Lab2 Binary Semantic Segmentation

資訊科學與工程研究所 313551079 張芷烜

1. Implementation Details

Lab2 目標是實作 UNet 和 ResNet34 + UNet 架構，藉由 Oxford-IIIT dataset 訓練完成 segmentation 的任務，區分出物件與背景（1 代表物件 0 代表背景）· 最後以 Dice score 為指標評估結果 ·

I. Training

Loss function : 0.3×BCE Loss+0.7×Dice Loss, Optimizer : Adam ,Epochs : 100 epochs, tqdm 顯示訓練進度 · 透過 best dice score 記錄每次 validation 的結果，如果結果比上次 train 的結果好則儲存 model ·

Forward : output = model(images) 再透過 loss 計算 output 跟 mask 之間的損失

Backward : loss.backward( ) 計算梯度 optimizer.step( ) 更新權重

計算出平均的 train_dice 再決定要不要儲存新的 model

```
best_val_dice = 0.0
# training loop
for epoch in range(args.epochs):
    model.train()
    train_loss = 0
    train_dice = 0
    for batch in tqdm(train_loader, desc="Training", leave=False):
        images, masks = batch["image"].to(device, dtype=torch.float32), batch["mask"].to(device, dtype=torch.float32)

        optimizer.zero_grad()
        outputs = model(images)
        loss = 0.3*criterion(outputs, masks) + 0.7*dice_loss(outputs, masks)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

        with torch.no_grad():
            train_dice += dice_score(outputs, masks).item()

    avg_train_loss = train_loss / len(train_loader)
    avg_train_dice = train_dice / len(train_loader)
    print(f"Epoch {epoch+1}/{args.epochs} - Train Loss: {avg_train_loss:.4f}, Dice Score: {avg_train_dice:.4f}")


    # validation loop
    avg_val_loss, avg_val_dice = evaluate(model, valid_loader, device)
    print(f"Epoch {epoch+1}/{args.epochs} - Validation Loss: {avg_val_loss:.4f}, Dice Score: {avg_val_dice:.4f}")
    # Save model
    if avg_val_dice > best_val_dice:
        best_val_dice = avg_val_dice
        save_path = f"saved_models/{model_name}_best.pth"
        torch.save(model.state_dict(), save_path)
        print(f"Model saved to {save_path} (Best Dice Score: {best_val_dice:.4f})")
```

II. Evaluation

將 model 設為 eval mode，關閉 BatchNorm 和 Dropout 並且使用訓練時儲存的均值與標準差 · 因為在 evluate 的時候 model 不需要更新，這能讓推論結果與訓練時的最佳模型一致，而不會因為 BN 或 Dropout 影響表現 ·

```python
def evaluate(model, dataloader, device):
    model.eval()
    dice_total = 0
    loss_total = 0
    criterion = torch.nn.BCEWithLogitsLoss()

    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Evaluating", leave=False):
            images, masks = batch["image"].to(device), batch["mask"].to(device)

            outputs = model(images)
            loss = 0.3*criterion(outputs, masks)+0.7*dice_loss(outputs, masks)  # BCE Loss + Dice Loss

            dice_total += dice_score(outputs, masks).item()
            loss_total += loss.item()

    avg_dice = dice_total / len(dataloader)
    avg_loss = loss_total / len(dataloader)

    # print(f"Validation - Loss: {avg_loss:.4f}, Dice Score: {avg_dice:.4f}")
    return avg_loss, avg_dice
```

