Binary Semantic Segmentation

Lab #2 report

B103040039

王宗義

Deep learning

Spring 2025

0. Introduction

0.1 Problem Statement

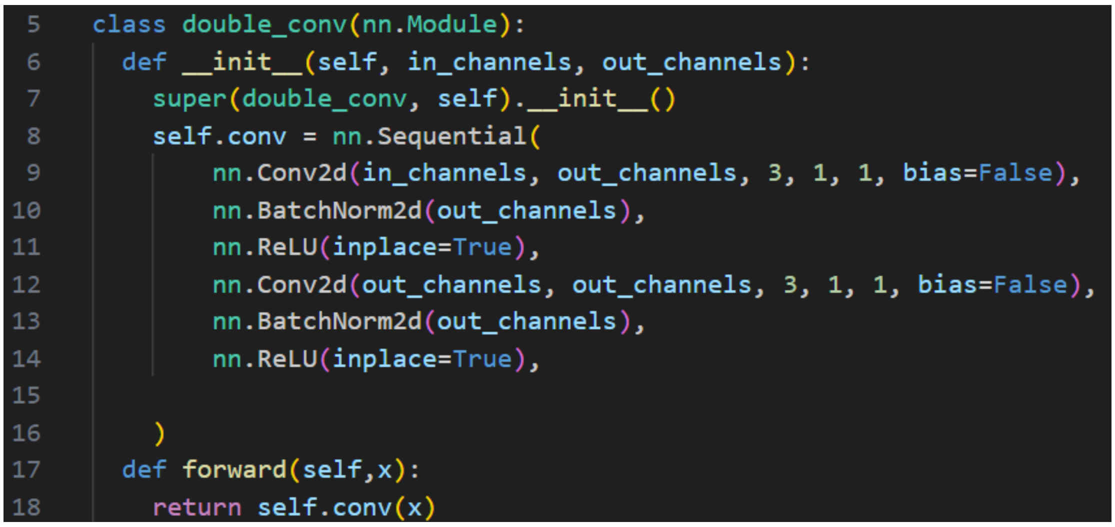
The goal of this lab is to accomplish binary semantic segmentation task. To do this, we need to implement two types of model regarding binary semantic segmentation (Unet, Resnet34_unet), training them with Oxford-IIIT Pet Dataset. We also hope the trained models can classify the background and object in the test data. In this experiment, we also introduce dice score to evaluate the quality of the models.

1. Implementation Details

1.1 models

1.1.1 Unet

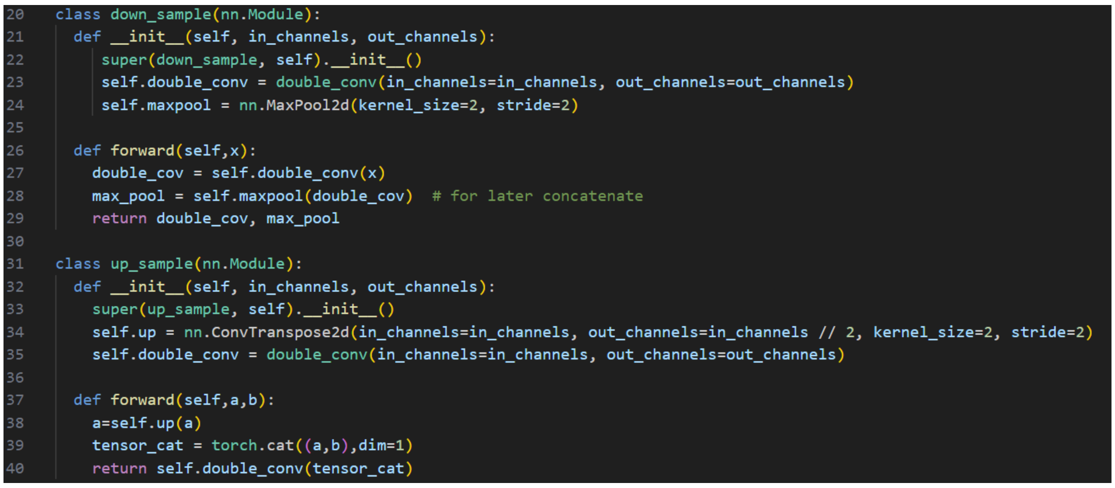
According the structure of Unet, it is clear that the network contains lots of double convolution layer, so we first construct this kind of pattern, taking this as a block. Check graph 1 for more detail.



```python
class double_conv(nn.Module):
  def __init__(self, in_channels, out_channels):
    super(double_conv, self).__init__()
    self.conv = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),

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

Graph 1. Double convolution block

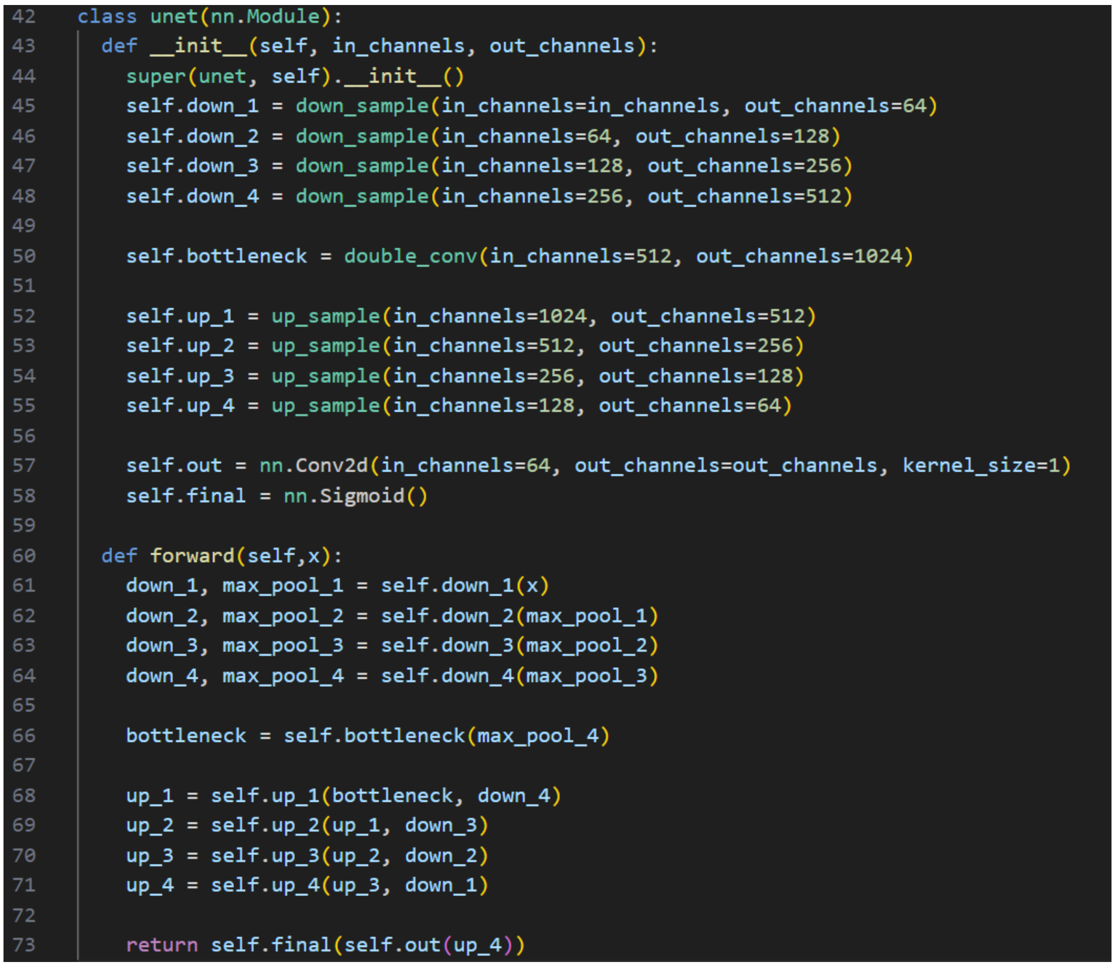
Furthermore, we can also perceive that there are three stages in Unet (up, down, bottom). The down stage consists of a “double convolution layer” and a “maxpool layer”, and the up stage consists of a “transpose convolution layer” and a “double convolution layer”, noting that this layer will concatenate other layer in the following implementation.



```python
class down_sample(nn.Module):
  def __init__(self, in_channels, out_channels):
    super(down_sample, self).__init__()
    self.double_conv = double_conv(in_channels=in_channels, out_channels=out_channels)
    self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

  def forward(self,x):
    double_cov = self.double_conv(x)
    max_pool = self.maxpool(double_cov)  # for later concatenate
    return double_cov, max_pool

class up_sample(nn.Module):
  def __init__(self, in_channels, out_channels):
    super(up_sample, self).__init__()
    self.up = nn.ConvTranspose2d(in_channels=in_channels, out_channels=in_channels // 2, kernel_size=2, stride=2)
    self.double_conv = double_conv(in_channels=in_channels, out_channels=out_channels)

