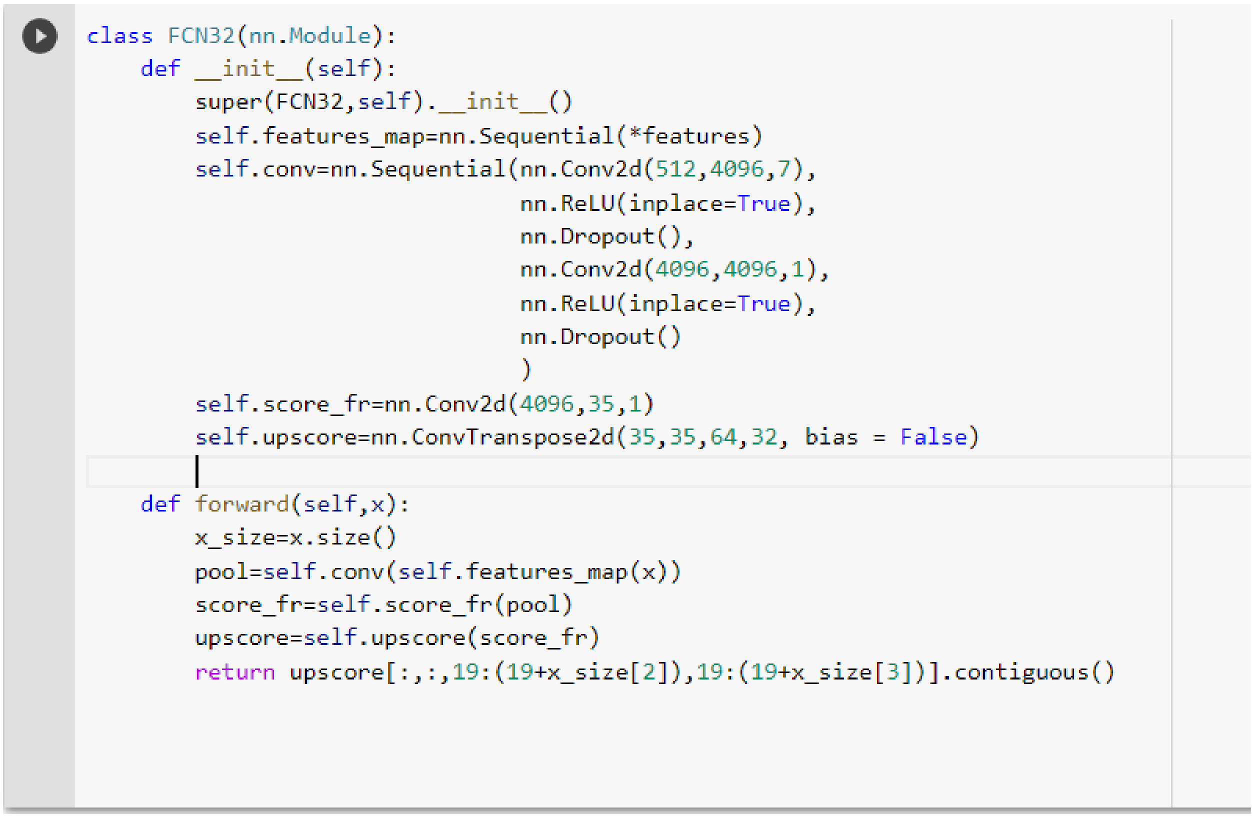
ASSIGNMENT 5

Submitted By: Charu Singh

Description of Code:

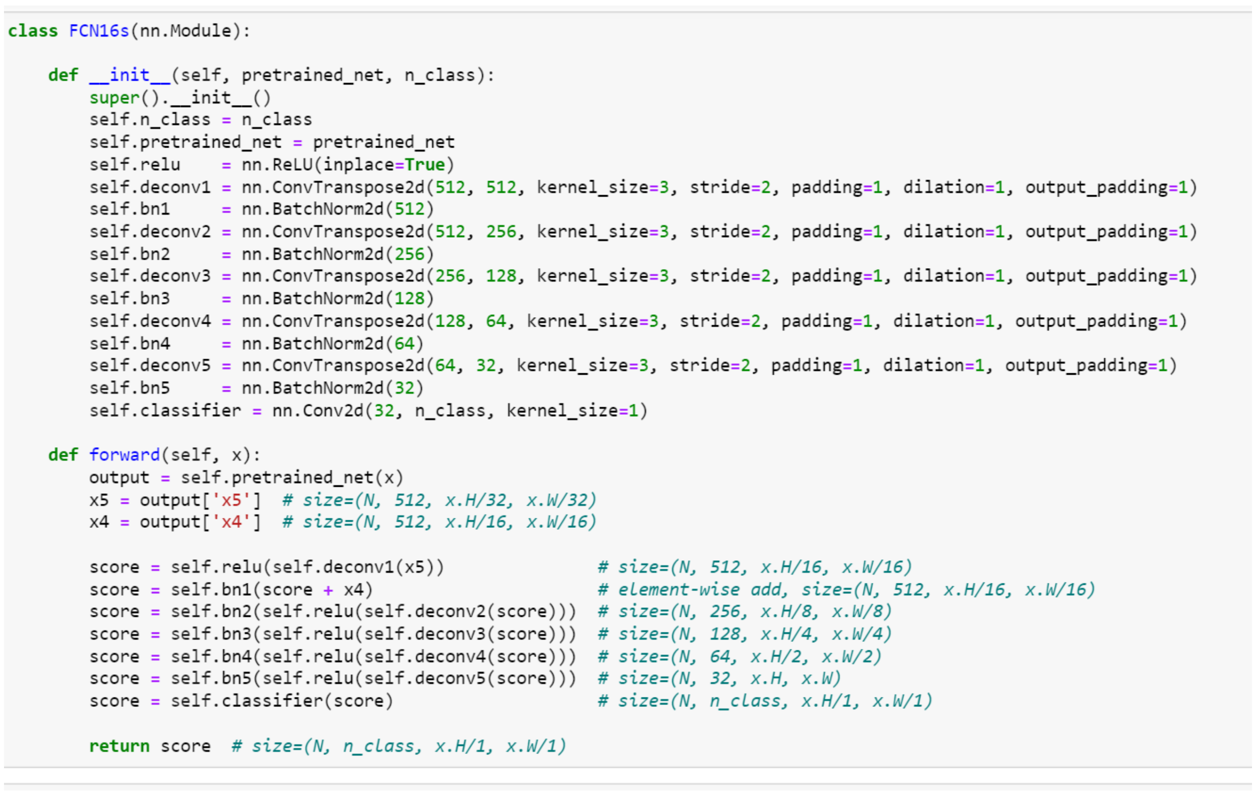
1. **FCN 32:**
   Define the architecture for FCN 32 which is built on pretrained model VGG16.
   We perform sampling by deconvolution (up sampling to get the output size larger).

   Convert classifier to FCN:
   VGG is known for its classification task. So, to make model suitable for dense prediction last FC layer is removed and is replaced with convolution. We append 1*1convolution with channel dimension 35 to predict scores for every class.



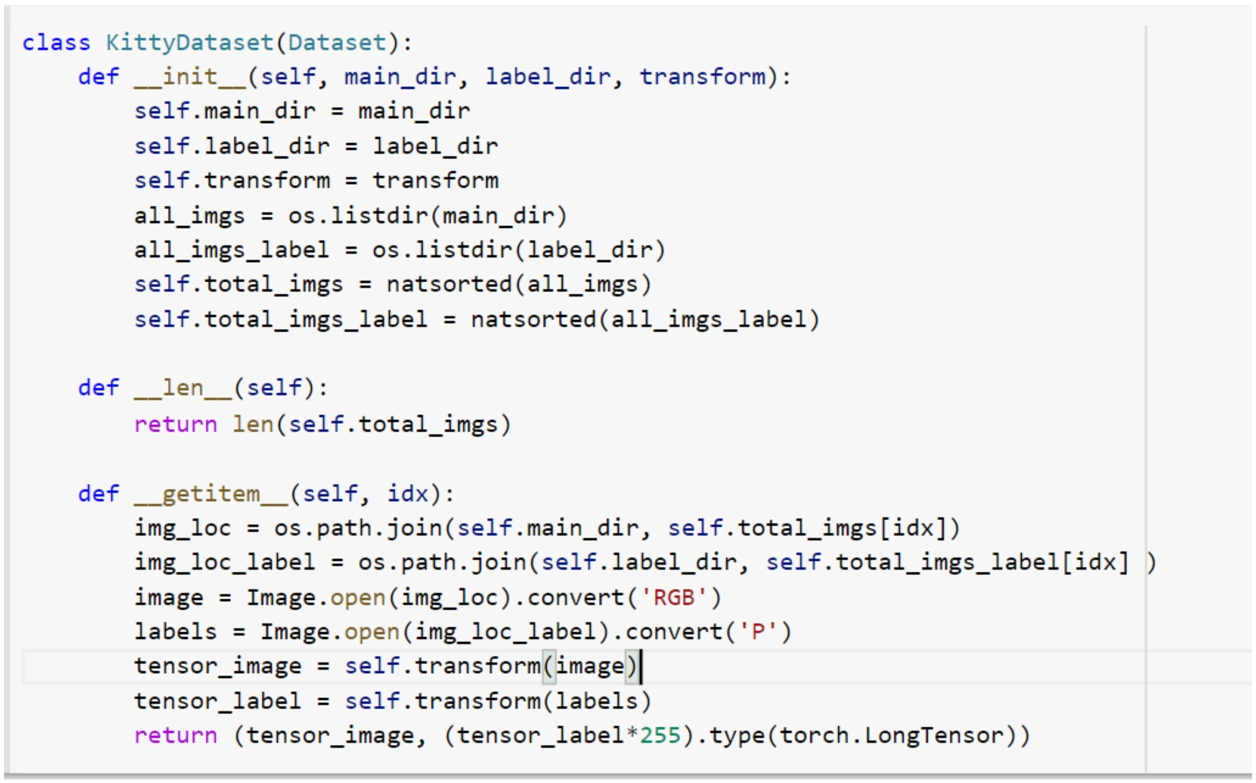
```
class FCN32(nn.Module):
    def __init__(self):
        super(FCN32,self).__init__()
        self.features_map=nn.Sequential(*features)
        self.conv=nn.Sequential(nn.Conv2d(512,4096,7),
                                nn.ReLU(inplace=True),
                                nn.Dropout(),
                                nn.Conv2d(4096,4096,1),
                                nn.ReLU(inplace=True),
                                nn.Dropout()
                                )
        self.score_fr=nn.Conv2d(4096,35,1)
        self.upscore=nn.ConvTranspose2d(35,35,64,32, bias = False)

    def forward(self,x):
        x_size=x.size()
        pool=self.conv(self.features_map(x))
        score_fr=self.score_fr(pool)
        upscore=self.upscore(score_fr)
        return upscore[:,:,19:(19+x_size[2]),19:(19+x_size[3])].contiguous()
```

