NYCU DL Lab3

Binary Semantic Segmentation

313552049 鄭博涵

1.Overview of your lab 3 (10%)

在這次 lab 中,實作了 UNet, ResNet 模型, 並用他們來辨別動物圖片, 我們使用了一個包含多種動物圖像的 dataset, 並對每個圖像進行 preprocessing, 如圖像縮放和標準化, 最終藉由 Dice score 高低來辨別 model 的好壞

2.Implementation Details (30%)

A.Details of your training, evaluating, inferencing code

Training code:

```python
def train(args):  # 從 get_args() 函數中獲得的命令行參數 有data_path, epochs, batch_size, learning_rate
    device = torch.device("cuda" if torch.cuda.is_available() else 'cpu')
    model = ResNet_UNet(2).to(device)
    # model = UNet(n_channels=3, n_classes=2).to(device)

    # set loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    # optimizer = optim.SGD(model.parameters(), lr=args.learning_rate, momentum=0.88)
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    train_loader = load_dataset(args.data_path, mode='train', batch_size=args.batch_size, shuffle = 'True')
    val_loader = load_dataset(args.data_path, mode='valid', batch_size=args.batch_size, shuffle = 'False')

    best_dice_score = 0.0   # 初始化最佳dice_score
    train_losses = []   # 用於儲存每個epoch的train_loss
    val_losses = []
    val_dice_scores = []

    model.train()   # set to train mode
    for epoch in range(args.epochs):
        running_loss = 0.0
        total_samples = 0
        for i, data in enumerate(train_loader):
            images = data['image'].to(device, dtype=torch.float32)  # data['image']: 包含了批次中的所有image 數據
            # if images.dim() == 3:  # 如果圖像是3維的 (channels, height, width)
            #     images = images.unsqueeze(0)  # 增加批次維度，使其變為 (1, channels, height, width)
            masks = data['mask'].to(device, dtype=torch.float32)    # data['mask'] 包含了批次中對應的mask 數據

            if i % 50 == 0:
                print(f"Current Batch Number: {i+1}")   # 顯示 (batch_size, height, width)

            optimizer.zero_grad()   # 梯度重置

            outputs = model(images) # 輸入是images
            masks = masks.squeeze(1).long()  # 去掉第2維度，即將形狀從 (N, 1, H, W) 變為 (N, H, W)
            loss = criterion(outputs, masks)    # masks 為Ground Truth

            loss.backward()
            optimizer.step()   # 更新模型的參數
```

```python
            batch_size = images.size(0)
            running_loss += loss.item() * batch_size    # .item(): 將tensor改成float (loss 是tensor)
            total_samples += batch_size

        avg_epoch_loss = running_loss / total_samples
        train_losses.append(avg_epoch_loss)

        # 在每個 epoch 結束後進行驗證
        val_loss, avg_dice_score = validate(model, val_loader, criterion, device)
        val_losses.append(val_loss)
        val_dice_scores.append(avg_dice_score)

        print(f"Epoch: {epoch+1}, Train Loss: {avg_epoch_loss}, Validation Loss: {val_loss:.4f}, Dice Score: {avg_dice_score:.4f}")

        # 保存最好的dice_score model
        if avg_dice_score > best_dice_score:
            best_dice_score = avg_dice_score
            # torch.save(model.state_dict(), 'saved_models/ResNet_UNet_best_model.pth')
            torch.save(model.state_dict(), 'saved_models/UNet_best_model.pth')
            print(f"Model saved with Dice Score: {best_dice_score:.4f}")

    # 繪製損失曲線
    epochs = range(1, len(train_losses) + 1)
    plt.figure()
    plt.plot(epochs, train_losses, label='Train Loss')
    plt.plot(epochs, val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train and Validation Loss over Epochs')
    plt.legend()
    # plt.savefig('ResNet_loss_curve.png')
    plt.savefig('UNet_loss_curve.png')
    plt.show()
```

