.depth//4]):
                x = skip64(x)

            num_channels_to_save1 = x.shape[1] // 2
            save_for_upside_1 = x[:,:num_channels_to_save
1,:,:].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_save
2,:,:].clone()
            x = self.skip64to128DN(x)
            for i,skip128 in enumerate(self.skip128DN_arr[:se
lf.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_save
3,:,:].clone()
            for i,skip128 in enumerate(self.skip128DN_arr[sel
f.depth//4:]):
                x = skip128(x)
            x = self.skip128dsDN(x)
            ## Coming up from the bottom of U on the other si
de:
            x = self.skip128usUP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.depth//4]):
                x = skip128(x)
            x[:,:num_channels_to_save3,:,:] =  save_for_upsid
e_3
```

```
            x = self.bn1UP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:se
lf.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_upsid
e_2
            x = self.bn2UP(x)
            x = self.skip64usUP(x)
            for i,skip64 in enumerate(self.skip64UP_arr[:sel
f.depth//4]):
                x = skip64(x)
            x[:,:num_channels_to_save1,:,:] =  save_for_upsid
e_1
            x = self.conv_out(x)
            return x
    def load_PurdueShapes5MultiObject_dataset(self, dataserve
r_train, dataserver_test ):
        self.train_dataloader = torch.utils.data.DataLoader(d
ataserver_train,
                     batch_size=self.dl_studio.batch_size,shuf
fle=True, num_workers=4)
        self.test_dataloader = torch.utils.data.DataLoader(da
taserver_test,
                             batch_size=self.dl_studio.batch_si
ze,shuffle=False, num_workers=4)

    class SegmentationLoss(nn.Module):
        """
        I wrote this class before I switched to MSE loss.  I
am leaving it here
        in case I need to get back to it in the future.
```

```
        Class Path:   DLStudio  ->  SemanticSegmentation  ->
SegmentationLoss
        """
        def __init__(self, batch_size):
            super().__init__()
            self.batch_size = batch_size
        def forward(self, output, mask_tensor):
            composite_loss = torch.zeros(1,self.batch_size)
            mask_based_loss = torch.zeros(1,5)
            for idx in range(self.batch_size):
                outputh = output[idx,0,:,:]
                for mask_layer_idx in range(mask_tensor.shape
[0]):
                    mask = mask_tensor[idx,mask_layer_id
x,:,:]
                    element_wise = (outputh - mask)**2
                    mask_based_loss[0,mask_layer_idx] = torc
h.mean(element_wise)
                composite_loss[0,idx] = torch.sum(mask_based_
loss)
            return torch.sum(composite_loss) / self.batch_siz
e


    def run_code_for_training_for_semantic_segmentation(self,
net):
        filename_for_out1 = "performance_numbers_" + str(sel
f.dl_studio.epochs) + ".txt"
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)
        criterion1 = nn.MSELoss()
        optimizer = optim.SGD(net.parameters(),
                     lr=self.dl_studio.learning_rate, momentu
m=self.dl_studio.momentum)
        start_time = time.perf_counter()
```

```
        loss_values = []
        for epoch in range(self.dl_studio.epochs):
            print("")

            running_loss_segmentation_mse = 0.0
            running_loss_segmentation_dice = 0.0
            running_loss_segmentation_combined = 0.0

            for i, data in enumerate(self.train_dataloader):
                im_tensor,mask_tensor,bbox_tensor =data['imag
e'],data['mask_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.dev
ice)
                mask_tensor = mask_tensor.type(torch.FloatTen
sor)
                mask_tensor = mask_tensor.to(self.dl_studio.d
evice)
                bbox_tensor = bbox_tensor.to(self.dl_studio.d
evice)
                optimizer.zero_grad()
                output = net(im_tensor)


                 # Calculate MSE loss

                segmentation_loss_mse = criterion1(output, ma
sk_tensor)
                running_loss_segmentation_mse += segmentation
_loss_mse.item()

                # Calculate Dice loss
                numerator = torch.sum(output * mask_tensor)
                denominator = torch.sum(output * output) + to
rch.sum(mask_tensor * mask_tensor)
                dice_coefficient = 2 * numerator / (denominat
```

```
or + (1e-6))
                segmentation_loss_dice = 1 - dice_coefficient
                running_loss_segmentation_dice += segmentatio
n_loss_dice.item()

                # Combine MSE and Dice losses with weights
                combined_loss =  segmentation_loss_mse  + 20
* segmentation_loss_dice

                #combined_loss.backward()
                combined_loss.backward()
                optimizer.step()

                running_loss_segmentation_combined += combine
d_loss.item()

                if i%100==99:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segm
entation_combined / float(100)
                    print("[epoch=%d/%d, iter=%4d  elapsed_ti
me=%3d secs]   Combined loss: %.3f" % (epoch+1, self.dl_studi
o.epochs, i+1, elapsed_time, avg_loss_segmentation))
                    FILE1.write("%.3f\n" % avg_loss_segmentat
ion)
                    FILE1.flush()

                    running_loss_segmentation_mse = 0.0
                    running_loss_segmentation_dice = 0.0
                    running_loss_segmentation_combined = 0.0

                    loss_values.append(avg_loss_segmentation)

