# DLP
# Lab2: Binary Semantic Segmentation
# Report

March 24, 2025

# Contents


1 Implementation Details 3
- 1.1 Training . . . 3
  - 1.1.1 Initialize . . . 3
  - 1.1.2 Train . . . 4
  - 1.1.3 Evaluate . . . 4
- 1.2 Evaluation . . . 5
- 1.3 Inference . . . 5
- 1.4 Model Architecture . . . 6
  - 1.4.1 UNet . . . 6
  - 1.4.2 ResNet34Unet . . . 9
- 1.5 Loss Function . . . 12

2 Data Preprocessing 13
- 2.1 Preprocessing . . . 13
- 2.2 What makes the preprocessing method unique . . . 14

3 Analyze the experiment results 14
- 3.1 Training Curves . . . 14
- 3.2 Performance Metrics . . . 14
- 3.3 Visualization of Segmentation Results . . . 15
- 3.4 Discussion . . . 16

4 Execution steps 16
- 4.1 How to run the code . . . 16

5 Discussion 17
- 5.1 Alternative architectures . . . 17
- 5.2 Potential Research Topics . . . 18

6 References 18

# 1 Implementation Details

## 1.1 Training

The training procedure is just a ordinary training procedure in PyTorch. For the implementation of the training process, I follow the steps below:

1. Load the model and dataset
2. Define the optimizer, learning rate scheduler, and loss function
3. Train the model
4. Evaluate the model after each epoch

For the detailed implementation of each step:

### 1.1.1 Initialize

Load the model and train, valid dataset, and define the optimizer, learning rate scheduler, and loss function.

```
# Load the model
if args.model == "unet":
    model = UNet()
elif args.model == "resnet34_unet":
    model = ResNet34Unet()
else:
    raise ValueError(f"Model {args.model} not found")

model.to(device)

# Load the dataset
train_dataset = load_dataset(args.data_path, "train")
valid_dataset = load_dataset(args.data_path, "valid")

train_loader = DataLoader(
    train_dataset, batch_size=args.batch_size, shuffle=True,
    num_workers=32
)
valid_loader = DataLoader(
    valid_dataset, batch_size=args.batch_size, shuffle=False,
    num_workers=32
)

optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=args.epochs)
loss_fn = torch.nn.BCELoss()
```

Listing 1: train.py: Initialize

### 1.1.2 Train

Train the model for a specified number of epochs, and use Binary Cross Entropy loss function and Dice loss function as the loss function.

```
for epoch in range(args.epochs):
    model.train()
    train_dice_score, train_bce_loss, train_dice_loss = 0, 0, 0

    for batch in tqdm(train_loader, desc=f"Epoch {epoch +
    1}/{args.epochs}"):
        images = batch["image"].to(device).float()
        masks = batch["mask"].to(device)

        optimizer.zero_grad()
        pred_masks = model(images)

        d_score = dice_score(pred_masks, masks)
        b_loss = loss_fn(pred_masks, masks)
        d_loss = dice_loss(pred_masks, masks)

        loss = b_loss + d_loss
        loss.backward()
        optimizer.step()

        train_dice_score += d_score.item()
        train_bce_loss += b_loss.item()
        train_dice_loss += d_loss.item()
```

Listing 2: train.py: Train

### 1.1.3 Evaluate

Evaluate on validation set and save the best model (and log the results on wandb)

```
eval_dice_score, eval_bce_loss, eval_dice_loss = evaluate(
    model, valid_loader, device
)
if eval_dice_score > best_eval_dice_score:
    best_eval_dice_score = eval_dice_score
    torch.save(
        model.state_dict(),
        f"saved_models/{args.model}_{epoch}_{eval_dice_score:.4f}.pth",
    )

if args.wandb:
    wandb.log(
        {
            "train/dice_score": train_dice_score / len(train_loader),
            "train/bce_loss": train_bce_loss / len(train_loader),
            "train/dice_loss": train_dice_loss / len(train_loader),
            "valid/dice_score": eval_dice_score,
            "valid/bce_loss": eval_bce_loss,
            "valid/dice_loss": eval_dice_loss,
        }
```

```
    )
```

