110550161_張維程_Lab3_Report

1. Implementation Details (30%)

1.1 Training

整個training 的流程大致上和其他task的訓練過程相同。
先宣告要使用的模型(UNet / ResNet34_UNet), 以及hyperparameters(epoch/lr),
loss function 這次選擇使用了Binary Cross Entropy loss , 並且optimizer 使用Adam 。
模型部分處理完之後, 便開始load dataset (有做data augmentation), 進行訓練(forward -> calculate loss -> backward -> update), 訓練過程中每隔5個epoch 我就會跑一次test dataset ,觀察dice score 的狀況, 並且只儲存dice score 最高的model。另外如果連續15個epoch的dice score 都沒有提升, 那麼訓練就會自動結束, 所以在訓練的過程中, 我的epoch 通常都開的比較大, 這樣也能夠節省調整epoch的時間。

```python
def train(args):
    # Set device
    device = torch.device(f'cuda:{args.gpu}' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")
    train_transform = A.Compose(
        [
            A.Resize(256, 256),
            A.HorizontalFlip(p=0.5),
            A.RandomResizedCrop(size=(256, 256), scale=(0.8, 1)),
            A.Rotate(limit=30, p=0.5),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2, p=0.5),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ],
    )
    valid_transform = A.Compose(
        [
            A.Resize(256, 256),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ],
    )

    # Load datasets
    train_dataset = load_dataset(args.data_path, mode='train', transform=train_transform)
    valid_dataset = load_dataset(args.data_path, mode='valid', transform=valid_transform)
    test_dataset = load_dataset(args.data_path, mode='test', transform= valid_transform) # valid_transform is same as test_transform
    print(f"Training dataset size: {len(train_dataset)}")
    print(f"Validation dataset size: {len(valid_dataset)}")
    print(f'Test dataset size: {len(test_dataset)}')

    # Create data loaders
    train_dataloader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4, pin_memory=True)
    valid_dataloader = DataLoader(valid_dataset, batch_size=args.batch_size, shuffle=False, num_workers=4, pin_memory=True)
    test_dataloader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False,num_workers=4, pin_memory=True)

    # Load model
    if args.network == 'unet':
        from models.unet import UNet
        model = UNet()
    elif args.network == 'resnet34_unet':
        model = ResNet34_UNet(num_classes=1)
    else:
        raise ValueError(f"Unknown network architecture: {args.network}")

    # Move model to device
    model.to(device)

    # Define loss function and optimizer
    criterion = torch.nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)

    # Initialize GradScaler
    scaler = torch.amp.GradScaler('cuda')

    # Initialize lists to store loss values
    train_losses = []
    valid_losses = []
    best_dice = 0
    best_epoch = 0
    no_improvement_epochs = 0
    model_save_path = f"./../saved_models/best_{args.network}_lr{args.learning_rate:.5f}_epoch{best_epoch}.pth"
    saved = False
```

```python
    # Training loop
    for epoch in range(args.epochs):
        train_loss = 0
        print(f"Epoch {epoch + 1}/{args.epochs}")

        # Training phase
        model.train()
        for batch in tqdm(train_dataloader, desc="Training", leave=False):
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            optimizer.zero_grad()

            # Forward pass and loss computation within autocast context
            with torch.amp.autocast('cuda'):#torch.cuda.amp.autocast():
                outputs = model(images)
                loss = criterion(outputs, masks)

            # Backward pass and optimization within scaler context
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

            train_loss += loss.item()

        # Validation phase
        avg_valid_loss, _, _ = evaluate(model, valid_dataloader, criterion, device)

        # Compute average losses
        avg_train_loss = train_loss / len(train_dataloader)
        train_losses.append(avg_train_loss)
        valid_losses.append(avg_valid_loss)

        print(f"Epoch {epoch + 1}/{args.epochs}, Training Loss: {avg_train_loss}, Validation Loss: {avg_valid_loss}")

        # Check if current epoch's dice score is the best
        if epoch % 5 == 0 :
            tmp_score = test(model, test_dataloader, device)
            print("dice score : ",tmp_score)
            if tmp_score > best_dice:
                if saved ==True and os.path.exists(model_save_path):
                    os.remove(model_save_path)
                best_dice = tmp_score
                best_epoch = epoch
                print(f'Best model saved at epoch {epoch}')
                model_save_path = f"/home/winston/dlp/lab/lab2/saved_models/best_{args.network}_lr{args.learning_rate:.5f}_epoch{best_epoch}.pth"
                torch.save(model.state_dict(), model_save_path)
                no_improvement_epochs = 0
                saved = True
            else:
                no_improvement_epochs += 1

        # Early stopping if no improvement in dice score for 15 consecutive epochs
        if no_improvement_epochs >= 3:
            print(f"No improvement in dice score for 15 consecutive epochs. Stopping training at epoch {epoch + 1}.")
            break

    # Save loss records
    loss_record_path = f"/home/winston/dlp/lab/lab2/saved_models/{args.network}_lr{args.learning_rate:.5f}_loss_record.txt"
    with open(loss_record_path, 'w') as f:
        for epoch, (train_loss, valid_loss) in enumerate(zip(train_losses, valid_losses), 1):
            f.write(f"Epoch {epoch}, Training Loss: {train_loss}, Validation Loss: {valid_loss}\n")
```