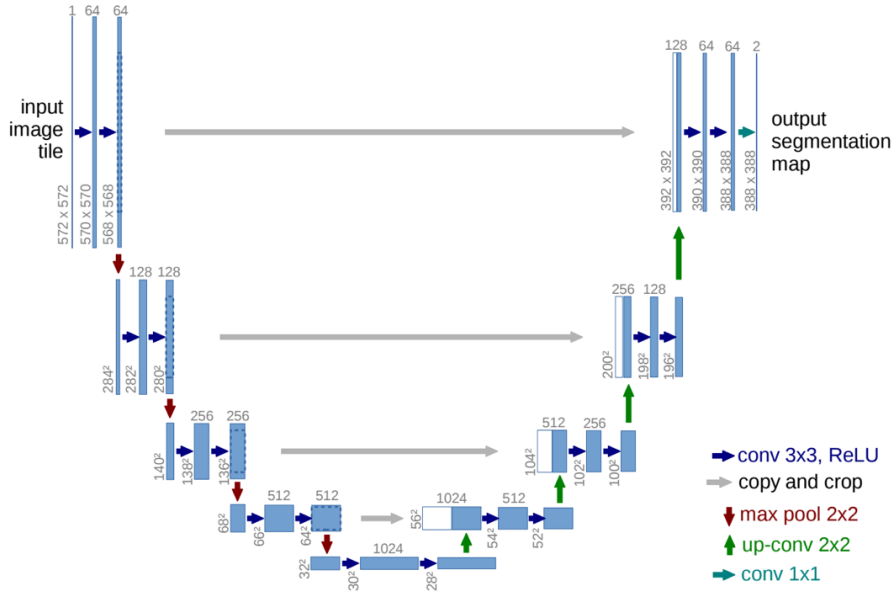
III. Inference

將儲存的 model load 進來，呼叫 evaluate 去計算評估結果，這邊要注意要使用 test data，其他步驟和 evaluate 一樣，直接使用 model.eval( )，在不更新 model 的情況下預測結果，這邊利用 utils.py 中的 plot_comparison 去可視化結果圖以供實驗比較用。

```python
model = load_model(args.model_path, args.model_type, device)

# Load dataset (Test Mode)
test_loader = load_dataset(args.data_path, "test", args)

with torch.no_grad():
    for i, batch in enumerate(tqdm(test_loader, desc="Running Inference")):
        images = batch["image"].to(device, dtype=torch.float32)
        gt_masks = batch["mask"].to(device, dtype=torch.float32)

        outputs = model(images)

        # Sigmoid activation and thresholding
        outputs = torch.sigmoid(outputs)
        preds = (outputs > 0.5).float()  # turn to 0 or 1

        # save predictions
        # pred_save_path = os.path.join(predictions_dir, f"prediction_{i}.png")
        # save_image(preds, pred_save_path)

        # save comparisons
        comparison_save_path = os.path.join(comparisons_dir, f"comparison_{i}.png")
        plot_comparison(images[0], gt_masks[0], preds[0], comparison_save_path)
        print(f"Comparison saved: {comparison_save_path}")
# evaluate
avg_loss, avg_dice = evaluate(model, test_loader, device)
print(f"Test - Loss: {avg_loss:.4f}, Dice Score: {avg_dice:.4f}")
```

```python
def plot_comparison(image, gt_mask, pred_mask, save_path):

    fig, ax = plt.subplots(1, 3, figsize=(12, 4))

    # tensor → numpy
    image = image.cpu().numpy().transpose(1, 2, 0)  # (C, H, W) → (H, W, C)
    gt_mask = gt_mask.cpu().numpy().squeeze()  # (1, H, W) → (H, W)
    pred_mask = pred_mask.cpu().numpy().squeeze()  # (1, H, W) → (H, W)

    # plot
    ax[0].imshow(image)
    ax[0].set_title("Input Image")
    ax[0].axis("off")

    ax[1].imshow(gt_mask, cmap="gray")
    ax[1].set_title("Ground Truth Mask")
    ax[1].axis("off")

    ax[2].imshow(pred_mask, cmap="gray")
    ax[2].set_title("Predicted Mask")
    ax[2].axis("off")

    plt.tight_layout()
    plt.savefig(save_path, bbox_inches='tight')
    plt.close()
```