  def forward(self,a,b):
    a=self.up(a)
    tensor_cat = torch.cat((a,b),dim=1)
    return self.double_conv(tensor_cat)
```

Graph 2. Down block and up block

The implementation of Unet is provided below, we can first build 4 down blocks, 4 up blocks, 1 bottleneck (bottom stage), and 1 output layer. Interestingly, in the forward function, the down layer before maxpooling will be saved and concatenate to up layer in up stage.

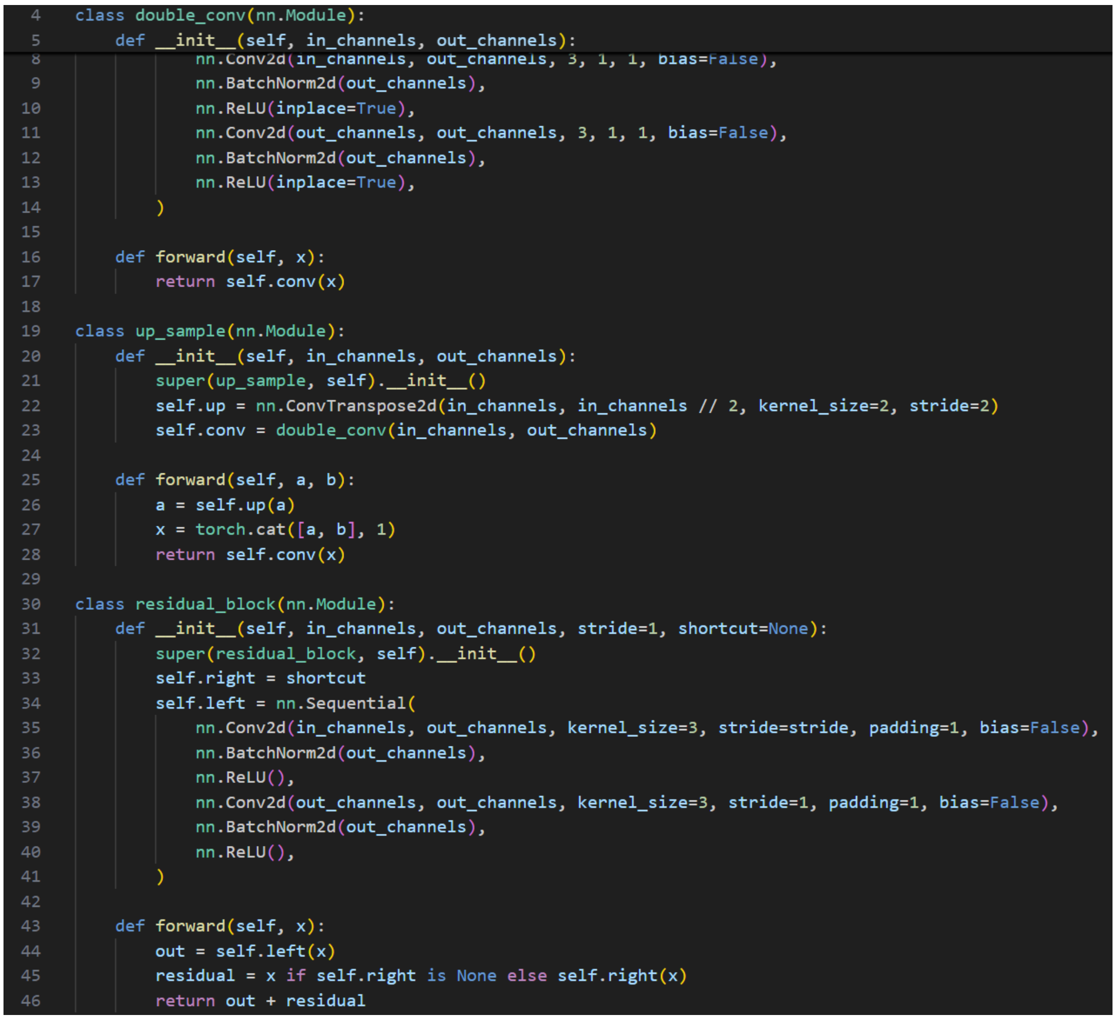


```python
class unet(nn.Module):
  def __init__(self, in_channels, out_channels):
    super(unet, self).__init__()
    self.down_1 = down_sample(in_channels=in_channels, out_channels=64)
    self.down_2 = down_sample(in_channels=64, out_channels=128)
    self.down_3 = down_sample(in_channels=128, out_channels=256)
    self.down_4 = down_sample(in_channels=256, out_channels=512)

    self.bottleneck = double_conv(in_channels=512, out_channels=1024)

    self.up_1 = up_sample(in_channels=1024, out_channels=512)
    self.up_2 = up_sample(in_channels=512, out_channels=256)
    self.up_3 = up_sample(in_channels=256, out_channels=128)
    self.up_4 = up_sample(in_channels=128, out_channels=64)

    self.out = nn.Conv2d(in_channels=64, out_channels=out_channels, kernel_size=1)
    self.final = nn.Sigmoid()

  def forward(self,x):
    down_1, max_pool_1 = self.down_1(x)
    down_2, max_pool_2 = self.down_2(max_pool_1)
    down_3, max_pool_3 = self.down_3(max_pool_2)
    down_4, max_pool_4 = self.down_4(max_pool_3)

    bottleneck = self.bottleneck(max_pool_4)

    up_1 = self.up_1(bottleneck, down_4)
    up_2 = self.up_2(up_1, down_3)
    up_3 = self.up_3(up_2, down_2)
    up_4 = self.up_4(up_3, down_1)

    return self.final(self.out(up_4))
```

Graph 3. Implementation of Unet

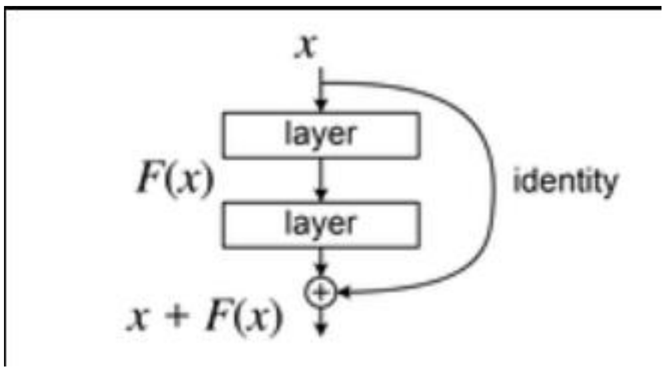
1.1.2 Resnet34_unet

The structure of Resnet34_unet combines the feature of Resnet with Unet, so the implementation might be a little bit complicated. To realize the structure, we can first create three types of blocks, double convolution block, up block, and residual block. The double convolution block and up block are similar to the structure in graph 1 and graph 2. In terms of the residual block [1] (see graph 5.), we can construct the block on the left and right first. The left block consists of 6 layers (2 * (conv + norm + relu)), while the right block can be any kind of layer sent to the residual block. The left and right block will be added together in the end of forwarding.



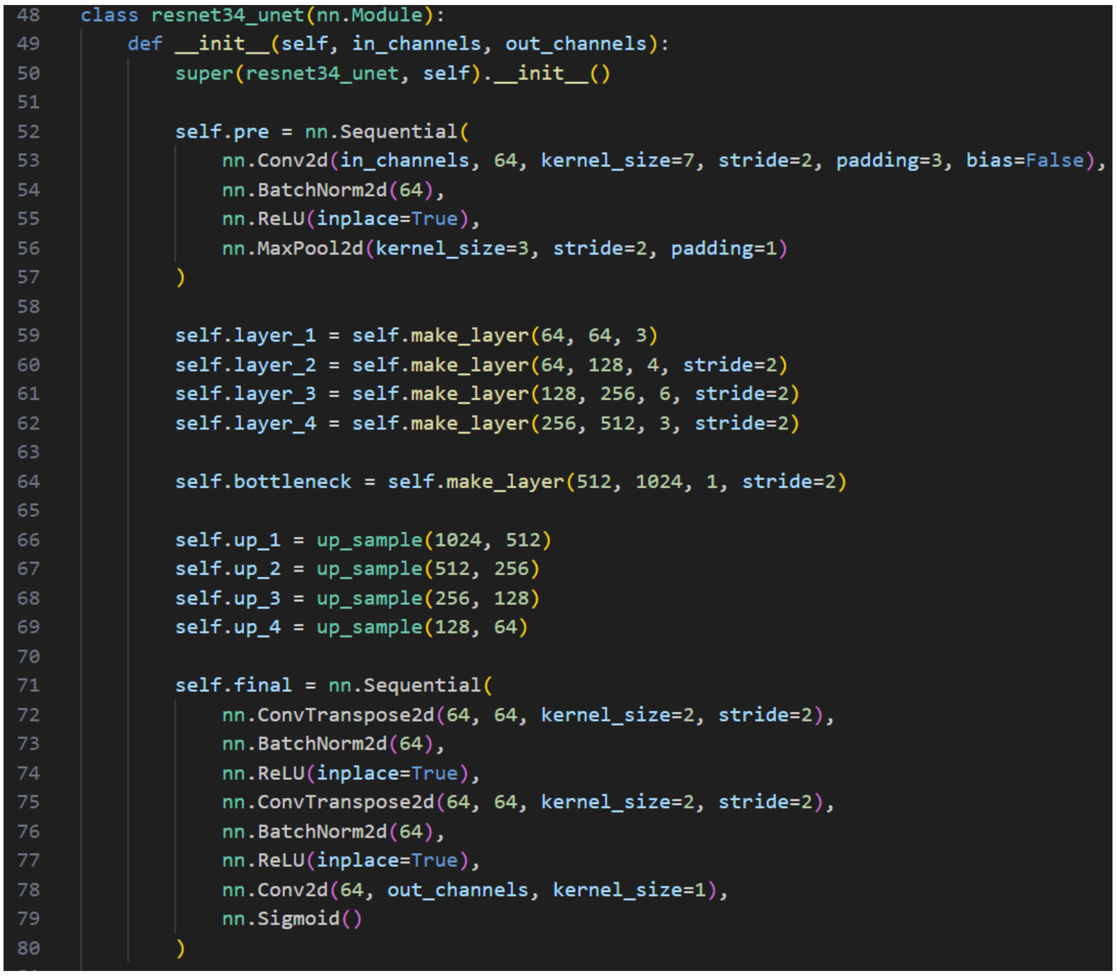
```python
class double_conv(nn.Module):
    def __init__(self, in_channels, out_channels):
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

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

class up_sample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(up_sample, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
        self.conv = double_conv(in_channels, out_channels)

    def forward(self, a, b):
        a = self.up(a)
        x = torch.cat([a, b], 1)
        return self.conv(x)

class residual_block(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, shortcut=None):
        super(residual_block, self).__init__()
        self.right = shortcut
        self.left = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )

