

DLP

LAB 3

謝侑哲

112550069

Github:

<https://github.com/youzhe0305/NYCU-DLP>

1. Overview

本次 lab 為·使用 UNet, ResNet34+UNet 來做 Binary Semantic Segmentation·
本次針對寵物圖片做訓練及應用·使用的資料集為 Oxford-IIIT Pet Dataset。這次主要做的分類是基於 trimap·把圖片分成前景及背景。在評估上則使用 Dice Score 評估 prediction 與 ground truth·把注意力放在前景的分割上。

2. Implementation Details

A. Details of your training, evaluating, inferencing code

(1) Training Code

Step1 初始化·建立工具:

```
hyper_parameter = {  
    'n_epoch': 200,  
    'batch_size': 80, # 3312 samples for train  
    'learning_rate': 0.001,  
    'regularization': 0,  
    'bilinear': True,  
}  
  
dataset = load_dataset('dataset', 'train')  
dataloader = DataLoader(dataset, hyper_parameter['batch_size'], shuffle=True)  
if os.path.exists('saved_models/model_Res34_UNet.pth'):  
    print('load trained model')  
    model = torch.load('saved_models/model_Res34_UNet.pth')  
else:  
    print('not load model')  
    model = ResNet34_Unet(in_channels=3, n_class=2, bilinear=hyper_parameter['bilinear'], device=device)  
model.train()  
optimizer = torch.optim.Adam(model.parameters(), lr=hyper_parameter['learning_rate'], weight_decay=hyper
```

與 Lab2 的訓練方式類似·先引入 unet.py, resnet34_unet.py 建立好的模型·接著根據之前的訓練結果調整 hyper parameter。

接著把 oxford_pet.py 建立好的 Dataset 引入·設定為 train 模式·取得訓練用的圖片跟 mask·接著再使用 Dataloader 能分 batch 讀取資料·並且做到隨機化(shuffle)·讓每次訓練的資料組合不一樣·增加 Robustness。最後引入一個與論文的 SGD 不同但理論上更好的 Adam optimizer·一個結合了 Momentum 跟 Adaptive Grad 的更新方式。

Step2 訓練過程

```
for epoch in range(hyper_parameter['n_epoch']):
    epoch_loss = []
    for idx, samples in enumerate(dataloader):
        img = samples['image'].type(torch.float).to(device)
        mask = samples['mask'].to(device) # class (2 class)
        trimap = samples['trimap'].to(device) # separate image to 3 class: 1-front-ground, 2-background, 3-unknown
        output = model(img)
        loss = model.get_loss(output, mask)
        loss.backward()
        epoch_loss.append(loss.item())
        optimizer.step()
        optimizer.zero_grad()
    avg_epoch_loss = sum(epoch_loss) / len(epoch_loss)
    validation_loss, validation_dice = evaluate(model, device)
    print(f'epoch: {epoch+1}, training_loss: {round(avg_epoch_loss,4)}, testing_loss: {round(validation_loss,4)}, validation_dice: {round(validation_dice,4)}')
```

首先用 for 迴圈做 n_epoch 次的訓練，接著 enumerate dataloader，每次取得一個 batch 的資料，再把它丟進 model，即可得到 output

接著把 prediction 跟 mask 傳入建在 model 裡的 loss function，因為 segmentation 可以視為 classify 的一個特例，所以這裡使用的跟 lab2 一樣是 CrossEntropy，得到 loss tensor 之後就可以對其做 backward propagation，得到每個 weight tensor 的 grad，使用 optimizer 做更新。

最後，引入 evaluate.py 裡寫得 validation function。測試每個 epoch 的 validation accuracy，保留最高 accuracy 的參數，避免 epoch 太多導致 overfitting 之後，留到的是 overfitting 的參數。

(2) Evaluating Code

Evaluating Code 大致上跟 Training Code 相同，只差在 dataset 是取 valid 的部分，並且 model 是作為參數傳入，以便在訓練時，把每個 epoch 的參數與 model 傳入。

```
dataset = load_dataset('dataset', 'valid')
dataloader = DataLoader(dataset, hyper_parameter['batch_size'], shuffle=False)
model = net
model.eval()
```

```

for idx, samples in enumerate(dataloader):
    img = samples['image'].type(torch.float).to(device)
    mask = samples['mask'].to(device) # class (2 class)
    trimap = samples['trimap'].to(device) # separete image t

    output = model(img)

    loss = model.get_loss(output, mask)
    epoch_loss.append(loss.item())

    # batch, channel, H, W
    class_prob = F.softmax(output, dim=1)
    prediction = torch.argmax(class_prob, dim=1)

    dice = dice_score(prediction, mask)
    epoch_dice.append(dice.item())

avg_dice = sum(epoch_dice) / len(epoch_dice)
avg_epoch_loss = sum(epoch_loss) / len(epoch_loss)
return avg_epoch_loss, avg_dice

```

Forwarding 的過程與 training 一樣，這裡唯一的差別就是把 forwarding 的 output 丟入 softmax function，得到屬於前景 or 背景的機率，再取機率大的那方作為 prediction 結果。

之後再用從 utilis.py 引入的 dice_score 計算 prediction 與 mask 的 dice score，用來評估 segmentation 的成果。Dice score 的計算如下：

```

def dice_score(pred_mask, gt_mask):
    # unify to (batch, H, W)
    gt_mask = gt_mask.view(gt_mask.shape[0], gt_mask.shape[2], gt_mask.shape[3])

    epsilon = 1e-6
    # for class only 0,1 dice take 1 part
    common_pixel_1 = torch.sum((pred_mask * gt_mask).type(torch.int), dim=(1,2))
    pred_pixel_1 = torch.sum((pred_mask==1).type(torch.int), dim=(1,2))
    gt_pixel_1 = torch.sum((gt_mask==1).type(torch.int), dim=(1,2))

    dice_score = (2*common_pixel_1 + epsilon) / (pred_pixel_1 + gt_pixel_1 + epsilon)
    return dice_score.mean()

