Lab2: Binary Semantic Segmentation

謝鎧駿 B103040021

1. Implementation Details

i. Training

Train 的步驟其實跟 Lab1 差不多，首先建立 Model，也就是 Unet or ResNet34_Unet，接著將 Dataset load 進來，並建立 DataLoader；接著選定要使用的 Optimizer 與 Loss function，Optimizer 我採用 Adam，因為 Adam 具有 Momentum 與自適應 Learning rate，而 Loss function 我選擇使用 Binary cross entropy，因為 BCE 非常適合用在處理 pixel 的二元分類問題，可以讓模型有效學習區分物體與背景。

且我有使用 scheduler 去調整 Learning rate。

並且因為第一次 train 的時候有出現梯度異常情況，所以我有用 autograd 去標記梯度異常的地方。

最後是設置 best dice score 為 0.8，(因為至少大於 0.8 才有分數…)，當 validation dice score 大於 best dice score 時，就更新 best dice score 並 save model。

```
optimizer = optim.AdamW(model.parameters(), lr=args.learning_rate)
criterion = nn.BCELoss()
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.98)
writer = SummaryWriter(f"runs/{args.model}/")
torch.autograd.set_detect_anomaly(True)
best_dice_score = 0.8
```

而訓練的步驟就是 Forward、Loss、Backward、Optimize。

```
progress = tqdm(enumerate(train_loader))
for idx, batch in progress:
    img = batch["image"].to(args.device)
    mask = batch["mask"].to(args.device)
    y_pred = model(img)
    loss = criterion(y_pred, mask)
    train_losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    dc = dice_score(y_pred, mask)
    train_dcs.append(dc.item())
```

ii. Evaluating

Evaluation 的步驟基本跟 training 步驟一樣，但是不用更新 weight 並且記得將 model 改成 Evaluation mode，而因為我有使用 tqdm 去記錄模型的訓練情況，發現一 epoch 需要跑到 7-8 分鐘，所以後來在 Evaluation 加入 torch.no_grad()來節省記憶體的使用，並希望能提升 Evaluation 的速度。

iii. Inferencing

Inference 的時候就真的跟 Evalution 一樣，但記得要先將 model 給 Load 進來就好了。

```
model = torch.load(f"../saved_models/{args.model}")
```

iv. Unet architecture

因為 Unet 每層都要用兩個 Convolution layer，所以先寫一個 Double Convolution 的 class，本來我第一次實作的時候是沒有加入 BatchNorm，但是在那次訓練中期卻突然出現 Loss 飆升的情況，我推測是模型 Overfitting 或是梯度變化過大，所以後來我才加入了 BatchNorm，去解決 Loss 突然飆升的情況，並期望能加速模型收斂。

```
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, dilation=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, dilation=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

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

接下來是實作 Unet 的 Downsampling，Down sample 相對 Up sample 來說較為簡單，每層都是一個 Double convolution 後接一個 Max pooling，比較要留意的地方是，我們必須將 convolution 的結果 return，以便在 Up sampling 的時候做 skip connection。

```
class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSample, self).__init__()
        self.conv = DoubleConv(in_channels, out_channels)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        down = self.conv(x) # we saved the convolutioned tensor
        p = self.pool(down)

        return down, p
```

再來是 Up sampling，每層是一個 Transpose convolution 接 Double convolution，但必須記得將 Down sampling 的特徵圖拿來連接。

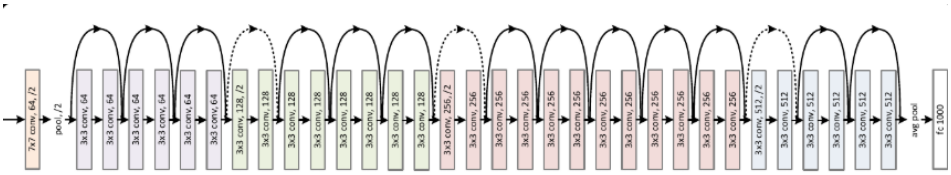
```
class UpSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UpSample, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels//2, kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        x = torch.cat([x1, x2], 1) # x2 is from encoder
        return self.conv(x)
```

最後就是組合出 Unet 啦，首先是 4 個 Doen sampling 後接一個 bottleneck，bottleneck 主要是升維的作用。後續再接 4 個 Up sampling，最後再通過一個 kernal size=1 的 Convolution，使其回到原本影像大小的圖片，也別忘了通過 Sidmoid 去把輸出對映到 0~1，我記得一開始忘記通過 Sidmoid，導致有 bug 產生。