Fig1.Training

1.2 Evaluation

Evlauation 的部分大致上和training 相似，只是dataset為Fig.1中第26行的valid dataset，因為evaluation是要給我檢視模型的訓練情況，而不是真的在訓練模型，所以模型不需要做更新Fig.2中的第5行有model.eval()

```python
import torch
from tqdm import tqdm

def evaluate(model, dataloader, criterion, device):
    model.eval()
    valid_loss = 0
    all_preds = []
    all_labels = []

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

            outputs = model(images)
            loss = criterion(outputs, masks)
            valid_loss += loss.item()

            preds = torch.sigmoid(outputs) > 0.5
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(masks.cpu().numpy())

    avg_valid_loss = valid_loss / len(dataloader)
    return avg_valid_loss, all_preds, all_labels
```

Fig2.Evaluation

1.3 Inferencing

和validation 時類似，只不過dataset是跑在testing dataset 而不是validation dataset，同樣不需要計算loss和更新model，只要load model 再把testing dataset 餵進model，並和GT算dice score 即可。

```python
if __name__ == '__main__':
    args = get_args()
    device = torch.device('cuda:6' if torch.cuda.is_available() else 'cpu')
    print("using device:", device)
    # 定義 transform
    transform = A.Compose(
        [
            A.Resize(256, 256),
            ToTensorV2(),
        ],
    )

    # 載入測試資料集
    test_dataset = load_dataset(args.data_path, mode='test', transform=transform)
    # print(f'Test dataset size: {len(test_dataset)}')
    test_dataloader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False,num_workers=4, pin_memory=False)

    # 載入模型
    model = load_model(args.model, device, args.network)

    dice = 0
    model.eval()
    with torch.no_grad() :
        for i, data in enumerate(test_dataloader):
            image = data['image'].to(device).to(torch.float)
            mask = data['mask'].to(device)
            outputs = model(image)

            if i < 5:
                input_img = image[0].cpu()
                label_mask = mask[0].cpu()
                output_mask = outputs[0].cpu()
                show_result(input_img, label_mask, output_mask,i)


            # print(type(outputs))
            # print(type(mask))
            dice += float(dice_score(outputs, mask))
            # if i == 0 :
            #     utils.show_predict(args.network, data['image'][0], data['mask'][0], outputs[0])
    print(args.model)
    print('Dice Score : ', round(dice/len(test_dataloader), 4))

