1. Overview of your lab 3 (10%)

In this lab, we want to implement the **binary image segmentation** on Oxford-IIIT Pet Dataset using two Encoder-Decoder network architectures UNet and RestNet34-Unet. In the report, we will introduce the code detail on Section2, train/validate/test data preprocessing detail in Section 3, experiment results/analysis in Section4 and provide more discussion in Section 6.

2. Implementation Details (30%)

A. Details of your training, evaluating, inferencing code

Training code:

We implement the training process in train() function in **train.py**, in this function, we first use 'oxford_pet.load_dataset()' to construct the 'SimpleOxfordPetDataset' class and then use 'DataLoader()' to construct a dataloader. There are three variables:

- train_loader (for training data with transformations used during training)
- EvalTrain_loader (for training data without transformations used during evaluating)
- EvalValid_loader (for validation data without transformations used during evaluating)

```
#Load dataset and define dataloader
train_dataset = load_dataset(data_path=args.data_path, mode='train', transform=Compose([RandomHorizontalFlip(p=0.5),
                                                                                        RandomVerticalFlip(p=0.5),
                                                                                        RandomRotation90degreee()]))
EvalTrainDataSet = load_dataset(data_path=args.data_path, mode='train', transform=None)
EvalValidDataSet = load_dataset(data_path=args.data_path, mode='valid', transform=None)

train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=16)
EvalTrain_loader = DataLoader(EvalTrainDataSet, batch_size=args.batch_size, num_workers=16)
EvalValid_loader = DataLoader(EvalValidDataSet, batch_size=args.batch_size, num_workers=16)
```

And we define model(network architecture) and network optimizer.

```
#Define device, network model and optimizer
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = unet.UNet().to(device) if args.Network=='Unet' else resnet34_unet.ResNet34_UNet().to(device)
optimizer = torch.optim.RAdam(model.parameters(), lr=args.learning_rate)
```

For each epoch and each batch data from dataloader, we forward the image through the network and get the predict results. Then, we use BCE Loss to compute the loss and optimize the network parameter.

```
model.train()
image = data['image'].to(torch.float32).to(device)
mask = data['mask'].to(torch.float32).to(device)
pred = model(image)

loss = nn.BCELoss()(pred,mask)
optimizer.zero_grad()
loss.backward()
optimizer.step()
Total_Loss += loss.item()
```

evaluating code:

We implement the evaluation at the end of each training epoch using the evaluate

function in evaluate.py to calculate the average dice score for each data batch.

```
for data in EvalTrain_loader:
    DiceScore_train += evaluate(model, data, device)
for data in EvalValid_loader:
    DiceScore_valid += evaluate(model, data, device)

DiceScore_train /= EvalTrain_loader.__len__()
DiceScore_valid /= EvalValid_loader.__len__()
```

For the evaluate function in evaluate.py, we predict the result and use dice_score in utils.py to compute the dice score of data.

```
def evaluate(net, data, device):
    net.eval()
    image = data['image'].to(torch.float32).to(device)
    mask = data['mask'].to(torch.float32).to(device)
    with torch.no_grad():
        pred = net(image)
        pred = torch.where(pred > 0.5, 1, 0)
        Dice_score = dice_score(pred,mask)
        return Dice_score
```

inferencing code:

We implement model inference in inference.py. In this file, we test the model on the testing dataset and randomly select five images to display the segmentation areas predicted by the model alongside the original segmentation areas from the dataset.

```
#Construct the dataloader and load the trained model paprameter
dataset = load_dataset(data_path=args.data_path, mode='test', transform=None)
loader = DataLoader(dataset, batch_size=args.batch_size, num_workers=16)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = unet.UNet().to(device) if args.Network=='Unet' else resnet34_unet.ResNet34_UNet().to(device)
model.load_state_dict(torch.load(args.model))

# Test the performance on testing dataset
Total = 0
for idx, data in enumerate(loader):
    BatchDiceScore = evaluate(model,data,device)
    Total += BatchDiceScore
    print(f"Testing Batch {idx}/{loader.__len__()}, Batch Average Dice Score: {BatchDiceScore}")

print(f"Average Testing Dice Score : {Total/loader.__len__()}")
```

