In [1]:
```
import torch
import torchvision

# Get torch version
torch_version = torch.__version__

# Get device name
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device_name = torch.cuda.get_device_name(0) if torch.cuda.is_available() else "CPU"

# Get torchvision version
torchvision_version = torchvision.__version__

print(f"torch_version: {torch_version}")
print(f"Device: {device}")
print(f"Device Name: {device_name}")
print(f"Torchvision Version: {torchvision_version}")
```

```
torch_version: 2.5.1
Device: cuda
Device Name: NVIDIA A30
Torchvision Version: 0.20.1
```

1 Model updated with ASPP

In [1]:
```
# this code is mostly borrowed from DLStudio

import torch.optim as optim
import torch
import torch.nn as nn
import copy
import time
import gzip
import pickle
import numpy as np
import os
import sys
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F


class DLStudio(object):
    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError(
                   '''DLStudio constructor can only be called with keyword argument
                      the following keywords: epochs, learning_rate, batch_size, mo
                      convo_layers_config, image_size, dataroot, path_saved_model,
                      image_size, convo_layers_config, fc_layers_config, debug_trai
                      debug_test''')
        learning_rate = epochs = batch_size = convo_layers_config = momentum = None
        image_size = fc_layers_config = dataroot =  path_saved_model = classes = us
```

```
        debug_train  = debug_test = None
        if 'dataroot' in kwargs                        :   dataroot = kwargs.pop('dat
        if 'learning_rate' in kwargs                   :   learning_rate = kwargs.pop
        if 'momentum' in kwargs                        :   momentum = kwargs.pop('mom
        if 'epochs' in kwargs                          :   epochs = kwargs.pop('epoch
        if 'batch_size' in kwargs                      :   batch_size = kwargs.pop('b
        if 'convo_layers_config' in kwargs             :   convo_layers_config = kwar
        if 'image_size' in kwargs                      :   image_size = kwargs.pop('i
        if 'fc_layers_config' in kwargs                :   fc_layers_config = kwargs.
        if 'path_saved_model' in kwargs                :   path_saved_model = kwargs.
        if 'classes' in kwargs                         :   classes = kwargs.pop('clas
        if 'use_gpu' in kwargs                         :   use_gpu = kwargs.pop('use_
        if 'debug_train' in kwargs                     :   debug_train = kwargs.pop('
        if 'debug_test' in kwargs                      :   debug_test = kwargs.pop('d
        if len(kwargs) != 0: raise ValueError('''You have provided unrecognizable k
        if dataroot:
            self.dataroot = dataroot
        if convo_layers_config:
            self.convo_layers_config = convo_layers_config
        if image_size:
            self.image_size = image_size
        if fc_layers_config:
            self.fc_layers_config = fc_layers_config
            if fc_layers_config[0] != -1:
                raise Exception("""\n\n\nYour 'fc_layers_config' construction optio
                                """The first element of the list of nodes in the fc
                                """because the input to fc will be set automaticall
                                """the final activation volume of the convolutional
        if  path_saved_model:
            self.path_saved_model = path_saved_model
        if classes:
            self.class_labels = classes
        if learning_rate:
            self.learning_rate = learning_rate
        else:
            self.learning_rate = 1e-6
        if momentum:
            self.momentum = momentum
        if epochs:
            self.epochs = epochs
        if batch_size:
            self.batch_size = batch_size
        if use_gpu is not None:
            self.use_gpu = use_gpu
            if use_gpu is True:
                if torch.cuda.is_available():
                    self.device = torch.device("cuda:0")
                else:
                    self.device = torch.device("cpu")
        else:
            self.device = torch.device("cpu")
        if debug_train:
            self.debug_train = debug_train
        else:
            self.debug_train = 0
        if debug_test:
```

```
            self.debug_test = debug_test
        else:
            self.debug_test = 0
        self.debug_config = 0


    def imshow(self, img):
        '''
        called by display_tensor_as_image() for displaying the image
        '''
        img = img / 2 + 0.5     # unnormalize
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()


    def dice_loss(preds: torch.Tensor, ground_truth: torch.Tensor):
        # prevents division by zero
        epsilon=1e-6

        # Step 1: Compute Dice Coefficient.
        # For the numerator, multiply the prediction with the ground truth and comp
        # the sum of elements(in H and W dimensions).
        # shape of preds and ground_truth is (N, C, H, W), so dim 1, 2, 3 are C, H,
        # we want to sum over C, H, W dimensions so that we aggregate the loss acro
        # all object classes (global dice loss. MSE do the samething)
        numerator = torch.sum(preds * ground_truth, dim=(1, 2, 3))

        # For the denominator,  multiply prediction with
        # itself and sum the elements(in H and W dimensions) and multiply ground
        # truth by itself and sum the elements(in H and W dimensions).
        denominator = torch.sum(preds ** 2, dim=(1, 2, 3)) + torch.sum(ground_truth

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

        # Step 3: Compute dice_loss = 1 - dice_coefficient
        dice_loss = 1 - dice_coefficient.mean()

        return dice_loss

    def display_tensor_as_image(self, tensor, title=""):
        '''
        This method converts the argument tensor into a photo image that you can di
        in your terminal screen. It can convert tensors of three different shapes
        into images: (3,H,W), (1,H,W), and (H,W), where H, for height, stands for t
        number of pixels in the vertical direction and W, for width, for the same
        along the horizontal direction.  When the first element of the shape is 3,
        that means that the tensor represents a color image in which each pixel in
        the (H,W) plane has three values for the three color channels.  On the othe
        hand, when the first element is 1, that stands for a tensor that will be
        shown as a grayscale image.  And when the shape is just (H,W), that is
        automatically taken to be for a grayscale image.
        '''
        tensor_range = (torch.min(tensor).item(), torch.max(tensor).item())
        if tensor_range == (-1.0,1.0):
```

```
            ##  The tensors must be between 0.0 and 1.0 for the display:
            print("\n\n\nimage un-normalization called")
            tensor = tensor/2.0 + 0.5     # unnormalize
        plt.figure(title)
        ###  The call to plt.imshow() shown below needs a numpy array. We must also
        ###  transpose the array so that the number of channels (the same thing as
        ###  number of color planes) is in the last element.  For a tensor, it woul
        ###  the first element.
        if tensor.shape[0] == 3 and len(tensor.shape) == 3:
#           plt.imshow( tensor.numpy().transpose(1,2,0) )
            plt.imshow( tensor.numpy().transpose(1,2,0) )
        ###  If the grayscale image was produced by calling torchvision.transform's
        ###  ".ToPILImage()", and the result converted to a tensor, the tensor shap
        ###  again have three elements in it, however the first element that stands
        ###  the number of channels will now be 1
        elif tensor.shape[0] == 1 and len(tensor.shape) == 3:
            tensor = tensor[0,:,:]
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        ###  For any one color channel extracted from the tensor representation of
        ###  image, the shape of the tensor will be (W,H):
        elif len(tensor.shape) == 2:
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        else:
            sys.exit("\n\n\nfrom 'display_tensor_as_image()': tensor for image is i
        plt.show()



    class SemanticSegmentation(nn.Module):
        """
        The purpose of this inner class is to be able to use the DLStudio platform
        experiments with semantic segmentation.  At its simplest level, the purpose
        semantic segmentation is to assign correct labels to the different objects
        scene, while localizing them at the same time.  At a more sophisticated lev
        system that carries out semantic segmentation should also output a symbolic
        expression based on the objects found in the image and their spatial relati
        with one another.

        The workhorse of this inner class is the mUNet network that is based on the
        network that was first proposed by Ronneberger, Fischer and Brox in the pap
        "U-Net: Convolutional Networks for Biomedical Image Segmentation".  Their U
        extracts binary masks for the cell pixel blobs of interest in biomedical im
        The output of their Unet can therefore be treated as a pixel-wise binary cl
        at each pixel position.  The mUnet class, on the other hand, is intended fo
        segmenting out multiple objects simultaneously form an image. [A weaker rea
        "Multi" in the name of the class is that it uses skip connections not only
        the two arms of the "U", but also also along the arms.  The skip connection
        original Unet are only between the two arms of the U.  In mUnet, each objec
        assigned a separate channel in the output of the network.

        This version of DLStudio also comes with a new dataset, PurdueShapes5MultiO
        for experimenting with mUnet.  Each image in this dataset contains a random
        of selections from five different shapes, with the shapes being randomly sc
        oriented, and located in each image.  The five different shapes are: rectan
        triangle, disk, oval, and star.
```

```
        Class Path:   DLStudio  ->  SemanticSegmentation
    """
    def __init__(self, dl_studio, max_num_objects, dataserver_train=None, datas
        super(DLStudio.SemanticSegmentation, self).__init__()
        self.dl_studio = dl_studio
        self.max_num_objects = max_num_objects
        self.dataserver_train = dataserver_train
        self.dataserver_test = dataserver_test


    class PurdueShapes5MultiObjectDataset(torch.utils.data.Dataset):
        """
        The very first thing to note is that the images in the dataset
        PurdueShapes5MultiObjectDataset are of size 64x64.  Each image has a ra
        number (up to five) of the objects drawn from the following five shapes
        rectangle, triangle, disk, oval, and star.  Each shape is randomized wi
        respect to all its parameters, including those for its scale and locati
        image.

        Each image in the dataset is represented by two data objects, one a lis
        other a dictionary. The list data objects consists of the following ite

            [R, G, B, mask_array, mask_val_to_bbox_map]

        and the other data object is a dictionary that is set to:

            label_map = {'rectangle':50,
                         'triangle' :100,
                         'disk'     :150,
                         'oval'     :200,
                         'star'     :250}

        Note that that second data object for each image is the same, as shown

        In the rest of this comment block, I'll explain in greater detail the e
        of the list in line (A) above.


        R,G,B:
        ------

        Each of these is a 4096-element array whose elements store the correspo
        color values at each of the 4096 pixels in a 64x64 image.  That is, R i
        of 4096 integers, each between 0 and 255, for the value of the red comp
        the color at each pixel. Similarly, for G and B.


        mask_array:
        ----------

        The fourth item in the list shown in line (A) above is for the mask whi
        numpy array of shape:

                        (5, 64, 64)
```

```
It is initialized by the command:

     mask_array = np.zeros((5,64,64), dtype=np.uint8)

In essence, the mask_array consists of five planes, each of size 64x64.
plane of the mask array represents an object type according to the foll
shape_index

        shape_index = (label_map[shape] - 50) // 50

where the label_map is as shown in line (B) above.  In other words, the
shape_index values for the different shapes are:

         rectangle:  0
          triangle:  1
              disk:  2
              oval:  3
              star:  4

Therefore, the first layer (of index 0) of the mask is where the pixel
50 are stored at all those pixels that belong to the rectangle shapes.
Similarly, the second mask layer (of index 1) is where the pixel values
are stored at all those pixel coordinates that belong to the triangle s
an image; and so on.

It is in the manner described above that we define five different masks
image in the dataset.  Each mask is for a different shape and the pixel
at the nonzero pixels in each mask layer are keyed to the shapes also.

A reader is likely to wonder as to the need for this redundancy in the
representation of the shapes in each image.  Such a reader is likely to
can't we just use the binary values 1s and 0s in each mask layer where
corresponding pixels are in the image?  Setting these mask values to 50
etc., was done merely for convenience.  I went with the intuition that
learning needed for multi-object segmentation would become easier if ea
was represented by a different pixels value in the corresponding mask.
ahead incorporated that in the dataset generation program itself.

The mask values for the shapes are not to be confused with the actual R
of the pixels that belong to the shapes. The RGB values at the pixels i
are randomly generated.  Yes, all the pixels in a shape instance in an
have the same RGB values (but that value has nothing to do with the val
to the mask pixels for that shape).


mask_val_to_bbox_map:
--------------------

The fifth item in the list in line (A) above is a dictionary that tells
bounding-box rectangle to associate with each shape in the image.  To i
what this dictionary looks like, assume that an image contains only one
rectangle and only one disk, the dictionary in this case will look like

    mask values to bbox mappings:  {200: [],
                                    250: [],
                                    100: [],
```

```
                                          50: [[56, 20, 63, 25]],
                                         150: [[37, 41, 55, 59]]}

        Should there happen to be two rectangles in the same image, the diction
        then be like:

            mask values to bbox mappings:  {200: [],
                                            250: [],
                                            100: [],
                                             50: [[56, 20, 63, 25], [18, 16, 32
                                            150: [[37, 41, 55, 59]]}

        Therefore, it is not a problem even if all the objects in an image are
        same type.  Remember, the object that are selected for an image are sho
        randomly from the different shapes.  By the way, an entry like '[56, 20
        25]' for the bounding box means that the upper-left corner of the BBox
        'rectangle' shape is at (56,20) and the lower-right corner of the same
        pixel coordinates (63,25).

        As far as the BBox quadruples are concerned, in the definition

                [min_x,min_y,max_x,max_y]

        note that x is the horizontal coordinate, increasing to the right on yo
        screen, and y is the vertical coordinate increasing downwards.

        Class Path:   DLStudio  ->  SemanticSegmentation  ->  PurdueShapes5Mult

        """
        def __init__(self, dl_studio, segmenter, train_or_test, dataset_file):
            super(DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset
            max_num_objects = segmenter.max_num_objects
            if train_or_test == 'train' and dataset_file == "PurdueShapes5Multi
                if os.path.exists("torch_saved_PurdueShapes5MultiObject-10000_d
                          os.path.exists("torch_saved_PurdueShapes5MultiObject_
                    print("\nLoading training data from torch saved file")
                    self.dataset = torch.load("torch_saved_PurdueShapes5MultiOb
                    self.label_map = torch.load("torch_saved_PurdueShapes5Multi
                    self.num_shapes = len(self.label_map)
                    self.image_size = dl_studio.image_size
                else:
                    print("""\n\n\nLooks like this is the first time you will b
                          """the dataset for this script. First time loading co
                          """a few minutes.  Any subsequent attempts will only
                          """a few seconds.\n\n\n""")
                    root_dir = dl_studio.dataroot
                    f = gzip.open(root_dir + dataset_file, 'rb')
                    dataset = f.read()
                    self.dataset, self.label_map = pickle.loads(dataset, encodi
                    torch.save(self.dataset, "torch_saved_PurdueShapes5MultiObj
                    torch.save(self.label_map, "torch_saved_PurdueShapes5MultiO
                    # reverse the key-value pairs in the label dictionary:
                    self.class_labels = dict(map(reversed, self.label_map.items
                    self.num_shapes = len(self.class_labels)
                    self.image_size = dl_studio.image_size
            else:
```

```
                root_dir = dl_studio.dataroot
                f = gzip.open(root_dir + dataset_file, 'rb')
                dataset = f.read()
                if sys.version_info[0] == 3:
                    self.dataset, self.label_map = pickle.loads(dataset, encodi
                else:
                    self.dataset, self.label_map = pickle.loads(dataset)
                # reverse the key-value pairs in the label dictionary:
                self.class_labels = dict(map(reversed, self.label_map.items()))
                self.num_shapes = len(self.class_labels)
                self.image_size = dl_studio.image_size

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

            def __getitem__(self, idx):
                image_size = self.image_size
                r = np.array( self.dataset[idx][0] )
                g = np.array( self.dataset[idx][1] )
                b = np.array( self.dataset[idx][2] )
                R,G,B = r.reshape(image_size[0],image_size[1]), g.reshape(image_siz
                im_tensor = torch.zeros(3,image_size[0],image_size[1], dtype=torch.
                im_tensor[0,:,:] = torch.from_numpy(R)
                im_tensor[1,:,:] = torch.from_numpy(G)
                im_tensor[2,:,:] = torch.from_numpy(B)
                mask_array = np.array(self.dataset[idx][3])
                max_num_objects = len( mask_array[0] )
                mask_tensor = torch.from_numpy(mask_array)
                mask_val_to_bbox_map =  self.dataset[idx][4]
                max_bboxes_per_entry_in_map = max([ len(mask_val_to_bbox_map[key])
                ##  The first arg 5 is for the number of bboxes we are going to nee
                ##  shapes are exactly the same, you are going to need five differe
                ##  The second arg is the index reserved for each shape in a single
                bbox_tensor = torch.zeros(max_num_objects,self.num_shapes,4, dtype=
                for bbox_idx in range(max_bboxes_per_entry_in_map):
                    for key in mask_val_to_bbox_map:
                        if len(mask_val_to_bbox_map[key]) == 1:
                            if bbox_idx == 0:
                                bbox_tensor[bbox_idx,key,:] = torch.from_numpy(np.a
                        elif len(mask_val_to_bbox_map[key]) > 1 and bbox_idx < len(
                            bbox_tensor[bbox_idx,key,:] = torch.from_numpy(np.array
                sample = {'image'        : im_tensor,
                          'mask_tensor'  : mask_tensor,
                          'bbox_tensor'  : bbox_tensor }
                return sample

        def load_PurdueShapes5MultiObject_dataset(self, dataserver_train, dataserve
            self.train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                        batch_size=self.dl_studio.batch_size,shuffle=True, num_work
            self.test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                batch_size=self.dl_studio.batch_size,shuffle=False,


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

```
            Class Path:   DLStudio  ->  SemanticSegmentation  ->  SkipBlockDN
            """
            def __init__(self, in_ch, out_ch, downsample=False, skip_connections=Tr
                super(DLStudio.SemanticSegmentation.SkipBlockDN, self).__init__()
                self.downsample = downsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch

                # UP is using ConvTranspose2d, DN is using Conv2d
                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:
                    # UP is using ConvTranspose2d, DN is using Conv2d
                    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, identity), 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, skip_connections=True
                super(DLStudio.SemanticSegmentation.SkipBlockUP, self).__init__()
                self.upsample = upsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch

                # DN is using Conv2d, UP is using ConvTranspose2d
                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:
                    # DN is using Conv2d, UP is using ConvTranspose2d
                    self.upsampler = nn.ConvTranspose2d(in_ch, out_ch, 1, stride=2,
            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 ASPP(nn.Module):
            """
            This class is for the Atrous Spatial Pyramid Pooling (ASPP) block. The
            block is used to capture the context information at multiple scales.  T
            block uses atrous convolutions with different rates to capture context
            information at different scales.  The ASPP block is placed on top of th
            feature extractor network.  The ASPP block uses the original convolutio
            feature map, the other branches use convolutional feature maps that are
            atrous convolutions with different rates to the original feature map.
            the three branches are then concatenated and passed through a 1x1 convo
            layer to obtain the final output of the ASPP block.
            """
            def __init__(self, in_ch, out_ch):
                super(DLStudio.SemanticSegmentation.ASPP, self).__init__()
                self.conv1 = nn.Conv2d(in_ch, out_ch, 1)

                # padding needs to be eqaul to dilation when kernel size is 3. So t
                # same as input size.
                # (formula: out_size = (in_size + 2*padding - dilation*(kernel_size
                self.conv2 = nn.Conv2d(in_ch, out_ch, 3, padding=6, dilation=6)
                self.conv3 = nn.Conv2d(in_ch, out_ch, 3, padding=12, dilation=12)
                self.conv4 = nn.Conv2d(in_ch, out_ch, 3, padding=18, dilation=18)
                self.conv5 = nn.Conv2d(in_ch, out_ch, 3, padding=24, dilation=24)

                # final conv for concatenation
                self.conv6 = nn.Conv2d(out_ch*5, out_ch, 1)
            def forward(self, x):
                out1 = self.conv1(x)
                out2 = self.conv2(x)
                out3 = self.conv3(x)
                out4 = self.conv4(x)
```

```
            out5 = self.conv5(x)
            out = torch.cat([out1, out2, out3, out4, out5], dim=1)
            out = self.conv6(out)
            return out


    class mUNet(nn.Module):
        """
        This network is called mUNet because it is intended for segmenting out
        multiple objects simultaneously form an image. [A weaker reason for "Mu
        the name of the class is that it uses skip connections not only across
        arms of the "U", but also also along the arms.]  The classic UNET was f
        proposed by Ronneberger, Fischer and Brox in the paper "U-Net: Convolut
        Networks for Biomedical Image Segmentation".  Their UNET extracts binar
        for the cell pixel blobs of interest in biomedical images.  The output
        UNET therefore can therefore be treated as a pixel-wise binary classifi
        each pixel position.

        The mUNet presented here, on the other hand, is meant specifically for
        simultaneously identifying and localizing multiple objects in a given i
        Each object type is assigned a separate channel in the output of the ne

        I have created a dataset, PurdueShapes5MultiObject, for experimenting w
        mUNet.  Each image in this dataset contains a random number of selectio
        five different shapes, with the shapes being randomly scaled, oriented,
        located in each image.  The five different shapes are: rectangle, trian
        disk, oval, and star.

        Class Path:   DLStudio  ->  SemanticSegmentation  ->  mUNet

        """
        def __init__(self, skip_connections=True, depth=16):
            super(DLStudio.SemanticSegmentation.mUNet, self).__init__()
            self.depth = depth // 2
            self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
            ##  For the DN(down) 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.SemanticSegmentation.SkipBloc
            self.skip64dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 64,
            self.skip64to128DN = DLStudio.SemanticSegmentation.SkipBlockDN(64,
            self.skip128DN_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128DN_arr.append(DLStudio.SemanticSegmentation.SkipBlo
            self.skip128dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(128,12



            # add ASPP block before going up
            self.aspp = DLStudio.SemanticSegmentation.ASPP(128, 128)



            ##  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.SemanticSegmentation.SkipBloc
            self.skip64usUP = DLStudio.SemanticSegmentation.SkipBlockUP(64, 64,
            self.skip128to64UP = DLStudio.SemanticSegmentation.SkipBlockUP(128,
            self.skip128UP_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128UP_arr.append(DLStudio.SemanticSegmentation.SkipBlo
            self.skip128usUP = DLStudio.SemanticSegmentation.SkipBlockUP(128,12
            self.conv_out = nn.ConvTranspose2d(64, 5, 3, stride=2,dilation=2,ou

        def forward(self, x):
            ##  Going down to the bottom of the U:
            x = nn.MaxPool2d(2,2)(nn.functional.relu(self.conv_in(x)))
            for i,skip64 in enumerate(self.skip64DN_arr[:self.depth//4]):
                x = skip64(x)

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

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



            # add ASPP block beforing going up
            x = self.aspp(x)



            ## Coming up from the bottom of U on the other side:
            x = self.skip128usUP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
                x = skip128(x)
            x[:,:num_channels_to_save3,:,:] =  save_for_upside_3
            x = self.bn1UP(x)
            for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
                x = skip128(x)
            x = self.skip128to64UP(x)
            for i,skip64 in enumerate(self.skip64UP_arr[self.depth//4:]):
                x = skip64(x)
```

```
            x[:,:num_channels_to_save2,:,:] =  save_for_upside_2
            x = self.bn2UP(x)
            x = self.skip64usUP(x)
            for i,skip64 in enumerate(self.skip64UP_arr[:self.depth//4]):
                x = skip64(x)
            x[:,:num_channels_to_save1,:,:] =  save_for_upside_1
            x = self.conv_out(x)
            return x


    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(DLStudio.SemanticSegmentation.SegmentationLoss, self).__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_idx,:,:]
                    element_wise = (outputh - mask)**2
                    mask_based_loss[0,mask_layer_idx] = torch.mean(element_wise
                composite_loss[0,idx] = torch.sum(mask_based_loss)
            return torch.sum(composite_loss) / self.batch_size


    def MSE_run_code_for_training_for_semantic_segmentation(self, net):
        filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

        # using MSE loss
        criterion1 = nn.MSELoss()

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

        # record the loss values
        total_loss = []

        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['image'],data['mask_ten
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
```

```
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)
                segmentation_loss = criterion1(output, mask_tensor)
                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_loss.item()
                if i%500==499:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segmentation / float(5
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   los
                    total_loss.append(avg_loss_segmentation)
                    FILE1.write("%.3f\n" % avg_loss_segmentation)
                    FILE1.flush()
                    running_loss_segmentation = 0.0
        print("\nFinished Training\n")
        self.save_model(net)

        return total_loss


    def DICE_run_code_for_training_for_semantic_segmentation(self, net):
        filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

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

        # record the loss values
        total_loss = []

        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['image'],data['mask_ten
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)

                # using DICE loss
                segmentation_loss = DLStudio.dice_loss(output, mask_tensor)

                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_loss.item()
                if i%500==499:
                    current_time = time.perf_counter()
```

```
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segmentation / float(5
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   MSE
                    total_loss.append(avg_loss_segmentation)
                    FILE1.write("%.3f\n" % avg_loss_segmentation)
                    FILE1.flush()
                    running_loss_segmentation = 0.0
        print("\nFinished Training\n")
        self.save_model(net)
        return total_loss


    def DICE_MSE_run_code_for_training_for_semantic_segmentation(self, net, Dic
        filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

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

        # record the loss values
        total_loss = []

        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['image'],data['mask_ten
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)



                # using DICE+MSE loss
                dice_scale = DiceScale
                segmentation_loss = dice_scale * DLStudio.dice_loss(output, mas
                segmentation_loss += nn.MSELoss()(output, mask_tensor)



                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_loss.item()
                if i%500==499:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segmentation / float(5
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   MSE
                    total_loss.append(avg_loss_segmentation)
                    FILE1.write("%.3f\n" % avg_loss_segmentation)
```

```
                    FILE1.flush()
                    running_loss_segmentation = 0.0
        print("\nFinished Training\n")
        self.save_model(net)
        return total_loss


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


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

                    for batch_im_idx in range(batch_size):
                        for mask_layer_idx in range(max_num_objects):
                            for i in range(image_size[0]):
                                for j in range(image_size[1]):
```

```
                                    if mask_layer_idx == 0:
                                        if 25 < outputs[batch_im_idx,mask_layer
                                            outputs[batch_im_idx,mask_layer_idx
                                        else:
                                            outputs[batch_im_idx,mask_layer_idx
                                    elif mask_layer_idx == 1:
                                        if 65 < outputs[batch_im_idx,mask_layer
                                            outputs[batch_im_idx,mask_layer_idx
                                        else:
                                            outputs[batch_im_idx,mask_layer_idx
                                    elif mask_layer_idx == 2:
                                        if 115 < outputs[batch_im_idx,mask_laye
                                            outputs[batch_im_idx,mask_layer_idx
                                        else:
                                            outputs[batch_im_idx,mask_layer_idx
                                    elif mask_layer_idx == 3:
                                        if 165 < outputs[batch_im_idx,mask_laye
                                            outputs[batch_im_idx,mask_layer_idx
                                        else:
                                            outputs[batch_im_idx,mask_layer_idx
                                    elif mask_layer_idx == 4:
                                        if outputs[batch_im_idx,mask_layer_idx,
                                            outputs[batch_im_idx,mask_layer_idx
                                        else:
                                            outputs[batch_im_idx,mask_layer_idx

                                display_tensor[2*batch_size+batch_size*mask_layer_i
                        self.dl_studio.display_tensor_as_image(
                           torchvision.utils.make_grid(display_tensor, nrow=batch_s



    def compute_iou(self, pred_mask, true_mask):
        intersection = (pred_mask & true_mask).sum()
        union = (pred_mask | true_mask).sum()
        return intersection / union if union > 0 else 0  # Avoid division by ze

    def run_code_for_compute_iou(self, net):
        net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
        net.eval()  # Set model to evaluation mode
        total_iou = 0
        total_accuracy = 0
        num_samples = 0

        with torch.no_grad():
            for i, data in enumerate(self.test_dataloader):
                im_tensor, mask_tensor = data['image'], data['mask_tensor']
                outputs = net(im_tensor)

                # Convert outputs to binary masks. (B,C,H,W) -> (B,H,W).
                # Each pixel is assigned the class index with the highest predi
                pred_masks = outputs.argmax(dim=1)
                true_masks = mask_tensor.argmax(dim=1)

                # Compute IoU for each image in batch
```

```
                for j in range(pred_masks.shape[0]):
                    iou = self.compute_iou(pred_masks[j] > 0.5, true_masks[j] >
                    total_iou += iou

                    # get same pixel for accuracy
                    correct_pixels = (pred_masks[j] == true_masks[j]).sum()
                    total_pixels = true_masks[j].numel()
                    total_accuracy += correct_pixels / total_pixels

                    num_samples += 1

        mean_iou = total_iou / num_samples
        mean_accuracy = total_accuracy / num_samples
        print(f"Mean IoU: {mean_iou:.4f}, Mean Pixel Accuracy: {mean_accuracy:.
```

Understanding of mUnet

mUNet is a modified version of the U-Net architecture designed for multi-object semantic segmentation. It follows the typical encoder-decoder structure, where the encoder extracts hierarchical features, and the decoder reconstructs segmented objects. Skip connections are used throughout to preserve spatial details and improve learning efficiency. A key enhancement in mUNet is the Atrous Spatial Pyramid Pooling (ASPP) block, which captures multi-scale context by applying dilated convolutions at different rates. This helps in segmenting objects of various sizes effectively.

When using Dice+MSE loss, do you think there should be a scaling factor to scale the Dice Loss? Why or Why not?

I think there should be a scaling factor to scale the Dice Loss because Dice loss is typically bounded—usually falling between 0 and 1 - whereas MSE loss can vary widely and reach much larger values. Without scaling, the MSE component might dominate the combined loss, which could cause the optimization to prioritize reducing MSE error over improving the overlap measure that Dice loss provides.

2 Training best curves

Result first, here is the table for summary:

	Mean IoU	Mean Pixel Accuracy
MSE, LR = 1e-5	0.1133	0.1751
MSE, LR = 1e-4	0.1058	0.0940
DICE, LR = 1e-5	0.1034	0.0435
DICE, LR = 1e-4	0.0737	0.6970
DICE + MSE, LR = 1e-5	0.1074	0.1035
DICE + MSE, LR = 1e-4	0.1059	0.0907

From the table, it's evident that using MSE loss with a learning rate of 1e-5 yields the best performance in terms of Mean IoU (0.1133), while DICE loss with the learning rate 1e-4 provides the best performance in terms of Mean Pixel Accuracy (0.6970). Interestingly, increasing the learning rate to 1e-4 generally leads to a decline in performance across both metrics for the most loss types. The combination of DICE and MSE does not significantly outperform either loss individually, suggesting limited synergy between them in this setup. Overall, lower learning rates tend to produce more stable and effective results.

Although both Mean IoU and Mean Pixel Accuracy appear relatively low, this is largely due to dithering effects. The model is able to outline the object shape reasonably well, but the interior regions often display dithering, leading to fragmented predictions that negatively impact these metrics.

2.1 MSE loss curve

In [ ]:
```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_MSE",
                    momentum = 0.9,
                    learning_rate = 1e-5,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

model = segmenter.mUNet(skip_connections=True, depth=16)
```

```
#model = segmenter.mUNet(skip_connections=False, depth=4)

