DLCV-HW1

- Student ID: B09611007

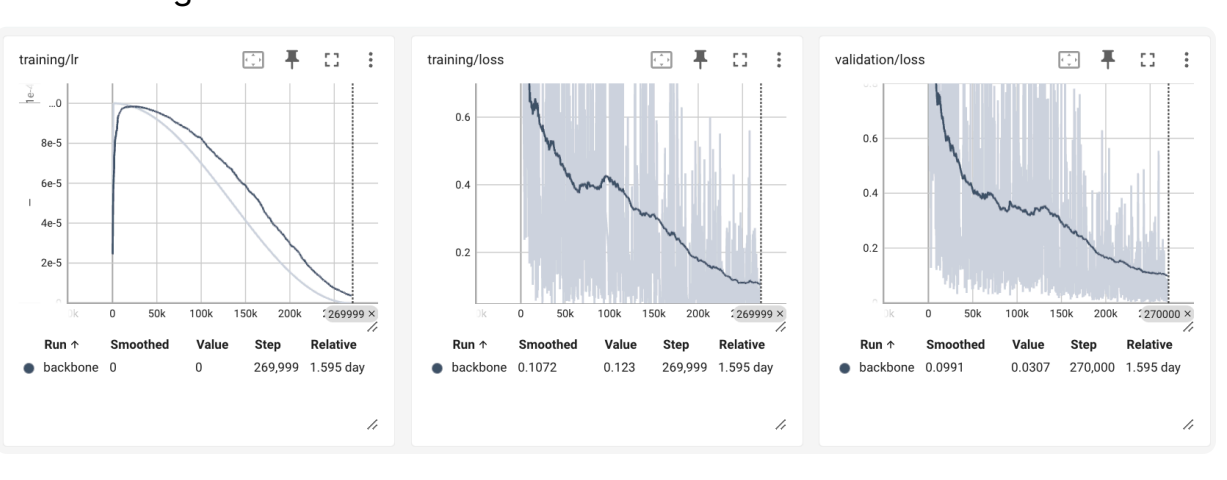
Problem 1: Self-Supervised Pre-training for Image Classification

1. (5%) Describe the implementation details of your SSL method for pre-training the ResNet50 backbone. (including but not limited to the name of the SSL method & data augmentation techniques you used, learning rate schedule, optimizer, and batch size setting for this pre-training phase)

- Self-Supervised Pre-training using Bootstrap Your Own Latent (BYOL) (https://github.com/lucidrains/byol-pytorch/tree/master)
- BYOL relies on two neural networks, referred to as **online networks** and **target networks**, that interact and learn from each other. From an augmented view of an image, the online network is trained to predict the target network representation of the same image under a different augmented view. At the same time, the target network is updated with a slow-moving average of the online network. (Reference (https://sh-tsang.medium.com/review-byol-bootstrap-your-own-latent-a-new-approach-to-self-supervised-learning-6f770a624441))
- Backbone: ResNet50
- Data Augmentation

```
RandomApply(
    T.ColorJitter(0.8, 0.8, 0.8, 0.2),
    p=0.3
),
T.RandomGrayscale(p=0.2),
T.RandomHorizontalFlip(),
RandomApply(
    T.GaussianBlur((3, 3), (1.0, 2.0)),
    p=0.2
),
T.RandomResizedCrop((128, 128)),
T.Normalize(
    mean=torch.tensor([0.485, 0.456, 0.406]),
    std=torch.tensor([0.229, 0.224, 0.225])),
```

- Pre-training Hyperparams
  - epoch: 1000
  - batch size: 256
  - initial learning rate: 0.00001
  - optimizer: Adam
  - scheduler: cosine annealing with warmup
- Pre-training Results