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

```
            # Plot loss values versus iterations

        plt.plot(range(len(loss_values)), loss_values)
        plt.xlabel('Iterations')
        plt.ylabel('Combined Loss')
        plt.title('Training Loss')
        plt.show()
        return loss_values

    def save_model(self, model):
        '''
        Save the trained model to a disk file
        '''
        torch.save(model.state_dict(), self.dl_studio.path_sa
ved_model)
    def run_code_for_testing_semantic_segmentation(self, ne
t):
        net.load_state_dict(torch.load(self.dl_studio.path_sa
ved_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['imag
e'],data['mask_tensor'],data['bbox_tensor']
                if i % 50 == 0:
                    print("\n\n\n\nShowing output for test ba
tch %d: " % (i+1))
                    outputs = net(im_tensor)
                    ## In the statement below: 1st arg for ba
tch items, 2nd for channels, 3rd and 4th for image size
                    output_bw_tensor = torch.zeros(batch_siz
e,1,image_size[0],image_size[1], dtype=float)
                    for image_idx in range(batch_size):
                        for layer_idx in range(max_num_object
```

```
s):
                            for m in range(image_size[0]):
                                for n in range(image_size
[1]):
                                    output_bw_tensor[image_id
x,0,m,n]  =  torch.max( outputs[image_idx,:,m,n] )
                    display_tensor = torch.zeros(7 * batch_si
ze,3,image_size[0],image_size[1], dtype=float)
                    for idx in range(batch_size):
                        for bbox_idx in range(max_num_object
s):
                            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] = 2
55
                                im_tensor[idx,0,i1:i2,j2] = 2
55
                                im_tensor[idx,0,i1,j1:j2] = 2
55
                                im_tensor[idx,0,i2,j1:j2] = 2
55
                    display_tensor[:batch_size,:,:,:] = outpu
t_bw_tensor
```

```
                    display_tensor[batch_size:2*batch_siz
e,:,:,:] = im_tensor

                    for batch_im_idx in range(batch_size):
                        for mask_layer_idx in range(max_num_o
bjects):
                            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[batc
h_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[batc
h_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_i
dx,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_id
x,mask_layer_idx,:,:]
                        self.dl_studio.display_tensor_as_image(
                           torchvision.utils.make_grid(display_te
nsor, nrow=batch_size, normalize=True, padding=2, pad_value=1
0))




##Code class inherited from Dr. Kak's DL studio code

##  semantic_segmentation.py

"""
This script should be your starting point if you wish to lear
n how to use the
mUnet neural network for semantic segmentation of images.  As
mentioned elsewhere in
```

```
the main documentation page, mUnet assigns an output channel
to each different type of
object that you wish to segment out from an image. So, given
a test image at the
input to the network, all you have to do is to examine each c
hannel at the output for
segmenting out the objects that correspond to that output cha
nnel.
"""


"""
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
"""


##  watch -d -n 0.5 nvidia-smi

from DLStudio import *

dls = DLStudio(

                  dataroot = "/home/skose/dataset/",
                  image_size = [64,64],
                  path_saved_model = "./saved_model_combine
d",
                  momentum = 0.9,
                  learning_rate = 1e-4,
                  epochs = 6,
```

```
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','o
val','star'),
                    use_gpu = True,
                )

segmenter = SemanticSegmentationCombined(
                    dl_studio = dls,
                    max_num_objects = 5,
                )

dataserver_train = DLStudio.SemanticSegmentation.PurdueShapes
5MultiObjectDataset(
                            train_or_test = 'train',
                            dl_studio = dls,
                            segmenter = segmenter,
                            dataset_file = "PurdueShapes5MultiO
bject-10000-train.gz",
                           )
dataserver_test = DLStudio.SemanticSegmentation.PurdueShapes5
MultiObjectDataset(
                            train_or_test = 'test',
                            dl_studio = dls,
                            segmenter = segmenter,
                            dataset_file = "PurdueShapes5MultiO
bject-1000-test.gz"
                           )
segmenter.dataserver_train = dataserver_train
segmenter.dataserver_test = dataserver_test

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_tr
ain, 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.par
ameters() 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_lay
ers)


combined_loss = segmenter.run_code_for_training_for_semantic_
segmentation(model)




segmenter.run_code_for_testing_semantic_segmentation(model)




plt.plot(mse_loss, label='MSE Loss')
plt.plot(dice_loss, label='Dice Loss')
plt.plot(combined_loss, label='Combined Loss')
plt.title('Loss vs Iteration')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

EXTRA CREDIT HOMEWORK 7:

The directory structure of the Filtered COCO subset of images used is storing images and masks of the 3 classes with the applied area constraint filter in separate directories names images and mask for each of the training and validation/test dataset.

The mUnet Model code used:

```
class mUnet(nn.Module):

    def __init__(self, skip_connections=True, depth=16):
        super().__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=skip_connections))
        self.skip64dsDN = SkipBlockDN(64, 64,   downsample=Tr
ue, skip_connections=skip_connections)
        self.skip64to128DN = SkipBlockDN(64, 128, skip_connec
tions=skip_connections )
        self.skip128DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128DN_arr.append(SkipBlockDN(128, 128, s
kip_connections=skip_connections))
        self.skip128dsDN = SkipBlockDN(128,128, downsample=Tr
ue, skip_connections=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=skip_connections))
        self.skip64usUP = SkipBlockUP(64, 64, upsample=True,
skip_connections=skip_connections)
        self.skip128to64UP = SkipBlockUP(128, 64, skip_connec
tions=skip_connections )
        self.skip128UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128UP_arr.append(SkipBlockUP(128, 128, s
kip_connections=skip_connections))
        self.skip128usUP = SkipBlockUP(128,128, upsample=Tru
e, skip_connections=skip_connections)
        self.conv_out = nn.ConvTranspose2d(64, 1, 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.dep
th//4]):
            x = skip64(x)

        num_channels_to_save1 = x.shape[1] // 2
        save_for_upside_1 = x[:,:num_channels_to_save1,:,:].c
lone()
        x = self.skip64dsDN(x)
        for i,skip64 in enumerate(self.skip64DN_arr[self.dept
h//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,:,:].c
lone()
        x = self.skip64to128DN(x)
```

```
        for i,skip128 in enumerate(self.skip128DN_arr[:self.d
epth//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,:,:].c
lone()
        for i,skip128 in enumerate(self.skip128DN_arr[self.de
pth//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.d
epth//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.d
epth//4]):
            x = skip128(x)
        x = self.skip128to64UP(x)
        for i,skip64 in enumerate(self.skip64UP_arr[self.dept
h//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.dep
th//4]):
            x = skip64(x)
        x[:,:num_channels_to_save1,:,:] =  save_for_upside_1
        x = self.conv_out(x)
        return x
```

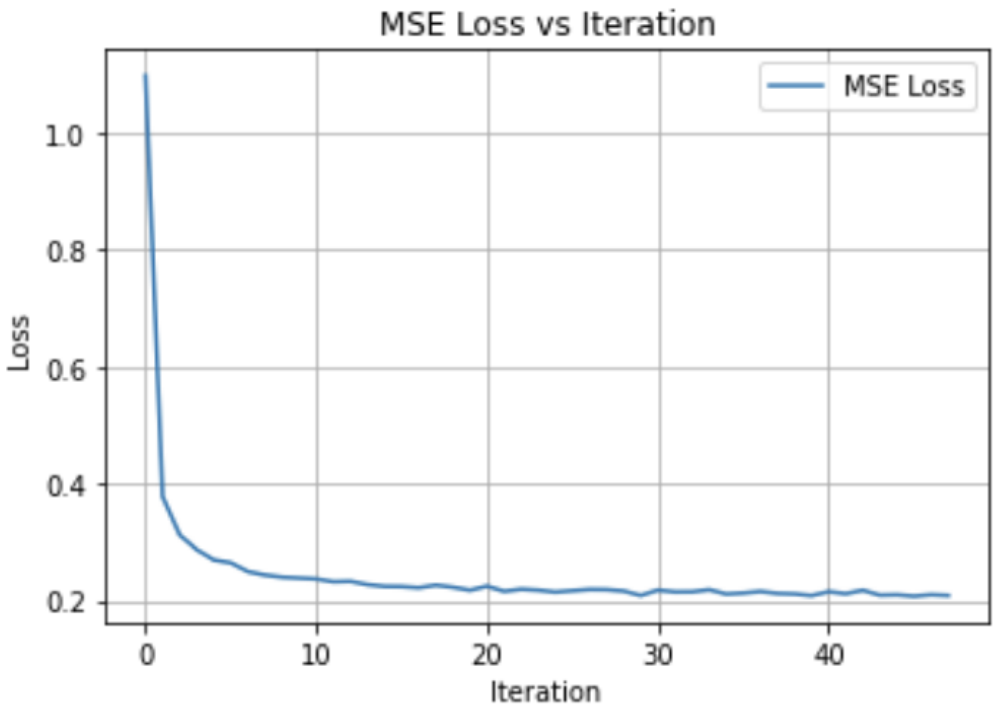
```
class SkipBlockDN(nn.Module):

    def __init__(self, in_ch, out_ch, downsample=False, skip_
connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, st
ride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
```