IV. UNet



UNet 架構分成 encoder 跟 decoder，中間藉由 skip connection 傳遞特徵資訊，防止模型再多層學習下忘記前面的特徵，每層輸入的特徵會經過 2 層的 3*3 conv 和 ReLU，再經過 max pooling down sampling 到 bottleneck 之後再經過 2*2 conv up sampling，最後透過 1*1 的 conv 讓 output 與 image size 對齊，

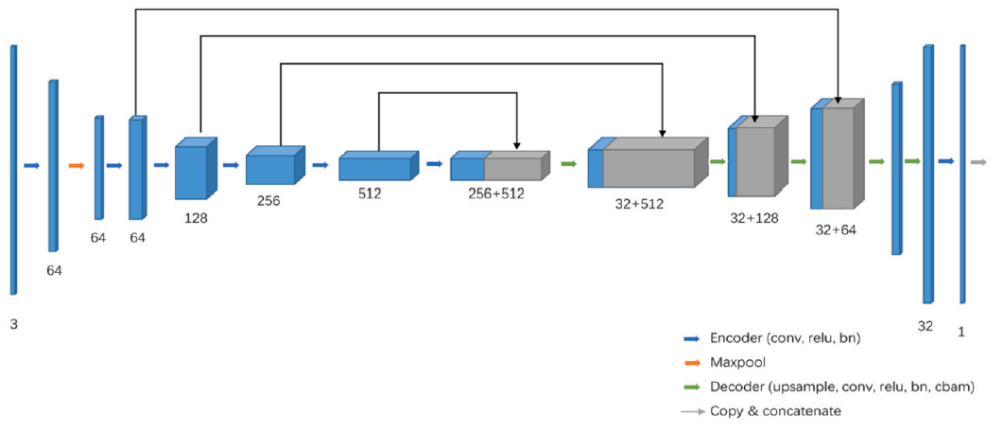
```
class Encoder(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),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

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

```
class Decoder(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Decoder, self).__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.decoder = nn.Sequential(
            nn.Conv2d(out_channels * 2, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip_connection):
        x = self.upconv(x)
        x = torch.cat([x, skip_connection], dim=1)
        return self.decoder(x)
```

Decoder 中要實作 skip connection 把這層的 x 先 up sample 之後再與上層的 skip connection x 做 concatenate，以實現 skip connection 的功能，這邊要很注意 concatenate 的時候 size 與 channel 有沒有對應好，

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

        # Encoder
        self.encoder1 = Encoder(in_channels, 64)  # 256*256*3 -> 256*256*64
        self.encoder2 = Encoder(64, 128)  # 256*256*64 -> 128*128*128
        self.encoder3 = Encoder(128, 256) # 128*128*128 -> 64*64*256
        self.encoder4 = Encoder(256, 512) # 64*64*256 -> 32*32*512
        self.encoder5 = Encoder(512, 1024) # bottleneck 32*32*512 -> 16*16*1024

        # Max pooling
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Decoder
        self.decoder1 = Decoder(1024, 512) # 16*16*1024 -> 32*32*512
        self.decoder2 = Decoder(512, 256) # 32*32*512 -> 64*64*256
        self.decoder3 = Decoder(256, 128) # 64*64*256 -> 128*128*128
        self.decoder4 = Decoder(128, 64) # 128*128*128 -> 256*256*64

        # Output
        self.output = nn.Conv2d(64, out_channels, kernel_size=1) # 256*256*64 -> 256*256*1

    def forward(self, x):
        # Encoder
        x1 = self.encoder1(x) # 256*256*3 -> 256*256*64
        x2 = self.encoder2(self.pool(x1)) # self.pool(x1) -> 128*128*64 endcode -> 128*128*128
        x3 = self.encoder3(self.pool(x2)) # 64*64*128 -> 64*64*256
        x4 = self.encoder4(self.pool(x3)) # 32*32*256 -> 32*32*512
        x5 = self.encoder5(self.pool(x4)) # bottleneck 16*16*512 -> 16*16*1024

        # Decoder
        x = self.decoder1(x5, x4) # 16*16*1024 -> 32*32*512
        x = self.decoder2(x, x3) # 32*32*512 -> 64*64*256
        x = self.decoder3(x, x2) # 64*64*256 -> 128*128*128
        x = self.decoder4(x, x1) # 128*128*128 -> 256*256*64

        # Output
        return self.output(x) # 256*256*64 -> 256*256*1
```