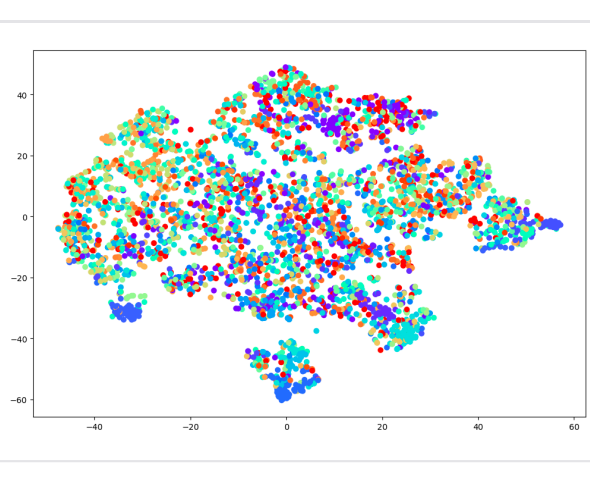
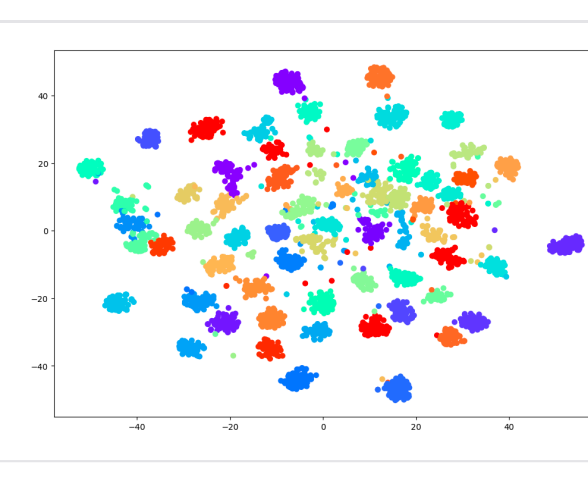
2. (20%) Please conduct the Image classification on Office-Home dataset as the downstream task. Also, please complete the following Table, which contains different image classification setting, and discuss/analyze the results.

- **Classifier:** Two FC layer (hidden_size=256) with dropout rate of 0.1

Setting	Pre-training (Mini-ImageNet)	Fine-tuning (Office-Home dataset)	Validation Acc. % (Office-Home dataset)
A	-	Train full model (backbone + classifier)	62.1
B	w/ label	Train full model (backbone + classifier)	61.3
C	w/o label	Train full model (backbone + classifier)	63.5
D	w/ label	Train classifier only	26.0
E	w/o label	Train classifier only	47.9

- **Pre-training with / without labels:** The results show that pretraining without labels (Self-Supervised Learning) outperforms pretraining with labels (Supervised Learning).
- In full model fine-tuning, the model pre-trained without labels achieved higher validation accuracy (C: 63.5% vs. B: 61.3%). Additionally, when only the classifier was fine-tuned, self-supervised pretraining also led to better performance (E: 47.9% vs. D: 26.0%). This suggests that self-supervised learning helps the model **capture more generalizable features** during pretraining, which may explain its superior performance even when only fine-tuning the classifier. Conversely, supervised learning may cause the model to **overfit to specific label information.**

3. (5%) Visualize the learned visual representation of setting C on the train set by implementing t-SNE (t-distributed Stochastic Neighbor Embedding) on the output of the second last layer. Depict your visualization from both the first and the last epochs. Briefly explain the results.

Epoch 1 (val_acc: 14.5%)	Epoch 200 (val_acc: 63.5%)
	

- **Epoch 1:** Data points are scattered without clear separation, showing the model hasn't learned meaningful features yet (reflected by low accuracy of 14.5%).
- **Epoch 200:** Data points form clear clusters, suggesting that the model has effectively learned to separate features (leading to higher accuracy of 63.5%).