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

MSELoss = segmenter.MSE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(MSELoss, label='MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

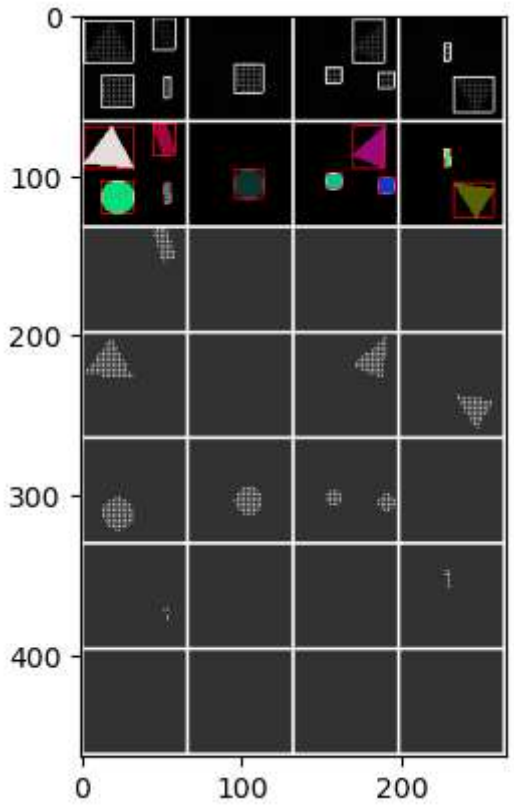
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=150 secs]   loss: 460.839
[epoch=1/10, iter=1000  elapsed_time=300 secs]   loss: 442.239
[epoch=1/10, iter=1500  elapsed_time=451 secs]   loss: 438.298
[epoch=1/10, iter=2000  elapsed_time=601 secs]   loss: 435.559
[epoch=1/10, iter=2500  elapsed_time=752 secs]   loss: 428.840

[epoch=2/10, iter= 500  elapsed_time=902 secs]   loss: 421.538
[epoch=2/10, iter=1000  elapsed_time=1053 secs]   loss: 411.002
[epoch=2/10, iter=1500  elapsed_time=1203 secs]   loss: 413.596
[epoch=2/10, iter=2000  elapsed_time=1354 secs]   loss: 409.424
[epoch=2/10, iter=2500  elapsed_time=1506 secs]   loss: 404.747

[epoch=3/10, iter= 500  elapsed_time=1657 secs]   loss: 396.363
[epoch=3/10, iter=1000  elapsed_time=1807 secs]   loss: 398.667
[epoch=3/10, iter=1500  elapsed_time=1958 secs]   loss: 404.793
[epoch=3/10, iter=2000  elapsed_time=2108 secs]   loss: 395.247
[epoch=3/10, iter=2500  elapsed_time=2259 secs]   loss: 395.237

[epoch=4/10, iter= 500  elapsed_time=2409 secs]   loss: 391.959
[epoch=4/10, iter=1000  elapsed_time=2560 secs]   loss: 399.444
[epoch=4/10, iter=1500  elapsed_time=2711 secs]   loss: 383.840
[epoch=4/10, iter=2000  elapsed_time=2861 secs]   loss: 390.976
[epoch=4/10, iter=2500  elapsed_time=3012 secs]   loss: 381.608

[epoch=5/10, iter= 500  elapsed_time=3163 secs]   loss: 389.602
[epoch=5/10, iter=1000  elapsed_time=3314 secs]   loss: 388.443
[epoch=5/10, iter=1500  elapsed_time=3464 secs]   loss: 380.071
[epoch=5/10, iter=2000  elapsed_time=3615 secs]   loss: 380.639
[epoch=5/10, iter=2500  elapsed_time=3765 secs]   loss: 381.612

[epoch=6/10, iter= 500  elapsed_time=3916 secs]   loss: 380.999
[epoch=6/10, iter=1000  elapsed_time=4066 secs]   loss: 386.553
[epoch=6/10, iter=1500  elapsed_time=4216 secs]   loss: 380.787
[epoch=6/10, iter=2000  elapsed_time=4367 secs]   loss: 385.149
[epoch=6/10, iter=2500  elapsed_time=4517 secs]   loss: 368.517

[epoch=7/10, iter= 500  elapsed_time=4668 secs]   loss: 374.060
[epoch=7/10, iter=1000  elapsed_time=4818 secs]   loss: 380.314
[epoch=7/10, iter=1500  elapsed_time=4969 secs]   loss: 374.254
[epoch=7/10, iter=2000  elapsed_time=5119 secs]   loss: 380.145
[epoch=7/10, iter=2500  elapsed_time=5270 secs]   loss: 379.026

[epoch=8/10, iter= 500  elapsed_time=5420 secs]   loss: 371.445
[epoch=8/10, iter=1000  elapsed_time=5571 secs]   loss: 375.062
[epoch=8/10, iter=1500  elapsed_time=5721 secs]   loss: 376.933
[epoch=8/10, iter=2000  elapsed_time=5872 secs]   loss: 378.848
[epoch=8/10, iter=2500  elapsed_time=6023 secs]   loss: 373.121

[epoch=9/10, iter= 500  elapsed_time=6173 secs]   loss: 376.371
[epoch=9/10, iter=1000  elapsed_time=6324 secs]   loss: 376.414
[epoch=9/10, iter=1500  elapsed_time=6474 secs]   loss: 371.094
[epoch=9/10, iter=2000  elapsed_time=6624 secs]   loss: 376.293
[epoch=9/10, iter=2500  elapsed_time=6775 secs]   loss: 363.549
```

```
[epoch=10/10, iter= 500  elapsed_time=6925 secs]   loss: 366.114
[epoch=10/10, iter=1000  elapsed_time=7076 secs]   loss: 380.403
[epoch=10/10, iter=1500  elapsed_time=7227 secs]   loss: 366.735
[epoch=10/10, iter=2000  elapsed_time=7377 secs]   loss: 372.156
[epoch=10/10, iter=2500  elapsed_time=7528 secs]   loss: 369.467

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

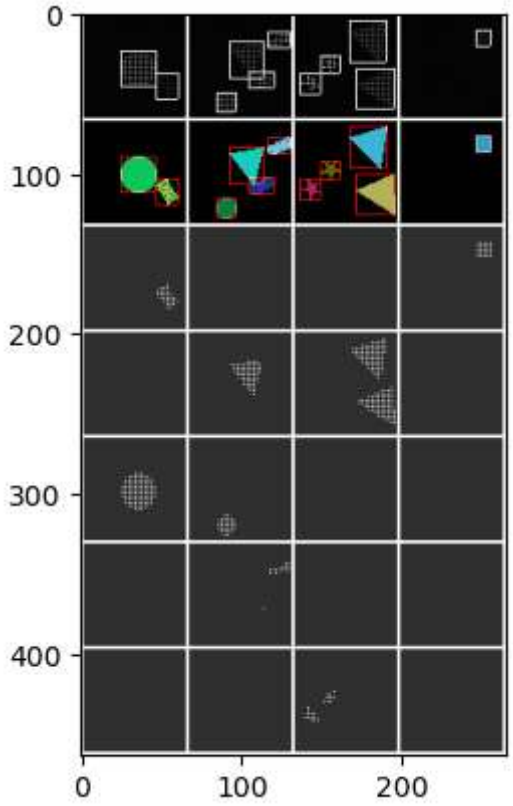
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



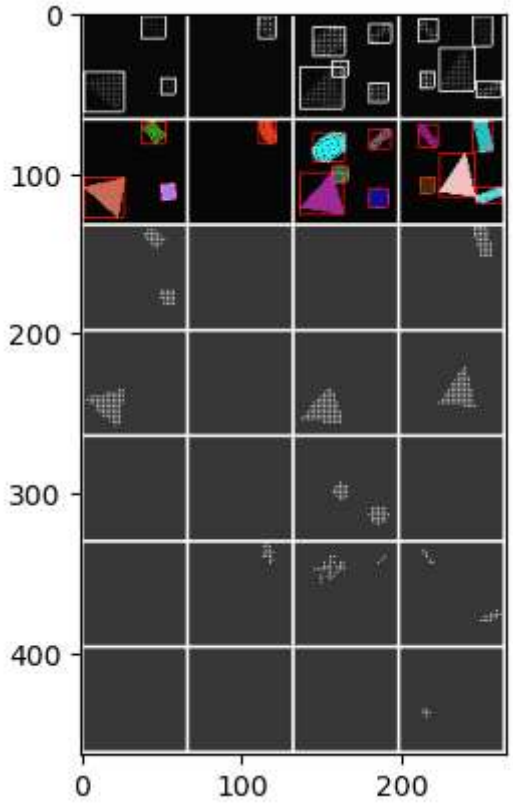

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



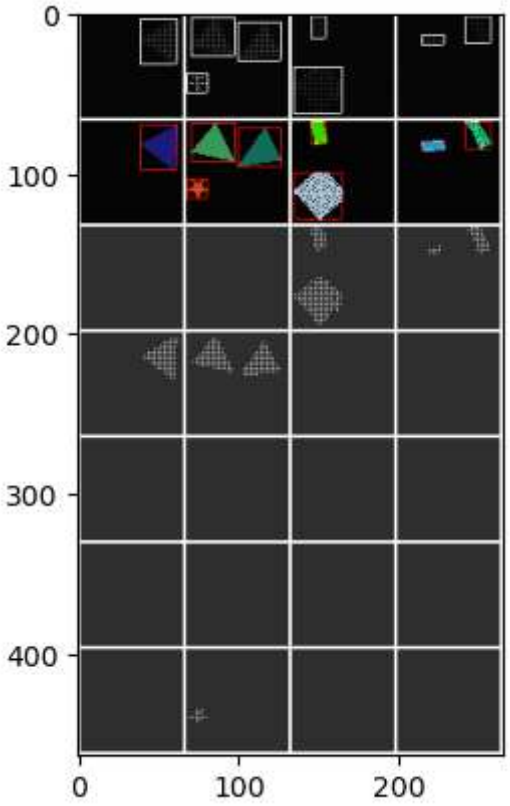
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



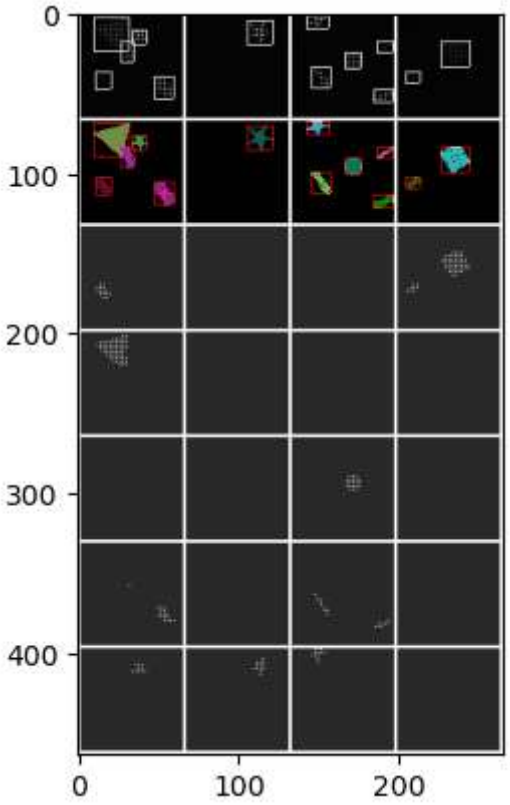
Showing output for test batch 151:

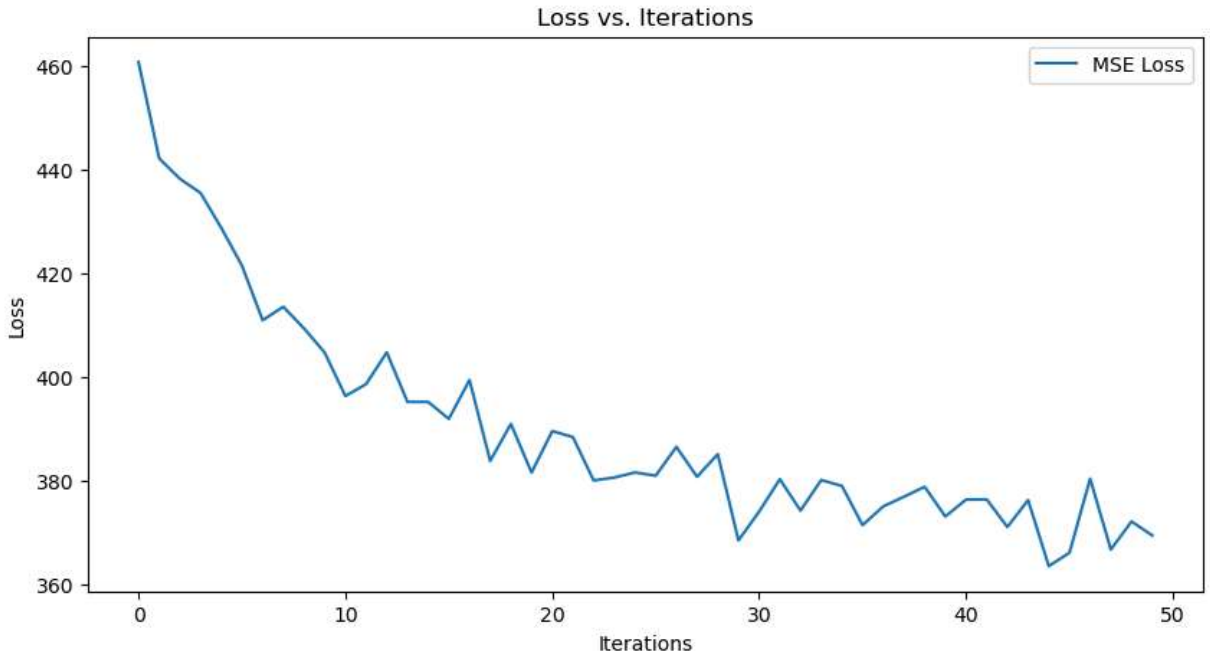
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



Showing output for test batch 201:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1133, Mean Pixel Accuracy: 0.1751
```

2.2 Dice loss curve

In [21]:

```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE",
                    momentum = 0.9,
                    learning_rate = 1e-5,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )
```

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

DICELoss = segmenter.DICE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(DICELoss, label='DICE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

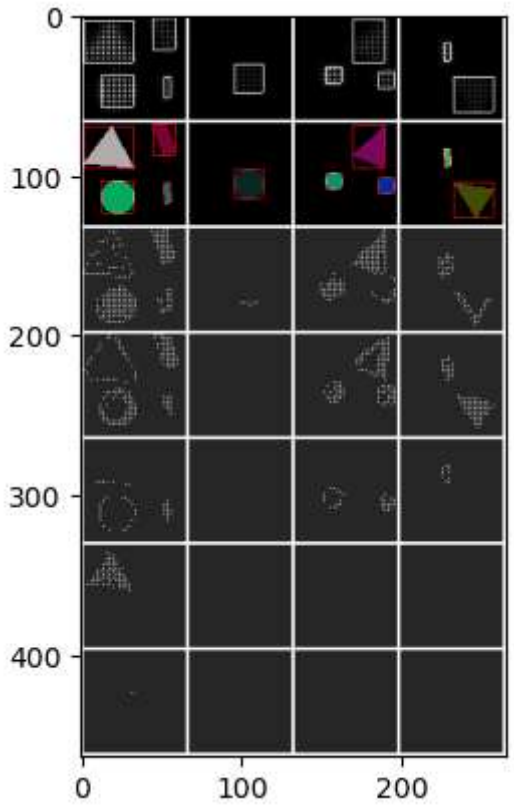
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=151 secs]   MSE loss: 0.925
[epoch=1/10, iter=1000  elapsed_time=301 secs]   MSE loss: 0.907
[epoch=1/10, iter=1500  elapsed_time=452 secs]   MSE loss: 0.884
[epoch=1/10, iter=2000  elapsed_time=602 secs]   MSE loss: 0.874
[epoch=1/10, iter=2500  elapsed_time=753 secs]   MSE loss: 0.864

[epoch=2/10, iter= 500  elapsed_time=903 secs]   MSE loss: 0.857
[epoch=2/10, iter=1000  elapsed_time=1054 secs]   MSE loss: 0.850
[epoch=2/10, iter=1500  elapsed_time=1204 secs]   MSE loss: 0.844
[epoch=2/10, iter=2000  elapsed_time=1355 secs]   MSE loss: 0.838
[epoch=2/10, iter=2500  elapsed_time=1506 secs]   MSE loss: 0.837

[epoch=3/10, iter= 500  elapsed_time=1657 secs]   MSE loss: 0.837
[epoch=3/10, iter=1000  elapsed_time=1807 secs]   MSE loss: 0.830
[epoch=3/10, iter=1500  elapsed_time=1958 secs]   MSE loss: 0.827
[epoch=3/10, iter=2000  elapsed_time=2108 secs]   MSE loss: 0.825
[epoch=3/10, iter=2500  elapsed_time=2259 secs]   MSE loss: 0.823

[epoch=4/10, iter= 500  elapsed_time=2409 secs]   MSE loss: 0.823
[epoch=4/10, iter=1000  elapsed_time=2560 secs]   MSE loss: 0.820
[epoch=4/10, iter=1500  elapsed_time=2710 secs]   MSE loss: 0.821
[epoch=4/10, iter=2000  elapsed_time=2861 secs]   MSE loss: 0.819
[epoch=4/10, iter=2500  elapsed_time=3011 secs]   MSE loss: 0.816

[epoch=5/10, iter= 500  elapsed_time=3161 secs]   MSE loss: 0.815
[epoch=5/10, iter=1000  elapsed_time=3312 secs]   MSE loss: 0.816
[epoch=5/10, iter=1500  elapsed_time=3463 secs]   MSE loss: 0.816
[epoch=5/10, iter=2000  elapsed_time=3613 secs]   MSE loss: 0.812
[epoch=5/10, iter=2500  elapsed_time=3764 secs]   MSE loss: 0.812

[epoch=6/10, iter= 500  elapsed_time=3914 secs]   MSE loss: 0.811
[epoch=6/10, iter=1000  elapsed_time=4065 secs]   MSE loss: 0.810
[epoch=6/10, iter=1500  elapsed_time=4215 secs]   MSE loss: 0.811
[epoch=6/10, iter=2000  elapsed_time=4366 secs]   MSE loss: 0.812
[epoch=6/10, iter=2500  elapsed_time=4517 secs]   MSE loss: 0.809

[epoch=7/10, iter= 500  elapsed_time=4667 secs]   MSE loss: 0.808
[epoch=7/10, iter=1000  elapsed_time=4818 secs]   MSE loss: 0.807
[epoch=7/10, iter=1500  elapsed_time=4968 secs]   MSE loss: 0.805
[epoch=7/10, iter=2000  elapsed_time=5119 secs]   MSE loss: 0.807
[epoch=7/10, iter=2500  elapsed_time=5269 secs]   MSE loss: 0.807

[epoch=8/10, iter= 500  elapsed_time=5419 secs]   MSE loss: 0.804
[epoch=8/10, iter=1000  elapsed_time=5570 secs]   MSE loss: 0.806
[epoch=8/10, iter=1500  elapsed_time=5721 secs]   MSE loss: 0.805
[epoch=8/10, iter=2000  elapsed_time=5872 secs]   MSE loss: 0.804
[epoch=8/10, iter=2500  elapsed_time=6022 secs]   MSE loss: 0.801

[epoch=9/10, iter= 500  elapsed_time=6173 secs]   MSE loss: 0.799
[epoch=9/10, iter=1000  elapsed_time=6324 secs]   MSE loss: 0.801
[epoch=9/10, iter=1500  elapsed_time=6475 secs]   MSE loss: 0.802
[epoch=9/10, iter=2000  elapsed_time=6626 secs]   MSE loss: 0.802
[epoch=9/10, iter=2500  elapsed_time=6778 secs]   MSE loss: 0.804
```

```
[epoch=10/10, iter= 500  elapsed_time=6928 secs]   MSE loss: 0.803
[epoch=10/10, iter=1000  elapsed_time=7080 secs]   MSE loss: 0.799
[epoch=10/10, iter=1500  elapsed_time=7231 secs]   MSE loss: 0.798
[epoch=10/10, iter=2000  elapsed_time=7382 secs]   MSE loss: 0.799
[epoch=10/10, iter=2500  elapsed_time=7533 secs]   MSE loss: 0.798

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

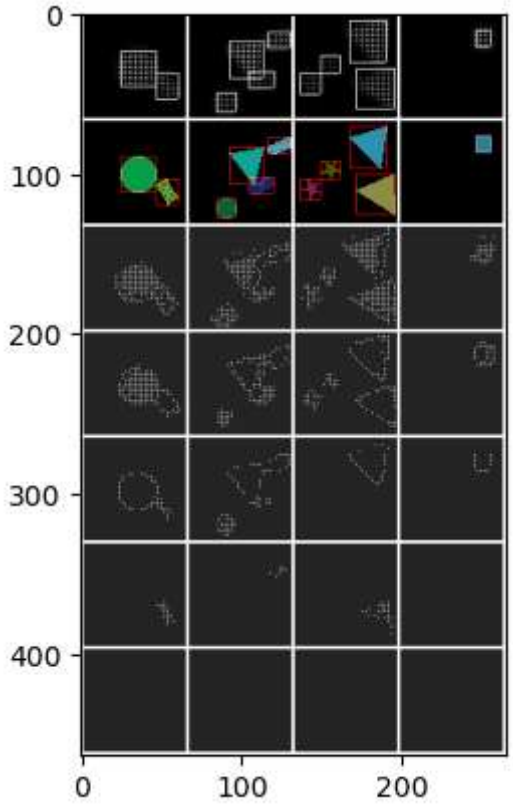
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



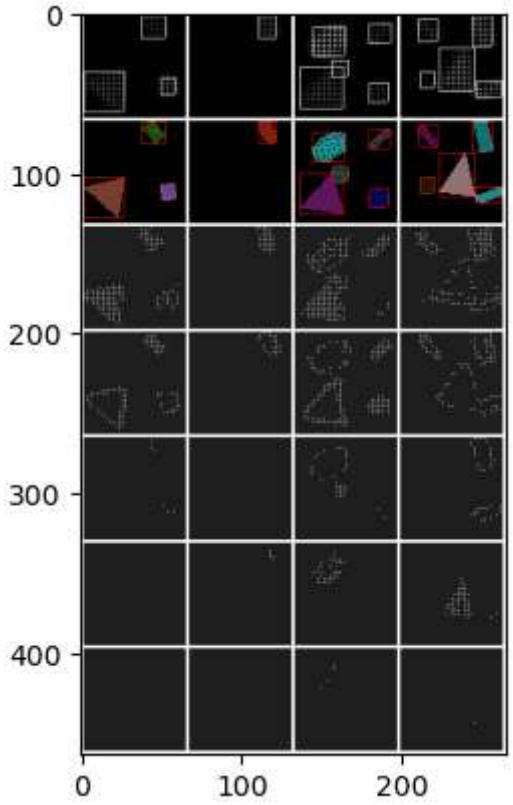

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



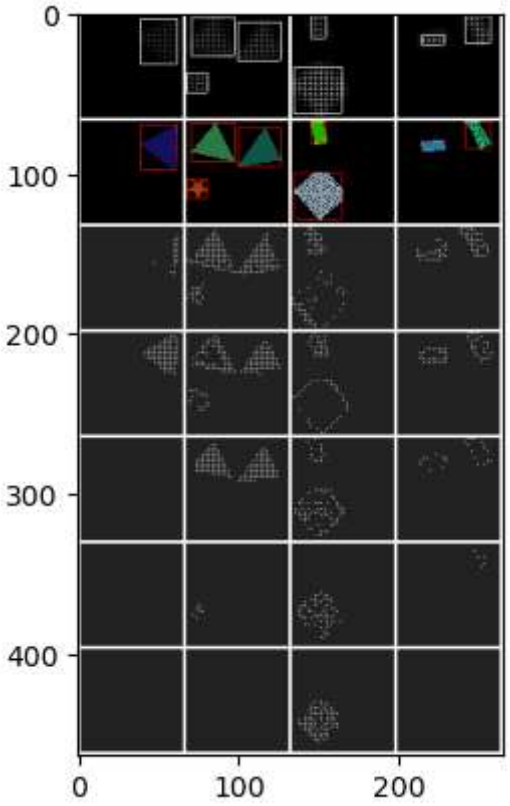
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



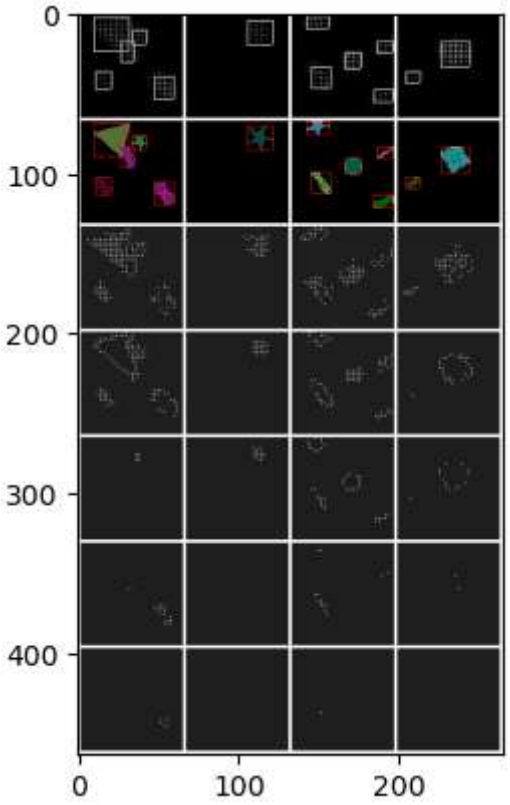
Showing output for test batch 151:

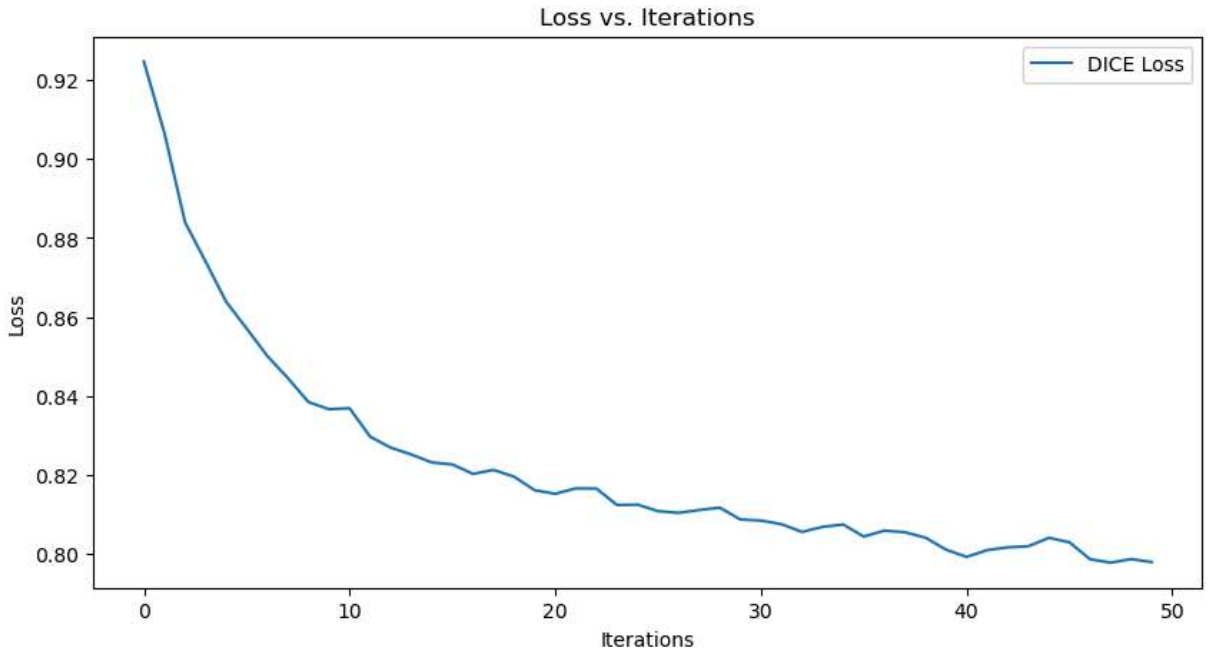
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1034, Mean Pixel Accuracy: 0.0435
```

2.3 MSE + Dice loss curve

In [22]:

```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_MSE_50",
                    momentum = 0.9,
                    learning_rate = 1e-5,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )
```

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

# DiceScale is a hyperparameter to scale the dice loss, adjust as needed
DICEMSELoss = segmenter.DICE_MSE_run_code_for_training_for_semantic_segmentation(mo