Listing 3: train.py: Evaluate

### 1.2 Evaluation

For evaluation, I use the same procedure as the training process, the only difference is the dataset and the model is not updated.

```python
def evaluate(net, data, device):
    # implement the evaluation function here
    net.eval()
    dice_scores = []
    bce_losses = []
    dice_losses = []
    bce_loss = torch.nn.BCELoss()
    with torch.no_grad():
        for batch in data:
            images = batch["image"].to(device).float()
            masks = batch["mask"].to(device)
            pred_masks = net(images)
            dice_scores.append(dice_score(pred_masks, masks).item())
            bce_losses.append(bce_loss(pred_masks, masks).item())
            dice_losses.append(dice_loss(pred_masks, masks).item())
    return np.mean(dice_scores), np.mean(bce_losses), np.mean(dice_losses)
```

Listing 4: evaluate.py: Evaluate

### 1.3 Inference

For inference, I use the same procedure as the evaluation process, and this time, I need to load the state dict of the model and calculate the accuracy on the test set.

```python
def inference(model, device):
    if "resnet34" in args.model:
        model = ResNet34Unet(in_channels=3)
    else:
        model = UNet(in_channels=3)

    model.load_state_dict(torch.load(args.model))
    model.to(device)
    model.eval()

    test_dataset = load_dataset(args.data_path, "test")
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size,
    shuffle=False)

    dice_score, _, _ = evaluate(model, test_loader, device=device)
    print(f"Dice score: {dice_score:.4f}")
```

Listing 5: inference.py: Inference

## 1.4 Model Architecture

### 1.4.1 UNet

For the UNet model, I use almost the same architecture as the one in the paper. The only difference if that in the original paper, the DownConv layers do not use padding, so the output size is smaller than the input size divided by 2. However, in this implementation, I use padding, so the output size is the same as the input size divided by 2, since by this way, it is easier to implement the upsampling layer, and also easier to integrate with the ResNet34.

The architecture of the UNet is shown below (the size of the input is 256x256, the size and the dimension of each layer output is annotated in the diagram):

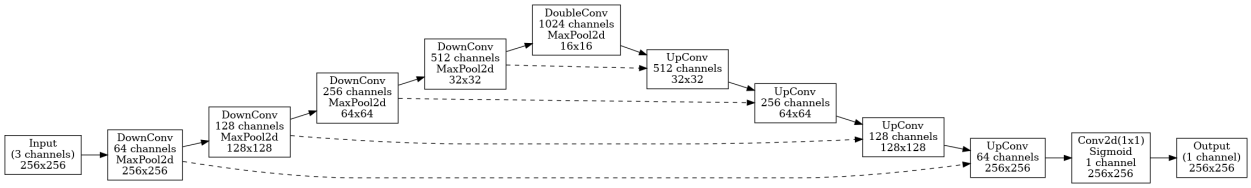


Figure 1: UNet Architecture

For the implementation of the UNet, I refer to the code from a public repository milesial/Pytorch-UNet.

To simply describe the architecture, it can be divided into 3 parts:

1. DoubleConv layers: The double convolutional layers, which is a series of convolutional layers with batch normalization and ReLU activation functions.
2. DownConv layers: The downsampling path of the UNet, which is a DoubleConv layer with max-pooling layers.
3. UpConv layers: The upsampling path of the UNet, which is a DoubleConv layer with bilinear upsampling layers.
4. Middle layer: The middle layer of the UNet, which is a DoubleConv layer.

