

# NYCU DL Lab3 – Binary Semantic Segmentation

Student ID: 313553014 Name: 廖怡誠

## 1. Overview

在 Lab3 的實驗中，需要實作 UNet 與 ResNet34\_UNet 兩個不同的 Binary Semantic Segmentation 模型，並將其訓練於 Oxford-IIIT Pet Dataset 上，所謂的 Semantic Segmentation 是將不同的物件進行分割。以本次的實驗作為舉例，我們需要將圖片分割成前景與背景兩區塊，其中前景所包含的物件為貓與狗，也就是貓與狗屬於同一類，模型不會再分割貓與狗兩個物件，也就是 Binary 的部分，模型的輸出只會介於 0 到 1 之間，1 代表的是前景物件，0 則代表背景，本實驗的目標是透過計算模型預測的分割結果與 Ground Truth 的分割結果之間的 Dice Score 越高越好。

## 2. Implementation Details

### A. Details of my training, evaluating, inferencing code

Training code 與前兩次的作業無太大的差異，順序依序是，讀取訓練與驗證(validation)資料、設置模型、Loss function 與 Optimizer、Forward pass、計算 Loss 與 Dice Score、backward pass、更新權重。其中 Loss function 使用 Binary Cross Entropy(BCE) Loss + Dice Loss，而為了加快訓練收斂的速度 Optimizer 使用 Adam，此外，為了避免梯度爆炸問題，我額外加入 clip gradient 方法，其功能類似 L2 regularization，可以有效控制梯度的變化。最後，我會將 Evaluating Dice Score 最高的該次模型權重儲存，如程式碼圖 1、2 所示。

```
def train(args):
    device = "cuda:0" if torch.cuda.is_available() else "cpu"

    EPOCHS = args.epochs
    BATCH_SIZE = args.batch_size
    LR = args.learning_rate
    NET = args.net
    PRE_TRAIN_MODEL = args.model
    DATA_PATH = args.data_path

    # Data preprocessing
    train_dataset = load_dataset(DATA_PATH, "train")
    val_dataset = load_dataset(DATA_PATH, "valid")
    train_dataloader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True)
    val_dataloader = DataLoader(val_dataset, BATCH_SIZE, shuffle=False)

    model = UNet(3, 1).to(device) if NET == "UNet" else ResNet34_UNet(3, 1).to(device)
    criterion = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=LR)

    prev_epochs = 0
    if PRE_TRAIN_MODEL is not None:
        checkpoint = torch.load(PRE_TRAIN_MODEL)
        model.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        prev_epochs = checkpoint['epoch']

    EPOCHS += prev_epochs

    best_dice_score = 0
    history = {"train_loss": [], "train_dice_score": [], "val_loss": [], "val_dice_score": []}
```

圖 1. Training 程式碼

```

for epoch in range(prev_epochs + 1, EPOCHS + 1):
    model.train()
    batch_train_loss = []
    batch_train_dice_score = []

    for batch in tqdm(train_dataloader):
        imgs, masks = batch["image"].to(device), batch["mask"].to(device)
        # batch, 1, W, H -> batch, W, H

        masks_pred = model(imgs).squeeze(1)
        loss = criterion(masks_pred, masks.float()) + dice_loss(masks_pred, masks.float())

        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        dice_score = cal_dice_score(masks_pred, masks.float())

        batch_train_dice_score.append(dice_score)
        batch_train_loss.append(loss.item())

    train_dice_score = sum(batch_train_dice_score) / len(batch_train_dice_score)
    train_loss = sum(batch_train_loss) / len(batch_train_loss)
    history["train_dice_score"].append(train_dice_score)
    history["train_loss"].append(train_loss)

    val_dice_score, val_loss = evaluate(model, val_dataloader, device)
    history["val_dice_score"].append(val_dice_score)
    history["val_loss"].append(val_loss)

    print(f"[ Train | {epoch:03d}/{EPOCHS:03d} ] loss = {train_loss:.5f}, acc = {train_dice_score:.5f}")

    if val_dice_score > best_dice_score:
        print(f"Best model found at epoch {epoch}, saving model")
        torch.save({'model_state_dict': model.state_dict(),
                    'epoch': epoch,
                    'optimizer_state_dict': optimizer.state_dict(),
                    }, './best.pth')
        best_dice_score = val_dice_score
    draw_history(history)
# draw model history
draw_history(history, True)

