Deep Learning Lab2

110550085 房天越

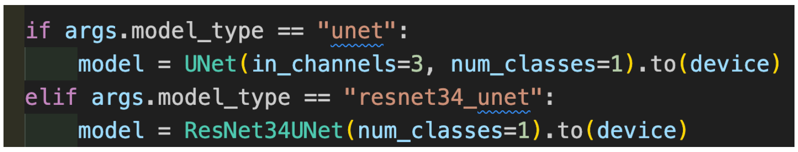
1. Implementation Details

I. Training, Evaluation, and InferenceT

A. Training

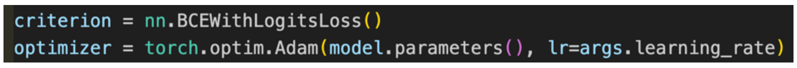
For the training part (and the inference part), I added an argument called model_type, which tells the code whether to use unet or resnet34_unet.

First, the model loads the dataset from oxford_pet.py. Then dothe model selection.



```
if args.model_type == "unet":
    model = UNet(in_channels=3, num_classes=1).to(device)
elif args.model_type == "resnet34_unet":
    model = ResNet34UNet(num_classes=1).to(device)
```

For the training process, I use Binary Cross Entropy with Logits Loss as the loss function, and Adam as the optimizer.



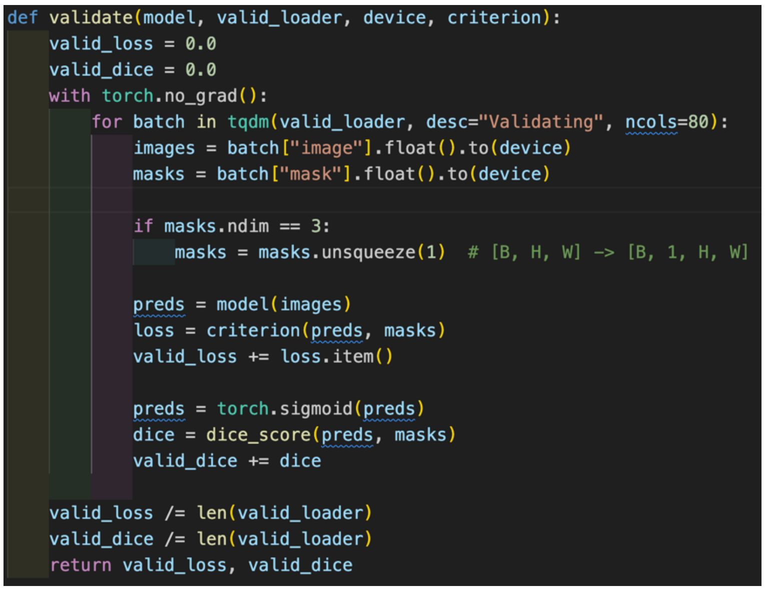
```
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
```

For the training loop, I use tqdm for progress monitoring and performs backpropagation and weight updates per batch.

Finally, if there is an improvement in dice score, which could be known by calling the evaluation function(validate), we save the model.

B. Evaluation

For this part, it does the evaluation for model performance using dice score for the validation dataset. I use sigmoid activation and thresholding for mask predictions.



```python
def validate(model, valid_loader, device, criterion):
    valid_loss = 0.0
    valid_dice = 0.0
    with torch.no_grad():
        for batch in tqdm(valid_loader, desc="Validating", ncols=80):
            images = batch["image"].float().to(device)
            masks = batch["mask"].float().to(device)

            if masks.ndim == 3:
                masks = masks.unsqueeze(1)  # [B, H, W] -> [B, 1, H, W]

            preds = model(images)
            loss = criterion(preds, masks)
            valid_loss += loss.item()

            preds = torch.sigmoid(preds)
            dice = dice_score(preds, masks)
            valid_dice += dice

    valid_loss /= len(valid_loader)
    valid_dice /= len(valid_loader)
    return valid_loss, valid_dice
```

C. Inference

In this part, we do inference for the test dataset, and finally save the pictures as .png files. The method is like the training part.