```

Fig3.Inference

下圖為input,gt output 的範例

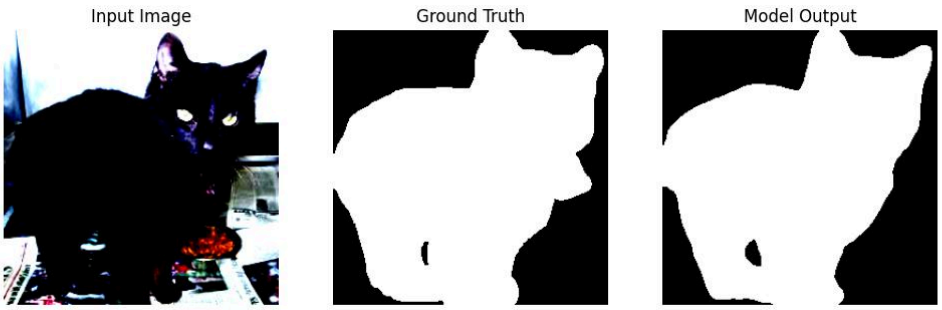


Fig.4 example

1.4.1 Unet detail

實作過程有參考

https://medium.com/analytics-vidhya/unet-implementation-in-pytorch-idiot-developer-da40d955f201

的影片以及內容

UNet 的架構相對簡單，由 Encoder (下採樣)、Bottleneck (橋接層) 和 Decoder (上採樣) 組成。整個網路由 8 個 conv_block (Fig.4) 組成，Encoder 和 Decoder 各有 4 個 block，中間透過 Bottleneck 連接兩端。

Encoder 由 4 層 conv_block 組成(enc1,enc2,enc3,enc4)，負責提取影像特徵，每層之間使用 MaxPooling (stride=2) 來縮小特徵圖尺寸，使網路能夠學習更深層的特徵。

Bottleneck 的目的是提升維度, 輸入512維, 輸出 1024 維的特徵。

Decoder (dec1,dec2,dec3,dec4)負責將 Encoder 中壓縮的特徵圖還原為與輸入相同大小的影像。

處理解析度問題時, 使用bilnear interpolation。最後output前過一層 1x1 的convolution 把output size壓到 out_channels, 用於生成最後的 segmentation mask。

```python
def conv_block(self, in_channels, out_channels):
        return 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)
        )
```

Fig.5 Unet Basic block

```python
# 實作過程有參考 https://medium.com/analytics-vidhya/unet-implementation-in-pytorch-idiot-developer-da40d955f201 的影片以及內容

class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super(UNet, self).__init__()

        # Encoder
        self.enc1 = self.conv_block(in_channels, 64)
        self.enc2 = self.conv_block(64, 128)
        self.enc3 = self.conv_block(128, 256)
        self.enc4 = self.conv_block(256, 512)

        # Bottleneck
        self.bottleneck = self.conv_block(512, 1024)

        # Decoder
        self.dec4 = self.conv_block(1024 + 512, 512)
        self.dec3 = self.conv_block(512 + 256, 256)
        self.dec2 = self.conv_block(256 + 128, 128)
        self.dec1 = self.conv_block(128 + 64, 64)

        # Output
        self.output = nn.Conv2d(64, out_channels, kernel_size=1)
        # self.output = nn.Sequential(nn.Conv2d(64, out_channels, kernel_size=1), nn.Sigmoid())

    def conv_block(self, in_channels, out_channels):
        return 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):
        # Encoder
        enc1 = self.enc1(x)
        enc2 = self.enc2(F.max_pool2d(enc1, 2))
        enc3 = self.enc3(F.max_pool2d(enc2, 2))
        enc4 = self.enc4(F.max_pool2d(enc3, 2))

        # Bottleneck
        bottleneck = self.bottleneck(F.max_pool2d(enc4, 2))

        # Decoder
        dec4 = self.dec4(torch.cat([F.interpolate(bottleneck, scale_factor=2, mode='bilinear', align_corners=True), enc4], dim=1))
        dec3 = self.dec3(torch.cat([F.interpolate(dec4, scale_factor=2, mode='bilinear', align_corners=True), enc3], dim=1))
        dec2 = self.dec2(torch.cat([F.interpolate(dec3, scale_factor=2, mode='bilinear', align_corners=True), enc2], dim=1))
        dec1 = self.dec1(torch.cat([F.interpolate(dec2, scale_factor=2, mode='bilinear', align_corners=True), enc1], dim=1))

        # Output
        output = self.output(dec1)

        return output
```

