Deep Learning: Lab 2 Binary Semantic Segmentation

長庚大學 B1126006 郭兆揚

Implementation Details

For **train.py**, I use a function, $training(\dots)$, to implement the training of U-Net and ResNet34 with U-Net decoder. And it needs the parameters below:

- $mode\_type$: to determine which type of model it being training
- $data\_path$: path to folder storing data
- $batch\_size$: how much data will be loaded a time by dataloader
- epochs: the maximum epoch numbers (when early stop feature doesn't be activated)
- lr: learning rate
- aug_type: the method to augment training dataset
- n_aug: the times data being augmented
  $\text{len(auged\_dataset)} = (1 + \text{n\_aug}) * \text{len(original\_dataset)}$

While getting in the $training(\dots)$ function, it will be based on $mode\_type$ to initial encounter model and the loss function $criterion$ which U-Net needs $CrossEntropyLoss$ require by its architecture in its original paper and its outputs have 2 channels is multi-class (foreground / background) classification , and for ResNet34_Unet is $BinaryCrossEntropyLoss$ for its outputs have only 1 channel which match sigmoid and threshold to determine foreground and background.

And initialize encounter training data based on aug_type and n_aug, and use dataset and $batch\_size$ to initialize the dataloader. Then, initialize $AdamW$ optimizer and $early\_stop$ object for encounter overfitting, and it has features below:

- stop training loop while validation loss didn't get lower than the historical lowest validation loss in continuous 7 epochs. PS: "7" is the trade-off of the timesaving (for my GPU has 8G VRAM only) and not mis-assert the training loop which may have chance to get better.
- Remember the $best\_model\_weight$ while a new weight reaches the lowest validation loss.

Getting into the training loop, the $early\_stop$ will check if it should activate in every epoch. For the traversal data loop, it must use $optimizer.zero\_grad()$ to clean the gradient and avoid the accumulation of gradient in previous iteration. And change the model to train mode, then take the image data and ground truth mask from element dataloader gave. The format of the masks will be different for $nn.CrossEntropyLoss$ of Pytorch on accept the masks should be 3D tensor, so I need to squeeze it into 3D tensor. Then, the model gives its predictions, and $loss\ function$ use the predictions and ground truth masks to calculate the loss. And calculate the gradients by backpropagation executed $loss.backward()$ then update the weights of model by $optimizer.step()$. After traversal all the sample in training dataset, using $evaluate(\dots)$ function to calculate the pixel-wise accuracy (dice score defined by Lab 2 spec.) and loss of the model on validation dataset. And use the loss of validation to check if the early stop mechanism should be activated. While reaching the limit of epochs or early stop is been activated, the weights of model that with smallest validation loss will be saved.

For **evaluate.py**, I also use a function, $evaluate(\dots)$ to calculate the sum of running validation loss and accuracy, it has parameters below:

- net: the model entity being used
- $criterion$: the loss function model used
- $val\_dataloader$: dataloader of validation dataset
- net_type: The type of the net / model

In $evaluate(\dots)$, it will switch the model to $eval$ mode, and use $torch.no\_grad()$ to close the computation of gradients. Then calculate the average loss and accuracy of each batch, and add to them to $val\_running\_loss$ and $val\_running\_dice$. Finally, return them back.

For **inference.py**, I implement a function, $infer(...)$, to check the ability of model / calculate the dice score of model on testing dataset. It has parameters below:

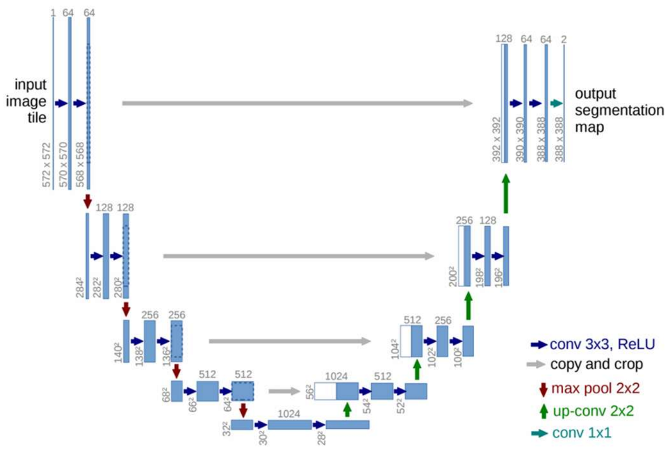
- $model\_name$: name of the model in $saved\_models$ folder wants to be test
- $model\_type$: type of the model
- $batch\_size$: how much data will be loaded a time by dataloader

- $data\_path$: the root directory of data

It will initialize the dataloader of testing dataset and initialize the model base on model type. Then, load the saved weights of model in $saved\_models$ folder base on $model\_name$. Then, calculate the average dice score in each batch and get the final average dice score of all batch, which is the outcome of inference.

