Deep Learning HW8

Kabir Batra

March 2025

1 ASPP Model

For this assignment I have updated mUnet in DLStudio by adding ASPP to it. mUnet has an encoder/decoder architecture. The encoder is responsible for taking the input image and converting it into a smaller more abstract representation. The decoder on the other hand is responsible for taking this abstract, smaller representation and mapping it back to the pixel level. The encoder in this model uses regular convolutions to reduce the size of the input. The decoder then uses transpose convolutions to increase the resolution of the encoded data. The enlargement is done gradually using multiple transpose convolutions. The encoder and decoders both use skipblocks in their architecture. This model also had skip connections between the corresponding levels of abstraction in the encoder and decoder. These skip connections are crucial to the models ability to carry out semantic segmentation. In this model as data passes through the multiple layers of the encoder, a portion of it is saved and then reintroduced as it goes through the decoder. This model uses MSE loss as the loss function.

To modify mUnet I added a 1x1 convolution and 3 Atrous convolutions at the end of the encoder. The output of these 4 convolutions is then concatenated together and passed into another 1x1 convolution that outputs a 128 channel output. Adding this ASPP essentially helps the model better understand both the fine details and the broader spatial features in an image. This makes the model better at identifying objects of varied sizes. The Atrous convolutions are responsible for "seeing" a bigger context of the image. The larger the dialation, the larger the context. The 1x1 convolution is responsible for retaining the finer details. Finally, the last 1x1 convolution is used to merge all the results together.

The code modifications I made can be seen below along with the model outputs.

```python
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 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.
                       SkipBlockDN(64, 64, skip_connections=skip_connections))
                self.skip64dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 64,
                      downsample=True, skip_connections=skip_connections)
                self.skip64to128DN = DLStudio.SemanticSegmentation.SkipBlockDN(64,
                   128, skip_connections=skip_connections )
                self.skip128DN_arr = nn.ModuleList()
                for i in range(self.depth):
                    self.skip128DN_arr.append(DLStudio.SemanticSegmentation.
                       SkipBlockDN(128, 128, skip_connections=skip_connections))
                self.skip128dsDN = DLStudio.SemanticSegmentation.SkipBlockDN
                   (128,128, downsample=True, skip_connections=skip_connections)
                ##  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.
                       SkipBlockUP(64, 64, skip_connections=skip_connections))
                self.skip64usUP = DLStudio.SemanticSegmentation.SkipBlockUP(64, 64,
                    upsample=True, skip_connections=skip_connections)
                self.skip128to64UP = DLStudio.SemanticSegmentation.SkipBlockUP(128,
                    64, skip_connections=skip_connections )
                self.skip128UP_arr = nn.ModuleList()
                for i in range(self.depth):
                    self.skip128UP_arr.append(DLStudio.SemanticSegmentation.
                        SkipBlockUP(128, 128, skip_connections=skip_connections))
                self.skip128usUP = DLStudio.SemanticSegmentation.SkipBlockUP
                    (128,128, upsample=True, skip_connections=skip_connections)
                self.conv_out = nn.ConvTranspose2d(64, 3, 3, stride=2,dilation=2,
                   output_padding=1,padding=2)

                ### MODIFICATION START
                self.conv_i = nn.Conv2d(128, 128, kernel_size=1)
                self.a1 = nn.Conv2d(128, 128, kernel_size=3, padding=2, dilation=2)
                self.a2 = nn.Conv2d(128, 128, kernel_size=3, padding=4, dilation=4)
                self.a3 = nn.Conv2d(128, 128, kernel_size=3, padding=6, dilation=6)
                self.conv_1 = nn.Conv2d(128*4, 128, kernel_size=1)
                ### MODIFICATION END

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

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

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

                ### MODIFICATION START
                c_out = self.conv_i(x)
                a_out1 = self.a1(x)
                a_out2 = self.a2(x)
                a_out3 = self.a3(x)

                a_concat = torch.cat([c_out, a_out1, a_out2, a_out3], dim=1)
                x = self.conv_1(a_concat)
                ### MODIFICATION END

                ## 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
```

Listing 1: ASPP

2 Training Curves

To get the training curves I first create a function to calculate the Dice loss between the ground truth mask and the model predicted mask. The code for doing this is highlighted below.

To generate training curves for best and worst case losses, I tweak the hyper parameters and record the results. Then I compare and contrast these results to find the best and worst-case training loss.

I train the model using learning rates of 0.001, 0.0001 and 0.00001. For the combined loss I try different multipliers for the Dice-loss. I multiply the dice loss by 50, 80 and 100.

I do this testing on both the PurdueShapes5 dataset and the COCO dataset. To do this I had to also create a custom dataset to load in the coco dataset information.

The code and results of all of this testing are detailed below.

```
class CustomDataset(Dataset):
  def __init__(self, dataset_path, type, transform=transforms.Compose([transforms.
     ToTensor(),transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])):
    self.img_path = [os.path.join(dataset_path, type+"_imgs", file) for file in os.
       listdir(os.path.join(dataset_path, type+"_imgs"))]
    self.ann_path = os.path.join(dataset_path, type+"_anns")
    self.transform = transform

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

  def __getitem__(self, idx):

    img = Image.open(self.img_path[idx])
    if img.mode != 'RGB':
        img = img.convert('RGB')

    img_name = self.img_path[idx].split(".")[0].split("\\")[-1]

    for i, ann in enumerate(glob.glob(os.path.join(self.ann_path, img_name+"_*.npy"
       ))):
        with open(ann, 'rb') as f:
            mask = np.load(f)

    if self.transform:
        img = self.transform(img)

    bbox_tensor = torch.zeros(3,3,4, dtype=torch.float)

    sample = {'image'        : img,
              'mask_tensor'  : mask,
              'bbox_tensor'  : bbox_tensor }
    return sample
```

Listing 2: COCO Dataloader

```python
import torch

def dice_loss(preds: torch.Tensor, ground_truth: torch.Tensor, epsilon: float = 1e-6) -> torch.Tensor:

    preds.requires_grad_(True)
    ground_truth.requires_grad_(True)

    # numerator
    num = torch.sum(preds * ground_truth, dim=(2, 3))

    # denominator
    den = torch.sum(preds * preds, dim=(2, 3)) + torch.sum(ground_truth * ground_truth, dim=(2, 3))

    # Dice coefficient
    dice_coefficient = num / torch.add(den, epsilon)

    # Dice divergence
    dice_loss = torch.sub(1.0,dice_coefficient)

    return torch.sum(dice_loss)
```