D. Some other details

I use set_seed(42) to make the results reproduceable, and use efficient data loading using num_workers parameter.

E. UNet

For UNet, basically I implemented it based on the structure given in the spec, which includes four down blocks and four up blocks. There is also a bottleneck in the middle to handle the upscaling.

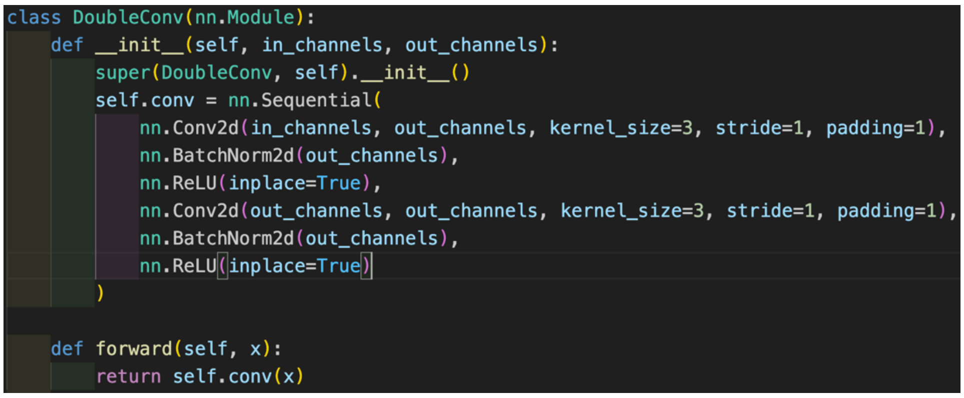
And here is a video I took for reference:

UNet 教學

Let’s break the structure into smaller parts:

A. DoubleConv

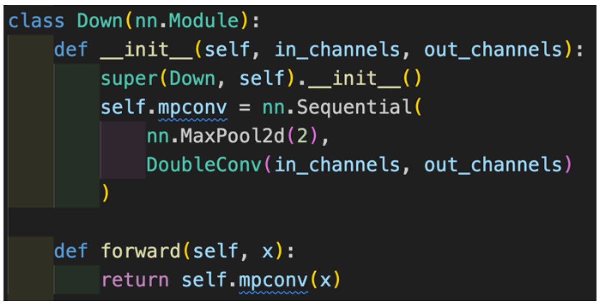
This module is the basic unit of UNet, which use two consecutive convolutional layers to extract features, padding = 1 maintains the spatial dimensions of the feature maps.



```python
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

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

B. Down

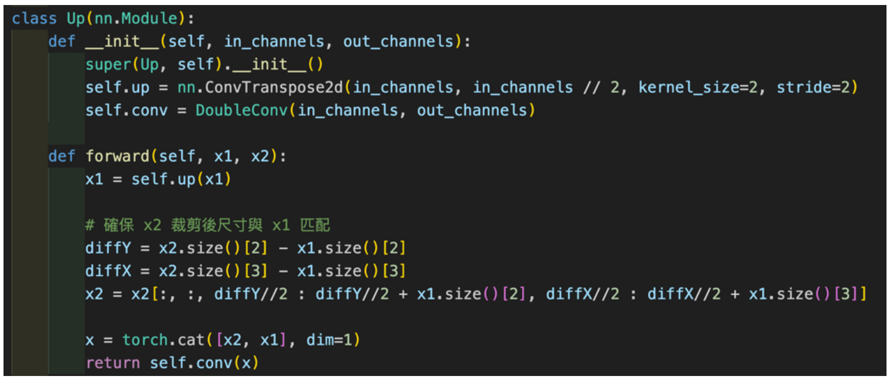
For each down module, first I apply a 2*2 max pooling to reduce the spatial dimensions, followed by a DoubleConv module to further extract features. This helps capture the global semantic information of the image.