    def forward(self, x):
        out = self.left(x)
        residual = x if self.right is None else self.right(x)
        return out + residual
```

Graph 4. Blocks in resnet34_unet



Graph 5. Residual block

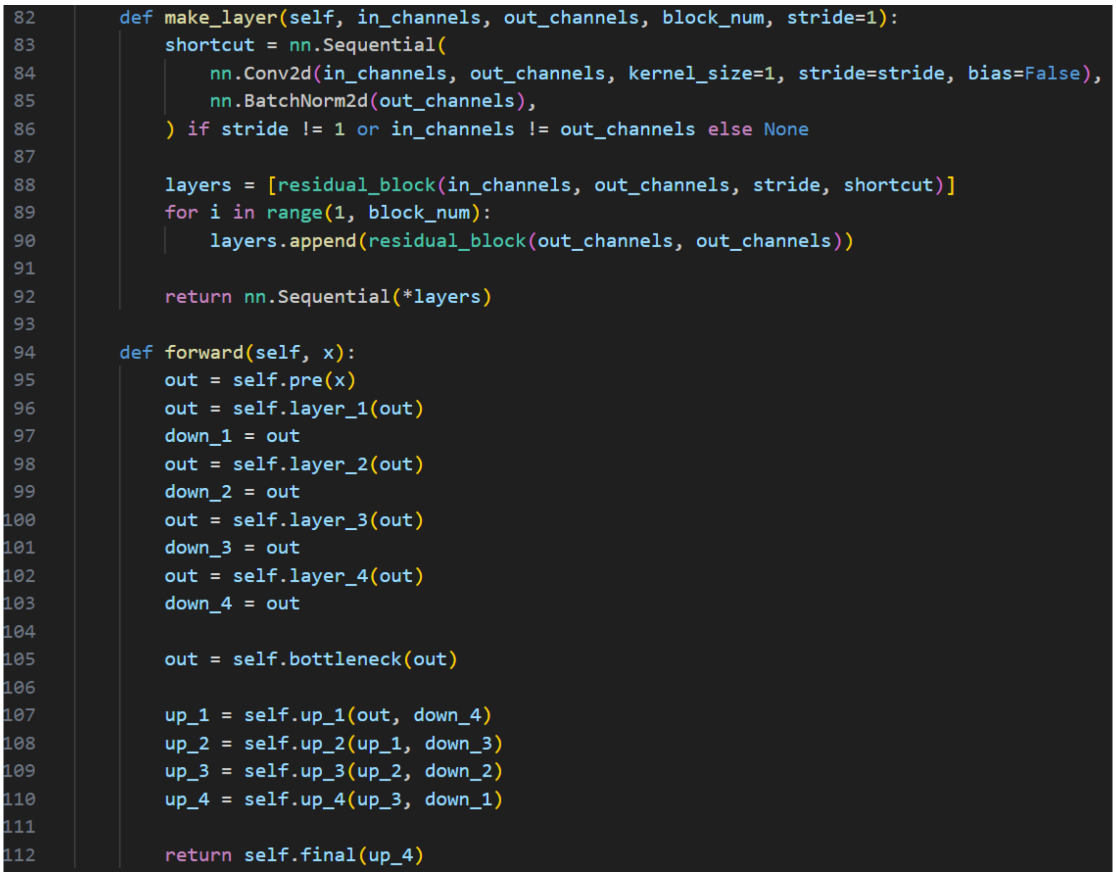
Graph 6 illustrates the constructor of resnet34_unet. The first-half is based on the structure of Resnet, while the rest is from Unet. In terms of the first-half part, we use make_layer() (as graph 7) to construct residual structure. In terms of the bottom layer, I use a residual block to construct it. And the rest of the parts is the same as Unet



```
class resnet34_unet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(resnet34_unet, self).__init__()

        self.pre = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

        self.layer_1 = self.make_layer(64, 64, 3)
        self.layer_2 = self.make_layer(64, 128, 4, stride=2)
        self.layer_3 = self.make_layer(128, 256, 6, stride=2)
        self.layer_4 = self.make_layer(256, 512, 3, stride=2)

        self.bottleneck = self.make_layer(512, 1024, 1, stride=2)

        self.up_1 = up_sample(1024, 512)
        self.up_2 = up_sample(512, 256)
        self.up_3 = up_sample(256, 128)
        self.up_4 = up_sample(128, 64)

        self.final = nn.Sequential(
            nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, out_channels, kernel_size=1),
            nn.Sigmoid()
        )
```

Graph 6. the constructor of resnet34_unet

In terms of forwarding, the network will save all the down blocks, and concatenate them will up blocks in the second-half of forwarding.



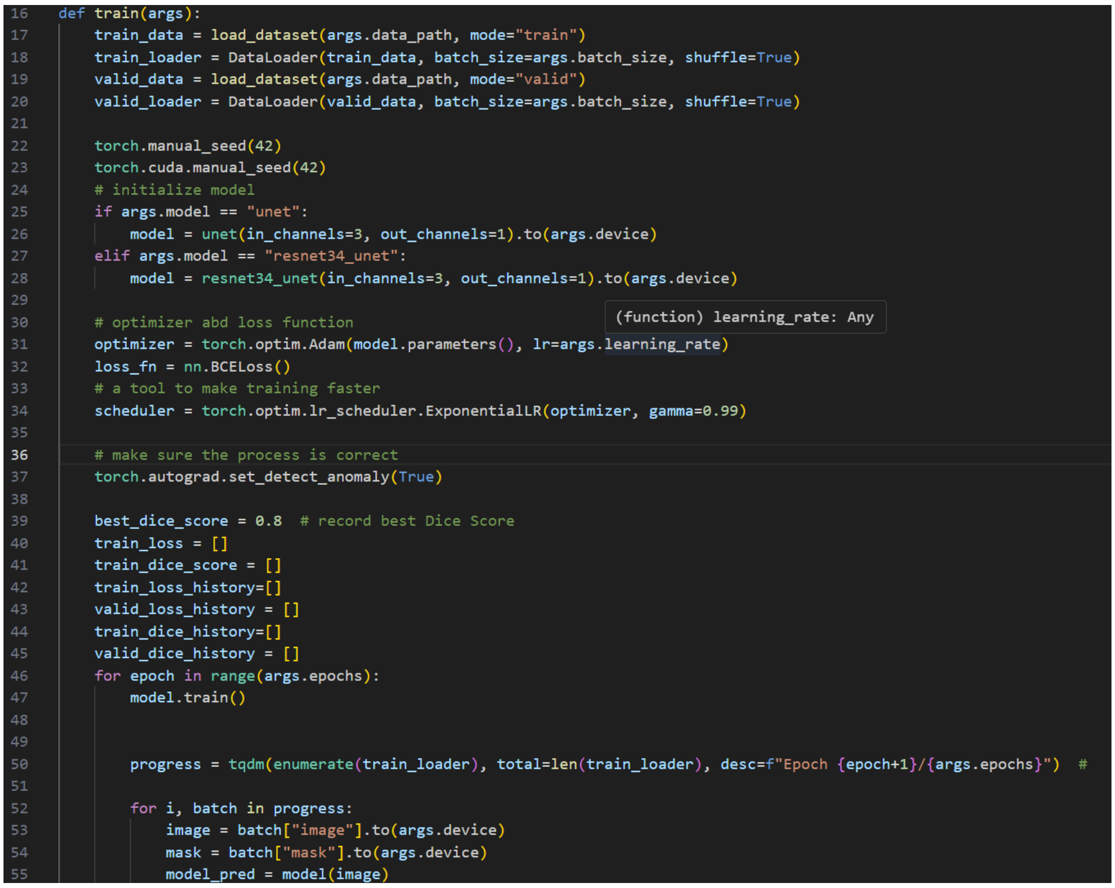
```python
    def make_layer(self, in_channels, out_channels, block_num, stride=1):
        shortcut = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(out_channels),
        ) if stride != 1 or in_channels != out_channels else None

        layers = [residual_block(in_channels, out_channels, stride, shortcut)]
        for i in range(1, block_num):
            layers.append(residual_block(out_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.pre(x)
        out = self.layer_1(out)
        down_1 = out
        out = self.layer_2(out)
        down_2 = out
        out = self.layer_3(out)
        down_3 = out
        out = self.layer_4(out)
        down_4 = out

        out = self.bottleneck(out)

        up_1 = self.up_1(out, down_4)
        up_2 = self.up_2(up_1, down_3)
        up_3 = self.up_3(up_2, down_2)
        up_4 = self.up_4(up_3, down_1)

        return self.final(up_4)
```

Graph 7. Other functions in resnet34_unet

1.2 training, inferencing, evaluate code

1.2.1 training

The procedure of training is similar to most of training in deep learning area, first we will initiate the a model corresponding to the input. Then we select optimizer (I use Adam in this lab) and loss function. The function used is BCE loss and dice loss because they can provide a better performance (website). Moreover, a scheduler (line 34) and set_detect_anomaly() [2] (line 37) are used to accelerate the training process and detect the error respectively. The progress (line 50) is a tool to record the process, telling the user the progress of training. Note that the random seed we set in this experiments is 42.

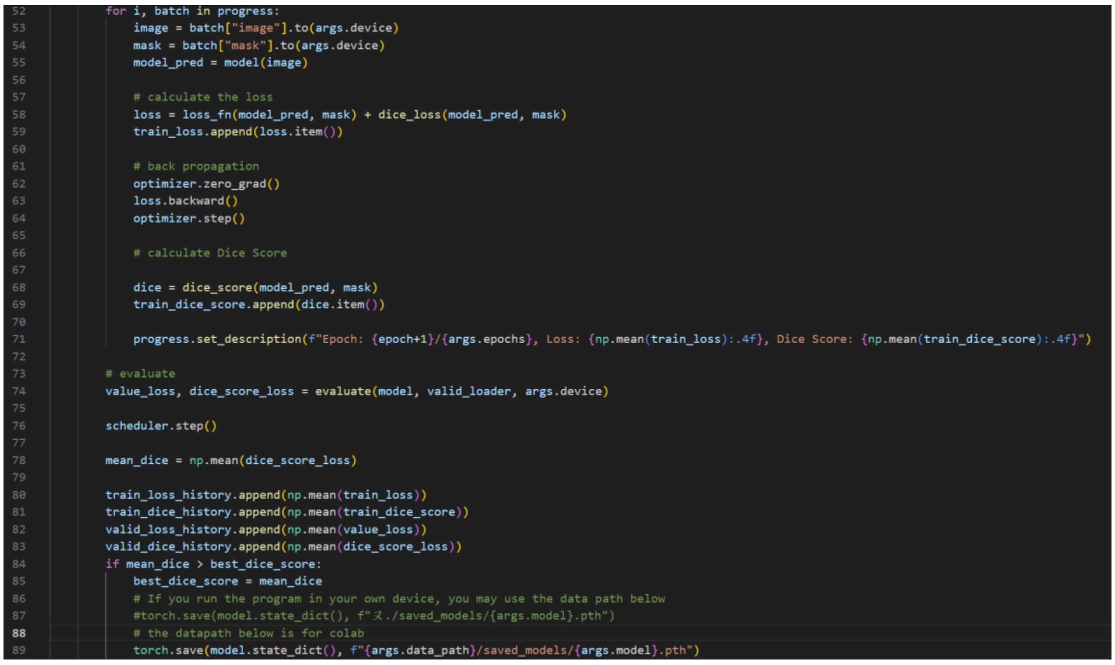


```python
def train(args):
    train_data = load_dataset(args.data_path, mode="train")
    train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True)
    valid_data = load_dataset(args.data_path, mode="valid")
    valid_loader = DataLoader(valid_data, batch_size=args.batch_size, shuffle=True)

    torch.manual_seed(42)
    torch.cuda.manual_seed(42)
    # initialize model
    if args.model == "unet":
        model = unet(in_channels=3, out_channels=1).to(args.device)
    elif args.model == "resnet34_unet":
        model = resnet34_unet(in_channels=3, out_channels=1).to(args.device)