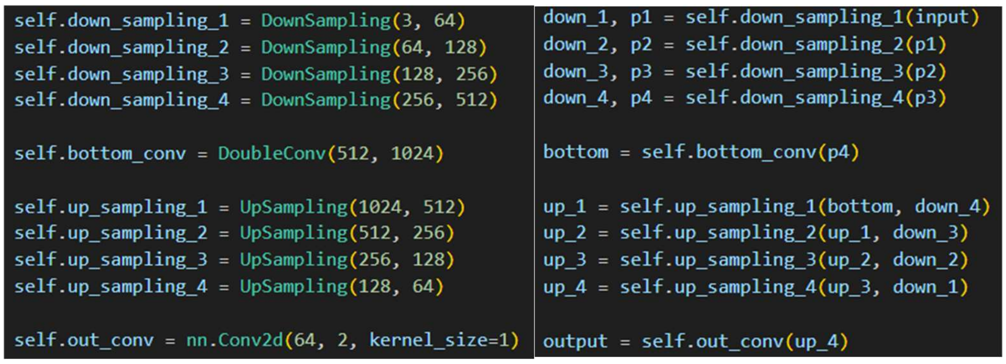
For the part of **models**, I construct my models based on the 3 papers in Lab 2, the architectures of models are quite like the original, but with some modifications in them for evaluating the model prediction accurately and conveniently (output size = mask size).

Talk about **U-Net** first, figure of model architecture shows that there were many double convolutions (consequence blue arrows) in network. So, I build a class name $DoubleConv$ as the double convolutions, including 2 of convolution layers + ReLU in each $DoubleConv$. In the convolution layers, I modify the padding number to ensure that the size of masks and network output remain the same.



architecture of U-Net

Then, I build the classes $DownSampling$ and $UpSampling$ use the class $DoubleConv$ and encounter operations. $DownSampling$ is for the net encoding step, so I implement it combines the double convolution and pooling operation, and let it return the value for down sampling (p) and the value for copy and crop $(crop)$. For $UpSampling$, I implement it combines the de-convolution (transpose convolution) and concatenate operation and return the value for up sampling. Additionally, I build the $bottom$ as the bottom structure of the U-Net architecture by a double convolution and a single convolution for output.

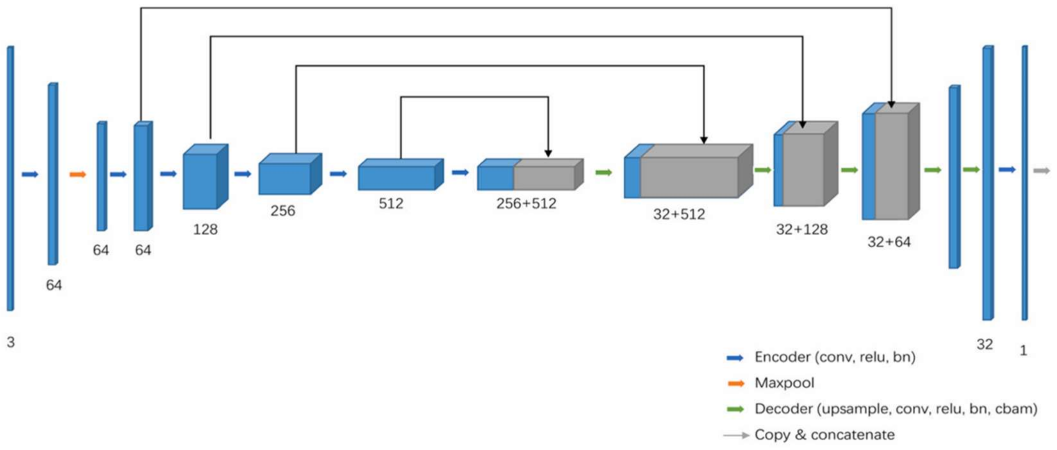


```
self.down_sampling_1 = DownSampling(3, 64)
self.down_sampling_2 = DownSampling(64, 128)
self.down_sampling_3 = DownSampling(128, 256)
self.down_sampling_4 = DownSampling(256, 512)

self.bottom_conv = DoubleConv(512, 1024)

self.up_sampling_1 = UpSampling(1024, 512)
self.up_sampling_2 = UpSampling(512, 256)
self.up_sampling_3 = UpSampling(256, 128)
self.up_sampling_4 = UpSampling(128, 64)

self.out_conv = nn.Conv2d(64, 2, kernel_size=1)
```

```
down_1, p1 = self.down_sampling_1(input)
down_2, p2 = self.down_sampling_2(p1)
down_3, p3 = self.down_sampling_3(p2)
down_4, p4 = self.down_sampling_4(p3)

bottom = self.bottom_conv(p4)

up_1 = self.up_sampling_1(bottom, down_4)
up_2 = self.up_sampling_2(up_1, down_3)
up_3 = self.up_sampling_3(up_2, down_2)
up_4 = self.up_sampling_4(up_3, down_1)

output = self.out_conv(up_4)
```

