

NYCU DL Lab3

Binary Semantic Segmentation

313552049 鄭博涵

1. Overview of your lab 3 (10%)

這次實驗的目的是要實作兩種不同的二元語義分割模型(Binary Semantic Segmentation)，分別是 UNet 和 ResNet34 結合 UNet 的模型，並將這些模型訓練於 Oxford-IIIT Pet Dataset 上，我們期望 model 的預測結果會是一個僅包含 0 和 1 的矩陣，其中 1 代表模型預測該位置為物件，而 0 代表背景。實驗的目標是透過計算預測結果與實際標註之間的 Dice Score 來評估模型的表現，希望計算結果越高越好

2. Implementation Details (30%)

A. Details of your training, evaluating, inferencing code

Training code:

```
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
```

首先透過 `torch.device` 來決定指定設備是否要用 GPU 進行訓練

若無法使用 CUDA 則使用 CPU

再來初始化 ResNet_UNet 模型(或是UNet模型)，並將其移動至指

定設備上，損失函數使用CrossEntropyLoss, optimizer選用 Adam

或 SGD

再透過 `load_dataset` 函數將資料分成 `train_dataset` 和 `val_dataset`, 其中訓練資料隨機打亂以增加訓練多樣性，驗證資料則不打亂以保持其順序性

`model.train()`: 將 `model` 進入 `train` 模式，並逐批（batch）載入資料進行訓練

每個批次中，影像資料與其對應的標註遮罩被載入 GPU，並進行 Forward pass 以生成模型預測，接著計算預測結果與實際標註之間的損失，並進行 backward 來更新模型參數

然後每隔 50 個批次，會輸出當前批次編號以追蹤訓練進度

```
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()
```

這段程式碼的主要功能是針對了 `model` 進行每一個 epoch 的驗證，最佳模型的保存，以及訓練過程中損失曲線的可視化，接著讓我來逐步介紹這段程式碼：

`batch_size = images.size(0)`: 計算當前批次的樣本數

`running_loss`: 用來儲存當前批次的損失總和，然後再除以總樣本數會得到平均損失 (`avg_epoch_loss`)

接著將每個 epoch 的平均損失值儲存到 `train_losses` 中
透過 `validate` 函數，計算驗證資料集的損失值和 Dice Score，並將結果分別儲存在 `val_losses` 和 `val_dice_scores` 中
在每個 epoch 結束後，將當前的平均損失，驗證損失以及 Dice Score 輸出顯示

如果當前的 Dice Score 優於之前的最佳值，則更新 `best_dice_score`，並將當前模型的狀態保存為最佳模型
損失曲線的繪製：

使用 Matplotlib 繪製訓練和驗證損失值的變化曲線，以視覺化模型在訓練過程中的表現

最後，將曲線保存為圖檔以供進一步分析

Evaluating code:

```
4   # test階段
5   # Evaluation
6   ~ import argparse
7   import torch
8   from utils import dice_score
9   from models.unet import UNet
10  from models.resnet34_unet import ResNet_UNet
11  from oxford_pet import load_dataset
12
13
14 ~ def evaluate(net, data_loader, device):
15     net.eval()
16     total_loss = 0
17     total_dice_score = 0
18     num_batches = len(data_loader)
19
20     criterion = torch.nn.CrossEntropyLoss()
21
22 ~     with torch.no_grad():
23 ~         for data in data_loader:
24             images = data['image'].to(device, dtype=torch.float32)
25             masks = data['mask'].to(device, dtype=torch.float32)
26
27             # 將masks從[batch_size, 1, height, width] 變為[batch_size, height, width]
28             masks = masks.squeeze(1).long() # 確保masks是LongTensor
29
30             outputs = net(images)
31             loss = criterion(outputs, masks)
32             total_loss += loss.item()
33
34             dice = dice_score(outputs, masks)
35             total_dice_score += dice
36
37     avg_loss = total_loss / num_batches
38     avg_dice_score = total_dice_score / num_batches
39
40     return avg_loss, avg_dice_score
```

net.eval(): 將模型設定為評估模式

torch.no_grad(): 確保在評估過程中不會計算梯度，這樣可以節省記

憶體並加快運算速度

total_loss 和 total_dice_score: 用於累積整個資料集的損失和 Dice Score

num_batches: 代表資料集被分割成的批次數量，用於計算最終的平均損失和 Dice Score

然後設定Loss function, 使用交叉熵損失 (CrossEntropyLoss)