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(DICEMSELoss, label='DICE + MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

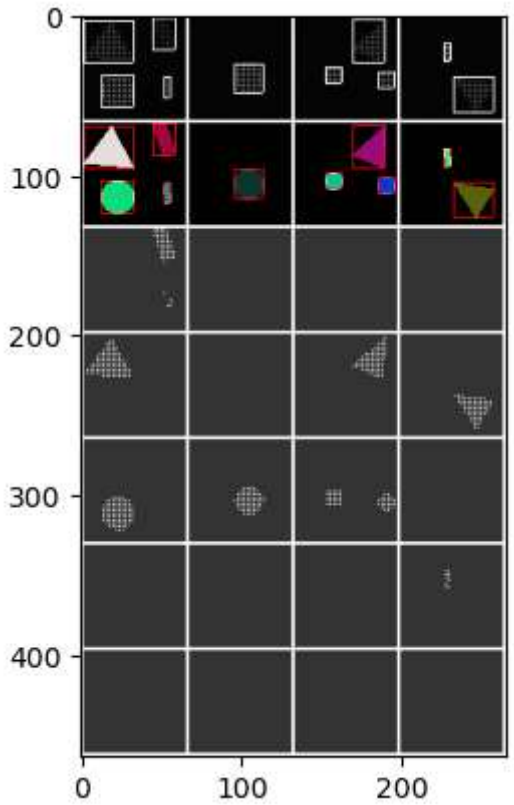
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=150 secs]   MSE loss: 502.520
[epoch=1/10, iter=1000  elapsed_time=301 secs]   MSE loss: 477.838
[epoch=1/10, iter=1500  elapsed_time=452 secs]   MSE loss: 482.912
[epoch=1/10, iter=2000  elapsed_time=602 secs]   MSE loss: 473.303
[epoch=1/10, iter=2500  elapsed_time=753 secs]   MSE loss: 468.412

[epoch=2/10, iter= 500  elapsed_time=904 secs]   MSE loss: 456.838
[epoch=2/10, iter=1000  elapsed_time=1055 secs]   MSE loss: 445.975
[epoch=2/10, iter=1500  elapsed_time=1206 secs]   MSE loss: 444.751
[epoch=2/10, iter=2000  elapsed_time=1357 secs]   MSE loss: 444.284
[epoch=2/10, iter=2500  elapsed_time=1508 secs]   MSE loss: 447.073

[epoch=3/10, iter= 500  elapsed_time=1659 secs]   MSE loss: 441.702
[epoch=3/10, iter=1000  elapsed_time=1811 secs]   MSE loss: 446.736
[epoch=3/10, iter=1500  elapsed_time=1961 secs]   MSE loss: 423.236
[epoch=3/10, iter=2000  elapsed_time=2113 secs]   MSE loss: 419.722
[epoch=3/10, iter=2500  elapsed_time=2264 secs]   MSE loss: 424.594

[epoch=4/10, iter= 500  elapsed_time=2415 secs]   MSE loss: 430.648
[epoch=4/10, iter=1000  elapsed_time=2566 secs]   MSE loss: 418.737
[epoch=4/10, iter=1500  elapsed_time=2717 secs]   MSE loss: 419.558
[epoch=4/10, iter=2000  elapsed_time=2869 secs]   MSE loss: 414.736
[epoch=4/10, iter=2500  elapsed_time=3020 secs]   MSE loss: 422.064

[epoch=5/10, iter= 500  elapsed_time=3171 secs]   MSE loss: 418.253
[epoch=5/10, iter=1000  elapsed_time=3322 secs]   MSE loss: 413.722
[epoch=5/10, iter=1500  elapsed_time=3474 secs]   MSE loss: 418.265
[epoch=5/10, iter=2000  elapsed_time=3625 secs]   MSE loss: 412.608
[epoch=5/10, iter=2500  elapsed_time=3776 secs]   MSE loss: 411.511

[epoch=6/10, iter= 500  elapsed_time=3927 secs]   MSE loss: 411.536
[epoch=6/10, iter=1000  elapsed_time=4078 secs]   MSE loss: 409.638
[epoch=6/10, iter=1500  elapsed_time=4228 secs]   MSE loss: 414.878
[epoch=6/10, iter=2000  elapsed_time=4379 secs]   MSE loss: 409.947
[epoch=6/10, iter=2500  elapsed_time=4530 secs]   MSE loss: 408.174

[epoch=7/10, iter= 500  elapsed_time=4681 secs]   MSE loss: 415.244
[epoch=7/10, iter=1000  elapsed_time=4832 secs]   MSE loss: 406.762
[epoch=7/10, iter=1500  elapsed_time=4982 secs]   MSE loss: 406.654
[epoch=7/10, iter=2000  elapsed_time=5133 secs]   MSE loss: 407.837
[epoch=7/10, iter=2500  elapsed_time=5285 secs]   MSE loss: 400.477

[epoch=8/10, iter= 500  elapsed_time=5435 secs]   MSE loss: 399.496
[epoch=8/10, iter=1000  elapsed_time=5586 secs]   MSE loss: 401.109
[epoch=8/10, iter=1500  elapsed_time=5737 secs]   MSE loss: 402.127
[epoch=8/10, iter=2000  elapsed_time=5889 secs]   MSE loss: 405.306
[epoch=8/10, iter=2500  elapsed_time=6040 secs]   MSE loss: 414.429

[epoch=9/10, iter= 500  elapsed_time=6190 secs]   MSE loss: 395.436
[epoch=9/10, iter=1000  elapsed_time=6342 secs]   MSE loss: 399.437
[epoch=9/10, iter=1500  elapsed_time=6493 secs]   MSE loss: 405.131
[epoch=9/10, iter=2000  elapsed_time=6644 secs]   MSE loss: 406.434
[epoch=9/10, iter=2500  elapsed_time=6795 secs]   MSE loss: 404.434
```

```
[epoch=10/10, iter= 500  elapsed_time=6947 secs]   MSE loss: 400.803
[epoch=10/10, iter=1000  elapsed_time=7097 secs]   MSE loss: 406.167
[epoch=10/10, iter=1500  elapsed_time=7249 secs]   MSE loss: 394.616
[epoch=10/10, iter=2000  elapsed_time=7401 secs]   MSE loss: 396.047
[epoch=10/10, iter=2500  elapsed_time=7552 secs]   MSE loss: 403.835

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

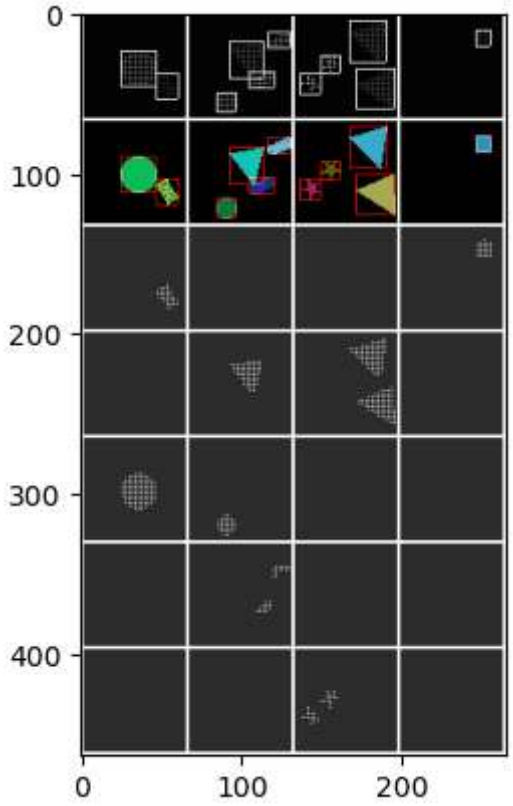
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



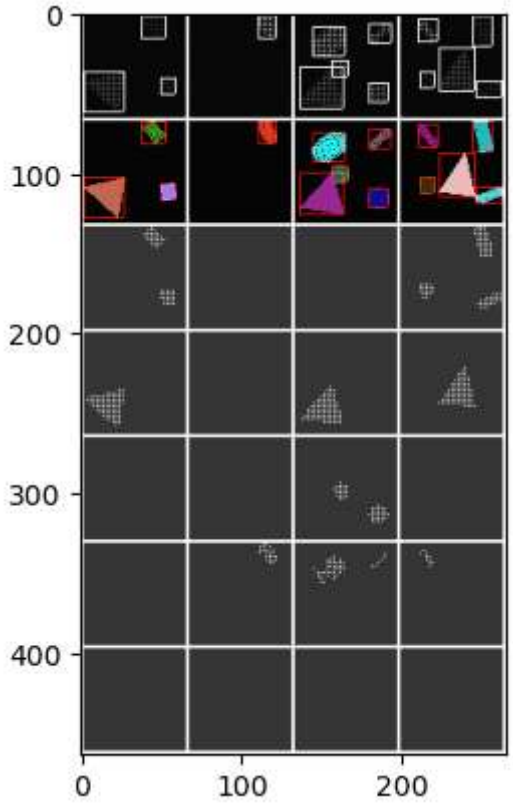

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



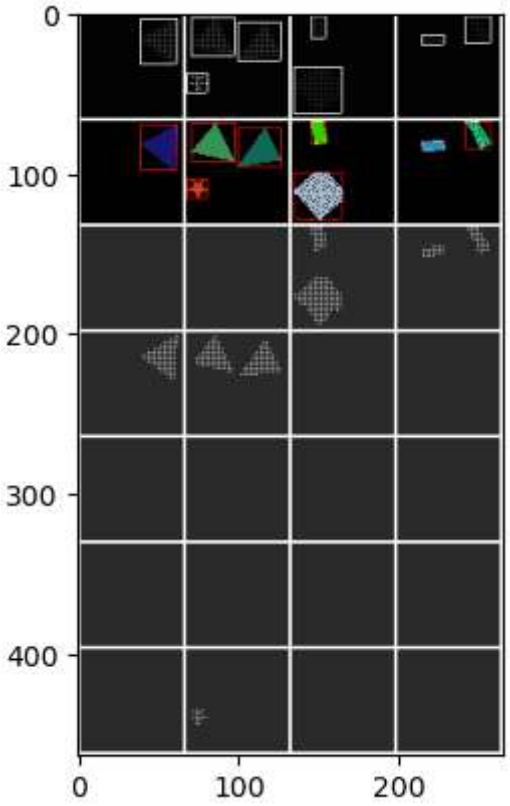
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



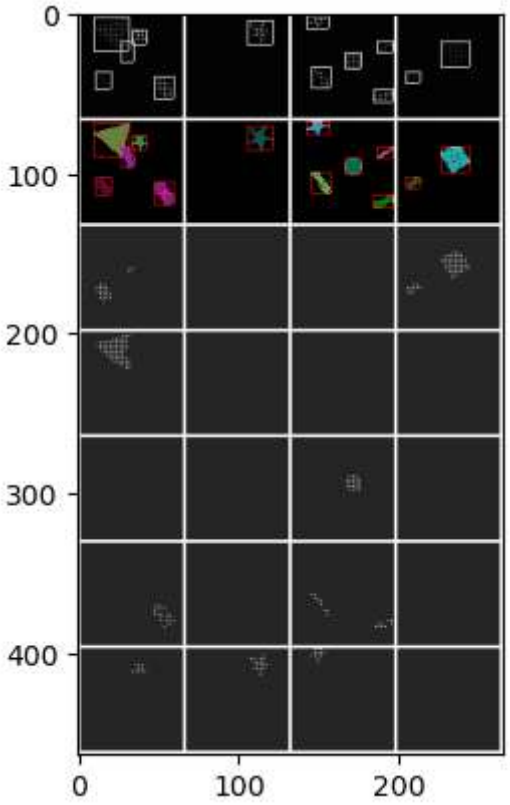
Showing output for test batch 151:

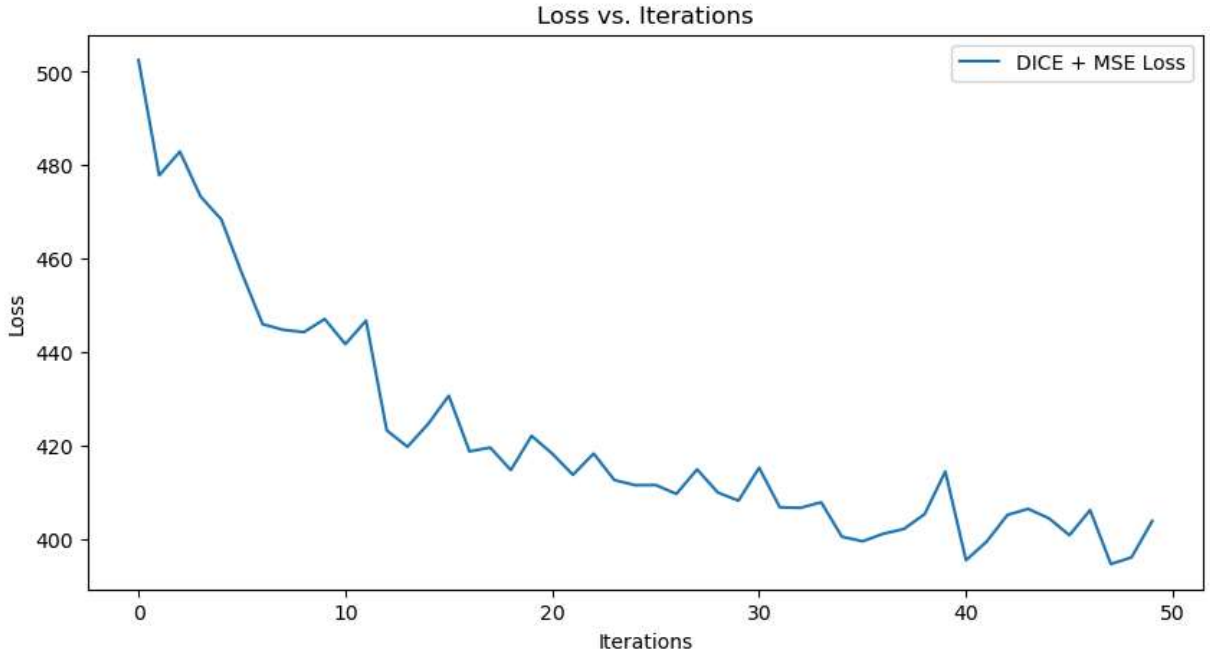
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1074, Mean Pixel Accuracy: 0.1035
```

3 Training worst curves

3.1 MSE loss curve

In [ ]:
```
# this code is mostly borrow from DLstudio's example
dls = DLStudio(
#                  dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                   dataroot = "./../data/datasets_for_DLStudio/data/",
                   image_size = [64,64],
                   path_saved_model = "./saved_model_MSE",
                   momentum = 0.9,
                   learning_rate = 1e-4,
                   epochs = 10,
                   batch_size = 4,
                   classes = ('rectangle','triangle','disk','oval','star'),
                   use_gpu = True,
                )
```

```
segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

MSELoss = segmenter.MSE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(MSELoss, label='MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

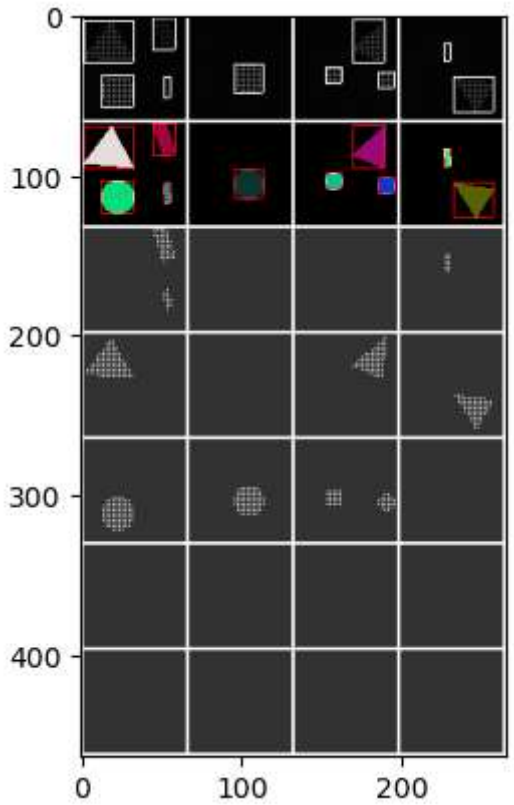
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=152 secs]   loss: 446.567
[epoch=1/10, iter=1000  elapsed_time=303 secs]   loss: 420.760
[epoch=1/10, iter=1500  elapsed_time=454 secs]   loss: 403.261
[epoch=1/10, iter=2000  elapsed_time=604 secs]   loss: 386.206
[epoch=1/10, iter=2500  elapsed_time=755 secs]   loss: 384.735

[epoch=2/10, iter= 500  elapsed_time=906 secs]   loss: 379.357
[epoch=2/10, iter=1000  elapsed_time=1057 secs]   loss: 383.416
[epoch=2/10, iter=1500  elapsed_time=1207 secs]   loss: 375.509
[epoch=2/10, iter=2000  elapsed_time=1359 secs]   loss: 367.598
[epoch=2/10, iter=2500  elapsed_time=1509 secs]   loss: 368.523

[epoch=3/10, iter= 500  elapsed_time=1660 secs]   loss: 368.174
[epoch=3/10, iter=1000  elapsed_time=1811 secs]   loss: 365.481
[epoch=3/10, iter=1500  elapsed_time=1962 secs]   loss: 365.292
[epoch=3/10, iter=2000  elapsed_time=2113 secs]   loss: 373.041
[epoch=3/10, iter=2500  elapsed_time=2263 secs]   loss: 363.686

[epoch=4/10, iter= 500  elapsed_time=2414 secs]   loss: 365.801
[epoch=4/10, iter=1000  elapsed_time=2566 secs]   loss: 368.937
[epoch=4/10, iter=1500  elapsed_time=2717 secs]   loss: 355.672
[epoch=4/10, iter=2000  elapsed_time=2869 secs]   loss: 357.318
[epoch=4/10, iter=2500  elapsed_time=3020 secs]   loss: 368.256

[epoch=5/10, iter= 500  elapsed_time=3171 secs]   loss: 360.771
[epoch=5/10, iter=1000  elapsed_time=3322 secs]   loss: 365.935
[epoch=5/10, iter=1500  elapsed_time=3473 secs]   loss: 362.723
[epoch=5/10, iter=2000  elapsed_time=3624 secs]   loss: 361.094
[epoch=5/10, iter=2500  elapsed_time=3775 secs]   loss: 357.179

[epoch=6/10, iter= 500  elapsed_time=3925 secs]   loss: 351.537
[epoch=6/10, iter=1000  elapsed_time=4076 secs]   loss: 358.972
[epoch=6/10, iter=1500  elapsed_time=4227 secs]   loss: 364.257
[epoch=6/10, iter=2000  elapsed_time=4377 secs]   loss: 366.506
[epoch=6/10, iter=2500  elapsed_time=4528 secs]   loss: 360.839

[epoch=7/10, iter= 500  elapsed_time=4679 secs]   loss: 357.885
[epoch=7/10, iter=1000  elapsed_time=4830 secs]   loss: 364.230
[epoch=7/10, iter=1500  elapsed_time=4981 secs]   loss: 351.422
[epoch=7/10, iter=2000  elapsed_time=5132 secs]   loss: 353.012
[epoch=7/10, iter=2500  elapsed_time=5283 secs]   loss: 371.268

[epoch=8/10, iter= 500  elapsed_time=5434 secs]   loss: 359.997
[epoch=8/10, iter=1000  elapsed_time=5585 secs]   loss: 357.009
[epoch=8/10, iter=1500  elapsed_time=5735 secs]   loss: 357.837
[epoch=8/10, iter=2000  elapsed_time=5886 secs]   loss: 359.978
[epoch=8/10, iter=2500  elapsed_time=6037 secs]   loss: 361.532

[epoch=9/10, iter= 500  elapsed_time=6189 secs]   loss: 357.377
[epoch=9/10, iter=1000  elapsed_time=6339 secs]   loss: 357.166
[epoch=9/10, iter=1500  elapsed_time=6490 secs]   loss: 361.841
[epoch=9/10, iter=2000  elapsed_time=6641 secs]   loss: 368.325
[epoch=9/10, iter=2500  elapsed_time=6792 secs]   loss: 349.296
```

```
[epoch=10/10, iter= 500  elapsed_time=6942 secs]   loss: 359.089
[epoch=10/10, iter=1000  elapsed_time=7093 secs]   loss: 350.737
[epoch=10/10, iter=1500  elapsed_time=7244 secs]   loss: 365.837
[epoch=10/10, iter=2000  elapsed_time=7395 secs]   loss: 362.589
[epoch=10/10, iter=2500  elapsed_time=7546 secs]   loss: 355.097

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

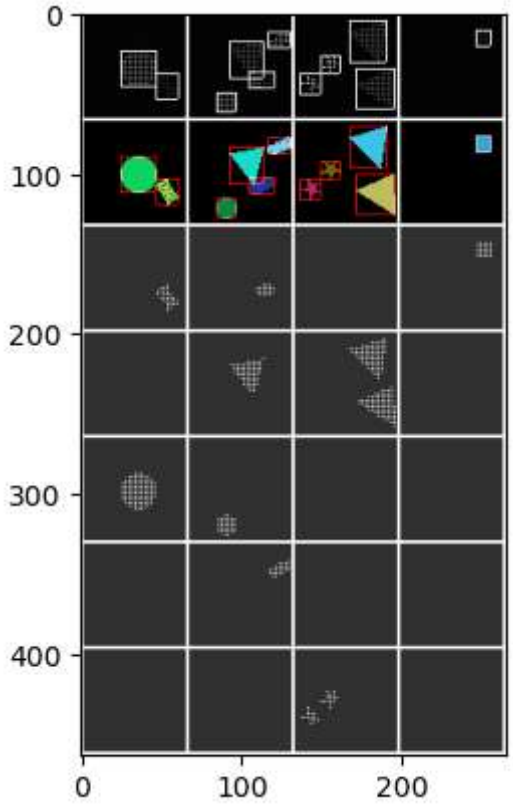
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



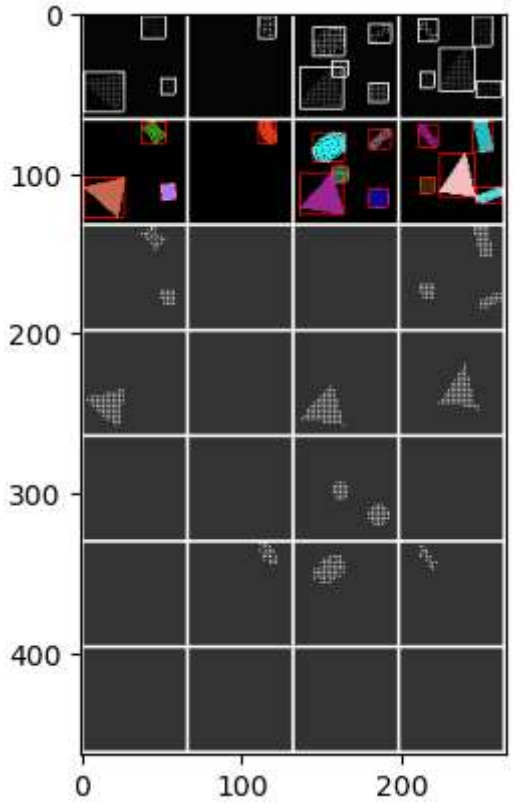

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



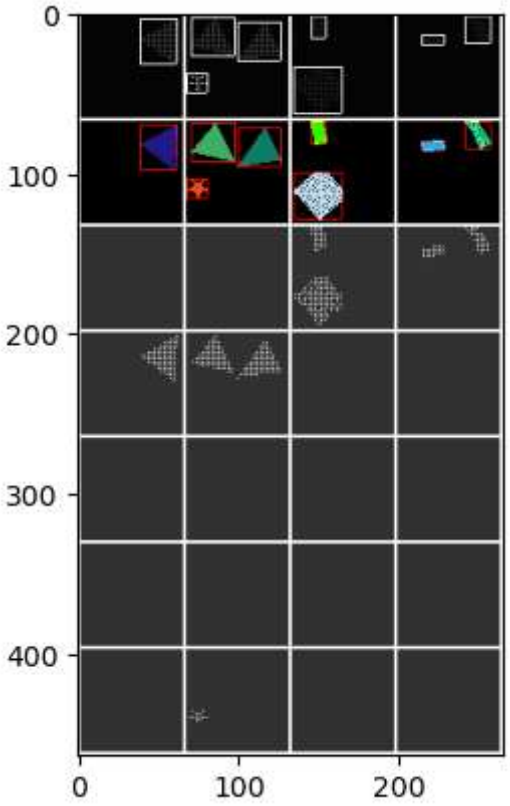
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



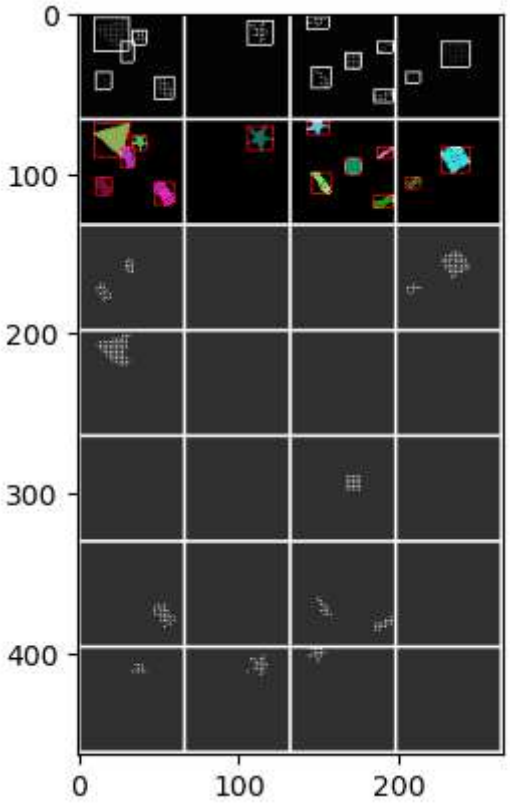
Showing output for test batch 151:

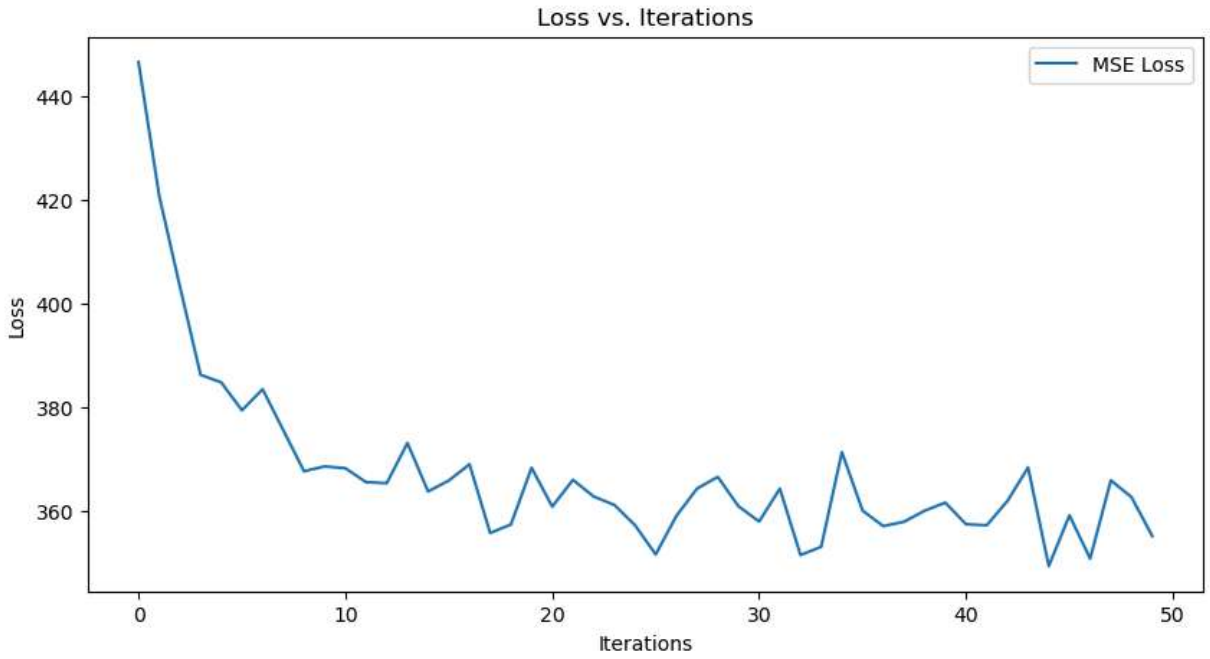
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1058, Mean Pixel Accuracy: 0.0940
```

3.2 Dice loss curve

In [ ]:

```
# this code is mostly borrowed from DLStudio

dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
```

```
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

DICELoss = segmenter.DICE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(DICELoss, label='DICE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