    # optimizer abd loss function
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
    loss_fn = nn.BCELoss()
    # a tool to make training faster
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.99)

    # make sure the process is correct
    torch.autograd.set_detect_anomaly(True)

    best_dice_score = 0.8  # record best Dice Score
    train_loss = []
    train_dice_score = []
    train_loss_history=[]
    valid_loss_history = []
    train_dice_history=[]
    valid_dice_history = []
    for epoch in range(args.epochs):
        model.train()


        progress = tqdm(enumerate(train_loader), total=len(train_loader), desc=f"Epoch {epoch+1}/{args.epochs}")  #

        for i, batch in progress:
            image = batch["image"].to(args.device)
            mask = batch["mask"].to(args.device)
            model_pred = model(image)
```

Graph 8. Training (part 1)

According to graph 9, the training process will check the data of each batch, calculating the loss as well as dice score, doing backpropagation, and optimize the model (gradient descend). The function of set_description (line 71) is to print out what is going on during training. After each epoch, the loss and dice score about training data and valid data will be preserved in a list (line 80 ~ line 83). In the end of each epoch, if the dice score is better than previous one, the dice score and weight of the model will be saved (.pth file).

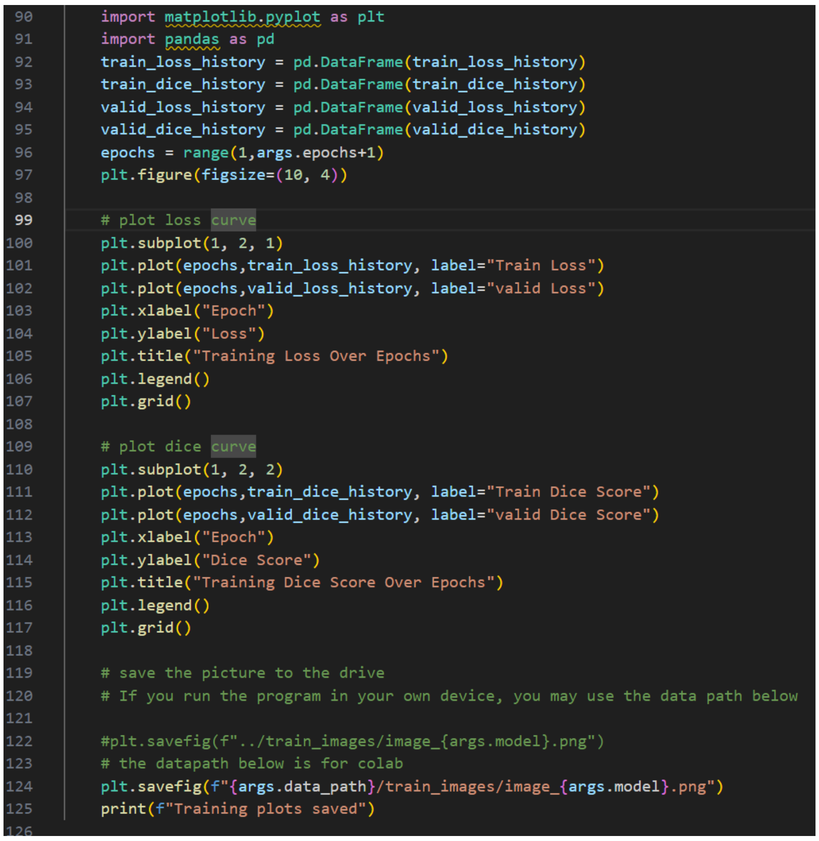
Note that the data path might be different since I am using google colab. Colab will delete the data in the directory if it is offline or interrupted for too long. To approach the issue, I decide to save .pth file in my google drive. (see line 86 ~ line 89 in graph 9)



```python
        for i, batch in progress:
            image = batch["image"].to(args.device)
            mask = batch["mask"].to(args.device)
            model_pred = model(image)

            # calculate the loss
            loss = loss_fn(model_pred, mask) + dice_loss(model_pred, mask)
            train_loss.append(loss.item())

            # back propagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # calculate Dice Score

            dice = dice_score(model_pred, mask)
            train_dice_score.append(dice.item())

            progress.set_description(f"Epoch: {epoch+1}/{args.epochs}, Loss: {np.mean(train_loss):.4f}, Dice Score: {np.mean(train_dice_score):.4f}")

        # evaluate
        value_loss, dice_score_loss = evaluate(model, valid_loader, args.device)

        scheduler.step()

        mean_dice = np.mean(dice_score_loss)

        train_loss_history.append(np.mean(train_loss))
        train_dice_history.append(np.mean(train_dice_score))
        valid_loss_history.append(np.mean(value_loss))
        valid_dice_history.append(np.mean(dice_score_loss))
        if mean_dice > best_dice_score:
            best_dice_score = mean_dice
            # If you run the program in your own device, you may use the data path below
            #torch.save(model.state_dict(), f"又./saved_models/{args.model}.pth")
            # the datapath below is for colab
            torch.save(model.state_dict(), f"{args.data_path}/saved_models/{args.model}.pth")
```

Graph 9. Training (part 2)

In order to visualize the process of training, the program in graph 10 is utilized to plot the relationship between training dataset and valid dataset. Similarly, the file will also be saved to my google drive (check line 119 ~ line 124 in graph 10).

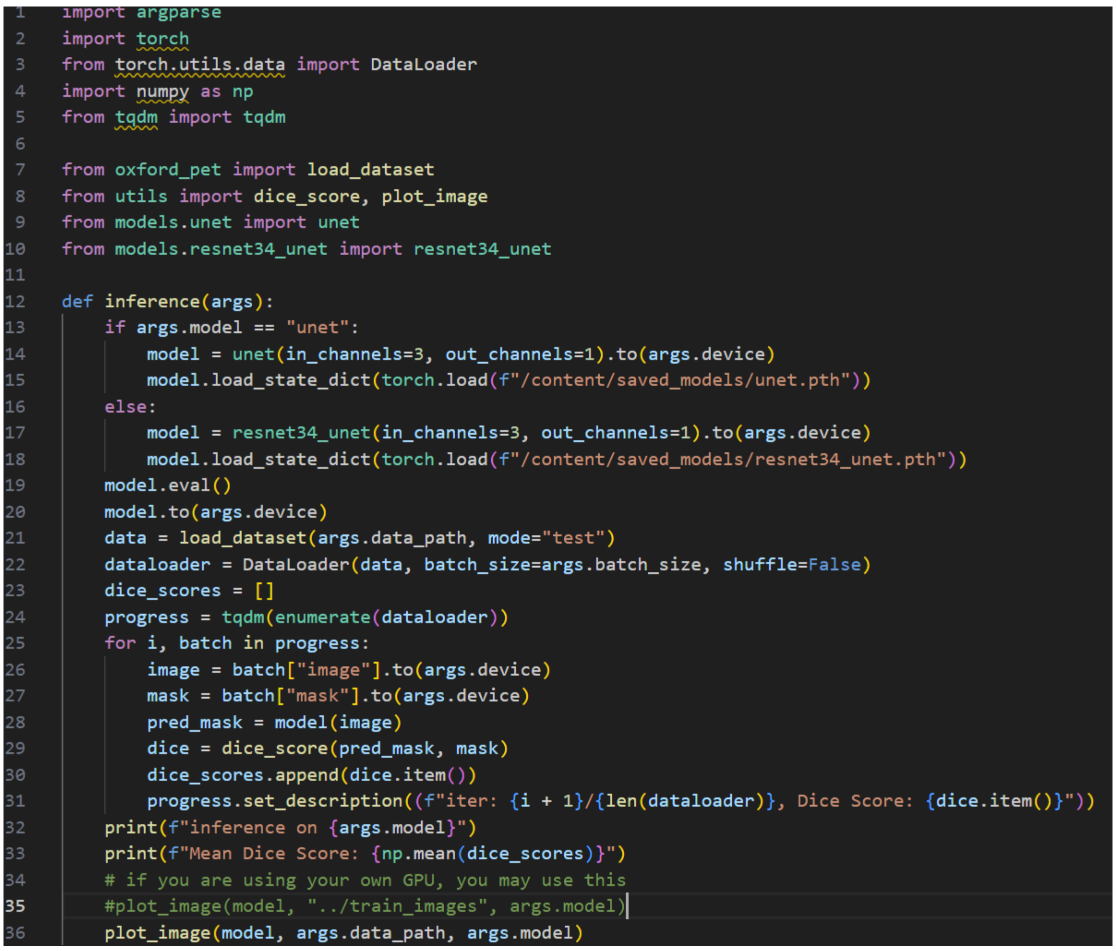