```

圖 2. Training 程式碼

Evaluating code 與 Training code 的差異不大，主要將模型切換成 evaluation 模式，確保模型的輸出穩定且有一致性，因為如 drop out、batch normalization 等方法，在 training 期間會對資料進行額外的處理，而這些處理在 evaluation 階段是不需要的。此外，使用 torch.no\_grad() 關閉梯度計算，節省計算資源並提高運行效率。

```

def evaluate(net, val_dataloader, device):
    criterion = nn.BCELoss()
    batch_val_loss = []
    batch_val_dice_score = []

    with torch.no_grad():
        net.eval()
        for batch in tqdm(val_dataloader):
            imgs, masks = batch["image"].to(device), batch["mask"].to(device)
            masks_pred = net(imgs.to(device)).squeeze(1)
            val_loss = criterion(masks_pred, masks.float()) + dice_loss(masks_pred, masks.float())
            dice_score = cal_dice_score(masks_pred, masks.float())

            batch_val_dice_score.append(dice_score)
            batch_val_loss.append(val_loss.item())
        val_dice_score = sum(batch_val_dice_score) / len(batch_val_dice_score)
        val_loss = sum(batch_val_loss) / len(batch_val_loss)
    return val_dice_score, val_loss

```

圖 3. Evaluation 程式碼

Inferencing code 基本上與 Evaluating code 相同，只差在需要將訓練好的模型權重讀取到模型中，以及我會觀察模型的輸出與 Ground Truth 的差異。

```
def inference(args):
    device = "cuda:0" if torch.cuda.is_available() else "cpu"

    NET = args.net
    DATA_PATH = args.data_path
    BATCH_SIZE = args.batch_size
    PRE_TRAIN_MODEL = args.model

    test_dataset = load_dataset(DATA_PATH, mode="test")
    test_dataloader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False)

    if NET == "UNet":
        model = UNet(3, 1).to(device)
    elif NET == "ResNet34_UNet":
        model = ResNet34_UNet(3, 1).to(device)

    criterion = nn.BCELoss()
    if PRE_TRAIN_MODEL is not None:
        checkpoint = torch.load(PRE_TRAIN_MODEL)
        model.load_state_dict(checkpoint['model_state_dict'])

    model.eval()

    batch_test_loss = []
    batch_test_dice_score = []
    with torch.no_grad():
        for batch in tqdm(test_dataloader):
            imgs, masks = batch["image"].to(device), batch["mask"].to(device)
            masks_pred = model(imgs.to(device)).squeeze(1)

            loss = criterion(masks_pred, masks.float()) + dice_loss(masks_pred, masks.float())
            dice_score = cal_dice_score(masks_pred, masks.float())

            # Show image
            # for i in range(len(masks)):
            #     show_img(masks[i], np.where(masks_pred.cpu().detach().numpy()[i] > 0.5, 1, 0))

            batch_test_dice_score.append(dice_score)
            batch_test_loss.append(loss.item())
        test_dice_score = sum(batch_test_dice_score) / len(batch_test_dice_score)
        test_loss = sum(batch_test_loss) / len(batch_test_loss)

    print(f"[ Test ] loss = {test_loss:.5f}, acc = {test_dice_score:.5f}")
```

圖 4. Inferencing 程式碼

## B. Details of my model (UNet & ResNet34\_UNet)

UNet 是 Encoder-Decoder 的架構，可以分成 contracting path(Encoder)與 expansive path (Decoder)，contracting path 對應到 Down Sampling，其負責截取圖片特徵，而 expansive path 則是對應到 Up Sampling，其負責圖片 localization，也就是其會提高經過 contracting path 擷取特徵的圖片解析度 (Resolution)，除此之外，還加入 skipping connection 的技術，通常將高解析轉換成低解析可以保留圖像細節 (Encoder)，而低解析轉換成高解析則會失真，通過 skipping 將含有圖像細節的圖片加入到 decoder 的過程中，即可彌補 Up sampling 失去的圖像細節。