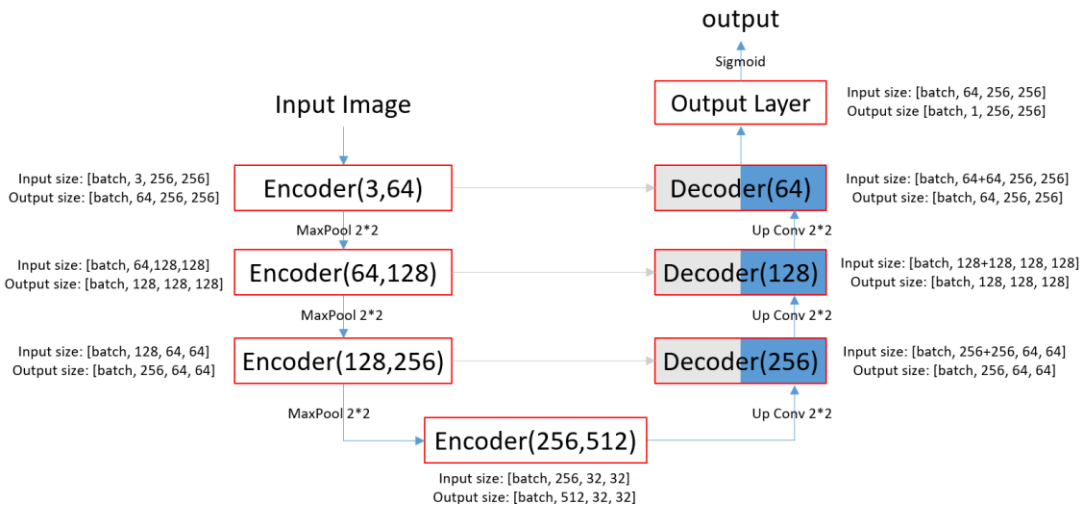
```
# Random inference 5 testing image and show the results
indices = random.sample(range(dataset.__len__()), 5)
fig, axs = plt.subplots(3, 5, figsize=(12, 6))
for i,idx in enumerate(indices):
    data = dataset.__getitem__(idx)
    image = torch.tensor(data['image']).to(torch.float32).to(device).unsqueeze(dim=0)
    with torch.no_grad():
        pred = model(image)

    axs[0,i].imshow(np.moveaxis(data['image'], 0, -1))
    axs[0,i].set_title(f'Figure {i+1}')
    axs[1,i].imshow(np.squeeze(data['mask'], axis=0))
    axs[1,i].set_title(f'Ground Truth {i+1}')
    axs[2,i].imshow(np.where(np.moveaxis(pred.cpu().squeeze(dim=0).numpy(), 0, -1) > 0.5, 1, 0))
    axs[2,i].set_title(f'Model Predict {i+1}')
plt.tight_layout()
plt.show()
```

B. Details of your model (UNet & ResNet34_UNet)

UNet:

The UNet is composed of 4 encoder blocks, 3 decoder blocks and 1 output layer. The overall flowchart is as following:



The architecture of Encoder(in_channels, out_channels):

- nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
- nn.ReLU(inplace=True),
- nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
- nn.ReLU(inplace=True)

The architecture of Decoder(channel):

- nn.Conv2d(channel*2, channel, kernel_size=3, padding=1)
- nn.ReLU(inplace=True),
- nn.Conv2d(channel, channel, kernel_size=3, padding=1)
- nn.ReLU(inplace=True)

The architecture of Output Layer:

- nn.Conv2d(64, 1, kernel_size=1)

The architecture of Up Conv 2*2:

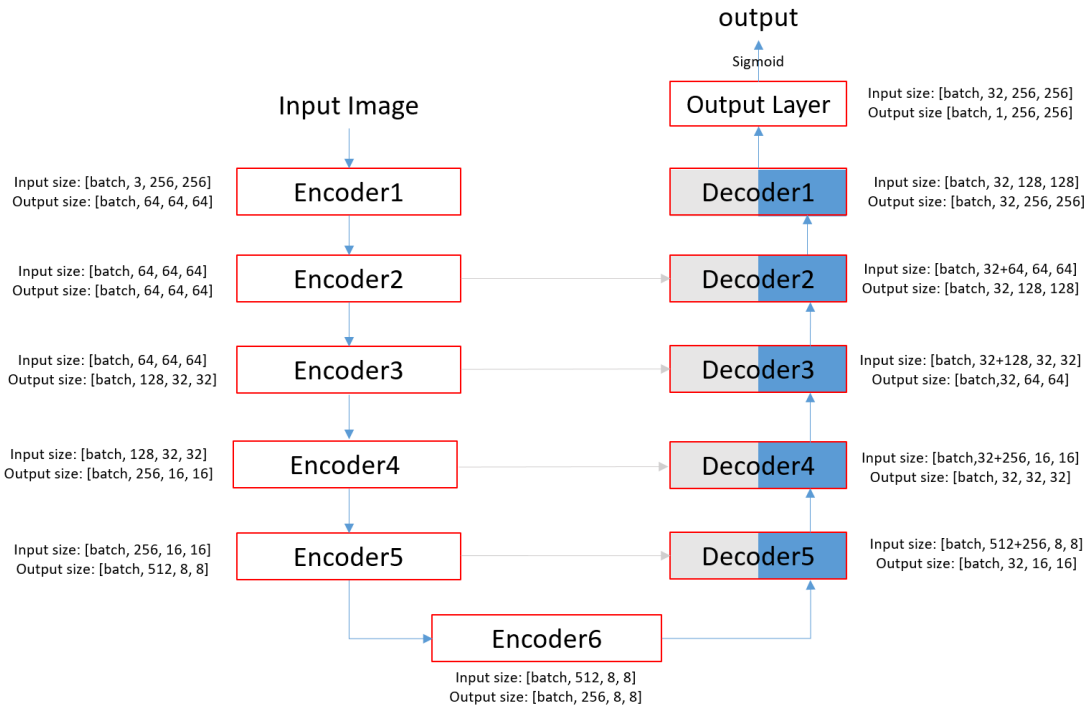
- nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)

The architecture of MaxPool 2*2:

- nn.MaxPool2d(kernel_size=2, stride=2)

ResNet34_UNet:

The ResNet34_UNet is composed by 6 encoders blocks, 5 decoder blocks and 1 output layer. The overall flowchart is as following:



In order to maintain a clean layout for the report, we will place the detailed structure of ResNet34_UNet in the 8.appendix at the end of the report.

C. Anything more you want to mention

In order to use the same training code and facilitate implementation, we made slight modifications to UNet architecture(We apply padding at each convolutional layer to preserve the same size and utilize a single-channel output instead of two channels at the output layer), but the overall structure remains largely the same.

3. Data Preprocessing (20%)

A. How you preprocessed your data?

For the training data used during training, we employ the following method for transformation:

- Random Horizontal Flip(p):

  For each data in the dataset, we randomly decide whether to apply horizontal flipping with a probability p (default = 0.5).

```python
class RandomHorizontalFlip(object):
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self,**sample):
        image, mask, trimap = sample['image'], sample['mask'], sample['trimap']
        if np.random.rand() < self.p:
            image = np.fliplr(image)
            mask = np.fliplr(mask)
            trimap = np.fliplr(trimap)
        return {'image': image, 'mask': mask, 'trimap':trimap}
```

- Random Vertical Flip(p):

  For each data in the dataset, we randomly decide whether to apply vertical flipping with a probability p (default = 0.5).

```python
class RandomVerticalFlip(object):
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self,**sample):
        image, mask, trimap = sample['image'], sample['mask'], sample['trimap']
        if np.random.rand() < self.p:
            image = np.flipud(image)
            mask = np.flipud(mask)
            trimap = np.flipud(trimap)
        return {'image': image, 'mask': mask, 'trimap':trimap}
```

- RandomRotation90degree:

  For each data in the dataset, we randomly select a degree of rotation [0, 90, 180, 270] for the image.

```python
class RandomRotation90degree(object):
    def __call__(self,**sample):
        image, mask, trimap = sample['image'], sample['mask'], sample['trimap']
        angle = random.randint(0, 3)
        image = np.rot90(image, -angle)
        mask = np.rot90(mask, -angle)
        trimap = np.rot90(trimap, -angle)
        return {'image': image, 'mask': mask, 'trimap':trimap}
```

For the training/validation/testing data used during evaluation, we keep them in their original form.