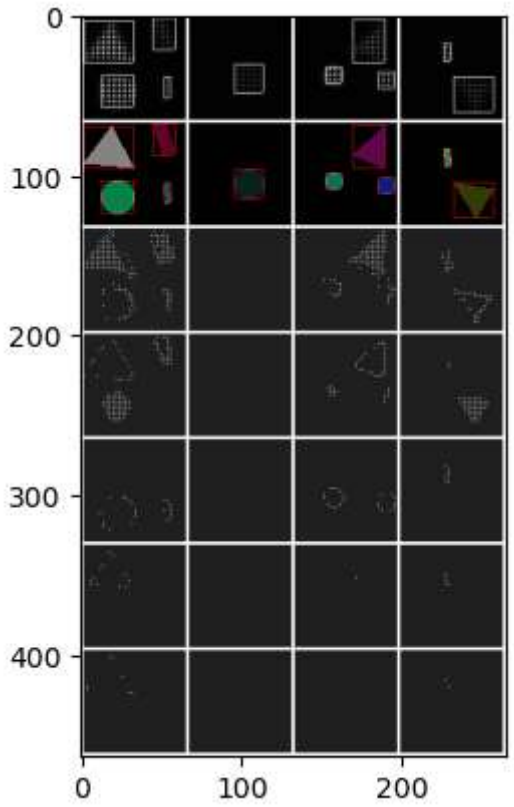
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=151 secs]   MSE loss: 0.876
[epoch=1/10, iter=1000  elapsed_time=302 secs]   MSE loss: 0.842
[epoch=1/10, iter=1500  elapsed_time=452 secs]   MSE loss: 0.833
[epoch=1/10, iter=2000  elapsed_time=603 secs]   MSE loss: 0.824
[epoch=1/10, iter=2500  elapsed_time=753 secs]   MSE loss: 0.821

[epoch=2/10, iter= 500  elapsed_time=904 secs]   MSE loss: 0.813
[epoch=2/10, iter=1000  elapsed_time=1055 secs]   MSE loss: 0.810
[epoch=2/10, iter=1500  elapsed_time=1207 secs]   MSE loss: 0.800
[epoch=2/10, iter=2000  elapsed_time=1358 secs]   MSE loss: 0.797
[epoch=2/10, iter=2500  elapsed_time=1509 secs]   MSE loss: 0.792

[epoch=3/10, iter= 500  elapsed_time=1660 secs]   MSE loss: 0.790
[epoch=3/10, iter=1000  elapsed_time=1811 secs]   MSE loss: 0.789
[epoch=3/10, iter=1500  elapsed_time=1962 secs]   MSE loss: 0.784
[epoch=3/10, iter=2000  elapsed_time=2113 secs]   MSE loss: 0.783
[epoch=3/10, iter=2500  elapsed_time=2264 secs]   MSE loss: 0.782

[epoch=4/10, iter= 500  elapsed_time=2415 secs]   MSE loss: 0.781
[epoch=4/10, iter=1000  elapsed_time=2566 secs]   MSE loss: 0.777
[epoch=4/10, iter=1500  elapsed_time=2717 secs]   MSE loss: 0.776
[epoch=4/10, iter=2000  elapsed_time=2868 secs]   MSE loss: 0.772
[epoch=4/10, iter=2500  elapsed_time=3019 secs]   MSE loss: 0.779

[epoch=5/10, iter= 500  elapsed_time=3170 secs]   MSE loss: 0.770
[epoch=5/10, iter=1000  elapsed_time=3321 secs]   MSE loss: 0.771
[epoch=5/10, iter=1500  elapsed_time=3472 secs]   MSE loss: 0.769
[epoch=5/10, iter=2000  elapsed_time=3622 secs]   MSE loss: 0.771
[epoch=5/10, iter=2500  elapsed_time=3774 secs]   MSE loss: 0.771

[epoch=6/10, iter= 500  elapsed_time=3924 secs]   MSE loss: 0.766
[epoch=6/10, iter=1000  elapsed_time=4075 secs]   MSE loss: 0.764
[epoch=6/10, iter=1500  elapsed_time=4226 secs]   MSE loss: 0.766
[epoch=6/10, iter=2000  elapsed_time=4377 secs]   MSE loss: 0.765
[epoch=6/10, iter=2500  elapsed_time=4528 secs]   MSE loss: 0.763

[epoch=7/10, iter= 500  elapsed_time=4679 secs]   MSE loss: 0.760
[epoch=7/10, iter=1000  elapsed_time=4830 secs]   MSE loss: 0.765
[epoch=7/10, iter=1500  elapsed_time=4981 secs]   MSE loss: 0.759
[epoch=7/10, iter=2000  elapsed_time=5132 secs]   MSE loss: 0.757
[epoch=7/10, iter=2500  elapsed_time=5284 secs]   MSE loss: 0.759

[epoch=8/10, iter= 500  elapsed_time=5434 secs]   MSE loss: 0.756
[epoch=8/10, iter=1000  elapsed_time=5586 secs]   MSE loss: 0.756
[epoch=8/10, iter=1500  elapsed_time=5737 secs]   MSE loss: 0.760
[epoch=8/10, iter=2000  elapsed_time=5888 secs]   MSE loss: 0.754
[epoch=8/10, iter=2500  elapsed_time=6039 secs]   MSE loss: 0.754

[epoch=9/10, iter= 500  elapsed_time=6190 secs]   MSE loss: 0.753
[epoch=9/10, iter=1000  elapsed_time=6341 secs]   MSE loss: 0.752
[epoch=9/10, iter=1500  elapsed_time=6492 secs]   MSE loss: 0.751
[epoch=9/10, iter=2000  elapsed_time=6643 secs]   MSE loss: 0.751
[epoch=9/10, iter=2500  elapsed_time=6794 secs]   MSE loss: 0.756
```

```
[epoch=10/10, iter= 500  elapsed_time=6945 secs]   MSE loss: 0.750
[epoch=10/10, iter=1000  elapsed_time=7097 secs]   MSE loss: 0.752
[epoch=10/10, iter=1500  elapsed_time=7248 secs]   MSE loss: 0.750
[epoch=10/10, iter=2000  elapsed_time=7400 secs]   MSE loss: 0.750
[epoch=10/10, iter=2500  elapsed_time=7551 secs]   MSE loss: 0.748

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

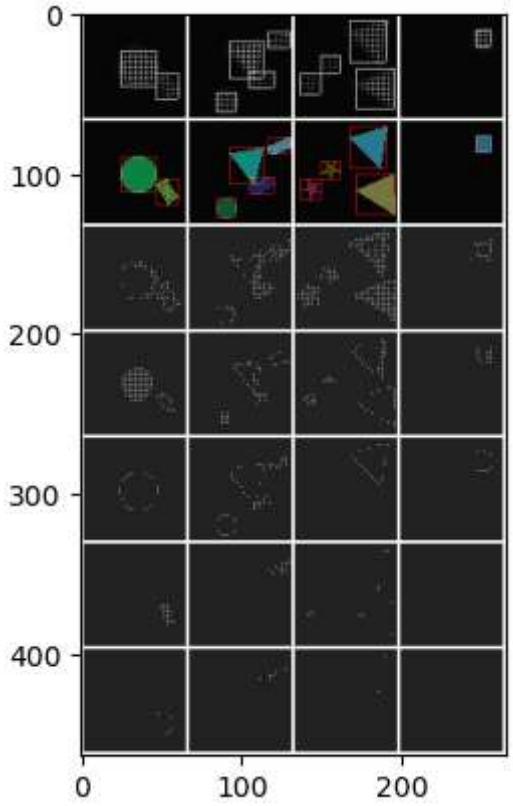
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



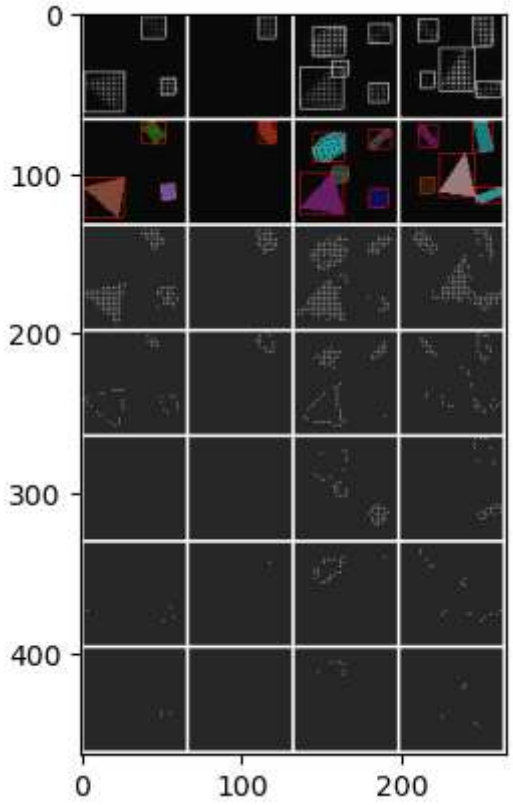

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



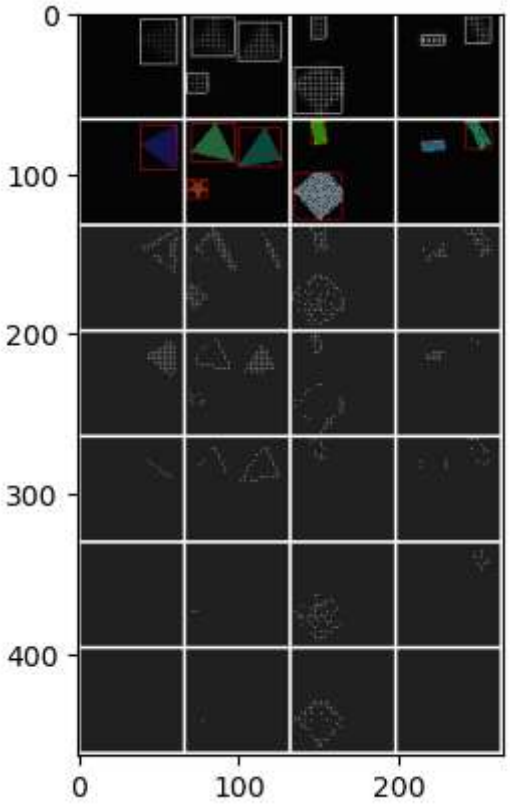
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



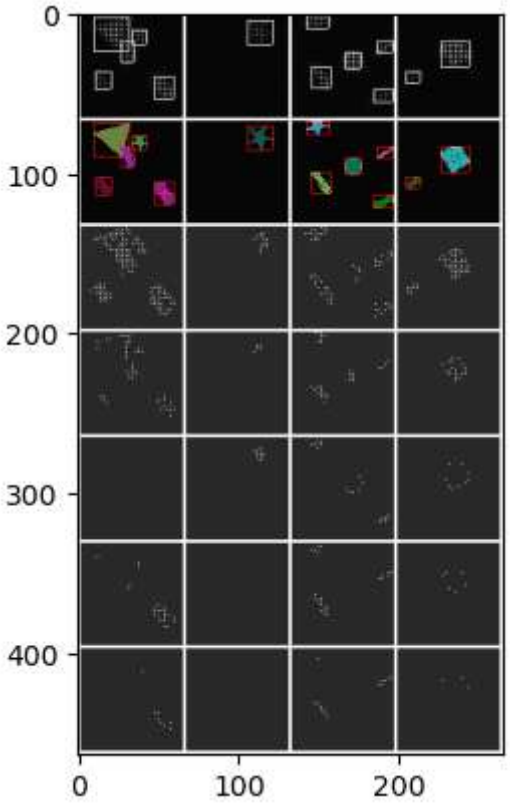
Showing output for test batch 151:

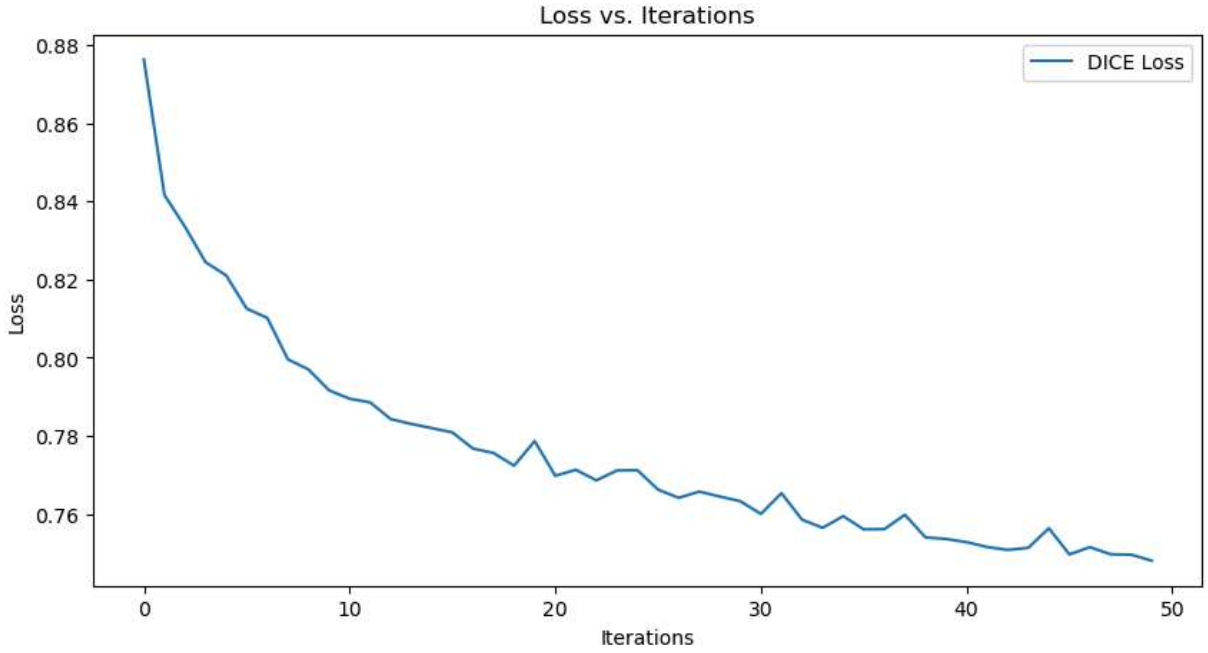
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.0737, Mean Pixel Accuracy: 0.6970
```

3.3 MSE + Dice loss curve

In [ ]:

```
# this code is mostly borrowed from DLStudio

dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_MSE_50",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
```

```
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

# DiceScale is a hyperparameter to scale the dice loss, adjust as needed
DICEMSELoss = segmenter.DICE_MSE_run_code_for_training_for_semantic_segmentation(mo

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(DICEMSELoss, label='DICE + MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

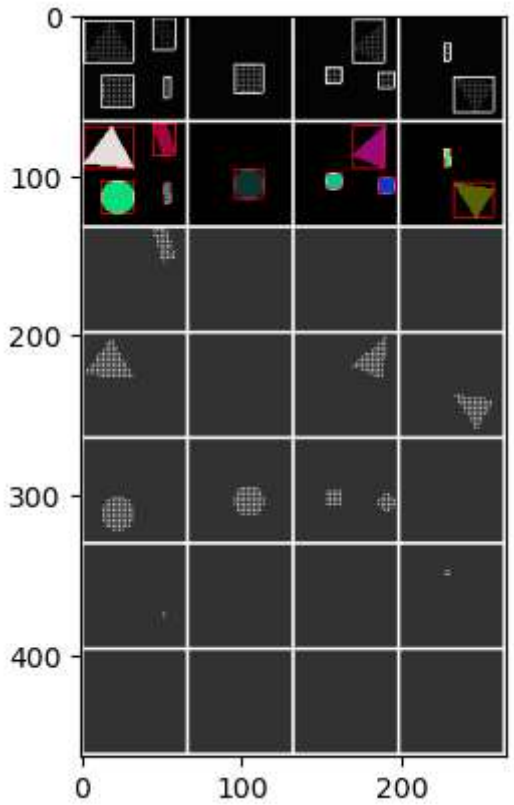
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=150 secs]   MSE loss: 482.681
[epoch=1/10, iter=1000  elapsed_time=300 secs]   MSE loss: 455.208
[epoch=1/10, iter=1500  elapsed_time=451 secs]   MSE loss: 440.066
[epoch=1/10, iter=2000  elapsed_time=601 secs]   MSE loss: 426.330
[epoch=1/10, iter=2500  elapsed_time=752 secs]   MSE loss: 409.871

[epoch=2/10, iter= 500  elapsed_time=902 secs]   MSE loss: 415.901
[epoch=2/10, iter=1000  elapsed_time=1053 secs]   MSE loss: 406.139
[epoch=2/10, iter=1500  elapsed_time=1203 secs]   MSE loss: 398.515
[epoch=2/10, iter=2000  elapsed_time=1354 secs]   MSE loss: 399.718
[epoch=2/10, iter=2500  elapsed_time=1505 secs]   MSE loss: 404.180

[epoch=3/10, iter= 500  elapsed_time=1655 secs]   MSE loss: 401.473
[epoch=3/10, iter=1000  elapsed_time=1807 secs]   MSE loss: 397.693
[epoch=3/10, iter=1500  elapsed_time=1958 secs]   MSE loss: 396.237
[epoch=3/10, iter=2000  elapsed_time=2109 secs]   MSE loss: 393.247
[epoch=3/10, iter=2500  elapsed_time=2259 secs]   MSE loss: 392.699

[epoch=4/10, iter= 500  elapsed_time=2410 secs]   MSE loss: 396.270
[epoch=4/10, iter=1000  elapsed_time=2561 secs]   MSE loss: 393.513
[epoch=4/10, iter=1500  elapsed_time=2712 secs]   MSE loss: 394.967
[epoch=4/10, iter=2000  elapsed_time=2863 secs]   MSE loss: 393.619
[epoch=4/10, iter=2500  elapsed_time=3014 secs]   MSE loss: 391.298

[epoch=5/10, iter= 500  elapsed_time=3165 secs]   MSE loss: 392.252
[epoch=5/10, iter=1000  elapsed_time=3316 secs]   MSE loss: 390.504
[epoch=5/10, iter=1500  elapsed_time=3468 secs]   MSE loss: 391.034
[epoch=5/10, iter=2000  elapsed_time=3619 secs]   MSE loss: 396.063
[epoch=5/10, iter=2500  elapsed_time=3770 secs]   MSE loss: 385.170

[epoch=6/10, iter= 500  elapsed_time=3921 secs]   MSE loss: 386.327
[epoch=6/10, iter=1000  elapsed_time=4072 secs]   MSE loss: 393.290
[epoch=6/10, iter=1500  elapsed_time=4223 secs]   MSE loss: 391.955
[epoch=6/10, iter=2000  elapsed_time=4374 secs]   MSE loss: 388.560
[epoch=6/10, iter=2500  elapsed_time=4524 secs]   MSE loss: 389.654

[epoch=7/10, iter= 500  elapsed_time=4675 secs]   MSE loss: 384.834
[epoch=7/10, iter=1000  elapsed_time=4826 secs]   MSE loss: 398.135
[epoch=7/10, iter=1500  elapsed_time=4977 secs]   MSE loss: 381.490
[epoch=7/10, iter=2000  elapsed_time=5127 secs]   MSE loss: 389.477
[epoch=7/10, iter=2500  elapsed_time=5278 secs]   MSE loss: 391.135

[epoch=8/10, iter= 500  elapsed_time=5429 secs]   MSE loss: 388.669
[epoch=8/10, iter=1000  elapsed_time=5580 secs]   MSE loss: 391.289
[epoch=8/10, iter=1500  elapsed_time=5731 secs]   MSE loss: 377.545
[epoch=8/10, iter=2000  elapsed_time=5882 secs]   MSE loss: 394.650
[epoch=8/10, iter=2500  elapsed_time=6033 secs]   MSE loss: 391.657

[epoch=9/10, iter= 500  elapsed_time=6184 secs]   MSE loss: 386.666
[epoch=9/10, iter=1000  elapsed_time=6335 secs]   MSE loss: 389.474
[epoch=9/10, iter=1500  elapsed_time=6486 secs]   MSE loss: 391.379
[epoch=9/10, iter=2000  elapsed_time=6637 secs]   MSE loss: 385.502
[epoch=9/10, iter=2500  elapsed_time=6788 secs]   MSE loss: 387.783
```

```
[epoch=10/10, iter= 500  elapsed_time=6938 secs]   MSE loss: 388.884
[epoch=10/10, iter=1000  elapsed_time=7089 secs]   MSE loss: 390.502
[epoch=10/10, iter=1500  elapsed_time=7240 secs]   MSE loss: 386.594
[epoch=10/10, iter=2000  elapsed_time=7391 secs]   MSE loss: 384.029
[epoch=10/10, iter=2500  elapsed_time=7542 secs]   MSE loss: 389.433

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

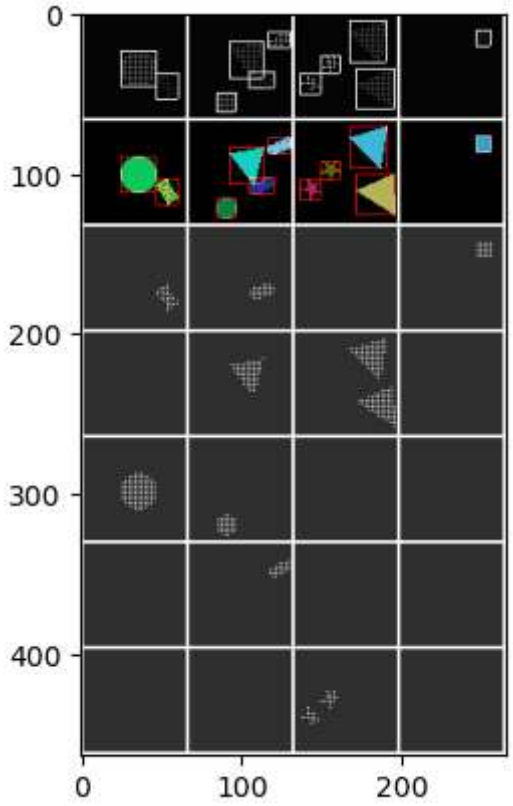
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



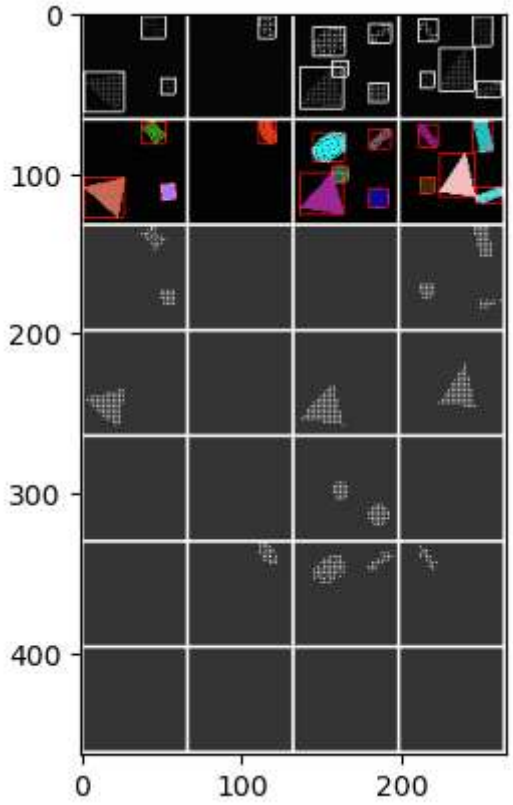

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



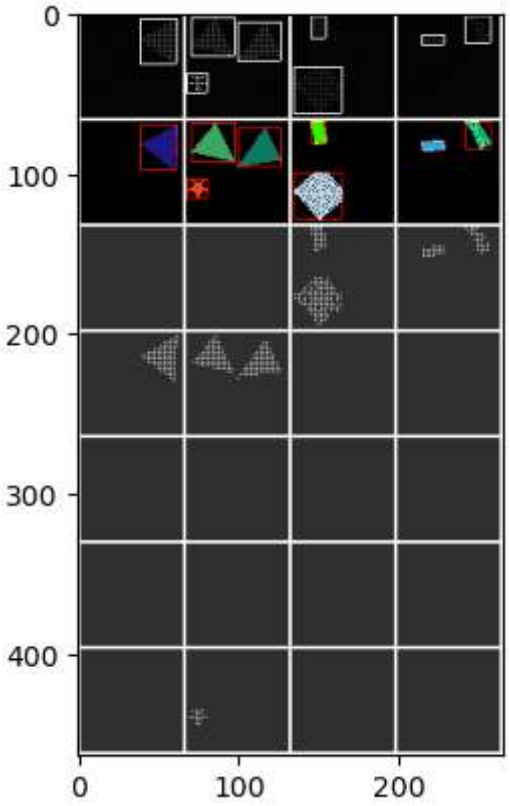
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



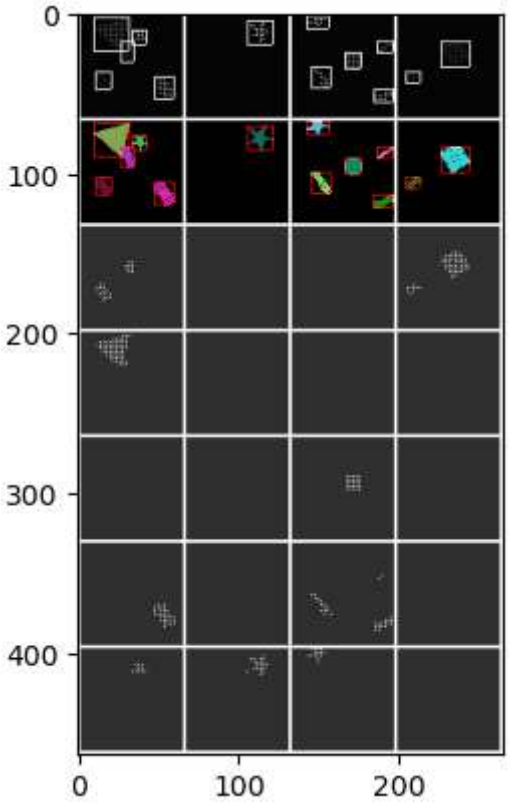
Showing output for test batch 151:

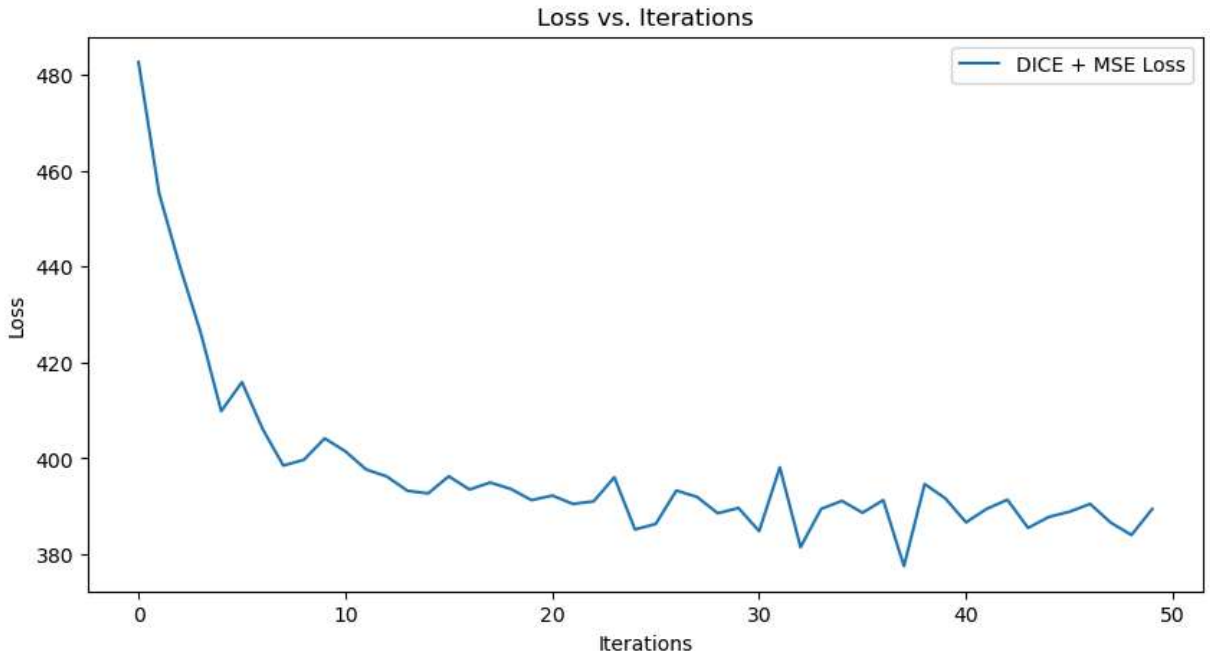
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1059, Mean Pixel Accuracy: 0.0907
```

4 Observation on coefficient effect of MSE+Dice loss

Result first, here is the table for summary:

	Mean IoU	Mean Pixel Accuracy
DICE + MSE, DiceScale=150	0.1075	0.1088
DICE + MSE, DiceScale=5	0.1102	0.1429

From the results, using a lower DiceScale value (5) leads to better performance in both Mean IoU and Mean Pixel Accuracy compared to a higher DiceScale (150). In my experience, a lower DiceScale, effectively placing more weight on the MSE component relative to the DICE loss, tends to produce better results. That said, in general, either significantly lower or higher DiceScale values often yield better performance, while mid-range values may be less

effective. This suggests that the balance between DICE and MSE loss needs to be carefully tuned depending on the dataset and model behavior.

4.1 Best set of scale values

In [ ]:
```
# this code is mostly borrowed from DLStudio

dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_MSE_5",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

# DiceScale is a hyperparameter to scale the dice loss, adjust as needed
DICEMSELoss = segmenter.DICE_MSE_run_code_for_training_for_semantic_segmentation(mo

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
```

```
plt.plot(DICEMSELoss, label='DICE + MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