Define FCN 16 architecture:



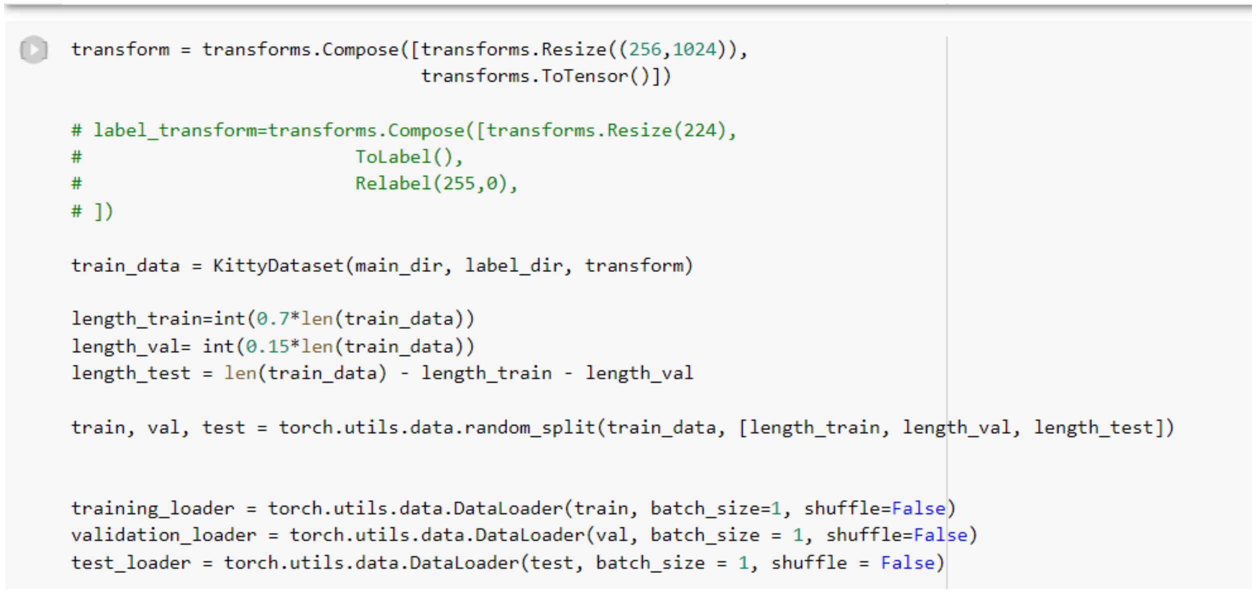
```
class FCN16s(nn.Module):

    def __init__(self, pretrained_net, n_class):
        super().__init__()
        self.n_class = n_class
        self.pretrained_net = pretrained_net
        self.relu    = nn.ReLU(inplace=True)
        self.deconv1 = nn.ConvTranspose2d(512, 512, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn1     = nn.BatchNorm2d(512)
        self.deconv2 = nn.ConvTranspose2d(512, 256, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn2     = nn.BatchNorm2d(256)
        self.deconv3 = nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn3     = nn.BatchNorm2d(128)
        self.deconv4 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn4     = nn.BatchNorm2d(64)
        self.deconv5 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn5     = nn.BatchNorm2d(32)
        self.classifier = nn.Conv2d(32, n_class, kernel_size=1)

    def forward(self, x):
        output = self.pretrained_net(x)
        x5 = output['x5']  # size=(N, 512, x.H/32, x.W/32)
        x4 = output['x4']  # size=(N, 512, x.H/16, x.W/16)

        score = self.relu(self.deconv1(x5))               # size=(N, 512, x.H/16, x.W/16)
        score = self.bn1(score + x4)                      # element-wise add, size=(N, 512, x.H/16, x.W/16)
        score = self.bn2(self.relu(self.deconv2(score)))  # size=(N, 256, x.H/8, x.W/8)
        score = self.bn3(self.relu(self.deconv3(score)))  # size=(N, 128, x.H/4, x.W/4)
        score = self.bn4(self.relu(self.deconv4(score)))  # size=(N, 64, x.H/2, x.W/2)
        score = self.bn5(self.relu(self.deconv5(score)))  # size=(N, 32, x.H, x.W)
        score = self.classifier(score)                    # size=(N, n_class, x.H/1, x.W/1)

        return score  # size=(N, n_class, x.H/1, x.W/1)
```

Download Kittydataset and apply transform on the images to get in the form tensor.

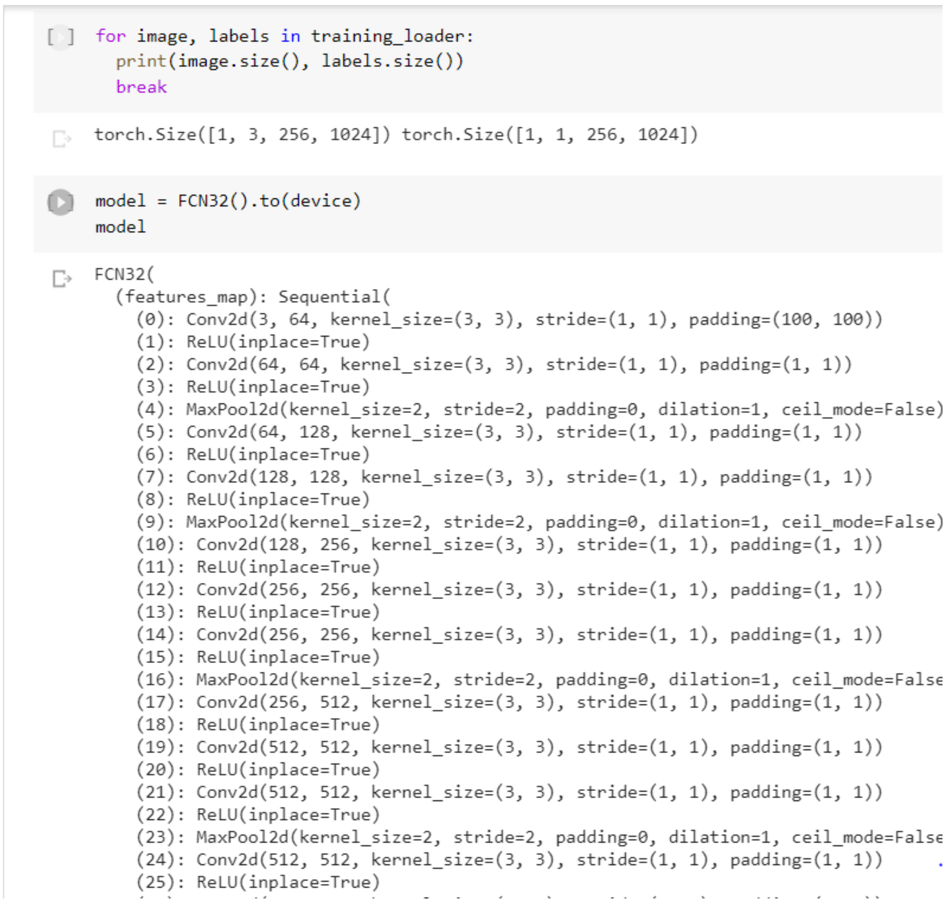


```python
class KittyDataset(Dataset):
    def __init__(self, main_dir, label_dir, transform):
        self.main_dir = main_dir
        self.label_dir = label_dir
        self.transform = transform
        all_imgs = os.listdir(main_dir)
        all_imgs_label = os.listdir(label_dir)
        self.total_imgs = natsorted(all_imgs)
        self.total_imgs_label = natsorted(all_imgs_label)

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

    def __getitem__(self, idx):
        img_loc = os.path.join(self.main_dir, self.total_imgs[idx])
        img_loc_label = os.path.join(self.label_dir, self.total_imgs_label[idx] )
        image = Image.open(img_loc).convert('RGB')
        labels = Image.open(img_loc_label).convert('P')
        tensor_image = self.transform(image)
        tensor_label = self.transform(labels)
        return (tensor_image, (tensor_label*255).type(torch.LongTensor))
```

Split your data into training, testing and validation.



```python
transform = transforms.Compose([transforms.Resize((256,1024)),
                                transforms.ToTensor()])

# label_transform=transforms.Compose([transforms.Resize(224),
#                          ToLabel(),
#                          Relabel(255,0),
# ])

train_data = KittyDataset(main_dir, label_dir, transform)

length_train=int(0.7*len(train_data))
length_val= int(0.15*len(train_data))
length_test = len(train_data) - length_train - length_val

train, val, test = torch.utils.data.random_split(train_data, [length_train, length_val, length_test])


training_loader = torch.utils.data.DataLoader(train, batch_size=1, shuffle=False)
validation_loader = torch.utils.data.DataLoader(val, batch_size = 1, shuffle=False)
test_loader = torch.utils.data.DataLoader(test, batch_size = 1, shuffle = False)
```