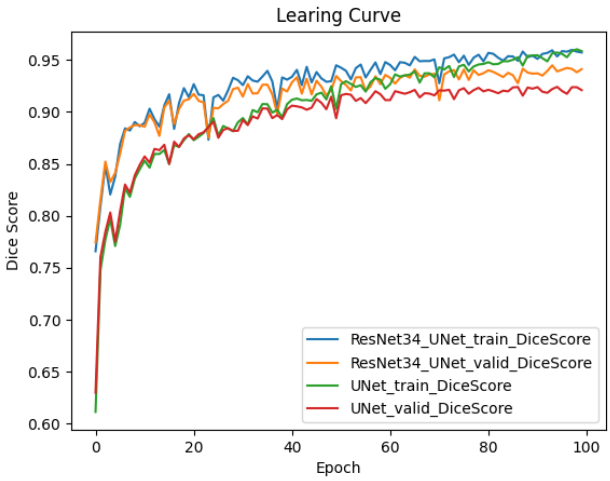
B. What makes your method unique?

In addition to horizontal and vertical flipping, we have also introduced rotation. Rotation can be performed at angles of 90/180/270 degrees.

4. Analyze on the experiment results (20%)

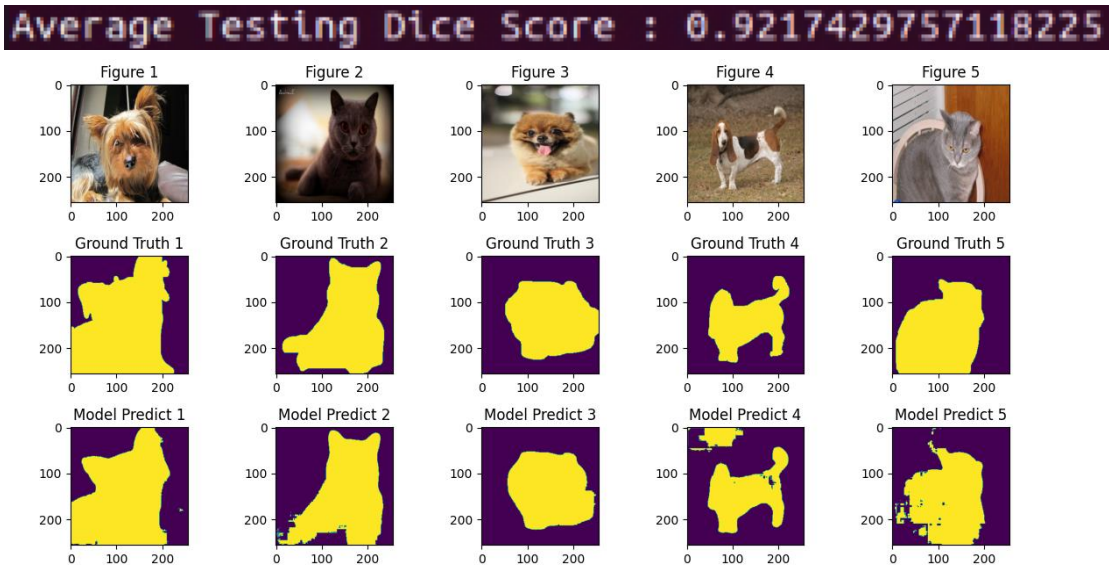
A. What did you explore during the training process?