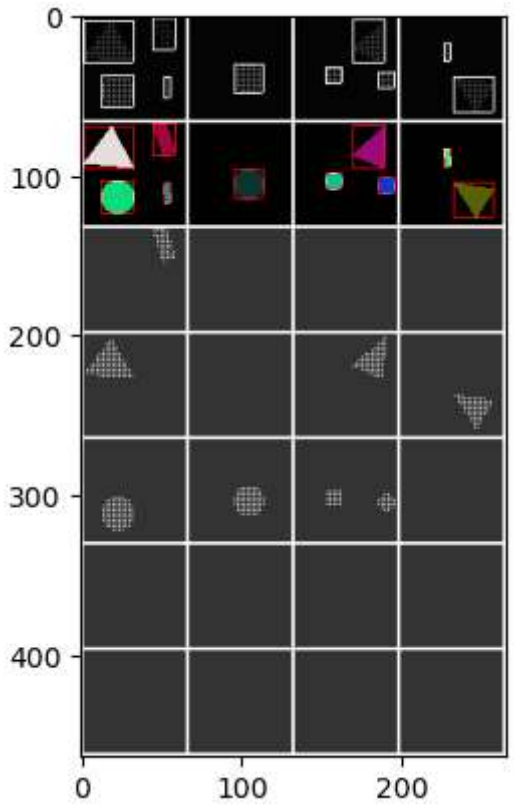
```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=151 secs]   MSE loss: 454.284
[epoch=1/10, iter=1000  elapsed_time=301 secs]   MSE loss: 413.857
[epoch=1/10, iter=1500  elapsed_time=452 secs]   MSE loss: 403.078
[epoch=1/10, iter=2000  elapsed_time=603 secs]   MSE loss: 403.257
[epoch=1/10, iter=2500  elapsed_time=754 secs]   MSE loss: 387.056

[epoch=2/10, iter= 500  elapsed_time=904 secs]   MSE loss: 375.678
[epoch=2/10, iter=1000  elapsed_time=1055 secs]   MSE loss: 377.136
[epoch=2/10, iter=1500  elapsed_time=1206 secs]   MSE loss: 380.228
[epoch=2/10, iter=2000  elapsed_time=1357 secs]   MSE loss: 378.240
[epoch=2/10, iter=2500  elapsed_time=1508 secs]   MSE loss: 376.212

[epoch=3/10, iter= 500  elapsed_time=1659 secs]   MSE loss: 364.049
[epoch=3/10, iter=1000  elapsed_time=1810 secs]   MSE loss: 367.341
[epoch=3/10, iter=1500  elapsed_time=1961 secs]   MSE loss: 373.282
[epoch=3/10, iter=2000  elapsed_time=2112 secs]   MSE loss: 373.684
[epoch=3/10, iter=2500  elapsed_time=2263 secs]   MSE loss: 368.071

[epoch=4/10, iter= 500  elapsed_time=2415 secs]   MSE loss: 366.608
[epoch=4/10, iter=1000  elapsed_time=2566 secs]   MSE loss: 359.721
[epoch=4/10, iter=1500  elapsed_time=2717 secs]   MSE loss: 369.826
[epoch=4/10, iter=2000  elapsed_time=2868 secs]   MSE loss: 365.487
[epoch=4/10, iter=2500  elapsed_time=3019 secs]   MSE loss: 366.673

[epoch=5/10, iter= 500  elapsed_time=3170 secs]   MSE loss: 371.234
[epoch=5/10, iter=1000  elapsed_time=3321 secs]   MSE loss: 359.865
[epoch=5/10, iter=1500  elapsed_time=3472 secs]   MSE loss: 364.070
[epoch=5/10, iter=2000  elapsed_time=3624 secs]   MSE loss: 366.406
[epoch=5/10, iter=2500  elapsed_time=3774 secs]   MSE loss: 361.783

[epoch=6/10, iter= 500  elapsed_time=3925 secs]   MSE loss: 362.297
[epoch=6/10, iter=1000  elapsed_time=4076 secs]   MSE loss: 358.046
[epoch=6/10, iter=1500  elapsed_time=4227 secs]   MSE loss: 362.021
[epoch=6/10, iter=2000  elapsed_time=4378 secs]   MSE loss: 368.668
[epoch=6/10, iter=2500  elapsed_time=4529 secs]   MSE loss: 366.762

[epoch=7/10, iter= 500  elapsed_time=4680 secs]   MSE loss: 359.109
[epoch=7/10, iter=1000  elapsed_time=4832 secs]   MSE loss: 363.564
[epoch=7/10, iter=1500  elapsed_time=4982 secs]   MSE loss: 362.318
[epoch=7/10, iter=2000  elapsed_time=5133 secs]   MSE loss: 371.268
[epoch=7/10, iter=2500  elapsed_time=5284 secs]   MSE loss: 358.246

[epoch=8/10, iter= 500  elapsed_time=5435 secs]   MSE loss: 359.797
[epoch=8/10, iter=1000  elapsed_time=5586 secs]   MSE loss: 366.859
[epoch=8/10, iter=1500  elapsed_time=5737 secs]   MSE loss: 366.405
[epoch=8/10, iter=2000  elapsed_time=5888 secs]   MSE loss: 360.971
[epoch=8/10, iter=2500  elapsed_time=6040 secs]   MSE loss: 358.893

[epoch=9/10, iter= 500  elapsed_time=6191 secs]   MSE loss: 361.934
[epoch=9/10, iter=1000  elapsed_time=6343 secs]   MSE loss: 367.030
[epoch=9/10, iter=1500  elapsed_time=6494 secs]   MSE loss: 359.149
[epoch=9/10, iter=2000  elapsed_time=6645 secs]   MSE loss: 358.581
[epoch=9/10, iter=2500  elapsed_time=6796 secs]   MSE loss: 362.392
```

```
[epoch=10/10, iter= 500  elapsed_time=6947 secs]   MSE loss: 359.357
[epoch=10/10, iter=1000  elapsed_time=7098 secs]   MSE loss: 356.198
[epoch=10/10, iter=1500  elapsed_time=7249 secs]   MSE loss: 360.042
[epoch=10/10, iter=2000  elapsed_time=7400 secs]   MSE loss: 362.546
[epoch=10/10, iter=2500  elapsed_time=7551 secs]   MSE loss: 369.977

Finished Training
```

```
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

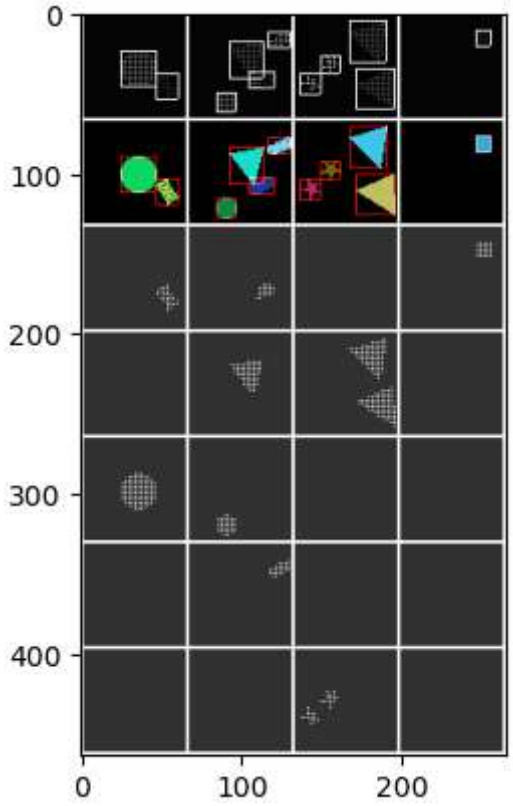
```
Showing output for test batch 1:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



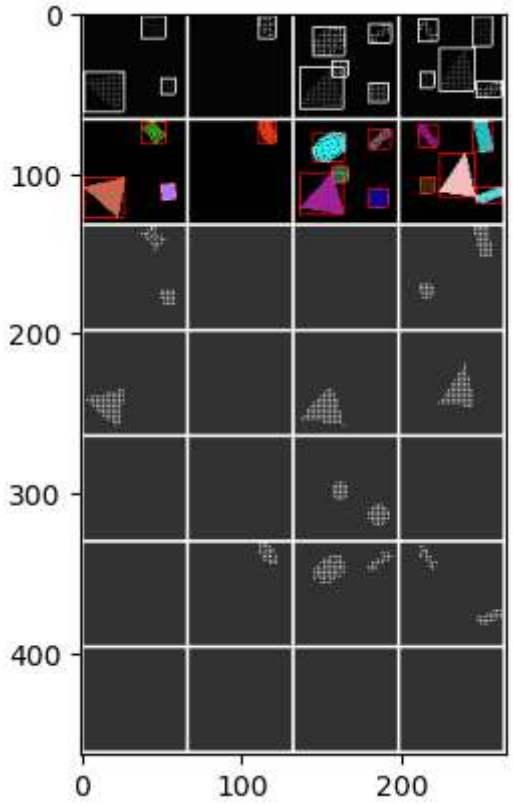

```
Showing output for test batch 51:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



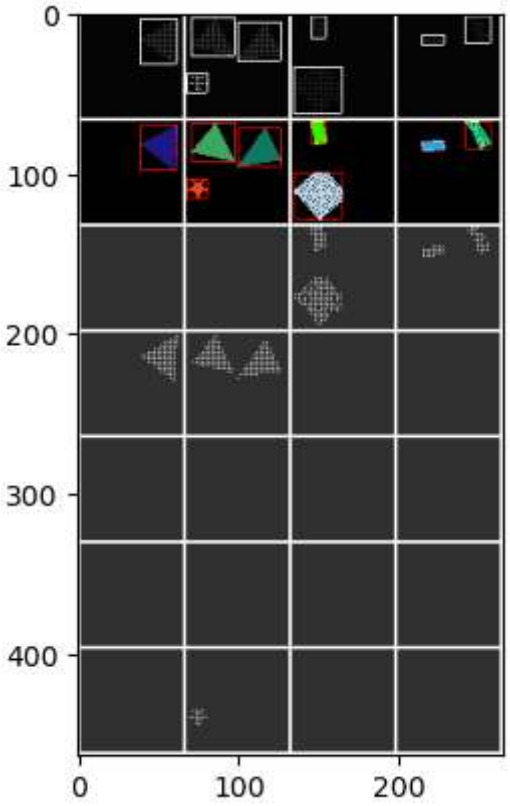
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



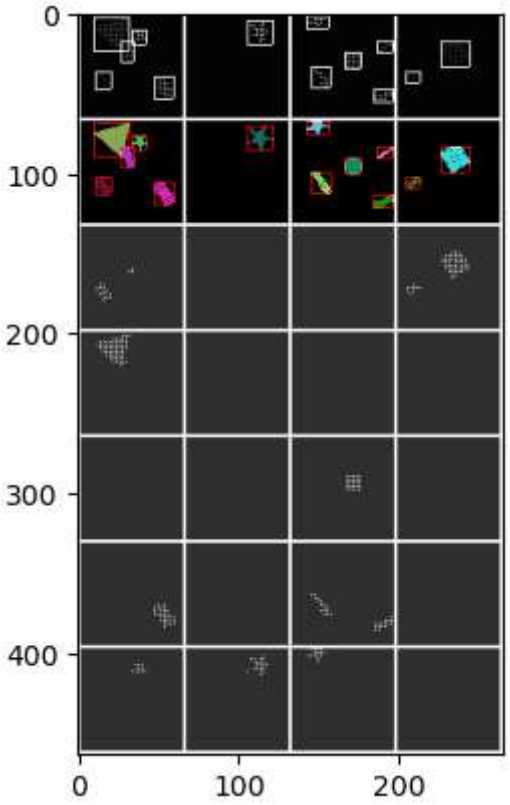
Showing output for test batch 151:

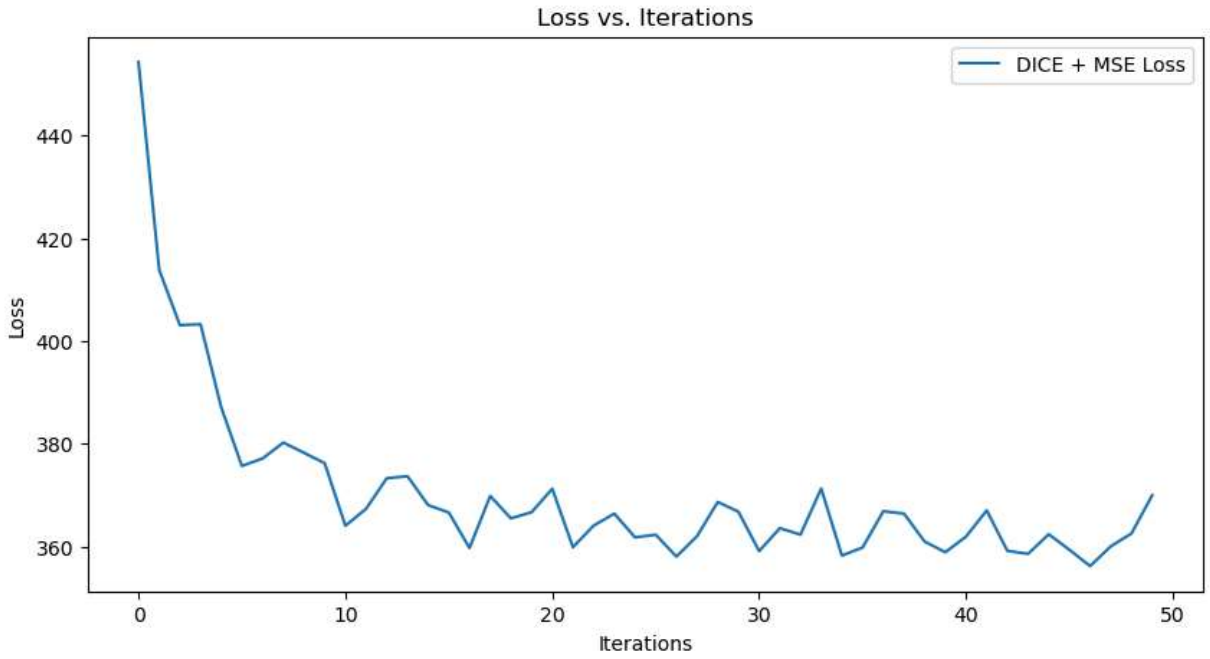
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1102, Mean Pixel Accuracy: 0.1429
```

4.2 Worst set of scale values

In [ ]:

```
# this code is mostly borrowed from DLStudio

dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_MSE_150",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('rectangle','triangle','disk','oval','star'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
```

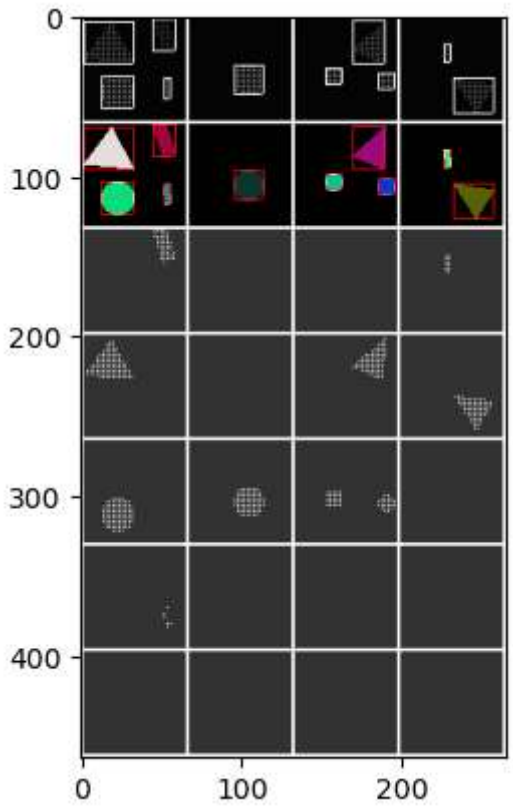
```
                )

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

segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train, dataserver_test)

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

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

# DiceScale is a hyperparameter to scale the dice loss, adjust as needed
DICEMSELoss = segmenter.DICE_MSE_run_code_for_training_for_semantic_segmentation(mo

segmenter.run_code_for_testing_semantic_segmentation(model)

plt.figure(figsize=(10,5))
plt.plot(DICEMSELoss, label='DICE + MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()

segmenter.run_code_for_compute_iou(model)
```

```
Loading training data from torch saved file
```

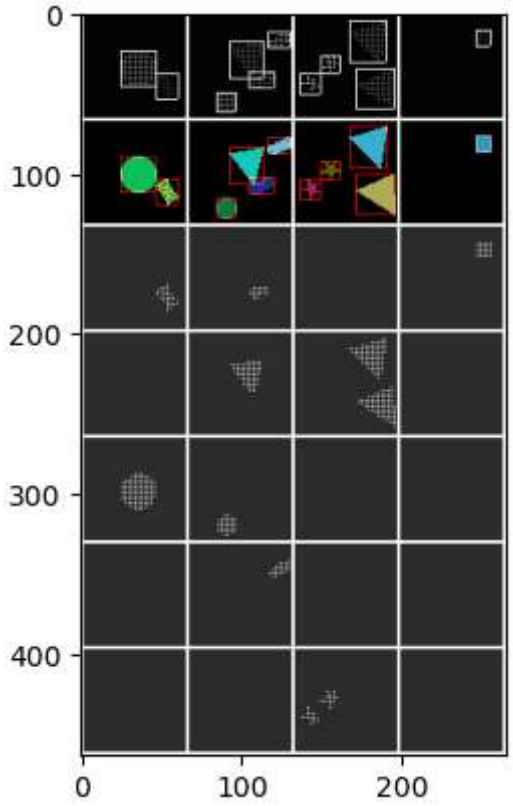
```
/tmp/ipykernel_1334651/2324940881.py:345: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.dataset = torch.load("torch_saved_PurdueShapes5MultiObject-10000_dataset.pt")
/tmp/ipykernel_1334651/2324940881.py:346: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  self.label_map = torch.load("torch_saved_PurdueShapes5MultiObject_label_map.pt")
```

```
The number of learnable parameters in the model: 7688005


[epoch=1/10, iter= 500  elapsed_time=150 secs]   MSE loss: 565.097
[epoch=1/10, iter=1000  elapsed_time=300 secs]   MSE loss: 525.639
[epoch=1/10, iter=1500  elapsed_time=451 secs]   MSE loss: 506.446
[epoch=1/10, iter=2000  elapsed_time=601 secs]   MSE loss: 484.417
[epoch=1/10, iter=2500  elapsed_time=752 secs]   MSE loss: 477.148

[epoch=2/10, iter= 500  elapsed_time=902 secs]   MSE loss: 471.868
[epoch=2/10, iter=1000  elapsed_time=1053 secs]   MSE loss: 471.255
[epoch=2/10, iter=1500  elapsed_time=1203 secs]   MSE loss: 461.399
[epoch=2/10, iter=2000  elapsed_time=1355 secs]   MSE loss: 459.424
[epoch=2/10, iter=2500  elapsed_time=1506 secs]   MSE loss: 460.952

[epoch=3/10, iter= 500  elapsed_time=1657 secs]   MSE loss: 453.169
[epoch=3/10, iter=1000  elapsed_time=1808 secs]   MSE loss: 456.344
[epoch=3/10, iter=1500  elapsed_time=1958 secs]   MSE loss: 460.876
[epoch=3/10, iter=2000  elapsed_time=2109 secs]   MSE loss: 454.547
[epoch=3/10, iter=2500  elapsed_time=2261 secs]   MSE loss: 451.403

[epoch=4/10, iter= 500  elapsed_time=2412 secs]   MSE loss: 450.768
[epoch=4/10, iter=1000  elapsed_time=2562 secs]   MSE loss: 457.803
[epoch=4/10, iter=1500  elapsed_time=2714 secs]   MSE loss: 448.469
[epoch=4/10, iter=2000  elapsed_time=2865 secs]   MSE loss: 447.387
[epoch=4/10, iter=2500  elapsed_time=3016 secs]   MSE loss: 447.378

[epoch=5/10, iter= 500  elapsed_time=3167 secs]   MSE loss: 454.379
[epoch=5/10, iter=1000  elapsed_time=3318 secs]   MSE loss: 443.760
[epoch=5/10, iter=1500  elapsed_time=3469 secs]   MSE loss: 444.038
[epoch=5/10, iter=2000  elapsed_time=3620 secs]   MSE loss: 452.046
[epoch=5/10, iter=2500  elapsed_time=3770 secs]   MSE loss: 448.598

[epoch=6/10, iter= 500  elapsed_time=3921 secs]   MSE loss: 444.826
[epoch=6/10, iter=1000  elapsed_time=4072 secs]   MSE loss: 453.059
[epoch=6/10, iter=1500  elapsed_time=4223 secs]   MSE loss: 448.068
[epoch=6/10, iter=2000  elapsed_time=4374 secs]   MSE loss: 445.869
[epoch=6/10, iter=2500  elapsed_time=4525 secs]   MSE loss: 445.500

[epoch=7/10, iter= 500  elapsed_time=4677 secs]   MSE loss: 445.232
[epoch=7/10, iter=1000  elapsed_time=4828 secs]   MSE loss: 443.800
[epoch=7/10, iter=1500  elapsed_time=4979 secs]   MSE loss: 448.684
[epoch=7/10, iter=2000  elapsed_time=5130 secs]   MSE loss: 446.868
[epoch=7/10, iter=2500  elapsed_time=5281 secs]   MSE loss: 449.351

[epoch=8/10, iter= 500  elapsed_time=5432 secs]   MSE loss: 442.237
[epoch=8/10, iter=1000  elapsed_time=5582 secs]   MSE loss: 445.655
[epoch=8/10, iter=1500  elapsed_time=5733 secs]   MSE loss: 447.608
[epoch=8/10, iter=2000  elapsed_time=5884 secs]   MSE loss: 446.896
[epoch=8/10, iter=2500  elapsed_time=6035 secs]   MSE loss: 449.794

[epoch=9/10, iter= 500  elapsed_time=6186 secs]   MSE loss: 449.107
[epoch=9/10, iter=1000  elapsed_time=6337 secs]   MSE loss: 443.681
[epoch=9/10, iter=1500  elapsed_time=6488 secs]   MSE loss: 441.705
[epoch=9/10, iter=2000  elapsed_time=6638 secs]   MSE loss: 450.449
[epoch=9/10, iter=2500  elapsed_time=6789 secs]   MSE loss: 444.377
```

```
[epoch=10/10, iter= 500  elapsed_time=6940 secs]   MSE loss: 444.848
[epoch=10/10, iter=1000  elapsed_time=7091 secs]   MSE loss: 446.177
[epoch=10/10, iter=1500  elapsed_time=7241 secs]   MSE loss: 437.580
[epoch=10/10, iter=2000  elapsed_time=7392 secs]   MSE loss: 446.604
[epoch=10/10, iter=2500  elapsed_time=7542 secs]   MSE loss: 450.985

Finished Training

Showing output for test batch 1:
/tmp/ipykernel_1334651/2324940881.py:828: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```

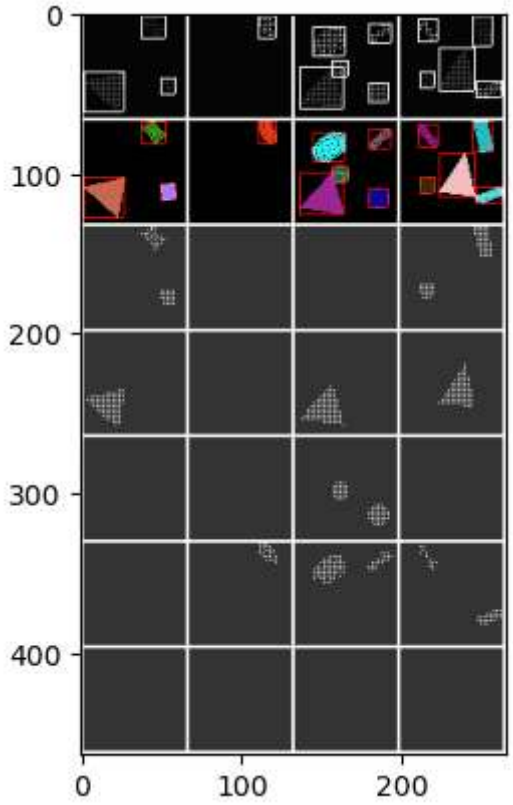



```
Showing output for test batch 51:
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



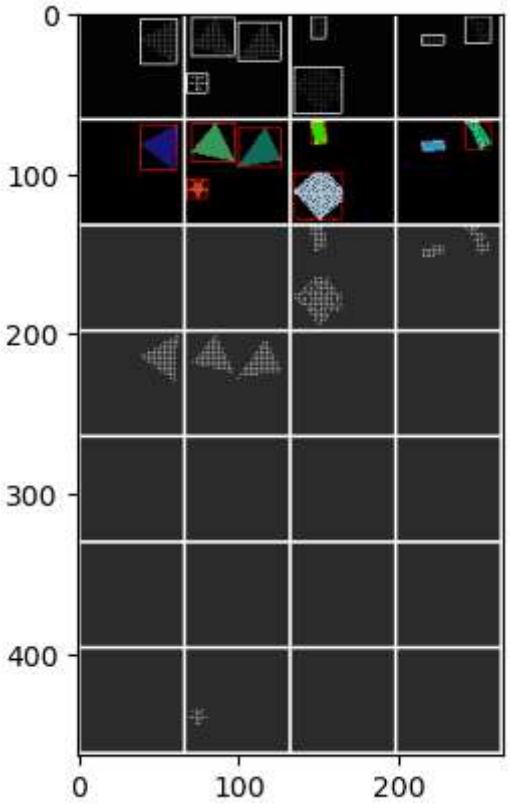
Showing output for test batch 101:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```



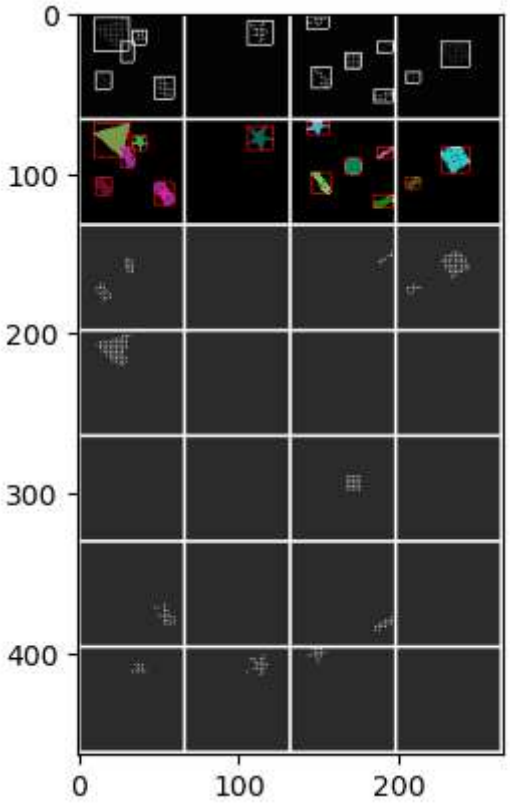
Showing output for test batch 151:

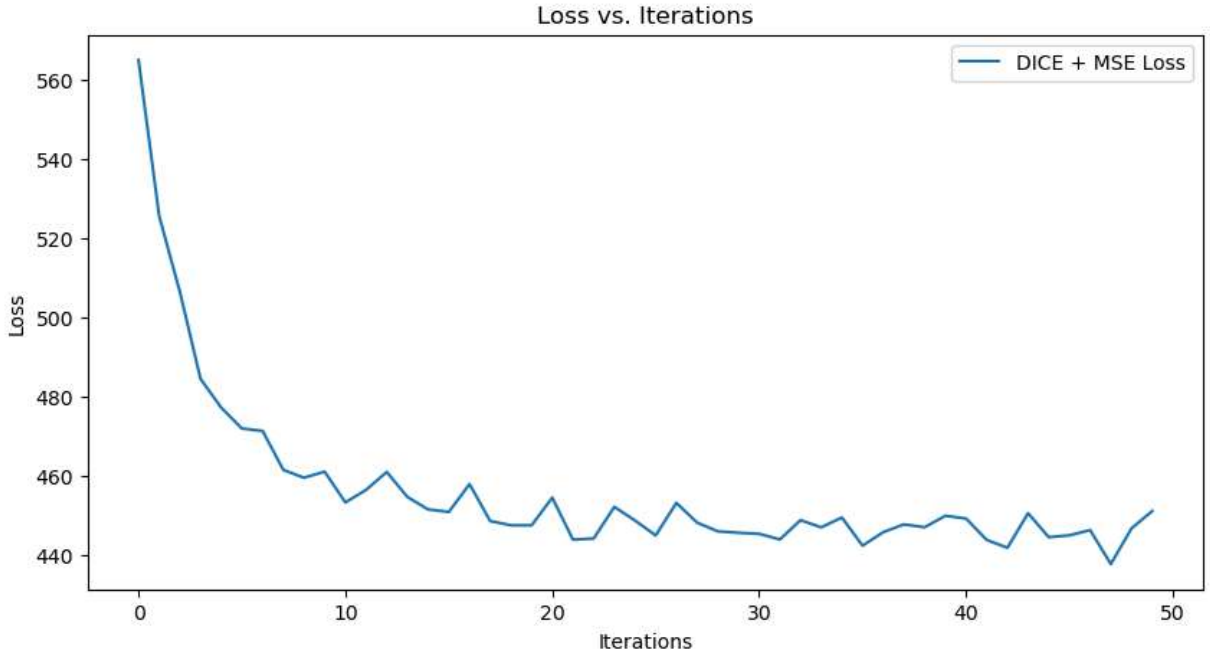
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```




```
Showing output for test batch 201:
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers). Got range [0.0..10.0].
```






```
/tmp/ipykernel_1334651/2324940881.py:908: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
Mean IoU: 0.1075, Mean Pixel Accuracy: 0.1088
```