再從 data_loader 中逐批讀取影像資料 (images) 和其對應的標註遮罩(masks) 輸入模型進行預測，並計算損失值與 Dice Score, 然後

累積至total_loss 和 total_dice_score 中

avg_loss 和 avg_dice_score: 分別通過total_loss與 total_dice_score

除以批次數量計算得到，最終返回這兩個值作為模型在整個資料上的性能指標

```
✓ 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}")
```

Argparse: 用來解析命令行參數，允許使用者自定義：資料路徑 (data_path), 訓練輪數 (epochs), 批次大小 (batch_size), 學習率 (learning_rate) 等重要參數

再來是

device = torch.device("cuda" if torch.cuda.is_available() else 'cpu'):

程式自動檢查 CUDA 是否可用，若可以則使用 GPU, 不行則用

cpu

再來test dataset透過 load_dataset 函數載入，並設定順序不打亂

接著初始化一個 UNet 或 ResNet-UNet 模型，並將模型移至指定

裝置(CPU or GPU)

再載入事先訓練好的模型權重，準備進行推論，這裡提供了多個不

同的預訓練模型權重檔案供選擇

使用 evaluate 函數在測試資料集上進行推論，計算模型的avg_loss
和avg_dice_score

最後將這些評估結果顯示出來，以供使用者了解模型的性能。

Inferencing code:

```
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 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()

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
```

Argparse 如同上面，有解釋過

接著是影像預處理 (preprocess_image) 的部分：

將影像大小調整為 (256, 256)，並將其轉換為 PyTorch 的張量格式

最後，函數會將處理後的影像增加一個批次維度，使其符合模型輸入的要求

再來是輸出影像生成 (generate_output_image)的部分：

這個函數根據模型的預測結果 (preds) 和影像尺寸生成對應的可視化影像。

我們假設預測結果為二值遮罩，其中 0 代表背景，1 代表前景

並背景設置為黑色，前景設置為白色，最終返回彩色的輸出影像

```
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)
```

首先是推論函數 (inference) :

這個函數接收模型，影像張量以及設備（CPU or GPU）作為輸入，並將影像張量轉移到指定設備，然後進行forward pass計算再利用 torch.argmax 從模型輸出的多通道結果中選擇概率最大的類別作為預測結果，並將其轉換為 NumPy 陣列格式返回

再來是結果保存函數 (save_output):

該函數將推論結果轉換為影像格式，並將其保存到指定的路徑下再使用 PIL 的 Image.fromarray 函數將 NumPy 陣列轉換為影像，並利用 os.path.join 來組合輸出路徑和影像名稱

3. main function:

解析命令行參數，設定運行裝置（CPU 或 GPU），並初始化model，再載入事先訓練好的模型權重，並將測試資料集載入為 DataLoader。

再來迭代test dataset進行推論，並將每張影像的推論結果保存為影像檔案，程式碼限制了最大批次數以避免處理過多資料推論結束後，將結果保存起來到當前目錄中

B. Details of your model (UNet & ResNet34_UNet)

UNet:

```
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

# 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, 過去跟出來的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
```

```

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)

```

首先在 UNet 架構中，核心的組件包括了 DoubleConvBlock, Down_Block, Up_Block 以及 OutConv 模組，這些模組的設計使得 UNet 能夠有效地進行影像分割任務，那我現在來逐一講解這些模組：

DoubleConvBlock：這個模組包含兩層卷積操作，每層之後接一個 Batch Normalization 以及 ReLU 激活函數，其作用是逐步提取影像的特徵，同時減少 overfitting 的風險

Down_Block: 由 DoubleConvBlock 和 MaxPooling 組成，實現了對影像的降採樣，提取更具抽象層次的特徵，並同時減少影像的

空間維度

Up_Block: 由上采樣（Upsampling）操作和 DoubleConvBlock 組成，用來恢復影像的空間解析度，在上采樣之後，我使用了“Skip Connection”來融合下採樣階段中提取的特徵，這種結構能夠保留更多的上下文信息，有助於提高分割的準確性