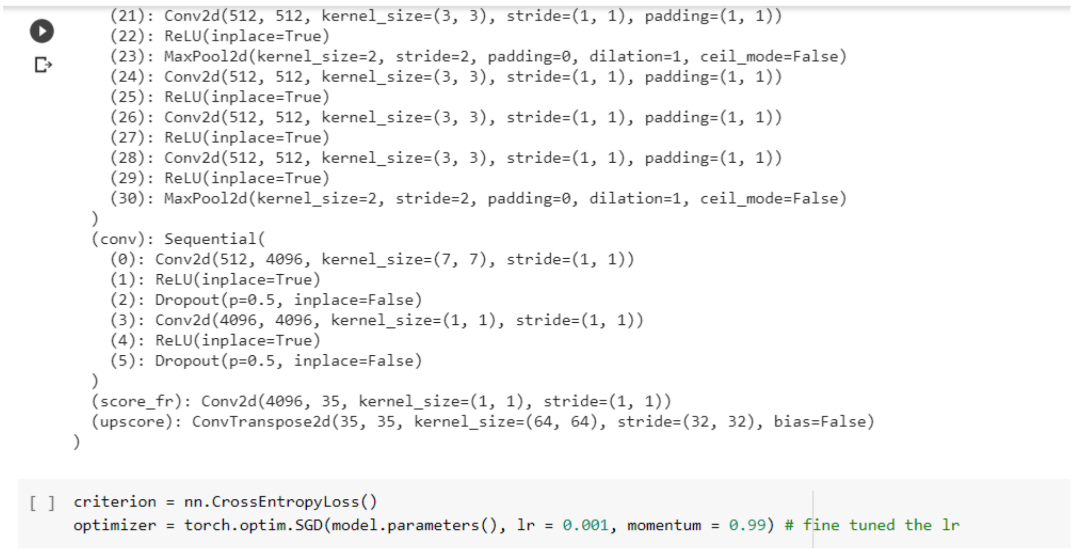
```
for image, labels in training_loader:
  print(image.size(), labels.size())
  break
```

```
torch.Size([1, 3, 256, 1024]) torch.Size([1, 1, 256, 1024])
```

```
model = FCN32().to(device)
model
```

```
FCN32(
  (features_map): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(100, 100))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
```

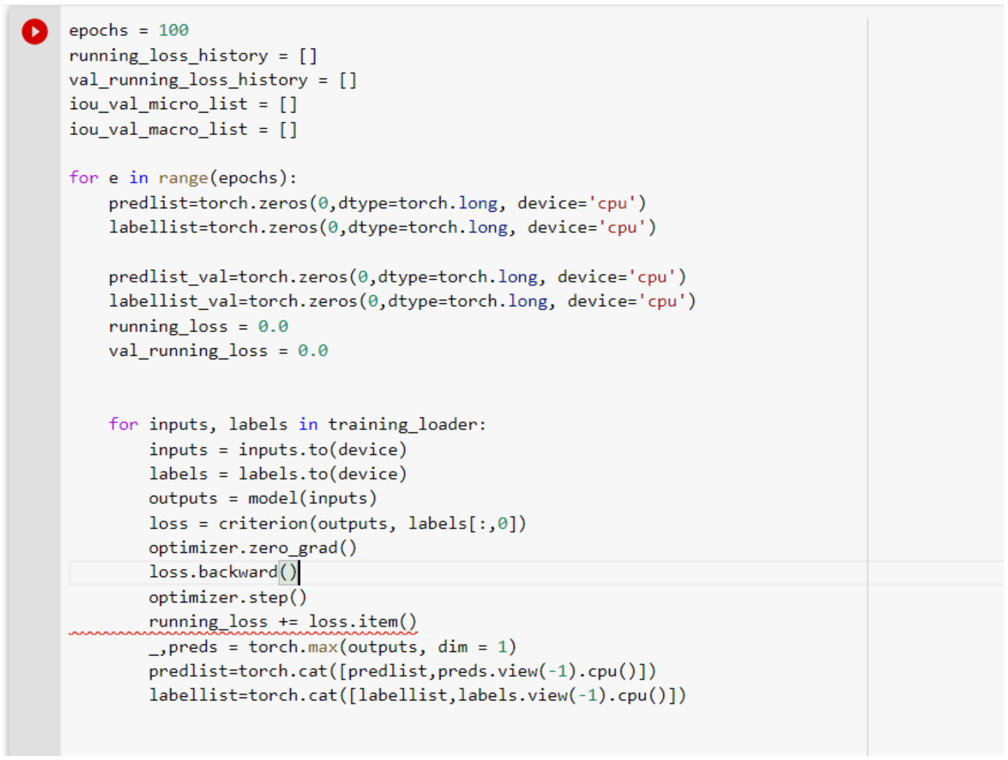
Define the loss function: Each output pixel of the network is compared to its corresponding ground truth segmentation image. Standard cross entropy is applied on each pixel.



```
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv): Sequential(
    (0): Conv2d(512, 4096, kernel_size=(7, 7), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Conv2d(4096, 4096, kernel_size=(1, 1), stride=(1, 1))
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
  )
  (score_fr): Conv2d(4096, 35, kernel_size=(1, 1), stride=(1, 1))
  (upscore): ConvTranspose2d(35, 35, kernel_size=(64, 64), stride=(32, 32), bias=False)
)
```

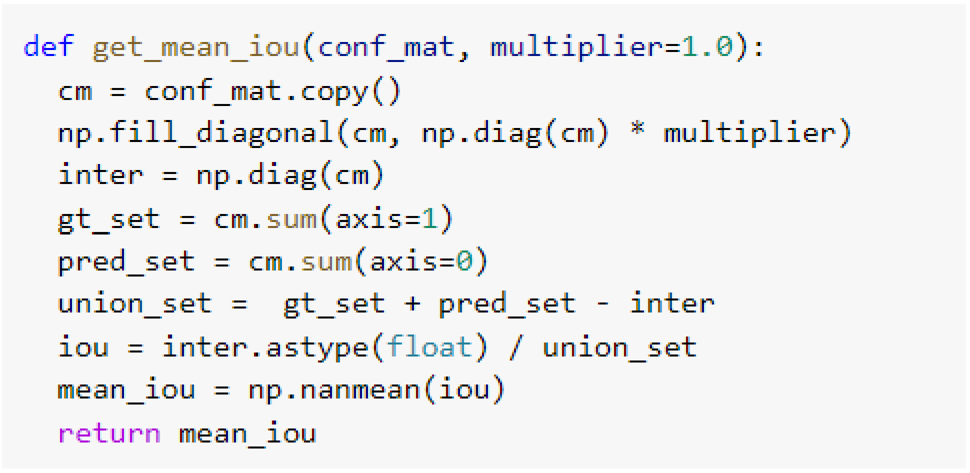
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.99) # fine tuned the lr
```

Training and validation loop:

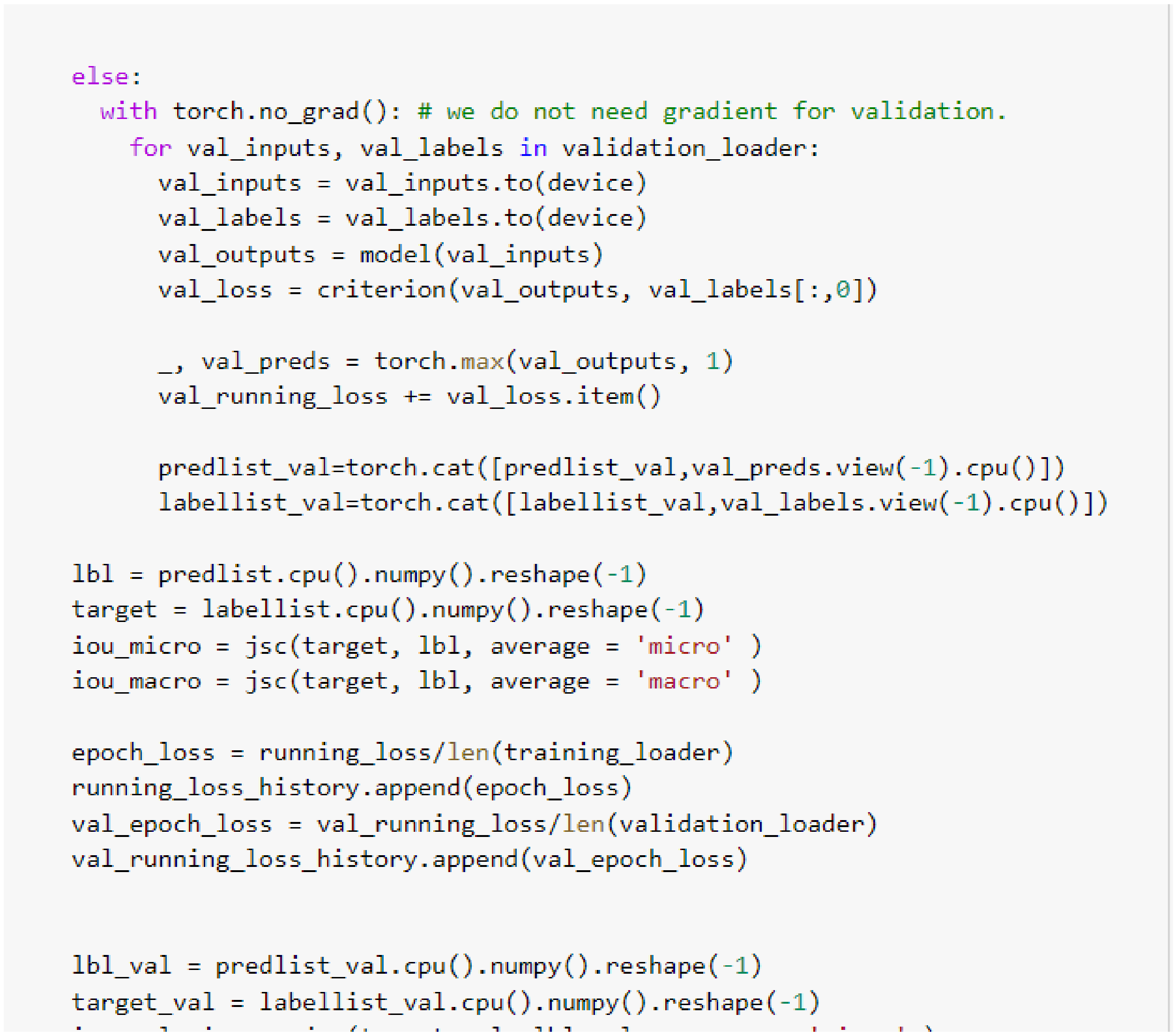