5 Evaluation and observation

Generate dataset

In [11]:
```
# from PIL import Image
# import os
# import json
# import numpy as np
# from pycocotools.coco import COCO
# from pycocotools import mask as maskUtils

# # Loads an image, resizes it to 256x256, scales its BBox and segmentation, and sa
# # Returns the updated annotations.
# def save_image(img_info, anns, img_dir, output_dir):
#     img_path = os.path.normpath(os.path.join(img_dir, img_info['file_name']))
#     os.makedirs(output_dir, exist_ok=True)

#     if not os.path.exists(img_path):
#         print(f"Skipping {img_path} - File not found.")
#         return
```

```
#     # Load the image and compute scaling factors
#     img = Image.open(img_path)
#     width, height = img.size
#     scale_w = 256 / width
#     scale_h = 256 / height

#     # Resize the image to 256x256 and update its info
#     img = img.resize((256, 256))
#     img_info['width'] = 256
#     img_info['height'] = 256

#     save_path = os.path.join(output_dir, img_info['file_name'])
#     img.save(save_path)

#     new_anns = []
#     for ann in anns:
#         # --- Update bounding box ---
#         x, y, w, h = ann['bbox']
#         x *= scale_w
#         y *= scale_h
#         w *= scale_w
#         h *= scale_h
#         ann['bbox'] = [x, y, w, h]
#         ann['area'] = w * h

#         # --- Update segmentation ---
#         if isinstance(ann['segmentation'], list):
#             # Polygon segmentation: scale each coordinate appropriately.
#             new_segmentation = []
#             for seg in ann['segmentation']:
#                 new_seg = []
#                 for i, coord in enumerate(seg):
#                     if i % 2 == 0:  # x coordinate
#                         new_seg.append(coord * scale_w)
#                     else:           # y coordinate
#                         new_seg.append(coord * scale_h)
#                 new_segmentation.append(new_seg)
#             ann['segmentation'] = new_segmentation
#         elif isinstance(ann['segmentation'], dict):
#             # RLE segmentation: decode, resize mask, then re-encode.
#             mask = maskUtils.decode(ann['segmentation'])
#             mask_img = Image.fromarray(mask.astype('uint8') * 255)
#             mask_img = mask_img.resize((256, 256), Image.NEAREST)
#             new_mask = (np.array(mask_img) > 127).astype(np.uint8)
#             new_rle = maskUtils.encode(np.asfortranarray(new_mask))
#             # Convert counts from bytes to string if needed.
#             if isinstance(new_rle['counts'], bytes):
#                 new_rle['counts'] = new_rle['counts'].decode('utf-8')
#             ann['segmentation'] = new_rle

#         new_anns.append(ann)

#     return new_anns
```

```
# def update_annotation_file(new_images, updated_anns, original_ann_file):
#     # Load the original annotation file to retrieve the categories information
#     with open(original_ann_file, 'r') as f:
#         original_data = json.load(f)

#     # Update the images and annotations fields
#     original_data['images'] = new_images
#     original_data['annotations'] = updated_anns

#     # Write the updated data back to the original file
#     with open(original_ann_file, 'w') as f:
#         json.dump(original_data, f)


# def extract_images(cat_names, img_dir, output_dir, ifVal):
#     cat_ids = coco.getCatIds(catNms=cat_names)
#     img_ids = coco.getImgIds(catIds=cat_ids)
#     min_area = 40000  # 200x200

#     valid_images = []
#     target_category = ["pizza", "cat", "bus"]
#     updated_anns_total = []

#     for img_id in img_ids:
#         img_info = coco.loadImgs(img_id)[0]
#         if ifVal and "COCO_train2014_" in img_info["file_name"]:
#             continue  # Skip training images for validation

#         ann_ids = coco.getAnnIds(imgIds=img_id, iscrowd=False)
#         anns = coco.loadAnns(ann_ids)

#         valid_anns = []
#         for ann in anns:
#             obj_category = coco.loadCats(ann['category_id'])[0]['name']
#             if obj_category in target_category and ann['area'] > min_area:
#                 valid_anns.append(ann)

#         if valid_anns:
#             new_anns = save_image(img_info, valid_anns, img_dir, output_dir)
#             if new_anns:
#                 valid_images.append(img_info)
#                 updated_anns_total.extend(new_anns)

#     return valid_images, updated_anns_total


# # Set COCO dataset paths (update paths as needed)
# ann_file_train = "./../data/annotations(HW8)/instances_train2014.json"
# image_dir_train = "./../data/train2014"
# output_dir_train = "./../data/Multi-instance_images_from_COCO(HW8)/train"
# os.makedirs(output_dir_train, exist_ok=True)

# ann_file_val = "./../data/annotations(HW8)/instances_val2014.json"
# image_dir_val = "./../data/val2014"
# output_dir_val = "./../data/Multi-instance_images_from_COCO(HW8)/val"
# os.makedirs(output_dir_val, exist_ok=True)
```

```
# all_new_images = []
# all_updated_anns = []

# # Process training data
# coco = COCO(ann_file_train)

# valid_images, updated_anns_total = extract_images(["pizza"], img_dir=image_dir_tr
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# valid_images, updated_anns_total = extract_images(["cat"], img_dir=image_dir_trai
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# valid_images, updated_anns_total = extract_images(["bus"], img_dir=image_dir_trai
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# update_annotation_file(all_new_images, all_updated_anns, ann_file_train)

# # Process validation data
# all_new_images = []
# all_updated_anns = []

# print(f"Loading COCO annotation file: {ann_file_val}")
# coco = COCO(ann_file_val)

# valid_images, updated_anns_total = extract_images(["pizza"], img_dir=image_dir_va
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# valid_images, updated_anns_total = extract_images(["cat"], img_dir=image_dir_val,
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# valid_images, updated_anns_total = extract_images(["bus"], img_dir=image_dir_val,
# all_new_images.extend(valid_images)
# all_updated_anns.extend(updated_anns_total)

# update_annotation_file(all_new_images, all_updated_anns, ann_file_val)
```

Dataloader

In [ ]:

```
# data loader for the COCO dataset

import os
import json
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image, ImageDraw
import torchvision.transforms as transforms
from pycocotools import mask as maskUtils
```

```python
class MyDataset(Dataset):
    def __init__(self, ann_file, image_dir, img_transform, mask_transform):
        # Load the annotation JSON file
        with open(ann_file, 'r') as f:
            self.annotation = json.load(f)

        self.image_dir = image_dir
        self.img_transform = img_transform
        self.mask_transform = mask_transform

        # Build an index mapping image id to its annotations
        self.img_id_to_anns = {}
        for ann in self.annotation['annotations']:
            img_id = ann['image_id']
            if img_id not in self.img_id_to_anns:
                self.img_id_to_anns[img_id] = []
            self.img_id_to_anns[img_id].append(ann)

        # List of image info dictionaries
        self.images = self.annotation['images']

        # Mapping from original category IDs to our indices.
        # Here we label: pizza (59) -> 1, cat (17) -> 2, bus (6) -> 3
        # Background is 0.
        self.catid_to_idx = {59: 1, 17: 2, 6: 3}


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


    def __getitem__(self, idx):
        # Get image info and path
        img_info = self.images[idx]
        img_path = os.path.join(self.image_dir, img_info['file_name'])

        # Open image and get original size
        image = Image.open(img_path).convert('RGB')
        width, height = image.size

        # Create an empty mask (background=0)
        mask = np.zeros((height, width), dtype=np.uint8)

        # Check if there are any annotations for this image
        if img_info['id'] in self.img_id_to_anns:
            for ann in self.img_id_to_anns[img_info['id']]:
                cat_id = ann['category_id']
                if cat_id in self.catid_to_idx:
                    label = self.catid_to_idx[cat_id]
                    segm = ann['segmentation']


                    # If segmentation is provided as a polygon format (list of list
                    if isinstance(segm, list):
                        # Create a blank mask for the polygon and draw the segmenta
```

```
                        poly_mask = Image.new('L', (width, height), 0)
                        for polygon in segm:
                            # polygon is a list of x,y coordinates
                            ImageDraw.Draw(poly_mask).polygon(polygon, outline=1, f
                        poly_mask = np.array(poly_mask, dtype=bool)

                    # Otherwise, It's in RLE format
                    else:
                        poly_mask = maskUtils.decode(segm).astype(bool)



                    # Assign label to mask where the object is present
                    mask[poly_mask] = label  # Set all pixels in the mask to the la

        # Apply the image transform
        image_tensor = self.img_transform(image)

        # Convert the mask to a PIL image and apply mask transforms.
        mask_pil = Image.fromarray(mask) # Convert to PIL image
        mask_resized = self.mask_transform(mask_pil)

        # Convert single-channel mask to one-hot format
        mask_tensor = mask_resized.long()
        num_classes = 4  # Background (0) + 3 classes
        one_hot_mask = torch.zeros((num_classes, mask_tensor.shape[1], mask_tensor.
        for i in range(num_classes):
            one_hot_mask[i] = (mask_tensor == i).float()

        # Dummy bounding box tensor (not used in this homework)
        bbox_tensor = torch.zeros(5, 4)

        sample = {'image': image_tensor, 'mask_tensor': one_hot_mask, 'bbox_tensor'
        return sample



img_transform = transforms.Compose([
    transforms.Resize((64, 64)),

    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])



# For masks, we only resize using nearest neighbor and convert to tensor.
mask_transform = transforms.Compose([
    transforms.Resize((64, 64), interpolation=Image.NEAREST),
    transforms.PILToTensor()
])


ann_file_train = "./../data/annotations(HW8)/instances_train2014.json"
image_dir_train = "./../data/Multi-instance_images_from_COCO(HW8)/train"
ann_file_val = "./../data/annotations(HW8)/instances_val2014.json"
```

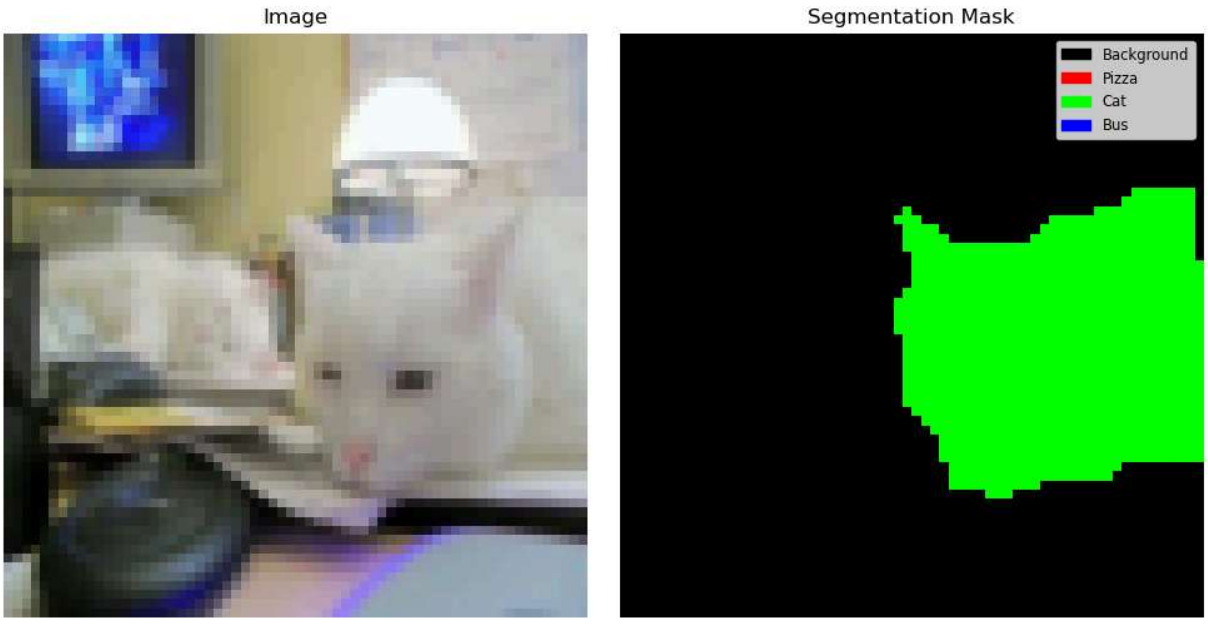
```
image_dir_val = "./../data/Multi-instance_images_from_COCO(HW8)/val"

train_dataset = MyDataset(ann_file=ann_file_train, image_dir=image_dir_train, img_t
val_dataset = MyDataset(ann_file=ann_file_val, image_dir=image_dir_val,img_transfor

# Create a DataLoader to iterate through the dataset
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=4, shuffle=True)

# Test a sample
sample = train_dataset[0]
image_tensor = sample['image']
mask_tensor = sample['mask_tensor']
print("Image shape:", image_tensor.shape)
print("Mask shape:", mask_tensor.shape)
```

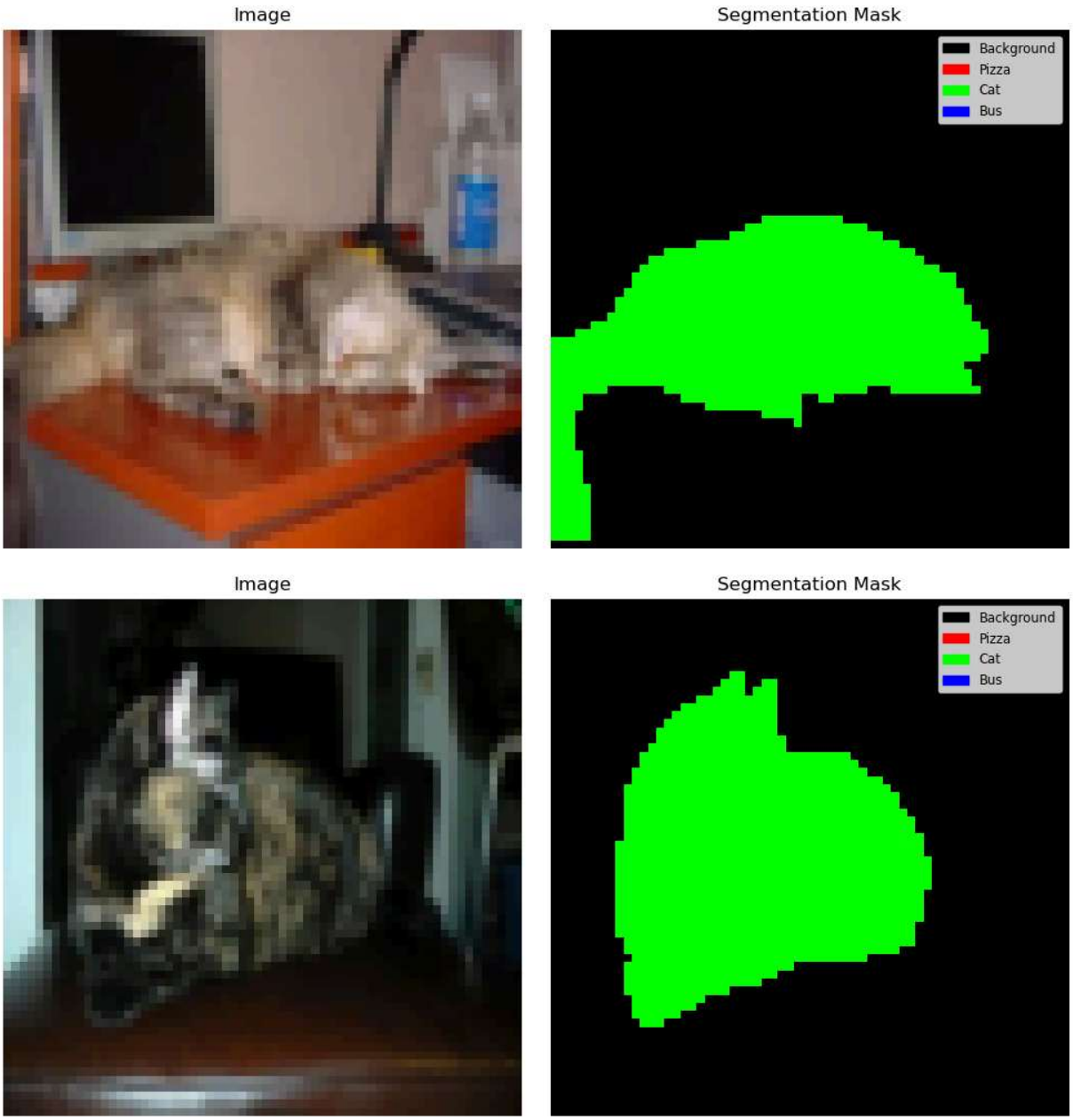
```
Image shape: torch.Size([3, 64, 64])
Mask shape: torch.Size([4, 64, 64])
```

In [ ]:

```
import matplotlib.pyplot as plt
import torch

# Get one batch of data
batch = next(iter(train_loader))
images = batch['image']   # shape: [B, 3, H, W]
masks = batch['mask_tensor']  # shape: [B, num_classes, H, W]

# Visualization with simple clear colors
class_colors = [
    [0, 0, 0],      # Background (black)
    [255, 0, 0],    # Pizza (red)
    [0, 255, 0],    # Cat (green)
    [0, 0, 255]     # Bus (blue)
]
class_names = ['Background', 'Pizza', 'Cat', 'Bus']

def unnormalize(img_tensor):
    img_tensor = img_tensor.clone()
    img_tensor = img_tensor * 0.5 + 0.5
    return img_tensor


num_examples = 3
for i in range(min(num_examples, len(images))):
    img = images[i]
    mask = masks[i]

    # Unnormalize image and convert to numpy
    img_np = unnormalize(img).permute(1, 2, 0).numpy()
    mask_np = mask.argmax(0).numpy()

    # Create a figure with two subplots
    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    # Original image
    axs[0].imshow(img_np)
```

```
    axs[0].set_title("Image")
    axs[0].axis("off")



    # Create a colored segmentation mask
    colored_mask = np.zeros((64, 64, 3), dtype=np.uint8)
    for j in range(4):  # 4 classes (including background)
        mask_j = mask_np == j # mask_j is a boolean array with True where mask_np =

        for channel in range(3):
            colored_mask[:,:,channel][mask_j] = class_colors[j][channel]

    # Display the colored mask on the second subplot
    axs[1].imshow(colored_mask)
    axs[1].set_title("Segmentation Mask")
    axs[1].axis("off")

    # Add a custom legend
    import matplotlib.patches as mpatches
    patches = []
    for j in range(4):
        color_rgb = [x/255 for x in class_colors[j]]  # Normalize to [0,1]
        patch = mpatches.Patch(color=color_rgb, label=class_names[j])
        patches.append(patch)

    axs[1].legend(handles=patches, loc='upper right', fontsize='small')

    plt.tight_layout()
    plt.show()
```





5.1 3 images MSE

In [14]:
```
# this code is mostly borrowed from DLStudio

import torch.optim as optim
import torch
import torch.nn as nn
import copy
import time
import gzip
import pickle
import numpy as np
import os
import sys
import torchvision
import matplotlib.pyplot as plt
```

```
import torch.nn.functional as F


class DLStudio(object):
    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError(
                   '''DLStudio constructor can only be called with keyword argument
                      the following keywords: epochs, learning_rate, batch_size, mo
                      convo_layers_config, image_size, dataroot, path_saved_model,
                      image_size, convo_layers_config, fc_layers_config, debug_trai
                      debug_test''')
        learning_rate = epochs = batch_size = convo_layers_config = momentum = None
        image_size = fc_layers_config = dataroot =  path_saved_model = classes = us
        debug_train  = debug_test = None
        if 'dataroot' in kwargs                       :   dataroot = kwargs.pop('dat
        if 'learning_rate' in kwargs                  :   learning_rate = kwargs.pop
        if 'momentum' in kwargs                       :   momentum = kwargs.pop('mom
        if 'epochs' in kwargs                         :   epochs = kwargs.pop('epoch
        if 'batch_size' in kwargs                     :   batch_size = kwargs.pop('b
        if 'convo_layers_config' in kwargs            :   convo_layers_config = kwar
        if 'image_size' in kwargs                     :   image_size = kwargs.pop('i
        if 'fc_layers_config' in kwargs               :   fc_layers_config = kwargs.
        if 'path_saved_model' in kwargs               :   path_saved_model = kwargs.
        if 'classes' in kwargs                        :   classes = kwargs.pop('clas
        if 'use_gpu' in kwargs                        :   use_gpu = kwargs.pop('use_
        if 'debug_train' in kwargs                    :   debug_train = kwargs.pop('
        if 'debug_test' in kwargs                     :   debug_test = kwargs.pop('d
        if len(kwargs) != 0: raise ValueError('''You have provided unrecognizable k
        if dataroot:
            self.dataroot = dataroot
        if convo_layers_config:
            self.convo_layers_config = convo_layers_config
        if image_size:
            self.image_size = image_size
        if fc_layers_config:
            self.fc_layers_config = fc_layers_config
            if fc_layers_config[0] != -1:
                raise Exception("""\n\n\nYour 'fc_layers_config' construction optio
                                """The first element of the list of nodes in the fc
                                """because the input to fc will be set automaticall
                                """the final activation volume of the convolutional
        if  path_saved_model:
            self.path_saved_model = path_saved_model
        if classes:
            self.class_labels = classes
        if learning_rate:
            self.learning_rate = learning_rate
        else:
            self.learning_rate = 1e-6
        if momentum:
            self.momentum = momentum
        if epochs:
            self.epochs = epochs
        if batch_size:
            self.batch_size = batch_size
```

```
        if use_gpu is not None:
            self.use_gpu = use_gpu
            if use_gpu is True:
                if torch.cuda.is_available():
                    self.device = torch.device("cuda:0")
                else:
                    self.device = torch.device("cpu")
        else:
            self.device = torch.device("cpu")
        if debug_train:
            self.debug_train = debug_train
        else:
            self.debug_train = 0
        if debug_test:
            self.debug_test = debug_test
        else:
            self.debug_test = 0
        self.debug_config = 0


    def imshow(self, img):
        '''
        called by display_tensor_as_image() for displaying the image
        '''
        img = img / 2 + 0.5     # unnormalize
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()


    def dice_loss(preds: torch.Tensor, ground_truth: torch.Tensor):
        # prevents division by zero
        epsilon=1e-6

        numerator = torch.sum(preds * ground_truth, dim=(1, 2, 3))
        denominator = torch.sum(preds ** 2, dim=(1, 2, 3)) + torch.sum(ground_truth

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

        # Step 3: Compute dice_loss = 1 - dice_coefficient
        dice_loss = 1 - dice_coefficient.mean()

        return dice_loss

    def display_tensor_as_image(self, tensor, title=""):

        tensor_range = (torch.min(tensor).item(), torch.max(tensor).item())
        if tensor_range == (-1.0,1.0):
            ##  The tensors must be between 0.0 and 1.0 for the display:
            print("\n\n\nimage un-normalization called")
            tensor = tensor/2.0 + 0.5     # unnormalize
        plt.figure(title)
        ###  The call to plt.imshow() shown below needs a numpy array. We must also
        ###  transpose the array so that the number of channels (the same thing as
        ###  number of color planes) is in the last element.  For a tensor, it woul
```

```
        ###  the first element.
        if tensor.shape[0] == 3 and len(tensor.shape) == 3:
#           plt.imshow( tensor.numpy().transpose(1,2,0) )
            plt.imshow( tensor.numpy().transpose(1,2,0) )
        ###  If the grayscale image was produced by calling torchvision.transform's
        ###  ".ToPILImage()", and the result converted to a tensor, the tensor shap
        ###  again have three elements in it, however the first element that stands
        ###  the number of channels will now be 1
        elif tensor.shape[0] == 1 and len(tensor.shape) == 3:
            tensor = tensor[0,:,:]
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        ###  For any one color channel extracted from the tensor representation of
        ###  image, the shape of the tensor will be (W,H):
        elif len(tensor.shape) == 2:
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        else:
            sys.exit("\n\n\nfrom 'display_tensor_as_image()': tensor for image is i
        plt.show()




    class SemanticSegmentation(nn.Module):
        def __init__(self, dl_studio, max_num_objects, dataserver_train=None, datas
            super(DLStudio.SemanticSegmentation, self).__init__()
            self.dl_studio = dl_studio
            self.max_num_objects = max_num_objects
            self.dataserver_train = dataserver_train
            self.dataserver_test = dataserver_test


        class PurdueShapes5MultiObjectDataset(torch.utils.data.Dataset):
            def __init__(self, dl_studio, segmenter, train_or_test, dataset_file):
                super(DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset
                max_num_objects = segmenter.max_num_objects
                if train_or_test == 'train' and dataset_file == "PurdueShapes5Multi
                    if os.path.exists("torch_saved_PurdueShapes5MultiObject-10000_d
                              os.path.exists("torch_saved_PurdueShapes5MultiObject_
                        print("\nLoading training data from torch saved file")
                        self.dataset = torch.load("torch_saved_PurdueShapes5MultiOb
                        self.label_map = torch.load("torch_saved_PurdueShapes5Multi
                        self.num_shapes = len(self.label_map)
                        self.image_size = dl_studio.image_size
                    else:
                        print("""\n\n\nLooks like this is the first time you will b
                              """the dataset for this script. First time loading co
                              """a few minutes.  Any subsequent attempts will only
                              """a few seconds.\n\n\n""")
                        root_dir = dl_studio.dataroot
                        f = gzip.open(root_dir + dataset_file, 'rb')
                        dataset = f.read()
                        self.dataset, self.label_map = pickle.loads(dataset, encodi
                        torch.save(self.dataset, "torch_saved_PurdueShapes5MultiObj
                        torch.save(self.label_map, "torch_saved_PurdueShapes5MultiO
                        # reverse the key-value pairs in the label dictionary:
                        self.class_labels = dict(map(reversed, self.label_map.items
```

```
                    self.num_shapes = len(self.class_labels)
                    self.image_size = dl_studio.image_size
            else:
                root_dir = dl_studio.dataroot
                f = gzip.open(root_dir + dataset_file, 'rb')
                dataset = f.read()
                if sys.version_info[0] == 3:
                    self.dataset, self.label_map = pickle.loads(dataset, encodi
                else:
                    self.dataset, self.label_map = pickle.loads(dataset)
                # reverse the key-value pairs in the label dictionary:
                self.class_labels = dict(map(reversed, self.label_map.items()))
                self.num_shapes = len(self.class_labels)
                self.image_size = dl_studio.image_size

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

        def __getitem__(self, idx):
            image_size = self.image_size
            r = np.array( self.dataset[idx][0] )
            g = np.array( self.dataset[idx][1] )
            b = np.array( self.dataset[idx][2] )
            R,G,B = r.reshape(image_size[0],image_size[1]), g.reshape(image_siz
            im_tensor = torch.zeros(3,image_size[0],image_size[1], dtype=torch.
            im_tensor[0,:,:] = torch.from_numpy(R)
            im_tensor[1,:,:] = torch.from_numpy(G)
            im_tensor[2,:,:] = torch.from_numpy(B)
            mask_array = np.array(self.dataset[idx][3])
            max_num_objects = len( mask_array[0] )
            mask_tensor = torch.from_numpy(mask_array)
            mask_val_to_bbox_map =  self.dataset[idx][4]
            max_bboxes_per_entry_in_map = max([ len(mask_val_to_bbox_map[key])
            ##  The first arg 5 is for the number of bboxes we are going to nee
            ##  shapes are exactly the same, you are going to need five differe
            ##  The second arg is the index reserved for each shape in a single
            bbox_tensor = torch.zeros(max_num_objects,self.num_shapes,4, dtype=
            for bbox_idx in range(max_bboxes_per_entry_in_map):
                for key in mask_val_to_bbox_map:
                    if len(mask_val_to_bbox_map[key]) == 1:
                        if bbox_idx == 0:
                            bbox_tensor[bbox_idx,key,:] = torch.from_numpy(np.a
                    elif len(mask_val_to_bbox_map[key]) > 1 and bbox_idx < len(
                        bbox_tensor[bbox_idx,key,:] = torch.from_numpy(np.array
            sample = {'image'         : im_tensor,
                      'mask_tensor'   : mask_tensor,
                      'bbox_tensor'   : bbox_tensor }
            return sample

    def load_PurdueShapes5MultiObject_dataset(self, dataserver_train, dataserve
        self.train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                    batch_size=self.dl_studio.batch_size,shuffle=True, num_work
        self.test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                          batch_size=self.dl_studio.batch_size,shuffle=False,
```

```
        class SkipBlockDN(nn.Module):

            def __init__(self, in_ch, out_ch, downsample=False, skip_connections=Tr
                super(DLStudio.SemanticSegmentation.SkipBlockDN, self).__init__()
                self.downsample = downsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch

                # UP is using ConvTranspose2d, DN is using Conv2d
                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:
                    # UP is using ConvTranspose2d, DN is using Conv2d
                    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, identity), dim=1)
                return out


        class SkipBlockUP(nn.Module):
            def __init__(self, in_ch, out_ch, upsample=False, skip_connections=True
                super(DLStudio.SemanticSegmentation.SkipBlockUP, self).__init__()
                self.upsample = upsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch

                # DN is using Conv2d, UP is using ConvTranspose2d
                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:
                    # DN is using Conv2d, UP is using ConvTranspose2d
                    self.upsampler = nn.ConvTranspose2d(in_ch, out_ch, 1, stride=2,
            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 ASPP(nn.Module):
            def __init__(self, in_ch, out_ch):
                super(DLStudio.SemanticSegmentation.ASPP, self).__init__()
                self.conv1 = nn.Conv2d(in_ch, out_ch, 1)

                # padding needs to be eqaul to dilation when kernel size is 3. So t
                # same as input size.
                # (formula: out_size = (in_size + 2*padding - dilation*(kernel_size
                self.conv2 = nn.Conv2d(in_ch, out_ch, 3, padding=6, dilation=6)
                self.conv3 = nn.Conv2d(in_ch, out_ch, 3, padding=12, dilation=12)
                self.conv4 = nn.Conv2d(in_ch, out_ch, 3, padding=18, dilation=18)
                self.conv5 = nn.Conv2d(in_ch, out_ch, 3, padding=24, dilation=24)

                # final conv for concatenation
                self.conv6 = nn.Conv2d(out_ch*5, out_ch, 1)
            def forward(self, x):
                out1 = self.conv1(x)
                out2 = self.conv2(x)
                out3 = self.conv3(x)
                out4 = self.conv4(x)
                out5 = self.conv5(x)
                out = torch.cat([out1, out2, out3, out4, out5], dim=1)
                out = self.conv6(out)
                return out


        class mUNet(nn.Module):
            def __init__(self, skip_connections=True, depth=16):
                super(DLStudio.SemanticSegmentation.mUNet, self).__init__()
                self.depth = depth // 2
                self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
                ##  For the DN(down) 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.SemanticSegmentation.SkipBloc
```

```
            self.skip64dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 64,
            self.skip64to128DN = DLStudio.SemanticSegmentation.SkipBlockDN(64,
            self.skip128DN_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128DN_arr.append(DLStudio.SemanticSegmentation.SkipBlo
            self.skip128dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(128,12



            # add ASPP block before going up
            self.aspp = DLStudio.SemanticSegmentation.ASPP(128, 128)



            ##  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.SemanticSegmentation.SkipBloc
            self.skip64usUP = DLStudio.SemanticSegmentation.SkipBlockUP(64, 64,
            self.skip128to64UP = DLStudio.SemanticSegmentation.SkipBlockUP(128,
            self.skip128UP_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128UP_arr.append(DLStudio.SemanticSegmentation.SkipBlo
            self.skip128usUP = DLStudio.SemanticSegmentation.SkipBlockUP(128,12



            #  change this into 4 classes
            # self.conv_out = nn.ConvTranspose2d(64, 5, 3, stride=2,dilation=2,
            self.conv_out = nn.ConvTranspose2d(64, 4, 3, stride=2,dilation=2,ou




        def forward(self, x):
            ##  Going down to the bottom of the U:
            x = nn.MaxPool2d(2,2)(nn.functional.relu(self.conv_in(x)))
            for i,skip64 in enumerate(self.skip64DN_arr[:self.depth//4]):
                x = skip64(x)

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

            x = self.bn2DN(x)
            num_channels_to_save3 = x.shape[1] // 2
```

```
            save_for_upside_3 = x[:,:num_channels_to_save3,:,:].clone()
            for i,skip128 in enumerate(self.skip128DN_arr[self.depth//4:]):
                x = skip128(x)
            x = self.skip128dsDN(x)



            # add ASPP block beforing going up
            x = self.aspp(x)



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


    class SegmentationLoss(nn.Module):
        def __init__(self, batch_size):
            super(DLStudio.SemanticSegmentation.SegmentationLoss, self).__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_idx,:,:]
                    element_wise = (outputh - mask)**2
                    mask_based_loss[0,mask_layer_idx] = torch.mean(element_wise
                composite_loss[0,idx] = torch.sum(mask_based_loss)
            return torch.sum(composite_loss) / self.batch_size


    def MSE_run_code_for_training_for_semantic_segmentation(self, net):
        filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)
```

```
            # using MSE loss
            criterion1 = nn.MSELoss()

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

            # record the loss values
            total_loss = []

            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['image'],data['mask_ten
                    im_tensor   = im_tensor.to(self.dl_studio.device)
                    mask_tensor = mask_tensor.type(torch.FloatTensor)
                    mask_tensor = mask_tensor.to(self.dl_studio.device)
                    bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                    optimizer.zero_grad()
                    output = net(im_tensor)
                    segmentation_loss = criterion1(output, mask_tensor)
                    segmentation_loss.backward()
                    optimizer.step()
                    running_loss_segmentation += segmentation_loss.item()
                    if i%500==499:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time
                        avg_loss_segmentation = running_loss_segmentation / float(5
                        print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   los
                        total_loss.append(avg_loss_segmentation)
                        FILE1.write("%.3f\n" % avg_loss_segmentation)
                        FILE1.flush()
                        running_loss_segmentation = 0.0
            print("\nFinished Training\n")
            self.save_model(net)

            return total_loss


        def DICE_run_code_for_training_for_semantic_segmentation(self, net):
            filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
            FILE1 = open(filename_for_out1, 'w')
            net = copy.deepcopy(net)
            net = net.to(self.dl_studio.device)

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

            # record the loss values
            total_loss = []

            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['image'],data['mask_ten
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)

                # using DICE loss
                segmentation_loss = DLStudio.dice_loss(output, mask_tensor)

                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_loss.item()
                if i%500==499:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segmentation / float(5
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   MSE
                    total_loss.append(avg_loss_segmentation)
                    FILE1.write("%.3f\n" % avg_loss_segmentation)
                    FILE1.flush()
                    running_loss_segmentation = 0.0
        print("\nFinished Training\n")
        self.save_model(net)
        return total_loss


    def DICE_MSE_run_code_for_training_for_semantic_segmentation(self, net, Dic
        filename_for_out1 = "performance_numbers_" + str(self.dl_studio.epochs)
        FILE1 = open(filename_for_out1, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

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

        # record the loss values
        total_loss = []

        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['image'],data['mask_ten
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)
```

```
                # using DICE+MSE loss
                dice_scale = DiceScale
                segmentation_loss = dice_scale * DLStudio.dice_loss(output, mas
                segmentation_loss += nn.MSELoss()(output, mask_tensor)



                segmentation_loss.backward()
                optimizer.step()
                running_loss_segmentation += segmentation_loss.item()
                if i%500==499:
                    current_time = time.perf_counter()
                    elapsed_time = current_time - start_time
                    avg_loss_segmentation = running_loss_segmentation / float(5
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   MSE
                    total_loss.append(avg_loss_segmentation)
                    FILE1.write("%.3f\n" % avg_loss_segmentation)
                    FILE1.flush()
                    running_loss_segmentation = 0.0
        print("\nFinished Training\n")
        self.save_model(net)
        return total_loss


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


    def run_code_for_visualize_semantic_segmentation(self, net):

        net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
        batch_size = self.dl_studio.batch_size
        image_size = self.dl_studio.image_size

        # Get one batch of data
        batch = next(iter(train_loader))
        images = batch['image']   # shape: [B, 3, H, W]
        masks = batch['mask_tensor']  # shape: [B, num_classes, H, W]

        # Visualization with simple clear colors
        class_colors = [
            [0, 0, 0],      # Background (black)
            [255, 0, 0],    # Pizza (red)
            [0, 255, 0],    # Cat (green)
            [0, 0, 255]     # Bus (blue)
        ]
        class_names = ['Background', 'Pizza', 'Cat', 'Bus']

        num_examples = 3

        with torch.no_grad():
            for B, data in enumerate(self.test_dataloader):
                images, masks= data['image'], data['mask_tensor']
```

```
                for i in range(min(num_examples, len(images))):
                    img = images[i]
                    mask = masks[i]


                    img_unnormalized = img * 0.5 + 0.5
                    # Unnormalize image and convert to numpy
                    img_np = img_unnormalized.permute(1, 2, 0).numpy()
                    mask_np = mask.argmax(0).numpy()

                    # Create a figure with two subplots
                    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

                    # Original image
                    axs[0].imshow(img_np)
                    axs[0].set_title("Image")
                    axs[0].axis("off")

                    # Create a properly colored segmentation mask
                    colored_mask = np.zeros((64, 64, 3), dtype=np.uint8)
                    for j in range(4):  # 4 classes (including background)
                        mask_j = mask_np == j
                        for c in range(3):  # RGB channels
                            colored_mask[:,:,c][mask_j] = class_colors[j][c]

                    # Display the colored mask
                    axs[1].imshow(colored_mask)
                    axs[1].set_title("Segmentation Mask")
                    axs[1].axis("off")

                    # Add a custom legend
                    import matplotlib.patches as mpatches
                    patches = []
                    for j in range(4):
                        color_rgb = [x/255 for x in class_colors[j]]  # Normali
                        patch = mpatches.Patch(color=color_rgb, label=class_nam
                        patches.append(patch)

                    axs[1].legend(handles=patches, loc='upper right', fontsize=

                    plt.tight_layout()
                    plt.show()

                break

    def run_code_for_testing_semantic_segmentation(self, net):
        net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
        batch_size = self.dl_studio.batch_size
        image_size = self.dl_studio.image_size

        count = 0

        with torch.no_grad():
            for i, data in enumerate(self.test_dataloader):
                # We no longer need the bbox tensor for visualization.
```

```
                im_tensor, mask_tensor, _ = data['image'], data['mask_tensor'],
                if i % 50 == 0:
                    count += 1
                    if count > 3:
                        break
                    print("Showing output for test batch %d:" % (i + 1))
                    outputs = net(im_tensor)

                    # Create a single-channel output image by taking the max ov
                    output_bw_tensor = torch.zeros(batch_size, 1, image_size[0]
                    for image_idx in range(batch_size):
                        for m in range(image_size[0]):
                            for n in range(image_size[1]):
                                output_bw_tensor[image_idx, 0, m, n] = torch.ma

                    # Prepare a display tensor:
                    # - First set of rows: the output_bw_tensor (replicated to
                    # - Next set of rows: the original image tensor
                    display_tensor = torch.zeros(2 * batch_size, 3, image_size[
                    display_tensor[:batch_size, :, :, :] = output_bw_tensor.rep
                    display_tensor[batch_size:2*batch_size, :, :, :] = im_tenso

                    # Additionally, visualize each mask layer separately below
                    mask_layers = []
                    # Use the actual number of channels from the network output
                    for mask_layer_idx in range(outputs.shape[1]):
                        # Extract the mask layer, then replicate it to 3 channe
                        mask_layer = outputs[:, mask_layer_idx, :, :].unsqueeze
                        mask_layers.append(mask_layer)

                    # Concatenate the main display_tensor with the mask layers
                    display_tensor = torch.cat([display_tensor] + mask_layers,

                    grid = torchvision.utils.make_grid(display_tensor, nrow=bat
                    self.dl_studio.display_tensor_as_image(grid)




    def compute_iou(self, pred_mask, true_mask):
        intersection = (pred_mask & true_mask).sum()
        union = (pred_mask | true_mask).sum()
        return intersection / union if union > 0 else 0  # Avoid division by ze

    def run_code_for_compute_iou(self, net):
        net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
        net.eval()  # Set model to evaluation mode
        total_iou = 0
        total_accuracy = 0
        num_samples = 0

        with torch.no_grad():
            for i, data in enumerate(self.test_dataloader):
                im_tensor, mask_tensor = data['image'], data['mask_tensor']
                outputs = net(im_tensor)
```

```
                    # Convert outputs to binary masks. (B,C,H,W) -> (B,H,W).
                    # Each pixel is assigned the class index with the highest predi
                    pred_masks = outputs.argmax(dim=1)
                    true_masks = mask_tensor.argmax(dim=1)

                    # Compute IoU for each image in batch
                    for j in range(pred_masks.shape[0]):
                        iou = self.compute_iou(pred_masks[j] > 0.5, true_masks[j] >
                        total_iou += iou

                        # get same pixel for accuracy
                        correct_pixels = (pred_masks[j] == true_masks[j]).sum()
                        total_pixels = true_masks[j].numel()
                        total_accuracy += correct_pixels / total_pixels

                        num_samples += 1

            mean_iou = total_iou / num_samples
            mean_accuracy = total_accuracy / num_samples
            print(f"Mean IoU: {mean_iou:.4f}, Mean Pixel Accuracy: {mean_accuracy:.
```

