Chris Ardohain
BME64600/ECE60146

Homework 7: Semantic Segmenation

Objective:

This homework focuses on the application of semantic segmentation U-Net model to the Purdue shapes dataset accompanying DLStudio.

Tasks:

Task 1: Semantic Segmentation by U-Net

U-Net is based on what is now a common encoder-decoder architecture where the encoder path focuses on extracting object featurs and decoder path focuses on mapping those objects back to the pixel space. This idea was first introduced as Fully Connected Networks but U-Net improves upon this architecture by allowing information to pass from encoder to decoder through multiple skip connections that improve the performance of the decoder by preserving fine detail related pixel relationships. The decoder path also takes advantage of transpose convolutions which greatly improves the efficiency of the decoding process by converting convolutions into matrix-vector products, which GPUs are uniquely built to compute.

Task 2: Code Modifications

Code modification for this assignment was relatively light as the semantic segmentation example provided in DLStudio – 2.3.6 (link) provided the framework for data loading, model architecture, and evaluation. The default loss calculation in the original code was MSE and the only modification that needed to be made was in the semantic_segmenation.py file with the inclusion of a if __name__ == "__main__": line for parallel processing tasks in windows.

The second modification related to the development of a Dice loss class which I based partially on example provided in the homework and partially on a Kaggle library found at this link. Lastly, we were asked to combine the two loss criteria into a single loss value. I took two different approaches to this, first scaling Dice by 20 and then by 400. The code modifications can be found in Appendix A as Code Snippets 1-3.

Task 3: Loss Comparisons

The first thing to note about comparing loss values is the large difference in scale between MSE and Dice loss. MSE loss exists as the average of the squares of differences between ground truth and predictions. Raw MSE loss values ranged from ~450 to ~350 over the six epochs. Dice loss, on the other hand, produces a value between 0 and 1 with 1 representing perfect overlap between ground truth and predictions. Therefore, comparison of the loss results is a somewhat messy task. For the purposes of this homework, I normalized the losses between 0 and 1 based upon the minimum and maximum loss values across all epochs for each model. This does not allow a true comparison between models in terms of performance, which would require fixing the loss function, but rather highlights to what extent models increased performance over their training iterations. The normalized loss graphs are provided in Figure 1.

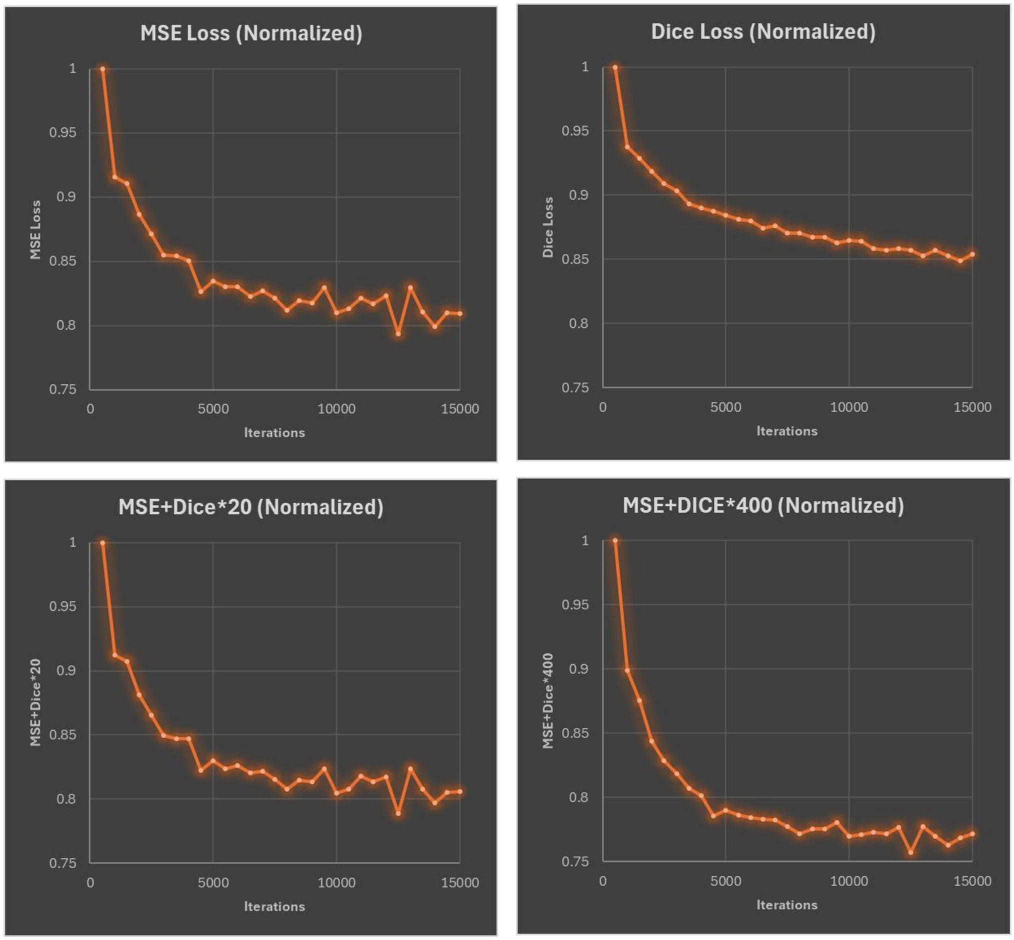

Figure 1: Loss graphs (normalized)