OutConv: 最後的輸出卷積層，將通道數從較大的特徵數量壓縮到需要分割的類別數，使用 1×1 的卷積核來確保輸出的空間大小不變，但僅調整通道數量

接下來是每個模組的實作細節：

Down_Block: 每次通過 MaxPooling 操作後，影像的空間維度（Height 和 Width）都會減半，但通道數（Channels）則會隨著卷積層的運算逐漸增加，這表示影像在越深層次的特徵圖中，空間資訊雖然減少，但更豐富的語義特徵會被提取出來

Up_Block: 使用了上采樣操作來逐步恢復影像的空間尺寸，通過與下採樣階段的對應層進行特徵融合（通過 Skip Connection），保證了上層特徵的細節與下層特徵的上下文信息得以有效結合。

ResNet32_UNet:

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from models.unet import DoubleConvBlock, up_block
5
6 # Reference: https://ithelp.ithome.com.tw/m/articles/10333931
7
8 """ Upsampling then double conv """
9
10 class DecoderBlock(nn.Module):
11
12     def __init__(self, in_channels, out_channels, up_in_channel=None, up_out_channel=None):
13         super().__init__()
14
15         if up_in_channel == None:
16             up_in_channel = in_channels
17         if up_out_channel == None:
18             up_out_channel = out_channels
19
20         self.up = nn.ConvTranspose2d(up_in_channel, up_out_channel, kernel_size=2, stride=2)
21         self.conv = DoubleConvBlock(in_channels, out_channels)
22
23     def forward(self, x1, x2):
24         x1 = self.up(x1)
25         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
26         return self.conv(x)
27
28
# 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
29 class ResidualBlock(nn.Module): # 用於逐步提取圖像的高層特徵
30     def __init__(self, in_channels, out_channels, stride=1, is_downsample=False):
31         super(ResidualBlock, self).__init__()
32
33         self.conv_x = nn.Sequential(
34             nn.Conv2d(in_channels, out_channels, 3, stride, padding=1),
35             nn.BatchNorm2d(out_channels),
36             nn.ReLU(inplace=True),
37
38             nn.Conv2d(out_channels, out_channels, 3, 1, padding=1),
39             nn.BatchNorm2d(out_channels),
40             nn.ReLU()
41         )
42
43         self.is_downsample = is_downsample
44
45         if self.is_downsample:
46             self.down_sample = nn.Sequential(
47                 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
48                 nn.BatchNorm2d(out_channels)
49             )
50
51     def forward(self, x):
52         if self.is_downsample:
53             residual = self.down_sample(x) # 如果需要，對x進行下採樣
54         else:
55             residual = x
56
57         x = self.conv_x(x)
58         # print(f'x: {x.shape}, residual: {residual.shape}')
59
60         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
61         x = F.relu(x)
62
63         return x
```

```

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),
        )

    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

```

首先, ResNet-UNet 結合了 ResNet 和 UNet 的優勢, 將深度殘差學習 (Residual Learning) 與對稱的編碼器-解碼器結構結合。那我接下來來講解實作細節:

encoder部分:

encoder 包含了 ResNet 的多個 Residual Block, 每個 Residual Block 內部包含兩層卷積層, 並通過一個跳躍連接 (Skip Connection) 將輸入直接傳遞到輸出, 而進行卷積運算時, 影像的空間維度 (Height 和 Width) 會減半(因為 stride = 2), 而通道數 (Channels) 則會逐步增加, 這意味著影像的特徵圖變得更加精細。

decoder部分 :

在 DecoderBlock 裡, 使用上采樣 (Upsampling) 操作來逐步恢復影像的空間尺寸, 每次上采樣後, 使用 DoubleConvBlock 進行卷積運算, 同時將來自編碼器的對應層特徵圖通過 Concatenate 操作進行融合。這樣一來, 解碼器能夠獲得更多的空間和上下文信息, 這對於精確分割目標區域至關重要。

3. Data Preprocessing (20%)

A. How you preprocessed your data?