```

首先找兩張圖同樣為 1 的部分，也就是同為前景的部分，計算共同的 pixel 數量有多少，再分別計算兩邊各自有多少為前景的部分。帶入 dice score 的公式，即可計算出，dice score 把評估重點放在我們想要切割的東西，也就是前景上面，比起單純比較差異更能夠體現出分割的效果是否符合需求。

(3) Inferencing Code

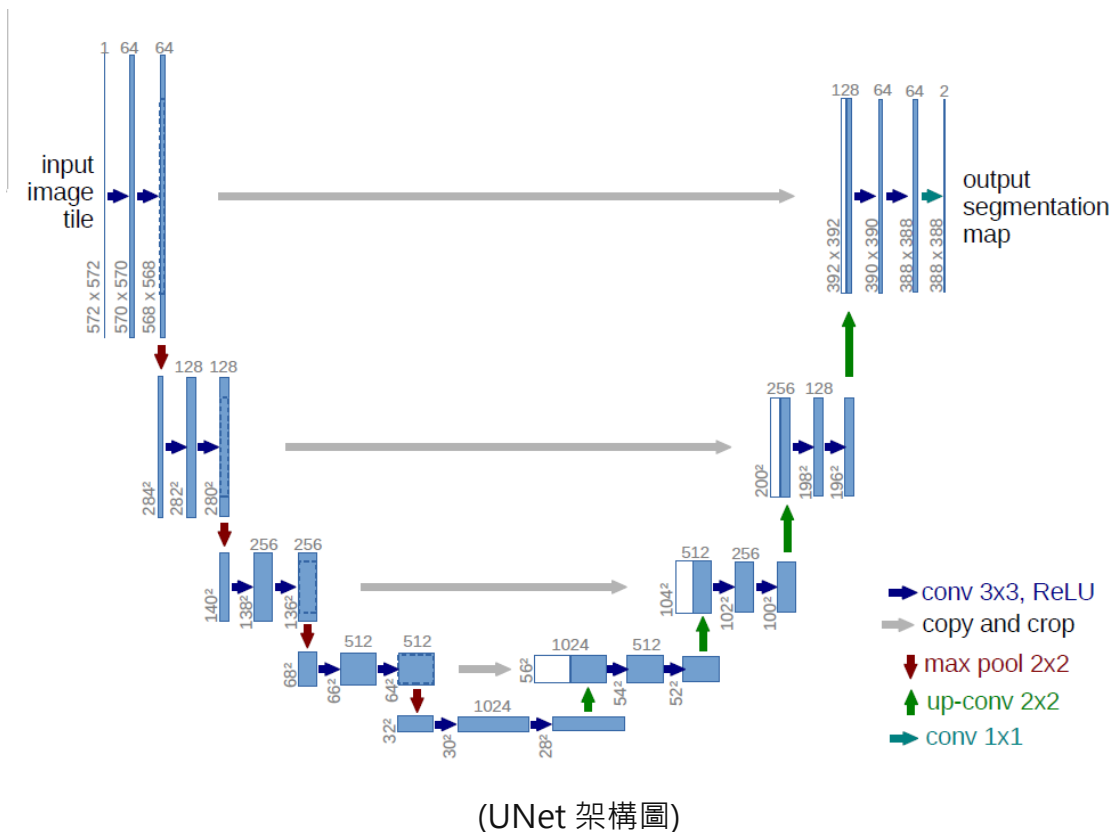
Inferencing 跟 evaluating 的寫法幾乎一樣，forward、評估的方式也相同，唯一的差別在於使用了 dataset 的不同部分，這裡使用的是 test 的部分。

```
dataset = load_dataset('dataset', 'test')
dataloader = DataLoader(dataset, hyper_parameter['batch_size'], shuffle=False)
model = torch.load('saved_models/model_Res34_UNet.pth')
model.eval()
```

B. Details of your model (UNet & ResNet34_UNet)

(1) UNet

UNet 基於 CNN，主要是用來處理圖片的 segmentation 任務，透過 shortcut connection 來讓分割的準度提升，網路架構圖如下，其主要分成 2 個 path，我將就這兩個 path 分別解釋：



- Contracting Path:

於架構圖中左側的部分，顧名思義，是用來把圖片縮小、提取重要特徵成 representation，之後再將 representation 交由後段的 expansive path decode 成 segmentation 圖。

其中有兩個重要的部分，Convolution 與 Pooling，可以看見 contracting path 都是由 2 個 Convolution+1 個 Pooling 的 Block 組成的，這兩個合稱為 Down Sampling，可以降低圖片的大小(解析度)，並且只保留對 segmentation 重要的特徵。每次 Down Sampling 都會讓圖片縮小 1/4，feature channel 則加深 2 倍。實作如下：

```

class DoubleConv(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels):
        super().__init__()
        self.first_conv = nn.Sequential(
            # padding = 1 -> maintain same size
            nn.Conv2d(in_channels=in_channels, out_channels=mid_channels, kernel_size=(3,3), padding_mode='reflect'),
            nn.BatchNorm2d(mid_channels), # feature channel
            nn.ReLU()
        )
        self.second_conv = nn.Sequential(
            nn.Conv2d(in_channels=mid_channels, out_channels=out_channels, kernel_size=(3,3), padding_mode='reflect'),
            nn.BatchNorm2d(out_channels), # feature channel
            nn.ReLU()
        )

    def forward(self, x):
        x_mid = self.first_conv(x)
        x_out = self.second_conv(x_mid)
        return x_out

```

首先建立兩層 Convolution 的 model 為 DoubleCov，並在每個 Convolution 後面加上 Batch Normalization 降低梯度消失或爆炸的問題，再補上 ReLU 增加 non-linear 的效果。接著把 MaxPooling 跟 DoubleConv 拼再一起，成為一個完整個 Down Sampling，對應到架構圖中就是一個向下的紅色箭頭與兩個向右的藍色箭頭。