```python
90   import matplotlib.pyplot as plt
91   import pandas as pd
92   train_loss_history = pd.DataFrame(train_loss_history)
93   train_dice_history = pd.DataFrame(train_dice_history)
94   valid_loss_history = pd.DataFrame(valid_loss_history)
95   valid_dice_history = pd.DataFrame(valid_dice_history)
96   epochs = range(1,args.epochs+1)
97   plt.figure(figsize=(10, 4))
98
99   # plot loss curve
100  plt.subplot(1, 2, 1)
101  plt.plot(epochs,train_loss_history, label="Train Loss")
102  plt.plot(epochs,valid_loss_history, label="valid Loss")
103  plt.xlabel("Epoch")
104  plt.ylabel("Loss")
105  plt.title("Training Loss Over Epochs")
106  plt.legend()
107  plt.grid()
108
109  # plot dice curve
110  plt.subplot(1, 2, 2)
111  plt.plot(epochs,train_dice_history, label="Train Dice Score")
112  plt.plot(epochs,valid_dice_history, label="valid Dice Score")
113  plt.xlabel("Epoch")
114  plt.ylabel("Dice Score")
115  plt.title("Training Dice Score Over Epochs")
116  plt.legend()
117  plt.grid()
118
119  # save the picture to the drive
120  # If you run the program in your own device, you may use the data path below
121
122  #plt.savefig(f"../train_images/image_{args.model}.png")
123  # the datapath below is for colab
124  plt.savefig(f"{args.data_path}/train_images/image_{args.model}.png")
125  print(f"Training plots saved")
126
```

Graph 10. Training (part 3)

1.2.2 inferencing

The process of inferencing is similar to the training. The difference is that we no longer need optimizer and back propagation. After instantiating the model specified, the weights of the model will be loaded by load_state_dict() function. Then, the program will calculate the dice score after each batch. In the end, the picture will be generated and saved to my google drive.

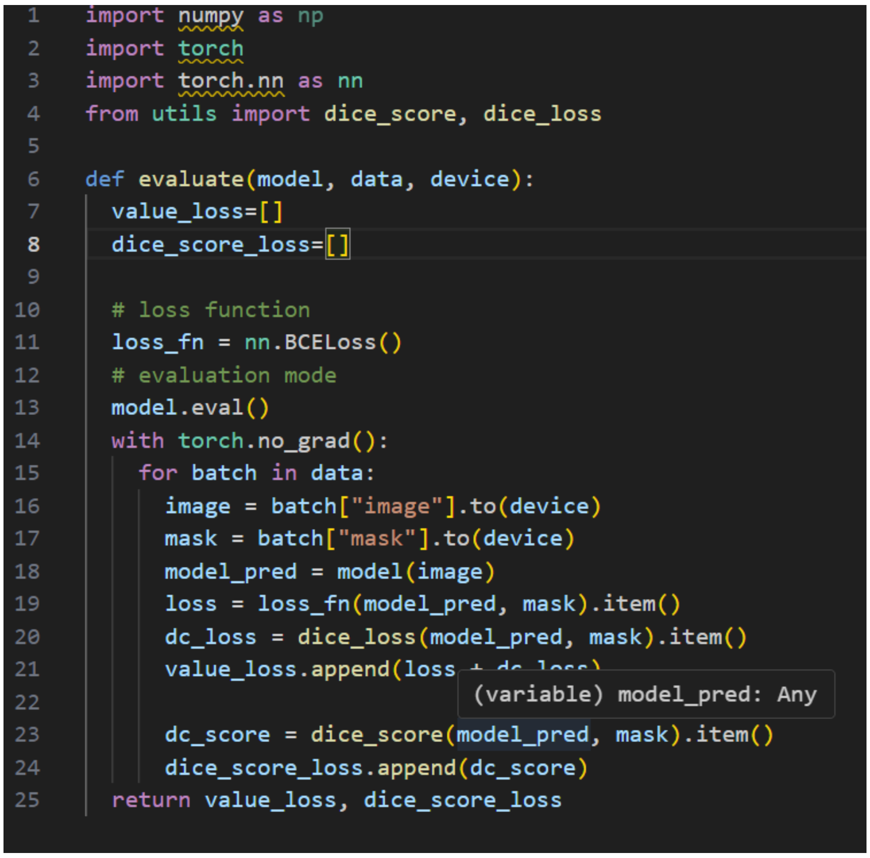


```python
import argparse
import torch
from torch.utils.data import DataLoader
import numpy as np
from tqdm import tqdm

from oxford_pet import load_dataset
from utils import dice_score, plot_image
from models.unet import unet
from models.resnet34_unet import resnet34_unet

def inference(args):
    if args.model == "unet":
        model = unet(in_channels=3, out_channels=1).to(args.device)
        model.load_state_dict(torch.load(f"/content/saved_models/unet.pth"))
    else:
        model = resnet34_unet(in_channels=3, out_channels=1).to(args.device)
        model.load_state_dict(torch.load(f"/content/saved_models/resnet34_unet.pth"))
    model.eval()
    model.to(args.device)
    data = load_dataset(args.data_path, mode="test")
    dataloader = DataLoader(data, batch_size=args.batch_size, shuffle=False)
    dice_scores = []
    progress = tqdm(enumerate(dataloader))
    for i, batch in progress:
        image = batch["image"].to(args.device)
        mask = batch["mask"].to(args.device)
        pred_mask = model(image)
        dice = dice_score(pred_mask, mask)
        dice_scores.append(dice.item())
        progress.set_description((f"iter: {i + 1}/{len(dataloader)}, Dice Score: {dice.item()}"))
    print(f"inference on {args.model}")
    print(f"Mean Dice Score: {np.mean(dice_scores)}")
    # if you are using your own GPU, you may use this
    #plot_image(model, "../train_images", args.model)
    plot_image(model, args.data_path, args.model)
```

Graph 11. Inferencing

1.2.3 evaluate

This program is used to calculate loss and dice score of valid data. Something worthy of paying attention is that I use torch.no_grad() [3] to reduce the number of calculation in the process. In other words, the model no longer need to trace too much variable during the process. Eventually, the function will return loss (in total) and dice score to the training process.



```python
import numpy as np
import torch
import torch.nn as nn
from utils import dice_score, dice_loss

def evaluate(model, data, device):
    value_loss=[]
    dice_score_loss=[]

    # loss function
    loss_fn = nn.BCELoss()
    # evaluation mode
    model.eval()
    with torch.no_grad():
        for batch in data:
            image = batch["image"].to(device)
            mask = batch["mask"].to(device)
            model_pred = model(image)
            loss = loss_fn(model_pred, mask).item()
            dc_loss = dice_loss(model_pred, mask).item()
            value_loss.append(loss + dc_loss)

            dc_score = dice_score(model_pred, mask).item()
            dice_score_loss.append(dc_score)
    return value_loss, dice_score_loss
```

Graph 12. evaluating

2. Data Preprocessing

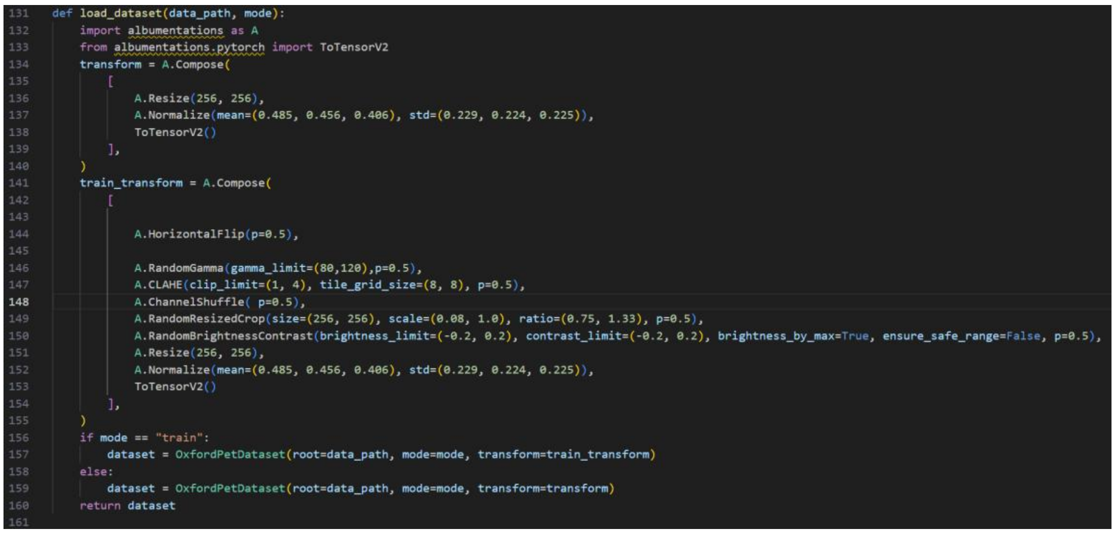
2.1 Preprocessing method

In terms of the data preprocessing, I choose the package called "albumentation" [4]. The reason is that this package provides a variety of transformation for the user to utilize. Moreover, albumentation provides a clear visualization in their website, which allows users to have a better understanding on the type of transformation they use.

Here are the steps of my transformation:

- Horizontal Flip:
  Flip the image horizontally with specific probability
- RandomGamma:
  Applies random gamma correction to the input image
- CLAHE:
  enhances the contrast of the input image
- ChannelShuffle:
  Shuffle the channel (RGB) of images

- RandomResizedCrop:
  crops a random portion of the input image and resizes the crop to a specified size
- RandomBrightnessContrast:
- Resize:
  resize to 256*256
- Normalization:
  apply normalization based on the forum [5]
- ToTensor
  Convert the image to tensor type



```python
def load_dataset(data_path, mode):
    import albumentations as A
    from albumentations.pytorch import ToTensorV2
    transform = A.Compose(
        [
            A.Resize(256, 256),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2()
        ],
    )
    train_transform = A.Compose(
        [