V. ResNet34+UNet



使用 ResNet34 作為 UNet 的 encoder，其中每層都有 residual block（2 層 conv + BN + ReLU），搭配 shortcut 連接，但要注意在 concatenate 時通道數不同時需要調整，透過一個 1*1 的 conv 調整通道數，ResNet34 有設計好每層需要幾個 residual block（3,4,6,3），且每層的 output 都要存下來給 UNet decode skip connection 用。

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, shortcut=None):
        super(ResidualBlock, self).__init__()

        self.basic = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(out_channels)
        )
        self.shortcut = shortcut
        self.relu = nn.ReLU(inplace=True)
```

```python
class ResNetEncoder(nn.Module):
    def __init__(self, in_channels):
        super(ResNetEncoder, self).__init__()

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

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

    def make_layer(self, in_channels, out_channels, blocks, stride=1):
        shortcut = None
        if stride != 1 or in_channels != out_channels:
            shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
                nn.BatchNorm2d(out_channels)
            )
        layers = [ResidualBlock(in_channels, out_channels, stride, shortcut)]
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x1 = self.encoder(x)      # 256*256*3 -> 64*64*64
        x2 = self.layer1(x1)      # 64*64*64 -> 64*64*64
        x3 = self.layer2(x2)      # 64*64*64 -> 32*32*128
        x4 = self.layer3(x3)      # 32*32*128 -> 16*16*256
        x5 = self.layer4(x4)      # 16*16*256 -> 8*8*512
        return x1, x2, x3, x4, x5
```

```python
class UNetDecoder(nn.Module):
    def __init__(self, in_channels, skip_channels, out_channels):
        super(UNetDecoder, self).__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.decoder = nn.Sequential(
            nn.Conv2d(out_channels + skip_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip_connection):
        x = self.upconv(x)
        # resnet output size is different from unet, so we need to interpolate
        if skip_connection is not None:
            if x.shape[2:] != skip_connection.shape[2:]:
                skip_connection = F.interpolate(skip_connection, size=x.shape[2:], mode='bilinear', align_corners=False)
            x = torch.cat([x, skip_connection], dim=1)
        return self.decoder(x)
```

上面那個可以看到當 decode 要處理 skip connection 的時候，由於 resnet34 的 size 會和 unet up sample 之後的 size 不同，這邊透過 F.interpolate( ) 插值成一樣的大小再 concatenate・

```
class ResNet34_Unet(nn.Module):
    def __init__(self, in_channels, out_channels=1):
        super(ResNet34_Unet, self).__init__()

        # Encoder
        self.encoder = ResNetEncoder(in_channels)

        # Decoder
        self.decoder1 = UNetDecoder(512, 256, 256) # 8*8*512 -> 16*16*256
        self.decoder2 = UNetDecoder(256, 128, 128) # 16*16*256 -> 32*32*128
        self.decoder3 = UNetDecoder(128, 64, 64) # 32*32*128 -> 64*64*64
        self.decoder4 = UNetDecoder(64, 64, 64) # 64*64*64 -> 128*128*64
        self.decoder5 = UNetDecoder(64, 0, 64) # 128*128*64 -> 256*256*64