Fig.6 Unet structure

1.4.2 ResNet34_UNet detail

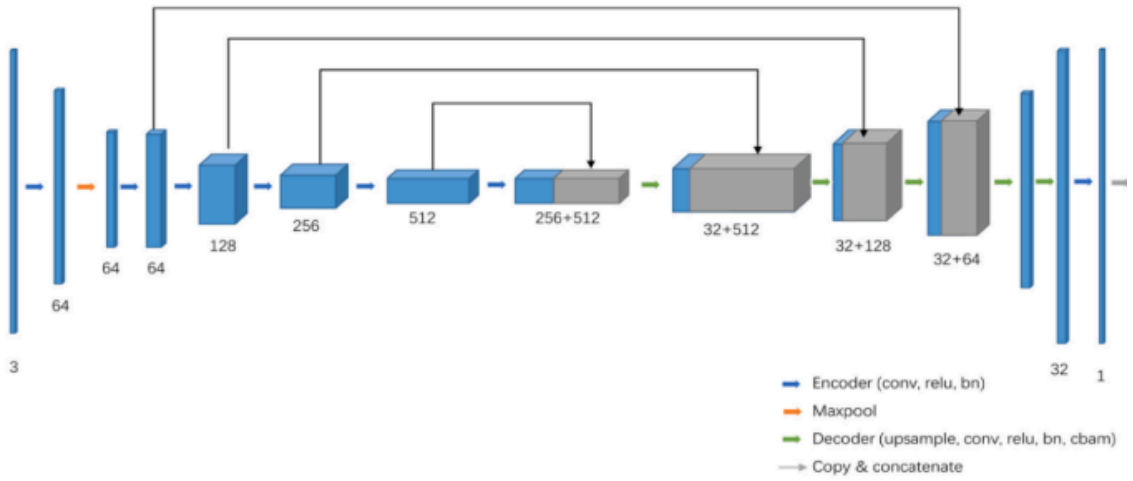


Fig.7 Overall structure

實作 ResNet34_UNet 時有參考該網站之內容:

https://blog.csdn.net/weixin_43977304/article/details/121497425

https://github.com/hank891008/Deep-Learning

在 ResNet34_UNet 的架構中, ResNet34 作為encoder部分, 由多個ResidualBlock 構成, 而 UNet 的結構則負責decoder。

1. ResidualBlock:

   裡面包含了Shortcut以及Main Path 了

   - Shortcut :在當 down 為 True 時, 表示需要進行降維操作, 此時Shortcut 使用一個 1x1 的conv layer和 batchnorm layer, 將輸入的特徵圖尺寸減半, 並依據 output channel 數操作。這樣才可以確保shortcut的輸出與 Main Path 的output在channel上對的起來, 才能夠相加。Down = False 時就直接吐出input (nn.Identity())
   - Main Path:Main Path包含兩個 3x3 的conv layer, 每個conv layer後面都接batchnorm layer再過一層Relu。當 down 為 True 時, 第一個卷積層的步幅設為 2, 才能完成Downsample。

```python
class ResidualBlock(nn.Module):
    def __init__(self, input_size, output_size, down=False):
        super(ResidualBlock, self).__init__()
        if down:
            self.shortcut = nn.Sequential(
                nn.Conv2d(input_size, output_size, kernel_size=1, stride=2),
                nn.BatchNorm2d(output_size)
            )
        else:
            self.shortcut = nn.Identity()

        self.block = nn.Sequential(
            nn.Conv2d(input_size, output_size, kernel_size=3, padding=1, stride=2 if down else 1, padding_mode="reflect"),
            nn.BatchNorm2d(output_size),
            nn.ReLU(inplace=True),
            nn.Conv2d(output_size, output_size, kernel_size=3, padding=1, padding_mode="reflect"),
            nn.BatchNorm2d(output_size),
            nn.ReLU(inplace=True)
        )