```
                out = out + torch.cat((identity, identity), d
im=1)
        return out


class SkipBlockUP(nn.Module):

    def __init__(self, in_ch, out_ch, upsample=False, skip_co
nnections=True):
        super().__init__()
        self.upsample = upsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if upsample:
            self.upsampler = nn.ConvTranspose2d(in_ch, out_c
h, 1, stride=2, dilation=2, output_padding=1, padding=0)
    def forward(self, x):
        identity = x
        out = self.convoT1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out  =  nn.ReLU(inplace=False)(out)
        if self.in_ch == self.out_ch:
            out = self.convoT2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.upsample:
            out = self.upsampler(out)
            identity = self.upsampler(identity)
```

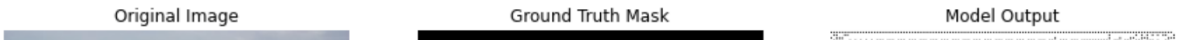
```
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + identity[:,self.out_ch:,:,:]
        return out
```

```
The number of learnable parameters in the model: 6996801


The number of layers in the model: 324
```

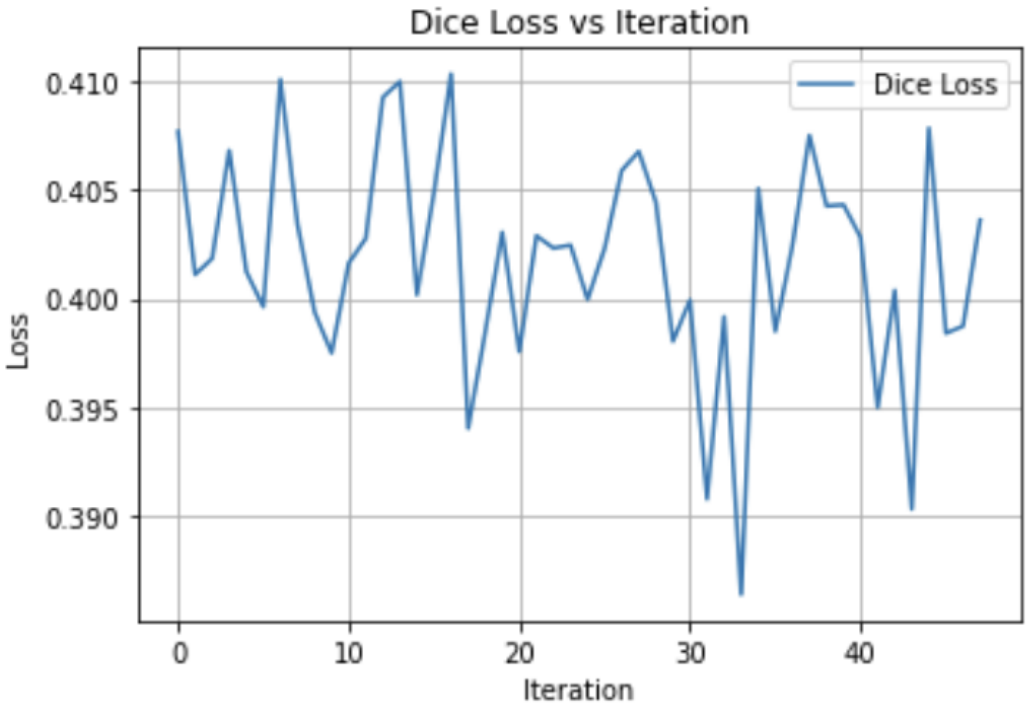
MSE loss curve:



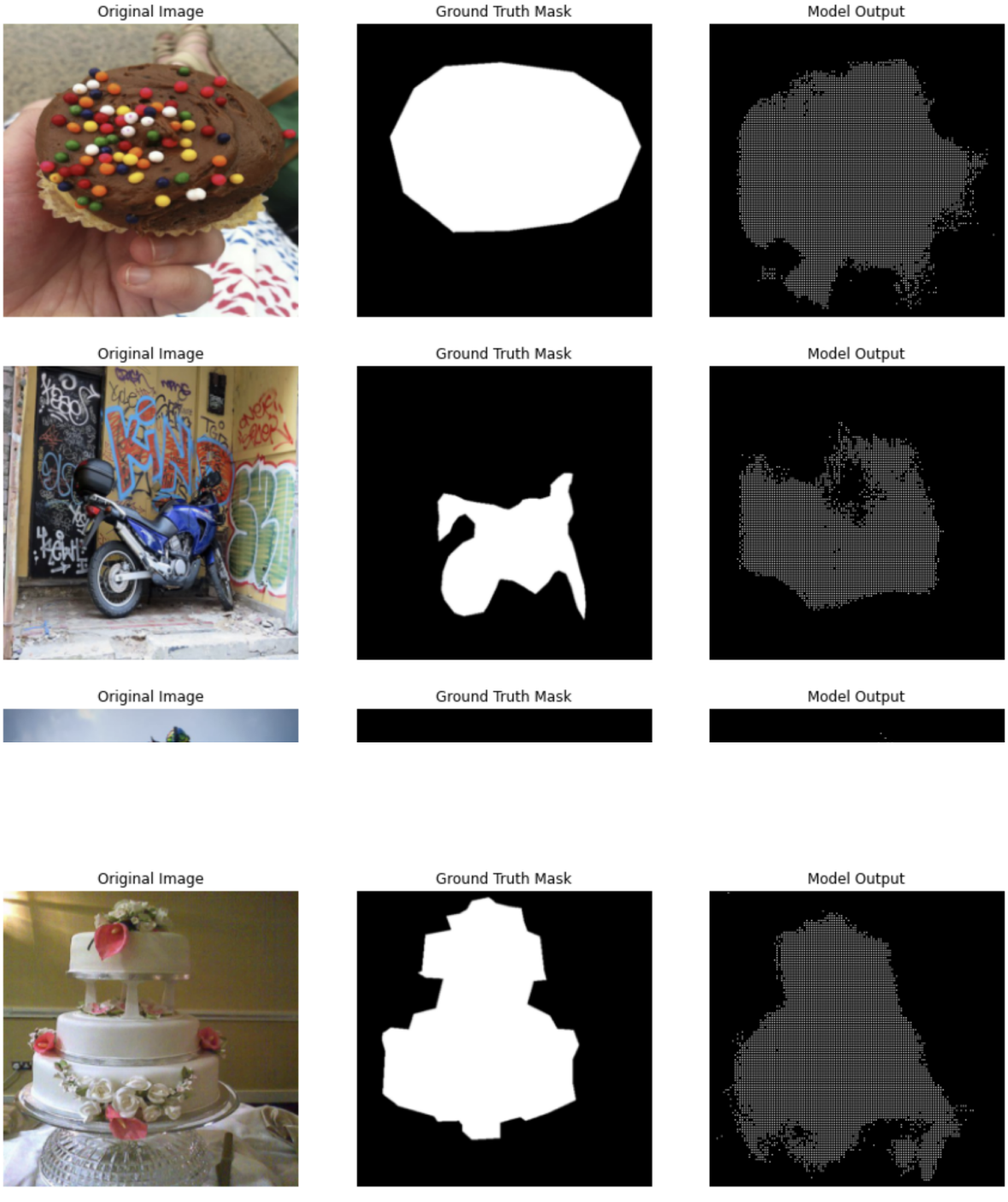
Test output:



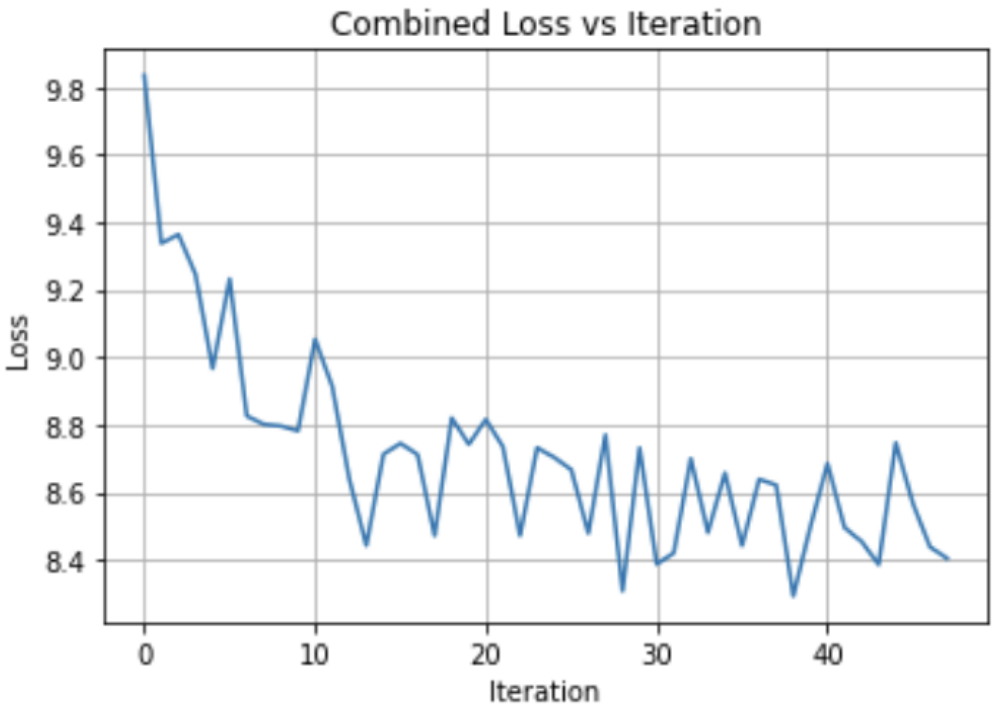
Dice loss curve:


Dice Loss vs Iteration
Dice Loss
0.410
0.405
0.400
0.395
0.390
Loss
0
10
20
30
40
Iteration

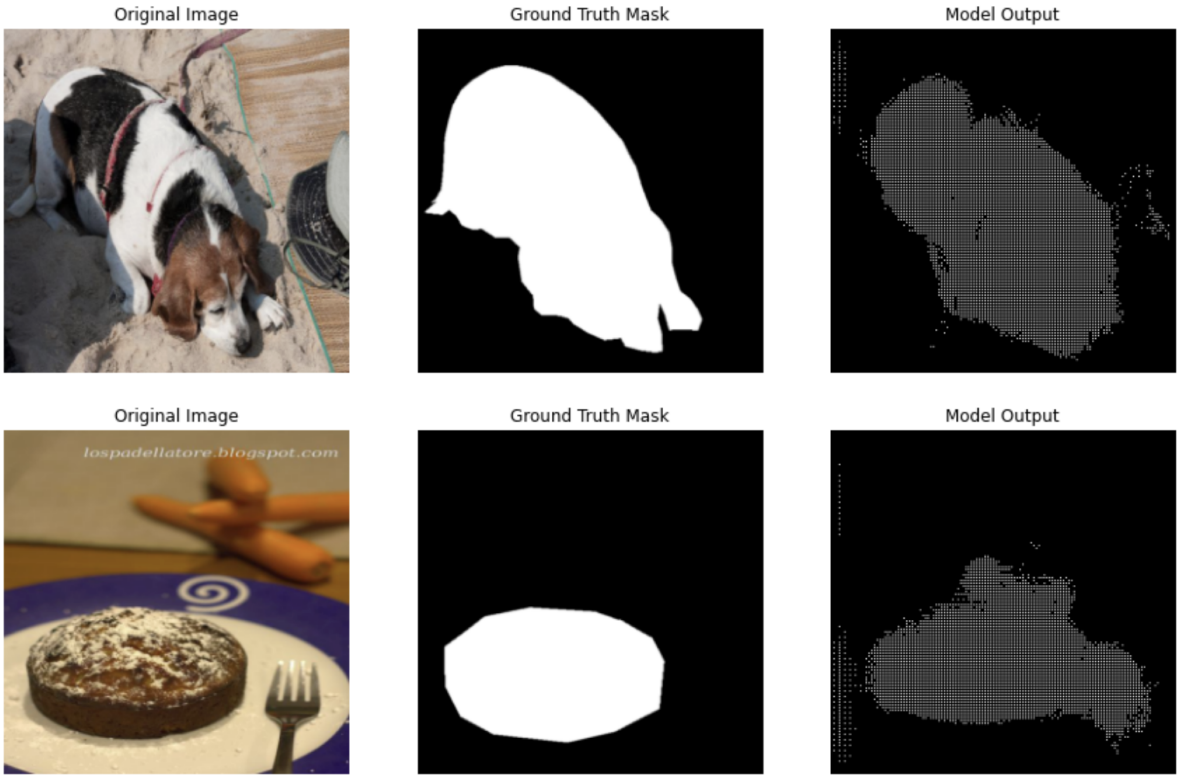
Output:

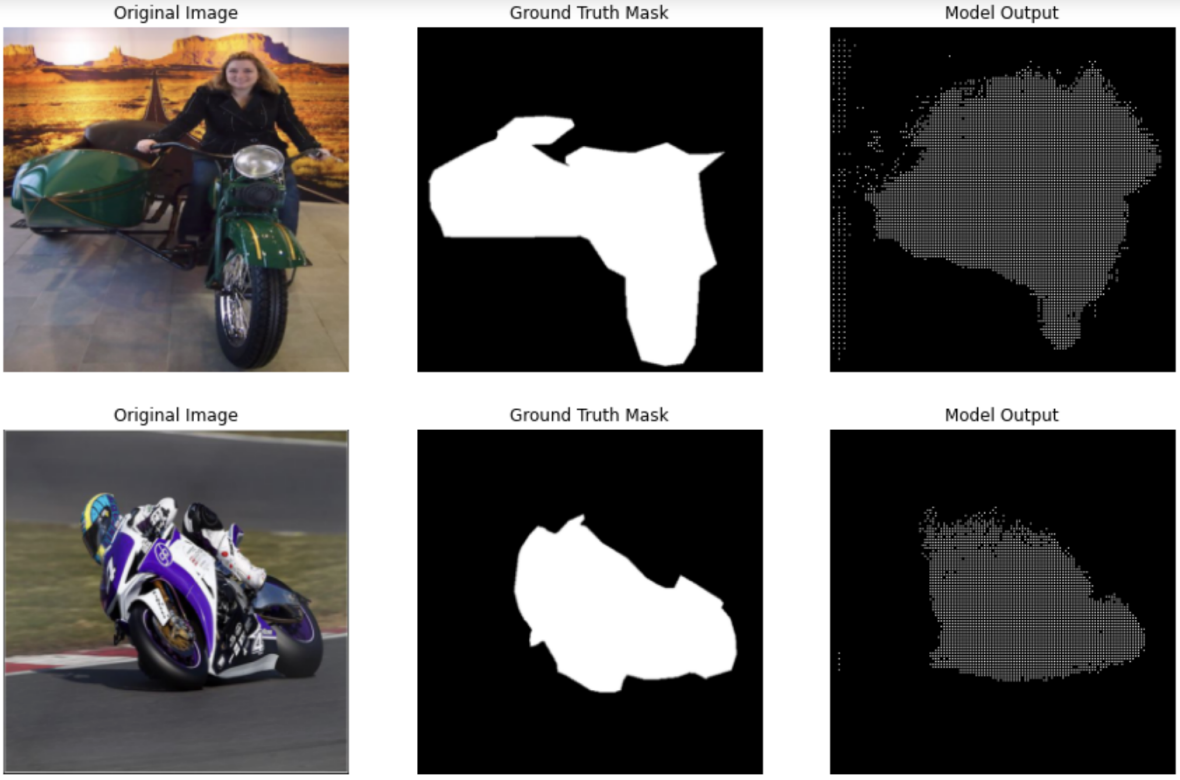


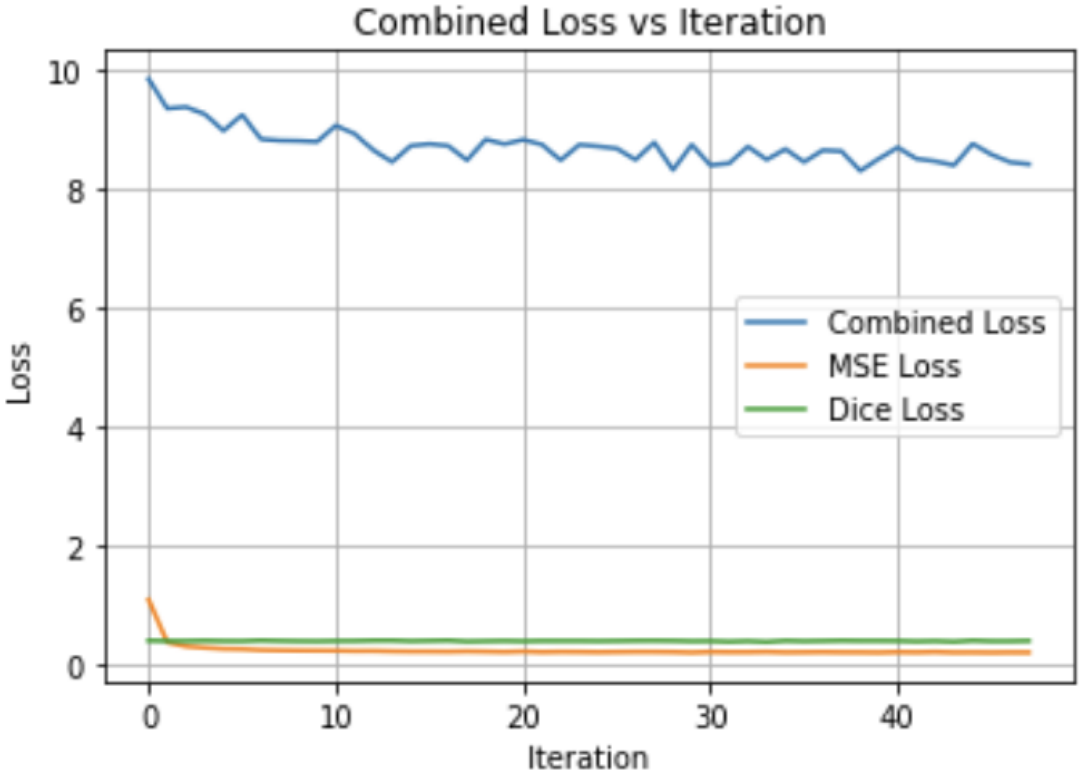
Combined loss curve:


Combined Loss vs Iteration
Loss
Iteration
9.8
9.6
9.4
9.2
9.0
8.8
8.6
8.4
0
10
20
30
40

Output:


Original Image
Ground Truth Mask
Model Output
Original Image
lospadellatore.blogspot.com
Ground Truth Mask
Model Output


Original Image
Ground Truth Mask
Model Output
Original Image
Ground Truth Mask
Model Output



Qualitative observations on the model test results for MSE loss vs. Dice+MSE loss for Coco subset data :

- The model trained with MSE loss struggles to preserve fine details and edges of the objects in the COCO dataset. Due to the nature of MSE loss, which penalizes pixel-wise differences heavily and the predicted masks appear blurred.
- The predicted masks exhibit better consistency in pixel intensities across the object when MSE loss is used but may lacks spatial accuracy and quality.
- Incorporating Dice loss alongside MSE loss improves the spatial accuracy of the predicted masks. Since Dice loss emphasizes spatial overlap between predicted and ground truth masks, better object shapes are observed in case of MSE + Dice combined loss.

- Since there is a class imbalance in the COCO dataset subset that we filtered, with cake class containing comparatively lesser number of images than dog and motorcycle class, Dice loss mitigates the effects of class imbalance by providing a more balanced measure of segmentation performance. This results in more accurate segmentation of minority classes or instances, in case of combined loss function usage.
- Overall, the model trained with Dice+MSE loss visibly produce slightly better segmentation results with sharper object boundaries, improved spatial accuracy, and better handling of class imbalance compared to the model trained solely with MSE loss.

Source Code for Extra Credit :