Comparing the initial MSE vs. Dice Loss highlights some significant differences between the two in terms of impact on model performance. The model based on MSE loss improved much more from the initial iteration than the Dice loss model, however, the Dice loss model produced a smoother loss curve indicating a stable learning process but possible underfitting. Given the large differences in raw values between the loss functions, combining the two presented a challenge. I initially followed the advice given in Piazza, scaling the Dice loss by 20, but the result was a curve that remained very similar to the original MSE loss. This is unsurprising since the MSE loss produced values in the 350 to 450 range while Dice now only scaled to between 16 and 20. I then decided to increase the scale value such that MSE and Dice would largely be on equal footing in terms of contribution to overall loss. This was accomplished by scaling Dice by 400 before its addition to MSE. The result highlights not only the greatest increase in performance from the initial model iteration, but also a smoother convergence.

Task 4: Qualitative Results

Unfortunately, visual inspection of test set predictions contradicts the results from the normalized training loss. Visual inspection of model outputs against the test set show that MSE alone performed well at identifying and mapping larger objects while some of the smaller objects or objects that overlapped were sometimes missed. Conversely, the Dice only model did well at identifying shape boundaries but often misclassified the objects and failed to fill in the pixels of solid objects. Combining the two produces some interesting results. When Dice is scaled by a factor of 20, it is comparable to MSE alone results but picks up just slightly more on small or occluded objects. However, when it is scaled by a factor of 400, many of the problems of Dice alone are reintroduced such as not filling in objects completely. Figure 2 provides a visual example of model outputs based on loss functions.

Personally, I believe that Dice loss performs sub optimally against this dataset for one of two reasons. The first may have to deal with the fact that the dataset is more balanced than imbalanced and Dice is designed to work more effectively with imbalanced datasets (small objects, big images). Secondly, it is possible that Dice is undertraining and would perform better with more training data, longer training time, or a more complex model (although the latter two are less likely to help given the simplicity of the images).