```
epochs = 100
running_loss_history = []
val_running_loss_history = []
iou_val_micro_list = []
iou_val_macro_list = []

for e in range(epochs):
    predlist=torch.zeros(0,dtype=torch.long, device='cpu')
    labellist=torch.zeros(0,dtype=torch.long, device='cpu')

    predlist_val=torch.zeros(0,dtype=torch.long, device='cpu')
    labellist_val=torch.zeros(0,dtype=torch.long, device='cpu')
    running_loss = 0.0
    val_running_loss = 0.0


    for inputs, labels in training_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels[:,0])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _,preds = torch.max(outputs, dim = 1)
        predlist=torch.cat([predlist,preds.view(-1).cpu()])
        labellist=torch.cat([labellist,labels.view(-1).cpu()])
```

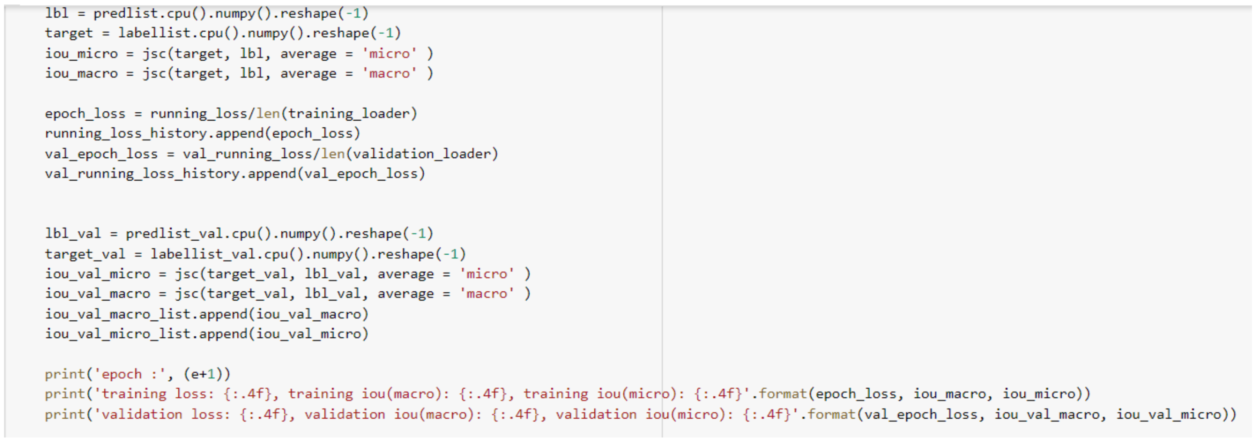
Calculate IOU at each iteration:



```
def get_mean_iou(conf_mat, multiplier=1.0):
  cm = conf_mat.copy()
  np.fill_diagonal(cm, np.diag(cm) * multiplier)
  inter = np.diag(cm)
  gt_set = cm.sum(axis=1)
  pred_set = cm.sum(axis=0)
  union_set =  gt_set + pred_set - inter
  iou = inter.astype(float) / union_set
  mean_iou = np.nanmean(iou)
  return mean_iou
```



```
else:
  with torch.no_grad(): # we do not need gradient for validation.
    for val_inputs, val_labels in validation_loader:
      val_inputs = val_inputs.to(device)
      val_labels = val_labels.to(device)
      val_outputs = model(val_inputs)
      val_loss = criterion(val_outputs, val_labels[:,0])

      _, val_preds = torch.max(val_outputs, 1)
      val_running_loss += val_loss.item()

      predlist_val=torch.cat([predlist_val,val_preds.view(-1).cpu()])
      labellist_val=torch.cat([labellist_val,val_labels.view(-1).cpu()])

lbl = predlist.cpu().numpy().reshape(-1)
target = labellist.cpu().numpy().reshape(-1)
iou_micro = jsc(target, lbl, average = 'micro' )
iou_macro = jsc(target, lbl, average = 'macro' )

epoch_loss = running_loss/len(training_loader)
running_loss_history.append(epoch_loss)
val_epoch_loss = val_running_loss/len(validation_loader)
val_running_loss_history.append(val_epoch_loss)


lbl_val = predlist_val.cpu().numpy().reshape(-1)
target_val = labellist_val.cpu().numpy().reshape(-1)
```



```
lbl = predlist.cpu().numpy().reshape(-1)
target = labellist.cpu().numpy().reshape(-1)
iou_micro = jsc(target, lbl, average = 'micro' )
iou_macro = jsc(target, lbl, average = 'macro' )

epoch_loss = running_loss/len(training_loader)
running_loss_history.append(epoch_loss)
val_epoch_loss = val_running_loss/len(validation_loader)
val_running_loss_history.append(val_epoch_loss)


lbl_val = predlist_val.cpu().numpy().reshape(-1)
target_val = labellist_val.cpu().numpy().reshape(-1)
```

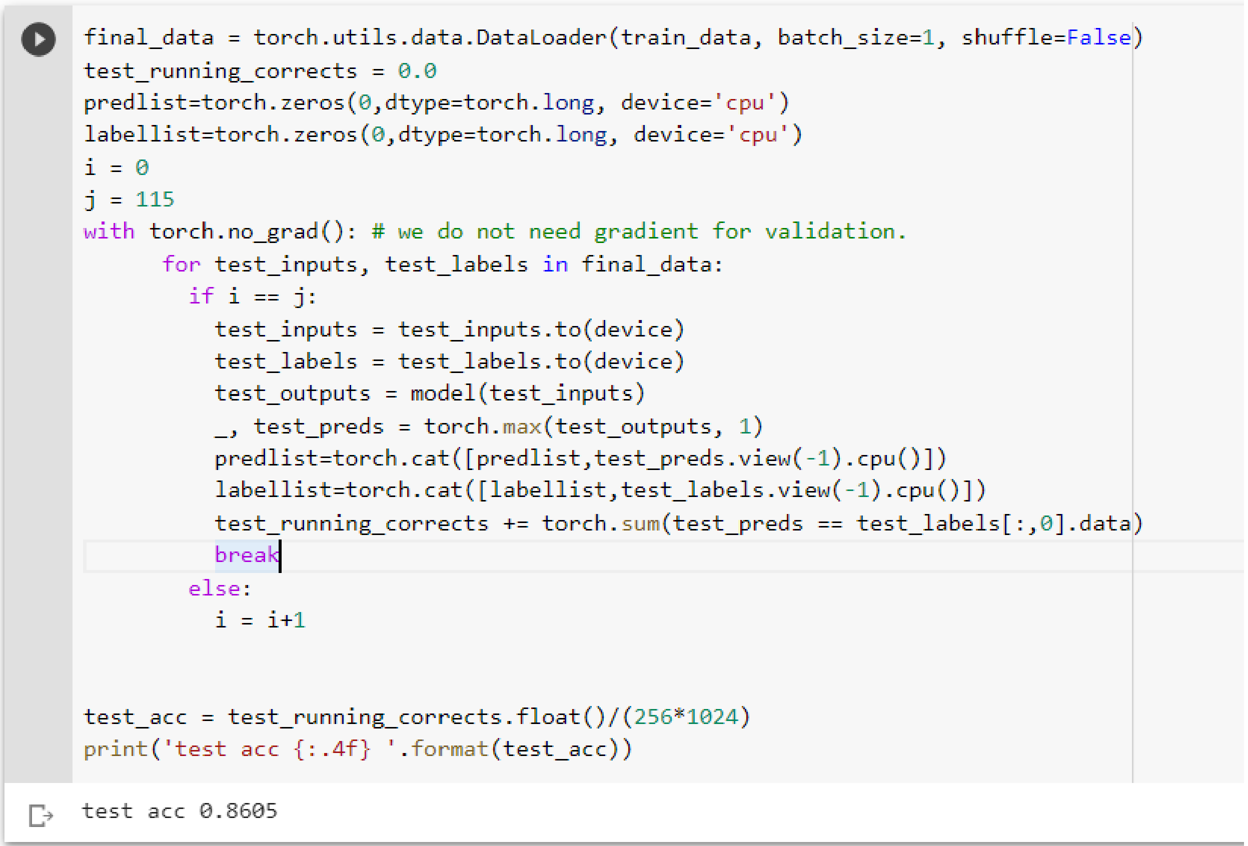
```
iou_val_micro = jsc(target_val, lbl_val, average = 'micro' )
iou_val_macro = jsc(target_val, lbl_val, average = 'macro' )
iou_val_macro_list.append(iou_val_macro)
iou_val_micro_list.append(iou_val_micro)

print('epoch :', (e+1))
print('training loss: {:.4f}, training iou(macro): {:.4f}, training iou(micro): {:.4f}'.format(epoch_loss, iou_macro, iou_micro))
print('validation loss: {:.4f}, validation iou(macro): {:.4f}, validation iou(micro): {:.4f}'.format(val_epoch_loss, iou_val_macro, iou_val_micro))
```

```
[ ]  test_running_corrects = 0.0
     predlist=torch.zeros(0,dtype=torch.long, device='cpu')
     labellist=torch.zeros(0,dtype=torch.long, device='cpu')

     with torch.no_grad(): # we do not need gradient for validation.
           for test_inputs, test_labels in test_loader:
             test_inputs = test_inputs.to(device)
             test_labels = test_labels.to(device)
             test_outputs = model(test_inputs)
             _, test_preds = torch.max(test_outputs, 1)
             predlist=torch.cat([predlist,test_preds.view(-1).cpu()])
             labellist=torch.cat([labellist,test_labels.view(-1).cpu()])
             test_running_corrects += torch.sum(test_preds == test_labels[:,0].data)


     test_acc = test_running_corrects.float()/ (len(test_loader)*1024*256)
     print('test acc {:.4f} '.format(test_acc))
```