另外，paper 中有提到，使用 mirror 做 padding 可以更好的減少邊界訊息的損失，因此這裡使用 reflect-padding 來做 padding，避免圖片在 Convolution 的過程中縮小，也不會因為 padding 導入差異很多的邊界訊息。

有了 Down Sampling 就可以把 contracting path 建出來，如下圖所示：

```

self.constracting_path_part1 = DoubleConv(in_channels=in_channels, mid_channels=64, out_channels=64)
self.constracting_path_part2 = DownSampling(in_channels=64, mid_channels=128, out_channels=128)
self.constracting_path_part3 = DownSampling(in_channels=128, mid_channels=256, out_channels=256)
self.constracting_path_part4 = DownSampling(in_channels=256, mid_channels=512, out_channels=512)
self.constracting_path_part5 = DownSampling(in_channels=512, mid_channels=1024//factor, out_channels=1024//f

```

- Expansive Path

架構圖的右側部分，類似於 decoder 的功能，可以把 contracting 提取出來的 representation 重新轉回正常大小的圖片。與 contracting path 的做法類似，這部分的實作上，我把 2 個 Convolution 與 1 個 up-convolution 合成 Up Sampling，可以用來將圖片恢復成原來的大小，並且把儲存在 feature channel 中的特徵整合在一起，變成之後 segmentation 分類的依據。實作如下

```

class UpSampling(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels, bilinear=True): # bilinear set True for preli

        if bilinear == True:
            # about align_corner: True: make source points on corner, False: make source point on crosshatch
            # ref: https://blog.csdn.net/wangweiwells/article/details/101820932
            self.up_conv = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, mid_channels, out_channels)
        else:
            # channel / 2 for add concat channel
            # stride-1 = the pixel between 2 source pixel
            # kernel_size - padding - 1 = padding number
            # Height = (s*(height-1)+1)+(k-p-1)*2 - (k-1) = s*height + k - 2*p - s, Width same method
            # ref: https://blog.csdn.net/qz\_37541097/article/details/120709865
            self.up_conv = nn.ConvTranspose2d(in_channels, out_channels=in_channels//2, kernel_size=(2,2), st
            self.conv = DoubleConv(in_channels, mid_channels, out_channels)

    def forward(self, x, concat):
        # (batch, channel, height, width)
        x = self.up_conv(x) # half
        concatenated_x = torch.cat((concat, x), dim=1) # concat img from contracting path, get high resolution f
        return self.conv(concated_x)

```

Up-convolution 的做法有兩個，一個是直接使用 Bilinear 插值的方式，把圖片直接放大，另一個是使用 Transposed Convolution 放大，Transposed Convolution 透過把原圖的 pixel 之間與外圍 padding 變大，再套入一般的 Convolution 的方式，把 padding 值與原值以學習過的 kernel 補成可以顯現特徵的值，並調整到正確的大小。

兩種方式各有好處，Bilinear 的優勢在於不需要學習，並且做起來更快速，所以減少參數量的同時，訓練時間可以很大程度的減少，缺點就是 model 的 capacity 會下降，可能會導致沒辦法找出很好的解。Transposed Convolution 則是用更多的參數來學習，訓練的會更慢，但是可以提升 model 的 capacity，適合再應對更複雜的資料時使用。

之後，用到 UNet 的一大特色: concat connection，把 contracting path 中得到的高解析度資料，直接 concat 到現有資料上，讓模型可以同時使用高解析度的資料與精煉過後的重要資料，再使用上述提到的 Double Convolution 做整合，就可以把資料做更好的特徵提取。

接著把 Upsampling 作好幾層，就可以組成 expensive path 了，實作如下，其中除以 factor 的部分是用來應對 bilinear 並不會改變 feature channel 深度的狀況，做出的深度調整。


```

self.constracting_path_part1 = DoubleConv(in_channels=in_channels, mid_channels=64, out_channels=64)
self.constracting_path_part2 = DownSampling(in_channels=64, mid_channels=128, out_channels=128)
self.constracting_path_part3 = DownSampling(in_channels=128, mid_channels=256, out_channels=256)
self.constracting_path_part4 = DownSampling(in_channels=256, mid_channels=512, out_channels=512)
self.constracting_path_part5 = DownSampling(in_channels=512, mid_channels=1024//factor, out_channels=1024//factor)

self.expansive_path_part1 = UpSampling(in_channels=1024, mid_channels=512//factor, out_channels=512//factor)
self.expansive_path_part2 = UpSampling(in_channels=512, mid_channels=256//factor, out_channels=256//factor)
self.expansive_path_part3 = UpSampling(in_channels=256, mid_channels=128//factor, out_channels=128//factor)
self.expansive_path_part4 = UpSampling(in_channels=128, mid_channels=64, out_channels=64, bilinear=bilinear)
self.output = nn.Sequential([
    nn.Conv2d(in_channels=64, out_channels=n_class, kernel_size=(1,1), padding_mode='reflect', padding=0),
])

```

(1) ResNet34-UNet

這個網路模型是把 ResNet34 與 UNet 的 expansive path 拼在一起，一個做為 encoder 另一個做為 decoder 的形式。以下將會先就 ResNet34 做解釋，再提他們是怎麼合在一起的。

- ResNet34

ResNet 全稱 Residual Net，其使用了一個稱為 residual mapping 的方法，用來解決 degradation 的問題。residual mapping 的方法是建立一個“shortcut connection”，直接把幾個 layer 前的資料用 identity 的方式原封不動加回幾個 layer 的輸出，如下圖。Layer 學習的東西就會從一般的輸出，變成學習輸出與原本資料的差異，也就是所謂的 residual。

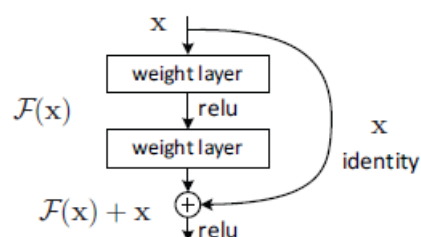
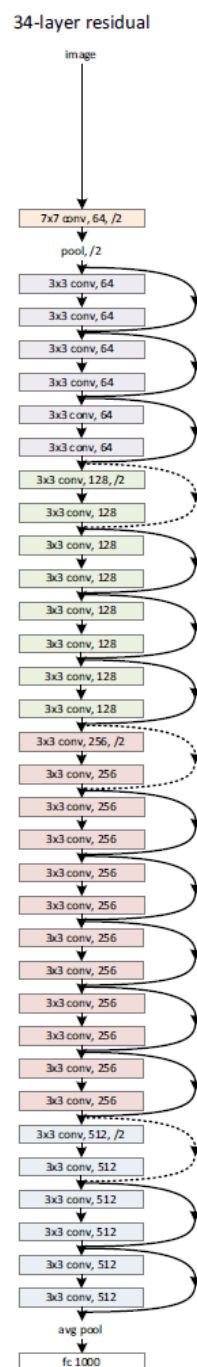


Figure 2. Residual learning: a building block.