Evaluating code:

```python

# test階段
# Evaluation
import argparse
import torch
from utils import dice_score
from models.unet import UNet
from models.resnet34_unet import ResNet_UNet
from oxford_pet import load_dataset

def evaluate(net, data_loader, device):
    net.eval()
    total_loss = 0
    total_dice_score = 0
    num_batches = len(data_loader)

    criterion = torch.nn.CrossEntropyLoss()

    with torch.no_grad():
        for data in data_loader:
            images = data['image'].to(device, dtype=torch.float32)
            masks = data['mask'].to(device, dtype=torch.float32)

            # 將masks從[batch_size, 1, height, width] 變為[batch_size, height, width]
            masks = masks.squeeze(1).long()  # 確保masks是LongTensor

            outputs = net(images)
            loss = criterion(outputs, masks)
            total_loss += loss.item()

            dice = dice_score(outputs, masks)
            total_dice_score += dice

    avg_loss = total_loss / num_batches
    avg_dice_score = total_dice_score / num_batches

    return avg_loss, avg_dice_score
```

```python
def get_args():
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')
    parser.add_argument('--data_path', type=str, default='./dataset', help='path of the input data')
    parser.add_argument('--epochs', '-e', type=int, default=10, help='number of epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=16, help='batch size')
    parser.add_argument('--learning_rate', '-lr', type=float, default=0.00012, help='learning rate')

    return parser.parse_args()

if __name__ == "__main__":
    args = get_args()  # 呼叫get_args()以獲取命令行參數

    device = torch.device("cuda" if torch.cuda.is_available() else 'cpu')
    test_loader = load_dataset(args.data_path, mode='test', batch_size=args.batch_size, shuffle=False)  # 修正mode和shuffle

    # 初始化模型結構

    # model = UNet(n_channels=3, n_classes=2).to(device)
    model = ResNet_UNet(num_classes=2).to(device)

    # 載入訓練好的權重
    # model_weights = torch.load('saved_models/ResNet_UNet_best_model_90.pth', weights_only=True)
    # model_weights = torch.load('saved_models/UNet_best_model_90.pth', weights_only=True)

    # model_weights = torch.load('saved_models/DL_Lab3_UNet_313552049_鄭博涵.pth', weights_only=True)
    model_weights = torch.load('saved_models/DL_Lab3_ResNet34_UNet_313552049_鄭博涵.pth', weights_only=True)

    model.load_state_dict(model_weights)

    avg_loss, avg_dice_score = evaluate(model, test_loader, device)

    print(f"Evaluation Loss: {avg_loss}, Evaluation Dice score: {avg_dice_score}")
```