```
test acc 0.8457
```

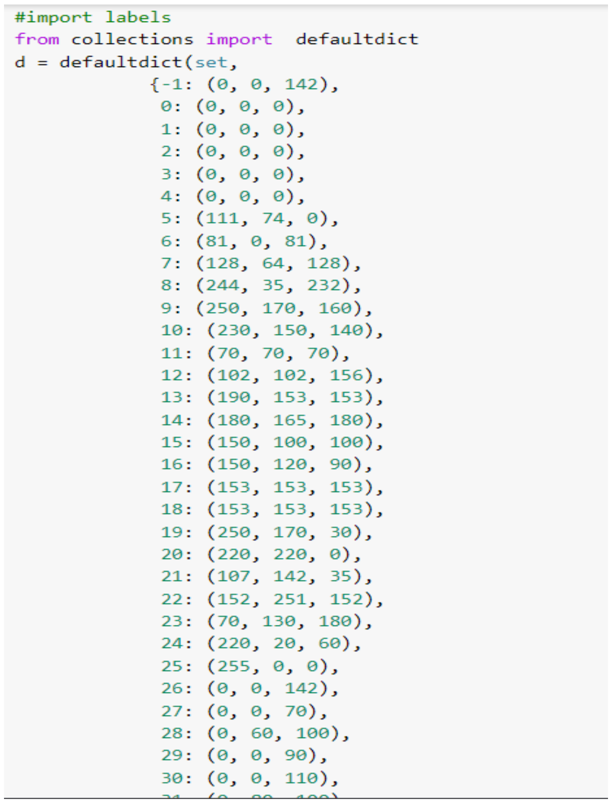
```
[ ]  from sklearn.metrics import jaccard_score as jsc

     lbl = predlist.cpu().numpy().reshape(-1)
     target = labellist.cpu().numpy().reshape(-1)
     print(jsc(target, lbl, average = 'micro' ))
```

```
0.7326034546231245
```



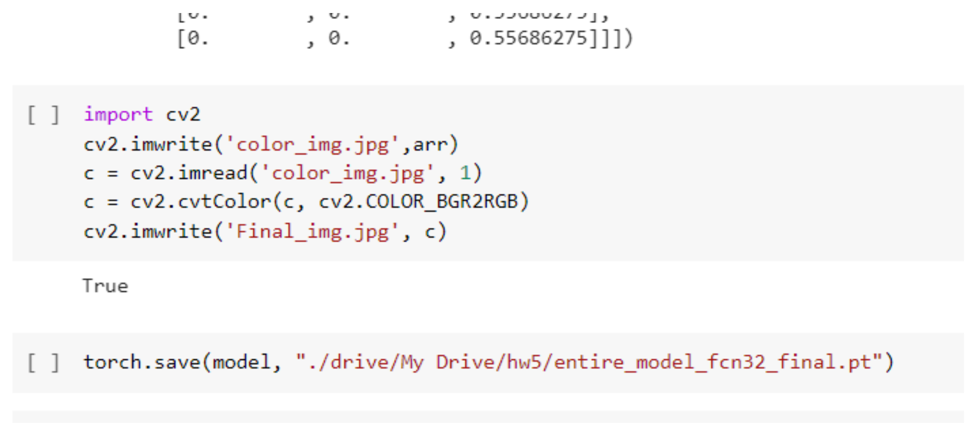
```
final_data = torch.utils.data.DataLoader(train_data, batch_size=1, shuffle=False)
test_running_corrects = 0.0
predlist=torch.zeros(0,dtype=torch.long, device='cpu')
labellist=torch.zeros(0,dtype=torch.long, device='cpu')
i = 0
j = 115
with torch.no_grad(): # we do not need gradient for validation.
      for test_inputs, test_labels in final_data:
        if i == j:
          test_inputs = test_inputs.to(device)
          test_labels = test_labels.to(device)
          test_outputs = model(test_inputs)
          _, test_preds = torch.max(test_outputs, 1)
          predlist=torch.cat([predlist,test_preds.view(-1).cpu()])
          labellist=torch.cat([labellist,test_labels.view(-1).cpu()])
          test_running_corrects += torch.sum(test_preds == test_labels[:,0].data)
          break
        else:
          i = i+1


test_acc = test_running_corrects.float()/(256*1024)
print('test acc {:.4f} '.format(test_acc))
```

```
test acc 0.8605
```



```
#import labels
from collections import  defaultdict
d = defaultdict(set,
            {-1: (0, 0, 142),
             0: (0, 0, 0),
             1: (0, 0, 0),
             2: (0, 0, 0),
             3: (0, 0, 0),
             4: (0, 0, 0),
             5: (111, 74, 0),
             6: (81, 0, 81),
             7: (128, 64, 128),
             8: (244, 35, 232),
             9: (250, 170, 160),
             10: (230, 150, 140),
             11: (70, 70, 70),
             12: (102, 102, 156),
             13: (190, 153, 153),
             14: (180, 165, 180),
             15: (150, 100, 100),
             16: (150, 120, 90),
             17: (153, 153, 153),
             18: (153, 153, 153),
             19: (250, 170, 30),
             20: (220, 220, 0),
             21: (107, 142, 35),
             22: (152, 251, 152),
             23: (70, 130, 180),
             24: (220, 20, 60),
             25: (255, 0, 0),
             26: (0, 0, 142),
             27: (0, 0, 70),
             28: (0, 60, 100),
             29: (0, 0, 90),
             30: (0, 0, 110),
```

Visualizing the results:



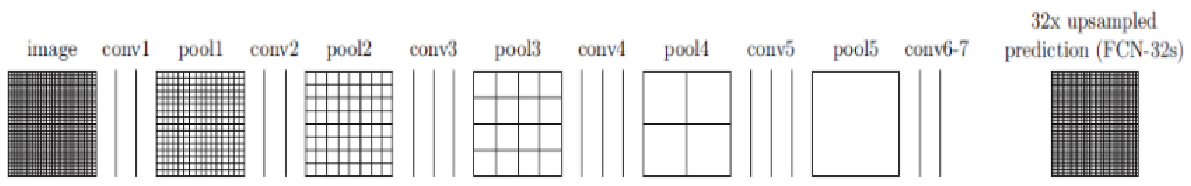
```
        [0.        , 0.        , 0.55686275]]])
```

```
[ ] import cv2
    cv2.imwrite('color_img.jpg',arr)
    c = cv2.imread('color_img.jpg', 1)
    c = cv2.cvtColor(c, cv2.COLOR_BGR2RGB)
    cv2.imwrite('Final_img.jpg', c)
```