Listing 3: Dice Loss

2.1 Best training curves for MSE loss

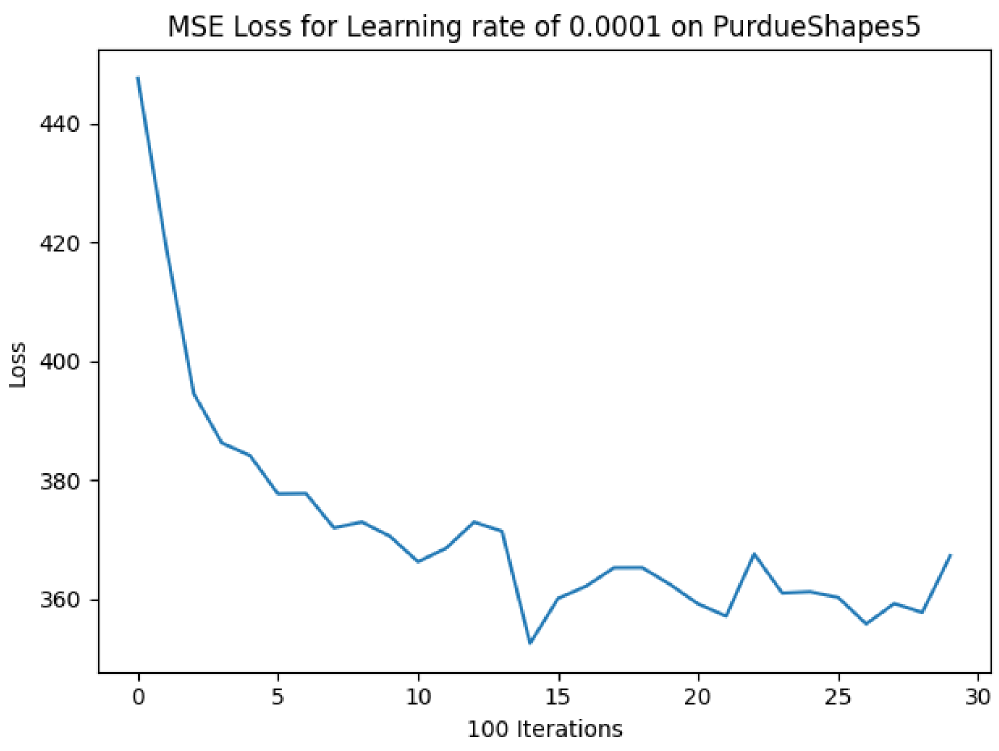


Figure 1: Best training Curve for Purdue Shapes 5 Dataset

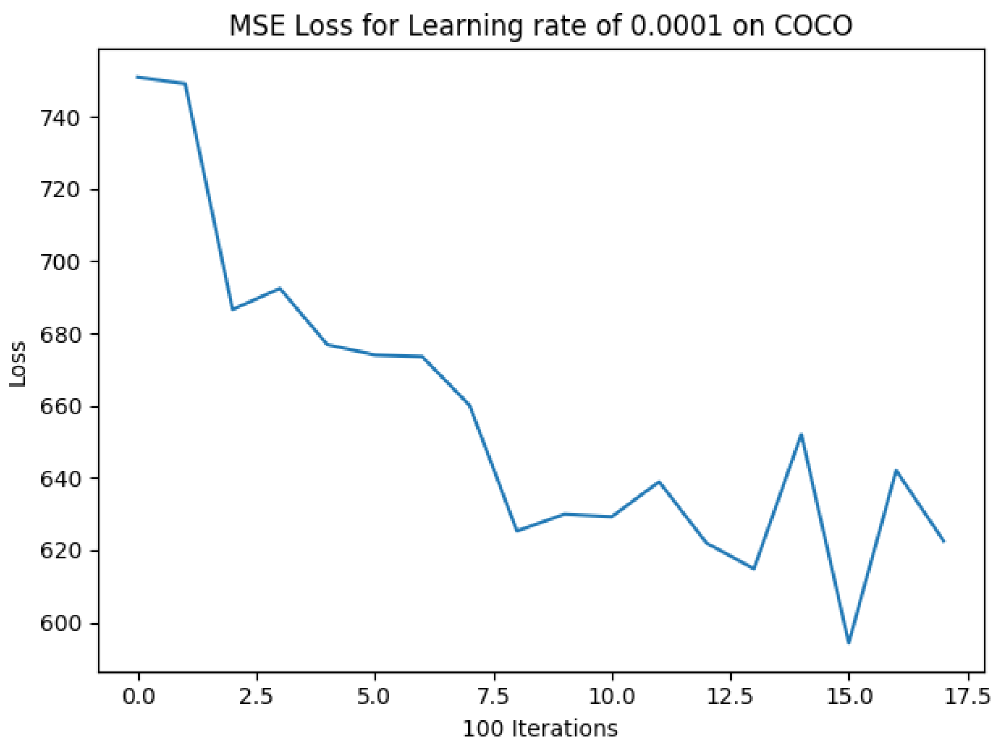


Figure 2: Best training Curve for COCO Dataset

2.2 Best training curves for Dice loss

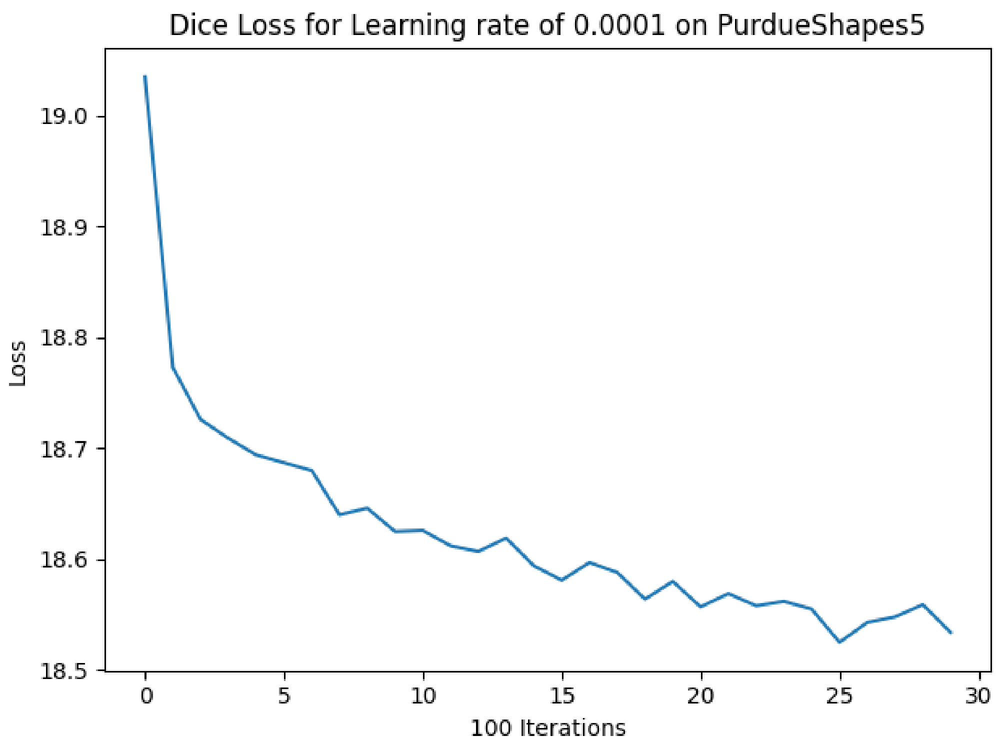


Figure 3: Best training Curve for Purdue Shapes 5 Dataset

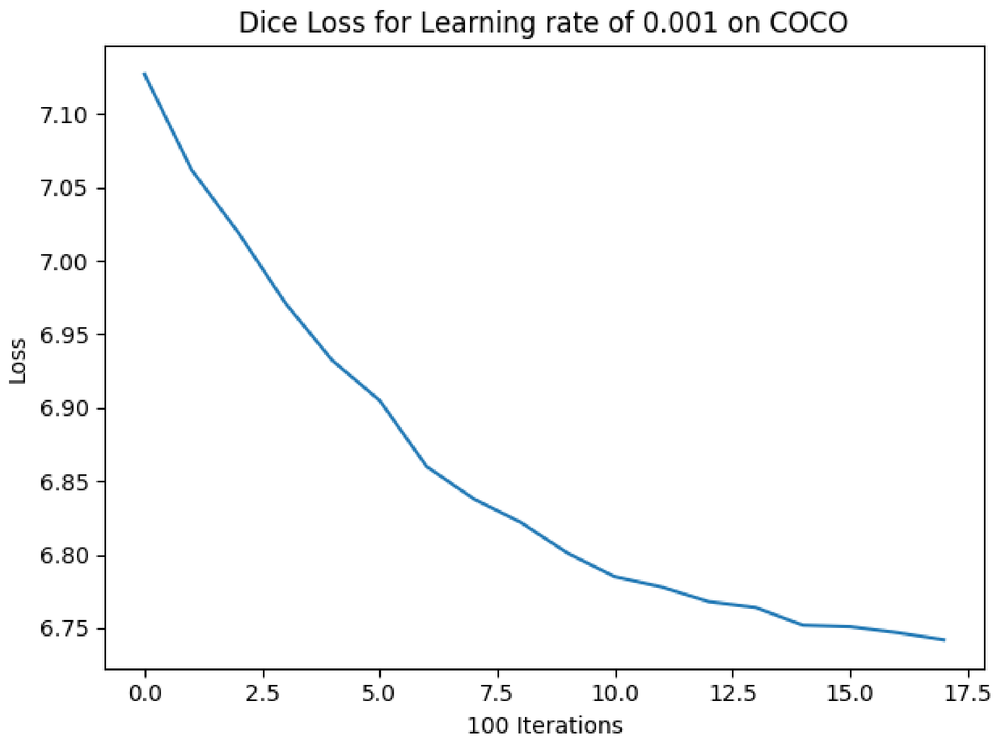


Figure 4: Best training Curve for COCO Dataset

2.3 Best training curves for MSE+Dice loss

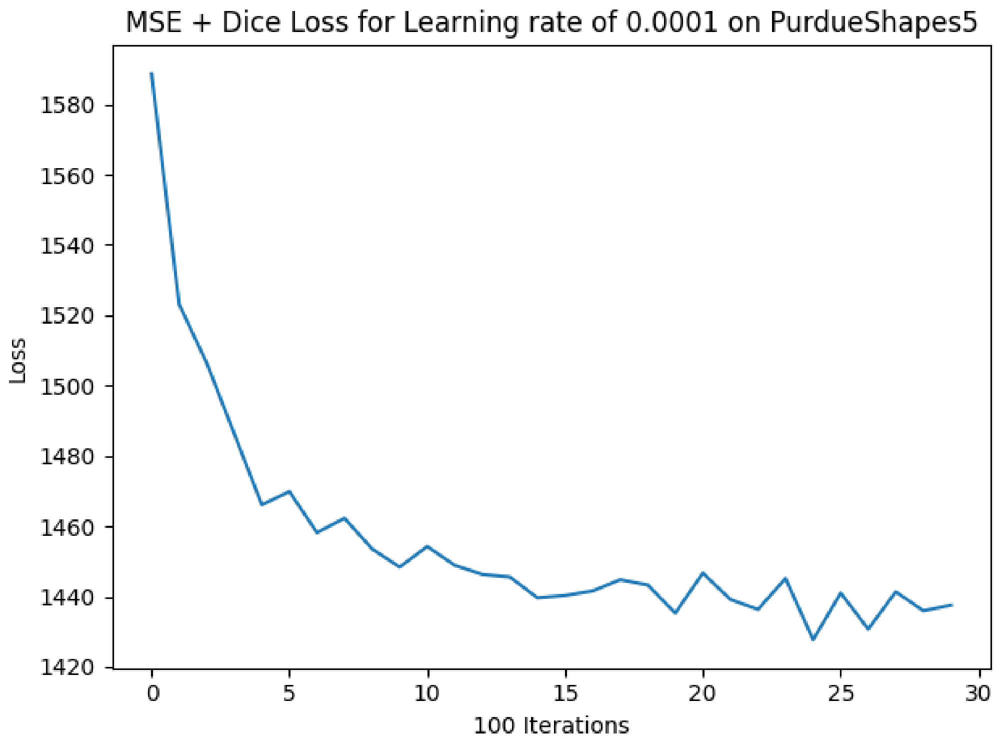


Figure 5: Best training Curve for Purdue Shapes 5 Dataset

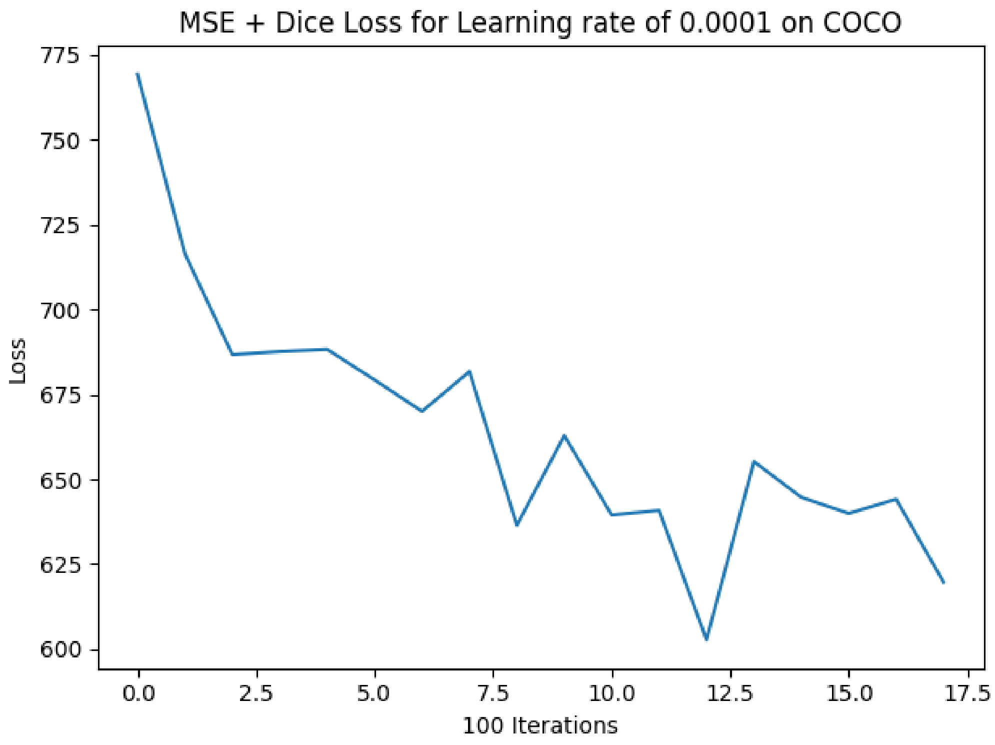


Figure 6: Best training Curve for COCO Dataset

2.4 Worst training curves for MSE loss

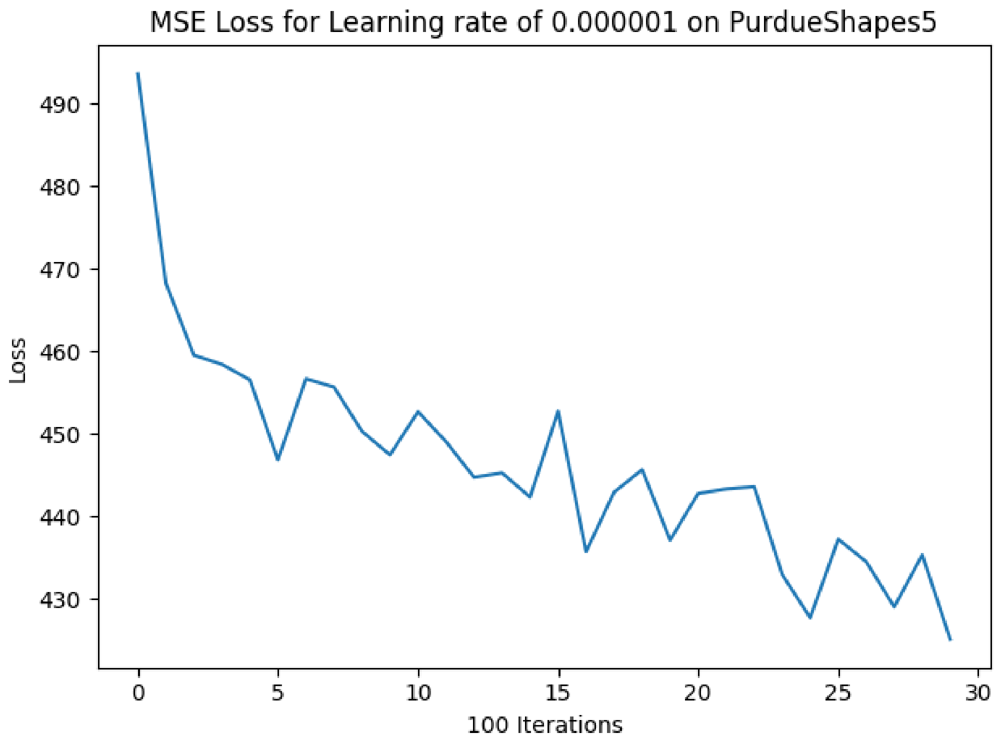


Figure 7: Worst training Curve for Purdue Shapes 5 Dataset

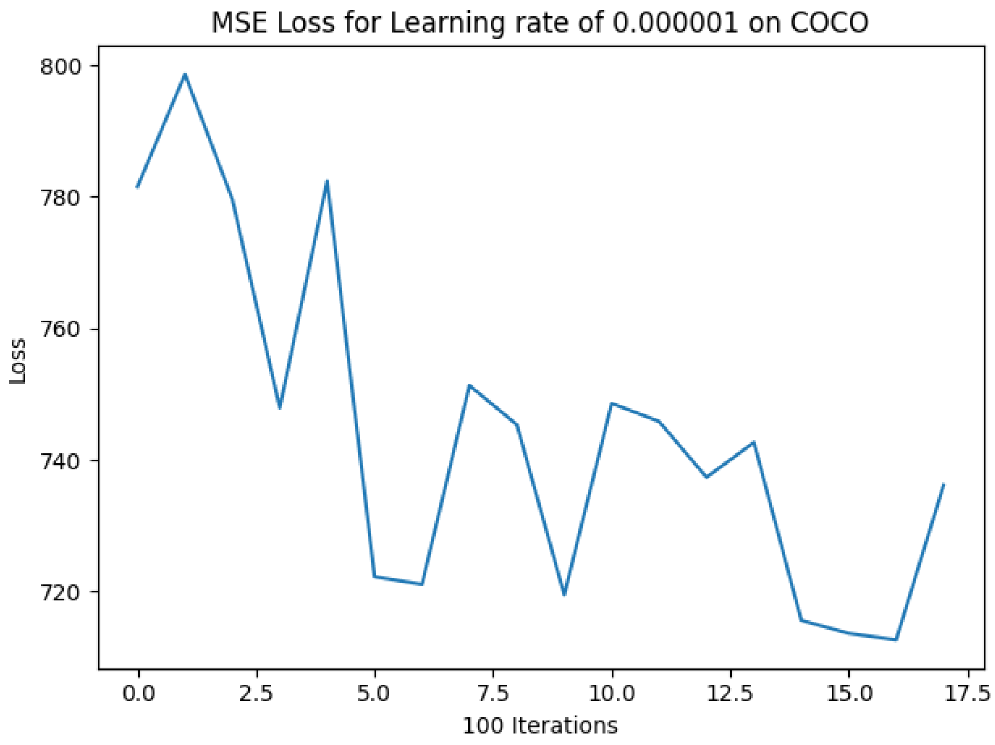


Figure 8: Worst training Curve for COCO Dataset

2.5 Worst training curves for Dice loss

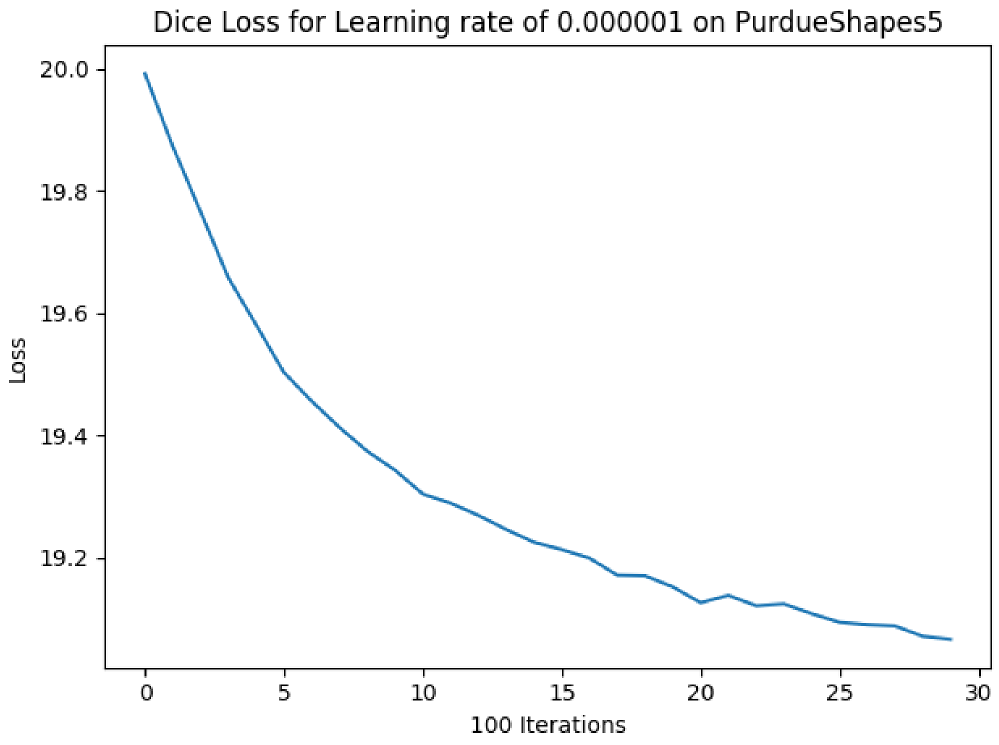


Figure 9: Worst training Curve for Purdue Shapes 5 Dataset

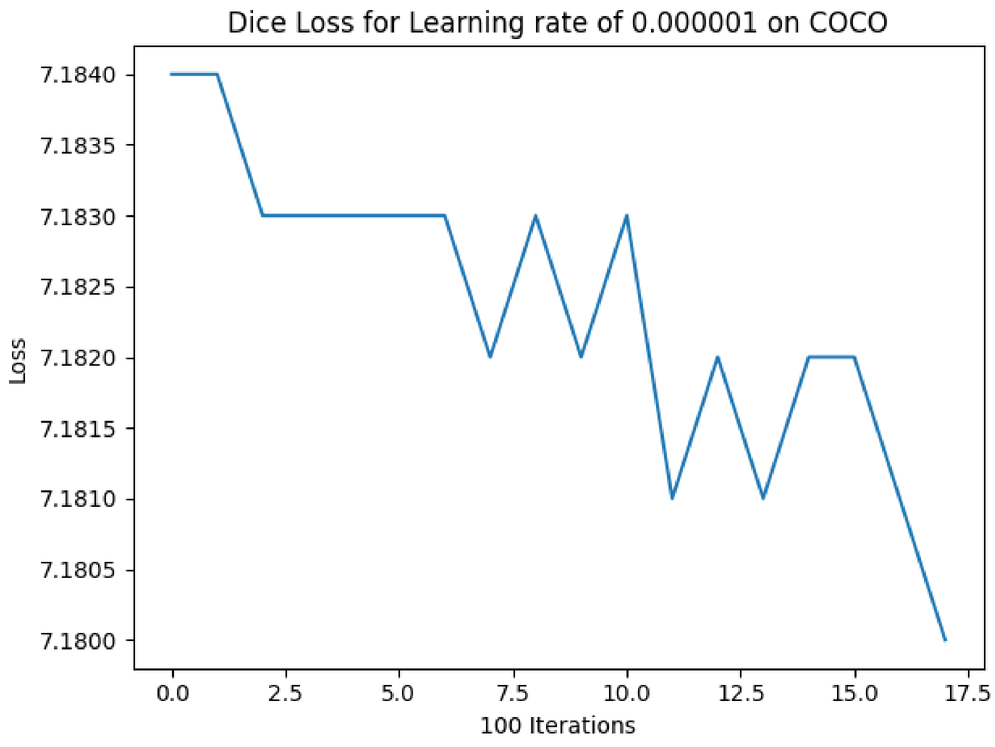


Figure 10: Worst training Curve for COCO Dataset

2.6 Worst training curves for MSE+Dice loss

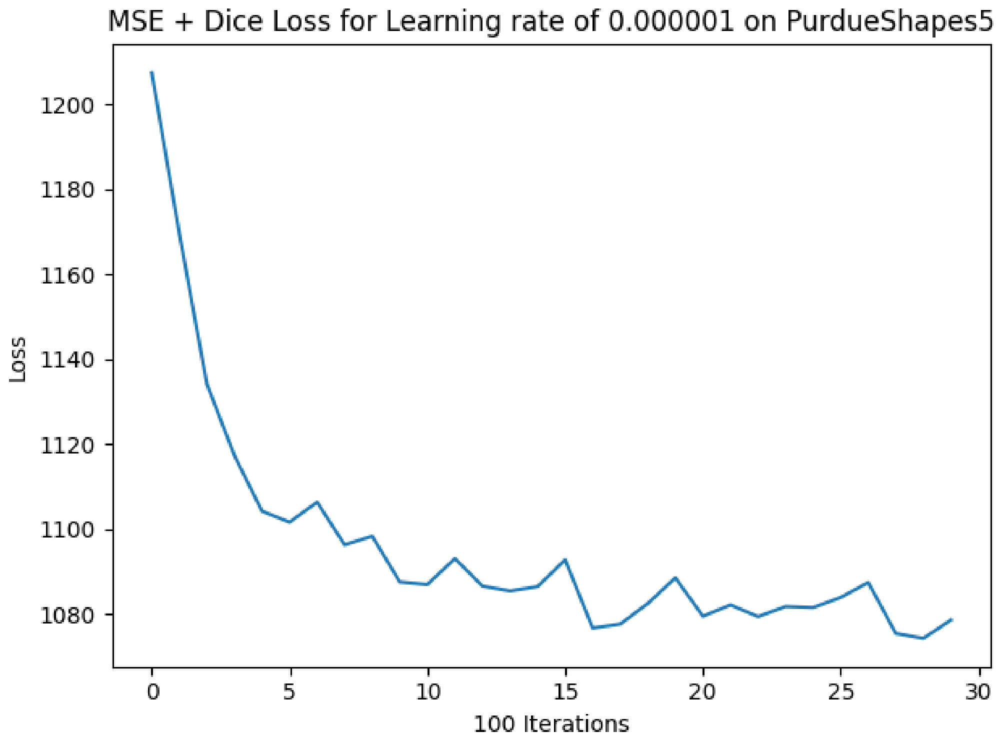


Figure 11: Worst training Curve for Purdue Shapes 5 Dataset

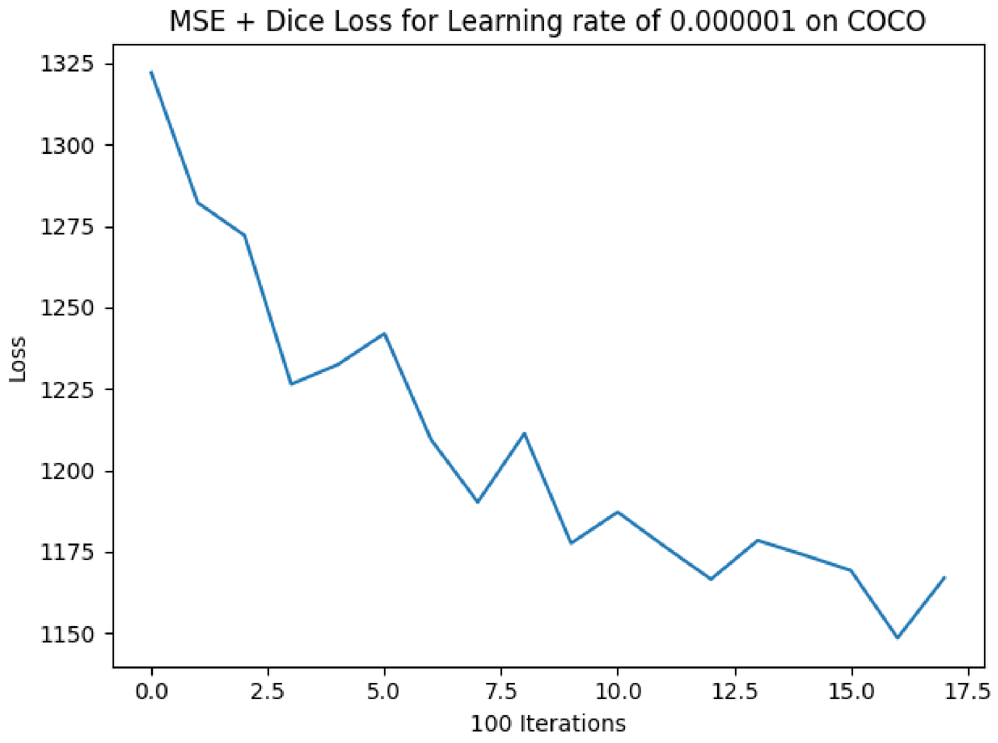


Figure 12: Worst training Curve for COCO Dataset

3 Observations on Coefficient effect

When we use a combination of MSE and Dice loss, we run into scaling problems. The MSE loss is unbounded and can be arbitrarily large, where as the Dice loss is bounded between 0 and 1. This means that when we combine both the losses, the effect of the Dice loss is drowned out. To tackle this problem we multiply the Dice loss by some constant coefficient to make its value more prominent.

For the PurdueShapes5 dataset the coefficients I used to multiply the Dice loss were 20, 40 and 60. The multiplier of 60 brought the Dice loss to almost the same scale as the mse loss. The other multipliers made the Dice loss more prominent but not as prominent as the MSE loss. Consequently as we can see in the graphs below, the multiplier of 60 performed the best.

Similarly for the COCO dataset I tried multipliers of values 60, 80 and 100. Following the same logic as above, the multiplier of 100 performed the best.

For both the datasets, the middle value of the multiplier performed the worst. This could be because in this state the dice loss is not prominent enough to make an actual effect in training, but still adds to the overall value of the loss without getting significantly minimized.

In conclusion, we observe that making both the losses have similar magnitudes helps the model train better. When both the losses contribute fairly equally to the total loss, the model learns the best. These results are highlighted in the graphs below:

3.1 Best training curves for MSE+Dice loss based on coefficients

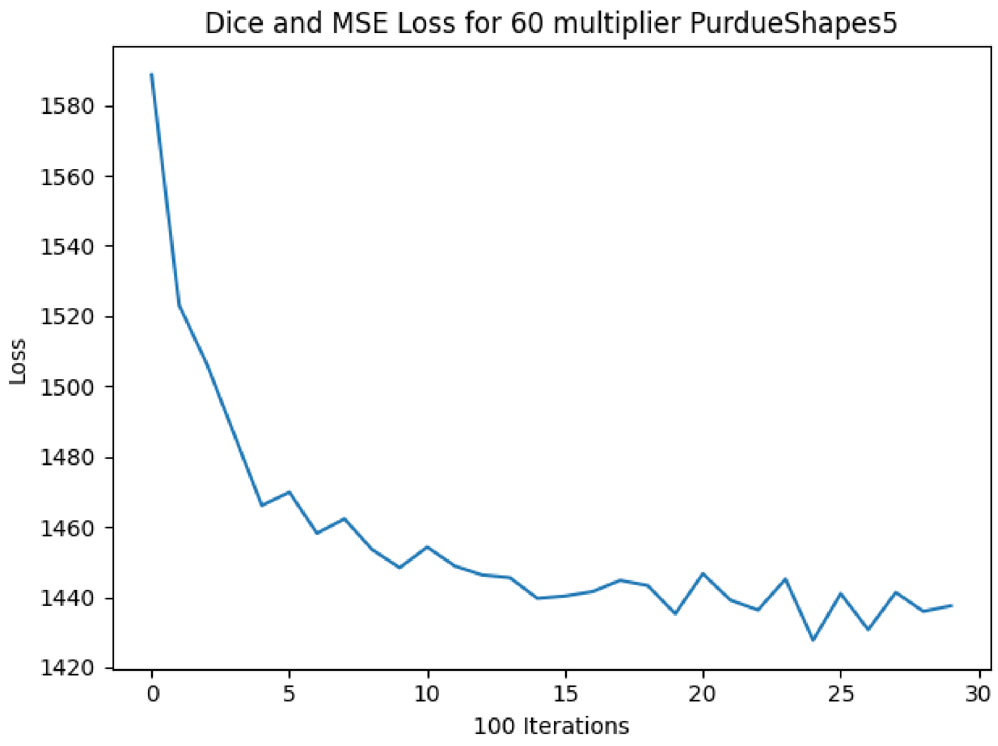


Figure 13: Best training Curve for Purdue Shapes 5 Dataset

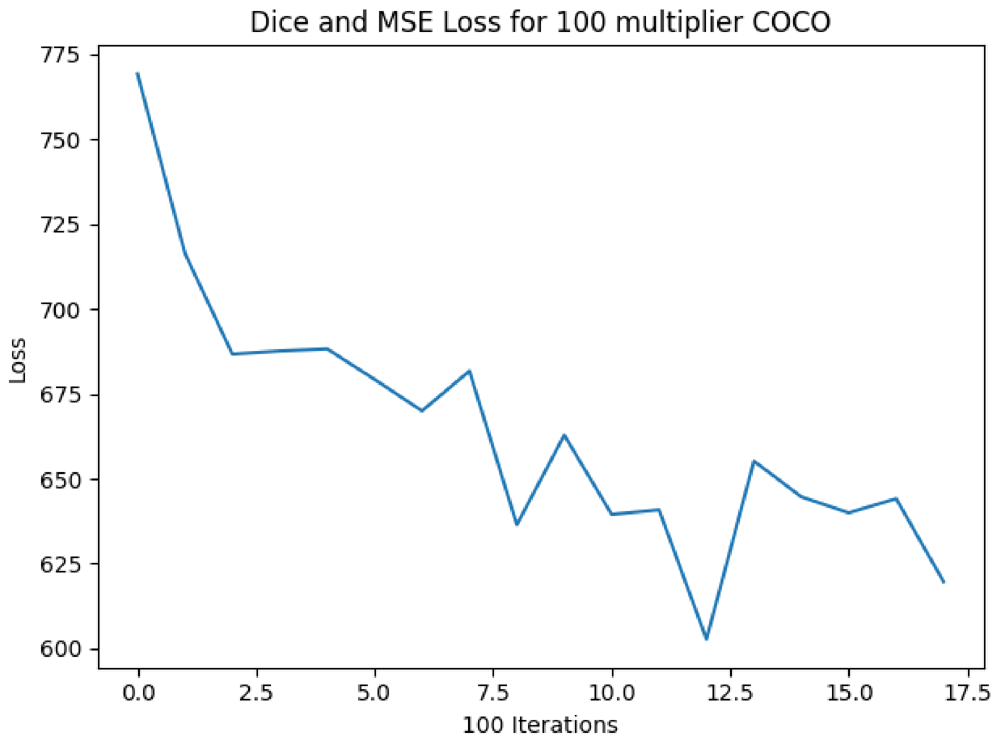


Figure 14: Best training Curve for COCO Dataset

3.2 Worst training curves for MSE+Dice loss based on coefficients

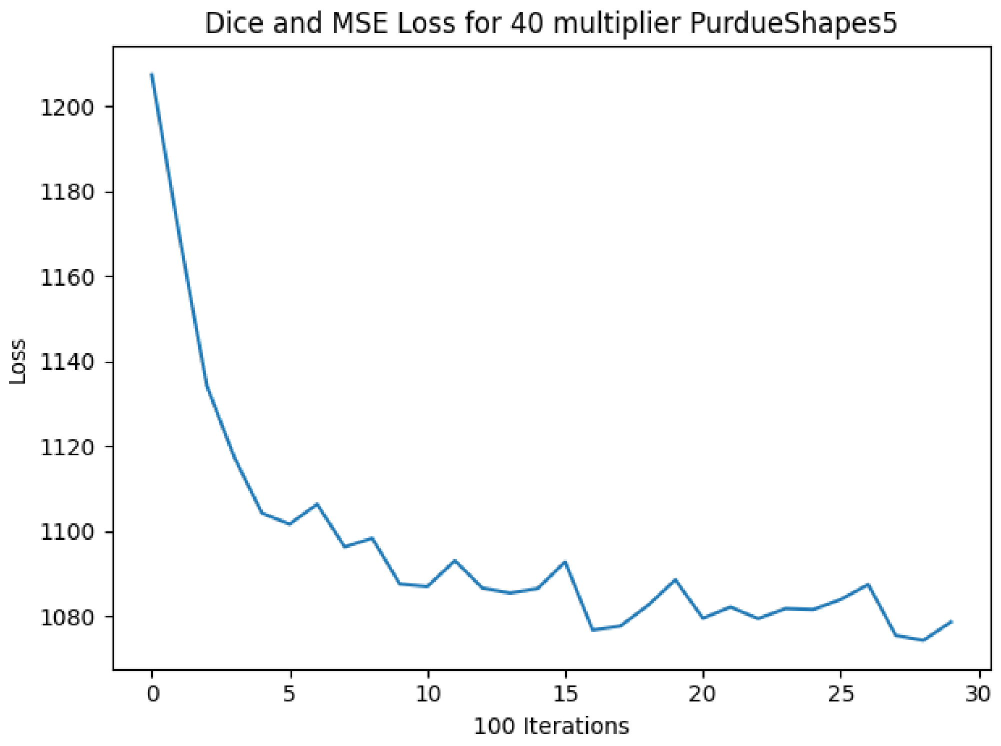


Figure 15: Worst training Curve for Purdue Shapes 5 Dataset

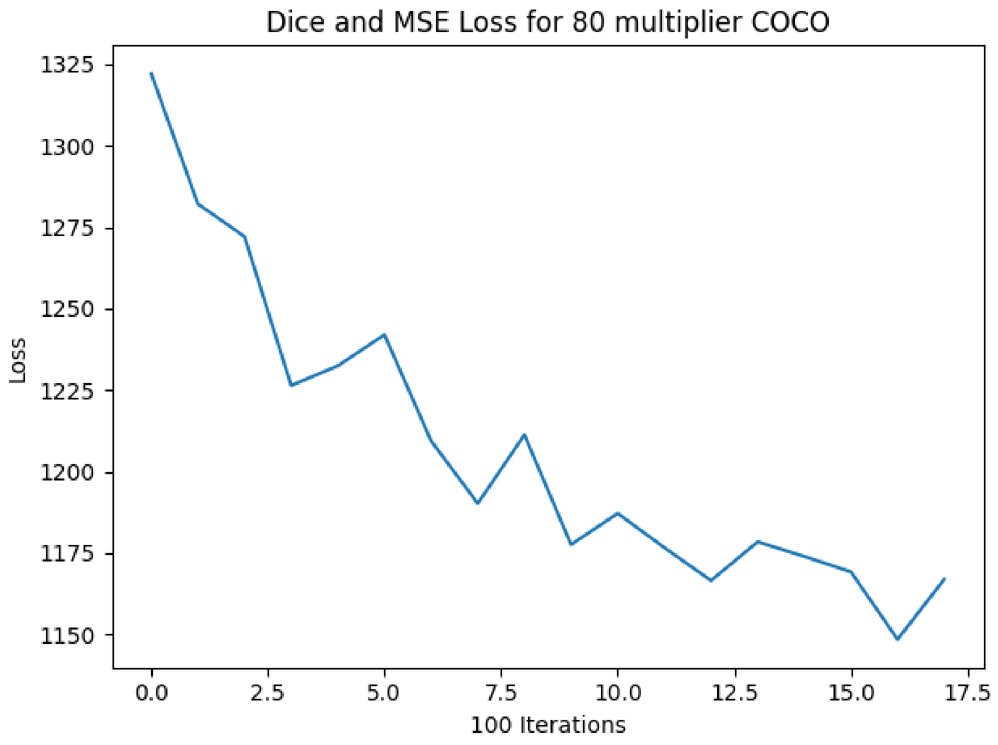


Figure 16: Worst training Curve for COCO Dataset

4 Evaluation and observations

To evaluate the COCO dataset I modify the testing function from DLStudio to plot the outputs from the COCO Dataset. I leave this function unmodified when I test the PurdueShapes5 dataset.

My Modified testing function is below:

```python
def run_code_for_testing_semantic_segmentation(self, net):
            plot_count = 0
            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
            out_count = 0
            with torch.no_grad():
                for i, data in enumerate(self.test_dataloader):
                    im_tensor,mask_tensor,bbox_tensor =data['image'],data['
                       mask_tensor'],data['bbox_tensor']
                    outputs = net(im_tensor)

                    outputs = outputs.cpu().detach().numpy()
                    im_tensor = im_tensor.cpu().detach().numpy()

                    if i % 50 == 0:
                        for idx in range(batch_size):
                            out_count += 1
                            fig, axs = plt.subplots(1, 4, figsize=(10, 10))
                            mask_to_plot_1 = np.zeros((256, 256, 3))
                            mask_to_plot_2 = np.zeros((256, 256, 3))
                            mask_to_plot_3 = np.zeros((256, 256, 3))

                            mask_to_plot_1[np.where((outputs[idx][0] > 25) & (
                               outputs[idx][0] < 85))] = [255, 0, 0]
                            mask_to_plot_2[np.where((outputs[idx][1] > 65) & (
                               outputs[idx][1] < 135))] = [0, 255, 0]
                            mask_to_plot_3[np.where((outputs[idx][2] > 115) & (
                               outputs[idx][2] < 185))] = [0, 0, 255]

                            im = im_tensor[idx].transpose(1, 2, 0)
                            im = im + 1
                            im = im / 2

                            axs[0].imshow(im.astype(np.float64))
                            axs[0].axis('off')
                            axs[0].set_title('Image')

                            axs[1].imshow(mask_to_plot_1.astype(np.uint8))
                            axs[1].axis('off')
                            axs[1].set_title('Pizza')

                            axs[2].imshow(mask_to_plot_2.astype(np.uint8))
                            axs[2].axis('off')
                            axs[2].set_title('Cat')

                            axs[3].imshow(mask_to_plot_3.astype(np.uint8))
                            axs[3].axis('off')
                            axs[3].set_title('Bus')
                            plt.savefig(str(out_count) + '_output' + '.png')
                            plt.clf()
                            plt.close()
```