Inferencing code:

```python
import numpy as np
import argparse
import torch
from torchvision import transforms
from PIL import Image
import os
from models.unet import UNet
from models.resnet34_unet import ResNet_UNet
from oxford_pet import load_dataset

def get_args():
    parser = argparse.ArgumentParser(description='Model inference on new images')
    # parser.add_argument('--model', type=str, default='saved_models/UNet_best_model_90.pth', help='Path to the trained model file')
    # parser.add_argument('--model', type=str, default='saved_models/ResNet_UNet_best_model_90.pth', help='Path to the trained model file')

    parser.add_argument('--model', type=str, default='saved_models/DL_Lab3_UNet_313552049_鄭博涵.pth', help='Path to the trained model file')
    # parser.add_argument('--model', type=str, default='saved_models/DL_Lab3_ResNet34_UNet_313552049_鄭博涵.pth', help='Path to the trained mode

    parser.add_argument('--data_path', type=str, default='./dataset', help='Path to the input data folder')
    parser.add_argument('--output_path', type=str, default='.', help='Path to save the output results')   # 預設存在當前目錄
    parser.add_argument('--batch_size', '-b', type=int, default=10, help='Batch size for inference')
    parser.add_argument('--device', default='cuda', help='Device to use for inference (cuda or cpu)')
    return parser.parse_args()

def preprocess_image(image_path):
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
    ])
    image = Image.open(image_path)
    return transform(image).unsqueeze(0)  # Add batch dimension

def generate_output_image(preds, image_size):
    output_image = np.zeros((image_size[1], image_size[0], 3), dtype=np.uint8)

    # 假設preds是0或1的二元掩膜，0代表背景，1代表前景
    # 可以根據需要設置不同的顏色
    background_color = [0, 0, 0]  # 黑色
    foreground_color = [255, 255, 255]  # 白色

    output_image[preds == 0] = background_color
    output_image[preds == 1] = foreground_color

    return output_image
```

```python
def inference(model, image_tensor, device):
    image_tensor = image_tensor.to(device)  # 將圖像張量移動到指定的設備 (CPU 或 GPU)
    with torch.no_grad():  # 禁用梯度計算 (在推理階段不需要計算梯度)
        outputs = model(image_tensor)  # 將圖像張量輸入模型，獲得輸出
        preds = torch.argmax(outputs, dim=1)  # 在通道維度上選取概率最大的類別作為預測結果
    return preds.cpu().numpy()  # 將預測結果移動到 CPU 並轉換為 NumPy 陣列

def save_output(output_image, output_path, image_name):
    output_image = Image.fromarray(output_image)
    output_image.save(os.path.join(output_path, image_name))

if __name__ == '__main__':
    args = get_args()
    device = torch.device(args.device if torch.cuda.is_available() else 'cpu')

    # 使用命令列引數指定的模型
    model = UNet(n_channels=3, n_classes=2).to(device)
    # model = ResNet_UNet(num_classes=2).to(device)

    # 載入訓練好的權重，並確保載入到正確的設備
    model_weights = torch.load(args.model, map_location=device)
    model.load_state_dict(model_weights)

    # 使用 load_dataset 載入數據集
    dataset = load_dataset(args.data_path, mode='test', batch_size=args.batch_size, shuffle=False)

    max_batches = 2  # 設置要處理的批次數量

    for i, batch in enumerate(dataset):
        if i >= max_batches:
            break  # 停止處理更多批次
        images = batch['image'].to(device, dtype=torch.float32)  # 確保圖像是 float32 類型
        outputs = inference(model, images, device)

        for j in range(outputs.shape[0]):
            output_image = generate_output_image(outputs[j], images[j].shape[-2:])
            save_output(output_image, args.output_path, f"output_{i}_{j}.png")

    print("Inference complete. Results saved to:", args.output_path)
```