DoubleConv The DoubleConv layer is a series of convolutional layers with batch normalization and ReLU activation functions. There was not BatchNorm in the original paper, but I add it for better performance.

```
class DoubleConv(nn.Module):
def __init__(self, in_channels, out_channels):
    super(DoubleConv, self).__init__()

    layers = [
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,
        bias=False),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1,
        bias=False),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
```

```
    ]

    self.nn = nn.Sequential(*layers)

def forward(self, x):
    return self.nn(x)
```

Listing 6: models/unet.py: DoubleConv

DownConv The DownConv layer is a DoubleConv layer with max-pooling layers. It is used to reduce the spatial size of the feature map and expand the channel dimension.

```
class DownConv(nn.Module):
def __init__(self, in_channels, out_channels):
    super(DownConv, self).__init__()

    layers = [nn.MaxPool2d(kernel_size=2), DoubleConv(in_channels,
    out_channels)]

    self.nn = nn.Sequential(*layers)

def forward(self, x):
    return self.nn(x)
```

Listing 7: models/unet.py: DownConv

UpConv The UpConv layer is a DoubleConv layer with bilinear upsampling layers. It is used to increase the spatial size of the feature map and reduce the channel dimension. The input from previous layer (the mid / UpConv layer) is concatenated with the output from the DownConv layer at the same spatial size after upsampling, and then the DoubleConv layer is applied.

```
class UpConv(nn.Module):
def __init__(self, in_channels, out_channels):
    super(UpConv, self).__init__()

    self.up = nn.ConvTranspose2d(in_channels, out_channels,
    kernel_size=2, stride=2)
    self.conv = DoubleConv(in_channels, out_channels)

def forward(self, x1, x2):
    x = self.up(x1)
    x = torch.cat([x, x2], dim=1)
    return self.conv(x)
```

Listing 8: models/unet.py: UpConv

Middle The middle layer is a DoubleConv layer, which is used to connect the upsampling path and the downsampling path.

```
self.mid = DownConv(down_channels[-1], up_channels[0])
```

Listing 9: models/unet.py: UNet

UNet The UNet model is a combination of the DoubleConv, DownConv, UpConv, and Middle layers. The output activation function is sigmoid, so we can get the binary segmentation mask and use Binary Cross Entropy loss function. I think the most confusing part is the skip connection, which is used to connect the output from the DownConv layer and the input from the UpConv layer at the same spatial size. Make sure which one should be popped from the stack is important.

```python
    class UNet(nn.Module):
    def __init__(
        self,
        in_channels: int = 3,
        down_channels: list[int] = [64, 128, 256, 512],
        up_channels: list[int] = [1024, 512, 256, 128, 64],
        out_channels: int = 1,
    ):
        super(UNet, self).__init__()

        self.in_conv = DoubleConv(in_channels, down_channels[0])

        self.down = nn.ModuleList()
        self.up = nn.ModuleList()

        for i in range(len(down_channels) - 1):
            self.down.append(DownConv(down_channels[i], down_channels[i +
            1]))

        self.mid = DownConv(down_channels[-1], up_channels[0])

        for i in range(len(up_channels) - 1):
            self.up.append(UpConv(up_channels[i], up_channels[i + 1]))

        self.out = nn.Sequential(
            nn.Conv2d(up_channels[-1], out_channels, kernel_size=1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.in_conv(x)

        x_rec = [x]
        for down in self.down:
            x_rec.append(down(x))
            x = x_rec[-1]

        x = self.mid(x)

        for up in self.up:
            x = up(x, x_rec.pop())
        return self.out(x)
```

Listing 10: models/unet.py: UNet

### 1.4.2 ResNet34Unet

For the ResNet34Unet model, I use the ResNet34 as the encoder and the UNet as the decoder. The ResNet34 model architecture is mostly same as the original paper and also the image in Lab2 Spec.

