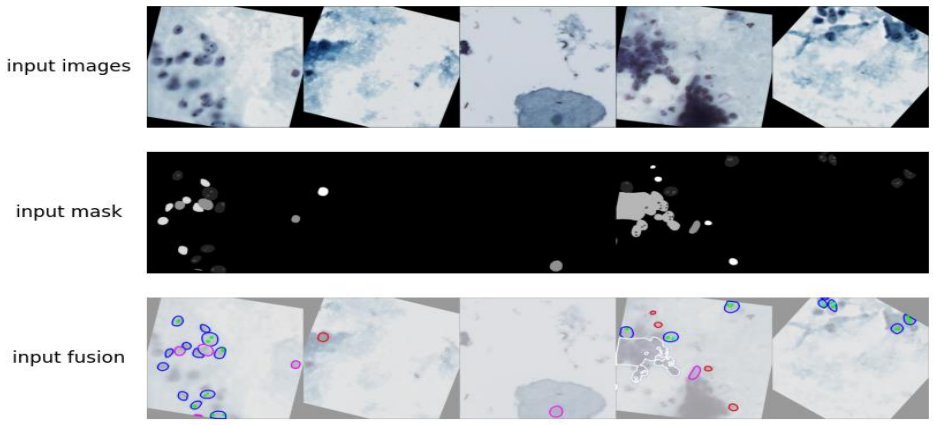
## DataLoader :

Dataloader 的部分，我針對 A~O 的 class 每個 class 都分成 7:3，分別加進 trainset, testset，確保 dataset 裡每個 class 平均分配，再對 trainset, testset 做 shuffle 確保每次 training 拿資料的順序不同，每個 iter 根據檔案名稱去取用每張 IMG。此外還發現如果用 os.listdir 去取用檔案名稱的話會 cpu 沒辦法快速處理完，這樣會拖累 GPU 的使用效率。

## Preprocess :

資料前處理的部分，我使用了 albumentations 函式庫，他可以同時對兩個不同的 data 做"一模一樣"的 transform，這樣可以使 IMG 跟 MASK 做完 transform 不會切在不同的位置。

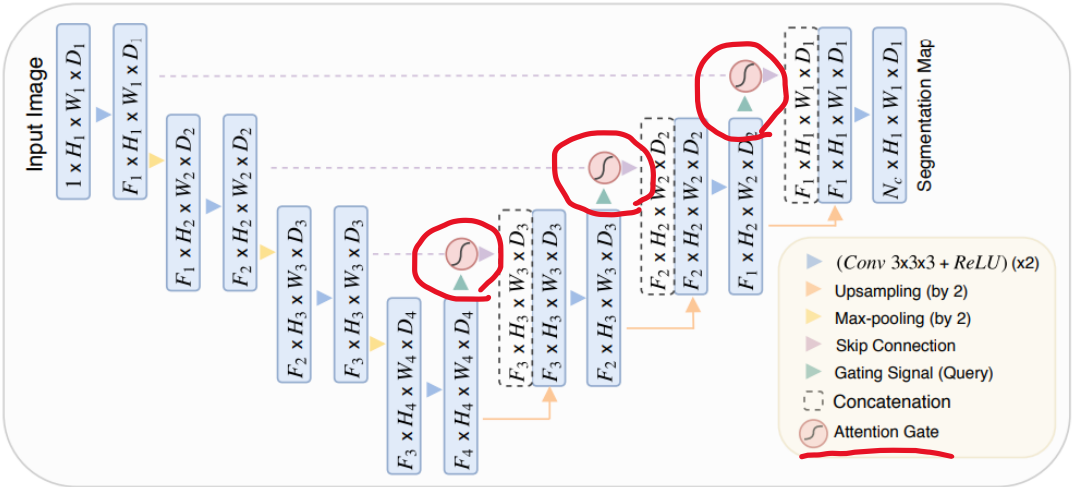
其中又發現如果今天圖片做完前處理後，若 batch 裡剛好取到很多個 mask 裡面沒有感興趣的地方(EX:全黑)的圖片，在訓練的時候會造成前後 batch 出來的結果差很多，導致\最後 loss 變得異常，所以在做 transform 的時候會檢查 MASK 裡是否有感興趣位置，若沒有的話會重新 transform 直到 MASK 裡有感興趣位置。