B. Details of your model (UNet & ResNet34_UNet)

UNet:

```python
class UNet(nn.Module):
    def __init__(self, n_channels=1, n_classes=2, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels    # n_channels: model一開始的輸入通道數
        self.n_classes = n_classes  # n_classes: model的最終輸出通道數
        self.bilinear = bilinear

        self.inC = (DoubleConvBlock(n_channels, 64))    # 先做一次DoubleConvBlock, 之後再做down
        self.down1 = (down_block(64, 128))
        self.down2 = (down_block(128, 256))
        self.down3 = (down_block(256, 512))
        self.down4 = (down_block(512, 1024))

        self.up1 = (up_block(1024, 512, bilinear))
        self.up2 = (up_block(512, 256, bilinear))
        self.up3 = (up_block(256, 128, bilinear))
        self.up4 = (up_block(128, 64, bilinear))
        self.outC = (OutConv(64, 2))

    def forward(self, x):
        # print(f"Input shape: {x.shape}")
        x1 = self.inC(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        # print(f"After down4: {x5.shape}")     // debugging line
        x = self.up1(x5, x4)    # 上採樣有兩個參數: 將下採樣過程中的特徵圖與上採樣過程中的特徵圖結合，以保留更多的空間信息
        # print(f"After up1: {x.shape}")        // debugging line
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        output = self.outC(x)
        # print(f"Output shape: {output.shape}")  // debugging line
        return output
```

```python
# 圖像維度減少2: 因為卷積核是3x3 且沒有進行零填充
""" stride: 步長, 代表卷積核每次移動 1 像素, 默認值為1. dilation: 膨脹率, 表示卷積核內部元素間的距離, 默認為1
    padding: 填充, 在輸入特徵圖的邊緣填充額外的像素(填充值為0), 默認為0 表示不填充 """

class DoubleConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, dilation=1, padding=1): # out_channel
        super(DoubleConvBlock, self).__init__()

        self.in_channels = in_channels   # 用來保存初始化時傳入的參數
        self.out_channels = out_channels
        self.stride = stride
        self.dilation = dilation
        self.padding = padding

        # bias = False: 因為BatchNorm中就提供了Bias的效果，所以這裡就不需要了
        conv1 = nn.Conv2d(int(self.in_channels), int(self.out_channels), kernel_size=3, stride=1, padding=int(self.padding), bias=False)
        bn1 = nn.BatchNorm2d(int(self.out_channels), affine=False)     # 做正則化, affine: 決定了該層是否有可學習的縮放, 平移參數(gamma and beta)
        relu1 = nn.ReLU(inplace = True)
        # 第二次convolution時, 進去跟出來的channel不變
        conv2 = nn.Conv2d(int(self.out_channels), int(self.out_channels), kernel_size=3, stride=1, padding=int(self.padding), bias=False)
        bn2 = nn.BatchNorm2d(int(self.out_channels), affine=False)

        relu2 = nn.ReLU(inplace = True)  # inplace: 代表是否創建一個新的Tensor來儲存ReLU後的數據, True代表直接在輸入數據上進行(inplace)

        UNet_block_list = []    # 創一個列表, 儲存一系列的層操作
        UNet_block_list.append(conv1)
        UNet_block_list.append(bn1)
        UNet_block_list.append(relu1)
        UNet_block_list.append(conv2)
        UNet_block_list.append(bn2)
        UNet_block_list.append(relu2)
        self.net = nn.Sequential(*UNet_block_list)  # *為解包運算符, 將list中每個元素作為獨立的參數傳給nn.Sequential

    def forward(self, x):
        for layer in self.net:
            x = layer(x)
        return x
```

```python
class down_block(nn.Module):  # 下降階段, 先做一個maxpooling 再做DoubleConv
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            DoubleConvBlock(in_channels, out_channels),
            nn.MaxPool2d(2) # 縮小2倍
        )

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


class up_block(nn.Module):
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        if bilinear:    # 放大2倍, align_corners: 輸入和輸出tensor的角點會對齊
            self.up = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=True)
        else:
            self.up = nn.Upsample(scale_factor=2, mode="nearest")
        self.conv = DoubleConvBlock(in_channels + in_channels // 2, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)     # 對 x1 做Upsample

        # print(f"x1 shape after upsample: {x1.shape}")      // debugging line
        # print(f"x2 shape from skip connection: {x2.shape}")   // debugging line

        # input is CHW
        diffX = x2.size()[3] - x1.size()[3] # 計算寬度差 (x) (第三個維度, width)
        diffY = x2.size()[2] - x1.size()[2] # 計算高度差 (y)

        # 填充x1 以匹配x2 的尺寸
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2, diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)  # 將 x2 和填充後的 x1 沿著通道維度 (dim=1) 進行拼接

        return self.conv(x) # 最後做DoubleConv

class OutConv(nn.Module):    # 最後只做一次Conv
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1) # 這裡卷積核大小只有1x1

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

在 UNet 中，會先做一次 DoubleConvBlock，之後就是四段的下降段，所以我將下降段作為一個 block，以便讓程式碼更加簡潔，

在 Down_Block 裡面，會使用 MaxPooling 在 DoubleConvBlock 之後，以縮小特徵圖的大小 之後則是做四段上升段，上升段裡面有 Skip Connection，將來自對應下採樣階段的特徵圖與上採樣後的特徵圖拼接在一起，最後再做一個 convolutional 結束

ResNet32_UNet:

```python

import torch
import torch.nn as nn
import torch.nn.functional as F
from models.unet import DoubleConvBlock, up_block

# Reference: https://ithelp.ithome.com.tw/m/articles/10333931

""" Upsampling then double conv """
class DecoderBlock(nn.Module):

    def __init__(self, in_channels, out_channels, up_in_channel=None, up_out_channel=None):
        super().__init__()

        if up_in_channel == None:
            up_in_channel = in_channels
        if up_out_channel == None:
            up_out_channel = out_channels

        self.up = nn.ConvTranspose2d(up_in_channel, up_out_channel, kernel_size=2, stride=2)
        self.conv = DoubleConvBlock(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        x = torch.cat([x1, x2], dim=1)  # 將上採樣後的輸出(x1) 與 來自編碼器的對應層輸出(x2) 拼接
        return self.conv(x)
```

```python
# is_downsample: 是否需要進行下採樣, 如果會的話input data的維度將會縮小, 下採樣由一個1x1卷積層, 批量歸一層組成
class ResidualBlock(nn.Module): # 用於逐步提取圖像的高層特徵
    def __init__(self, in_channels, out_channels, stride=1, is_downsample=False):
        super(ResidualBlock, self).__init__()

        self.conv_x = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, stride, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),

            nn.Conv2d(out_channels, out_channels, 3, 1, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )

        self.is_downsample = is_downsample

        if self.is_downsample:
            self.down_sample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2), # stride = 2: 表示特徵圖的高度,寬度會減少一半
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        if self.is_downsample:
            residual = self.down_sample(x)  # 如果需要, 對x進行下採樣
        else:
            residual = x

        x = self.conv_x(x)
        # print(f"x: {x.shape}, residual: {residual.shape}")

        x = residual + x # residual connection, 目的是允許信息繞過卷積層直接傳遞到後面的層 -> 減少梯度消失問題, 增加梯度穩定度
        x = F.relu(x)
        return x
```

```python
class ResNet_UNet(nn.Module):
    def __init__(self, num_classes):
        super(ResNet_UNet, self).__init__()

        self.encoder1 = nn.Sequential(  # kernel_size=7: 論文敘述
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2, padding=3), # padding: 填充特徵圖的邊緣像素
            nn.BatchNorm2d(64),
            nn.ReLU()
        )

        self.pool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.encoder2 = nn.Sequential(
            ResidualBlock(in_channels=64, out_channels=64),
            ResidualBlock(64, 64),
            ResidualBlock(64, 64)
        )

        self.encoder3 = nn.Sequential(
            ResidualBlock(64, 128, stride=2, is_downsample=True),   # 縮小空間尺寸並增加通道數
            ResidualBlock(128, 128),    # ResidualBlock次數, 論文有提供
            ResidualBlock(128, 128),
            ResidualBlock(128, 128)
        )

        self.encoder4 = nn.Sequential(
            ResidualBlock(128, 256, stride=2, is_downsample=True),
            ResidualBlock(256, 256),
            ResidualBlock(256, 256),
            ResidualBlock(256, 256),
            ResidualBlock(256, 256),
            ResidualBlock(256, 256)
        )

        self.encoder5 = nn.Sequential(
            ResidualBlock(256, 512, 2, True),
            ResidualBlock(512, 512),
            ResidualBlock(512, 512),
        )
```