```python
class Down(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Down, self).__init__()
        self.mpconv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

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

C. Up

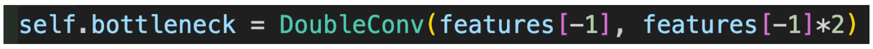
For the upsampling part, a transposed convolution is used to upscale the feature map, and the upsampled features are concatenated with the corresponding skip connection from the encoder. A DoubleConv module then fuses these features. Cropping or padding is applied to ensure matching dimensions.



```python
class Up(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Up, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)

        # 確保 x2 裁剪後尺寸與 x1 匹配
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]
        x2 = x2[:, :, diffY//2 : diffY//2 + x1.size()[2], diffX//2 : diffX//2 + x1.size()[3]]

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

D. Bottleneck

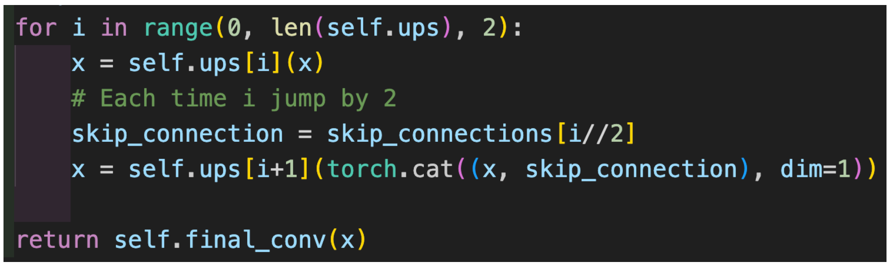
The bottleneck is a structure between the encoder and decoder, the bottleneck uses an intensified DoubleConv structure responsible for capturing the deepest features.



```
self.bottleneck = DoubleConv(features[-1], features[-1]*2)
```

E. Final Convolution

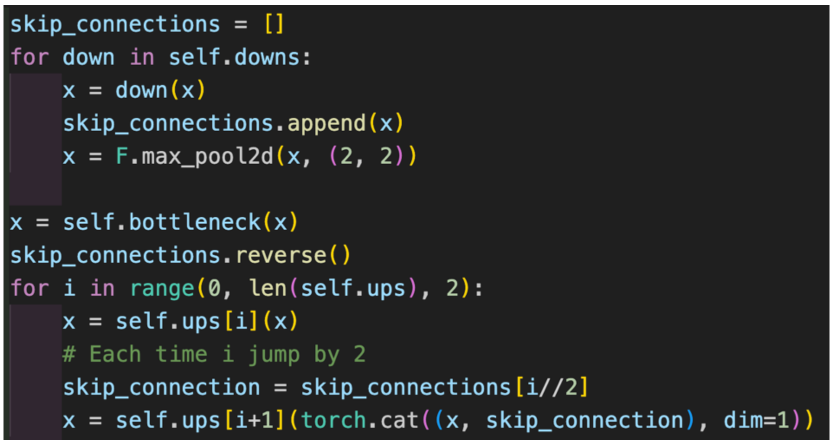
Finally, I use a 1*1 convolution to transform the result to my final prediction.



```
for i in range(0, len(self.ups), 2):
    x = self.ups[i](x)
    # Each time i jump by 2
    skip_connection = skip_connections[i//2]
    x = self.ups[i+1](torch.cat((x, skip_connection), dim=1))