Listing 4: Tesing Function Code

4.1 MSE Evaluation

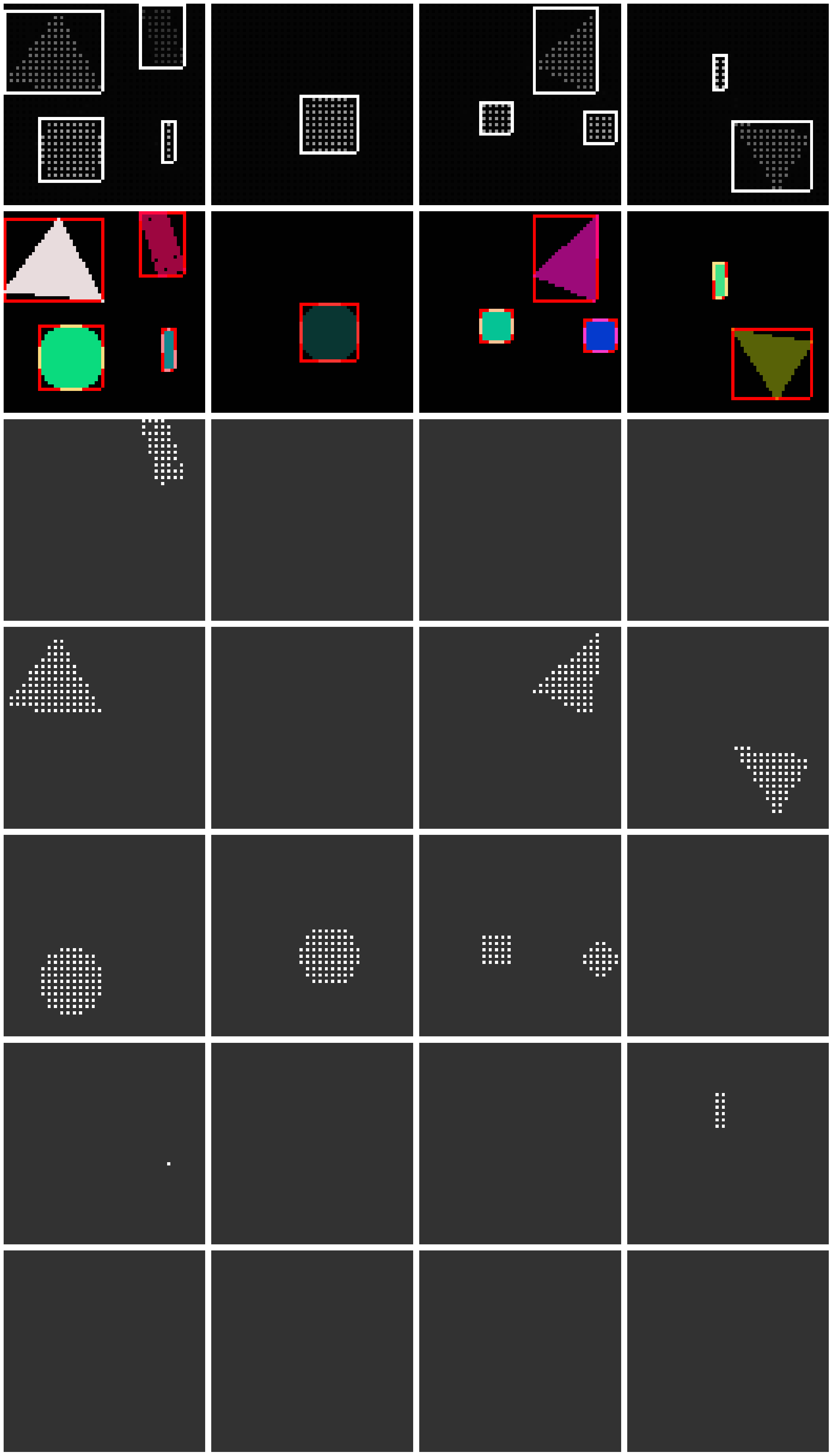


Figure 17: Result 1 for Purdue Shapes 5 Dataset

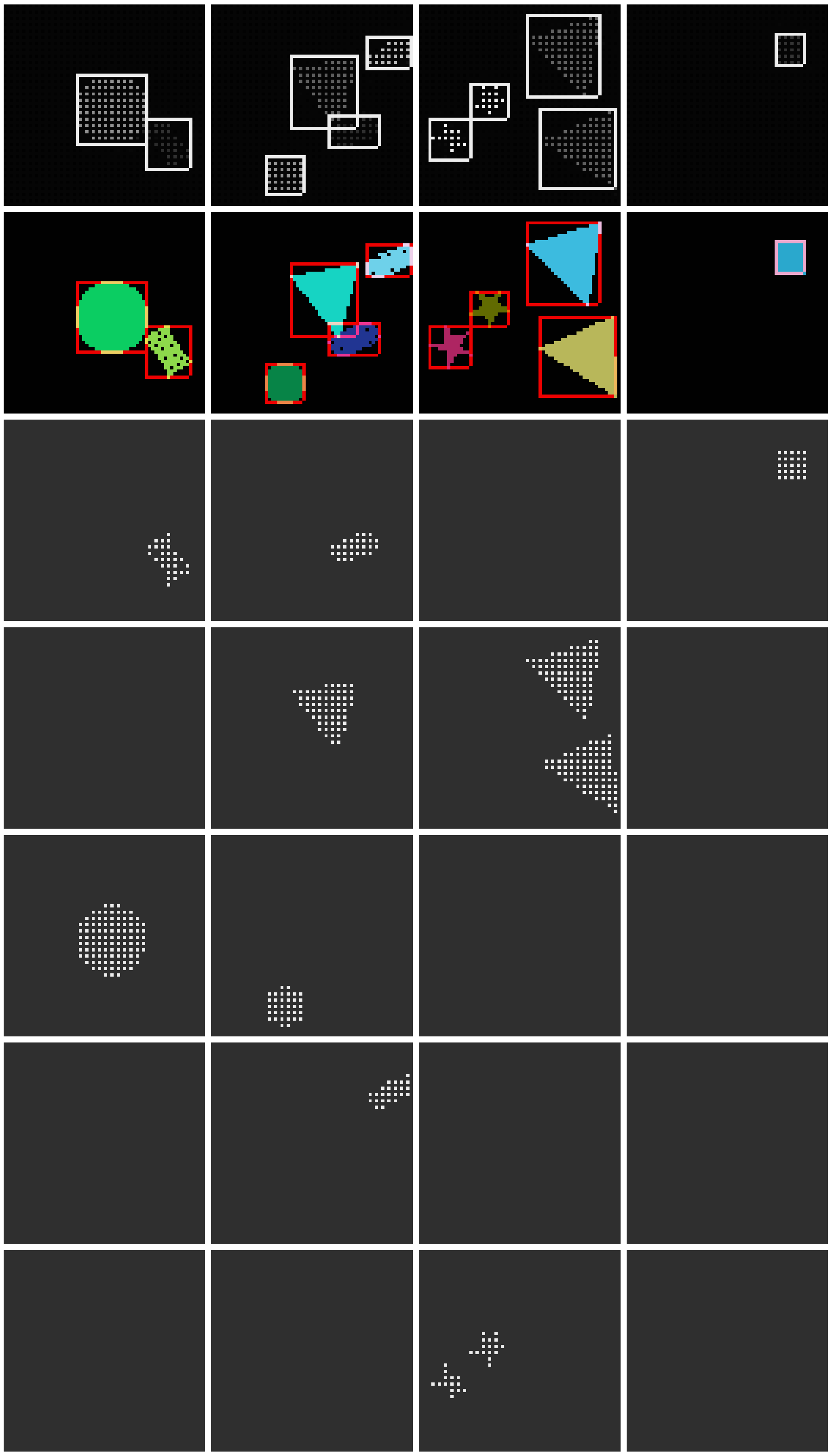


Figure 18: Result 2 for Purdue Shapes 5 Dataset

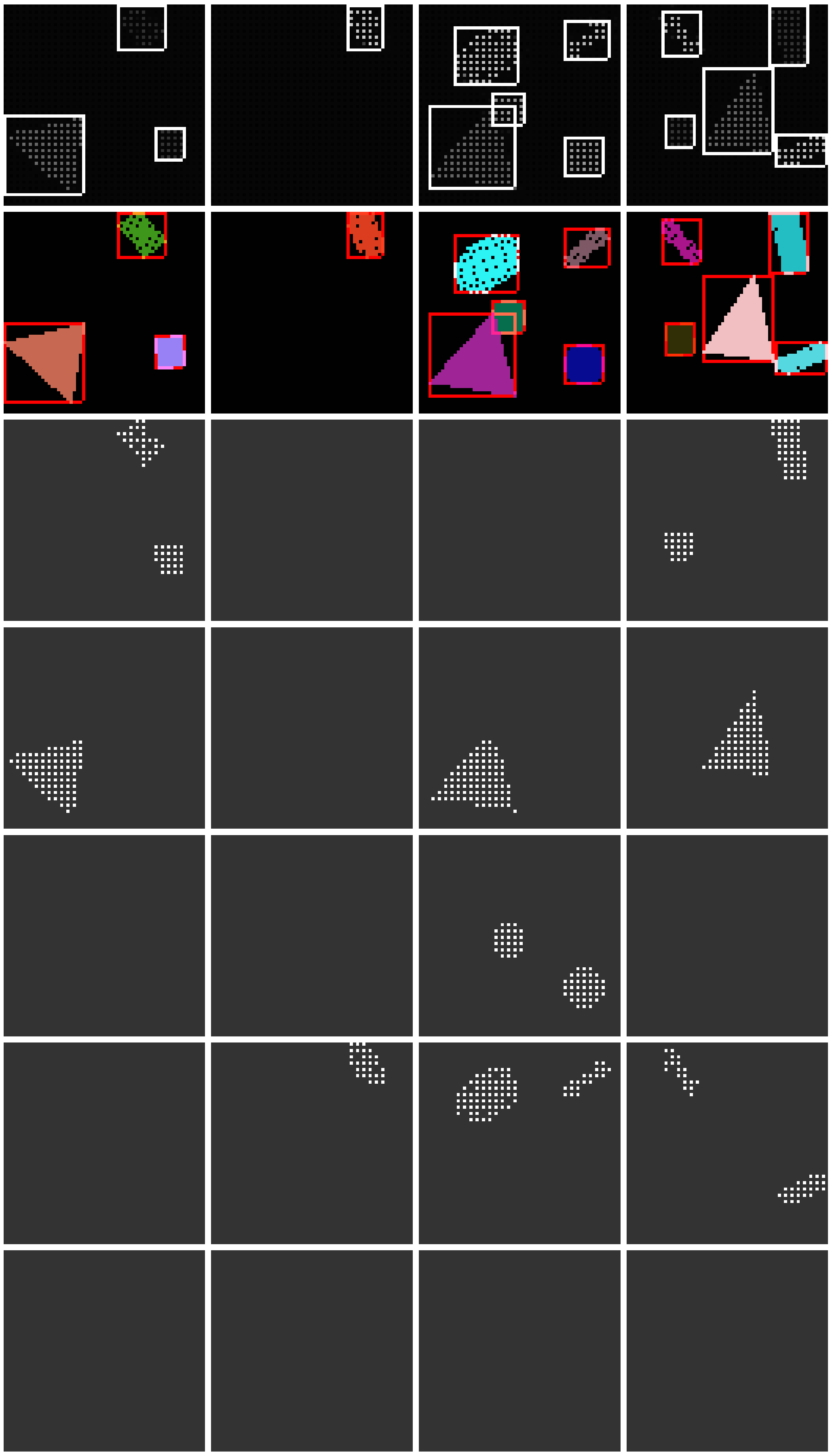


Figure 19: Result 3 for Purdue Shapes 5 Dataset

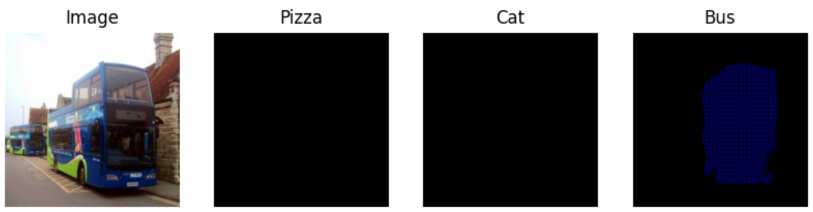


Figure 20: Result 1 for COCO Dataset

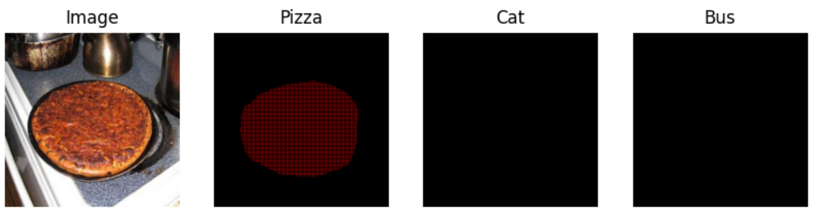


Figure 21: Result 2 for COCO Dataset

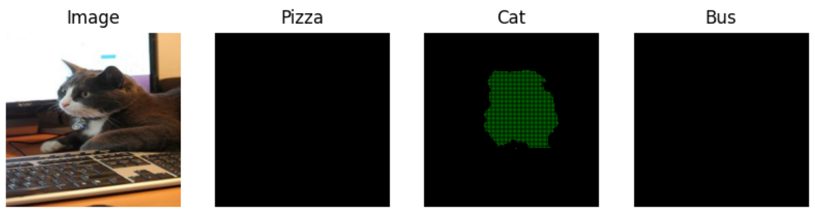


Figure 22: Result 3 for COCO Dataset

4.2 Dice Evaluation

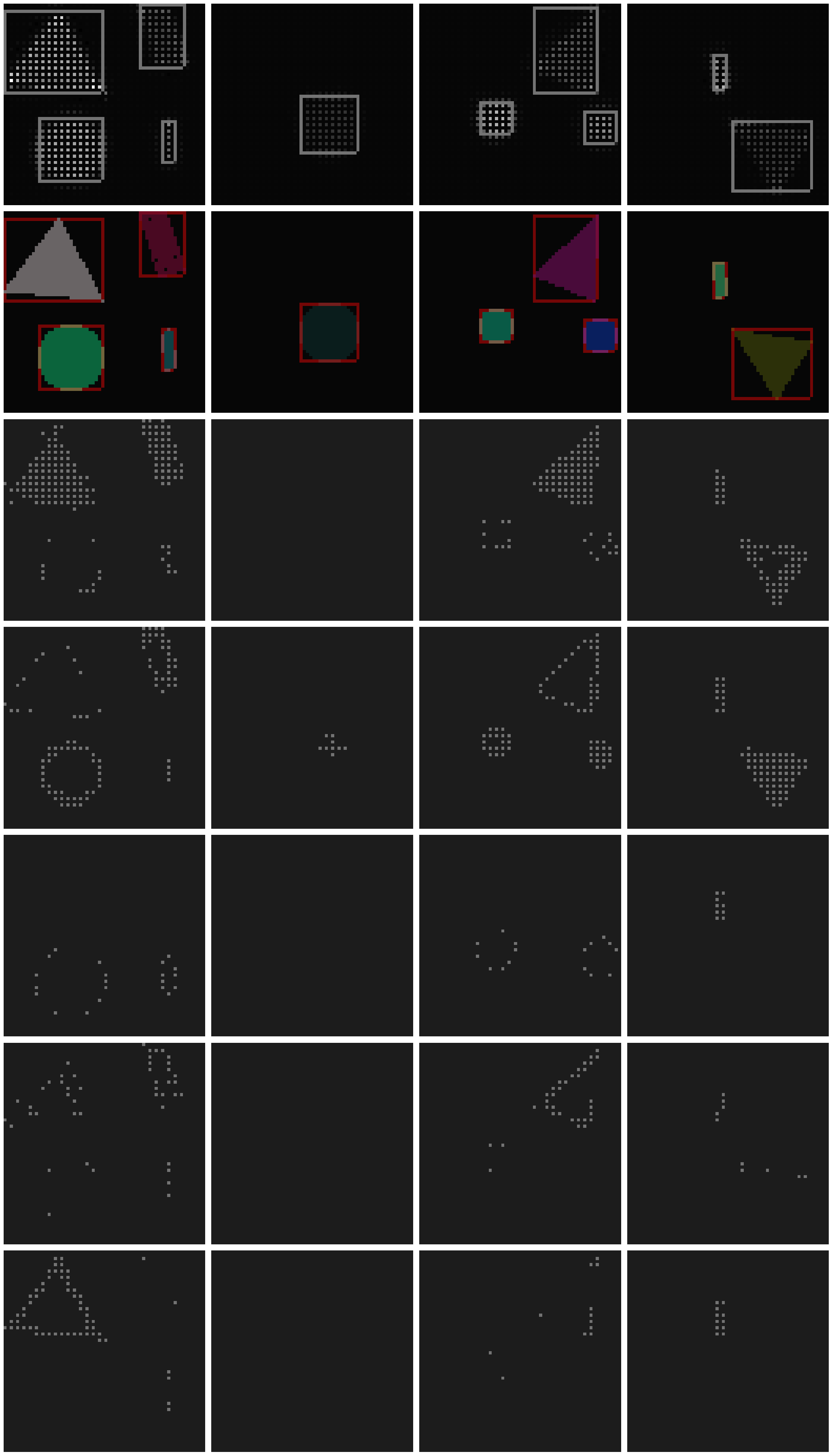


Figure 23: Result 1 for Purdue Shapes 5 Dataset

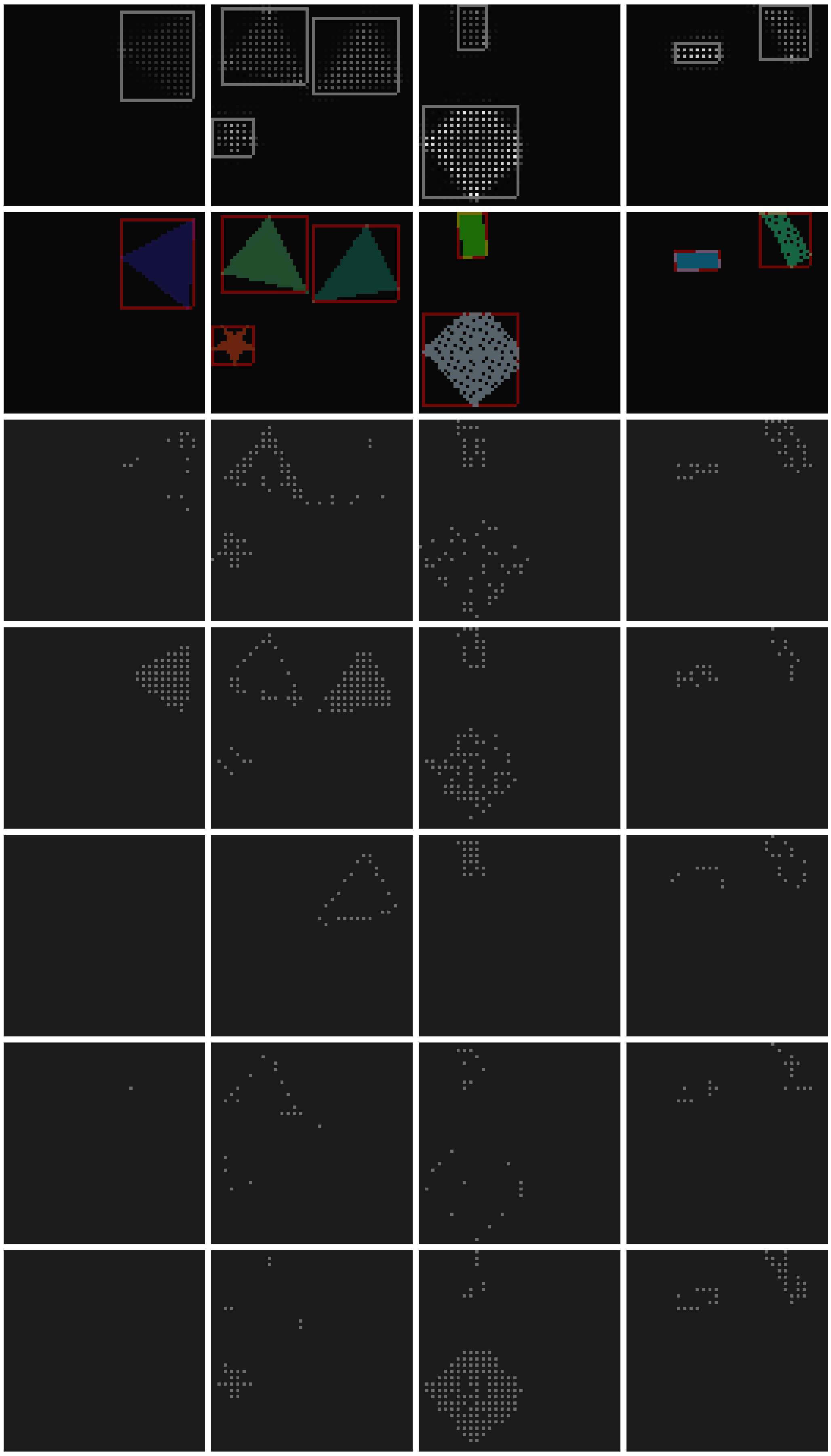


Figure 24: Result 2 for Purdue Shapes 5 Dataset

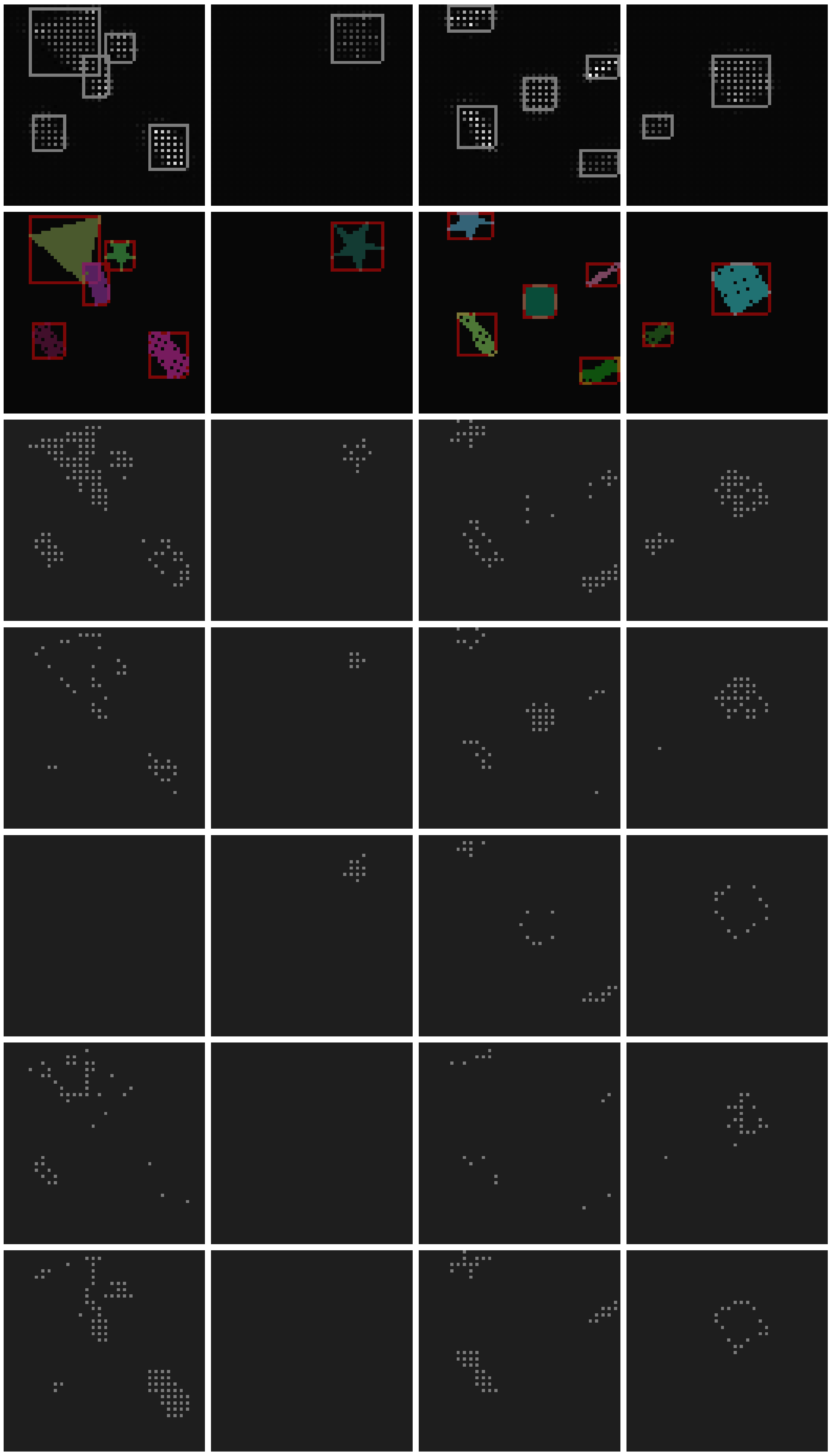


Figure 25: Result 3 for Purdue Shapes 5 Dataset