        # Output
        self.output = nn.Conv2d(64, out_channels, kernel_size=1) # 256*256*64 -> 256*256*1

    def forward(self, x):
        # x1: 64*64*64, x2: 64*64*64, x3: 32*32*128, x4: 16*16*256, x5: 8*8*512
        x1, x2, x3, x4, x5 = self.encoder(x)

        x = self.decoder1(x5, x4) # 8*8*512 -> 16*16*256
        x = self.decoder2(x, x3) # 16*16*256 -> 32*32*128
        x = self.decoder3(x, x2) # 32*32*128 -> 64*64*64
        x = self.decoder4(x, x1) # 64*64*64 -> 128*128*64
        x = self.decoder5(x, None) # 128*128*64 -> 256*256*64

        x = self.output(x) # 256*256*64 -> 256*256*1

        return x
```

VI. Diss Score & Dice loss

BCE loss : Binary Cross Entropy Loss，在此次任務中為了區分物件與背景，將 pixel 分成 0 跟 1，這類二分類問題適合使用 BCE loss

$$\text{BCE} = -\frac{1}{N}\sum_{i}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

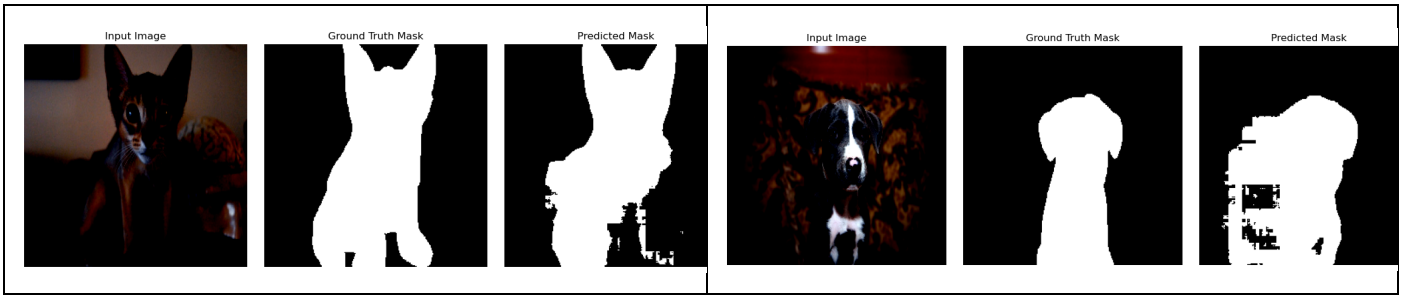
一開始我只用 BCE loss 當作我的 loss function，但我發現這樣的 loss 下降不代表 Dice score 表現會好，後來變改用 BCE loss + Dice loss 並藉由 scale weight 去調整 loss 的比重，這邊因為最後的評分結果是由 Dice score 決定，因次我設定 loss = 0.3 BCE loss + 0.7 Dice loss・

且此處的 dice loss = 1 – dice score

```
def dice_score(pred_mask, gt_mask, threshold=0.5):
    pred_mask = torch.sigmoid(pred_mask)
    pred_mask = (pred_mask > threshold).float()

    intersection = torch.sum(pred_mask * gt_mask, dim=[2, 3])
    union = torch.sum(pred_mask, dim=[2, 3]) + torch.sum(gt_mask, dim=[2, 3])