學習 Residual 的好處在於，一般的神經網路在加深之後，可能不需要這麼深就會有解，所以理論上的最佳解釋後面的幾層都做恆等映射。但是對於網路來說要學出恆等映射本身是困難的，這就導致一個在少量層數有最佳解的問題，在層數加深之後，反而得到的解更差，或至少得花更多時間去訓練出恆等

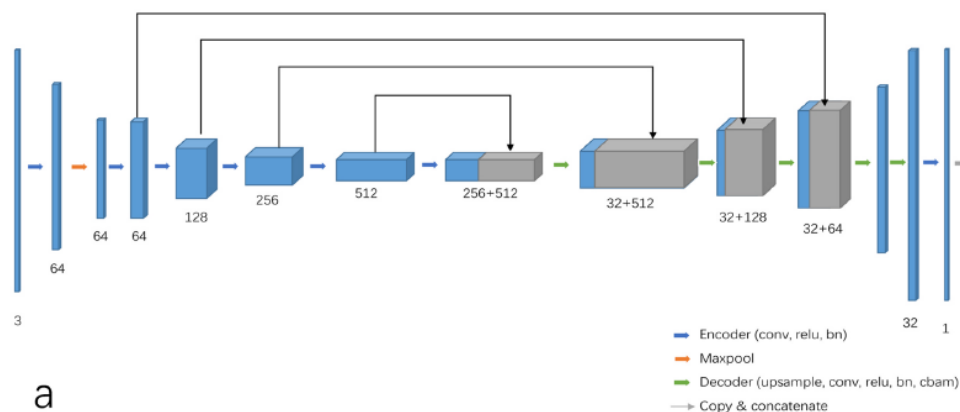
映射，這個問題稱為 degradation 問題。而如果 layers 學習的是 residual，那要達到恆等映射只要讓 residual 歸零就好，這對於網路而言是相對容易學習的，尤其用了 ReLU 之類的 activation function，只要是負數的任意值，都可以做到，因此降低了 degradation 問題的影響，可以讓網路變得更多層。

ResNet34 就是用了 residual mapping 方法的 CNN，其總共有 34 層，架構圖如下所示：



這個架構跟 UNet 的 contracting path 非常像，因此就直接被拿來取代 contracting path 作為 encoder 使用。

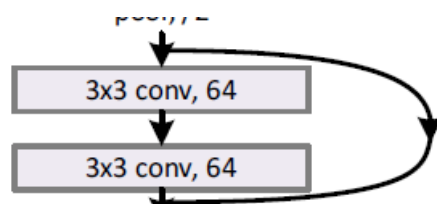
- ResNet34-UNet



如圖所示，左邊 encoder 的部分是 ResNet34，圖中畫出了 ResNet34 的 6 個部分，其中可以再對應到上面 ResNet34 的詳細圖。右側則是 UNet 的 expansive path，把 ResNet 每個 part 的資料再 concat 過去就成功地把它拼在一起了。

- 實作

首先從 ResNet 的基本單元做起，基本單元如下圖，由 2 個 Convolution layer 構成，在這兩個 Convolution 的前後連了一條 shortcut connection。



實作於下圖，在 Convolution Layer 的後面一樣會加上 Normalization 與 ReLU，分別處理梯度的消失、爆炸與 non-linear 的問題。接著建立一個 shortcut connection，實務上因為會有 down sampling 的狀況，所以視條件判斷是否要用 Convolution 來調整 residual mapping 映射過去的資料的大小及 feature channel 深度。另外也實作了 make_layer function 用來疊起大數量的 ResBlock。

```

class ResBlock(nn.Module): # layers with one shortcut connection
    def __init__(self, in_channels, out_channels, downsampling=False):
        super().__init__()
        self.downsampling = downsampling

        stride = 2 if downsampling == True else 1 # when block in tail of one part, use stride 2 to scale down
        self.model = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=(3,3), stride=1, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=(3,3), stride=stride, padding=1),
            nn.BatchNorm2d(out_channels),
        )
        if downsampling == True: # shortcut will be 2 times larger than x, use conv to make shortcut smaller.
            self.shortcut_connection = nn.Sequential(
                nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=(3,3), stride=2, padding=1),
                nn.BatchNorm2d(out_channels),
                nn.Identity()
            )
        elif in_channels != out_channels: # image will be same, channel will be changed...
        else:
            self.shortcut_connection = nn.Identity()
        self.relu = nn.ReLU()

    def make_part(self, n_block, in_channels, out_channels, downsampling = True):
        part = nn.Sequential()
        part.append(ResBlock(in_channels, out_channels, downsampling=downsampling))
        for i in range(n_block-1):
            part.append(ResBlock(out_channels, out_channels))
        return part