```
    True
```

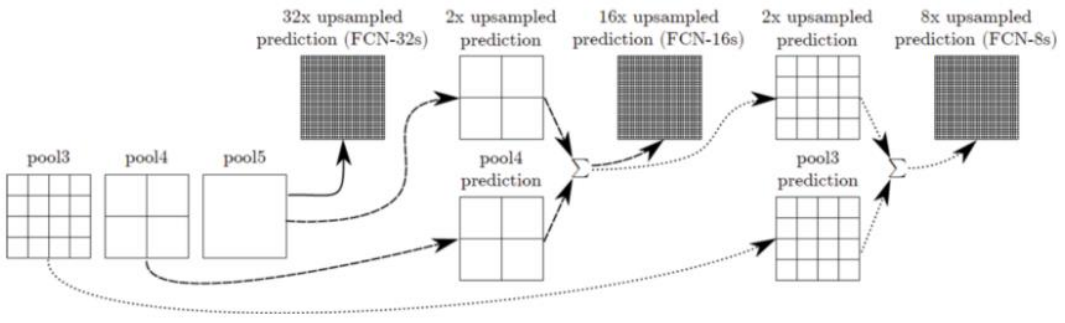
```
[ ] torch.save(model, "./drive/My Drive/hw5/entire_model_fcn32_final.pt")
```

Comparison of FCN 32 and FCN 16 (MOTIVATION):



FCN-32s

After convolution layer 7 the output size is small, to compensate this 32* up sampling is done to make the output size same as input size. But it makes output label map rough. This is because as we go deeper spatial location information is lost. That means shallow layers have more location information. Therefore, we go for FCN16 so that we can combine both to enhance the output. We do element wise addition to fuse the output. In this ouyput from pool 5 is 2* upsampled and is fused with pool 4 and do 16*upsampling.


32x upsampled prediction (FCN-32s)
2x upsampled prediction
16x upsampled prediction (FCN-16s)
2x upsampled prediction
8x upsampled prediction (FCN-8s)
pool3
pool4
pool5
pool4 prediction
pool3 prediction

Fusing for FCN-16s and FCN-8s

DISCUSSION OF RESULTS:

***Evaluation Metrics*:**

1. **Per Pixel Accuracy:**
   It outputs the class prediction accuracy per pixel.

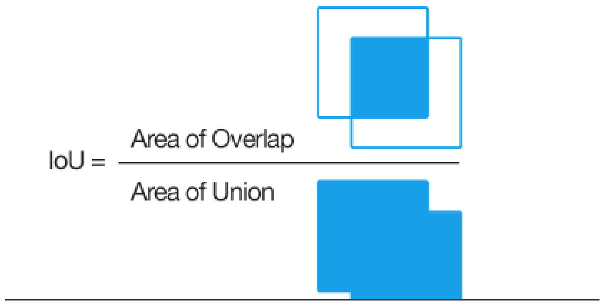
$$acc(P, GT) = \frac{|\text{pixels correctly predicted}|}{|\text{total nb of pixels}|}$$

2. **IOU (intersection over Union):**

It computes the number of pixels in intersection between the ground truth and predicted segmentation maps for a given class, divided by the number of pixels in union between two segmentation maps.

$$P(class), GT(class)) = \frac{|P(class) \cap GT(class)|}{|P(class) \cup GT(class)|}$$