而在我的 UNet 的模型架構實作中，主要分成 Down Sampling、Up Sampling 與 Double Convolution 三個部分，首先，Double Convolution 是有兩個 Convolution 組成的 block，如圖 5.所示，將 max pooling 與結合 Double Conv 即為 Down Sampling，如圖 6.所示；反之 Conv Transpose 又稱為 de-

convolution 即為 Up Sampling，在這個 class 中，在通過 Conv Transpose 之後，會接著做 skipping connection 的圖片合併，如圖 7.所示。

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

        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size= 3, padding= 1, bias= False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace= True),
            nn.Conv2d(out_channels, out_channels, kernel_size= 3, padding= 1, bias= False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace= True),
        )

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

圖 5. Double conv 程式碼

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

        self.down_sampling = nn.Sequential(
            nn.MaxPool2d(kernel_size= 2, stride= 2),
            DoubleConv(in_channels, out_channels)
        )

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

圖 6. Down Sampling 程式碼

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

        self.de_conv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size= 2, stride= 2)
        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x, cropped_x):
        x = self.de_conv(x)
        x = torch.cat([cropped_x, x], dim= 1)
        return self.conv(x)
```

圖 7. Up Sampling 程式碼

我的 UNet 架構會將前面介紹的 Double conv、Up Sampling、Down Sampling 組合起來，其是由四個 Down Sampling 與四個 Up Sampling 組合而成，Down Sampling 的輸出會 skipping connection 到 Up Sampling 中組合，在模型的最後加入 Sigmoid 將輸出限制在[0, 1]，且輸出的大小應與輸入的大小相同 ( $H * W$ )，完整的輸出為( $B, C, W, H$ )，因為是 binary 所以  $C$  為 1，而  $W * H$  中的每個 pixel 分別對應到其所屬類別的機率。

```

class UNet(nn.Module):
    def __init__(self, num_channels, num_classes):
        super(UNet, self).__init__()

        self.contracting1 = DoubleConv(num_channels, 64)
        self.contracting2 = DownSampling(64, 128)
        self.contracting3 = DownSampling(128, 256)
        self.contracting4 = DownSampling(256, 512)
        # connect between contracting path and expansive path
        self.contracting5 = DownSampling(512, 1024)
        self.expansive1 = UpSampling(1024, 512)
        self.expansive2 = UpSampling(512, 256)
        self.expansive3 = UpSampling(256, 128)
        self.expansive4 = UpSampling(128, 64)
        self.output = nn.Sequential(
            nn.Conv2d(64, num_classes, kernel_size=1),
            nn.Sigmoid()
        )

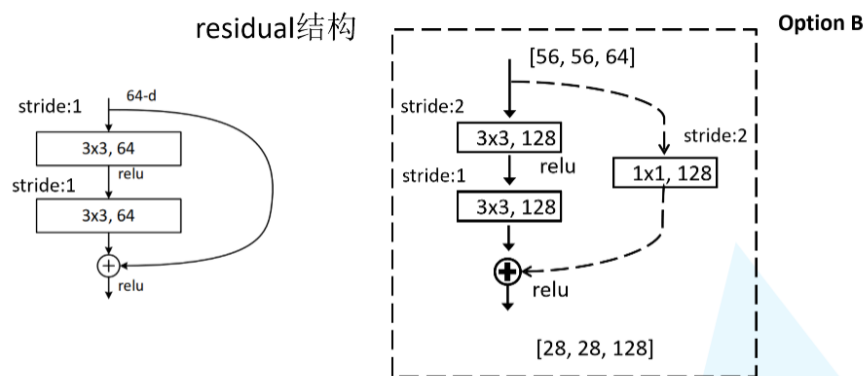
    def forward(self, x):
        c1 = self.contracting1(x)
        c2 = self.contracting2(c1)
        c3 = self.contracting3(c2)
        c4 = self.contracting4(c3)
        c5 = self.contracting5(c4)
        x = self.expansive1(c5, c4)
        x = self.expansive2(x, c3)
        x = self.expansive3(x, c2)
        x = self.expansive4(x, c1)