```

得到 ResBlock 之後，接著把 UNet 的 UpSampling 拿過來做一點修改，主要修改了 UNet 會先把原資料的 feature channel 砍半再拿來 concat 的部分，以及某些 UpSampling 沒有做資料 concat，只有單純的 UpSampling。

```

class UpSampling(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels, bilinear=True): # bilinear set
        super().__init__()

        if bilinear == True:
            self.up_conv = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, mid_channels, out_channels)
        else:
            self.up_conv = nn.ConvTranspose2d(in_channels, out_channels=in_channels, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, mid_channels, out_channels)
        # self.norm = nn.BatchNorm2d(in_channels)

    def forward(self, x, concat=None):
        # (batch, channel, height, width)
        if concat != None:
            concatenated_x = torch.cat((concat, x), dim=1) # concat img from contracting path, get 2x channel
        else:
            concatenated_x = x
        up = self.up_conv(concated_x) # half
        # up = self.norm(up)
        return self.conv(up)

```

接著把各個組件分配對應的參數，初始化在網路中

```

self.Res_part1 = nn.Sequential(
    nn.Conv2d(in_channels=in_channels, out_channels=64, kernel_size=(7,7), stride=2, padding=3),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(3,3), stride=(2,2), padding=(1,1) ) # scale down 2
)
self.Res_part2 = self.make_part(3, in_channels=64, out_channels=64, downspmping=False)
self.Res_part3 = self.make_part(4, in_channels=64, out_channels=128) # scale down 2
self.Res_part4 = self.make_part(6, in_channels=128, out_channels=256) # scale down 2
self.Res_part5 = self.make_part(3, in_channels=256, out_channels=512) # scale down 2
self.Res_part6 = self.make_part(1, in_channels=512, out_channels=256, downspmping=False)
self.expansive_path_part1 = UpSampling(in_channels=256+512, mid_channels=32, out_channels=512)
self.expansive_path_part2 = UpSampling(in_channels=32+256, mid_channels=32, out_channels=256)
self.expansive_path_part3 = UpSampling(in_channels=32+128, mid_channels=32, out_channels=128)
self.expansive_path_part4 = UpSampling(in_channels=32+64, mid_channels=32, out_channels=64)
self.expansive_path_part5 = UpSampling(in_channels=32, mid_channels=32, out_channels=32, b

```

最後如下圖，把組件串在一起，把對應的 concat 對象作為參數傳入，就完成了 ResNet34-UNet 的建構。

```

def forward(self, x):
    ...
    c1
    c2
    c3
    c4
    c5
    c6
    ...

    c1 = self.Res_part1(x)
    c2 = self.Res_part2(c1)
    c3 = self.Res_part3(c2)
    c4 = self.Res_part4(c3)
    c5 = self.Res_part5(c4)
    c6 = self.Res_part6(c5)

    d1 = self.expansive_path_part1(c6, c5)
    d2 = self.expansive_path_part2(d1, c4)
    d3 = self.expansive_path_part3(d2, c3)
    d4 = self.expansive_path_part4(d3, c2)
    d5 = self.expansive_path_part5(d4)
    output = self.output(d5)

```

3. Data Preprocessing

A. How you preprocessed your data?

Data 的 preprocess 上，我主要實驗了兩個方法，第一個方法是在 UNet 論文中寫到的 Elastic Deformation，他可以模仿物理上的彈性變化，把圖片做形變再用 Bicubic 插值法把值填上，以此來擴增資料集比較少見的變化。第二個方法是 Random Crop，隨機切割圖片上的一部份並放大成原圖片的大小，這算是一個在

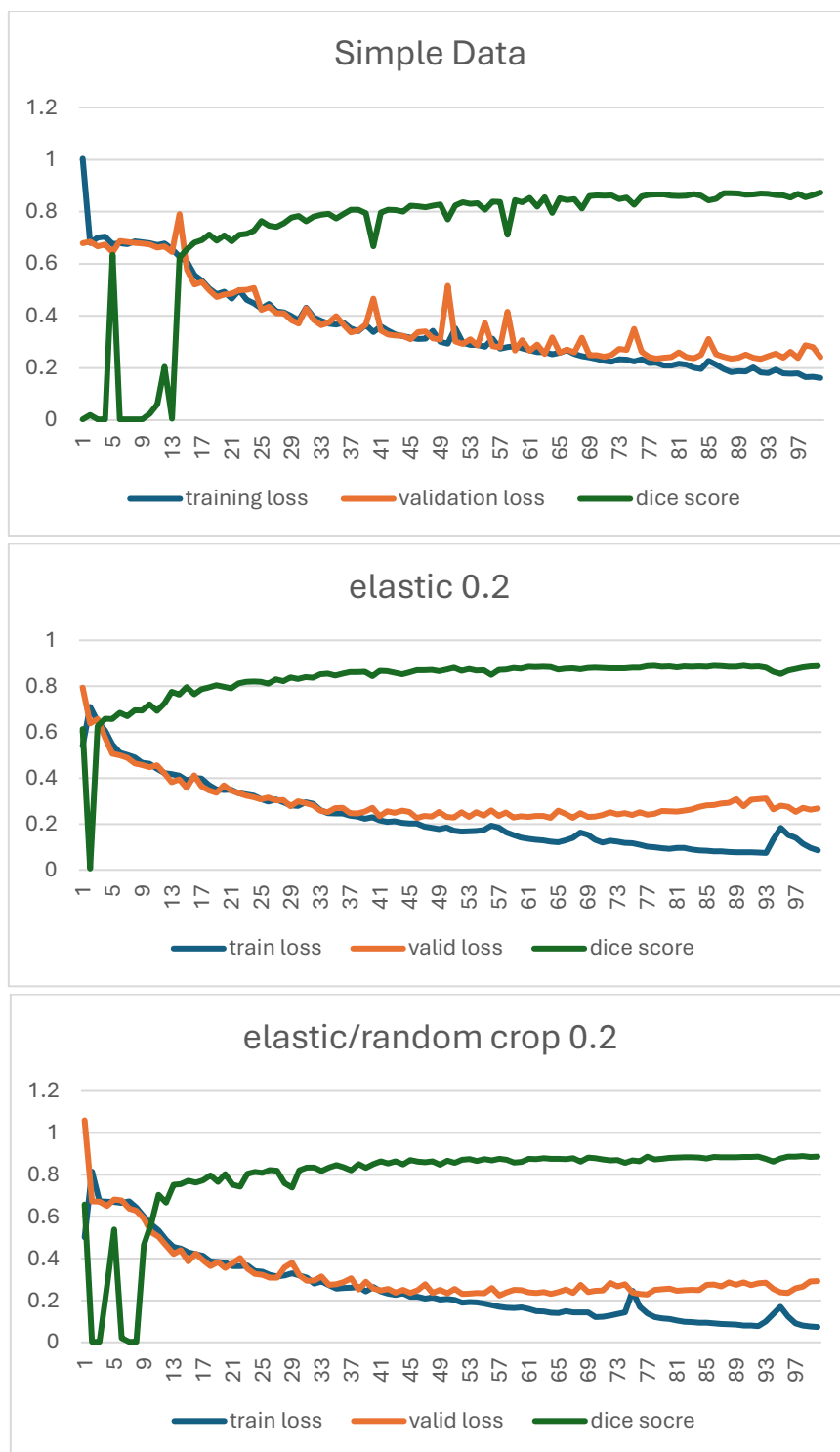
segmentation 任務上常見的 data augmentation 方法，透過隨機切割圖片可以讓模型針對某些部分的特徵能有更好的學習，而不是總學習一個整體。另外，我也有嘗試使用水平翻轉與垂直翻轉等方法，但初步試驗中，發現這兩個方法對於整體的 evaluation loss 並沒有提升，反而有下降的情況，因此把這兩種方法排除。