return self.final_conv(x)
```

F. Skip Connections

This is a special part that passes the characteristic to the decoder, this can help recover spatial details which are crucial for segmentation tasks.



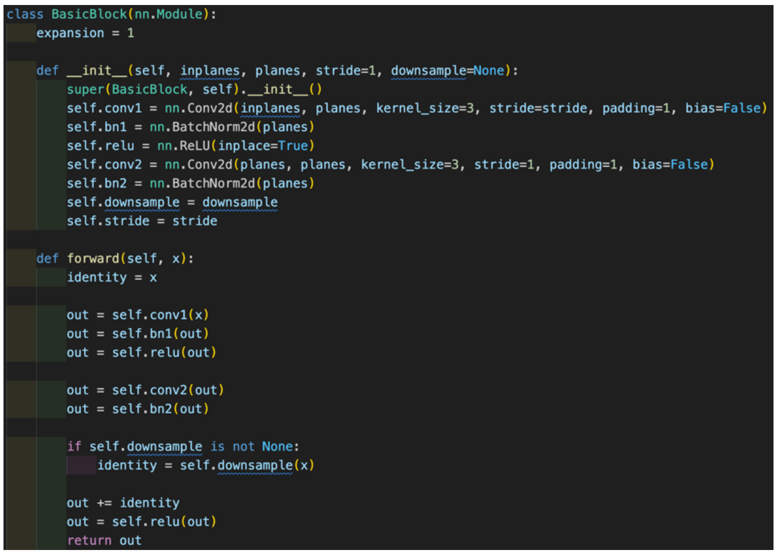
```
skip_connections = []
for down in self.downs:
    x = down(x)
    skip_connections.append(x)
    x = F.max_pool2d(x, (2, 2))

x = self.bottleneck(x)
skip_connections.reverse()
for i in range(0, len(self.ups), 2):
    x = self.ups[i](x)
    # Each time i jump by 2
    skip_connection = skip_connections[i//2]
    x = self.ups[i+1](torch.cat((x, skip_connection), dim=1))
```

F. UNet with ResNet34 as backbone (ResNet34UNet)

For ResNet34UNet, the main difference is to use ResNet34 as the encoder. Let's also break into smaller parts of it.

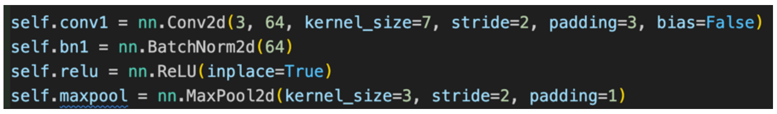
A. Use ResNet34 as the encoder

ResNet34 starts with initial convolution and maxpooling, followed by multiple BasicBlocks with residual connections which help mitigate gradient vanishing in deep networks.



```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
        return out
```

(Basic Block)

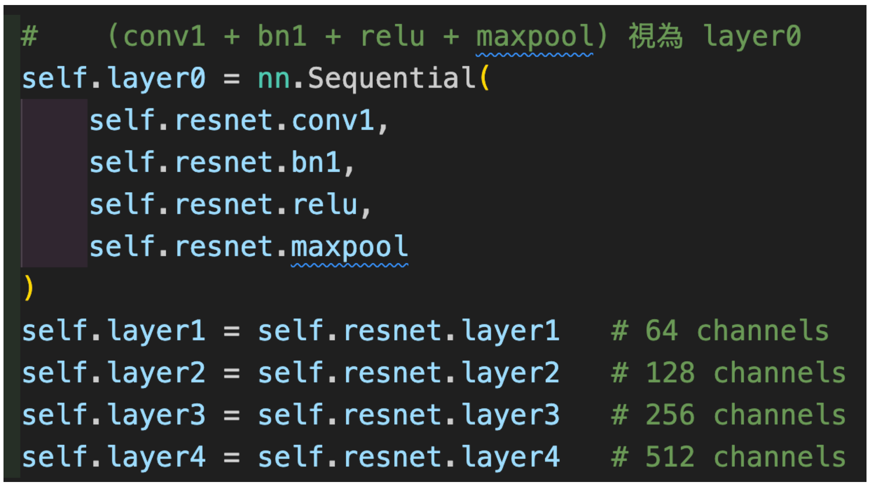


```python
self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
self.bn1 = nn.BatchNorm2d(64)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```