```

Fig.7 Residual Block

2. Encoder 部分一開始會先有一個初始層(init), 把原本輸入圖片(3,256,256)過 conv layer 跟maxpool 之後變成(64,64,64)的特徵。接著就是4個down sample block , 把channel 數量提高的同時, 把圖片壓小, 4個block的內容如下。

   - down ample block 1:3 個 ResidualBlock, 通道數保持為 64, 不進行 downsample

- down sample block 2：3 個 ResidualBlock, channel數從 64　增加到 128, 第一個block 做downsample
- downsample block 3：3 個 ResidualBlock, channel數從 128 增加到 256, 第一個block 做downsample
- downsample block 4：3 個 ResidualBlock, channel數從 256 增加到 512, 第一個block 做downsample

3. 而down 做完之後要再過一個bottleneck，一樣是使用上面的Residual_Block。
4. 過完bottleneck之後就到了Encoder Unet 的部分, 和先前提到的Unet完全一樣, 上面講過了所以不再重複敘述。

```
class ResNet34_UNet(nn.Module):
    def __init__(self, input_size=3, output_size=1, num_block=[3, 4, 6, 3]):
        super(ResNet34_UNet, self).__init__()
        self.init = nn.Sequential(
            nn.Conv2d(input_size, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.downs = nn.ModuleList()

        self.downs.append(nn.Sequential( #down sample block  1
            ResidualBlock(64, 64, False),
            ResidualBlock(64, 64, False),
            ResidualBlock(64, 64, False)
        ))

        self.downs.append(nn.Sequential( #down sample block  2
            ResidualBlock(64, 128, True),
            ResidualBlock(128, 128, False),
            ResidualBlock(128, 128, False)
        ))

        self.downs.append(nn.Sequential( #down sample block  3
            ResidualBlock(128, 256, True),
            ResidualBlock(256, 256, False),
            ResidualBlock(256, 256, False)
        ))

        self.downs.append(nn.Sequential( #down sample block  4
            ResidualBlock(256, 512, True),
            ResidualBlock(512, 512, False),
            ResidualBlock(512, 512, False)
        ))

        self.btnk = ResidualBlock(512,1024, True)
```

```
        self.ups = nn.ModuleList()
        for feature in reversed([64, 128, 256, 512]):
            self.ups.append(nn.ConvTranspose2d(feature * 2, feature, kernel_size=2, stride=2))
            self.ups.append(DoubleConv(feature * 2, feature))

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

    def forward(self, x):
        x = self.init(x)
        skips = []
        for down in self.downs:
            x = down(x)
            skips.append(x)
        x = self.btnk(x)
        for up in self.ups:
            if isinstance(up, nn.ConvTranspose2d):
                x = up(x)
                x = torch.cat((x, skips.pop()), dim=1)
            else:
                x = up(x)
        x = self.output(x)
        return x

# assert False, "Not implemented yet!"
```

Fig.8 ResNet34_UNet

1.5 Implement details

training 全程都在 GPU 上進行, 有一點提升訓練速度很重要的是使用 torch.cuda.amp 進行自動混合精度 (Automatic Mixed Precision, AMP) 訓練, 用了這個之後大大的加速了整個training process。

2. Data Preprocessing

資料前處理的部分, 我import了 Albumentations 來進行, 對data進行了一系列的操作來提升模型的學習能力。其中比較特別的應該是最後一點標準化, 依照ImageNet 的mean與std結果作標準化, 讓Input 比較符合統計過的結果, 這個用torch .transform 沒辦法做。