        return self.output(x)

```

圖 8. UNet Architecture 程式碼

ResNet34\_UNet 是將 Residual Network 結合到 UNet 的架構中，Residual Net 作為 Encoder，UNet 作為 Decoder。Residual block 的設計解決了資訊在過深的網路架構中會無法有效往下傳遞的退化問題 (degradation problem)。Residual block 可以分成兩個部分，如圖 9 所示，左邊的主分支由兩層 3\*3 的 convolution 組成，右邊的連接線稱為 shortcut 分支，而 shortcut 又有兩種操作方式，如果輸入與輸出的維度相同時，會使用左圖的 identity shortcuts，也就是直接傳遞下去；如果輸出的維度增加時，會使用右圖中的方法，將輸入通過一層 1\*1 的 convolution 降維度。



注意：主分支与shortcut的输出特征矩阵shape必须相同

圖 9. Residual block 示意圖

在我的實作中，我按照圖 9 示意圖中的架構設計，由於會有兩種情況，因此在主分支的第一層 convolution 會以 stride 的變數控制，shortcut 也會根據輸入與輸出維度是否相同來調整。

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

        self.residual_conv = 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),
        )
        self.ac = nn.ReLU()
        self.shortcut = shortcut

    def forward(self, x):
        out = self.residual_conv(x)
        residual = x if self.shortcut is None else self.shortcut(x)
        out += residual
        return self.ac(out)
```

圖 10. Residual block 程式碼

ResNet34\_UNet 的架構圖如圖 11 所示，因為 Residual net 的架構也能夠實現 Down Sampling 的功能來萃取圖片的特徵，因此 Encoder 的部分使用 ResNet34 的架構是可行的，其中共有五層的 Layer，在除了第一層 Layer 之外的四層都會在層與層之間都會進行圖 9.右邊的操作，即加入一層 1\*1 的 convolution。而 Decoder 的部分，則使用原本 UNet 中的 Double convolution 架構，與 Encoder 相對應，所以一樣有五層的 Layer，將 resnet34\_conv5 作為中間的 bottleneck 層，並將每次 Up Sampling 的輸出依序與 resnet34\_conv4、resnet34\_conv3、resnet34\_conv2、resnet34\_conv1 結合，即為 skipping connection 的操作，最後按照 paper 上的架構，在完成五層 Up Sampling 之後，會在通過一層 1 \* 1 的 convolution 將維度降到 1。