實作如下圖，使用 torchvision 的 transform，在每次 dataset getitem 時，就會套用轉換。MyTransforms 的 class 則是用來讓他能同時轉換 image, mask, trimap。

```
training_data_transform = My_transforms(transforms.Compose([ # 因為放在Compose外面，所以image, mask, trimap
    transforms.Resize((256,256)),
    transforms.RandomChoice([
        #transforms.RandomHorizontalFlip(p=0.2),
        #transforms.RandomVerticalFlip(p=0.2),
        transforms.RandomApply([
            transforms.ElasticTransform(alpha=30, interpolation=transforms.InterpolationMode.BICUBIC)
        ], p=0.2),
        transforms.RandomApply([
            transforms.RandomCrop((256,256)),
        ], p=0.2),
    ]),
    transforms.Resize((256,256)),
    ]),
)
```

B. What makes your method unique?

首先看到底下三張實驗結果，模型為 Res34-UNet，其分別代表:1. 沒有套用 data augmentation、套用以 0.2 機率 elastic deformation、以 0.2 機率套用 elastic deformation 或 random crop。分別依 0.001 learning rate 使用 Adam optimizer 訓練 100 epoch 的結果圖。

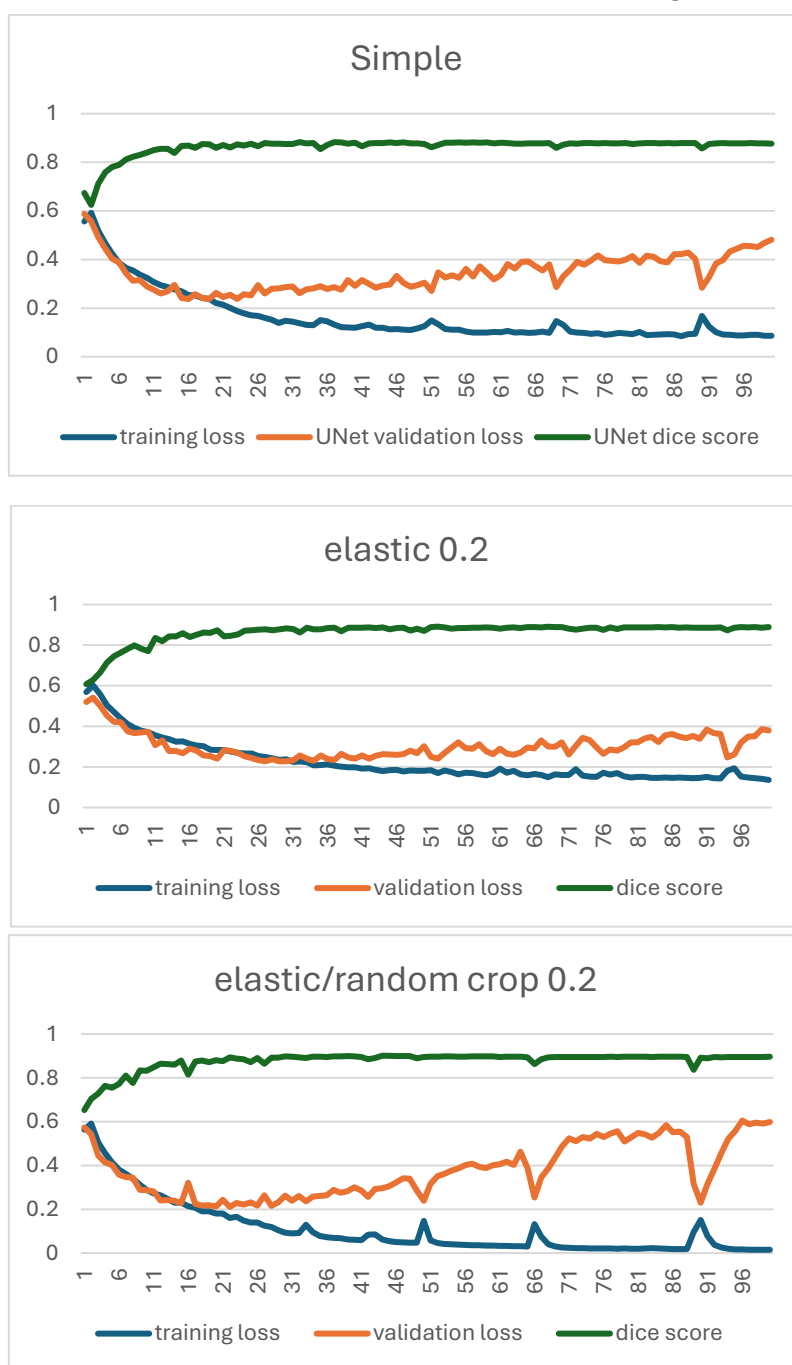


可以看出，三者最後收斂的 dice score 與 validation loss 並沒有太大的差別。但是使用了 data augmentation 的組別，validation 下降的速度卻比沒使用 data augmentation 的更快，代表透過 data augmentation 訓練讓模型的泛化能力更快的成長，可以用更少的運算資源得到相同的效果。