            A.HorizontalFlip(p=0.5),

            A.RandomGamma(gamma_limit=(80,120),p=0.5),
            A.CLAHE(clip_limit=(1, 4), tile_grid_size=(8, 8), p=0.5),
            A.ChannelShuffle( p=0.5),
            A.RandomResizedCrop(size=(256, 256), scale=(0.08, 1.0), ratio=(0.75, 1.33), p=0.5),
            A.RandomBrightnessContrast(brightness_limit=(-0.2, 0.2), contrast_limit=(-0.2, 0.2), brightness_by_max=True, ensure_safe_range=False, p=0.5),
            A.Resize(256, 256),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2()
        ],
    )
    if mode == "train":
        dataset = OxfordPetDataset(root=data_path, mode=mode, transform=train_transform)
    else:
        dataset = OxfordPetDataset(root=data_path, mode=mode, transform=transform)
    return dataset

```

Graph 13. Data preprocessing

2.2 Why my dataset is unique

I perceive my transformation(dataset) is unique since I choose the transformation to strengthen the difference between the object and background (brightness or contrast) and shuffle the channel of data randomly, hoping to improve the variety in the dataset. What's more, RandomResizeCrop is a famous type of transformation in practice [6]. I am convinced that it can improve my models.

3. Analyze the experiment results

3.1 hyperparameter setting

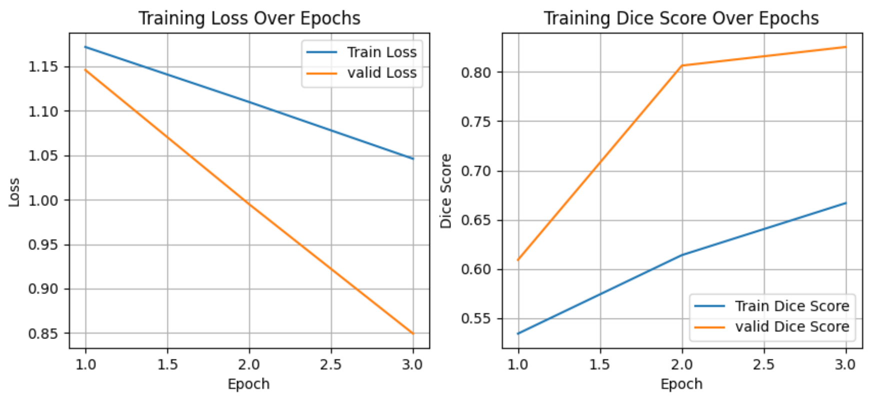
The hyperparameter settings are provided below:

- Batch size : 32
- Epochs : 200
- Learning rate : 1e-5

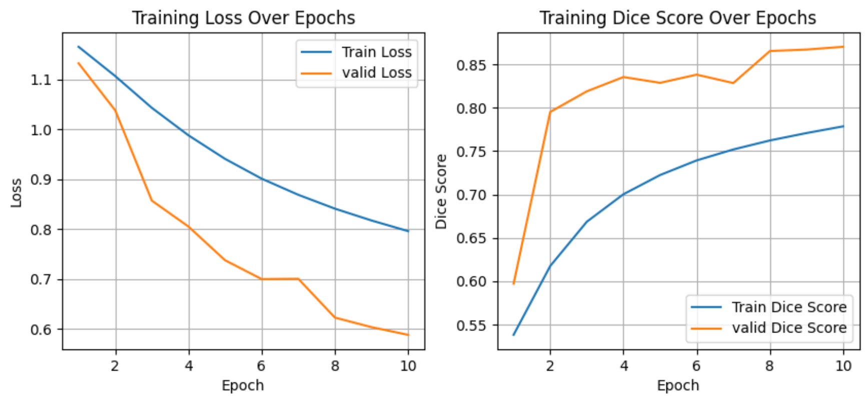
- Loss function : BCE loss + dice loss [7]
- Optimizer : Adam

3.2 the experiment on epoch

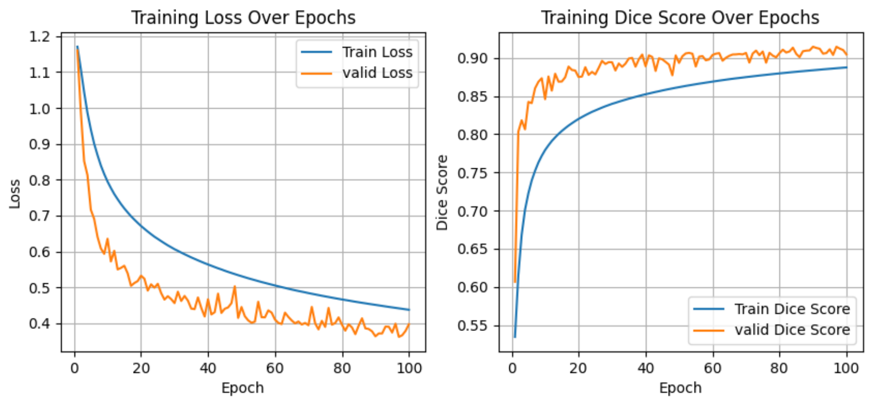
One thing interesting to me is that what is the origin of my models, and how the number of epochs affect them. The pictures below illustrate the difference after 3, 10, and 100 epochs (take Resnet34_Unet as example, other settings are constant):



Graph 14. 3 epochs



Graph 15. 10 epochs

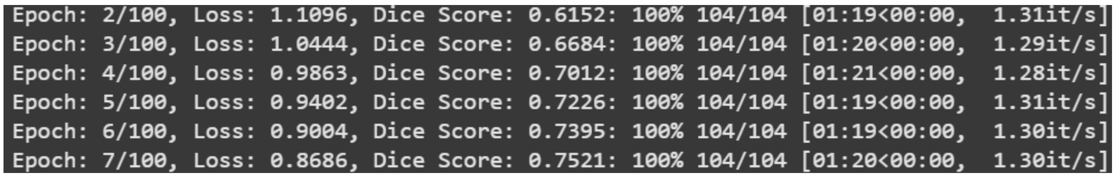


Graph 16. 100 epochs

We can perceive that the loss and dice score will decrease dramatically at first and keep stable after a while. And the larger number of epochs usually comes with better performance.

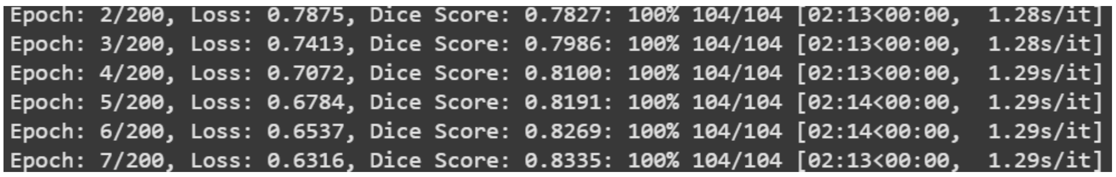
3.3 Unet V.S. Resnet34_Unet

According to the experiments, the biggest different between the two models are the training time. From graph 17 and graph 18 we can perceive that the training time of Resnet34_Unet each epoch is roughly 1.5 minutes, while the one of Unet is over 2 minutes. Once the number of epochs is quite large, the disparity of training time between two models will increase.



```
Epoch: 2/100, Loss: 1.1096, Dice Score: 0.6152: 100% 104/104 [01:19<00:00,  1.31it/s]
Epoch: 3/100, Loss: 1.0444, Dice Score: 0.6684: 100% 104/104 [01:20<00:00,  1.29it/s]
Epoch: 4/100, Loss: 0.9863, Dice Score: 0.7012: 100% 104/104 [01:21<00:00,  1.28it/s]
Epoch: 5/100, Loss: 0.9402, Dice Score: 0.7226: 100% 104/104 [01:19<00:00,  1.31it/s]
Epoch: 6/100, Loss: 0.9004, Dice Score: 0.7395: 100% 104/104 [01:19<00:00,  1.30it/s]
Epoch: 7/100, Loss: 0.8686, Dice Score: 0.7521: 100% 104/104 [01:20<00:00,  1.30it/s]
```

Graph 17. Time of Resnet34_Unet (training)



```
Epoch: 2/200, Loss: 0.7875, Dice Score: 0.7827: 100% 104/104 [02:13<00:00,  1.28s/it]
Epoch: 3/200, Loss: 0.7413, Dice Score: 0.7986: 100% 104/104 [02:13<00:00,  1.28s/it]
Epoch: 4/200, Loss: 0.7072, Dice Score: 0.8100: 100% 104/104 [02:13<00:00,  1.29s/it]
Epoch: 5/200, Loss: 0.6784, Dice Score: 0.8191: 100% 104/104 [02:14<00:00,  1.29s/it]
Epoch: 6/200, Loss: 0.6537, Dice Score: 0.8269: 100% 104/104 [02:14<00:00,  1.29s/it]
Epoch: 7/200, Loss: 0.6316, Dice Score: 0.8335: 100% 104/104 [02:13<00:00,  1.29s/it]
```

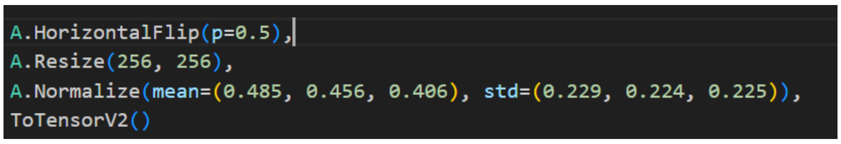
Graph 18. Time of Unet (training)

3.4 The experiment on preprocessing

In order to find out how the preprocess affect the performances of our models, I design three types of transformation (data preprocessing) during the experiments. The following will provide more details, and results included. Note that the result is based on resnet34_unet (100 epochs)

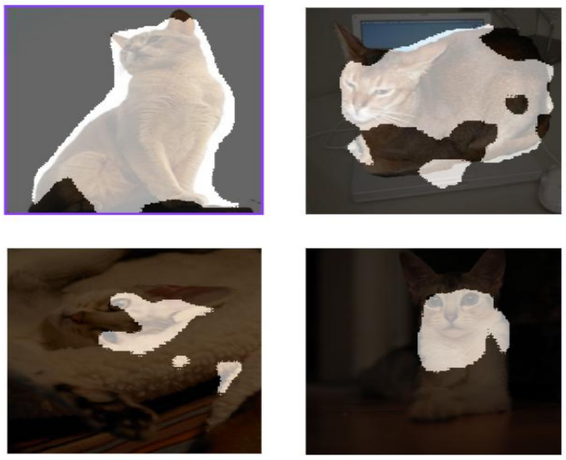
(1) Type 1:

The transformation with only horizontal flip (probabilities = 0.5), resizing and Normalization.

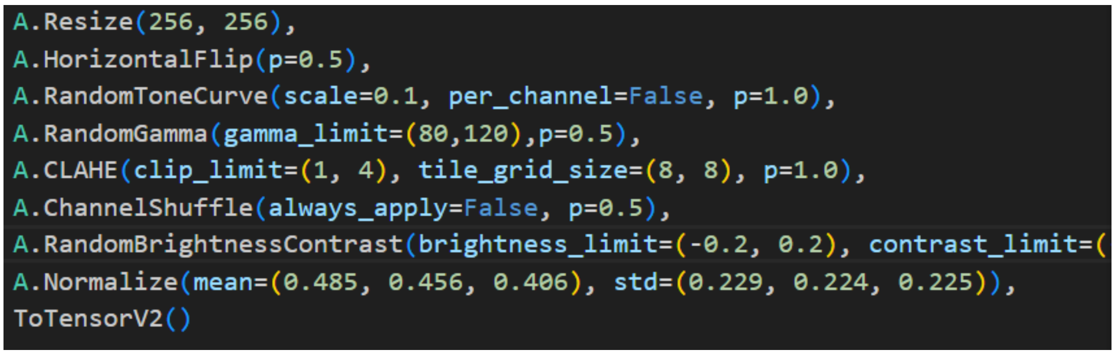