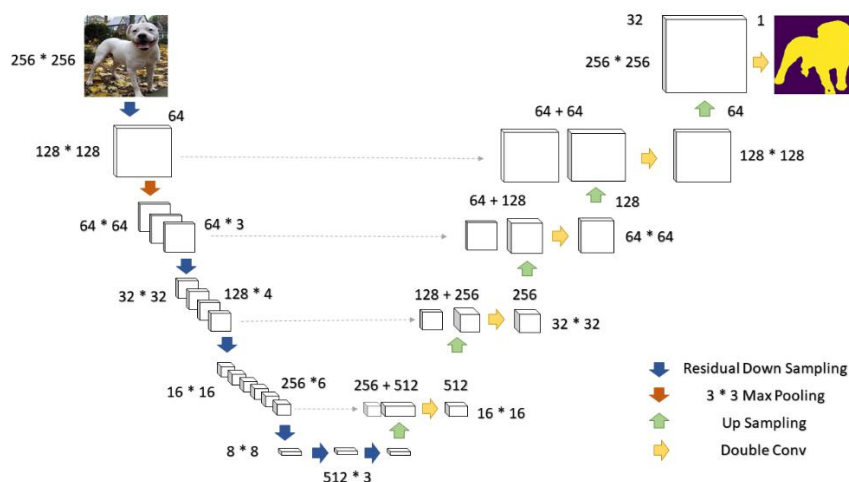


圖 11. ResNet34\_UNet 簡易架構圖

```

class ResNet34_UNet(nn.Module):
    def __init__(self, num_channels, num_classes):
        super(ResNet34_UNet, self).__init__()

        # encoder
        self.resnet34_conv1 = nn.Sequential(
            nn.Conv2d(num_channels, 64, kernel_size= 7, stride= 2, padding= 3, bias= False),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        shortcut2 = nn.Sequential(
            nn.Conv2d(in_channels= 64, out_channels= 64, kernel_size= 1, stride= 1, bias= False),
            nn.BatchNorm2d(64)
        )
        self.resnet34_conv2 = nn.Sequential(
            nn.MaxPool2d(kernel_size= 3, stride= 2, padding= 1),
            Residual_Block(in_channels= 64, out_channels= 64, stride= 1, shortcut= shortcut2),
            *[Residual_Block(in_channels= 64, out_channels= 64) for _ in range(2)]
        )
        shortcut3 = nn.Sequential(
            nn.Conv2d(in_channels= 64, out_channels= 128, kernel_size= 1, stride= 2, bias= False),
            nn.BatchNorm2d(128)
        )
        self.resnet34_conv3 = nn.Sequential(
            Residual_Block(in_channels= 64, out_channels= 128, stride= 2, shortcut= shortcut3),
            *[Residual_Block(in_channels= 128, out_channels= 128) for _ in range(3)]
        )
        shortcut4 = nn.Sequential(
            nn.Conv2d(in_channels= 128, out_channels= 256, kernel_size= 1, stride= 2, bias= False),
            nn.BatchNorm2d(256)
        )
        self.resnet34_conv4 = nn.Sequential(
            Residual_Block(in_channels= 128, out_channels= 256, stride= 2, shortcut= shortcut4),
            *[Residual_Block(in_channels= 256, out_channels= 256) for _ in range(5)]
        )
        shortcut5 = nn.Sequential(
            nn.Conv2d(in_channels= 256, out_channels= 512, kernel_size= 1, stride= 2, bias= False),
            nn.BatchNorm2d(512)
        )
        self.resnet34_conv5 = nn.Sequential(
            Residual_Block(in_channels= 256, out_channels= 512, stride= 2, shortcut= shortcut5),
            *[Residual_Block(in_channels= 512, out_channels= 512) for _ in range(2)]
        )

        # decoder
        self.unet_up_conv1 = UpSampling(512, 512)
        # cropped + expansive_path
        self.unet_double_conv1 = DoubleConv(512 + 256, 512)
        self.unet_up_conv2 = UpSampling(512, 256)
        self.unet_double_conv2 = DoubleConv(256 + 128, 256)
        self.unet_up_conv3 = UpSampling(256, 128)
        self.unet_double_conv3 = DoubleConv(128 + 64, 128)
        self.unet_up_conv4 = UpSampling(128, 64)
        self.unet_double_conv4 = DoubleConv(64 + 64, 64)
        self.unet_up_conv5 = UpSampling(64, 32)
        self.output = nn.Sequential(
            nn.Conv2d(32, num_classes, kernel_size= 1),
            nn.Sigmoid()
        )

    def forward(self, x):

        # encoder
        e1 = self.resnet34_conv1(x)
        e2 = self.resnet34_conv2(e1)
        e3 = self.resnet34_conv3(e2)
        e4 = self.resnet34_conv4(e3)
        e5 = self.resnet34_conv5(e4)

        # decoder
        d1 = torch.cat([self.unet_up_conv1(e5), e4], dim= 1)
        x = self.unet_double_conv1(d1)
        d2 = torch.cat([self.unet_up_conv2(x), e3], dim= 1)
        x = self.unet_double_conv2(d2)
        d3 = torch.cat([self.unet_up_conv3(x), e2], dim= 1)
        x = self.unet_double_conv3(d3)
        d4 = torch.cat([self.unet_up_conv4(x), e1], dim= 1)
        x = self.unet_double_conv4(d4)
        d5 = self.unet_up_conv5(x)
        x = self.output(d5)

        return x