In [15]:

```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_MSE_COCO",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('pizza', 'cat', 'bus'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )



segmenter.train_dataloader = train_loader
segmenter.test_dataloader  = val_loader


model = segmenter.mUNet(skip_connections=True, depth=16)

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

MSELoss = segmenter.MSE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_visualize_semantic_segmentation(model)

plt.figure(figsize=(10,5))
```

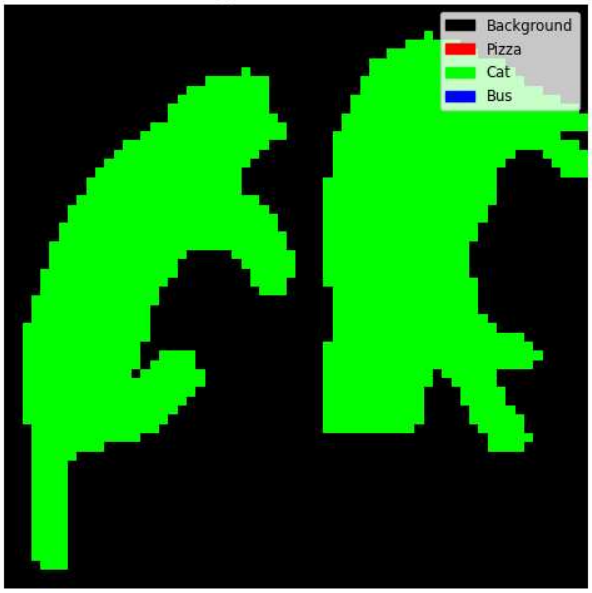
```
plt.plot(MSELoss, label='MSE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()
```

```
The number of learnable parameters in the model: 7687428


[epoch=1/10, iter= 500  elapsed_time= 42 secs]   loss: 0.469

[epoch=2/10, iter= 500  elapsed_time= 96 secs]   loss: 0.213

[epoch=3/10, iter= 500  elapsed_time=132 secs]   loss: 0.177

[epoch=4/10, iter= 500  elapsed_time=167 secs]   loss: 0.161

[epoch=5/10, iter= 500  elapsed_time=201 secs]   loss: 0.153

[epoch=6/10, iter= 500  elapsed_time=236 secs]   loss: 0.148

[epoch=7/10, iter= 500  elapsed_time=268 secs]   loss: 0.145

[epoch=8/10, iter= 500  elapsed_time=302 secs]   loss: 0.143

[epoch=9/10, iter= 500  elapsed_time=338 secs]   loss: 0.142

[epoch=10/10, iter= 500  elapsed_time=371 secs]   loss: 0.142

Finished Training
```

```
/tmp/ipykernel_1334651/406671668.py:608: FutureWarning: You are using `torch.load` w
ith `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

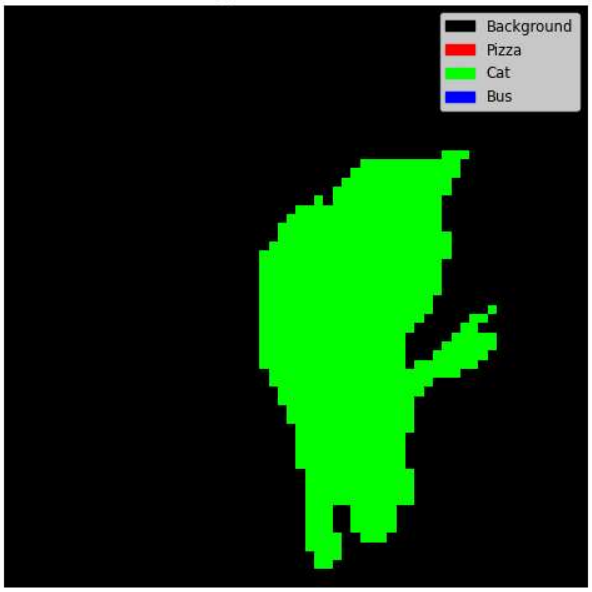
Image

Segmentation Mask



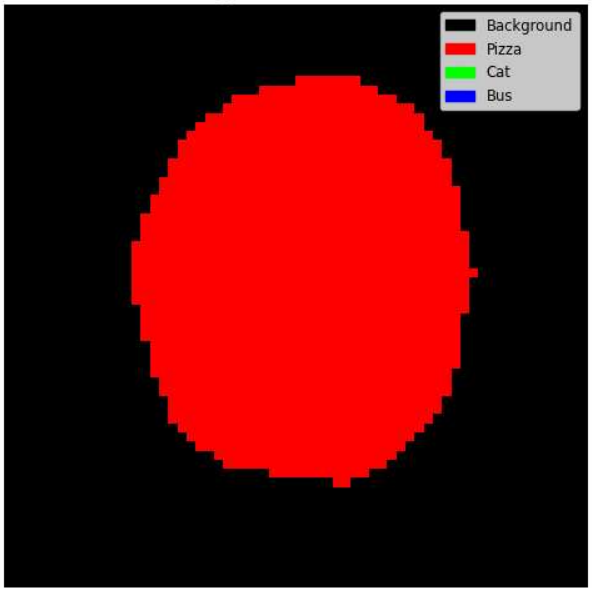
Image

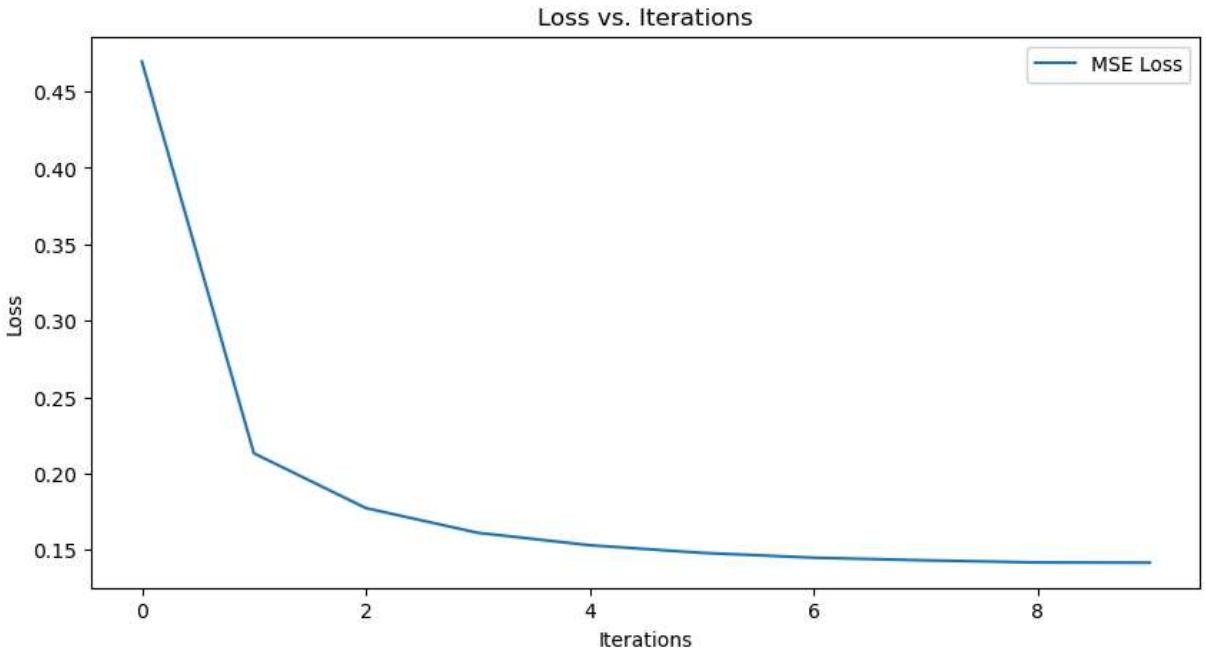
Segmentation Mask



Image

Segmentation Mask





5.2 3 images Dice

In [16]:

```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_COCO",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('pizza', 'cat', 'bus'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )



segmenter.train_dataloader = train_loader
segmenter.test_dataloader  = val_loader


model = segmenter.mUNet(skip_connections=True, depth=16)

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

DICELoss = segmenter.DICE_run_code_for_training_for_semantic_segmentation(model)

segmenter.run_code_for_visualize_semantic_segmentation(model)
```

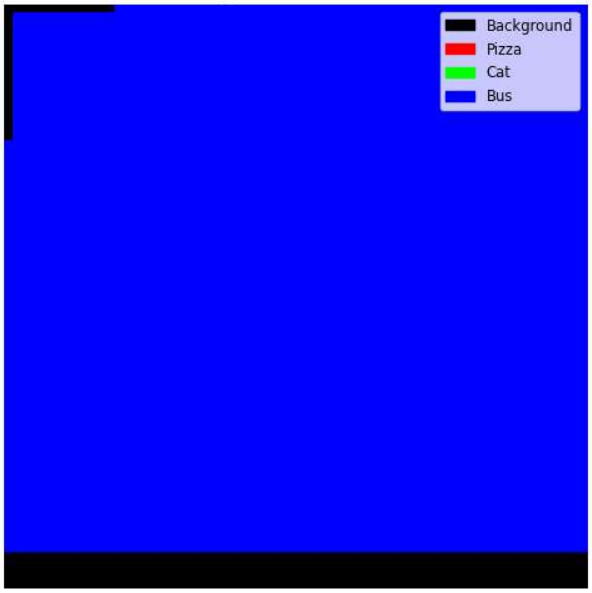
```
plt.figure(figsize=(10,5))
plt.plot(MSELoss, label='DICE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()
```

```
The number of learnable parameters in the model: 7687428


[epoch=1/10, iter= 500  elapsed_time= 16 secs]   MSE loss: 0.925

[epoch=2/10, iter= 500  elapsed_time= 50 secs]   MSE loss: 0.687

[epoch=3/10, iter= 500  elapsed_time= 85 secs]   MSE loss: 0.479

[epoch=4/10, iter= 500  elapsed_time=119 secs]   MSE loss: 0.378

[epoch=5/10, iter= 500  elapsed_time=155 secs]   MSE loss: 0.347

[epoch=6/10, iter= 500  elapsed_time=190 secs]   MSE loss: 0.336

[epoch=7/10, iter= 500  elapsed_time=225 secs]   MSE loss: 0.328

[epoch=8/10, iter= 500  elapsed_time=259 secs]   MSE loss: 0.327

[epoch=9/10, iter= 500  elapsed_time=292 secs]   MSE loss: 0.324

[epoch=10/10, iter= 500  elapsed_time=328 secs]   MSE loss: 0.321

Finished Training
```

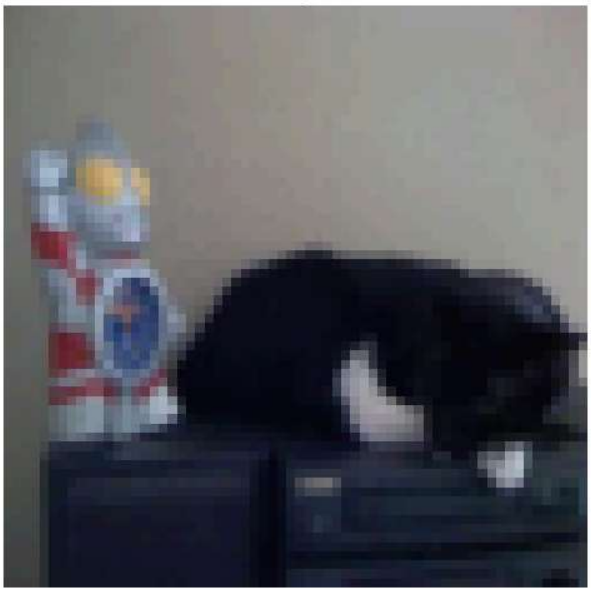
```
/tmp/ipykernel_1334651/406671668.py:608: FutureWarning: You are using `torch.load` w
ith `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

Image

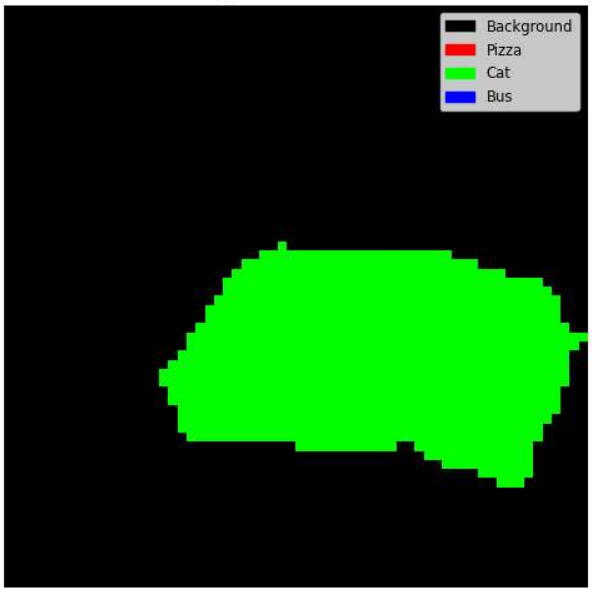
Segmentation Mask



Image

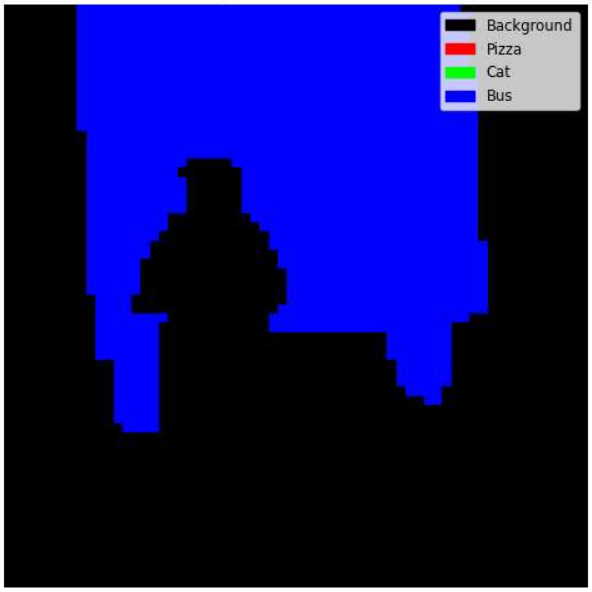


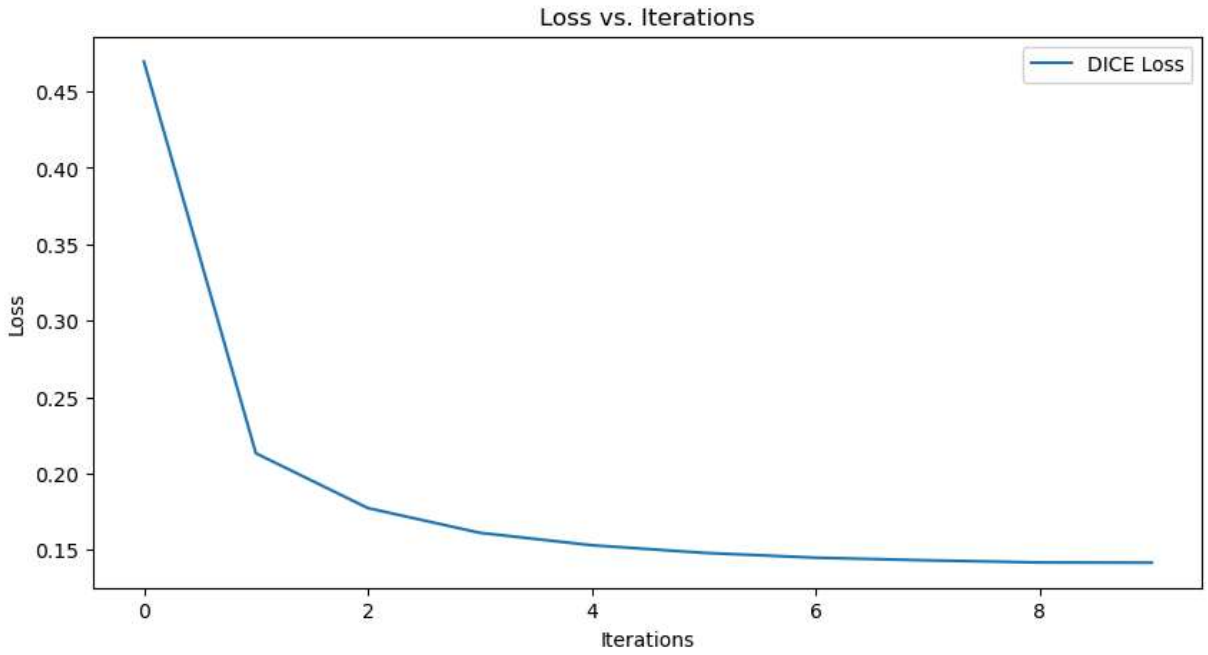
Segmentation Mask



Image

Segmentation Mask





5.3 3 images Dice + MSE

In [17]:

```
dls = DLStudio(
#                   dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                    dataroot = "./../data/datasets_for_DLStudio/data/",
                    image_size = [64,64],
                    path_saved_model = "./saved_model_DICE_MSE_COCO",
                    momentum = 0.9,
                    learning_rate = 1e-4,
                    epochs = 10,
                    batch_size = 4,
                    classes = ('pizza', 'cat', 'bus'),
                    use_gpu = True,
                )

segmenter = DLStudio.SemanticSegmentation(
                    dl_studio = dls,
                    max_num_objects = 5,
                )



segmenter.train_dataloader = train_loader
segmenter.test_dataloader  = val_loader


model = segmenter.mUNet(skip_connections=True, depth=16)

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

DICE_MSE_Loss = segmenter.DICE_MSE_run_code_for_training_for_semantic_segmentation(

segmenter.run_code_for_visualize_semantic_segmentation(model)
```