```
#imports
import os
import torch
import torchvision
import copy
import time
import random
import numpy as np
import requests
import matplotlib.pyplot as plt
import skimage
import skimage.io as io
import cv2
from tqdm import tqdm
from PIL import Image
from pycocotools.coco import COCO
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
```

```
#Dataset generation code same as Homework 6 code with slight
modifications for mask area contraints
class DatasetGenerator():
    def __init__(
        self, root_dir, annotation_path, classes, min_area=40
000  # 200x200 = 40000 pixels
    ):
        self.root_dir = root_dir
        self.annotation_path = annotation_path
        self.classes = classes
        self.min_area = min_area

        self.coco = COCO(annotation_path)
        self.catIds = self.coco.getCatIds(catNms=classes)
        self.categories = self.coco.loadCats(self.catIds)
        self.categories.sort(key=lambda x: x["id"])
        self.class_dir = {}

        self.coco_labels_inverse = {
            c["id"]: idx for idx, c in enumerate(self.categor
ies)
        }

    def create_dir(self):
        for c in self.classes:
            dir_path = os.path.join(self.root_dir, c)
            self.class_dir[c] = dir_path
            os.makedirs(dir_path, exist_ok=True)

    def download_images(self, download=True, val=False):
        img_paths = {c: [] for c in self.classes}
        img_masks = {c: [] for c in self.classes}

        for c in tqdm(self.classes):
```

```
            class_id = self.coco.getCatIds(c)
            img_ids = self.coco.getImgIds(catIds=class_id)
            images = self.coco.loadImgs(img_ids)

            for image in images:
                annIds = self.coco.getAnnIds(
                    imgIds=image["id"], catIds=class_id, iscr
owd=False
                )
                annotations = self.coco.loadAnns(annIds)

                valid_annotations = [
                    ann
                    for ann in annotations
                    if ann["area"] >= self.min_area
                ]

                if len(valid_annotations) == 1:
                    ann = valid_annotations[0]
                    mask = self.coco.annToMask(ann)
                    mask_path = os.path.join(
                        self.root_dir, c, image["file_name"].
replace(".jpg", "_mask.png")
                    )
                    # Convert mask array to binary mask with
0 for background and 1 for mask
                    binary_mask = (mask > 0).astype(np.uint8)
* 255

                    # Create PIL image from the binary mask a
nd save it
                    Image.fromarray(binary_mask, mode='L').sa
ve(mask_path)

                    img_path = os.path.join(self.root_dir, c,
image["file_name"])
```

```
                    if download:
                        if self.download_image(img_path, imag
e["coco_url"]):
                            self.convert_image(img_path)
                            img_paths[c].append(img_path)
                            img_masks[c].append(mask_path)
                    else:
                        img_paths[c].append(img_path)
                        img_masks[c].append(mask_path)

        return img_paths, img_masks

    # Download image from URL using requests
    def download_image(self, path, url):
        try:
            img_data = requests.get(url).content
            with open(path, "wb") as f:
                f.write(img_data)
            return True
        except Exception as e:
            print(f"Caught exception: {e}")
        return False

    # Resize image
    def convert_image(self, path):
        im = Image.open(path)
        if im.mode != "RGB":
            im = im.convert(mode="RGB")
            im = im.resize((256,256)) # resize the image to 2
56x256 size before downloading
        im.save(path)
```

```
classes = ['cake', 'dog', 'motorcycle']

# Download training images
train_downloader = DatasetGenerator('/Users/skose/Downloads/C
ocoDatasetTrain',
            '/Users/skose/Downloads/CocoDataset/annotations/i
nstances_train2017.json',
            classes)

train_downloader.create_dir()
train_img_paths, train_img_masks = train_downloader.download_
images(download=True)




# Download validation images
test_downloader = DatasetGenerator('/Users/skose/Downloads/Co
coDatasetTest',
            '/Users/skose/Downloads/CocoDataset/annotations/i
nstances_val2017.json',
            classes)

test_downloader.create_dir()
test_img_paths, test_img_masks = test_downloader.download_ima
ges(download=True, val=True)




#Custom Dataset Class
class CocoDatasetSubset(torch.utils.data.Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
```

```
        self.image_dir = os.path.join(root_dir, 'images')
        self.mask_dir = os.path.join(root_dir, 'mask')
        self.image_files = os.listdir(self.image_dir)

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

    def __getitem__(self, idx):
        img_name = self.image_files[idx]
        img_path = os.path.join(self.image_dir, img_name)
        mask_path = os.path.join(self.mask_dir, img_name.spli
t('.')[0] + '_mask.png')

        # Open image and mask
        image = Image.open(img_path).convert('RGB')
        mask = Image.open(mask_path).convert('L')

        if self.transform:
            image = self.transform(image)
            mask = self.transform(mask)

        sample = {'image' : image,
                  'mask'  : mask}

        return sample




#mUnet architecture class inherited from Dr. Kak's DL studio
module mUnet class code
class mUnet(nn.Module):

    def __init__(self, skip_connections=True, depth=16):
        super().__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=skip_connections))
        self.skip64dsDN = SkipBlockDN(64, 64,   downsample=Tr
ue, skip_connections=skip_connections)
        self.skip64to128DN = SkipBlockDN(64, 128, skip_connec
tions=skip_connections )
        self.skip128DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128DN_arr.append(SkipBlockDN(128, 128, s
kip_connections=skip_connections))
        self.skip128dsDN = SkipBlockDN(128,128, downsample=Tr
ue, skip_connections=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=skip_connections))
        self.skip64usUP = SkipBlockUP(64, 64, upsample=True,
skip_connections=skip_connections)
        self.skip128to64UP = SkipBlockUP(128, 64, skip_connec
tions=skip_connections )
        self.skip128UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128UP_arr.append(SkipBlockUP(128, 128, s
kip_connections=skip_connections))
        self.skip128usUP = SkipBlockUP(128,128, upsample=Tru
e, skip_connections=skip_connections)
```

```
        self.conv_out = nn.ConvTranspose2d(64, 1, 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.dep
th//4]):
            x = skip64(x)

        num_channels_to_save1 = x.shape[1] // 2
        save_for_upside_1 = x[:,:num_channels_to_save1,:,:].c
lone()
        x = self.skip64dsDN(x)
        for i,skip64 in enumerate(self.skip64DN_arr[self.dept
h//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,:,:].c
lone()
        x = self.skip64to128DN(x)
        for i,skip128 in enumerate(self.skip128DN_arr[:self.d
epth//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,:,:].c
lone()
        for i,skip128 in enumerate(self.skip128DN_arr[self.de
pth//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.d
epth//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.d
epth//4]):
            x = skip128(x)
        x = self.skip128to64UP(x)
        for i,skip64 in enumerate(self.skip64UP_arr[self.dept
h//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.dep
th//4]):
            x = skip64(x)
        x[:,:num_channels_to_save1,:,:] =  save_for_upside_1
        x = self.conv_out(x)
        return x



class SkipBlockDN(nn.Module):

    def __init__(self, in_ch, out_ch, downsample=False, skip_
connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
```

```
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, st
ride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + torch.cat((identity, identity), d
im=1)
        return out


class SkipBlockUP(nn.Module):

    def __init__(self, in_ch, out_ch, upsample=False, skip_co
nnections=True):
        super().__init__()
        self.upsample = upsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
```

```
        self.out_ch = out_ch
        self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch, 3, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if upsample:
            self.upsampler = nn.ConvTranspose2d(in_ch, out_c
h, 1, stride=2, dilation=2, output_padding=1, padding=0)
    def forward(self, x):
        identity = x
        out = self.convoT1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out  =  nn.ReLU(inplace=False)(out)
        if self.in_ch == self.out_ch:
            out = self.convoT2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.upsample:
            out = self.upsampler(out)
            identity = self.upsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + identity[:,self.out_ch:,:,:]
        return out




# Define transformations
transform = transforms.Compose([
  transforms.Resize((256, 256)),
```

```
 transforms.ToTensor()
])