    dice = (2.0 * intersection + 1e-6) / (union + 1e-6)
    return dice.mean()
```

2. Data PreProcessing

使用 albumentations 套件實作 Data augmentation，同時對 image 跟 mask 做標準化、水平垂直翻轉、旋轉 90 度、隨機改變亮度跟對比度，最後統一調整 size 成 256*256・透過這樣的操作可以使模型有更強的適應力・



觀察資料發現有些 test data 他的亮度特別低，這樣亮度不均得資料如果能在 train 之前先使用 gamma 校正後再訓練，並且在 data augmentation 中也有隨機提升亮度得功能，可以提升訓練效果，下面結果可以顯示提升效果，但也會有仍然無法處理得 task，如右圖，推測是背景與物件相似性太高，

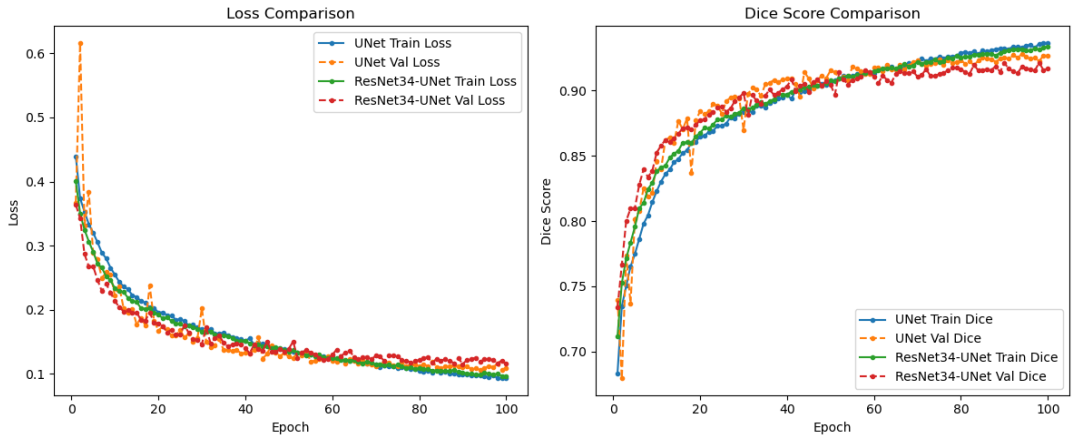

```
# apply gamma correction
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
brightness = np.mean(gray)

if brightness < self.gamma_threshold:
    image = self.apply_gamma(image, gamma=self.gamma_value)
```

```
def apply_gamma(self, image, gamma=1.5):
    table = np.array([((i / 255.0) ** (1.0 / gamma)) * 255 for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(image, table)
```

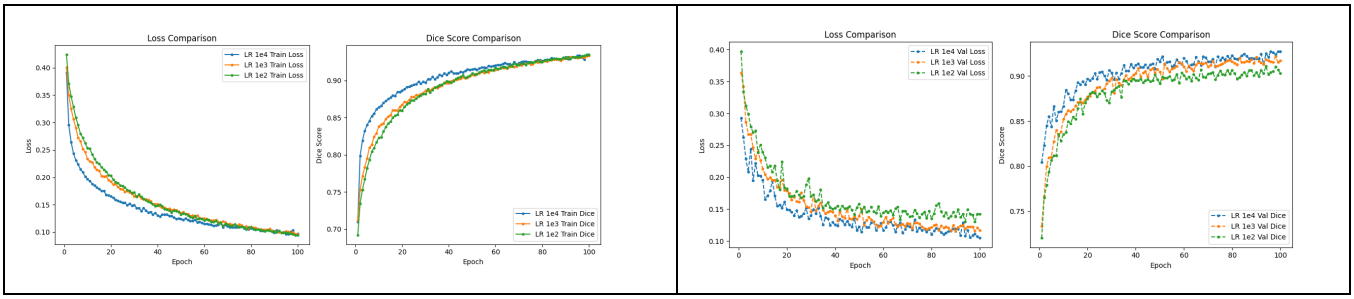
```
def load_dataset(data_path, mode, args):
    assert mode in {"train", "valid", "test"}

    # implement the load dataset function here
    train_t = A.Compose([
        A.Normalize(),
        A.HorizontalFlip(p=0.25),  # apply horizontal flip to 50% of images
        A.VerticalFlip(p=0.25),  # apply vertical flip to 50% of images
        A.RandomRotate90(p=0.25),  # apply random rotation to 50% of images
        A.RandomBrightnessContrast(p=0.25),  # apply random brightness and contrast
        A.Resize(256, 256),
        ToTensorV2(),
    ], additional_targets={"trimap": "mask"})
    vaild_t = A.Compose([
        A.Normalize(),
        A.Resize(256, 256),
        ToTensorV2(),
    ], additional_targets={"trimap": "mask"})