```python
        self.bridge = nn.Sequential(
            nn.Conv2d(512, 1024, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.decoder1 = DecoderBlock(in_channels=1024, out_channels=512)
        self.decoder2 = DecoderBlock(512, 256)  # DecoderBlock內部會幫你拼接
        self.decoder3 = DecoderBlock(256, 128)
        self.decoder4 = DecoderBlock(128, 64)
        # in_channel為128是因為: 來自前一個解碼器步驟的 upsampling 輸出(64) + encoder對應層的特徵圖(64)
        self.decoder5 = DecoderBlock(in_channels=128, out_channels=64, up_in_channel=64, up_out_channel=64)

        self.lastlayer = nn.Sequential(
            nn.ConvTranspose2d(in_channels=64, out_channels=64, kernel_size=2, stride=2),
            nn.Conv2d(64, num_classes, kernel_size=3, padding=1, bias=False)
        )
        self.drop_out = nn.Dropout(0.5)
    def forward(self, x):
        e1 = self.encoder1(x)
        pool1 = self.pool(e1)
        e2 = self.encoder2(pool1)
        e3 = self.encoder3(e2)
        e4 = self.encoder4(e3)
        e5 = self.encoder5(e4)

        bridge = self.bridge(e5)

        d1 = self.decoder1(bridge, e5)  # 兩個會在通道維度上進行拼接 (torch.cat([x1, x2], dim=1))
        d1 = self.drop_out(d1)
        d2 = self.decoder2(d1, e4)
        d2 = self.drop_out(d2)
        d3 = self.decoder3(d2, e3)
        d3 = self.drop_out(d3)
        d4 = self.decoder4(d3, e2)
        d4 = self.drop_out(d4)
        d5 = self.decoder5(d4, e1)

        out = self.lastlayer(d5)

        return out
```

Encoder 使用 ResNet34，包含多層 ResidualBlock，每個 ResidualBlock 中有兩個卷積層，並使用 residual connection 來減少梯度消失問題

Decoder 使用 UNet 結構，每個 DecoderBlock 由 upsampling 和兩個 convolution layer 組成，並透過 Concatenate 將對應的 Encoder 特徵圖與 Decoder 的特徵圖進行拼接

3.Data Preprocessing (20%)

A.How you preprocessed your data?

```python
def _preprocess_mask(mask): # 做preprocess (只有0跟1 -> 方便計算dice score)
    mask = mask.astype(np.float32)  # 將mask數據轉換為 float32類型
    mask[mask == 2.0] = 0.0      # 將所有像素值為2的 轉換成0 (像素值2 表示不確定區域 -> 轉換為背景)
    mask[(mask == 1.0) | (mask == 3.0)] = 1.0   # 像素值為 1 or 3的 轉換為1 (前景標籤)
    return mask
```

在這邊，我們對 mask 進行預處理，將值變成 0 or 1

```python
# dice_score: 衡量兩組數據相似度的指標, 等於兩倍交集大小除以兩個集合大小的總和
def dice_score(preds, masks):  # setA: predicted segmentation mask setB: ground truth mask
    smooth = 1e-10  # 用來避免除以0的情況
    preds = torch.sigmoid(preds)    # 對preds做預處理, 使其範圍在[0, 1]間 (masks已有做預處理, 值為0 or 1)
    # 使其值變成0 or 1  (if preds > 0.5 -> 轉成1 -> 加.float() -> 變成 1.0)
    preds = (preds > 0.5).float()   # 0是後(背)景, 1是前景

    intersection = (preds * masks).sum() # 只有preds 和masks等於1時 結果才為1 -> 再將其相加 -> 兩個交集的像素數量
    C, H, W = preds.shape  # 獲取preds的形狀(CHW)
    total_pixels = 2.0 * C * H * W
    dice = 2.0 * intersection / (total_pixels + smooth)
    return dice
```

並在 dice_score 計算時, preds 的資料值也預處理成 0 or 1

(因為 intersection 這樣才可方便處理)