# Create dataset instance
train_dataset = CocoDatasetSubset(root_dir='/home/skose/CocoT
rainDataset2', transform=transform)

test_dataset = CocoDatasetSubset(root_dir='/home/skose/CocoDa
tasetVal2', transform=transform)
#Create Dataloaders
train_dataloader = torch.utils.data.DataLoader(train_dataset,
batch_size=4, shuffle=True, num_workers=4)

test_dataloader = torch.utils.data.DataLoader(test_dataset, b
atch_size=4, shuffle=True, num_workers=4)




model = mUnet(skip_connections=True, depth=16)

number_of_learnable_params = sum(p.numel() for p in model.par
ameters() 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_lay
ers)




device = torch.device('cuda') if torch.cuda.is_available() el
se torch.device('cpu')
```

```
device




#Training loop similar to Dr. Kak's DL studio code
def run_code_for_training_for_semantic_segmentation_MSE(net,
data_loader, device, epochs=1):
    filename_for_out1 = "performance_numbers_" + str(epochs)
+ ".txt"
    FILE1 = open(filename_for_out1, 'w')
    net = copy.deepcopy(net)
    net = net.to(device)
    criterion1 = nn.MSELoss()
    optimizer = torch.optim.SGD(net.parameters(),
                 lr=1e-4, momentum=0.9)
    start_time = time.perf_counter()
    loss_values = []
    for epoch in range(epochs):
        print("")
        running_loss_segmentation = 0.0
        for i, data in enumerate(data_loader):
            im_tensor,mask_tensor =data['image'],data['mask']
            im_tensor   = im_tensor.to(device)
            mask_tensor = mask_tensor.type(torch.FloatTensor)
            mask_tensor = mask_tensor.to(device)

            optimizer.zero_grad()
            output = net(im_tensor)

            segmentation_loss = criterion1(output, mask_tenso
r)
            segmentation_loss.backward()
            optimizer.step()
            running_loss_segmentation += segmentation_loss.it
```

```
em()

            if i%100==99:
                current_time = time.perf_counter()
                elapsed_time = current_time - start_time
                avg_loss_segmentation = (running_loss_segment
ation / float(100))
                print("[epoch=%d/%d, iter=%4d  elapsed_time=%
3d secs]   MSE loss: %.3f" % (epoch+1,epochs, i+1, elapsed_ti
me, avg_loss_segmentation))
                FILE1.write("%.3f\n" % avg_loss_segmentation)
                FILE1.flush()
                running_loss_segmentation = 0.0
                loss_values.append(avg_loss_segmentation)

    FILE1.close()
    print("\nFinished Training\n")
    torch.save(net.state_dict(), "./saveModel0")
    return loss_values




mse_loss = run_code_for_training_for_semantic_segmentation_MS
E(net=model,data_loader=train_dataloader,
                                                        de
vice=device,epochs= 6)



def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
#Testing loop similar to Dr. Kak's DL studio code
def run_code_for_testing_semantic_segmentation_MSE(net, test_
dataloader):
    # Load the trained model
    net.load_state_dict(torch.load("/home/skose/saveModel0"))

    # Define batch size
    batch_size = 4

    # Loop over the entire dataloader
    for j, data in enumerate(test_dataloader):
        # Get images and masks from the data
        images, masks = data['image'], data['mask']

        # Perform inference using the model
        outputs = net(images)

        # Print output every 10 iterations
        if j % 10 == 0:
            print("\n\n\n\nShowing output new for test batch
%d: " % (j+1))

        # Plot images, masks, and network outputs for each sa
mple in the batch
        for i in range(batch_size):
            image_np = images[i].permute(1, 2, 0).numpy()
            mask_np = masks[i].squeeze().numpy()
            output_np = outputs[i].permute(1, 2, 0).detach().
numpy().squeeze()

            plt.figure(figsize=(15, 5))

            # Plot the original image
            plt.subplot(1, 3, 1)
```

```
            plt.imshow(image_np)
            plt.title('Original Image')
            plt.axis('off')

            # Plot the ground truth mask
            plt.subplot(1, 3, 2)
            plt.imshow(mask_np, cmap='gray')
            plt.title('Ground Truth Mask')
            plt.axis('off')

            # Plot the network output
            plt.subplot(1, 3, 3)
            # Normalize the output_np to the range [0, 1]
            output_np_normalized = (output_np - np.min(output
_np)) / (np.max(output_np) - np.min(output_np))
            # Threshold value
            threshold = 0.5
            # Apply thresholding
            output_mask_bw = np.where(output_np_normalized >=
threshold, 1.0, 0.0)

            plt.imshow(output_mask_bw, cmap='gray')
            plt.title('Model Output')
            plt.axis('off')

            plt.show()

# Call the function with your model and test dataloader
run_code_for_testing_semantic_segmentation_MSE(model, test_da
taloader)



plt.plot(mse_loss, label='MSE Loss')
plt.title('MSE Loss vs Iteration')
plt.xlabel('Iteration')
```

```
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()




#For Dice Loss
def run_code_for_training_for_semantic_segmentation_dice(net,
data_loader, device, epochs=1):
    filename_for_out1 = "performance_numbers_" + str(epochs)
+ ".txt"
    FILE1 = open(filename_for_out1, 'w')
    net = copy.deepcopy(net)
    net = net.to(device)
    criterion1 = nn.MSELoss()

    optimizer = torch.optim.SGD(net.parameters(),
                lr=1e-4, momentum=0.9)
    start_time = time.perf_counter()
    loss_values = []
    for epoch in range(epochs):
        print("")
        running_loss_segmentation = 0.0
        for i, data in enumerate(data_loader):
            im_tensor,mask_tensor =data['image'],data['mask']
            im_tensor   = im_tensor.to(device)
            mask_tensor = mask_tensor.type(torch.FloatTensor)
            mask_tensor = mask_tensor.to(device)
            optimizer.zero_grad()
            output = net(im_tensor)

            numerator = torch.sum(output * mask_tensor)
            denominator = torch.sum(output * output) + torch.
```

```
sum(mask_tensor * mask_tensor)
            dice_coefficient = 2 * numerator / (denominator +
(1e-6))
            segmentation_loss_dice = 1 - dice_coefficient

            running_loss_segmentation += segmentation_loss_di
ce.item()
            segmentation_loss_dice.backward()
            optimizer.step()

            if i%100==99:
                current_time = time.perf_counter()
                elapsed_time = current_time - start_time
                avg_loss_segmentation = (running_loss_segment
ation / float(100))
                print("[epoch=%d/%d, iter=%4d  elapsed_time=%
3d secs]   Dice loss: %.3f" % (epoch+1,epochs, i+1, elapsed_t
ime, avg_loss_segmentation))
                FILE1.write("%.3f\n" % avg_loss_segmentation)
                FILE1.flush()
                running_loss_segmentation = 0.0
                loss_values.append(avg_loss_segmentation)

    FILE1.close()
    print("\nFinished Training\n")
    torch.save(net.state_dict(), "./saveModel1")
    return loss_values




dice_loss = run_code_for_training_for_semantic_segmentation_d
ice(net=model,data_loader=train_dataloader,
                                                        de
vice=device,epochs= 6)
```

```
def run_code_for_testing_semantic_segmentation_dice(net, test
_dataloader):
    # Load the trained model
    net.load_state_dict(torch.load("/home/skose/saveModel1"))