Here P is the predicted segmentation map and G is ground truth. P(class) is the binary mask indication each pixel is predicted class or not.



Accuracy:

FCN32:

Mean IOU(Micro): `0.8326034546231245`

Mean IOU(Macro): `0.25`

FCN16:

Mean IOU (micro): `0.68`

Mean IOU (macro): `0.22`

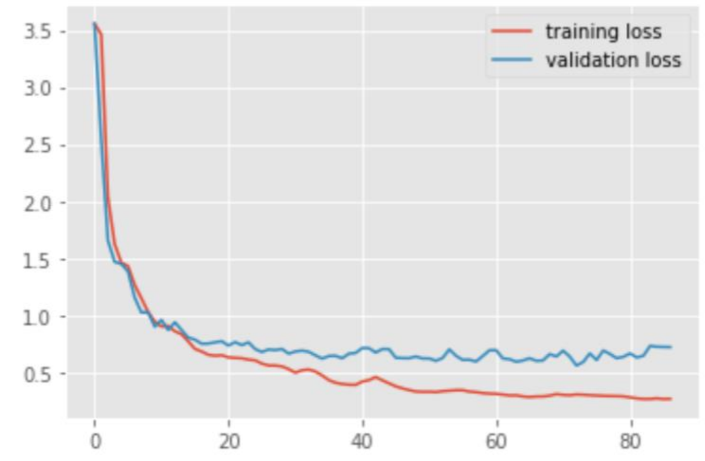
Effects of Parameters choices:

I tried with Adam and SGD optimizer. It seems that introducing weight decay in optimizer function is lowering the quality of the output**.** Learning Rate is 0.001 and momentum is 0.99.I ran it for 100 epochs. Batch size is 1.

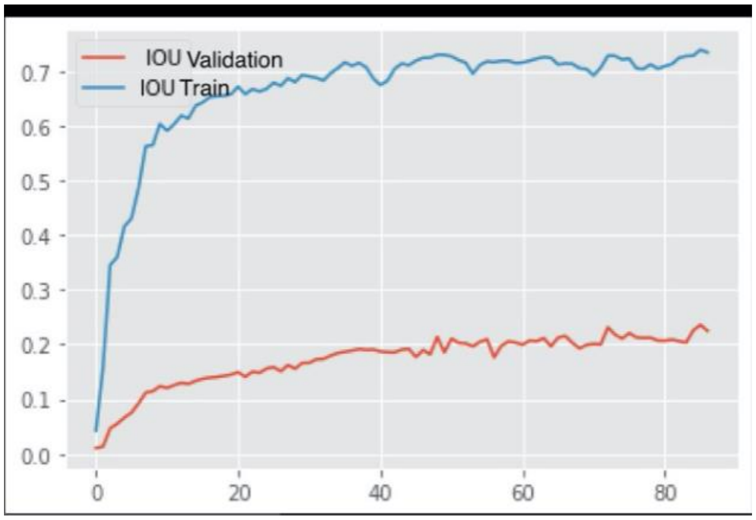
FCN 32:

Evaluation of loss function:

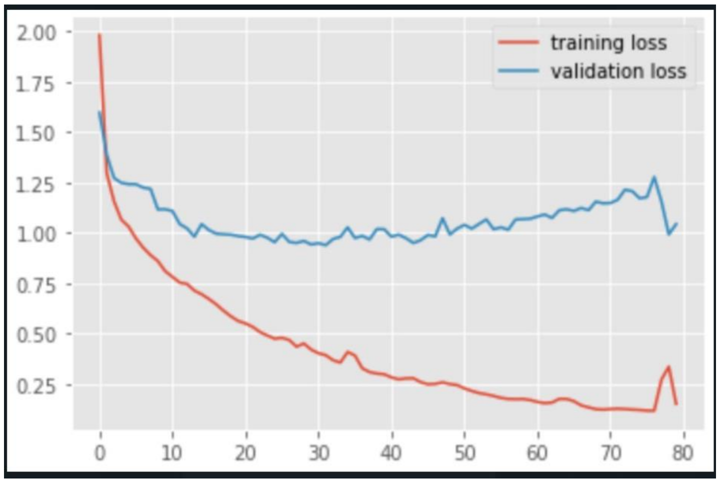
```
<matplotlib.legend.Legend at 0x7fcdae2e0668>
```

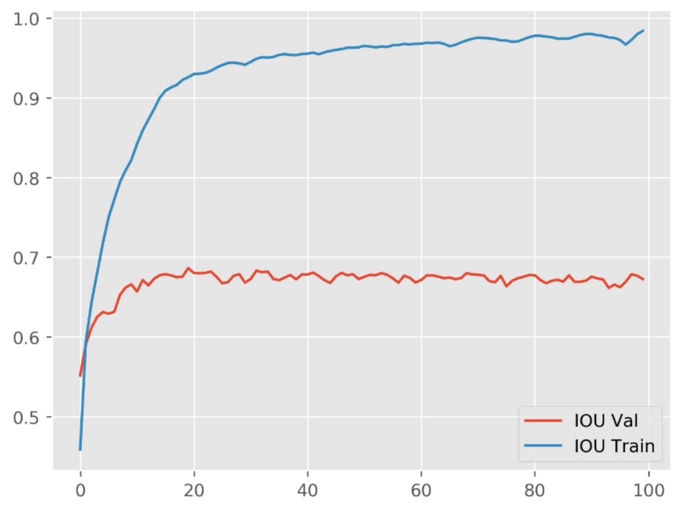


IOU at each iteration



FCN 16:





Comparison of FCN32 and FCN16**:**

FCN 16 perform better than FCN32 as FCN 16 incorporates spatial features as well from the shallow layers whereas in FCN32 spatial information is lost. FCN16 will look for fine grained structures cause of its larger encoder depth whereas FCN16 will result in more erroneously labeled boundary pixels due to its coarser segmentation.

VISUALIZING THE RESULTS:

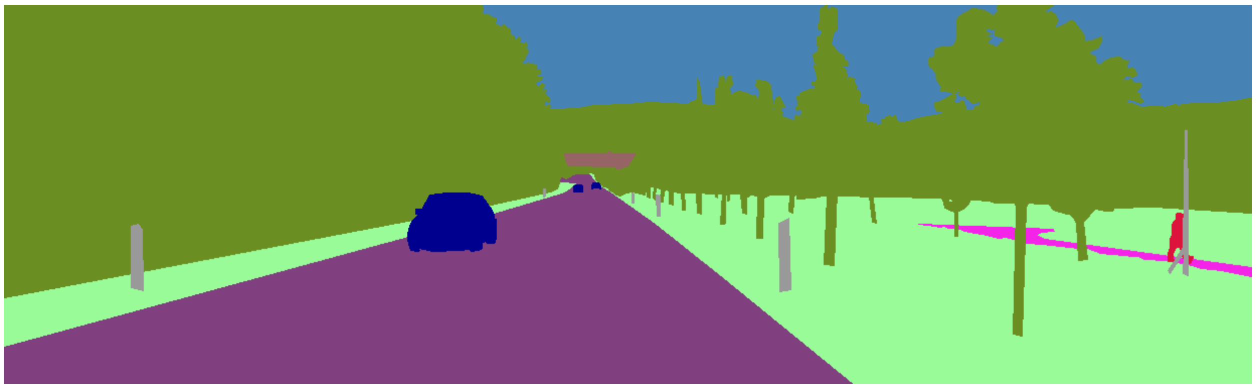
We have some failed results cases. Reason may be of the insufficient number of images in the dataset. One more reason could be this model is built on pretrained model VGG 16 and maybe features are different for this dataset.