另外，data augmentation 理論上應該要增加模型的泛化效果的上限，讓 validation loss 應該要更低於沒有使用 data augmentation。但這裡並沒有這個

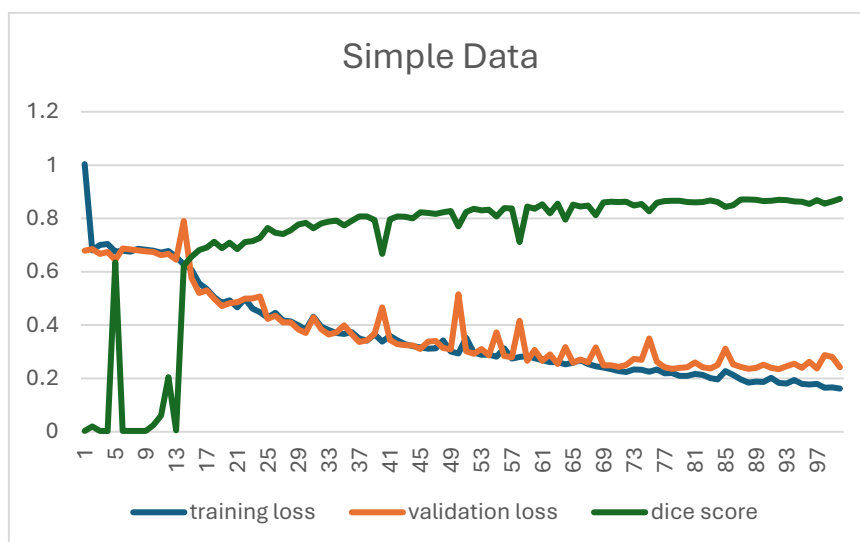
現象，我認為是因為我採用的方式是，直接用形變後的圖片取代原圖，而不是多複製一份的方式，因此在形變之後，會少了一份原本的訓練數據，形成一個 trade-off，讓整體只有下降速度變快，卻沒有降更低。針對這個問題，之後可以改良 data augmentation 的方式，把形變之前的結果多複製一份作訓練。

另外，以下 3 張圖為 UNet 套用不同 data augmentation 的結果，得到的結果基本上跟 Res34-UNet 相同，不過收斂之後 overfitting 的狀況更加明顯一點。

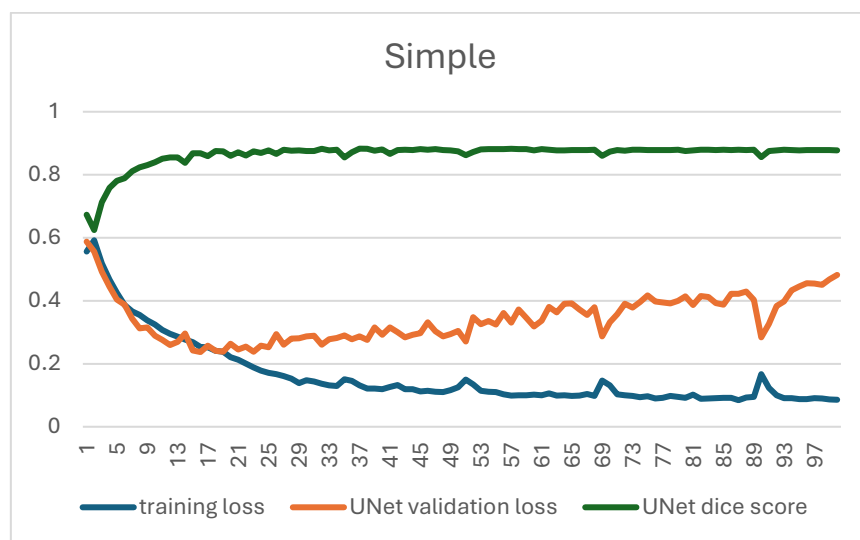


4. Analyze on the experiment results

A. What did you explore during the training process?



(Res34-UNet)



(UNet)

上圖與 3-B 的圖相同，從中可以發現在訓練這個資料集時，很容易產生 overfitting 的問題，也就是 training loss 與 validation loss 的兩條線開始叉開，當兩條線開始叉開時，dice score 就會陷入停滯狀態。因此，我發現在

訓練過程中，阻礙 dice score 上升的關鍵因素在於模型容不容易 overfitting，因此若要改善分數，可以優先從這個方面著手。

B. Found any characteristics of the data?

在實驗的過程中，發現 data 很容易進入到 overfitting 的狀態，也就是 training loss 不斷降低，validation loss 卻收斂在偏高的數值。到後面也會有雖然 Dice Score 沒有下降，但是 validation loss 卻不斷上升的狀況，可以看出在 segmentation 的訓練上，如果 overfitting，除了精準度下降之外，可能導致的另一個結果是模型的信心程度越來越低。

5. Execution command

我的 hyper parameter 是直接寫在 training 與 testing 的 code 中，透過 dictionary 來做傳遞的，所以 command 上都比較簡單。Command 如下：