```python
"""預處理: 對圖像和標籤進行隨機水平或垂直翻轉"""
def random_flip(sample):
    image = sample['image']
    mask = sample['mask']
    trimap = sample['trimap']

    if np.random.rand() > 0.5:
        image = np.fliplr(image).copy()
        mask = np.fliplr(mask).copy()
        trimap = np.fliplr(trimap).copy()
    if np.random.rand() > 0.5:
        image = np.flipud(image).copy()
        mask = np.flipud(mask).copy()
        trimap = np.flipud(trimap).copy()

    return {'image': image, 'mask': mask, 'trimap': trimap}
```

還有對圖像, label 進行隨機性地水平 or 垂直翻轉，以達到

Data Augmentation 的效果

B. What makes your method unique?

在 UNet 和 ResNet34 + UNet 當中，他們都有重複性的做 double convolution，所以我建了一個 DoubleConvBlock，以便將重複的操作變得更簡潔，並且在兩個 model 中皆加入了 DropOut Layer 來防止 Overfitting

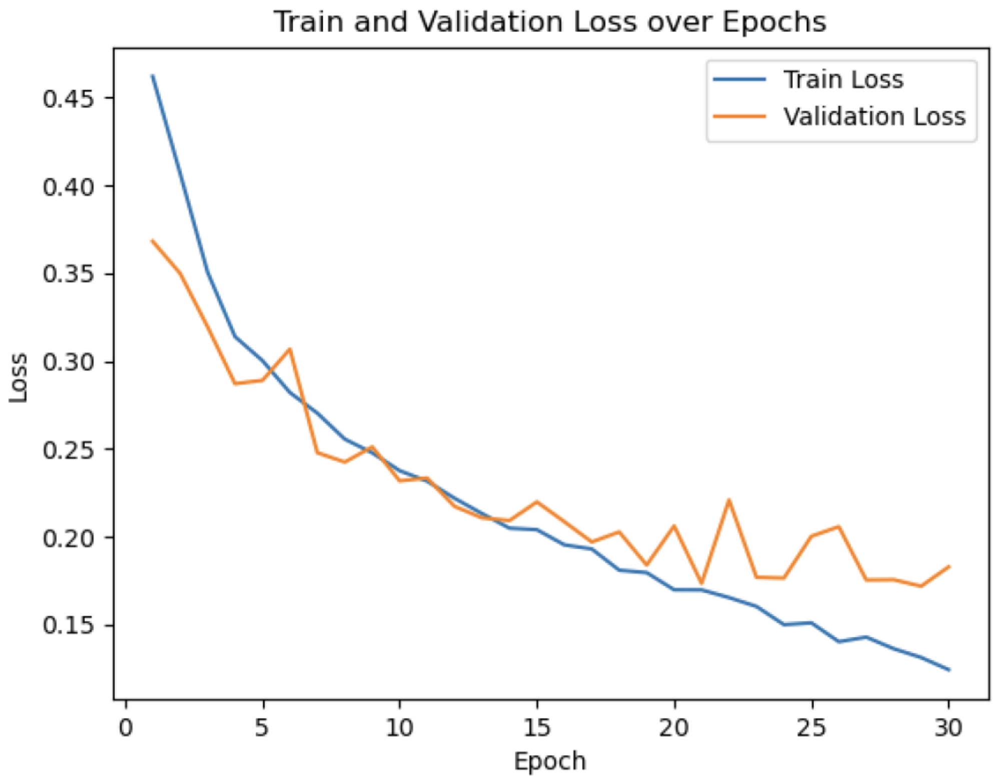
4. Analyze on the experiment results (20%)

A. What did you explore during the training process?

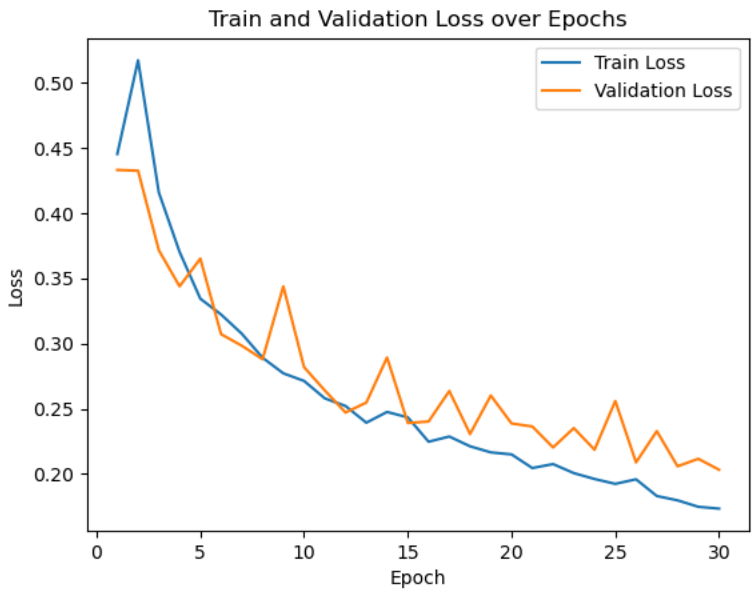
調整 hyperparameter，例如 learning rate, Batch_size, batch_size 適中會把 training 的速度調高一些，太高 GPU 會記憶體不足，無法運行

以下是 training & validation loss 圖:

UNet:



ResNet34 + UNet:



B. Found any characteristics of the data?

我觀察到了前景（Pet）和背景在顏色，紋理和形狀上有明顯

的區別，這有助於模型學習區分這兩者

並且我觀察到有些圖片的光照條件,畫質都不太一樣，有的光線充足，有的光線較弱，這對 model 如何保持一致性預測是個挑戰

5.Execution command (0%)

A.The command and parameters for the training process