```
A.HorizontalFlip(p=0.5),
A.Resize(256, 256),
A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
ToTensorV2()
```

Graph 19. Transformation type 1

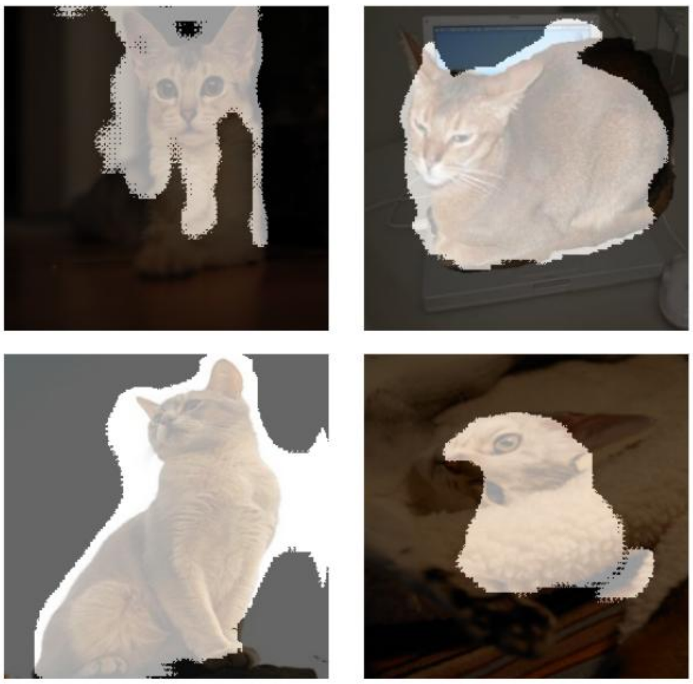

Graph 20. Result of type 1

(2) Type 2:
The transform with a series of transformation. changing the relationship between bright and dark (RandomToneCurve), and level of gamma (RandomGamma), applying Contrast Limited Adaptive Histogram Equalization (CLAHE) etc.

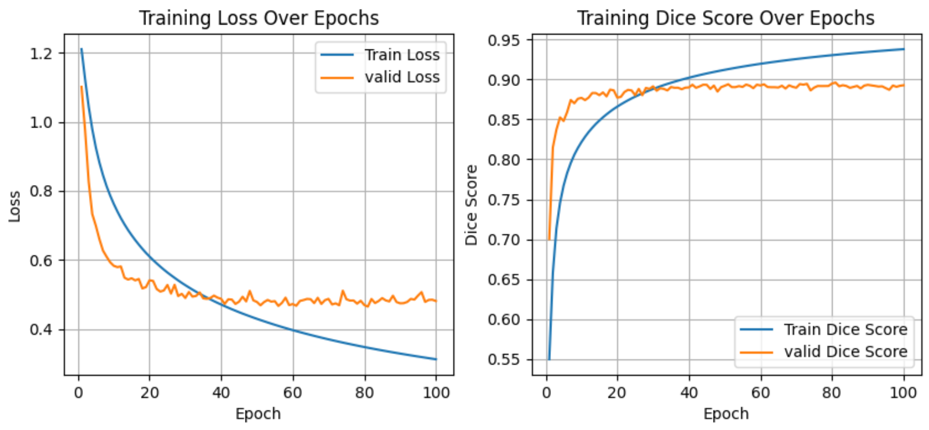


```
A.Resize(256, 256),
A.HorizontalFlip(p=0.5),
A.RandomToneCurve(scale=0.1, per_channel=False, p=1.0),
A.RandomGamma(gamma_limit=(80,120),p=0.5),
A.CLAHE(clip_limit=(1, 4), tile_grid_size=(8, 8), p=1.0),
A.ChannelShuffle(always_apply=False, p=0.5),
A.RandomBrightnessContrast(brightness_limit=(-0.2, 0.2), contrast_limit=(
A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
ToTensorV2()
```

Graph 21. Transformation type 2

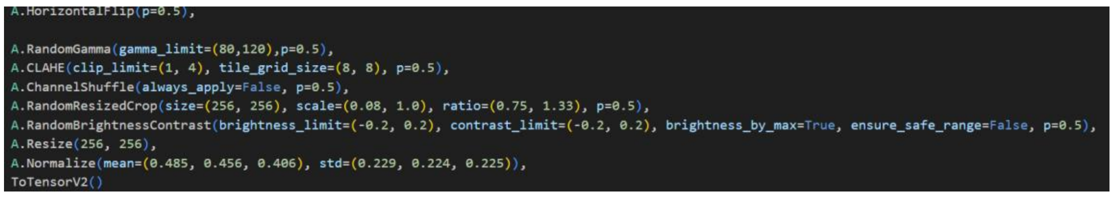

Graph 22. Result of type 2

The flaw of this type of transformation is that it may lead to overfitting. According to graph 23, the valid loss and valid dice score can not match the trend of training curve very well, which is a sign of overfitting. [8]



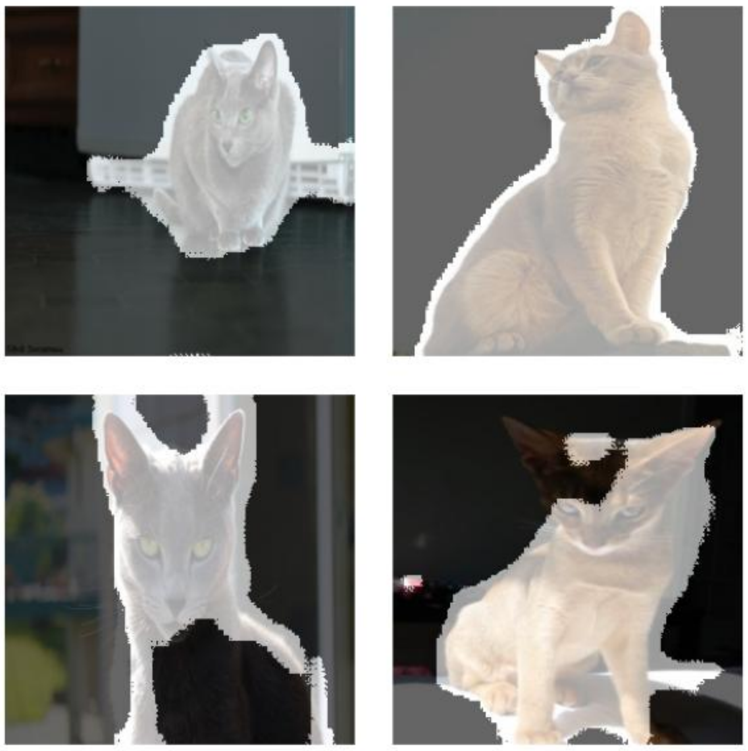
Graph 23. Curve of overfitting

(3) Type 3:
The transformation is very similar to type 2, but the image will firstly experience horizontal flip. Furthermore, I also applied RandomSizeCrop, which will crop a certain area and resize to 256*256. In the end, the picture will be resized to 256*256, experienced normalization.



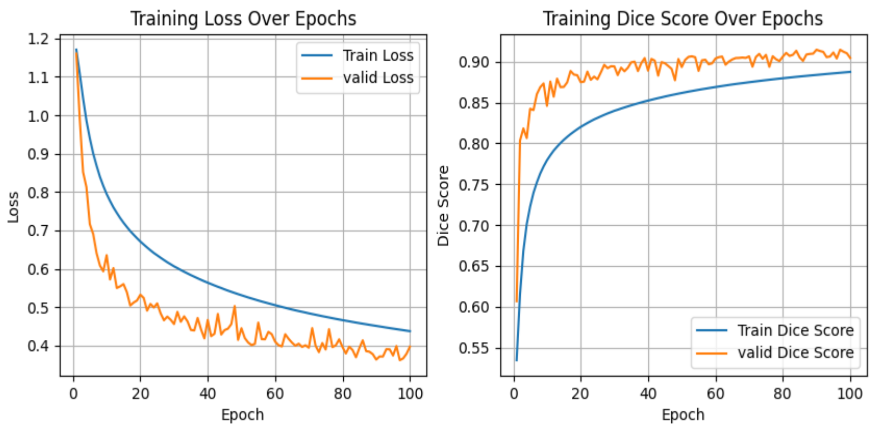
```
A.HorizontalFlip(p=0.5),