Problem 2: Semantic Segmentation

1. (3%) Draw the network architecture of your VGG16-FCN32s model (model A).

```
VGG16_FCN32s(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(10
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ce
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ce
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, c
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, c
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, c
  )
  (fc6): Sequential(
    (0): Conv2d(512, 4096, kernel_size=(7, 7), stride=(1, 1))
    (1): ReLU()
    (2): Dropout2d(p=0.5, inplace=False)
  )
  (fc7): Sequential(
    (0): Conv2d(4096, 4096, kernel_size=(1, 1), stride=(1, 1))
    (1): ReLU()
    (2): Dropout2d(p=0.5, inplace=False)
  )
  (score_fr): Sequential(
    (0): Conv2d(4096, 7, kernel_size=(1, 1), stride=(1, 1))
    (1): ReLU()
  )
  (upscore): Sequential(
    (0): ConvTranspose2d(7, 7, kernel_size=(64, 64), stride=(32, 32))
  )
)
```

2. (3%) Draw the network architecture of the improved model (model B) and explain it differs from your VGG16-FCN32s model.

```
DeepLabV3_ResNet50(
  (model): DeepLabV3(
    (backbone): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), paddi
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, tr
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilati
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1),
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
          (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), b
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
          (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
          (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
        )
      )
      (layer2): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1)
          (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
            (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1)
          (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1)
          (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1)
          (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (relu): ReLU(inplace=True)
        )
      )
      (layer3): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1)
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1),
            (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
        (4): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
        (5): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
      )
      (layer4): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1)
            (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=Tr
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tru
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1
          (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=Tr
          (relu): ReLU(inplace=True)
        )
      )
    )
    (classifier): DeepLabHead(
      (0): ASPP(
        (convs): ModuleList(
          (0): Sequential(
            (0): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1),
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
            (2): ReLU()
          )
          (1): ASPPConv(
            (0): Conv2d(2048, 256, kernel_size=(3, 3), stride=(1, 1),
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
            (2): ReLU()
          )
          (2): ASPPConv(
            (0): Conv2d(2048, 256, kernel_size=(3, 3), stride=(1, 1),
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
            (2): ReLU()
          )
          (3): ASPPConv(
            (0): Conv2d(2048, 256, kernel_size=(3, 3), stride=(1, 1),
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
            (2): ReLU()
          )
          (4): ASPPPooling(
            (0): AdaptiveAvgPool2d(output_size=1)
            (1): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1),
            (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
            (3): ReLU()
          )
        )
        (project): Sequential(
          (0): Conv2d(1280, 256, kernel_size=(1, 1), stride=(1, 1), b
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
          (2): ReLU()
          (3): Dropout(p=0.5, inplace=False)
        )
      )
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
      (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
      (3): ReLU()
      (4): Conv2d(256, 7, kernel_size=(1, 1), stride=(1, 1))
    )
    (aux_classifier): FCNHead(
      (0): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), paddi
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
      (2): ReLU()
      (3): Dropout(p=0.1, inplace=False)
      (4): Conv2d(256, 7, kernel_size=(1, 1), stride=(1, 1))
    )
  )
)
```

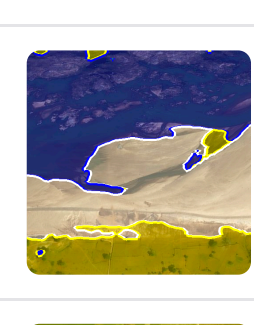
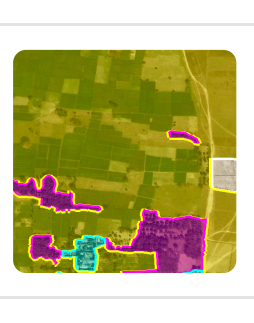
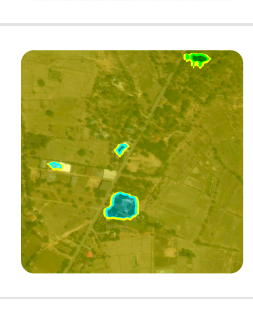
- **DeepLab V3**, introduced in the paper Rethinking Atrous Convolution for Semantic Image Segmentation (https://arxiv.org/abs/1706.05587), is selected as the model B. It utilizes **atrous (dilated) convolutions** to expand the receptive field without losing spatial resolution. The model also integrates **Atrous Spatial Pyramid Pooling (ASPP)** to capture multi-scale contextual information, making it highly effective in handling objects of varying sizes in complex scenes.
- To further adapt the model to the **imbalanced training dataset** *(0: 1109, 1: 1675, 2: 1000, 3: 416, 4: 697, 5: 801, 6: 72)*, **Multiclass Dice Loss** (https://arxiv.org/pdf/1707.03237v3) is employed (Reference Code (https://discuss.pytorch.org/t/implementation-of-dice-loss/53552/5)). Samples of class #6 (Unknown region) were excluded while calculating the loss.