Training: python3 src/train.py

Inferencing: python3 src/inference.py

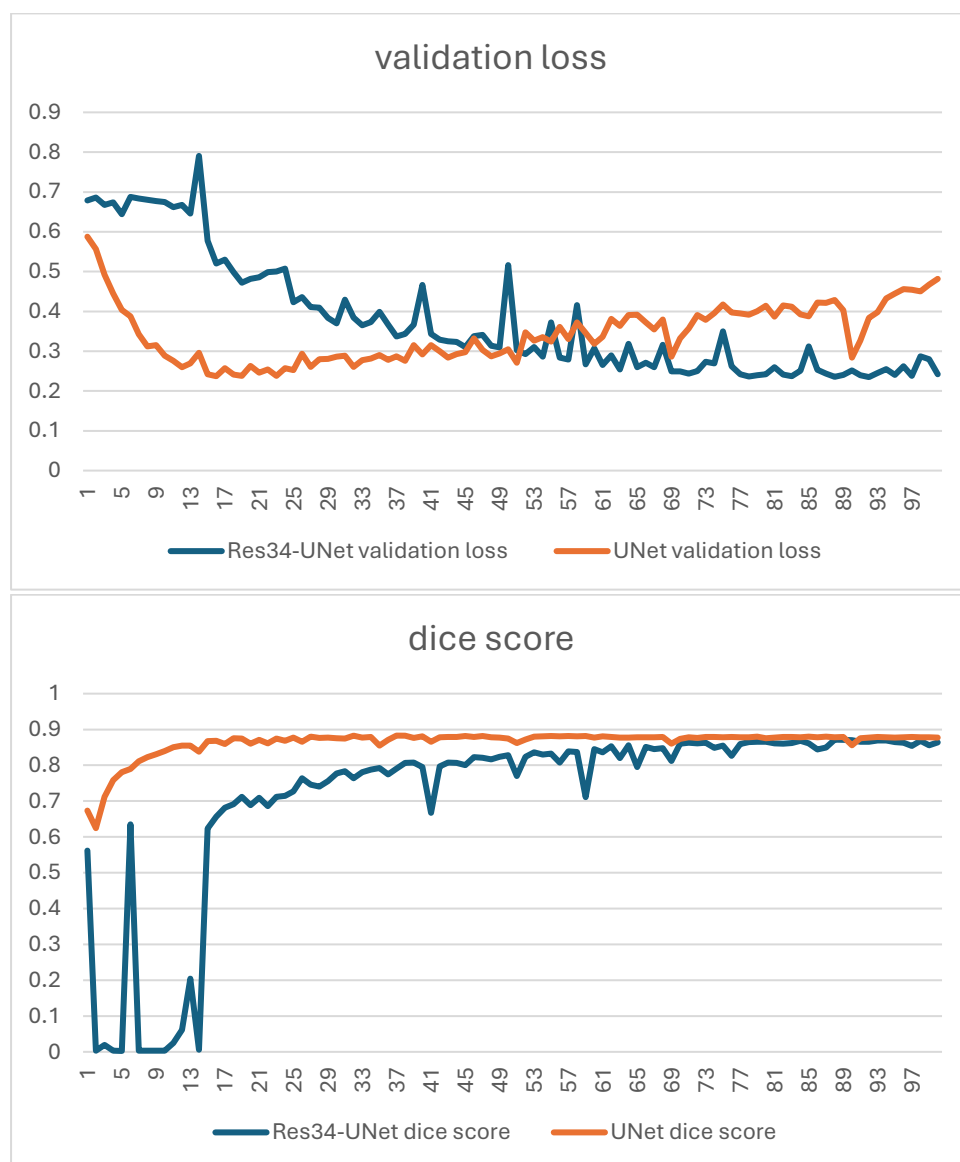
Hyper parameter 的調整則是直接在 function 內調整：

```
hyper_parameter = {  
    'n_epoch': 200,  
    'batch_size': 20, # 3312 samples for train  
    'learning_rate': 0.001,  
    'regularization': 0,  
    'bilinear': True  
}
```

6. Analyze on the experiment results

A. What architecture may bring better results?

以下是 Res34-UNet 與 UNet 在 learning rate 0.001, Adam optimizer · 訓練 100 個 epoch 的結果圖。



從圖中可以觀察到，UNet 的 validation loss 與 dice score 的收斂速度都比 Res34-UNet 更快。兩者的 dice score 最終收斂在差不多的地方。不過 Res34-UNet 的 validation loss 卻收斂的更低，代表 Res34-UNet 能給出更有信心的答案，整體來看，Res34-UNet 的訓練效果應該更好。並且雖然 UNet 只花更少的 epoch 就可以達到收斂，但是 UNet 訓練一個 epoch 的速度比 Res34-UNet 更慢，占用的 GPU 記憶體也更多。因此在訓練到同樣收斂的結果時，其實兩者花的時間差不多。

因此可以做出結論，在兩者比較上，Res34-UNet 的模型訓練出來的結果更好，透過 residual mapping 來加深模型，確實讓模型的 capacity 更大，可以分類的更精準一點。

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

針對基於 UNet 的圖片 segmentation 任務，我認為潛在的研究議題有三：Encoder、Data Augmentation、Model Depression。

Encoder:

在這次的 LAB 中，第二個 task 是把 ResNet 與 UNet 作為 Encoder、Decoder 拼接在一起，而我也第一次知道可以用這樣的方式來組合不同的模型。因此，我認為可以嘗試不同的圖片壓縮模型作為 Encoder，諸如：InceptionNet、MobileNet 等等，讓模型在準度、效能上可能有不同的優化，甚至可以套用 Transformer 來做 Encoder，看看 Seq2Seq 的模型跟 UNet 配合的效果怎麼樣，能不能超越傳統上的 CNN-based 模型。

Data Augmentation:

這次 LAB 中，因為時間的關係，我只能簡單實驗 4 種不同的 Data Augmentation 方式，不過這次我遇到了明顯的 Overfitting，因此我認為可以在 Data Augmentation 上面做更多的實驗，包括我這次未能做到的角度旋轉、翻轉等等的操作，試著讓模型的 generalization 能力上升。

Model Depression:

這次的 LAB 中，我遇到的一個大問題是，顯卡的記憶體用來訓練 UNet 等模型並不是很夠用，很容易不小心就爆記憶體。實際上或許解決 segmentation 任務可以不需要這麼多的參數。因此我認為可以針對模型的壓縮著手，尋找方式壓縮模型的參數量，或者是替換成 MobileNet 之類的更節能的 Encoder 等等。