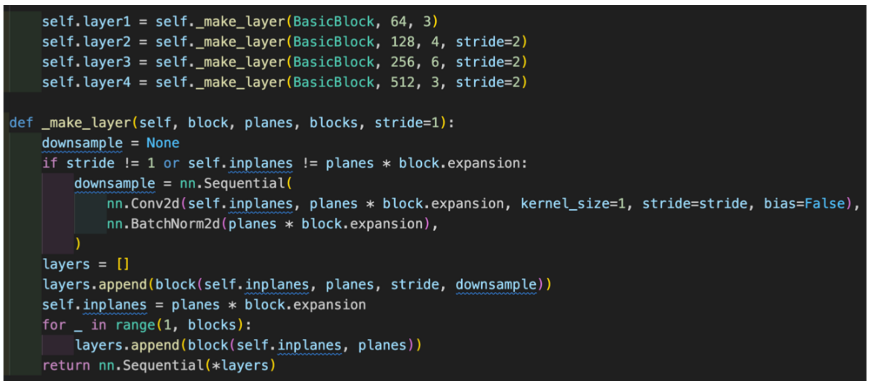
(ResNet34 initial structure)

B. Layered Encoder

The ResNet34 structure is divided into 4 stages, from layer0 to layer4. The spatial dimensions reduce progressively, capturing features from local to global scales.



```python
#    (conv1 + bn1 + relu + maxpool) 視為 layer0
self.layer0 = nn.Sequential(
        self.resnet.conv1,
        self.resnet.bn1,
        self.resnet.relu,
        self.resnet.maxpool
)
self.layer1 = self.resnet.layer1   # 64 channels
self.layer2 = self.resnet.layer2   # 128 channels
self.layer3 = self.resnet.layer3   # 256 channels
self.layer4 = self.resnet.layer4   # 512 channels
```

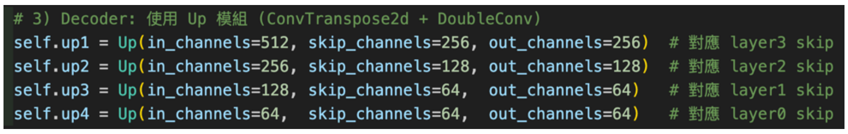


```
    self.layer1 = self._make_layer(BasicBlock, 64, 3)
    self.layer2 = self._make_layer(BasicBlock, 128, 4, stride=2)
    self.layer3 = self._make_layer(BasicBlock, 256, 6, stride=2)
    self.layer4 = self._make_layer(BasicBlock, 512, 3, stride=2)

def _make_layer(self, block, planes, blocks, stride=1):
    downsample = None
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            nn.Conv2d(self.inplanes, planes * block.expansion, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(planes * block.expansion),
        )
    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes))
    return nn.Sequential(*layers)
```

C. Upsampling Module in Decoder

The decoder uses Up modules that employ transposed convolution for upsampling, followed by concatenation with the corresponding encoder features by skip connection.

The difference between this model and UNet model is that here we should also handle the channel dimension alignment between different layers using parameters like in_channels, skip_channels, and out_channels.



```
# 3) Decoder: 使用 Up 模組 (ConvTranspose2d + DoubleConv)
self.up1 = Up(in_channels=512, skip_channels=256, out_channels=256)  # 對應 layer3 skip
self.up2 = Up(in_channels=256, skip_channels=128, out_channels=128)  # 對應 layer2 skip
self.up3 = Up(in_channels=128, skip_channels=64,  out_channels=64)   # 對應 layer1 skip
self.up4 = Up(in_channels=64,  skip_channels=64,  out_channels=64)   # 對應 layer0 skip
```

D. Final Output and Size Restoration

Finally, we use a 1*1 convolution to convert the channels into the desired number of classes. Bilinear interpolation is then used to restore the output to the original input dimensions, ensuring segmentation output matches the input size.

2. Data Preprocessing

This part of code is defined in oxford_pet.py. According to the difference between modes, two kinds of operations are taken.