<u>Learining Cure:</u>

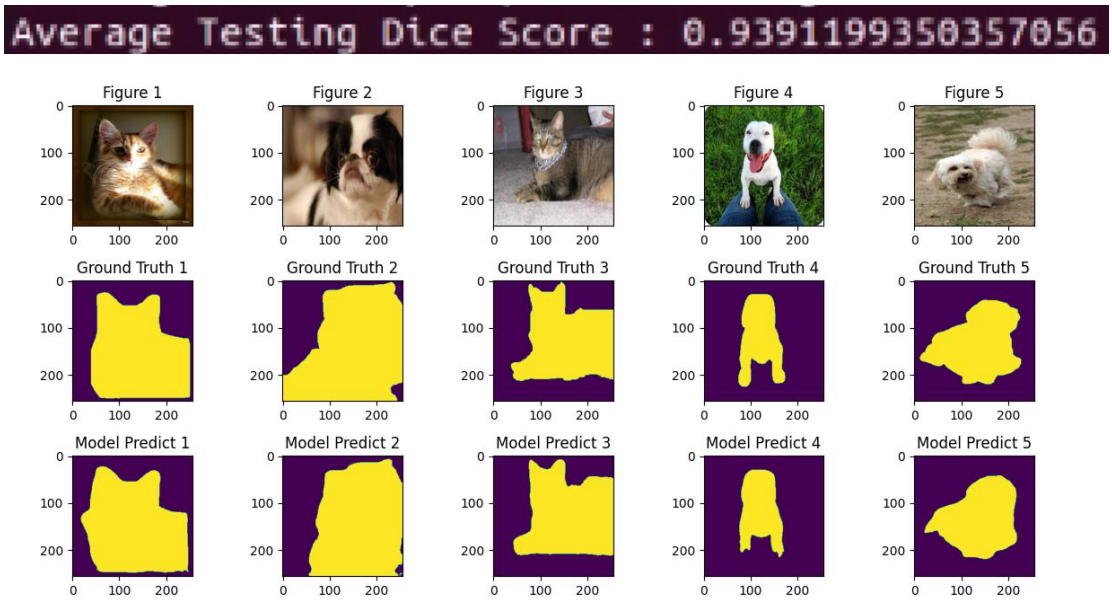


Testing Result(Dice Score and the segmentation results on random 5 image):

Unet:



ResNet34_UNet:

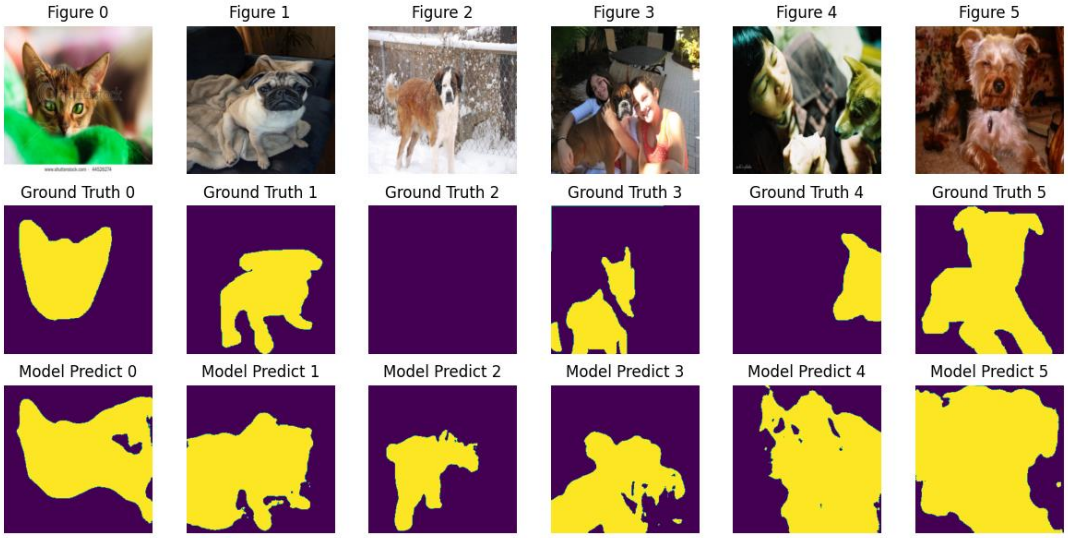


Discussion:

1. Although we only trained on the training set, we can observe from the learning curve that the Dice scores of both the training and validation sets are very close, indicating that the model has good generalization ability.
2. In the experiment, we observe that the optimizers 'RAdam' and 'SGD' perform better in handling image-related tasks (such as image classification or image segmentation). On the contrary, 'Adam' or 'RMSprop' exhibit poorer optimization performance.
3. ResNet34_UNet has more parameters compared to UNet, yet it requires shorter training time for the same max_epoch.

B. Found any characteristics of the data?

In this section, we select some representative training data that have low Dice scores to showcase the results and analyze the reasons.



We conclude that the model performs poorly on the following specific types of data:

- When the ground truth labels are not accurate enough (e.g., Figure 2).
- When the background has the similar color as the pet (e.g., Figure 1, 5).
- When there are people interfering near the pet (e.g., Figure 3, 4).
- When the background is blurred (e.g., Figure 0).

5. Execution command (0%)

A. The command and parameters for the training process

Training hyperparameter:

- epochs: 100
- batch_size: 32
- lr: 2e-4
- optimizer: RAdam
- Loss function: BCELoss

<u>Usage:</u>

if you want to train “UNet”:

- python train.py --data_path=path_to_the_oxford-iiit-pet --epochs=100 --batch_size=32 --learning_rate=2e-4 --Network=Unet

if you want to train “ResNet34_UNet”:

- python train.py --data_path=path_to_the_oxford-iiit-pet --epochs=100 --batch_size=32 --learning_rate=2e-4 --Network=ResNet34_UNet

B. The command and parameters for the inference process

Hyperparameter:

- Batch_size=32

Usage:

If you want to perform inference with UNet:

- python inference.py --model=path_to_the_DL_Lab3 _UNet_312554002_陳明宏.pth --data_path=path_to_the_oxford-iiit-pet --batch_size=32 --Network=Unet

If you want to perform inference with ResNet34_UNet:

- python inference.py --model=path_to_the_DL_Lab3_ResNet34_UNet_312554002_陳明宏.pth --data_path= path_to_the_oxford-iiit-pet --batch_size=32 --Network=ResNet34_UNet

6. Discussion (20%)

A. What architecture may bring better results?

From the experimental results shown in Section 4, we observe that ResNet34_UNet achieves better results in terms of training speed and generalization on the validation/testing dataset. This is because ResNet34_UNet has more parameters(extend the model flexibility) than UNet and utilizes a bottleneck structure(easy to back propagate the gradient to shallow layers.). Although ResNet34_UNet yields better results, we can also see that UNet achieves very good performance!

B. What are the potential research topics in this task?

- Semantic Segmentation:
  In this topic, we will classify each pixel in the image, which involves boundary segmentation and label classification.
- Instance Segmentation:
  This topic is based on semantic segmentation, we will segment objects of the same class into different independent boundaries.
- Panoramic Segmentation:
  This topic is based on instance segmentation. Instead of solely focusing on the objects of interest, we also consider the background.
- Video Segmentation:
  In this topic, we will consider video segmentation. This can be seen as segmenting multiple images and stacking them together. However, the challenge lies in ensuring that the segmented images maintain smooth transitions between frames, similar to the original video (i.e., smoothness of connections between images in each frame).
- Volumetric image segmentation.

In this topic, we consider volumetric image segmentation. Unlike image segmentation, volumetric segmentation takes into account an additional dimension. The challenge of this topic lies not only in the increased data size but also in how to ensure smoothness in the output segmentation.

7. Reference:

- https://blog.csdn.net/github_36923418/article/details/83273107
- https://medium.com/ching-i/%E5%BD%B1%E5%83%8F%E5%88%86%E5%89%B2-image-segmentation-%E8%AA%9E%E7%BE%A9%E5%88%86%E5%89%B2-semantic-segmentation-1-53a1dde9ed92

8. Appendix:

The detail structure of ResNet34_Unet:

Enocder1: ResNet34[0]

```
(conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
```

Enocder2: ResNet34[1]

```
(layer1): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

Enocder3: ResNet34[2]

```
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

Enocder4: ResNet34[3]

```
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (4): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (5): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

Enocder5: ResNet34[4]

```
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

Encoder 6:

Sequential(

- BasicBlock(

  (conv1): Conv2d(512, 256, kernel_size=(1,1), stride=(1,1), padding=(0,0))

  (bn1): BatchNorm2d(256)

  (relu): ReLU(inplace=True)

  (conv2): Conv2d(256, 256, kernel_size=(1,1), stride=(1, 1), padding=(0,0))

  (bn2): BatchNorm2d(256)

  (downsample): Sequential(

    (0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1,1))

    (1): BatchNorm2d(256)

)

)
- BasicBlock(

  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

  (bn1): BatchNorm2d(256)

  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

  (bn2): BatchNorm2d(256)

)
- BasicBlock(

  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(bn1): BatchNorm2d(256)

(relu): ReLU(inplace=True)

(conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(bn2): BatchNorm2d(256)

)

)

Decoder 5:

```
(block7): Decoder(
  (up_conv_relu): Sequential(
    (0): ConvTranspose2d(768, 32, kernel_size=(2, 2), stride=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
  )
)
```

Decoder 4:

```
(block8): Decoder(
  (up_conv_relu): Sequential(
    (0): ConvTranspose2d(288, 32, kernel_size=(2, 2), stride=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
  )
)
```

Decoder 3:

```
(block9): Decoder(
  (up_conv_relu): Sequential(
    (0): ConvTranspose2d(160, 32, kernel_size=(2, 2), stride=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
  )
)
```

Decoder 2:

```
(block10): Decoder(
  (up_conv_relu): Sequential(
    (0): ConvTranspose2d(96, 32, kernel_size=(2, 2), stride=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
  )
)
```

Decoder 1:

```
(block11): Decoder(
  (up_conv_relu): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(2, 2), stride=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
  )
)
```

Output Layer:

- Conv2d(32, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))