The architecture of the ResNet34Unet is shown below (the size of the input is 256x256, the size and the dimension of each layer output is annotated in the diagram):

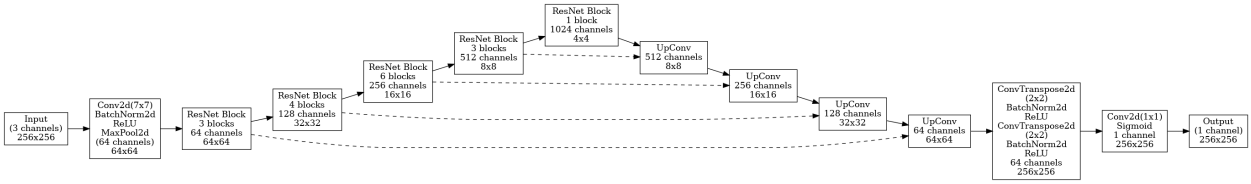


Figure 2: ResNet34Unet Architecture

For the left part of the ResNet34Unet, it is the same as the original ResNet34 model. And for the right part, it is the UNet model with two extra ConvTranspose2d layers to upsample the feature map to the original size (since the input is lowered by 4 times at the first layer of the ResNet34, so we need to upsample it by 4 times to get the original size).

To break down the ResNet34Unet, it can be divided into 3 parts:

1. ResBlock: The Residual Block of the ResNet34, used to extract the features from the input.
2. UpConv: The upsampling path of the UNet, which is a ConvTranspose2d layer with bilinear upsampling layers, used to upsample the feature map to the original size.

ResBlock The ResBlock layer is the same as the Residual Block of the ResNet34 model.

There is a classmethod `make_layer` to make a series of ResBlock layers, which is used to make a series of ResBlock layers, and automatically add the downsampling path to the first ResBlock layer when the channel number is doubled.

```
class ResBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, down_sample:
    bool = False):
        super(ResBlock, self).__init__()

        self.residual = nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size=3,
                padding=1,
                stride=2 if down_sample else 1,
                bias=False,
            ),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
            padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
        )
```

```
19
20          self.shortcut = (
21              nn.Sequential(
22                  nn.Conv2d(
23                      in_channels,
24                      out_channels,
25                      kernel_size=1,
26                      bias=False,
27                      stride=2 if down_sample else 1,
28                  ),
29                  nn.BatchNorm2d(out_channels),
30              )
31              if down_sample
32              else nn.Identity()
33          )
34
35          self.relu = nn.ReLU(inplace=True)
36
37      def forward(self, x):
38          return self.relu(self.residual(x) + self.shortcut(x))
39
40      @classmethod
41      def make_layer(
42          cls, in_channels: int, out_channels: int, blocks: int,
            down_sample: bool = False
43      ):
44          layers = [cls(in_channels, out_channels, down_sample)]
45          for _ in range(1, blocks):
46              layers.append(cls(out_channels, out_channels))
47          return nn.Sequential(*layers)
```

Listing 11: models/resnet34_unet.py: ResBlock

UpConv Same as one in the UNet model (see 8).

ResNet34Unet The ResNet34Unet model is a combination of the ResBlock and UpConv layers. The output activation function is sigmoid, so we can get the binary segmentation mask and use Binary Cross Entropy loss function.

The implementation is different from the one in the Lab2 Spec, since the original UNet concatenates tensors with same dimensions, so I think we should follow this rule. To meet this requirement, I use another ResBlock as the bottleneck, then forward the output with the previous ResBlocks' output to the UNet decoder.

By following this rule, the input of the first UpConv layer is by default 1024 dim (bottleneck output) $-$(ConvTranspose2d)$->$ 512 dim + 512 dim (last ResBlock output) = 1024 dim, and undergoes a DoubleConv layer to reduce the channel dimension to 512 dim.