```
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，以方便計算 dice score，具體來說是將所有像素值為 2 的 轉換成 0，(像素值為 2 代表不確定區域)，再來像素值為 1 or 3 的轉成 1，最後 return mask

```
# 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
```

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

具體來說是將 $\text{preds} > 0.5$ 的轉成 1, $\text{preds} \leq 0.5$ 的轉成 0

-> 因為 intersection 這樣才可方便處理

```
"""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}
```

random_flip(sample) 函數的主要目的是對輸入的影像樣本進行隨機的水平或垂直翻轉，以下是實作細節：

首先從輸入的 sample 字典中提取出影像、標註和 trimap 三個元素

再來使用 np.random.rand() 生成一個在 [0, 1) 之間的隨機數，當該數值大於 0.5 時，對影像和標註進行水平翻轉操作，np.fliplr 函數實現了對影像進行左右翻轉，並且使用 copy() 確保生成的翻轉結果為新的數據副本，而非對原數據的引用，

同理，隨機數大於 0.5 時，使用 np.flipud 函數對影像進行上下翻轉，這一操作通過改變影像的垂直方向來增加資料的多樣性最終，將經過可能翻轉操作的影像及其標註組合為一個新的字典返回，供後續的處理步驟使用

B. What makes your method unique?

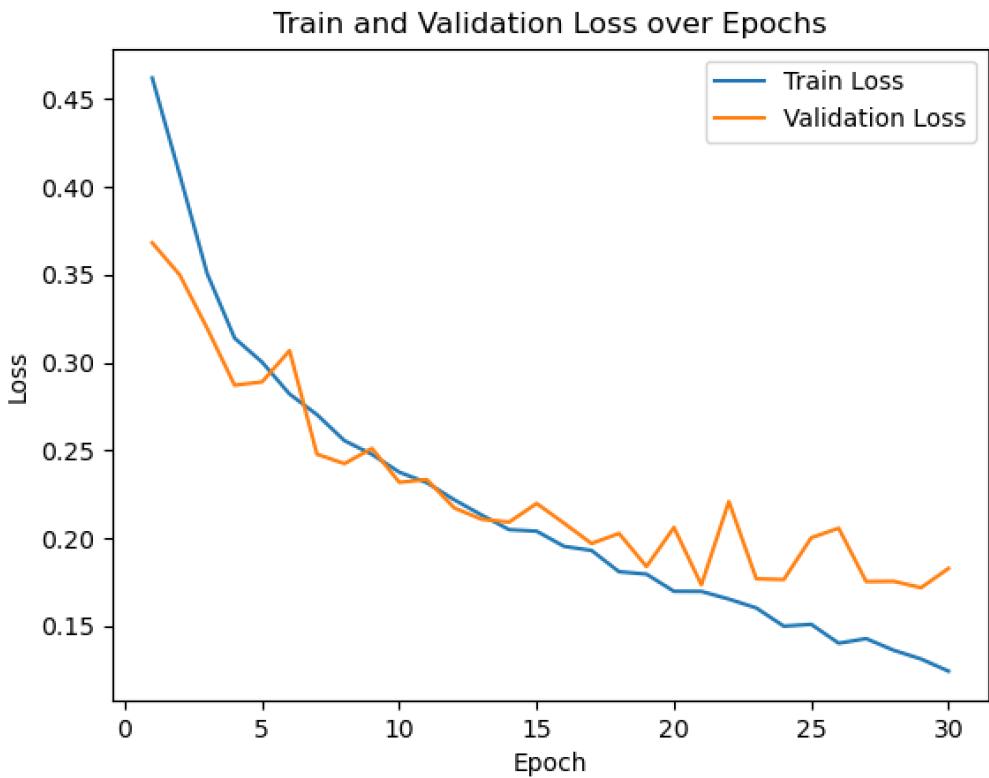
在 UNet 和 ResNet34 + UNet 當中，他們都有重複性的做 double convolution，所以我建了一個 DoubleConvBlock，以便將重複的操作變得更簡潔，並且在兩個 model 中皆加入了 DropOut Layer 來防止 Overfitting，除此之外，使用了 data augmenting 技術，才能夠有效提升模型在不同視角下的識別能力，通過在訓練過程中不斷變換影像的方向，防止模型過度擬合於特定的影像模式或方向性特徵