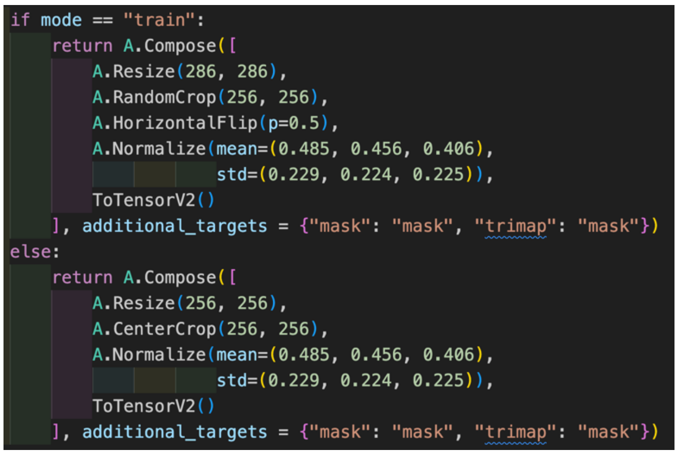
For the training part, the pictures are first resized to a slightly larger size than 256*256, then we do random crop to make it 256*256 again. Then, we do a random horizontal flip with p = 0.5, and finally normalize it. By doing so, we can improve the variety in the training dataset, which helps the model to recognize the patterns more, and could finally give us a better result.

And for the evaluation and testing part, we just resize the pictures into 256*256, do a center crop and then normalize them.

The normalization parameters is according to the statistical results made by

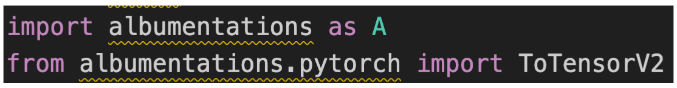
ImageNet, including the mean and standard deviation.

Here's the key part of code:



```
if mode == "train":
    return A.Compose([
        A.Resize(286, 286),
        A.RandomCrop(256, 256),
        A.HorizontalFlip(p=0.5),
        A.Normalize(mean=(0.485, 0.456, 0.406),
                    std=(0.229, 0.224, 0.225)),
        ToTensorV2()
    ], additional_targets = {"mask": "mask", "trimap": "mask"})
else:
    return A.Compose([
        A.Resize(256, 256),
        A.CenterCrop(256, 256),
        A.Normalize(mean=(0.485, 0.456, 0.406),
                    std=(0.229, 0.224, 0.225)),
        ToTensorV2()
    ], additional_targets = {"mask": "mask", "trimap": "mask"})
```

To do so we need to import:



```
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

Using albumentations is the most special part, and it can significantly increase the variety of the training data and yields better results.

For this part, I took this GitHub repo for reference:

Hank891008-GitHub_Repo

3. Analyze the experiment results

A. Different hyperparameters

For both UNet and ResNet34_UNet, I use learning rate = 0.0001. I've tried 0.00001, this would make the training process extremely long for the model to converge.

I ran 100 epochs with batch size 12 for UNet, and 200 epochs with batch size 32 for ResNet34UNet.

B. Something I discovered during the training process

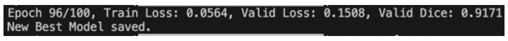
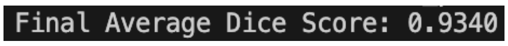
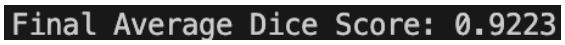
Before I actually train the models, I thought ResNet34UNet would take a longer time to train because there are more blocks and needs more computation. However, it turns out that UNet takes a significantly longer time than ResNet34UNet. I think this might be because in the implementation of UNet, each encoder layer preserves larger feature maps for skip connection. Also, ResNet has residual connections and deeper BasicBlocks, which help extract features more efficiently.

C. Some Special Characteristics of the dataset

In this dataset, the objects are significantly larger than common image semantic segmentation tasks. For the common tasks, since we want only a little part of the image, Focal Loss might have a better performance. However, for this project, using BCE Loss might have a better performance because the objects account for a larger part of the picture.

D. Experimental Results

For both UNet and ResNet34UNet, they have an accuracy more than 0.9 on both validation dataset and test dataset.

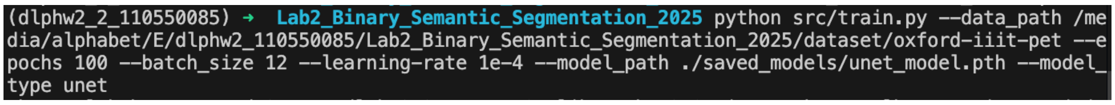
	UNet	ResNet34_UNet
Valid	 `Epoch 96/100, Train Loss: 0.0564, Valid Loss: 0.1508, Valid Dice: 0.9171` `New Best Model saved.`	At epoch 196, Dice Score is about 0.91
Test	 `Final Average Dice Score: 0.9340`	 `Final Average Dice Score: 0.9223`

We can see that UNet has a better performance than ResNet34_UNet even though it has fewer epochs than ResNet34_UNet, but ResNet34_UNet trains significantly faster, so it's a trade-off between efficiency and performance.

4. Execution Steps

A. Training

To reproduce the model, please use commands like, a text version is as follows:



```
(dlphw2_2_110550085) ➜  Lab2_Binary_Semantic_Segmentation_2025 python src/train.py --data_path /me
dia/alphabet/E/dlphw2_110550085/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet --e
pochs 100 --batch_size 12 --learning-rate 1e-4 --model_path ./saved_models/unet_model.pth --model_
type unet
```

python src/train.py --data_path /media/alphabet/E/dlphw2_110550085/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet --epochs 100 --batch_size 12 --learning-rate 1e-4 --model_path ./saved_models/unet_model.pth --model_type unet

Here are parameters you can adjust:

--data_path, this is the path where the dataset locates.

--epochs, this is the number of epochs you want to train.

--batch_size, this is the size of batch you want to use.

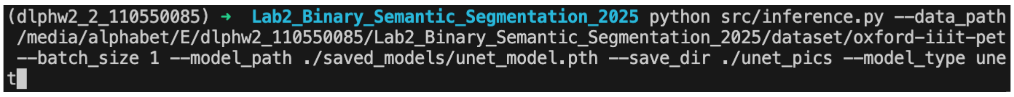
--learning_rate, this is the learning rate you can adjust.

--model_path, this is the model path to save, also if there is an existing model for the given route, then the training code would use it and keep training.

--model_type, this is the model type to use, you can choose unet or resnet34_unet

B. Inference

For inference, you can use a similar command like training, a text version is also given below:



```
(dlphw2_2_110550085) ➜  Lab2_Binary_Semantic_Segmentation_2025 python src/inference.py --data_path
 /media/alphabet/E/dlphw2_110550085/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet
 --batch_size 1 --model_path ./saved_models/unet_model.pth --save_dir ./unet_pics --model_type une
t
```

python src/inference.py --data_path /media/alphabet/E/dlphw2_110550085/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet --batch_size 1 --model_path ./saved_models/unet_model.pth --save_dir ./unet_pics --model_type unet

Here are parameters you can adjust:

--data_path, this is the path where the dataset locates.

--batch_size, this is the size of batch you want to use.

--model_path, this is the model path to save, also if there is an existing model for the given route, then the training code would use it and keep training.

--model_type, this is the model type to use, you can choose unet or resnet34_unet

-- save_dir, this is the path you want to store the results.

5. Discussion

A. Some Alternative Structures

I. Attention U-Net / Transformer-based models

These structures incorporate attention or transformers to capture global dependencies, enhancing fine segmentation details.

II. Hybrid Models

Combining CNNs with RNNs or GNNs might better model spatial and contextual relationships for certain tasks.

B. Potential Research Directions

This task has a high potential when we want to retrieve a special object in a large picture (finding ROI), after retrieving the part we are interested in, we can reduce the computational resources and still have good performances.

Maybe the detection of pedestrians, or other obstacles on the road for the auto-driving cars can used some methods like this.