```

圖 12. ResNet34\_UNet 程式碼



### 3. Data Preprocessing

#### A. How my preprocessed my data?

在 Data preprocessing 中，我使用套件 albumentations 對資料進行 data augmentation，由於助教預留的 code 有 self.transform(\*\*sample)，才學習到這個套件的使用，與 torch 自己的 transform 不同，albumentations 其可以一次對 image 與 mask 轉換，這樣可以避免 image 與 mask 進行含隨機性轉換，所發生的相同圖片但轉換結果不同的問題，簡單來說，不會出現 image 旋轉 30 度，mask 旋轉-30 度的錯誤情況，此外，也可以省略分開對 image 和 mask 轉換的程式碼。而在這次的實驗中，我對 Data 進行了以下的圖像處理：Resize 256 \* 256、隨機水平左右翻轉、隨機平移與旋轉、亮度調整與加速收斂的標準化。而旋轉的角度限制在物件合理的正負 30 度之內，基於相同的原因也沒有使用垂直翻轉。

```
train_transforms = A.Compose([
    A.Resize(256, 256),
    A.HorizontalFlip(),
    A.ShiftScaleRotate(shift_limit= 0.2 , scale_limit= 0.2 , rotate_limit= 15) ,
    A.ColorJitter(hue= 0) ,
    A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
])
test_transforms = A.Compose([
    A.Resize(256, 256),
    A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
])
```

圖 13. Data augmentation 程式碼

#### B. What makes my method unique?

在 Data augmentation 中，最特別的是使用 Normalization，其也是加速模型收斂的重要方法。除此之外，在觀察 dataset 之後，可以發現圖像基本上都是平移與小幅度旋轉，因此通過 ShiftScaleRotate 可以讓模型更容易學習，最後，我也同時對 mask 進行相同的處理，避免模型在訓練對不上的問題，通過這些方法，使模型提升泛化的能力，也減少 overfitting 的問題。

### 4. Analyze on the experiment results

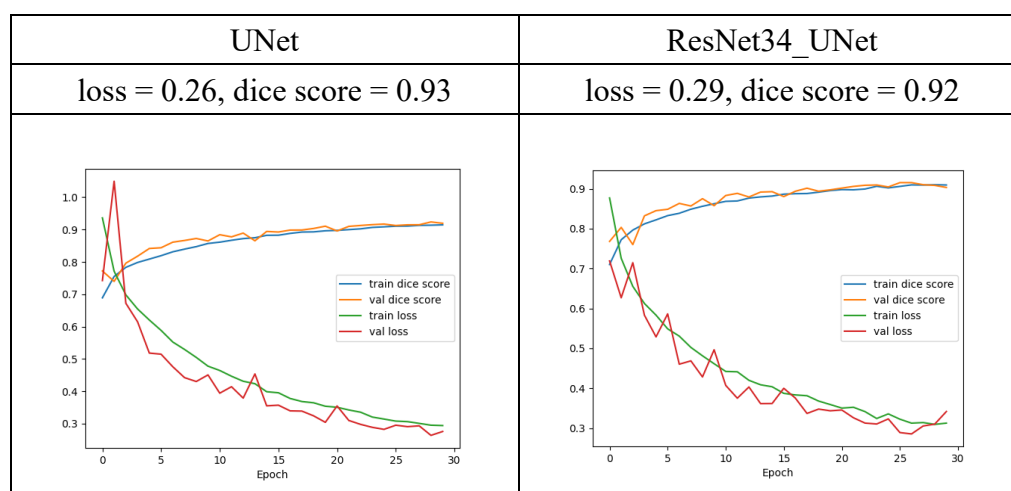
以下為我本次實驗所使用的 hyperparameters:

- Batch Size: 8
- Loss Function: Binary Cross Entropy Loss + Dice Loss
- Optimizer: Adam (learning rate: 1e-3)
- Epoch: 30

