    transform = train_t if mode == "train" else vaild_t


    dataset = OxfordPetDataset(root=data_path, mode=mode, transform=transform)
```

3. Analyze the experiment results

Hyperparameter settings : Batch size : 16, epochs 100, learing rate 1e-3, optimizer : Adam



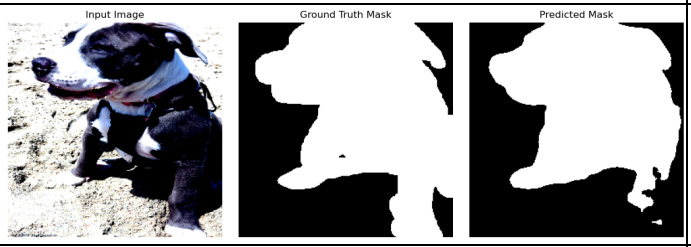
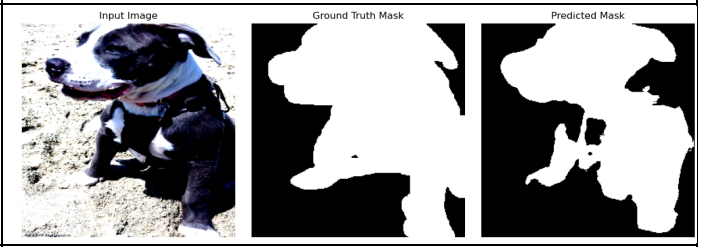
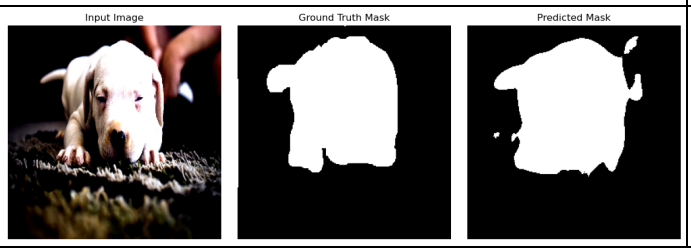
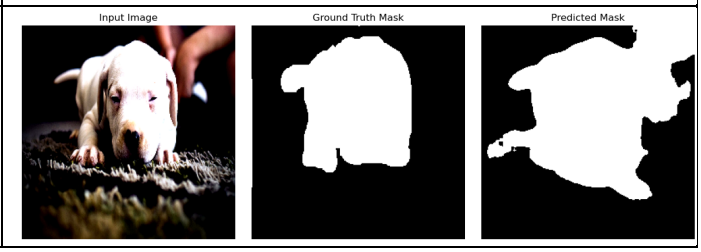
從上圖訓練結果圖顯示，ResNet34-UNet 再前期收斂速度稍微比 UNet 快一點，但整體沒有很大的落差，兩者訓練出來的最終結果也差不多，但在實驗中可以發現 UNet 前期訓練速度較慢．



嘗試不同 learning rate 在 ResNet34-UNet 上，發現 1e4 收斂效率比較好，但其實最後都可以到達一樣的準確率．

LR : 1e3	LR: 1e4
Test – Loss: 0.1136, Dice Score: 0.9249	Test – Loss: 0.1062, Dice Score: 0.9288

♦ 比較兩個 model 可視化結果圖

ResNet34-UNet	UNet
	
	

兩者的最終準確率雖然相差無幾，但是有各自擅長的任務，UNet 傾向抓較小的特徵輪廓，會導致整

體最後結果會比較瑣碎，而 ResNet34-UNet 較喜歡平滑且整體的特徵趨向，但也會導致太複雜的圖會誤判相近的大區域變成一整體．

4. Execution steps

Python 3.9

執行 pip install -r requirement.txt 安裝環境

輸入以下指令開始執行training

- python -u src/train.py --data_path ./dataset --epochs 100 --batch_size 16 --learning-rate 1e-3 --model resnet34_unet