So, according to the above description, the implementation is as follows:

```
1 class ResNet34Unet(nn.Module):
2     def __init__(
3         self,
4         in_channels: int = 3,
5         out_channels: int = 1,
```

```
        blocks: list[int] = [3, 4, 6, 3],
        down_channels: list[int] = [64, 64, 128, 256, 512],
        up_channels: list[int] = [1024, 512, 256, 128, 64],
    ):
        super(ResNet34Unet, self).__init__()

        self.in_conv = nn.Sequential(
            nn.Conv2d(
                in_channels,
                down_channels[0],
                kernel_size=7,
                padding=3,
                stride=2,
                bias=False,
            ),
            nn.BatchNorm2d(down_channels[0]),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )

        self.down = nn.ModuleList()
        for i in range(len(down_channels) - 1):
            self.down.append(
                ResBlock.make_layer(
                    down_channels[i],
                    down_channels[i + 1],
                    blocks[i],
                    down_sample=(down_channels[i] != down_channels[i +
                    1]),
                )
            )

        self.mid = ResBlock.make_layer(
            down_channels[-1],
            up_channels[0],
            1,
            down_sample=(down_channels[-1] != up_channels[0]),
        )

        self.up = nn.ModuleList()
        for i in range(len(up_channels) - 1):
            self.up.append(UpConv(up_channels[i], up_channels[i + 1]))

        self.out = nn.Sequential(
            nn.ConvTranspose2d(
                up_channels[-1], up_channels[-1], kernel_size=2, stride=2
            ),
            nn.BatchNorm2d(up_channels[-1]),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(
                up_channels[-1], up_channels[-1], kernel_size=2, stride=2
            ),
            nn.BatchNorm2d(up_channels[-1]),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(up_channels[-1], out_channels, kernel_size=1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.in_conv(x)
        x_rec = []
        for layer in self.down:
            x = layer(x)
            x_rec.append(x)
        x = self.mid(x)
        for layer in self.up:
            x = layer(x, x_rec.pop())
        return self.out(x)
```

Listing 12: models/resnet34_unet.py: ResNet34Unet

### 1.5 Loss Function

For loss function, two loss functions are used: Binary Cross Entropy loss function and Dice loss function.

Binary Cross Entropy loss function The Binary Cross Entropy loss function is a loss function that measures the performance of a binary classification model, so it is suitable for the binary segmentation task. Since it is implemented in PyTorch, so we won't need to dig into the details of the implementation.

Dice loss function The Dice loss function is a loss function that measures the performance of a binary segmentation model, so it is suitable for the binary segmentation task.

$$\text{DiceLoss}(p, y) = 1 - \frac{2\sum_{i=1}^{N} y_i p_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} p_i} \tag{1}$$

and surprisingly, the gradient of the DiceLoss can be represented as a simple formula (in the referenced repository):

$$\frac{\partial \text{DiceLoss}(p, y)}{\partial p} = 1 - DiceLoss(p, y) \tag{2}$$

after we figure out the formula of the loss and its gradient, we can implement it in the code.

```
def dice_score(pred_mask, gt_mask):
    # implement the Dice score here
    assert pred_mask.shape == gt_mask.shape

    if pred_mask.ndim == 3: # expand (C, H, W) -> (1, C, H, W)
        pred_mask = pred_mask.unsqueeze(1)

    pred_mask = torch.where(pred_mask > 0.5, True, False)
    gt_mask = torch.where(gt_mask > 0.5, True, False)

    common = torch.sum(pred_mask & gt_mask, dim=(1, 2, 3)) # common
    pixels between pred and gt
```

```
    union = torch.sum(pred_mask, dim=(1, 2, 3)) + torch.sum(gt_mask,
    dim=(1, 2, 3))

    union = torch.where(union == 0, 1, union) # avoid division by zero

    return (2 * common / union).mean() # average over the batch


def dice_loss(pred_mask, gt_mask):
    return 1 - dice_score(pred_mask, gt_mask)
```

Listing 13: utils.py: dice_loss

# 2 Data Preprocessing

## 2.1 Preprocessing

For preprocessing, I use the following steps:

1. Resize the image to 256x256
2. Randomly crop the image to 256x256 or rotate the image by at most 30 degrees
3. Randomly flip the image horizontally or vertically
4. Randomly adjust the brightness or gamma of the image
5. Randomly add Gaussian noise or blur to the image
6. Normalize the image with the mean and standard deviation of the ImageNet dataset

and implement it in the `load_dataset` function using the functions from `albumentations` library.

```
A.Compose([
    A.Resize(256, 256),
    A.OneOf([
        A.RandomResizedCrop((256, 256), scale=(0.8, 1.0), ratio=(0.75,
        1.33), p=1.0),
        A.Rotate(limit=30, p=1.0),
    ], p=0.5),
    A.OneOf([
        A.HorizontalFlip(p=1.0),
        A.VerticalFlip(p=1.0),
    ], p=0.5),
    A.OneOf([
        A.RandomBrightnessContrast(p=1.0),
        A.RandomGamma(p=1.0),
    ], p=0.3),
    A.OneOf([
        A.GaussNoise(p=1.0),
        A.GaussianBlur(p=1.0),
    ], p=0.2),
    A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ToTensorV2(),
```

```
21 ])
```

Listing 14: oxford_pet.py: load_dataset

### 2.2 What makes the preprocessing method unique

I choose some transformations of geometric, photometric, and noise to augment the data. But if we impose too many transformations, the image would be too different from the original image, and the model would not be able to learn the features.

So, I use the `OneOf` function to choose one of the transformations to apply to the image. And use the parameter $p$ to control the probability of each transformation, prevent the image from being too different.

By this method, the model can learn from moderately augmented data, and the performance is better than using only the original data.

## 3 Analyze the experiment results

### 3.1 Training Curves

The training curves are shown in the following figure 3.

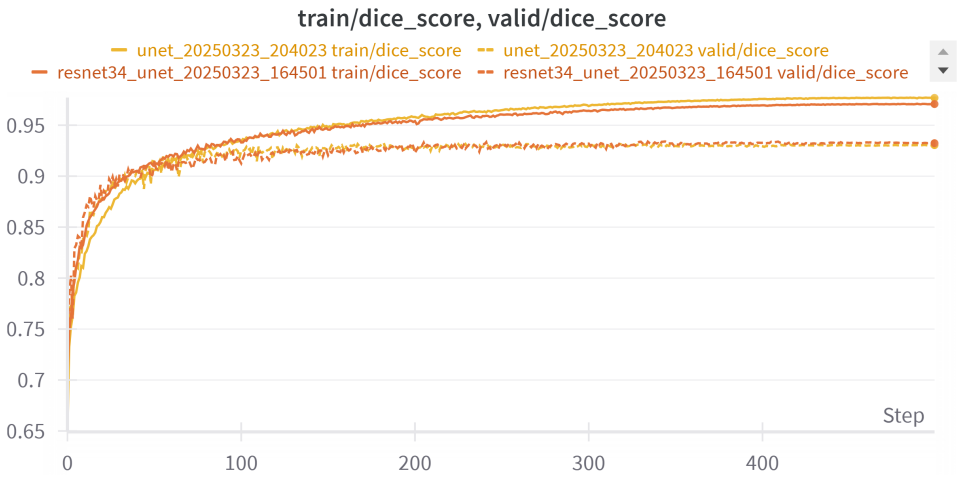


Figure 3: Training curves: Dice score. X-axis is the number of epochs, Y-axis is the Dice score.

There two pairs of curves, one is for ResNet34 UNet (Orange) and the other is for UNet (Yellow). And the solid line is the dice score of the training set, and the dashed line is the dice score of the validation set. We can see from the figure, the performance of the two models are almost the same, but the ResNet34 UNet converges faster than UNet, and UNet finally convergs to a higher train accuracy than ResNet34 Unet. But the validation accuracy of ResNet34 UNet is higher than UNet, which means that ResNet34 UNet is less likely to overfit than UNet.