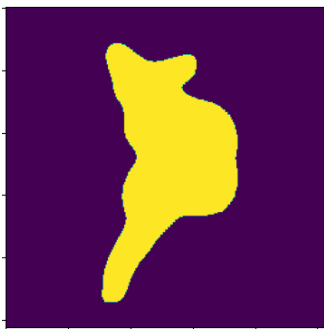

## A. What did me explore during the training process?

下表是本次實驗的結果，在 testing data 中，UNet 的 loss 為 0.26、dice score 為 0.93，ResNet34\_UNet 的 loss 為 0.29、dice score 為 0.92。在訓練的過程中，我觀察到 ResNet34\_UNet 在前期 dice score 提高的速度相比 UNet 快上許多，而在接近後期時，兩個模型的 dice score 會差不多，代表說 ResNet34\_UNet 泛化的能力比較好。此外，在實驗的過程中，原先我有按照 UNet 的論文描述，將 batch size 設定為 1 進行訓練，但是在相同的 epoch 之下與其他 batch size 相比，dice score 並沒有比較好，且提高的幅度較小，我認為是 batch size 為 1 時，因為每 1 張 image 就是一個 batch，代表權重更新次數是最多的，這會使得模型在前期的變化波動較大，因為每張圖片的特徵都不太一樣；相反地，batch size 越大，就會是多張圖片取平均再更新權重，這會使得模型的變化較為平滑，兩者各有好壞，batch size 較小訓練速度偏慢，但泛化能力最好；batch size 較大則是相反，但因為資源與時間的限制下，將 batch size 設定為 8，原先需要跑 100 epochs 才會到 90%，提升到只需大約 30 epochs。



## B. Found any characteristics of the data?

我從 inferencing data 中挑出兩張效果不好與兩張效果較好的圖片進行分析，從第一、二列的圖片可以觀察到這兩張圖的 ground truth mask 較複雜，且 original image 前景的顏色也與背景較為相近，第一張圖片的灰貓在背景偏灰的地方，第二章的白色小狗的背景則是偏白色。反之，第三、四列的圖片前景與背景顏色區分較明顯且 mask 的複雜度也較低，因此模型得到的結果也都較好。總而言之，data 的背景與前景顏色過於相近時，模型效果較差，反之亦然。

Org image	Ground Truth	ResNet34_UNet	UNet
			
			
			
			

## 5. Execution command

### A. The command and parameters for the training process

在 training 的程式碼中，除了 TA 預留的參數之外，我還額外加入兩個 net 與 model 的參數，net 的功能是選擇目前的訓練要使用 UNet 還是

ResNet34\_UNet 架構，而 model 則是當需要繼續訓練模型時，會輸入的模型權重路徑。我使用的指令為：

```
python src/train.py --data_path ./dataset/oxford-iiit-pet --epoch 100 -b 4
```

```
def get_args():
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')
    parser.add_argument('--data_path', type=str, help='path of the input data')
    parser.add_argument('--epochs', '-e', type=int, default=5, help='number of epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=1, help='batch size')
    parser.add_argument('--learning-rate', '-lr', type=float, default=1e-3, help='learning rate')
    parser.add_argument('--net', '-n', type=str, default="UNet")
    parser.add_argument('--model', default=None, help='path to the stored model weight')

    return parser.parse_args()
```

圖 14. Training 傳入的參數

## B. The command and parameters for the inference process

在 inference 的程式碼中，其使用的參數與 training 時的大同小異，主要是輸入模型權重的位置。

```
python src/inference.py --data_path ./dataset/oxford-iiit-pet -b 4 --model best.pth
```

```
def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model', default='MODEL.pth', help='path to the stored model weight')
    parser.add_argument('--data_path', type=str, help='path to the input data')
    parser.add_argument('--batch_size', '-b', type=int, default=1, help='batch size')
    parser.add_argument('--net', '-n', type=str, default="UNet")

    return parser.parse_args()
```

圖 15. Inferencing 傳入的參數

## 6. Discussion

### A. What architecture may bring better results?

第一個方法，參考了 U-Mamba 這篇論文的想法(圖 16)，將 Mamba 架構加入到 Double Conv 的架構之中，透過 Mamba 的特性可以更好的獲取空間資訊，也能使 Down Sampling 保留更多的圖像細節，結合以上兩個優點就可以使 Up Sampling 得到更精準的分割圖。