- python -u src/train.py --data_path ./dataset --epochs 100 --batch_size 16 --learning-rate 1e-3 --model unet

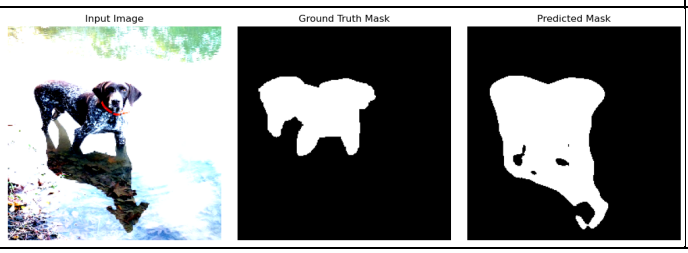
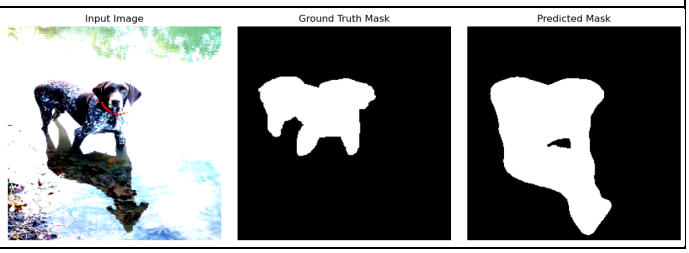
訓練完之後在saved_models底下會有儲存好的best model，再執行以下指令inference

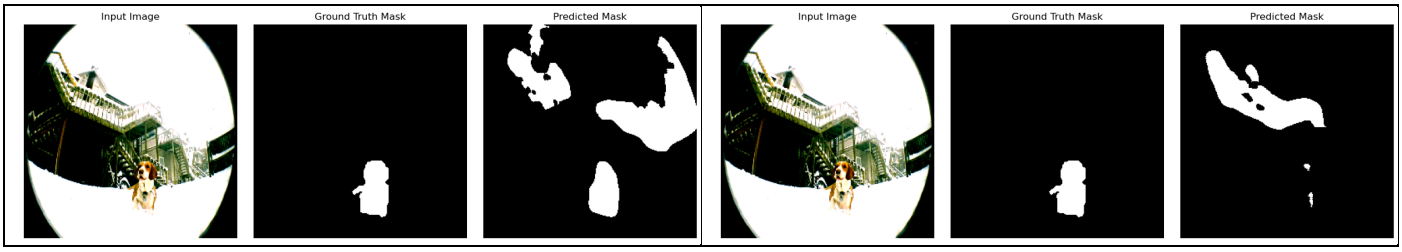
- python src/inference.py --model_path ./saved_models/resnet34_unet_best.pth \
  --model_type resnet34_unet \
  --data_path ./dataset \
  --batch_size 16 \
  --output_dir ./output
- python src/inference.py --model_path ./saved_models/unet_best.pth \
  --model_type unet \
  --data_path ./dataset \
  --batch_size 16 \
  --output_dir ./output

5. Discussion

透過下面這兩個範例可以看出來，當模型對於複雜的環境和影子時，單就這個 data 和 model 架構沒有辦法很好的辨別他要區分的物件跟背景．這有幾種方法可以解決，除了換另一種適合的模型架構之外，也可以給他更多的 data 或是 data augmentation 讓模型的泛化能力更好．或者是也可以考慮串接 pre-trained 好的 model．以這個應用為例，我們可以串接 SAM．

- 串接 SAM 可以先透過 SAM 的 bounding block 找到我們想要 segment 的物件縮小圖像後再丟入 data 進行訓練．

ResNet34-UNet	UNet
	



Sematic segmentation 這項任務適合用在多種其他訓練需求的前處理，以 inpainting 為例，要找到要修補的區域常常也要先找到適合的 mask 再針對 mask 進行修補，這樣可以省下許多不必要的計算與資源的浪費．除此之外，也可以應用在各種需要分割影像的場景上，例如醫療影像分割，自駕車場景分割等等．