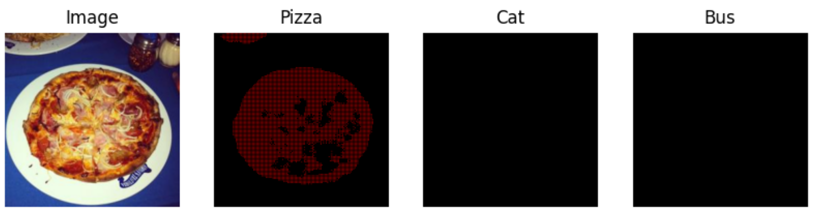


Figure 26: Result 1 for COCO Dataset

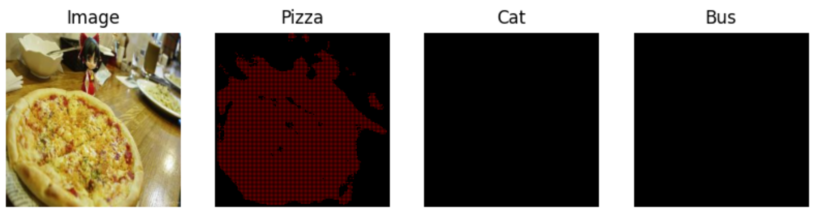


Figure 27: Result 2 for COCO Dataset

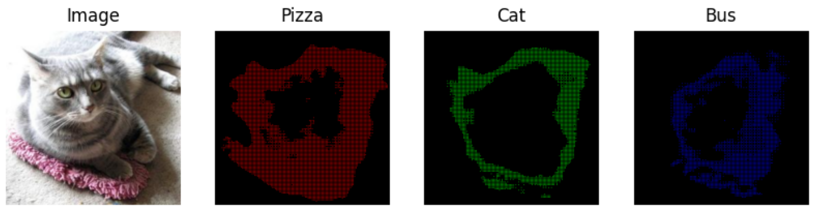


Figure 28: Result 3 for COCO Dataset

4.3 MSE + Dice Evaluation

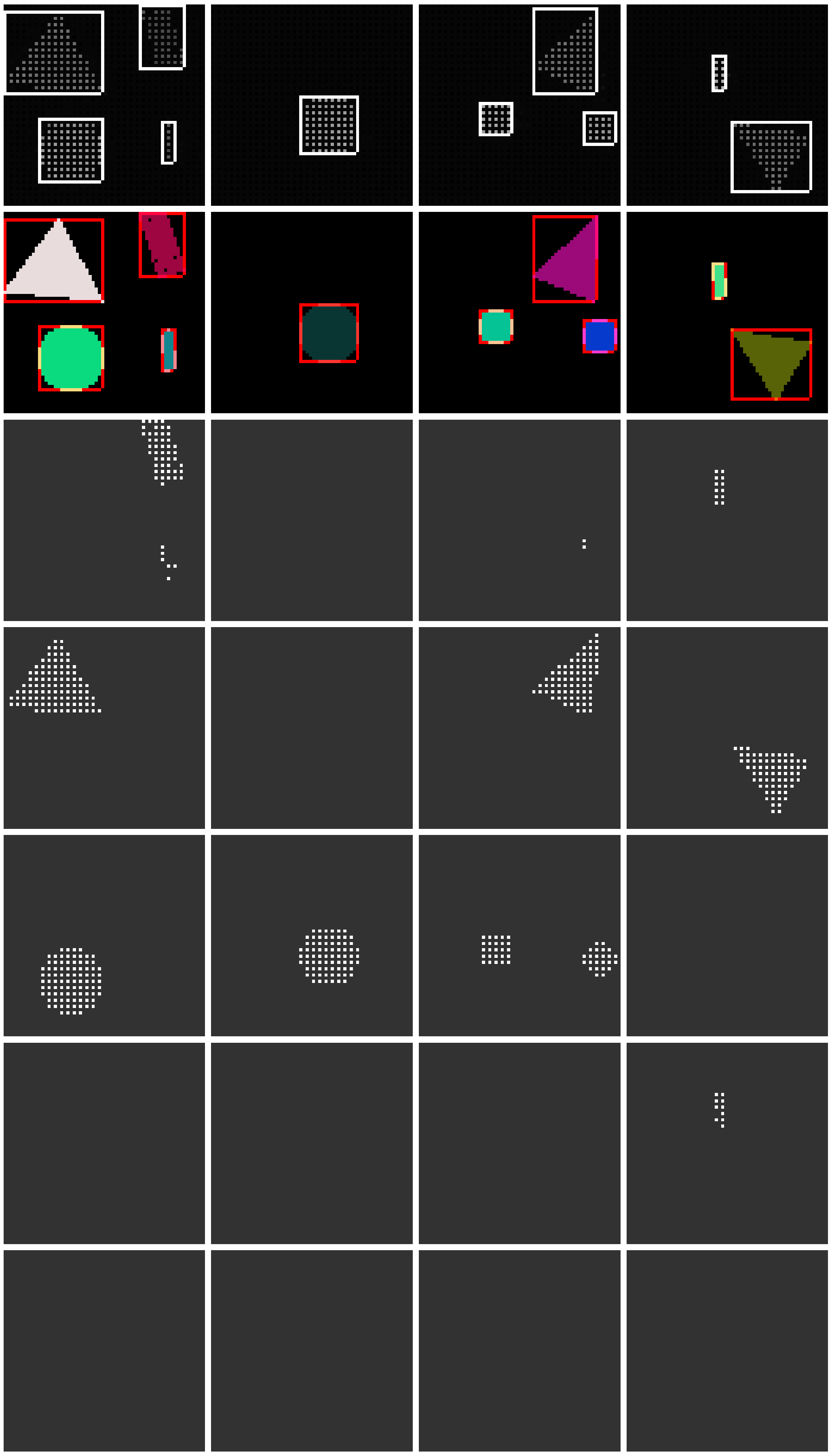


Figure 29: Result 1 for Purdue Shapes 5 Dataset

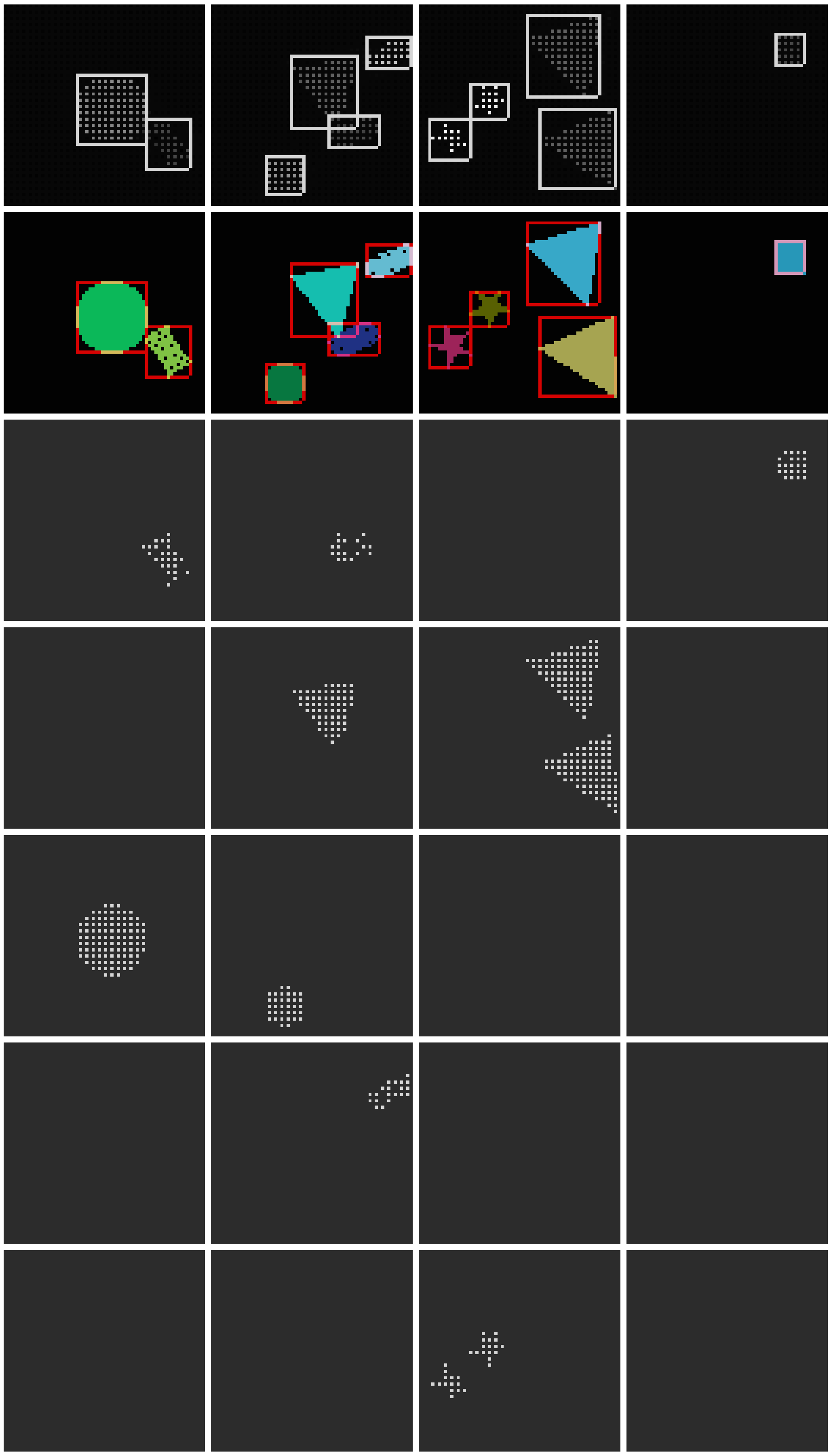


Figure 30: Result 2 for Purdue Shapes 5 Dataset

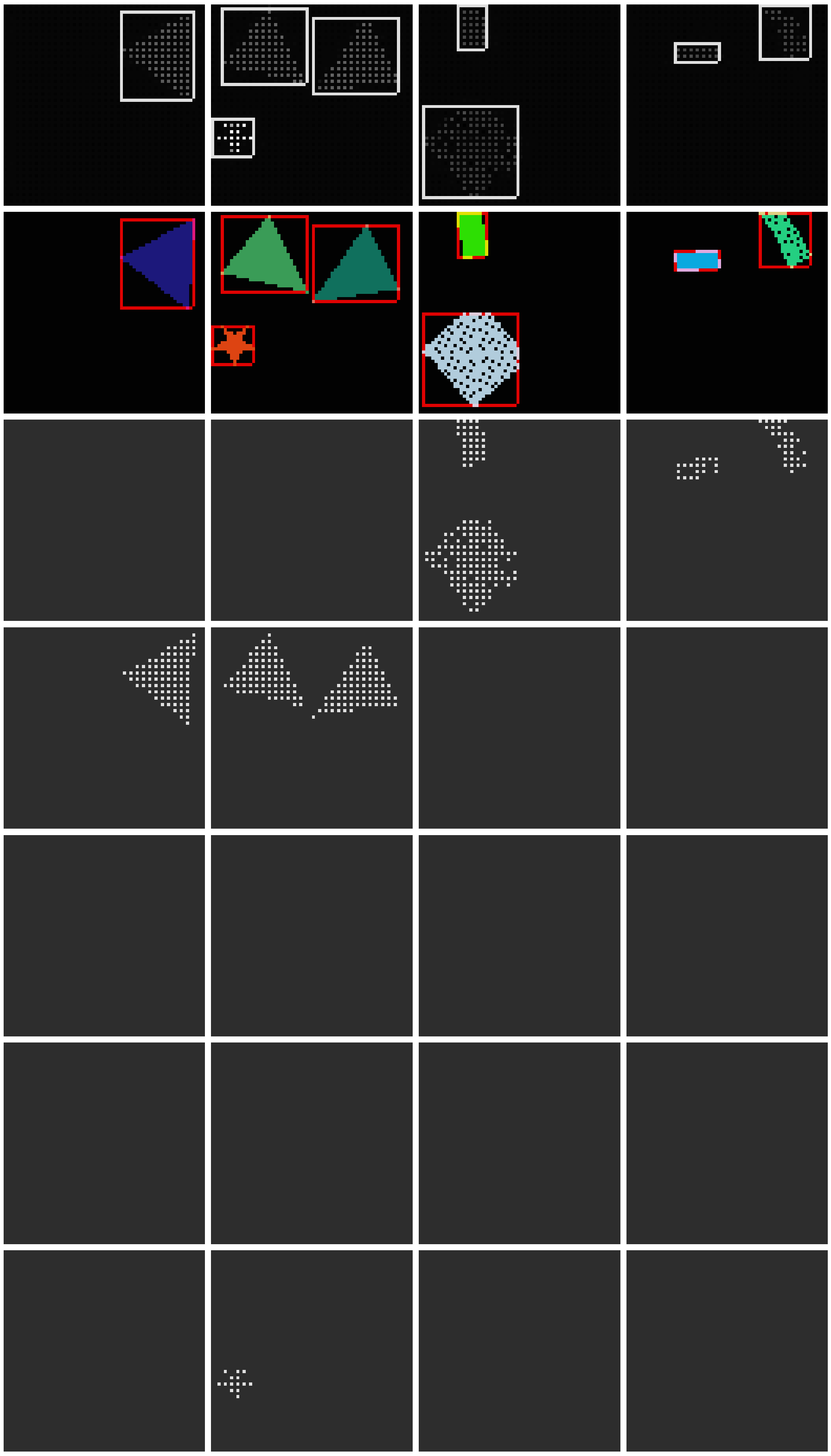


Figure 31: Result 3 for Purdue Shapes 5 Dataset

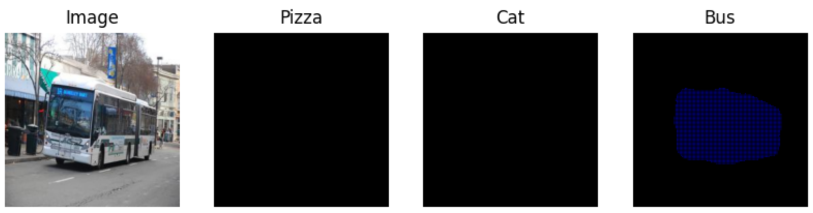


Figure 32: Result 1 for COCO Dataset

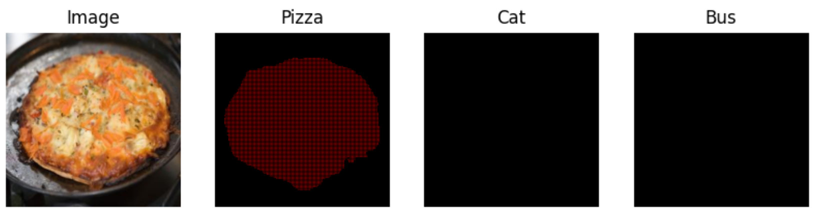


Figure 33: Result 2 for COCO Dataset

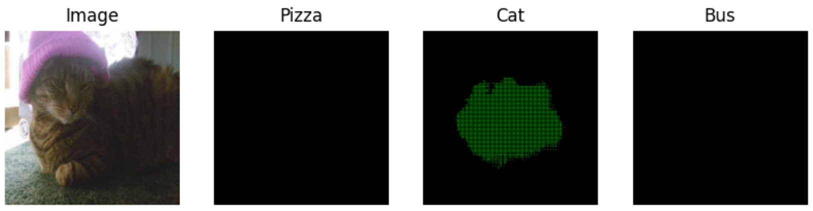


Figure 34: Result 3 for COCO Dataset

4.4 Observations

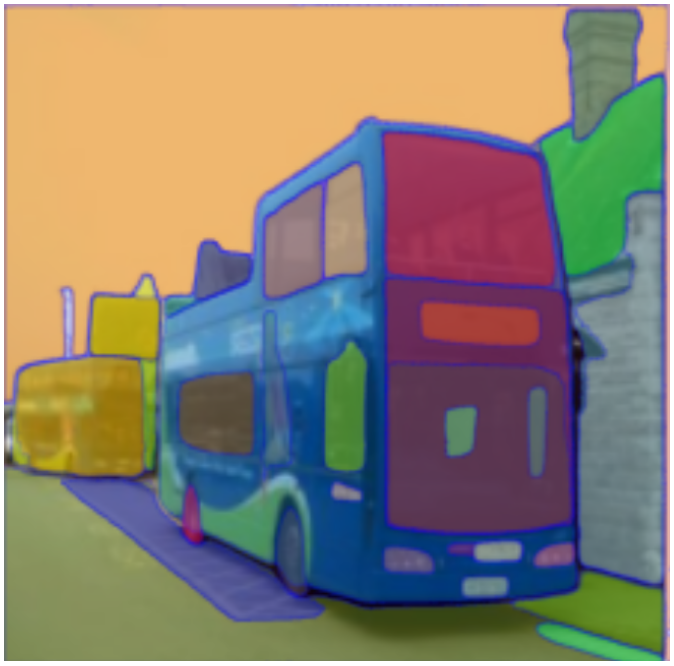
As we can see from the results above, a combination of MSE and Dice loss gives us the best segmentation. This performance if followed by only MSE loss and finally only Dice loss performs the worse.

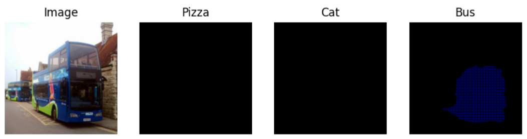
The combination of both losses likely performs the best because these losses help minimize two different factors. The dice loss focuses on the shape and overlap of the predicted mask whereas the MSE loss ensures pixel-wise accuracy. The addition of MSE loss to Dice loss helps stabilize it. The addition of MSE loss gives more stable gradients and helps with convergence. This combination also leads to less overfitting and allows the model to generalize well.

Dice loss likely performs the worst since in general it is a non-linear and non-convex function. Minimizing such a function is difficult and often times unstable. For this reason Dice loss is unable to perform really well.