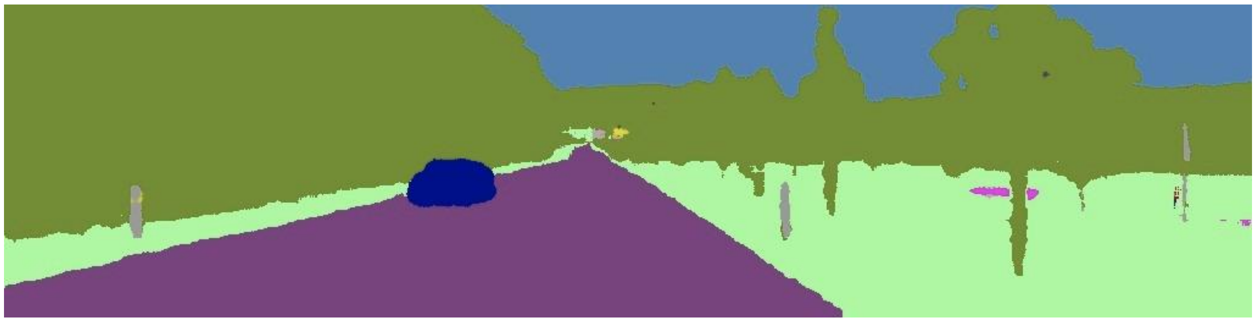
1. **SUCCESSFUL RESULTS:**

FCN 16 Good:

183

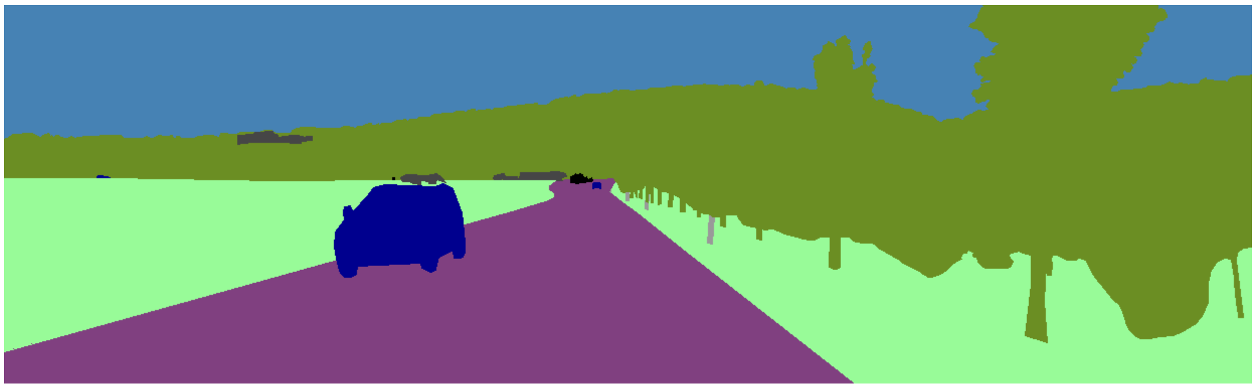
Ground Truth



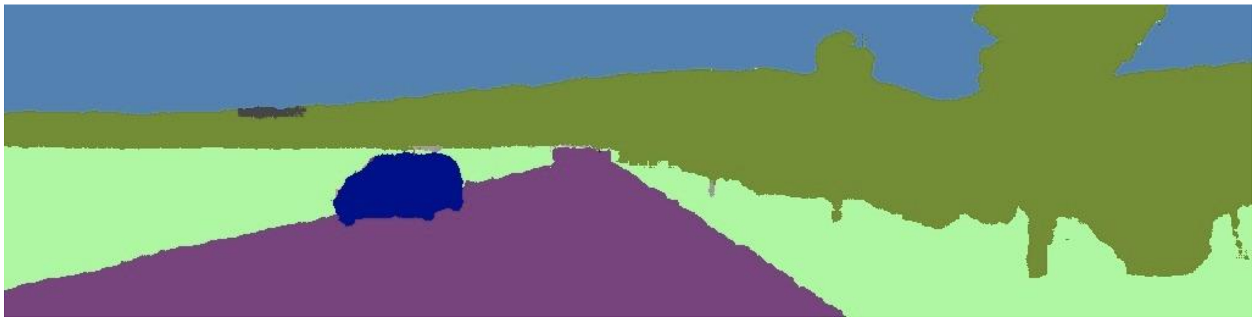


184

Ground Truth



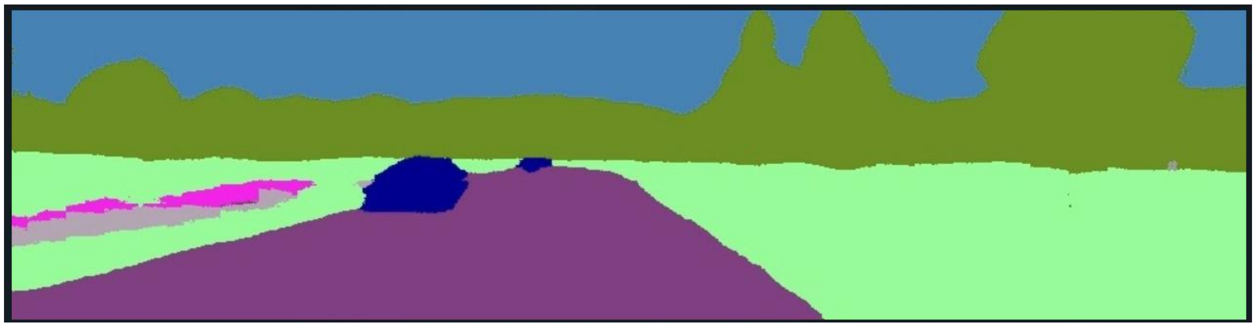
Predicted



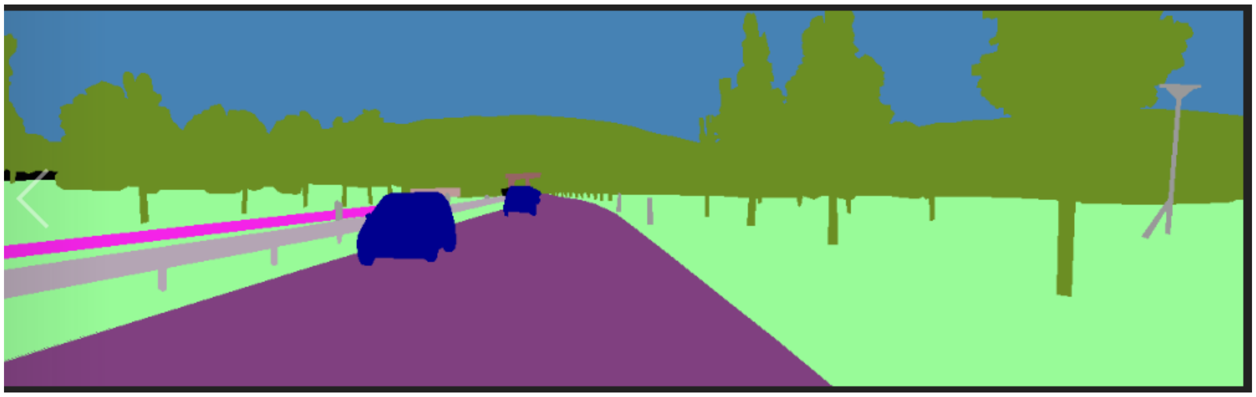
FCN 32:

180:

Ground truth:

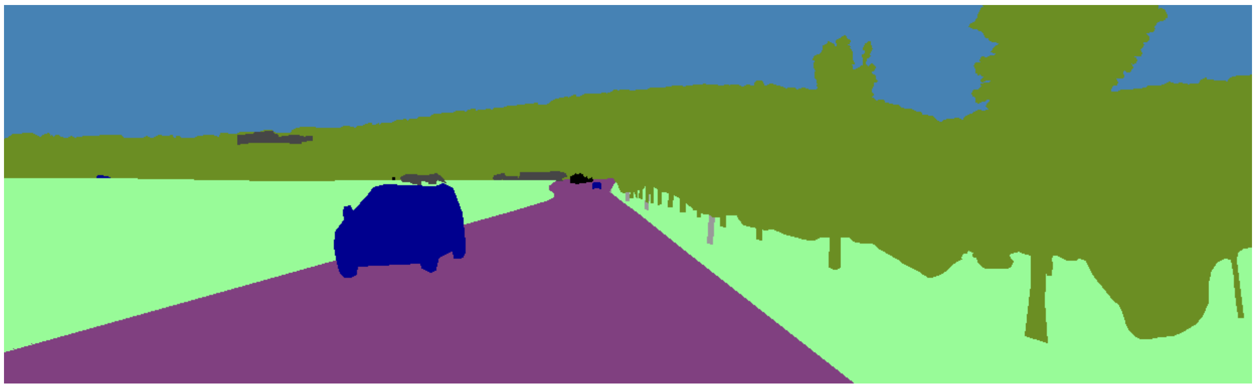


Predicted:

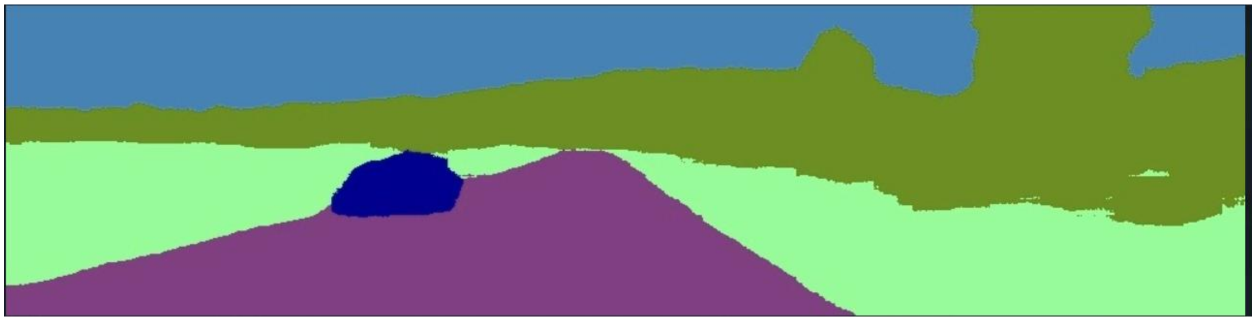


184

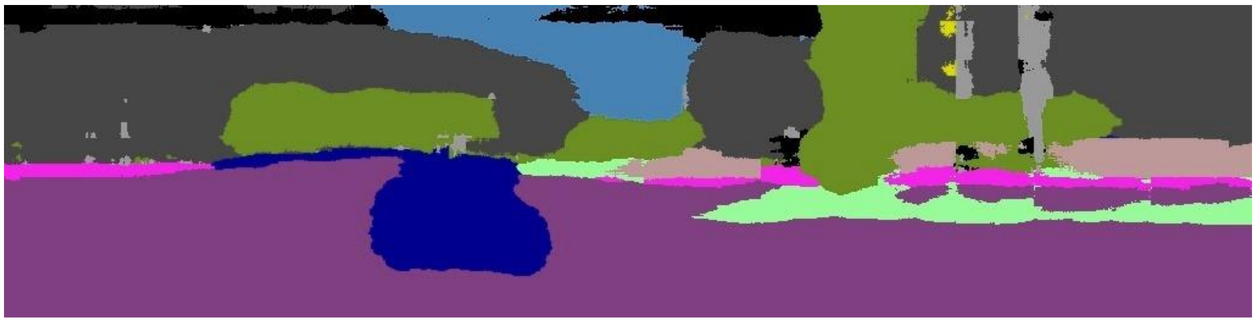
Ground truth



Result:



2. **FAILED RESULTS**: Model is not learning enough features. Introducing more convolutional layers with a greater number of filters will increase the accuracy.
   FCN32: Image 170
   Output Image



Ground truth Image

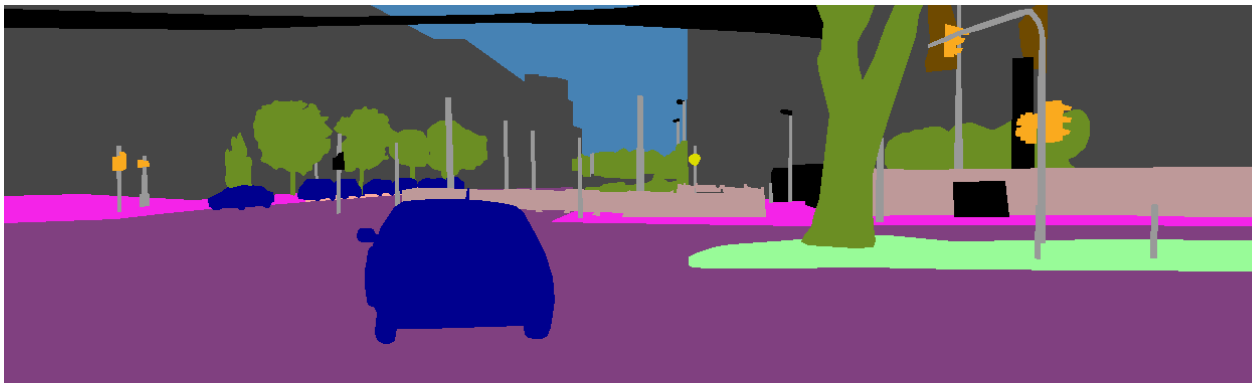
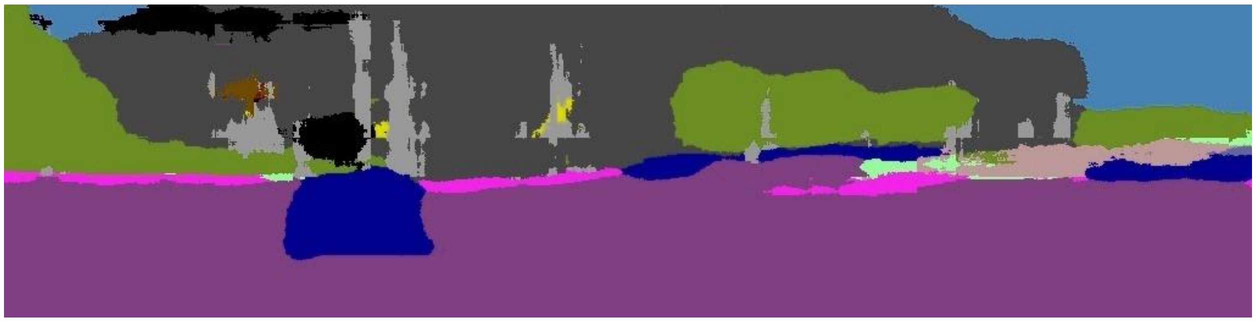
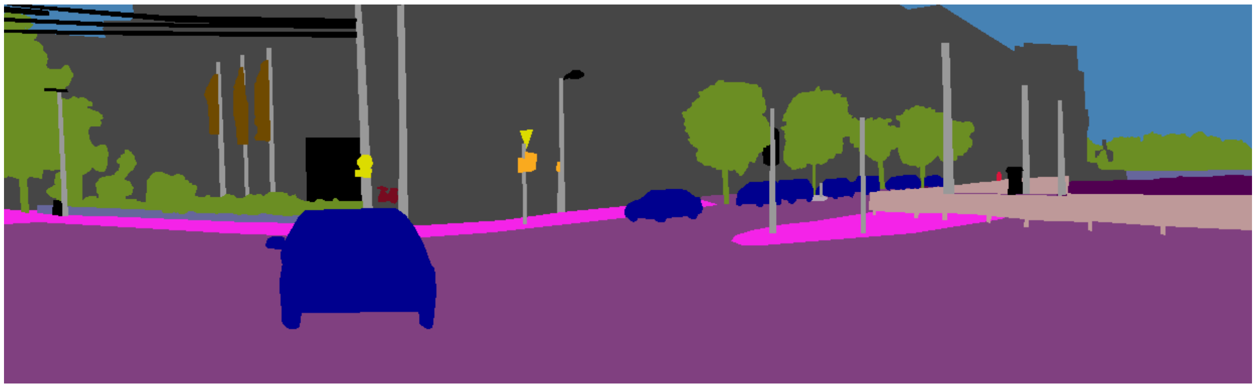


Image 171:

Output Image



Ground truth Image

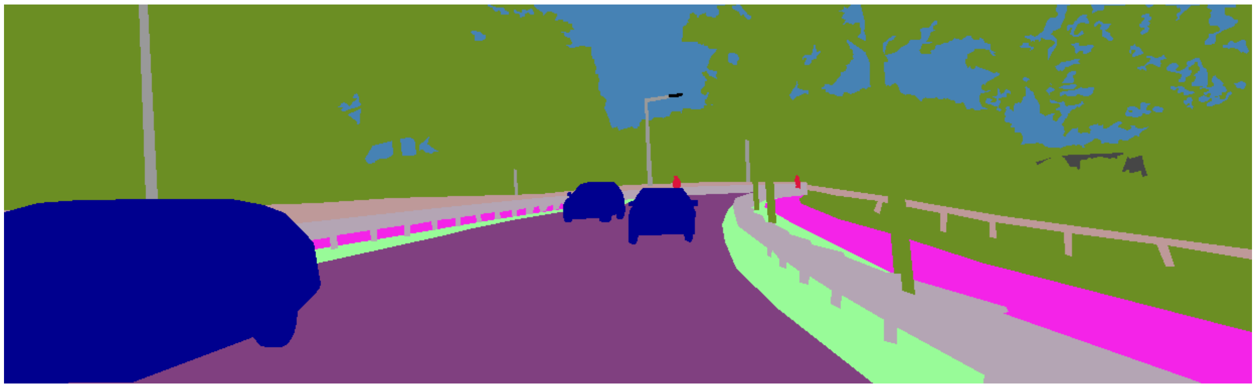


FCN 16:

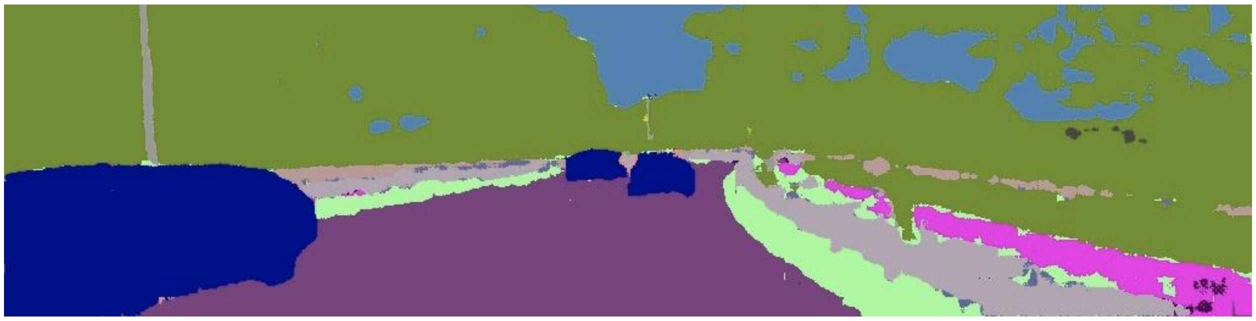
Failed Cases:

175

Ground truth

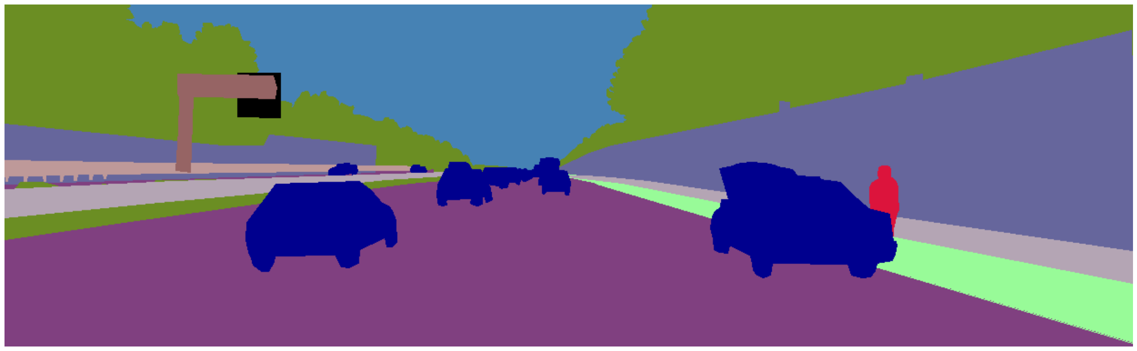


Result:

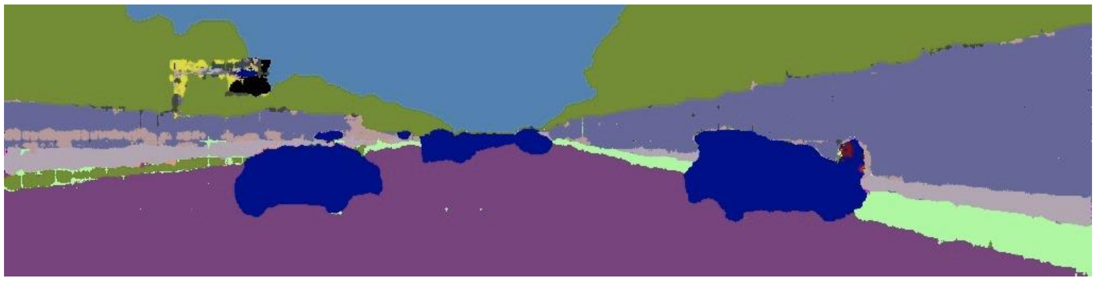


198:

Ground truth



Result



References:

1. https://d2l.ai/chapter_computer-vision/fcn.html
2. http://deeplearning.net/tutorial/fcn_2D_segm.html
3. https://github.com/Gurupradeep/FCN-for-Semantic-Segmentation
4. https://github.com/pochih/FCN-pytorch/blob/master/python/fcn.py
5. https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html