- 隨機水平翻轉 (A.HorizontalFlip(p=0.5)) ：以 50% 的機率對影像進行水平翻轉, 減少model對input方向的過度依賴。
- 隨機裁切 (A.RandomResizedCrop(size=(256, 256), scale=(0.8, 1))) : 在 80% ~ 100% 的範圍內隨機裁切影像, 再調整至 256x256, 讓model學習input不同區域的特徵, 增加影像的變異性。
- 隨機旋轉 (A.Rotate(limit=30, p=0.5)) : 以 50% 機率將影像隨機旋轉 ±30°, 幫助模型學習不同旋轉角度下的特徵, 提高對目標物件的適應性。
- 顏色擾動 (A.ColorJitter) : 以 50% 的機率調整影像的亮度、對比度、飽和度和色相 (hue), 避免模型過度依賴固定的光線條件, 提高模型的適應性。

- 標準化 (A.Normalize(mean, std))：使用 ImageNet 的標準化參數 (mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))，確保影像數據符合常見的預處理方式，有助於提升模型的穩定性。

```
train_transform = A.Compose(
    [
        A.Resize(256, 256),
        A.HorizontalFlip(p=0.5),
        A.RandomResizedCrop(size=(256, 256), scale=(0.8, 1)),
        A.Rotate(limit=30, p=0.5),
        A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2, p=0.5),
        A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
        ToTensorV2(),
    ],
)
```

Fig.6 Transform details

3. Analyze the experiment result

底下列出了分別訓練兩種網路時，取得最好dice score 的hyper parameters
值得注意的是兩者的lr 相差了許多，我覺得是因為 ResNet34_UNet 有用到Residual block，這種結構能夠讓gradient更穩的back propagate ,可以緩解gradient loss的問題，所以用比較大的lr 比較不會讓model 不穩定或者是發散。而較為傳統的 UNet，沒有 Residual block，因此需要較低的學習率來確保穩定收斂。

	UNet	ResNet34_UNet
learning rate	0.0002	0.002
epoch	65	325
batch	32	64
Optimizer	Adam	Adam
Dice score Result	0.9337	0.9428

1. With or without data preprocessing

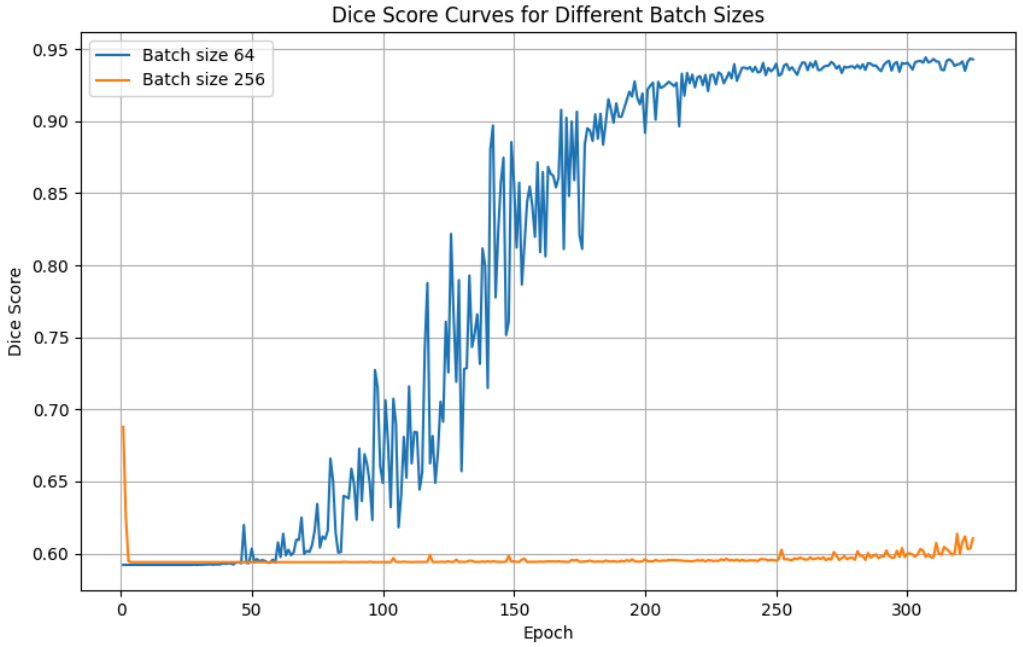
一開始還沒有寫transform時我有維持一模一樣的hyper parameters，但是試了好幾組參數dice score 都無法達到0.9 以上，比較表格如下：