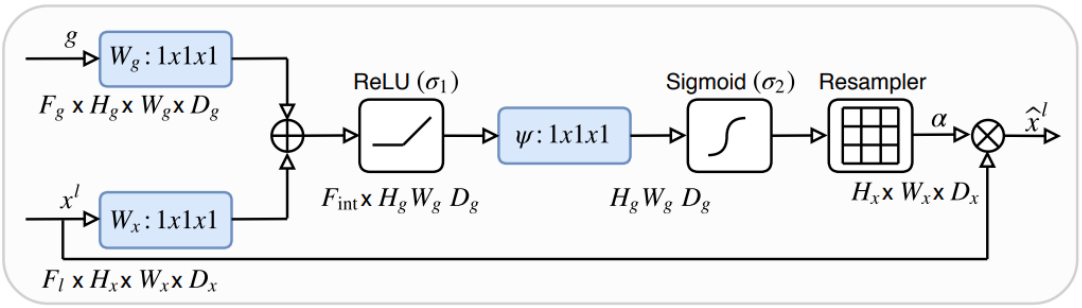

模型架構(一)

## UNet :

模型架構的部分，我使用了 pytorch-UNet 的模型，並參考了 attention-Unet 的架構(如圖二)自己改良成這次的 model，在每一次 concate 前加上了 mask-Attention-gate，這大大的提升了模型的 performance。




模型架構(二)




attention Gate (圖三)

Attention gate:

DownSmple layer($X^L$)，Upsample layer ($G$ )，$X^L$與$G$ 分別會先各自經過一個 CNN(維度,img size 不變)，兩者相加之後原本 MASK 有值的地方會變得更大，經過一個 Relu layer 後會過濾掉負值，經過一個 CNN 和一個 sigmoid layer 可以得到一個針對每個 pixel 的 attention score，最後再將這個 attention score 乘回原本的 mask，這個動作相當於利用 attention score 去加強我們所感興趣的部分。實際上的 implement 如圖四。

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

        self.same_conv1 = nn.Conv2d(out_channels, out_channels, kernel_size=1)
        self.same_conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=1)
        self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)


    def forward(self, g, k):
        #q.size = k.size/2
        g = self.up(g)
        k = self.same_conv1(k)
        a = g+k
        a = F.relu(a)
        a = self.same_conv2(a)
        a = F.sigmoid(a)
        a = a*k

        return a
```

Attenrion gate implement (圖四)

## Show Batch

圖片 show 的部分參考了 https://github.com/tumeng0302/lab5_utlities.git 的程式碼

## Loss

Loss 的部分除了用 cross entropy，我還加了 dice loss，加了這個之後 loss 下降變慢許多，performance 也變好許多。

```python
def multiclass_dice_coeff(input: Tensor, target: Tensor, reduce_batch_first: bool = False, epsilon: float = 1e-6):
    # Average of Dice coefficient for all classes
    return dice_coeff(input.flatten(0, 1), target.flatten(0, 1), reduce_batch_first, epsilon)


def dice_loss(input: Tensor, target: Tensor, multiclass: bool = False):
    # Dice loss (objective to minimize) between 0 and 1
    fn = multiclass_dice_coeff if multiclass else dice_coeff
    return 1 - fn(input, target, reduce_batch_first=True)
```

DICE loss (圖五)

## Result:

由於電腦壞掉所以只有少少的訓練 epoch 和 result，來不及做 matrix 比較。

圖片裡 gt vs pred 如果 pred 的結果跟 gt 有重疊(預測正確)就顯示白色，若沒有重疊(預測錯誤)就顯示紅色。

最後我只跑了 50 個 epoch，其中發現一開始 avg loss 降得很快(第一個 epoch 就跑到 0.03 左右)，後來都卡在這附近且降得很緩慢，但是 performance 卻有越來越好的趨勢，後來我認為是 avg loss 是看所有的 batch，但每一個 batch 算出的 loss 還是有大有小，模型還是會根據大的 loss 去更新，所以還是會越來越好。

其中也發現，訓練到越後面模型才開始比較能學會對 class 分類正確，訓練不多的情況下他只會學到 segment。