A.RandomGamma(gamma_limit=(80,120),p=0.5),
A.CLAHE(clip_limit=(1, 4), tile_grid_size=(8, 8), p=0.5),
A.ChannelShuffle(always_apply=False, p=0.5),
A.RandomResizedCrop(size=(256, 256), scale=(0.08, 1.0), ratio=(0.75, 1.33), p=0.5),
A.RandomBrightnessContrast(brightness_limit=(-0.2, 0.2), contrast_limit=(-0.2, 0.2), brightness_by_max=True, ensure_safe_range=False, p=0.5),
A.Resize(256, 256),
A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
ToTensorV2()
```

Graph 24. Transformation type 3



Graph 25. Result of type 4

According to graph 25, 22, and 20, we can conclude that type 3 is the best transformation among the experiments. Moreover, the issue about overfitting is fixed now (see graph 26)

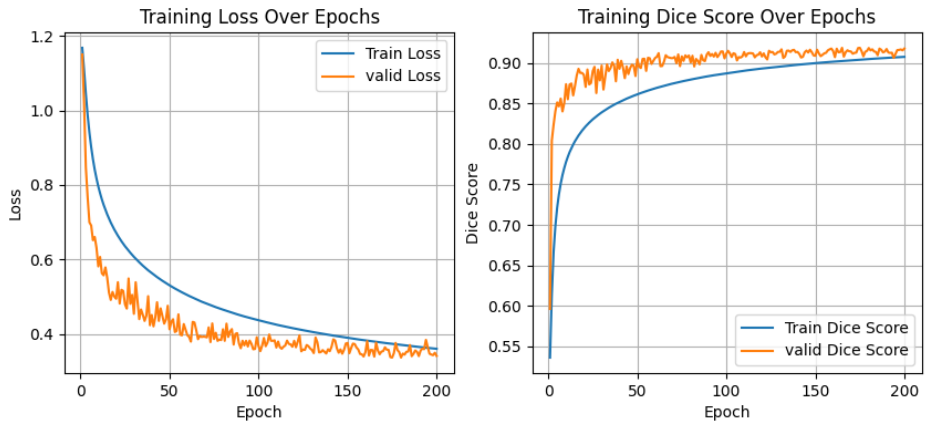


Graph 26. Training curve (without overfitting)

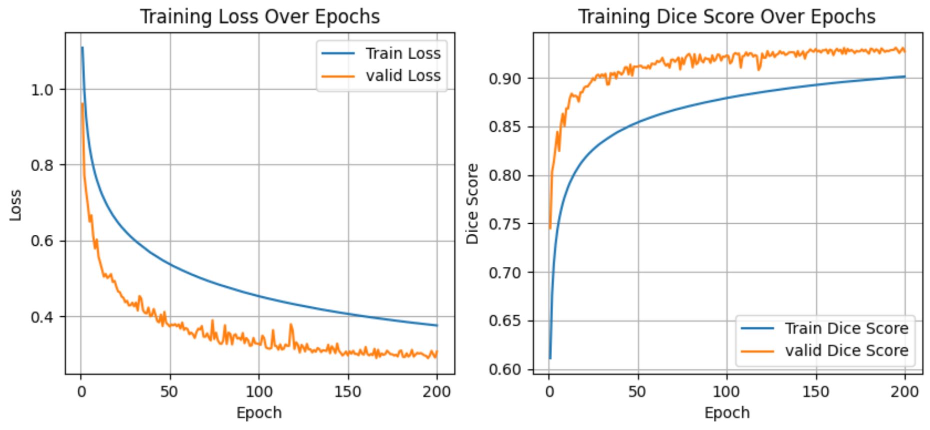
3.5 The final result

After realizing the effect of epoch number and finding a better transformation, we can finally get the setting to make the best performance during the experiments. So, we set the number of epochs to 200 and applying type 3 data preprocessing, and the result is provided below.

(1) The curve of loss and dice score during training:

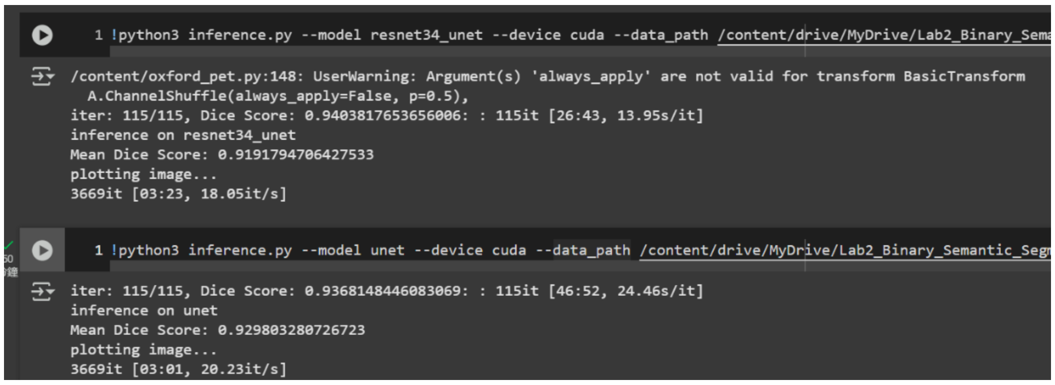


Graph 27. The curve of Resnet34_Unet



Graph 28. The curve of Unet

(2) The average dice scores of the models

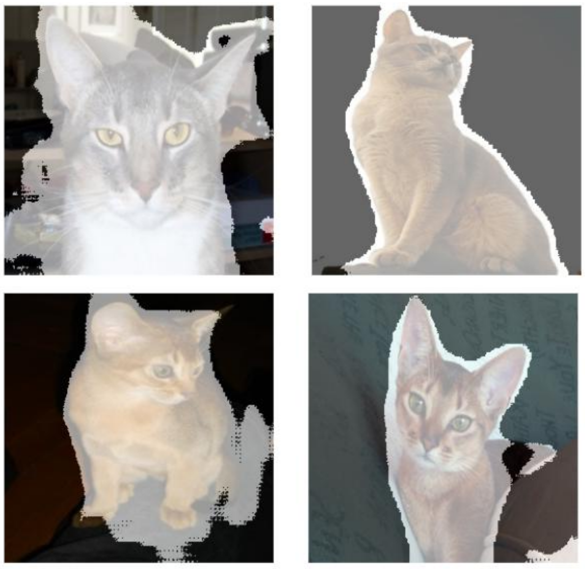


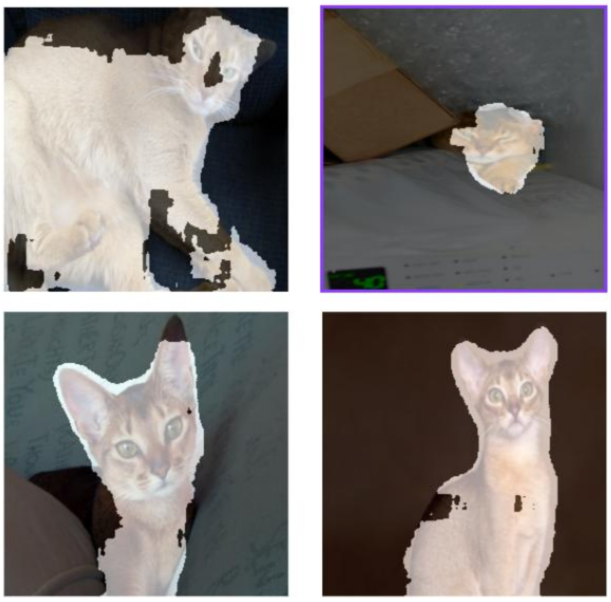
```
!python3 inference.py --model resnet34_unet --device cuda --data_path /content/drive/MyDrive/Lab2_Binary_Sema
/content/oxford_pet.py:148: UserWarning: Argument(s) 'always_apply' are not valid for transform BasicTransform
  A.ChannelShuffle(always_apply=False, p=0.5),
iter: 115/115, Dice Score: 0.9403817653656006: : 115it [26:43, 13.95s/it]
inference on resnet34_unet
Mean Dice Score: 0.9191794706427533
plotting image...
3669it [03:23, 18.05it/s]
```

```
!python3 inference.py --model unet --device cuda --data_path /content/drive/MyDrive/Lab2_Binary_Semantic_Segm
iter: 115/115, Dice Score: 0.9368148446083069: : 115it [46:52, 24.46s/it]
inference on unet
Mean Dice Score: 0.929803280726723
plotting image...
3669it [03:01, 20.23it/s]
```

Graph 29. The best average dice scores of two models

(3) The result of the models


Graph 30. The output images from Resnet34_Unet

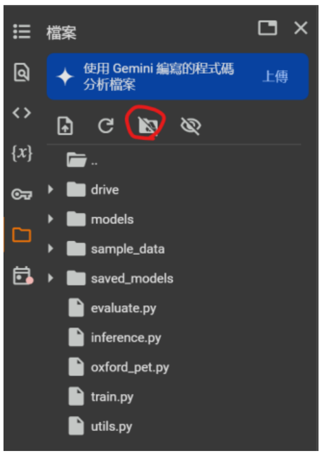

Graph 31. The output images from Unet

4. Execution steps

Because the GPU in my laptop is not powerful enough to train a model, I decided to train and inference my models on google collab, which provide three types of powerful GPU and the capability to link with google drive. The step to execute on collab is provided below.

4.1 Training step:

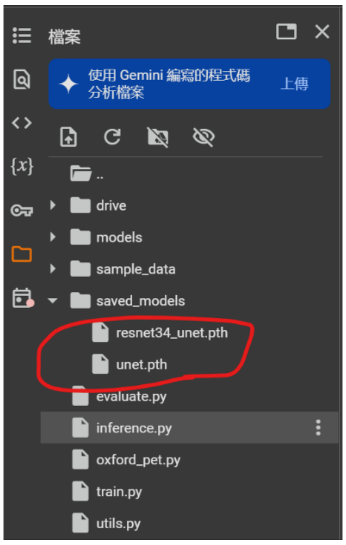
- Upload the source code to the directory of google collab
- Link the directory with google drive



- Create a directory called "saved_models"
  reasons: the root directory is very complicated that it's hard to find the right directory (/content/) when using google collab. Furthermore, google collab would delete your file in the directory once disconnecting to the server. To approach the issues, I decide to save all of my results in google drive in order to conserve them permanently.
- Input the command :
  " !python3 train.py --model resnet34_unet(or unet) --device cuda --data_path /content/drive/MyDrive/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet --epochs 100 --batch_size 32 --learning_rate 1e-5 "
  you can also run the command(python3…) in the built-in terminal of collab, but I think executing the command in the Jupyter notebook of collab is easier to monitor.
- check the result, my code will save the .pth file as well as the picture about training in google drive. If you run the code with local GPU (vscode for example) you may change the related path (check the comment in line 86 ~ line 89 of train.py)
  Note that I add a directory called "trained_image" in my zip file in order to store the information about training.

4.2 Inferencing step:

- upload the source code to google collab
- Upload the .pth file (trained model) to google collab and link to google drive



- Use **A100 GPU** to inference according to the experience, other GPU (L4 or T4) might run out of memory during the process. Note that one disadvantage is that the usage of computational unit might increase.
- input command
  "!python3 inference.py --model resnet34_unet(or unet) --device cuda --data_path /content/drive/MyDrive/Lab2_Binary_Semantic_Segmentation_2025/dataset/oxford-iiit-pet --batch_size 32"
- Check the result the result (image) will be saved to google drive and the dice score can be checked on the monitor Same as training step, you might change the path to fulfill your local environment if needed. (check the command in line 34 ~ line 36 of inference.py)
  Note that the output images will be saved with dataset
  (check line 34 in utils.py)

5. Discussion

5.1 alternative architecture

5.1.1 deeper layer

To improve the performances of our model by revising the architecture. Maybe we can provide deeper layer (such as resent50) to make the model better. However, another issue may occur because the deeper structure might not preserve the feature of an object perfectly according to some researches. [1]

5.1.2 CBAM blocks:

We may apply CBAM blocks to the model. According to the research, the blocks can enhance features representation. [9]

5.2 potential research direction

5.2.1 identify good transformation for models

We can not deny that there are various transformations nowadays. One of the issues we can further discuss is that which kind of transformation is the best in this task. In this case, we can train our model more efficiently and reduce the resources it takes. Moreover, data augmentation is also worth considering, so that models can check various image in a limited training dataset.

5.2.2 the methods to stabilize the models

During the experiments, I found that the performance is good to over half of the test data, while it sometimes is bad on certain data. I think it is a good issue we can elaborate on.

6. References:

[1]: https://meetonfriday.com/posts/fb19d450/
[2]: https://blog.csdn.net/q7w8e9r4/article/details/134620547
[3]: https://pytorch.org/docs/stable/generated/torch.no_grad.html
[4]: https://albumentations.ai/docs/
[5]: https://stackoverflow.com/questions/58151507/why-pytorch-officially-use-mean-0-485-0-456-0-406-and-std-0-229-0-224-0-2
[6]: https://github.com/yaoyao-liu/meta-transfer-learning/issues/10
[7]: https://arxiv.org/pdf/2110.08322
[8]: https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting/tutorial
[9]: https://arxiv.org/abs/1807.06521
[10]:
https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbkU5RGRWRklFOE9hRkw4bm1OQlZfSzNwNjNVZ3xBQ3Jtc0trUC16eExSY0FzbHdCS0s2OUhNN3ZlZnFic0V5Nkp2WW93QWtuRjd2N0pFLXRPODBIb3ppRTFqQmQwalc4emRfNTVRa2hhODZTZElNY3Nqc2xGeW96M3BkYVBI

Q2gtbGV2ZHBWZkp2QURORzdNQVppcw&q=https%3A%2F%2Flearnpytorch
.io%2F&v=V_xro1bcAuA
[11]: https://www.learnpytorch.io/