$$DiceLoss(y, \overline{p}) = 1 - \frac{(2y\overline{p} + 1)}{(y + \overline{p} + 1)}$$

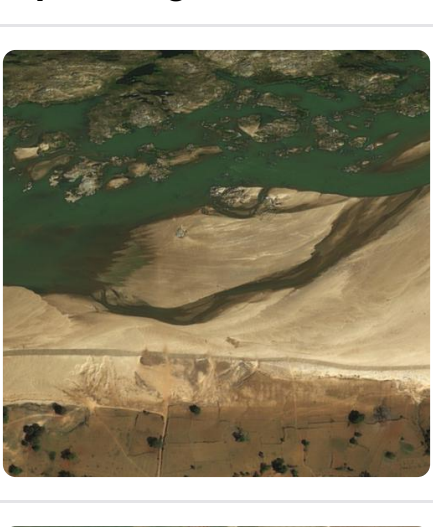
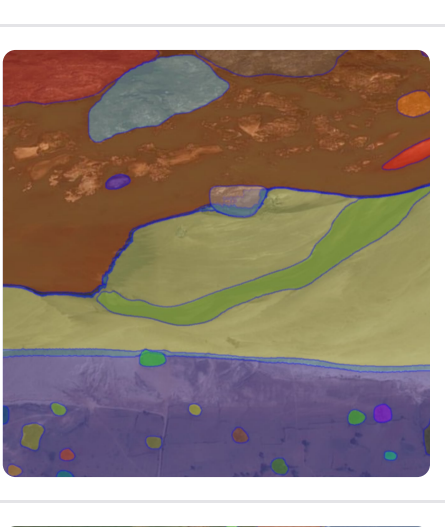
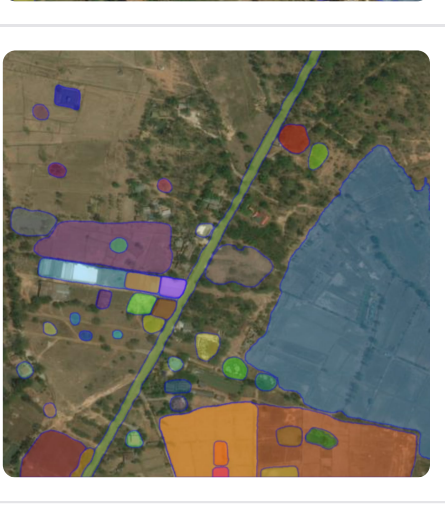
3. (1%) Report mIoUs of two models on the validation set.

Model ID	Architecture	Val. mIoU
A	VGG16-FCN32s	0.592
B	DeepLabV3	0.747

4. (3%) Show the predicted segmentation mask of "validation/0013_sat.jpg", "validation/0062_sat.jpg", "validation/0104_sat.jpg" during the early, middle, and the final stage during the training process of the improved model.

Image ID	Ground Truth	Early Stage (mIoU: 0.526)	Middle Stage (mIoU: 0.704)	Final Stage (mIoU: 0.747)
0013				
0062				
0104				

5. (10%) Use segment anything model (SAM) to segment three of the images in the validation dataset, report the result images and the method you use.

Image ID	Input Image	Result of SAM
0013		
0062		
0104		

- I used the model to find all the objects automatically on the official website of segment anything (https://segment-anything.com/demo)