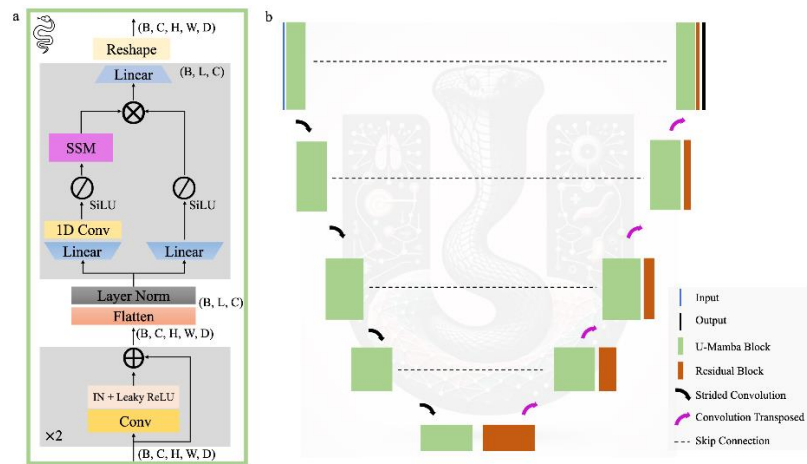
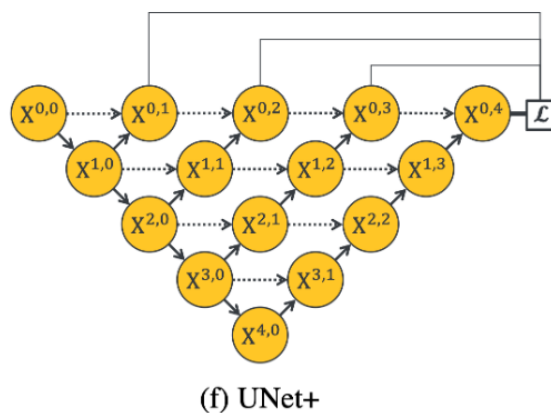


圖 16. U-Mamba 架構圖

第二個想法，參考了 UNet++，最初版的 UNet 缺點是沒辦法知道最合適的深度為合，以及 skipping connection 被限制只能與同層級的特徵圖串接在一起，這會影響到 UNet 的彈性與泛化能力，因此 UNet++ 這篇論文使用叢集式的訓練方法，將不同深度的 UNet 疊在一起訓練，讓彼此之間的特徵圖可以互相使用，並重新設計 skipping connection 的连接方法，使模型能夠有效的特徵萃取。



(f) UNet++

圖 17. UNet++ 架構圖

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

Segmentation 這個 task 主要應用在醫學領域方面，透過 segmentation 技術分割出不同的細胞或器官，並對其進行診斷，然而，目前的問題是人體的器官和細胞太多樣性，以目前 segmentation model 的能力來說，在分割的效果上不全然都是優異的，因此我認為可以如同前面的 U-Mamba 一樣對架構進行修改，來提高模型泛化與特徵萃取的能力，例如使用 wavelet transform 來獲取圖片的高低頻率，使模型可以更好的學習特徵。

## 7. Reference

- UNet architecture
  - [https://github.com/milesial/Pytorch-UNet/blob/master/unet/unet\\_parts.py](https://github.com/milesial/Pytorch-UNet/blob/master/unet/unet_parts.py)
- Residual network 、 ResNet34 \_ UNet architecture:
  - <https://meetonfriday.com/posts/fb19d450/>
  - [https://blog.csdn.net/weixin\\_44350337/article/details/115474009](https://blog.csdn.net/weixin_44350337/article/details/115474009)
  - [https://github.com/zhoudaxia233/PyTorch-Unet/blob/master/resnet\\_unet.py](https://github.com/zhoudaxia233/PyTorch-Unet/blob/master/resnet_unet.py)
  - [https://pytorch.org/vision/main/\\_modules/torchvision/models/resnet.html#resnet34](https://pytorch.org/vision/main/_modules/torchvision/models/resnet.html#resnet34)
- Data Augmentation
  - <https://ithelp.ithome.com.tw/articles/10296027>
  - <https://blog.csdn.net/u014264373/article/details/114144303>
  - <https://github.com/hank891008/Deep-Learning/tree/main/Lab3>
- UNet++:
  - <https://arxiv.org/abs/1807.10165>
- U-Mamba
  - <https://arxiv.org/abs/2401.04722>