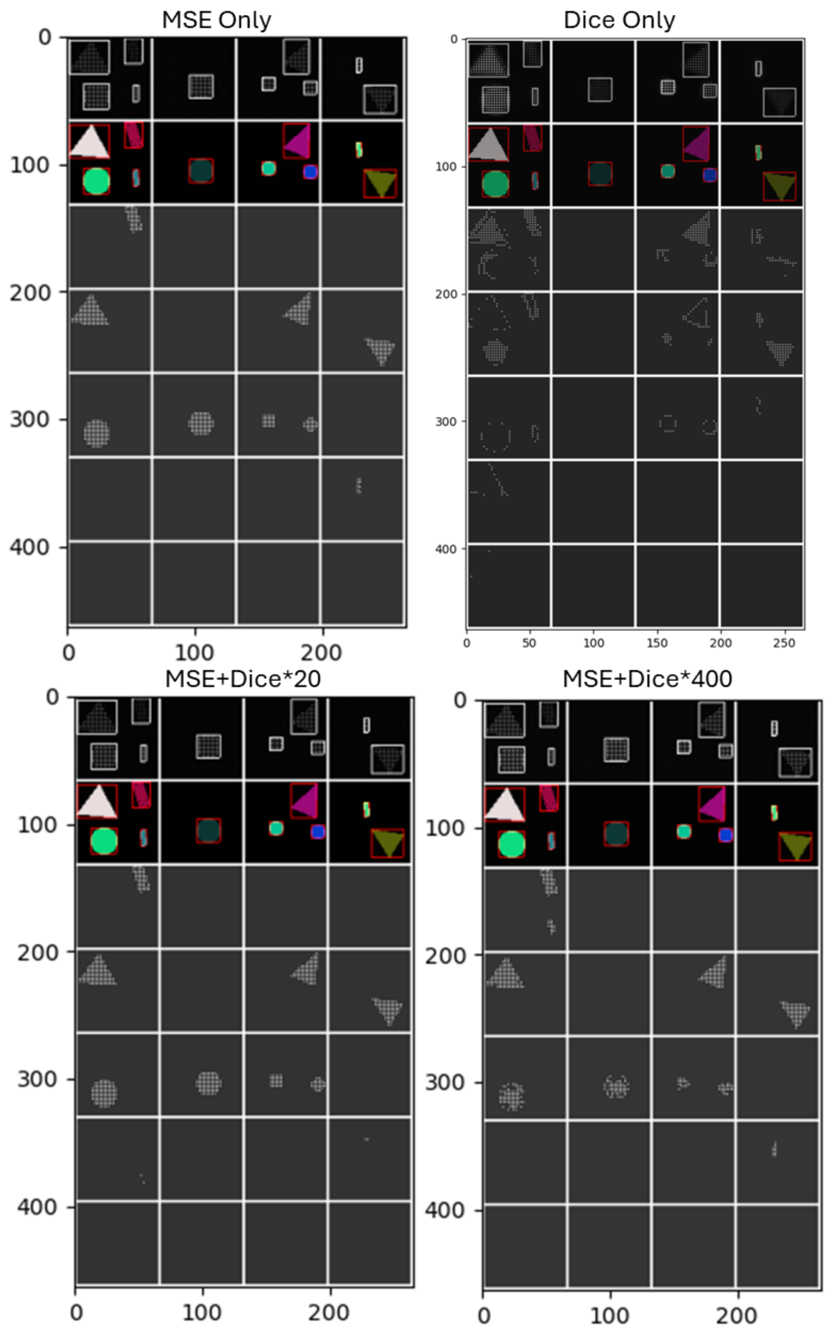

Figure 2: Visual Comparison of Test Set Results

Appendix A – Code

```python
# Adjusted code taken from https://www.kaggle.com/code/bigironsphere/loss-function-library-keras-pytorch#Dice-Loss
    class DiceLoss(nn.Module):
        def __init__(self, weight=None, size_average=True):
            super(SemanticSegmentation.DiceLoss, self).__init__()

        def forward(self, inputs, targets, smooth=1e-6):
            #comment out if your model contains a sigmoid or equivalent activation layer
            #inputs = F.sigmoid(inputs)

            #flatten label and prediction tensors
            inputs = inputs.view(-1)
            targets = targets.view(-1)

            numer = (inputs*targets).sum()
            denom = (inputs*inputs).sum()+(targets*targets).sum()+smooth
            dice = (2*numer)/denom

            return 1-dice
            #return 20*(1-dice)
            #return 400*(1 - dice)
```

Code Snippet 1: DiceLoss Class for calculation and forward pass.

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

        #Add in Criterion 2 (Dice Loss)
        criterion1 = nn.MSELoss()
        criterion2= SemanticSegmentation.DiceLoss()

        optimizer = optim.SGD(net.parameters(),
                     lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)
        start_time = time.perf_counter()
        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_tensor'],data['bbox_tensor']
                im_tensor   = im_tensor.to(self.dl_studio.device)
                mask_tensor = mask_tensor.type(torch.FloatTensor)
                mask_tensor = mask_tensor.to(self.dl_studio.device)
                bbox_tensor = bbox_tensor.to(self.dl_studio.device)
                optimizer.zero_grad()
                output = net(im_tensor)

                #MSE Loss
                segmentation_loss = criterion1(output, mask_tensor)
                #Dice Loss
                #segmentation_loss = criterion2(output, mask_tensor)
                #MSE+Dice Loss
                #segmentation_loss = criterion1(output, mask_tensor) + criterion2(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(500)
                    print("[epoch=%d/%d, iter=%4d  elapsed_time=%3d secs]   MSE Loss: %.3f" % (epoch+1, self.dl_studio.epochs, i+1, elapsed_time, 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)
```

Code Snippet 2: Modification to Training code block for the inclusion of Dice as a loss function.

```python
#run code
if __name__=="__main__":

    seed = 1234
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False
    os.environ['PYTHONHASHSEED'] = str(seed)

    dls = DLStudio(
    #                  dataroot = "/home/kak/ImageDatasets/PurdueShapes5MultiObject/",
                      #dataroot = "./data/PurdueShapes5MultiObject/",
                      dataroot="C:/BME_646/data/DLStudio_Data/data/",
                      image_size = [64,64],
                      path_saved_model = "./saved_model",
                      momentum = 0.9,
                      learning_rate = 1e-4,
                      epochs = 6,
                      batch_size = 4,
                      classes = ('rectangle','triangle','disk','oval','star'),
                      use_gpu = True,
                  )

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

    dataserver_train = SemanticSegmentation.PurdueShapes5MultiObjectDataset(
                              train_or_test = 'train',
                              dl_studio = dls,
                              segmenter = segmenter,
                              dataset_file = "PurdueShapes5MultiObject-10000-train.gz",
                            )
    dataserver_test = 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.requires_grad)
    print("\n\nThe number of learnable parameters in the model: %d\n" % number_of_learnable_params)

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


    segmenter.run_code_for_training_for_semantic_segmentation(model)

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

Code Snippet 3: Modification of Semantic_Segmentation.py code to manage parallel process in windows (if __name__=="__main__":).