	UNet	ResNet34_UNet
With data preprocessing	0.9337	0.9428
without data preprocessing	0.8843	0.8614

可以從上面的比較推斷data preprocessing 對於模型訓練的好處，讓model有比較好的泛化性。

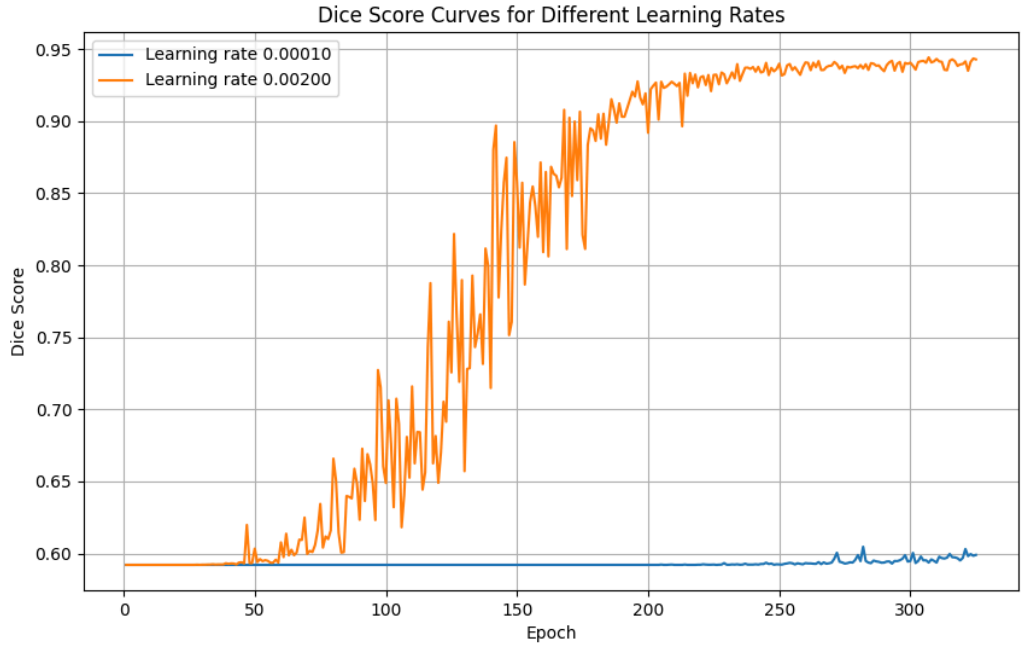
2. Batch size

另外在訓練時因為跟gpu 有時候有其他人一起在用, memory 容量不一定是相同的, 所以有設定不同的batch size, 結果發現了batch size 對於model performance 有著很大的影響, 下圖為針對resnet34_unet不同batch size進行訓練時的dice score , 可以很明顯的看出batch = 256時, model基本上是完全沒有在學習的, 應該是因為當batch size 從 64 增加到 256 時, 每個 epoch 的梯度更新次數會減少, 使得權重更新smooth一點, 同時也降低model探索optimal solution 的能力。如果使用相同的學習率, 較大的 batch size 可能導致每次更新的梯度平均效果過強, 從而使得模型難以跳出loacal area。



3. Learning Rate

下圖為針對resnet34_unet不同lr進行訓練時的dice score , 可以很明顯的看出lr = 0.0001時, dice score 完全沒有上升, 應該是因為梯度更新步伐過小, model weight 無法有效調整, 導致分割 score 長期停留在 0.59左右。