5 Extra Credit

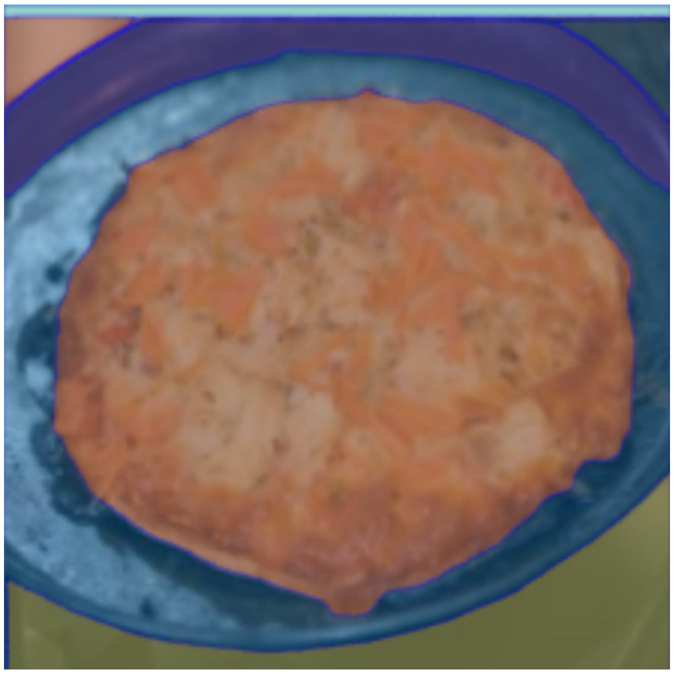
5.1 COCO Comparisons

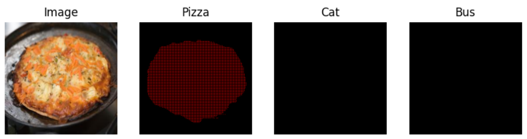

(a) Sam Output



(b) My Output

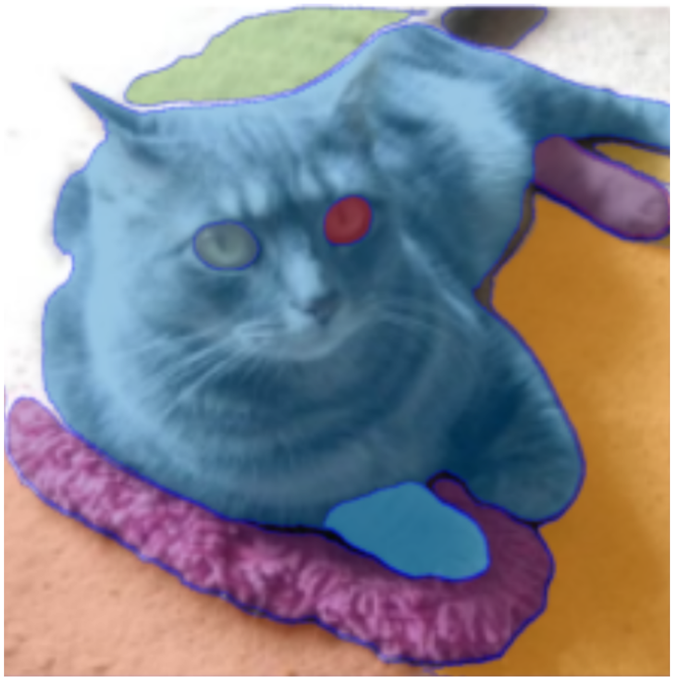
Figure 35: Comparisons

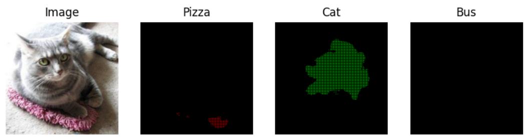

(a) Sam Output



(b) My Output

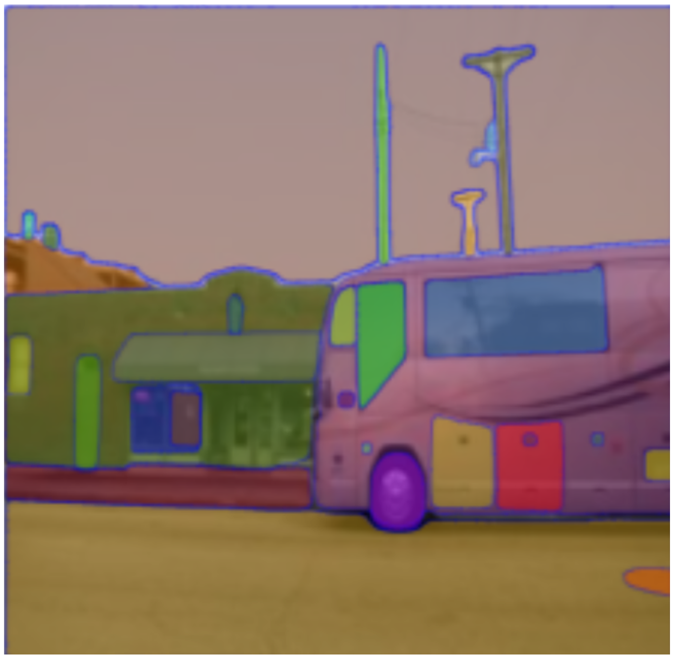
Figure 36: Comparisons

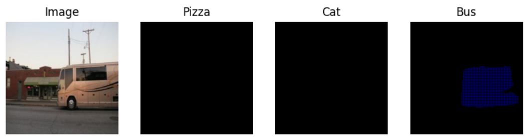

(a) Sam Output



(b) My Output

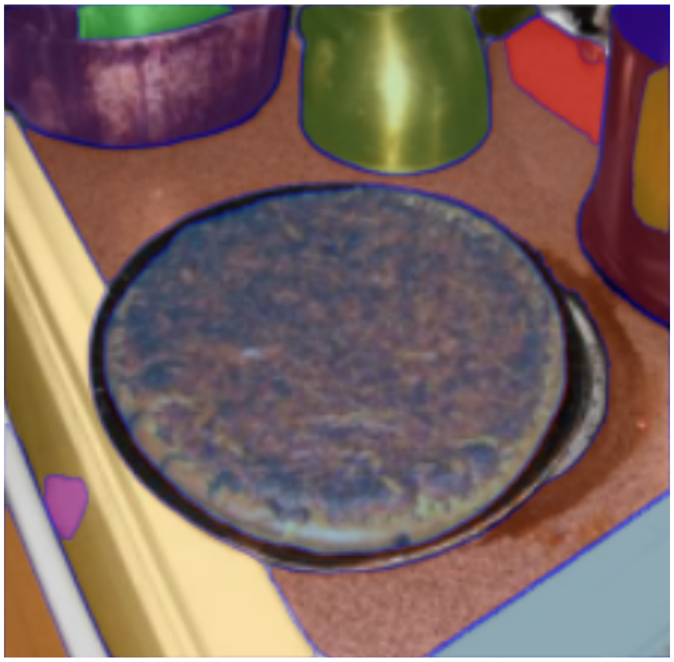
Figure 37: Comparisons

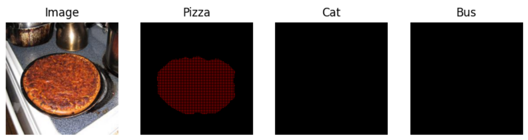

(a) Sam Output



(b) My Output

Figure 38: Comparisons


(a) Sam Output



(b) My Output

Figure 39: Comparisons

5.2 PurdueShapes5 Comparisons

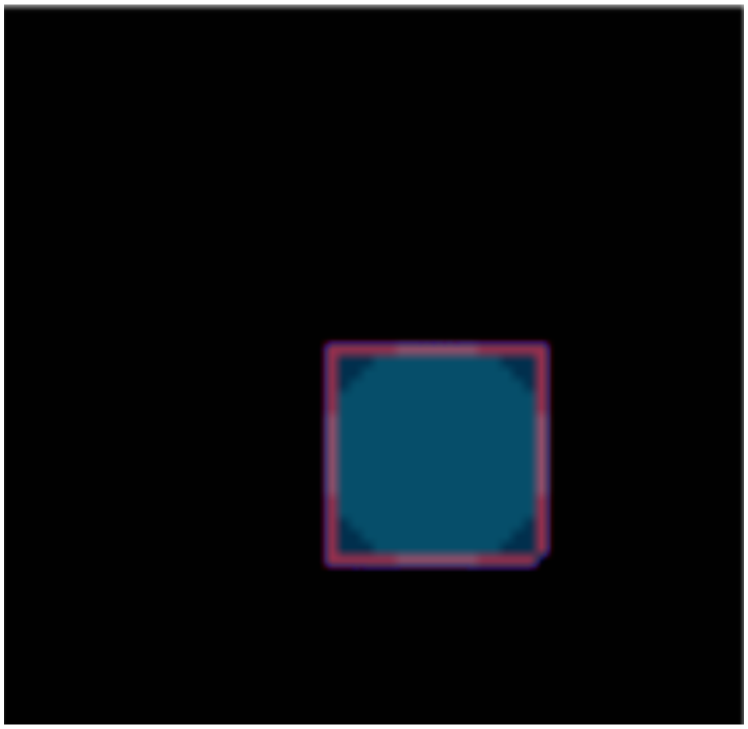


Figure 40: SAM Output 1

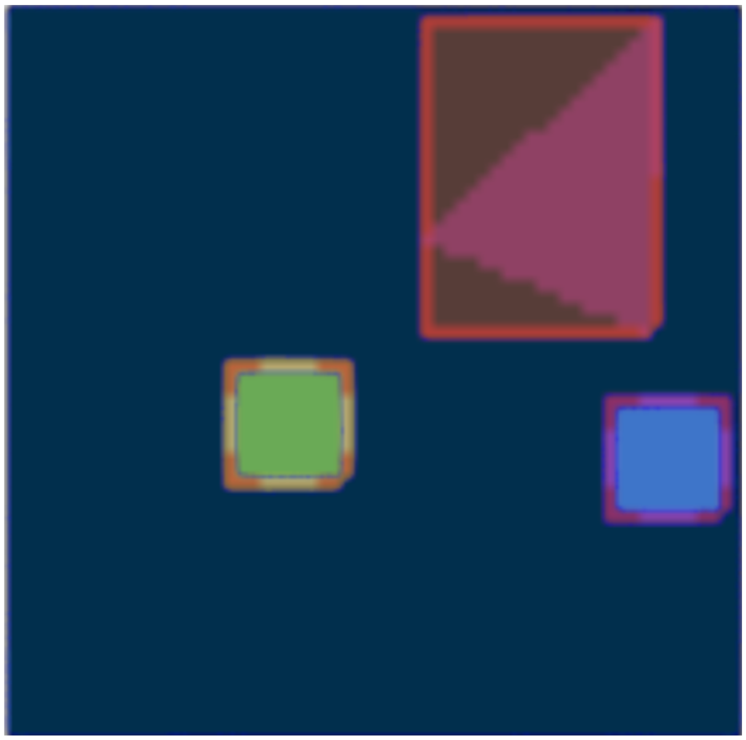


Figure 41: SAM Output 1

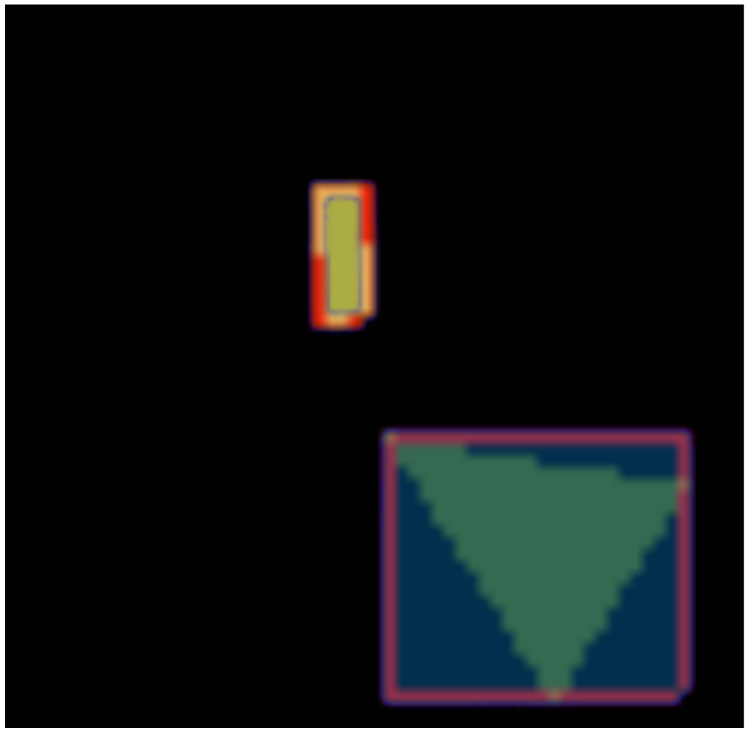


Figure 42: SAM Output 1

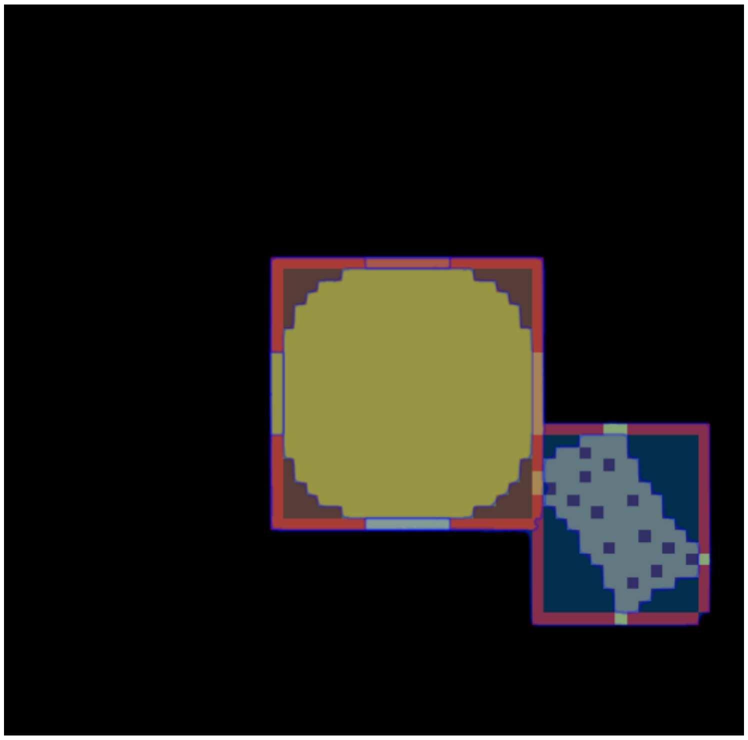


Figure 43: SAM Output 1

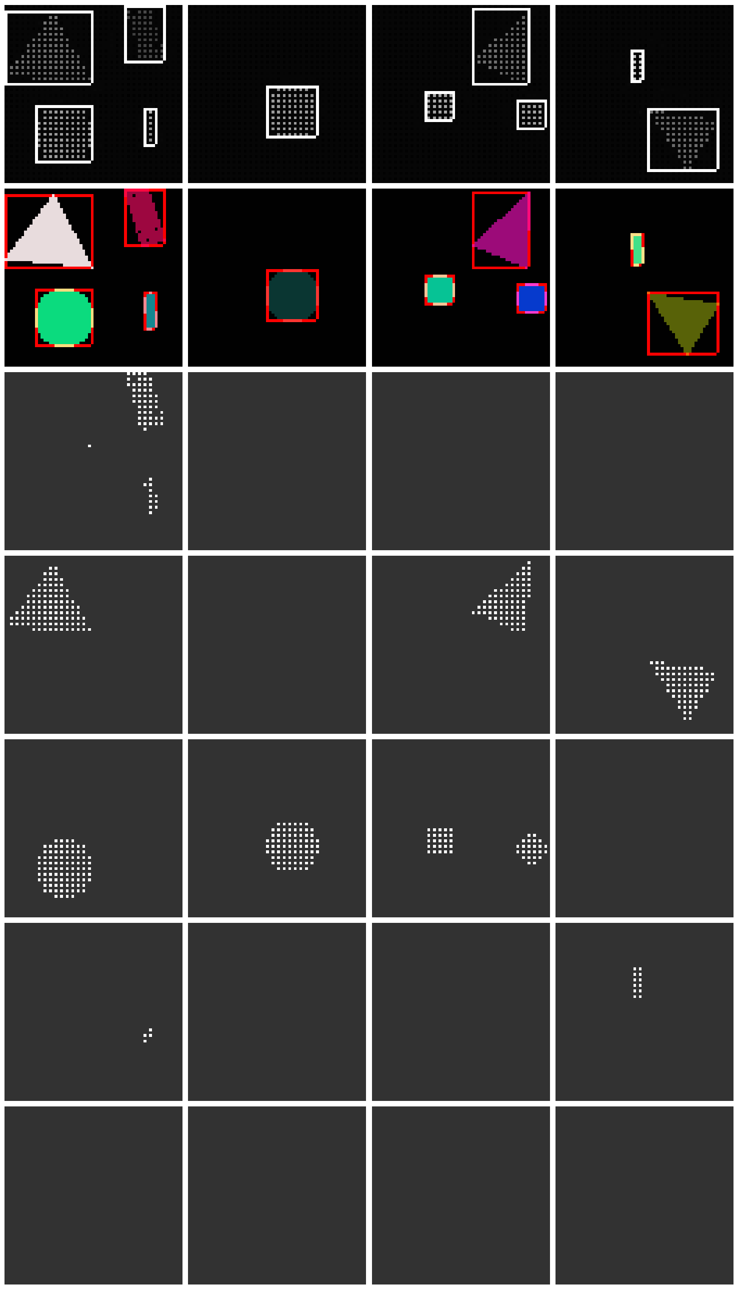


Figure 44: My Output for Image 1-4

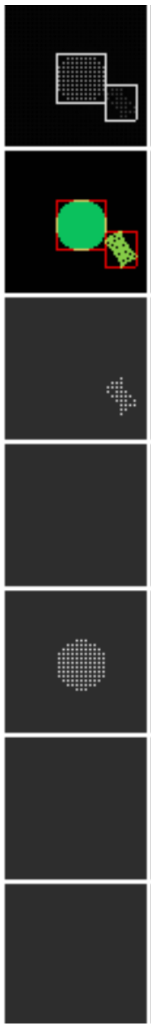


Figure 45: My Output for Image 5

5.3 Observations

When it comes to completeness, the SAM model far outperforms my implementation of mUnet. My model often only predicts the middle of the object, whereas SAM predicts the entire object most time.

On a similar note, SAM is able to accurately predict the boundary of most objects. My model on the other hand does not have smooth boundaries. SAM has smooth and complete boundaries separating different objects.

When it comes to false positives, my implementation of mUnet is conservative in predictions and consequently has a small number of false positives. Whereas, SAM is aggressive with its predictions and segmentation which sometimes leads to a few false positives or over segmentation. This is especially visible in the Bus images.