### 3.2 Performance Metrics

I use the Dice score on test set to evaluate the performance of the model. With the parameter given in the execution section 17, the performance of the model is shown in the following table 1

and figure 4.

| Model | Dice score |
|---|---|
| UNet | 0.9318 |
| ResNet34 UNet | 0.9330 |

Table 1: Dice score of the test set. The above is the ResNet34 UNet, and the below is the UNet.


Figure 4: Dice score of the test set. The above is the UNet, and the below is the ResNet34 UNet.

### 3.3 Visualization of Segmentation Results

The visualization of the segmentation results is shown in the following figure. There are five samples in the figure, the first row is the original image, the second row is the ground truth mask, the third row is the prediction from the UNet, and the fourth row is the prediction from the ResNet34 UNet.

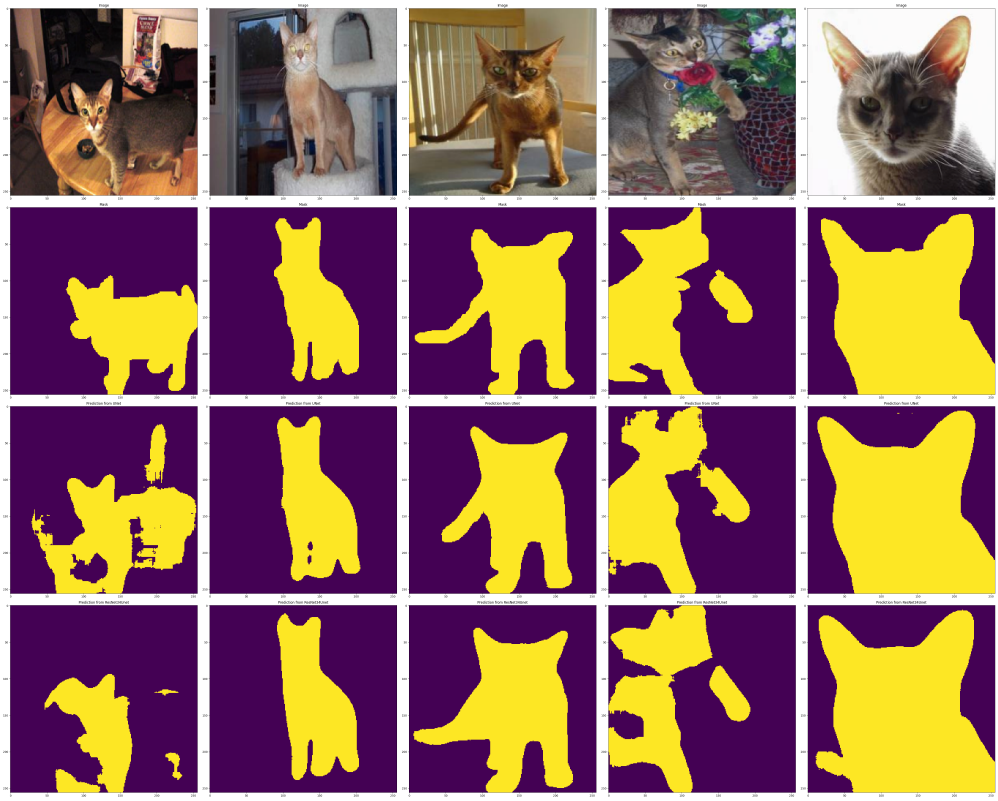

Figure 5: Visualization of segmentation results. The above is the UNet, and the below is the ResNet34 UNet.

We can see from the figure, Resnet34 UNet has a better performance than UNet, it can predict the small objects better, and it also performs when dealing with the boundary of the objects. This

behavior is more obvious in sample 1 and 3, where the prediction from Unet is totally distorted, but the prediction from Resnet34 UNet is still good.