4. Analyze on the experiment results (20%)

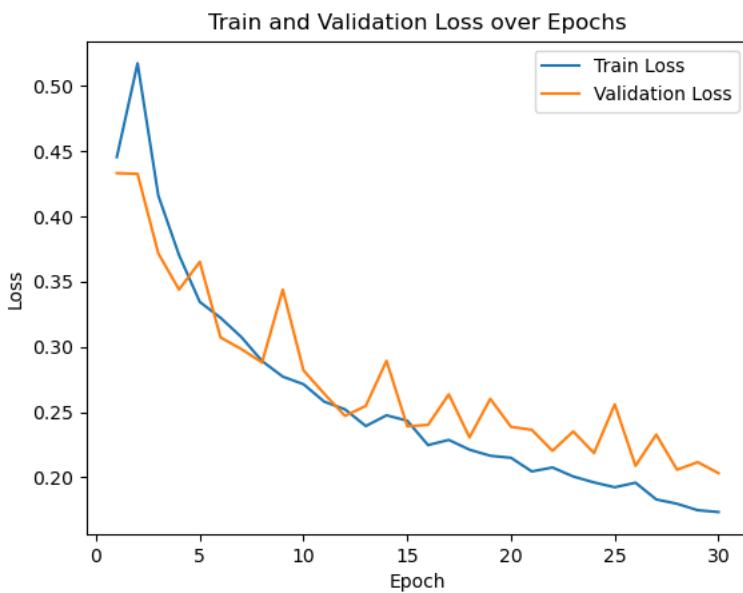
A. What did you explore during the training process?

調整 hyperparameter，例如 learning rate, Batch_size, batch_size 適中會使 training 的速度變快一些，太高 GPU 會記憶體不足，無法運行，然後 optimizer 的選擇蠻重要的
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

在這之前要先將資料集存入 dataset/oxford-iiit-pet，將 model 權重存

入 saved_models 資料夾，如下：

```
└── dataset/oxford-iiit-pet
    ├── annotations
    ├── images
    └── saved_models
        ├── DL_Lab3_ResNet34_UNet_313552049_鄭博涵.pth
        └── DL_Lab3_UNet_313552049_鄭博涵.pth
└── src
    ├── __pycache__
        ├── oxford_pet.cpython-312.pyc
        ├── utils.cpython-312.pyc
    ├── models
    ├── evaluate.py
    ├── inference.py
    ├── oxford_pet.py
    ├── train.py
    ├── utils.py
    └── Report.pdf
```

然後在 command line 執行：

```
python src/train.py --data_path ./dataset/oxford-iiit-pet
```

成功的話會顯示出以下結果：

```

(jack) (base) sqoojack@sqoojack-computer:~/文件/博涵公司/交大/暑修-深度學習/lab03/DL_Lab3_313552049_鄭博涵$ python src/train.py --data_p
ath ./dataset/oxford-iiit-pet
Current Batch Number: 1
Current Batch Number: 51
Current Batch Number: 101
Current Batch Number: 151
Epoch: 1, Train Loss: 0.46341942650252493, Validation Loss: 0.3951, Dice Score: 0.8069
Model saved with Dice Score: 0.8069
Current Batch Number: 1
Current Batch Number: 51

```

B. The command and parameters for the inference process:

Evaluate.py:

ResNet:

首先要將 line 68 行的 Lab3_ ResNet34 改成 Lab3_ResNet34 才對
(也就是這邊多打一個空格)

```

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

```

接著 command line 打:

```
python src/evaluate.py --data_path ./dataset/oxford-iiit-pet
```

成功的話會需要等 1 分鐘左右，會跑出以下畫面：

(當初交 code 時沒有顯示進度條，所以要稍等一下才會顯示出結果)

```

FileNotFoundException: [Errno 2] No such file or directory: './dataset/annotations/test.txt'
(jack) (base) sqoojack@sqoojack-computer:~/文件/博涵公司/交大/暑修-深度學習/lab03/DL_Lab3_313552049_鄭博涵$ python src/evaluate.py --data_path
./dataset/oxford-iiit-pet
Evaluation Loss: 0.20652328710193218, Evaluation Dice score: 0.9033476114273071