```python
def get_args(): # args: arguments
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')   # 初始化一個 argparse.ArgumentParser 物件
    parser.add_argument('--data_path', type=str, default = './dataset', help='path of the input data') # 指定輸入數據的路徑
    parser.add_argument('--epochs', '-e', type=int, default=30, help='number of epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=20, help='batch size')   # 設置批量大小(總共是3311)
    parser.add_argument('--learning_rate', '-lr', type=float, default=0.00012, help='learning rate')

    return parser.parse_args()
```

這次的 epoch 比之前需要的少很多，大約 30 個就飽和了

B.The command and parameters for the inference process

```python
def get_args():
    parser = argparse.ArgumentParser(description='Model inference on new images')
    parser.add_argument('--model', type=str, default='saved_models/UNet_best_model_90.pth', help='Path to the trained model file')
    # parser.add_argument('--model', type=str, default='saved_models/ResNet_UNet_best_model_90.pth', help='Path to the trained model file')
    parser.add_argument('--data_path', type=str, default='./dataset', help='Path to the input data folder')
    parser.add_argument('--output_path', type=str, default='.', help='Path to save the output results')   # 預設存在當前目錄
    parser.add_argument('--batch_size', '-b', type=int, default=10, help='Batch size for inference')
    parser.add_argument('--device', default='cuda', help='Device to use for inference (cuda or cpu)')
    return parser.parse_args()
```

6.Discussion(20%)

A.What architecture may bring better results?

增加 DropOut 層以預防 Overfitting 的現象，並且將 optimizer 的 SGD 改成 Adam 可以讓 dice score 從 0.82 到 0.9

並且將學習率維持在 0.0001 或是 0.00012 差不多，可以達到 Dice Score 最大化，並且調整到適中的 Batch_size

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

Potential Research Topics: 可以探討不同的 Data Augmentation 技術對模型訓練的影響，特別是在光照條件變化大的場景中，如隨機裁剪、旋轉和顏色變換

或是在在不損失精度的情況下，研究如何降低模型的計算複雜度和 parameter 數量，使其適用於移動設備或嵌入式系統