    # Define batch size
    batch_size = 4

    # Loop over the entire dataloader
    for j, data in enumerate(test_dataloader):
        # Get images and masks from the data
        images, masks = data['image'], data['mask']

        # Perform inference using the model
        outputs = net(images)

        # Print output every 10 iterations
        if j % 10 == 0:
            print("\n\n\n\nShowing output new for test batch
%d: " % (j+1))

        # Plot images, masks, and network outputs for each sa
mple in the batch
        for i in range(batch_size):
            image_np = images[i].permute(1, 2, 0).numpy()
            mask_np = masks[i].squeeze().numpy()
            output_np = outputs[i].permute(1, 2, 0).detach().
numpy().squeeze()

            plt.figure(figsize=(15, 5))
```

```
            # Plot the original image
            plt.subplot(1, 3, 1)
            plt.imshow(image_np)
            plt.title('Original Image')
            plt.axis('off')

            # Plot the ground truth mask
            plt.subplot(1, 3, 2)
            plt.imshow(mask_np, cmap='gray')
            plt.title('Ground Truth Mask')
            plt.axis('off')

            # Plot the network output
            plt.subplot(1, 3, 3)
            # Normalize the output_np to the range [0, 1]
            output_np_normalized = (output_np - np.min(output
_np)) / (np.max(output_np) - np.min(output_np))
            # Threshold value
            threshold = 0.5
            # Apply thresholding
            output_mask_bw = np.where(output_np_normalized >=
threshold, 1.0, 0.0)

            plt.imshow(output_mask_bw, cmap='gray')
            plt.title('Model Output')
            plt.axis('off')

            plt.show()

# Call the function with your model and test dataloader
run_code_for_testing_semantic_segmentation_dice(model, test_d
ataloader)
```

```
plt.plot(dice_loss, label='Dice Loss')
plt.title('Dice Loss vs Iteration')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()




#For Combined loss
def run_code_for_training_for_semantic_segmentation_combined
(net, data_loader, device, epochs=1):
    filename_for_out1 = "performance_numbers_" + str(epochs)
+ ".txt"
    FILE1 = open(filename_for_out1, 'w')
    net = copy.deepcopy(net)
    net = net.to(device)
    criterion1 = nn.MSELoss()

    optimizer = torch.optim.SGD(net.parameters(),
                 lr=1e-4, momentum=0.9)
    start_time = time.perf_counter()
    loss_values = []
    for epoch in range(epochs):
        print("")
        running_loss_segmentation_mse = 0.0
        running_loss_segmentation_dice = 0.0
        running_loss_segmentation_combined = 0.0
        for i, data in enumerate(data_loader):
            im_tensor,mask_tensor =data['image'],data['mask']
            im_tensor   = im_tensor.to(device)
            mask_tensor = mask_tensor.type(torch.FloatTensor)
```

```
            mask_tensor = mask_tensor.to(device)
            optimizer.zero_grad()
            output = net(im_tensor)

            # Calculate MSE loss

            segmentation_loss_mse = criterion1(output, mask_t
ensor)
            running_loss_segmentation_mse += segmentation_los
s_mse.item()

            # Calculate Dice loss
            numerator = torch.sum(output * mask_tensor)
            denominator = torch.sum(output * output) + torch.
sum(mask_tensor * mask_tensor)
            dice_coefficient = 2 * numerator / (denominator +
(1e-6))
            segmentation_loss_dice = 1 - dice_coefficient
            running_loss_segmentation_dice += segmentation_lo
ss_dice.item()

            # Combine MSE and Dice losses with weights
            combined_loss =  segmentation_loss_mse  + 20 * se
gmentation_loss_dice

            #combined_loss.backward()
            combined_loss.backward()
            optimizer.step()

            running_loss_segmentation_combined += combined_lo
ss.item()

            if i%100==99:
                current_time = time.perf_counter()
                elapsed_time = current_time - start_time
                avg_loss_segmentation = running_loss_segmenta
```

```
tion_combined / float(100)
                print("[epoch=%d/%d, iter=%4d  elapsed_time=%
3d secs]   Combined loss: %.3f" % (epoch+1, epochs, i+1, elap
sed_time, avg_loss_segmentation))
                FILE1.write("%.3f\n" % avg_loss_segmentation)
                FILE1.flush()

                running_loss_segmentation_combined = 0.0
                running_loss_segmentation_mse = 0.0
                running_loss_segmentation_dice = 0.0

                loss_values.append(avg_loss_segmentation)


    FILE1.close()
    print("\nFinished Training\n")
    torch.save(net.state_dict(), "./saveModel2")
    return loss_values







combined_loss = run_code_for_training_for_semantic_segmentati
on_combined(net=model,data_loader=train_dataloader,
                                                        de
vice=device,epochs= 6)







plt.plot(combined_loss, label='Combined Loss')
plt.title('Combined Loss vs Iteration')
plt.xlabel('Iteration')
```

```
plt.ylabel('Loss')
plt.grid(True)
plt.show()



def run_code_for_testing_semantic_segmentation(net, test_data
loader):
    # Load the trained model
    net.load_state_dict(torch.load("/home/skose/saveModel2"))

    # Define batch size
    batch_size = 4

    # Loop over the entire dataloader
    for j, data in enumerate(test_dataloader):
        # Get images and masks from the data
        images, masks = data['image'], data['mask']

        # Perform inference using the model
        outputs = net(images)

        # Print output every 10 iterations
        if j % 10 == 0:
            print("\n\n\n\nShowing output new for test batch
%d: " % (j+1))

        # Plot images, masks, and network outputs for each sa
mple in the batch
        for i in range(batch_size):
            image_np = images[i].permute(1, 2, 0).numpy()
            mask_np = masks[i].squeeze().numpy()
            output_np = outputs[i].permute(1, 2, 0).detach().
numpy().squeeze()

            plt.figure(figsize=(15, 5))
```

```
            # Plot the original image
            plt.subplot(1, 3, 1)
            plt.imshow(image_np)
            plt.title('Original Image')
            plt.axis('off')

            # Plot the ground truth mask
            plt.subplot(1, 3, 2)
            plt.imshow(mask_np, cmap='gray')
            plt.title('Ground Truth Mask')
            plt.axis('off')

            # Plot the network output
            plt.subplot(1, 3, 3)
            # Normalize the output_np to the range [0, 1]
            output_np_normalized = (output_np - np.min(output
_np)) / (np.max(output_np) - np.min(output_np))
            # Threshold value
            threshold = 0.5
            # Apply thresholding
            output_mask_bw = np.where(output_np_normalized >=
threshold, 1.0, 0.0)

            plt.imshow(output_mask_bw, cmap='gray')
            plt.title('Model Output')
            plt.axis('off')

            plt.show()

# Call the function with your model and test dataloader
run_code_for_testing_semantic_segmentation(model, test_datalo
ader)
```

```
plt.plot(combined_loss, label='Combined Loss')
plt.plot(mse_loss, label='MSE Loss')
plt.plot(dice_loss, label='Dice Loss')
plt.title('Combined Loss vs Iteration')
plt.xlabel('Iteration')
plt.legend()
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```