```

UNet:

```

60     # model = UNet(n_channels=3, n_classes=2).to(device)
61     model = ResNet_UNet(num_classes=2).to(device)
62
63     # 載入訓練好的權重
64     # model_weights = torch.load('saved_models/ResNet_UNet_best_model_90.pth', weights_only=True)
65     # model_weights = torch.load('saved_models/UNet_best_model_90.pth', weights_only=True)
66
67     # model_weights = torch.load('saved_models/DL_Lab3_UNet_313552049_鄭博涵.pth', weights_only=True)
68     model_weights = torch.load('saved_models/DL_Lab3_ResNet34_UNet_313552049_鄭博涵.pth', weights_only=True)

```

將 line 61, 68 的註解去掉，並將 line 60, 67 加上註解

並在 command line 打:

```
python src/evaluate.py --data_path ./dataset/oxford-iiit-pet
```

跑的時候，要等約 2-3 分鐘，成功的話會跑出以下畫面：

```
(jack) (base) sqoojack@sqoojack-computer:~/文件/博涵公司/交大/暑修-深度學習/lab03/DL_Lab3_313552049_鄭博涵$ python src/evaluate.py --data_path ./dataset/oxford-iiit-pet
Evaluation Loss: 0.17526986430520597, Evaluation Dice score: 0.9199652671813965
```

Inference.py:

UNet:

直接執行 command line:

```
python src/inference.py --model
```

```
saved_models/DL_Lab3_UNet_313552049_鄭博涵.pth --
```

```
data_path ./dataset/oxford-iiit-pet
```

執行完結果會長這樣：

```
输出_1_0.png
输出_1_1.png
输出_1_2.png
输出_1_3.png
输出_1_4.png
输出_1_5.png
输出_1_6.png
输出_1_7.png
输出_1_8.png
输出_1_9.png
Report.pdf
57
58 # 初始化模型結構
問題 輸出 偵錯主控台 終端機 連接埠
(jack) (base) sqoojack@sqoojack-computer:~/文件/博涵公司/交大/暑修-深度學習/lab03/DL_Lab3_313552049_鄭博涵$ python src/inference.py --model save
d_models/DL_Lab3_UNet_313552049_鄭博涵.pth --data_path ./dataset/oxford-iiit-pet
/home/sqoojack/文件/博涵公司/交大/暑修-深度學習/lab03/DL_Lab3_313552049_鄭博涵/src/inference.py:69: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module `implicity`. It is possible to construct malicious pickle data which can execute arbitrary code during unpickling. See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more detail). In a future release, the default value for `weights_only` will be `False` due to the nature of the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowedlist by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
model_weights = torch.load(args.model, map_location=device)
Inference complete. Results saved to: .
```

(這邊我沒有將 `torch.load` 的 `weights_only` 設為 `True`，所以會跑出上面那些訊息，但仍可執行，並將照片存到主目錄中 `output_0_1.png`

~ `output_1_9.png` 的部分)

ResNet:

執行 ResNet 時，在 inference.py 的 65 行要把他註解掉，並

66 行的註解拿掉

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

然後 command line 輸入：

```
python src/inference.py --model
```

```
saved_models/DL_Lab3_ResNet34_UNet_313552049_鄭博涵.pth --
```

```
data_path ./dataset/oxford-iiit-pet
```

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

除此之外，可以嘗試使用更多樣的資料增強技術，如隨機旋轉、隨

機裁剪、隨機亮度調整等，可以進一步提高模型的泛化能力，從而

在未知資料上的表現更佳

並且除了 Adam 優化器，還可以嘗試其他如 RMSprop、AdamW

等優化器，可能可以比 Adam 更能收斂到更好的結果

再來還可以使用動態學習率調度（例如 ReduceLROnPlateau 或

Cosine Annealing）來根據模型的學習進度自動調整學習率，避免

過早陷入局部最小值或模型的早停，除此之外也可以考慮使用混合

精度訓練（Mixed Precision Training）來加速訓練過程，同時減少

內存佔用，這樣不僅能提升訓練速度，還能允許使用更大的 Batch

Size，這樣可能可以獲得更好的結果

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

我們可以探討不同的 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

除此之外，我們可以探討如何將影像分割與其他相關任務（如目標檢測、邊界檢測等）結合在一起進行 Multi-task Learning，從而提高模型的整體性能並減少單一任務的過擬合風險

或是我們可以探討如何在影像分割任務中來進行不確定性評估，研究如何利用 Bayes 方法或集成模型來量化模型的預測不確定性，這對於關鍵任務（如醫療影像診斷）尤為重要