```
class UNet(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(UNet, self).__init__()
        self.down_convolution_1 = DownSample(in_channels, 64)
        self.down_convolution_2 = DownSample(64, 128)
        self.down_convolution_3 = DownSample(128, 256)
        self.down_convolution_4 = DownSample(256, 512)

        self.bottle_neck = DoubleConv(512, 1024)

        self.up_convolution_1 = UpSample(1024, 512)
        self.up_convolution_2 = UpSample(512, 256)
        self.up_convolution_3 = UpSample(256, 128)
        self.up_convolution_4 = UpSample(128, 64)

        self.out = nn.Conv2d(in_channels=64, out_channels=num_classes, kernel_size=1)
        self.last = nn.Sigmoid()

    def forward(self, x):
        down_1, p1 = self.down_convolution_1(x)
        down_2, p2 = self.down_convolution_2(p1)
        down_3, p3 = self.down_convolution_3(p2)
        down_4, p4 = self.down_convolution_4(p3)

        b = self.bottle_neck(p4)

        up_1 = self.up_convolution_1(b, down_4)
        up_2 = self.up_convolution_2(up_1, down_3)
        up_3 = self.up_convolution_3(up_2, down_2)
        up_4 = self.up_convolution_4(up_3, down_1)

        out = self.out(up_4)
        output = self.last(out)
        return output
```

v. ResNet34_Unet architecture

我的 ResNet34 主要是依照下圖去製作出來，但要記得因為要跟 Unet Decoder 做連接，所以最後不用通過全連結層。



首先是 ResidualBlock，透過觀察上面的架構圖可以發現 ResidualBlock 都是兩層的 Convolution 組成，所以我們先寫一個 ResidualBlock 的 Module，方便後面使用。

特別注意到因為 BatchNorm 中就提供了 Bias 的效果，所以 Conv2d 的 bias 就要設置為 False。

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channel, out_channel, stride=1, shortcut=None):
        super(ResidualBlock, self).__init__()
        self.left = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, 3, stride, 1, bias=False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(),
            nn.Conv2d(out_channel, out_channel, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU()
        )
        self.right = shortcut
    def forward(self, x):
        out = self.left(x)
        residual = x if self.right is None else self.right(x)
        out = out + residual
        return out
```

接著就可以先建立 ResNet34_Unet 中 ResNet 的部分，首先我們先處理 ResNet 最前面的 Convolution 與 Max pooling，因為跟 ResidualBlock 不同，所以必須獨立撰寫，參數的部分就是直接照著寫。

```
class resnet_34_unet(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(resnet_34_unet, self).__init__()
        # Conv1 = 7x7, 64, stride=2
        # 3x3 Max pool, stride = 2
        self.pre_layer = nn.Sequential(
            nn.Conv2d(in_channels, 64, 7, 2, 3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 2, 1)
        )
        # Conv2, 3 ResidualBlock
        self.layer1 = self.make_layer(64, 64, 3)
        # Conv3, 1 stride=2 ResidualBlock and 3 ResidualBlock
        self.layer2 = self.make_layer(64, 128, 4, stride=2)
        # Conv4, 1 stride=2 ResidualBlock and 5 ResidualBlock
        self.layer3 = self.make_layer(128, 256, 6, stride=2)
        # Conv3, 1 stride=2 ResidualBlock and 2 ResidualBlock
        self.layer4 = self.make_layer(256, 512, 3, stride=2)
        # BottleNeck, use a ResidualBlock to implement
        self.bottleneck = self.make_layer(512, 1024, 1, stride=2)
```

再來說明 make_layer，我們回到最上面的 ResNet34 的架構圖，可以發現雖然是一直重複做 ResidualBlock 的事情，但有時候卻要 convolution 時 stride=2 來縮小圖片，也就是架構圖中的虛線箭頭，所以我們寫一個 make_layer function 來處理此問題。

可以看到只有第一個 layer 會傳 stride 跟 shortcut 參數給 Residual Block，回到 ResidualBlock 可以看到 shortcut 的作用是把維度變成跟 out_channel 一樣，這樣才能被相加。

```
def make_layer(self, in_channel, out_channel, block_num, stride=1):
    shortcut = nn.Sequential(
        nn.Conv2d(in_channel, out_channel, 1, stride, bias=False),
        nn.BatchNorm2d(out_channel),
    )
    layers = []
    # first layer may need to down sample
    layers.append(ResidualBlock(in_channel, out_channel, stride, shortcut))
    for i in range(1, block_num):
        layers.append(ResidualBlock(out_channel, out_channel))
    return nn.Sequential(*layers)
```