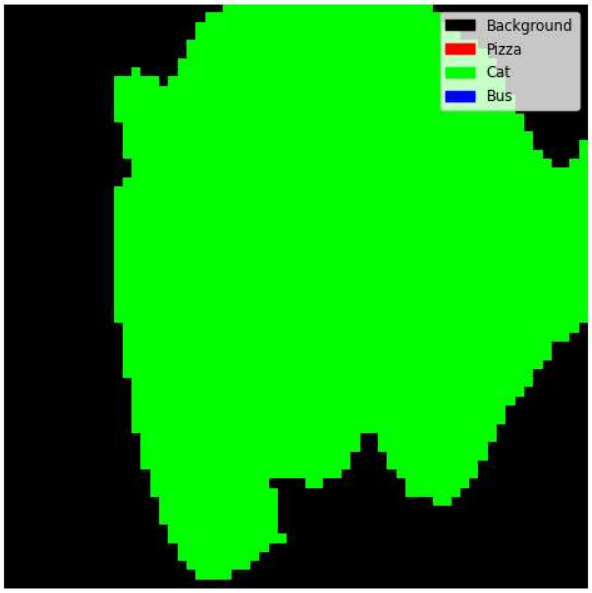
```
plt.figure(figsize=(10,5))
plt.plot(MSELoss, label='DICE Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs. Iterations')
plt.legend()
plt.show()
```

```
The number of learnable parameters in the model: 7687428


[epoch=1/10, iter= 500  elapsed_time= 17 secs]   MSE loss: 20.369

[epoch=2/10, iter= 500  elapsed_time= 52 secs]   MSE loss: 15.768

[epoch=3/10, iter= 500  elapsed_time= 89 secs]   MSE loss: 15.377

[epoch=4/10, iter= 500  elapsed_time=124 secs]   MSE loss: 15.282

[epoch=5/10, iter= 500  elapsed_time=160 secs]   MSE loss: 15.024

[epoch=6/10, iter= 500  elapsed_time=196 secs]   MSE loss: 14.936

[epoch=7/10, iter= 500  elapsed_time=230 secs]   MSE loss: 14.696

[epoch=8/10, iter= 500  elapsed_time=264 secs]   MSE loss: 14.694

[epoch=9/10, iter= 500  elapsed_time=299 secs]   MSE loss: 14.597

[epoch=10/10, iter= 500  elapsed_time=335 secs]   MSE loss: 14.357

Finished Training
```

```
/tmp/ipykernel_1334651/406671668.py:608: FutureWarning: You are using `torch.load` w
ith `weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/ma
in/SECURITY.md#untrusted-models for more details). In a future release, the default
value for `weights_only` will be flipped to `True`. This limits the functions that c
ould be executed during unpickling. Arbitrary objects will no longer be allowed to b
e loaded via this mode unless they are explicitly allowlisted by the user via `torc
h.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
` for any use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
```

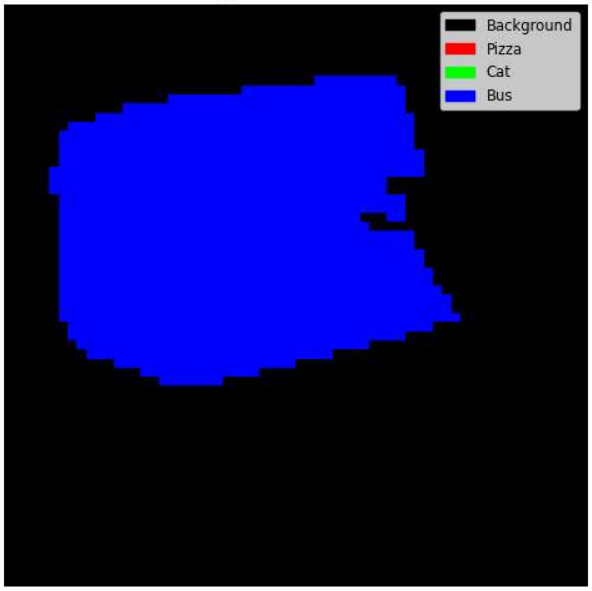
Image

Segmentation Mask

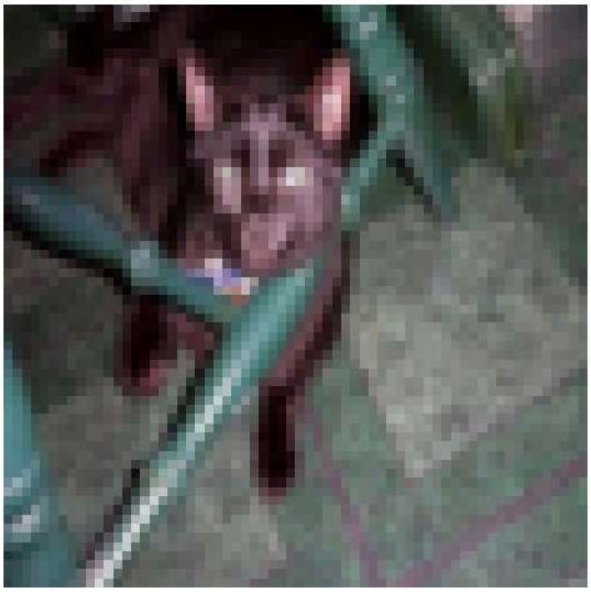


Image

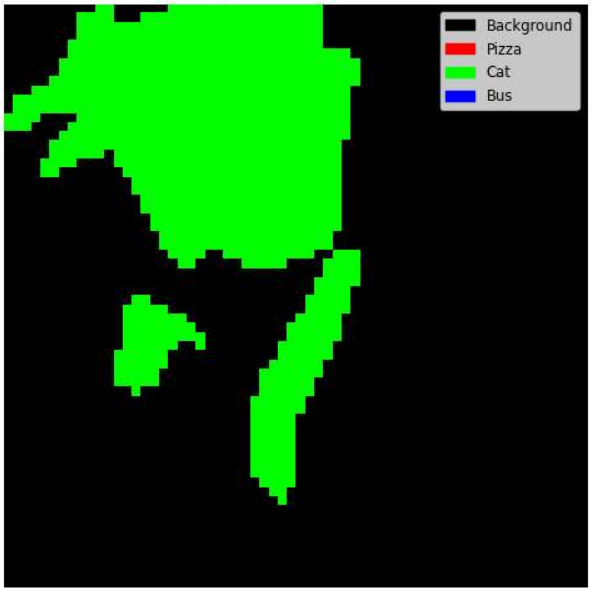
Segmentation Mask

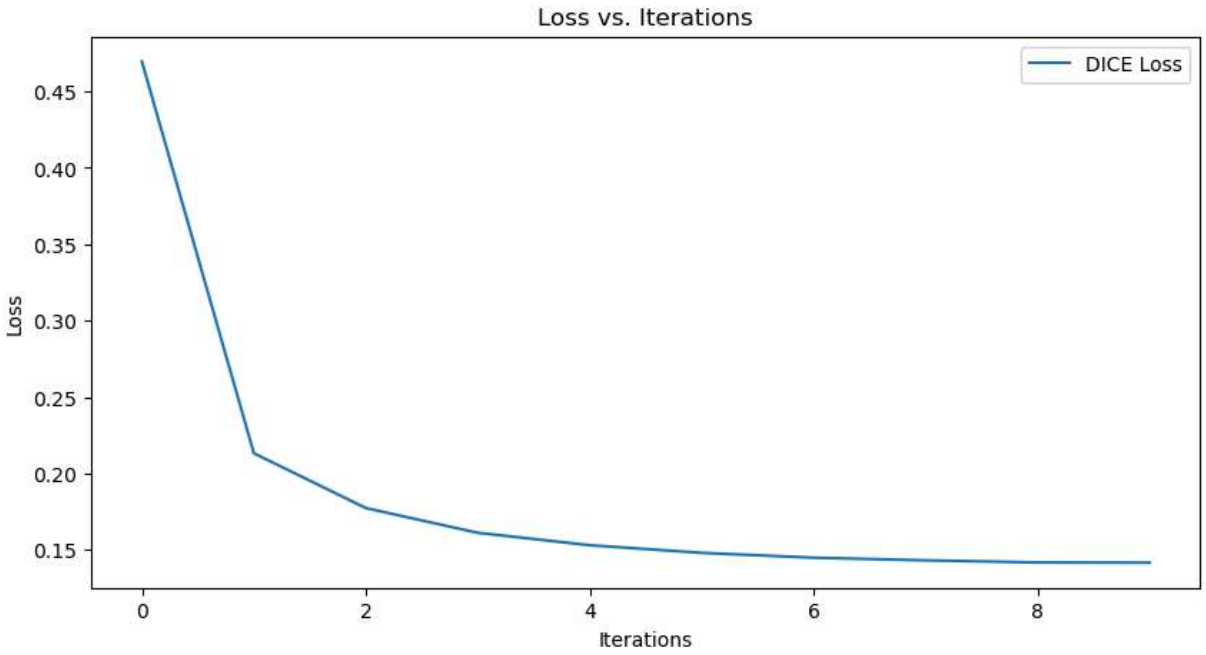


Image



Segmentation Mask





5.4 Observation

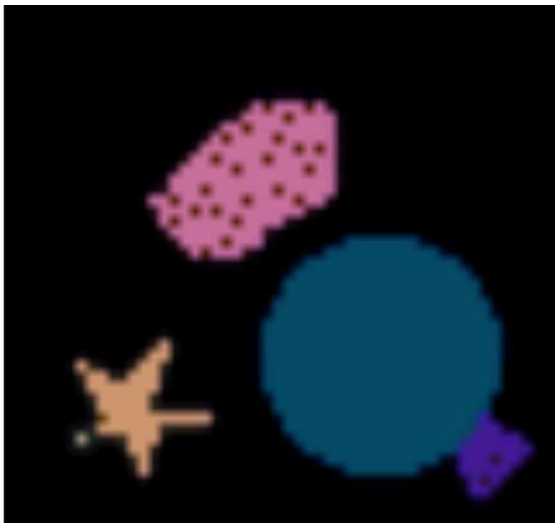
Across all three loss functions, MSE, Dice, and the combined Dice+MSE, the models achieved consistently strong performance on the semantic segmentation task. The resulting segmentation outputs were visually accurate, this suggests that the network architecture is robust enough to learn effectively regardless of the loss formulation, and the dataset is well-structured for training. While Dice loss is theoretically better suited for segmentation due to its focus on overlap, in this case, all loss functions led to comparable and satisfactory results.

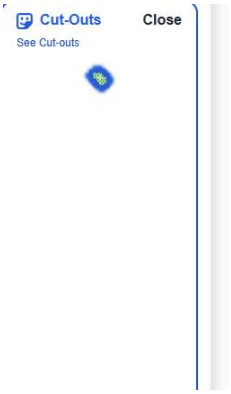
6 Bonus

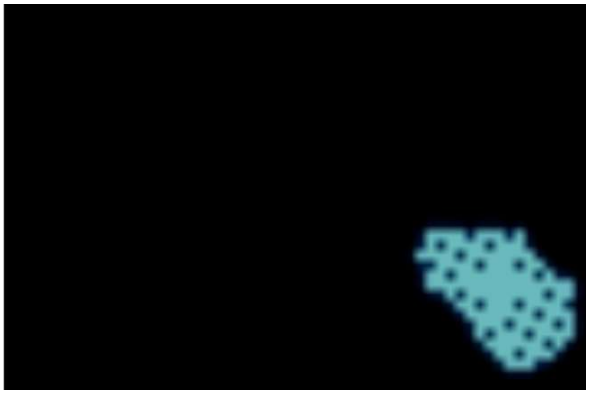
6.1 5 shapes images

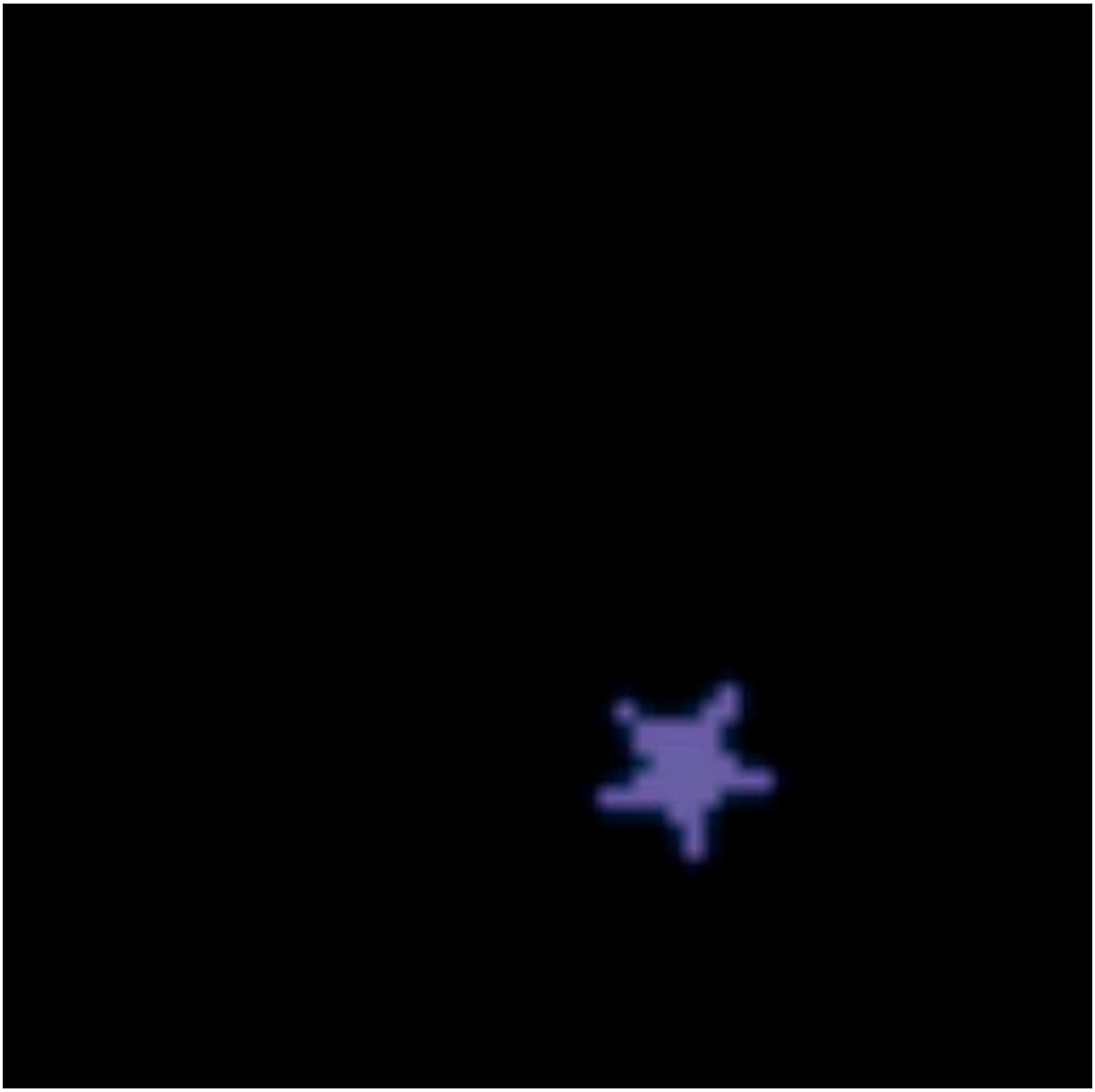
In [8]:
```
from IPython.display import Image, display
display(Image(filename="saved_images/Screenshot6.png"))
display(Image(filename="saved_images/Screenshot7.png"))
display(Image(filename="saved_images/Screenshot8.png"))
display(Image(filename="saved_images/Screenshot9.png"))
display(Image(filename="saved_images/Screenshot10.png"))
```


Tools
Upload
Gallery
Cut-Outs
Close
See Cut-outs

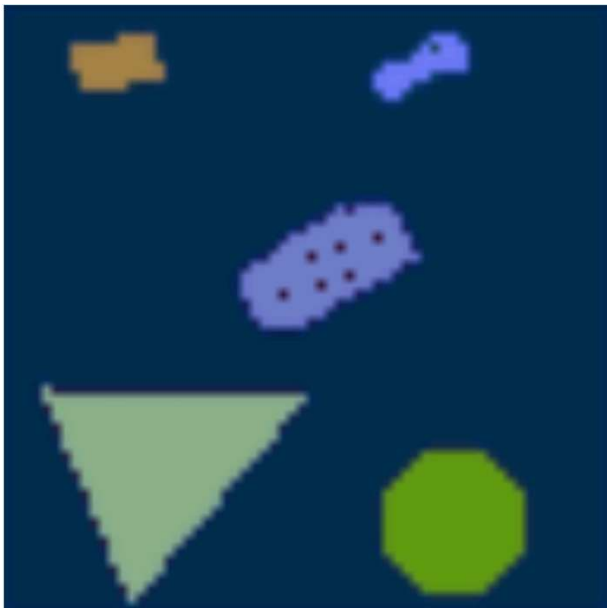


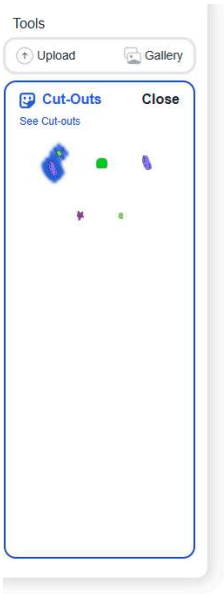

Cut-Outs
Close
See Cut-outs





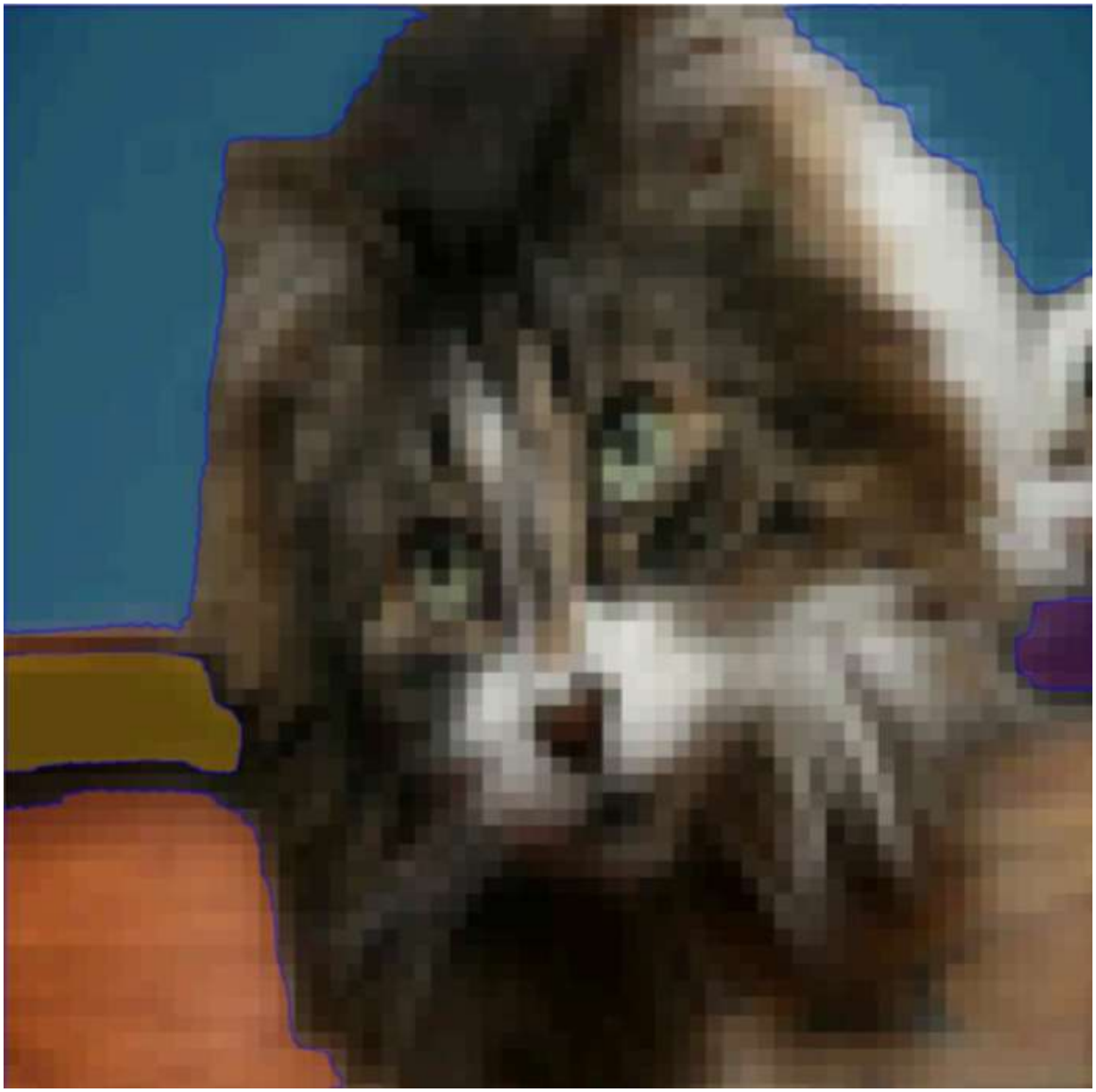

Tools
Upload
Gallery
Cut-Outs
Close
See Cut-outs

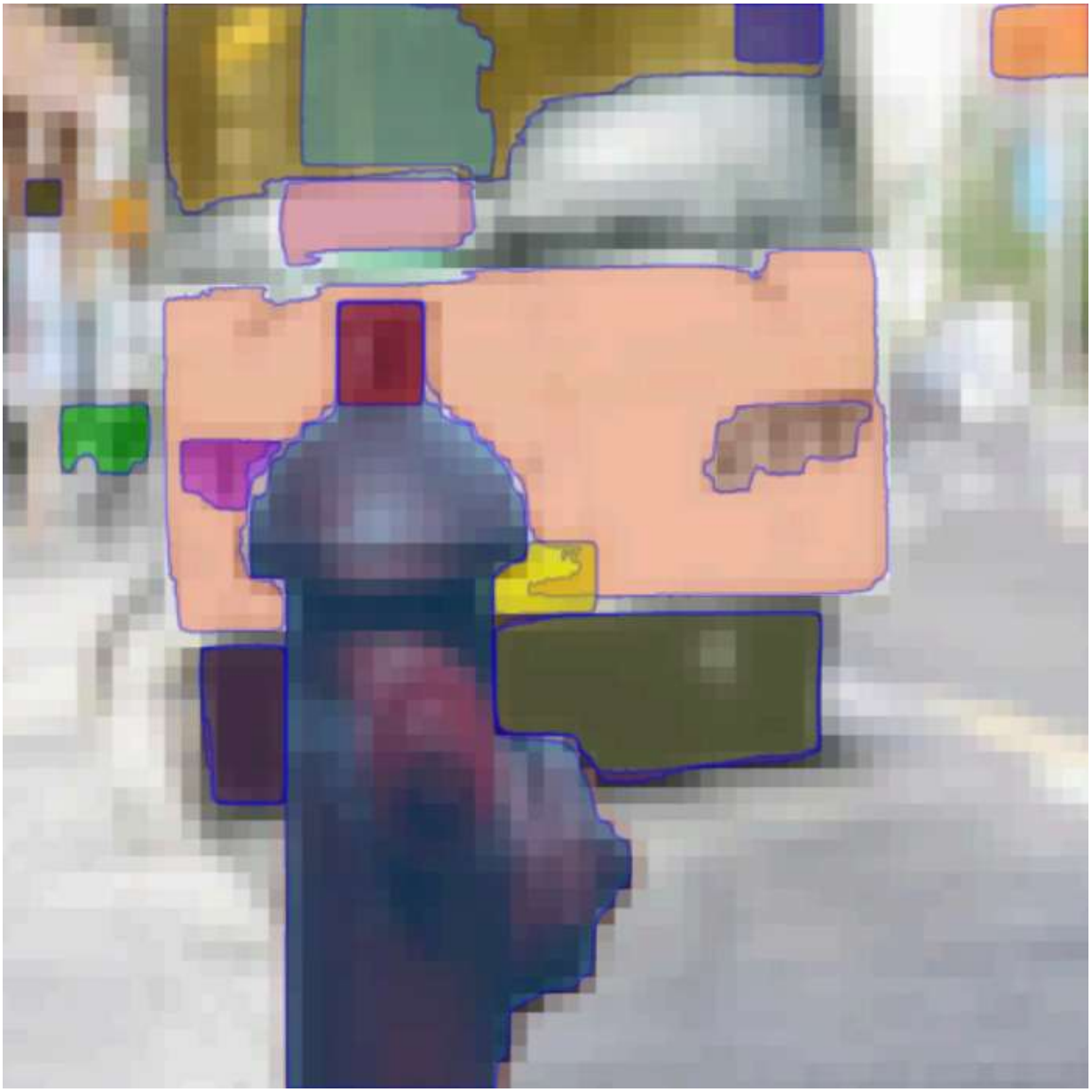


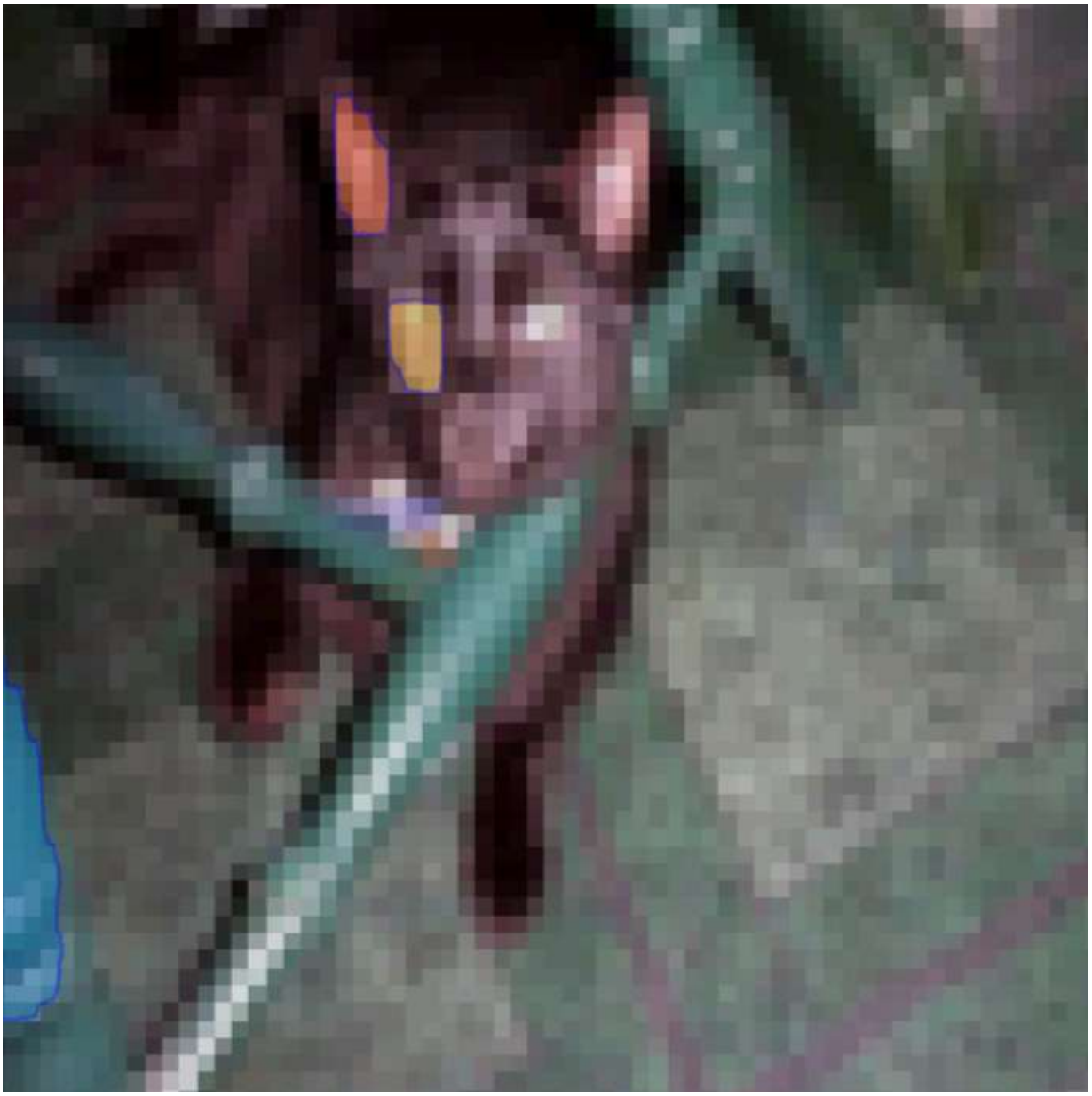


6.2 5 COCO images

In [7]:
```
from IPython.display import Image, display
display(Image(filename="saved_images/Screenshot1.png"))
display(Image(filename="saved_images/Screenshot2.png"))
display(Image(filename="saved_images/Screenshot3.png"))
display(Image(filename="saved_images/Screenshot4.png"))
display(Image(filename="saved_images/Screenshot5.png"))
```







6.3 Observations

One thing I observed is that the images become noticeably more blurry after processing with SAM. This is likely due to the encode and decode pipeline used in SAM's architecture, which may involve rescaling or internal resizing operations that introduce some loss of spatial detail. While I'm not entirely certain, it's reasonable to assume that these steps, common in transformer-based vision models, can degrade fine-grained visual quality, especially at low resolutions like 64×64.

Shape Images

I used the "Everything" mode in SAM to minimize human intervention and ensure a fully automatic segmentation process. As shown, SAM is indeed capable of segmenting everything in the scene, showcasing its strong general-purpose segmentation ability. However, it struggles with the specific five shape classes we are targeting, rectangle, triangle,

disk, oval, and star. This is likely because SAM is not trained or tuned for shape classification tasks, and its segmentation relies primarily on visual boundaries rather than semantic or geometric class labels.

COCO Images

To ensure a fair comparison, I fed SAM the same 64×64 image input as used in our model. This helps make the comparison more intuitive and balanced. Again, I used the "Everything" mode to minimize manual input. SAM performs well in detecting and segmenting various objects in the scene, reaffirming its strong general segmentation capability. However, it does not perform well on the three target classes, cat, pizza, and bus. This is likely because SAM is not class-aware and does not distinguish between specific categories. Instead, it focuses on segmenting all visible object boundaries, regardless of class relevance, which limits its usefulness in class-specific segmentation tasks.