### 3.4 Discussion

Throughout this project, I firstly experiment with both of the two models, but I found that ResNet34 UNet has a 3x faster training speed than UNet, so I use ResNet34 UNet for following experiments.

Secondly, I conducted experiments on learning rate, batch size, and number of epochs, and I found that the learning rate have a significant impact on the performance of the model, but the batch size and number of epochs have a little impact. For example, when the learning rate is set to 1e-4 to 1e-3, the performance of ResNet34 UNet about the same, as for learnging rate 1e-5, it converges too slow that it cant converge to a good accuracy in 500 epochs. The batch size is set to 64, is not because it leads to a better accuracy, but because it leads to A LOT faster training speed without out of memory error. And the number of epochs is set to 500, is because I found that the performance of the model is almost saturated after 500 epochs, and it is not necessary to train the model for more epochs.

After I decided the parameters, I conducted experiments on the transformation of the images, and I found that the performance of the model is not sensitive to the transformation of the images. Originally, I conducted lots of transformations on the training image, and the performance is almost the same as those experiment with half of the transformations. After then, I found that normalization and rotation have a significant impact on the performance of the model, and other transformations have little impact on the performance of the model. So, as a result, I kept some of the transformations, and using OneOf to prevent over-transformation, and normalize the image at the end.

## 4 Execution steps

### 4.1 How to run the code

Before running the code, make sure you are in the right directory.

```
$ tree

├── dataset
├── saved_models
└── src
```

You can run the code by executing the following command:
First, install the dependencies:

```
pip install -r requirements.txt
```

Listing 15: Install the dependencies

Then, run the training code:

```
# Run the code for training
python src/train.py --data_path dataset/oxford-iiit-pet/
```

Listing 16: How to run the training code

there are other parameters you can use to customize the training process:

- `--device`: the device to run the code on (default: cuda:1)
- `--seed`: the seed for the random number generator (default: 88)
- `--epochs`: the number of epochs to train the model (default: 500)
- `--batch_size`: the batch size for the training process (default: 64)
- `--learning_rate`: the learning rate for the training process (default: 7e-4)
- `--model`: the model to use for the training process (default: resnet34_unet), you can choose from: resnet34_unet, unet
- `--wandb`: whether to use wandb for logging (default: True)

For my own results, I used the following commands:

```
# ResNet34 UNet
python src/train.py \
    --data_path dataset/oxford-iiit-pet/ \
    -d cuda:1 \
    -s 228922

# UNet
python src/train.py \
    --data_path dataset/oxford-iiit-pet/ \
    -d cuda:1 \
    -s 228922 \
    -m unet
```

Listing 17: How to run the training code for my own results

After training, you can run the inference code:

```
# Run the code for inference
# Replace the model_path with the path to the model you want to use for the inference
# Model name should contain resnet34_unet or unet
python src/inference.py --data_path dataset/oxford-iiit-pet/ --model_path saved_models/model.pth
```

Listing 18: How to run the inference code

# 5 Discussion

## 5.1 Alternative architectures

I think in this task, we can use more complex architectures to improve the performance of the model, since the input image contains various conditions, including different brightness, different contrast, different size, different angle, and the object itself can have a weird shape (especially cats).

So, as an alternative, we can conduct experiments on the most famous model: Transformer. We can leverage its powerful attention mechanism to improve the performance of the model, let it more focus on the important parts of the image. And more on, we can use Mixure-of-Experts to improve the performance of the model, let it more robust to different conditions.

### 5.2 Potential Research Topics

Using transformer is a good idea, but everyone is using it nowadays, and the parameters is getting larger and larger. So, I think it is a good idea to distill the trained transformer model on this task into a smaller model, trying to keep the performance of the model and reduce the parameters. Or, on the other hand, we can try to use a small model, but try some fancy method to deal with the input image with various conditions, finally get comparable performance with transformer-based models.

## 6 References

milesial/Pytorch-UNet