4. Execution steps (0%)

train.py 中有設定seed = 42 所以照著指令跑能夠reproduce 一樣的結果

```
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)
if torch.cuda.is_available():
torch.cuda.manual_seed_all(seed)
```

Training command 在(src資料夾底下執行):

UNet : python train.py -e 65 -b 64 -lr 0.0002 -n unet -g 0

ResNet34 : python train.py -e 325 -b 64 -lr 0.002 -n resnet34_unet -g 0

有以下參數可以進行調整

- --data_path, '../../dataset/oxford-iiit-pet' (default)
- --epochs , 325(default)
- --batch_size , 128 (default)
- --learning_rate, 1e-3 (default)
- --network, resnet34_unet or unet
- --gpu,0(default)

```
def get_args():
    parser = argparse.ArgumentParser(description='Train model')
    parser.add_argument('--data_path', '-p', type=str, default='./../dataset/oxford-iiit-pet', required=False, help='Path to input data')
    parser.add_argument('--epochs', '-e', type=int, default=5, help='Number of training epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=128, help='Batch size')
    parser.add_argument('--learning_rate', '-lr', type=float, default=1e-5, help='Learning rate')
    parser.add_argument('--network', '-n', type=str, default='resnet34_unet', help='Network architecture')
    parser.add_argument('--gpu', '-g', type=int, default=0, help='GPU device number')
    return parser.parse_args()
```

Inference command (src資料夾底下執行):

UNet : python inference.py -m modelname.pth -n unet

ResNet34 : python inference.py -m modelname.pth -n ResNet34_UNet

其中modelname需要是存在saved_models/下的pth檔案

```
(dlp_lab2) winston@gpu7:~/dlp/lab/lab2/src$ python inference.py -m best_unet_lr0.00020_epoch65_batch32.pth -n unet
using device: cuda:7
Dice Score :  0.9337
```

- --data_path, '../../dataset/oxford-iiit-pet' (default)
- --model , 打儲存在saved_models底下的model name 即可,MODEL.pth (default)
- --batch_size , 64 (default)
- --network, ResNet34_UNet or unet

```
def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model','-m', default='MODEL.pth', help='path to the stored model weight')
    parser.add_argument('--data_path','-d', type=str, default = "./../dataset/oxford-iiit-pet",help='path to the input data')
    parser.add_argument('--batch_size', '-b', type=int, default=64, help='batch size')
    parser.add_argument('--network','-n', type=str, default='ResNet34_UNet', help='model type')
    return parser.parse_args()
```

5. Discussion (20%) Please discuss what alternative architectures could potentially yield better results for this task. Provide reasoning for your choices based on your understanding of the model and dataset. Additionally, identify potential research directions related to this task and explain why these topics are worth exploring. Any other reflections or discussions that you find relevant can also be included in this section.

在這次lab的task中, 我們主要採用了 UNet 和 ResNet34_UNet 兩種架構來處理 Oxford-IIIT Pet 資料集。儘管這兩者的dice score 都已經超過0.9, 但兩者都已經是蠻舊的model ,現在有更新且效果更好的model 可以使用，像基於unet 再發展的
U-Net＋＋，他基於unet有更多的shortcut, 所以應該是能夠捕捉到各多圖片裡的資訊, 表現也理應當會比unet來的更好。而現在常用的Attention method 我不確定是否合適, 因為觀察這個 Oxford-IIIT Pet 裡的背景和動物所佔的比例其實並沒有相差很大, 所以不確定現有的attention method 是否能取得更好的result。

Transformer-based Models像是Swin Transformer應該也能夠做的不錯, 因為先前提過的背景和動物所佔的比例其實並沒有相差很大，所以更能夠利用全局自注意力, 來捕捉長距離的特徵，進而在邊緣檢測和細節恢復上表現出色。