最後就可以把全部兜起來了，如下圖所示，首先是 ResNet34 的 pre_layer 跟 4 個 make_layer，而 bottleneck 我也用 make_layer 去做，其實就是一個 ResidualBlock，接著就是直接拿 Unet 的 Decoder 來用，並且最後要記得將圖片大小還原！

```
class resnet_34_unet(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(resnet_34_unet, self).__init__()
        # Conv1 = 7x7, 64, stride=2
        # 3x3 Max pool, stride = 2
        self.pre_layer = nn.Sequential(
            nn.Conv2d(in_channels, 64, 7, 2, 3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 2, 1)
        )
        # Conv2, 3 ResidualBlock
        self.layer1 = self.make_layer(64, 64, 3)
        # Conv3, 1 stride=2 ResidualBlock and 3 ResidualBlock
        self.layer2 = self.make_layer(64, 128, 4, stride=2)
        # Conv4, 1 stride=2 ResidualBlock and 5 ResidualBlock
        self.layer3 = self.make_layer(128, 256, 6, stride=2)
        # Conv3, 1 stride=2 ResidualBlock and 2 ResidualBlock
        self.layer4 = self.make_layer(256, 512, 3, stride=2)
        # BottleNeck, use a ResidualBlock to implement
        self.bottleneck = self.make_layer(512, 1024, 1, stride=2)

        self.up_convolution_1 = UpSample(1024, 512)
        self.up_convolution_2 = UpSample(512, 256)
        self.up_convolution_3 = UpSample(256, 128)
        self.up_convolution_4 = UpSample(128, 64)

        self.last = 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, num_classes, kernel_size=1),
            nn.Sigmoid()
        )
```

2. Data Prepocessing

i. How you preprocess data

其實我一開始想說先試試看沒有任何 Preprocess 的結果，然而卻開始遇到很多 Bug。首先是，TA 的程式碼中有用到 unsqueeze()，此 function 的輸入必須是 Tensor，所以必須先將影像從 numpy 轉換成 tensor；第二，模型的輸入必須是 float32，而我們 RGB 的影像輸入卻是 0~255，所以需要先將影像給標準化。

```
albumentations.Resize(256, 256),
albumentations.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
ToTensorV2()
```

解決完這兩個問題後，就能順利開始訓練模型，一開始 Loss 正常的持續往下掉，然而卻突然在某個 Epoch，Loss 就飆升到 0.4 多，並且一直居高不下，此時我推測是模型 Overfitting、或是梯度變化過大導致 Loss 急劇上升、也有可能是 Learning rate 過大導致模型在低 Loss 附近無法穩定，因此最後我只能每個推測都去解決，而我主要是做 Data Augmentation 去降低 Overfitting，當中包含：

- 水平翻轉
- 旋轉
- 縮放 20%
- 隨機裁剪並縮放
- 改變亮度、對比度
- 隨機遮擋

而我使用的是 Albumentations 而不是 torchvision，因為做 Segmentation 需要同步變換 Mask，而 Albumentations 才可以做到同步變換 Mask。

```
albumentations.RandomScale(scale_limit=0.2, p=0.5),
albumentations.HorizontalFlip(p=0.5),
albumentations.Affine(scale=(0.8, 1.2), translate_percent=(0.1, 0.1), rotate=(-30, 30)),
albumentations.RandomResizedCrop(size=(256, 256), scale=(0.8, 1.0), p=0.5),
albumentations.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2, p=0.5),
albumentations.GridDropout(ratio=0.3, random_offset=True, p=0.5),
albumentations.Resize(256, 256),
albumentations.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
ToTensorV2()
```

ii. What makes your preprocessing method unique

我認為比較獨特的 preprocessing method 是 Normalization 的參數我是使用 ImageNet 的標準化參數，為什麼使用此標準化參數呢?

首先是因為我發現 DataSet 中的貓狗圖片顏色分布和 ImageNet 類似，

且通過此參數之標準化，會使 Dataset 更符合實際影像。

並且因為貓狗的佔圖片比例很大，模型可能會學習到，整張圖幾乎都是前景，導致邊界部分的預測不夠準確。因此我在 Data Augmentation 中也加入了隨機遮擋，讓模型去學習不完整的物件輪廓。

3. Analyze the experiment results

i. **Exploration during the training process**

I. Hyperparameter settings:

- Batch size = 32
- Epoch = 400
- Learning rate = 1e-4
- Optimizer = Adam
- Loss function = BCELoss + Dice loss

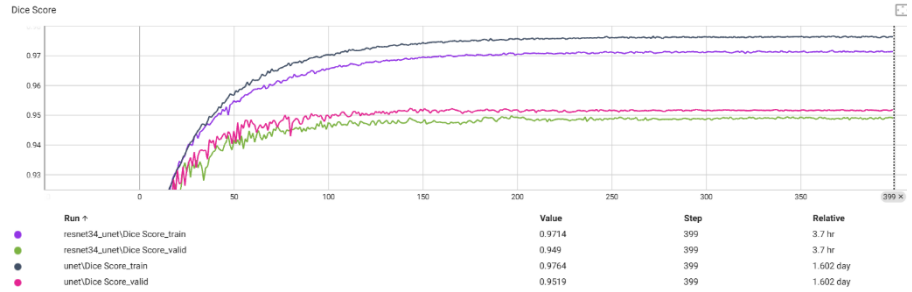
在這邊 Epoch 使用 200，是因為一開始使用 400，但導致訓練時常太久，且發現將 Epoch 設置為 200 時，Loss 也已經收斂，所以就採用 200 了。

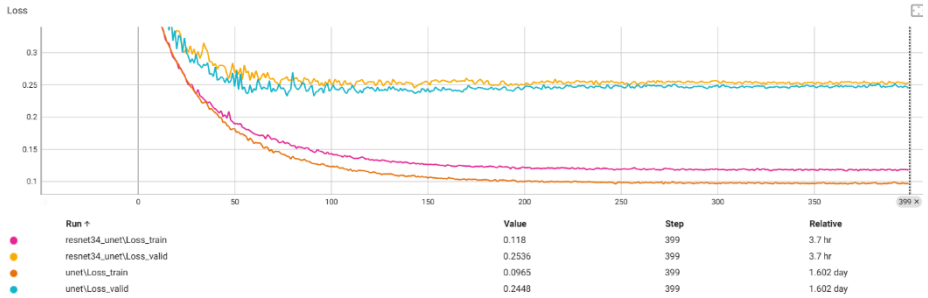
而 Loss function 的部分，本來是只採用 BCELoss function，但後來去研究 Binary semantic segmentation 的問題後，發現很多人會再加上一個 Dice Loss 去加強模型對於邊緣的細節處理，因此我後來也加入了 Dice Loss。

II. Training process

首先，我第一次訓練時 Unet 突然 Loss 飆升、dice score 暴跌，而經過 Data Augmentation 與 Batch Normalization 調整後，就可以穩定收斂了。而我的 Unet 與 ResNet34_Unet 表現其實旗鼓相當，ResNet34_Unet 在前期的表現稍微好一點，然而到後期，Unet 的表現好一點。

但 Unet 與 ResNet34_Unet 差異最大的是 training 的速度，我的 Unet 訓練時長為 3.7hr，而 ResNet34_Unet 則是 1.6day。





ii. Observation of dataset

在 dataset 中我們可以發現貓跟狗佔整張圖片的比例是很高的，因此選用 BCELoss 會有不錯的表現，而不會去選用 Focal loss，因為此 Loss 是特別適合處理小物件的。但是後來我有加入 Dice loss 來讓模型更關注於邊界的細節。

並且因為貓狗的佔圖片比例很大，因此我在 Data Augmentation 中也加入了隨機遮擋，讓模型去學習不完整的物件輪廓。

iii. Testing result

```
inference on unet.pth
Mean Dice Score: 0.9521967944891556
```

```
inference on resnet34_unet.pth
Mean Dice Score: 0.9504568329231717
```

從 Test result 可以發現兩模型的結果相近，而 Unet 略好一些，但兩者的分數都挺高的。

我的 testing result 會放在 dataset/oxford-iiit-pet/output_imgs

Unet: ResNet34_Unet:

4. Execution steps

訓練參數設定：

- Model : unet / resnet34_unet
- Device : cpu / cuda
- Epoch : 400
- Data path : "../dataset/oxford-iit-pet/"
- Batch size : 32
- Learning rate : 1e-4

Train :

python train.py --model unet --device cuda --data_path "../dataset/oxford-iiit-pet" --epochs 400 --batch_size 32 --learning_rate 1e-4

Inference : 不需要 epochs 以及 learning_rate

python inference.py --model unet.pth --device cuda \

--data_path "../dataset/oxford-iiit-pet/" --batch_size 1

5. Discussion

i. Alternative architectures

針對 Unet 來說的話，我認為實作 Attention U-Net 應該會比 Unet 有更好的結果，因為引入了 Attention，可以使模型更去注意一些細節及重要的區塊。

再者是 ResNet34_Unet，我認為若將 ResNet34 與 Unet 分開訓練，再將其結合起來 fine-tuning，我認為會讓模型能夠更快收斂並提升 Segmentation 效果。

最後，我認為如果模型要處理的 task 變得更複雜的話，我認為可以透過 Ensemble learning 的方式提升模型的能力，相信會更能去處理複雜的問題。

ii. Potential research directions

提到 Image segmantation 的話一定要提到醫學影像分析，Image segmantation 可以幫助自動識別和分割影像中的病變或目標區域，這能有效的提高醫療診斷的準確性和效率。

再者是自駕車，Image segmantation 可以用來區分道路和非道路區域、車輛和行人，能顯著提升車輛在動態環境中的安全性和導航能力，並且自駕車也是近年來發展非常夯的研究。