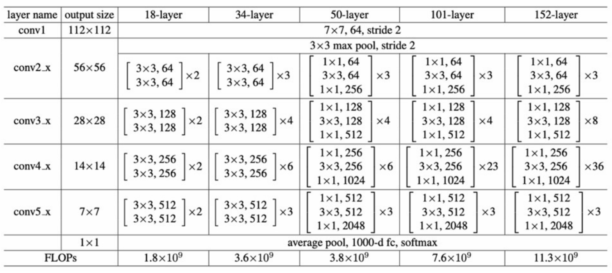
architecture of U-Net in code

For the part of $\boldsymbol{ResNet34}\ +\ \boldsymbol{UNet}$, the architecture of model in that paper is not very clear, but I still implement it according to the figure paper give.

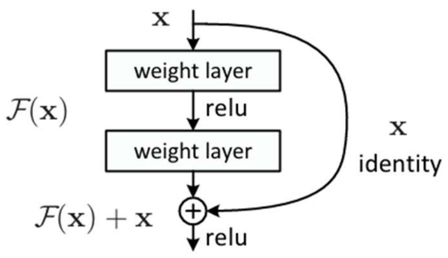


architecture of $ResNet34\ +\ UNet$

The encoder is ResNet34. According to the original paper of ResNet34, it has a structure called $Block$ which contains convolution, ReLU, batch normalizing and identity mapping showing repeatedly.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
x
identity

architecture of $ResNet34$ and $Block$

The $Block$ of ResNet34 contains 2 convolutions with 3x3 kernels, batch normalizations and identity mapping operations. As the operation in U-Net, I also use padding to ensure the output of network will not shrink. And I also notice that the size of identity mapping might have problem while the down sampling happens which will shrink the size and increase the number of channels of feature maps by convolutions with more $out\_channels$ compared to $in\_channels$ and stride 2. So, I use the method mentioned in the original paper to fix this problem, using a convolution with 1x1 kernel and stride 2 to re-map the identity, then can connect to the feature map after down sampling correctly. As for situations that are not down sampling, it can execute those operations with same $out\_channels$ and $in\_channels$, fixed size of feature maps. And for $conv1\_x$ parts, I implement it by same configuration with original paper but adding $padding = 3$ to ensure the size of feature being halved exactly (without other size shrinking).

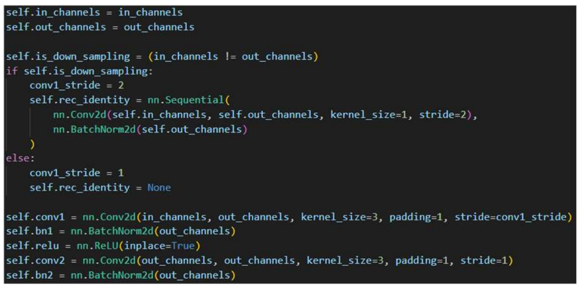
The $padding = 3$ is from:

$$S_{out} = \frac{S_{in} - k + 1 + 2P}{stride} \;; \quad k: kernel\ size, P: number\ of\ padding$$

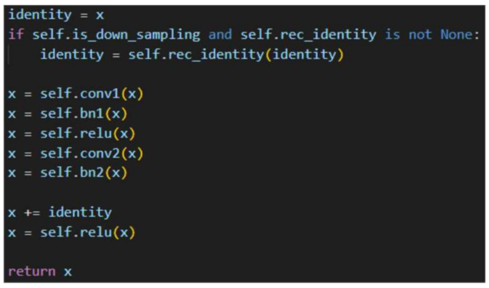
Now $k = 7; stride = 2$, want $S_{out} = \frac{S_{in}}{2}$ exactly:

$$2P = 7 - 1$$

$$P = 3$$



```
self.in_channels = in_channels
self.out_channels = out_channels

self.is_down_sampling = (in_channels != out_channels)
if self.is_down_sampling:
    conv1_stride = 2
    self.rec_identity = nn.Sequential(
        nn.Conv2d(self.in_channels, self.out_channels, kernel_size=1, stride=2),
        nn.BatchNorm2d(self.out_channels)
    )
else:
    conv1_stride = 1
    self.rec_identity = None

self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, stride=conv1_stride)
self.bn1 = nn.BatchNorm2d(out_channels)
self.relu = nn.ReLU(inplace=True)
self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, stride=1)
self.bn2 = nn.BatchNorm2d(out_channels)
```



```
identity = x
if self.is_down_sampling and self.rec_identity is not None:
    identity = self.rec_identity(identity)

x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.conv2(x)
x = self.bn2(x)

x += identity
x = self.relu(x)

return x
```

architecture of $Block$ in code

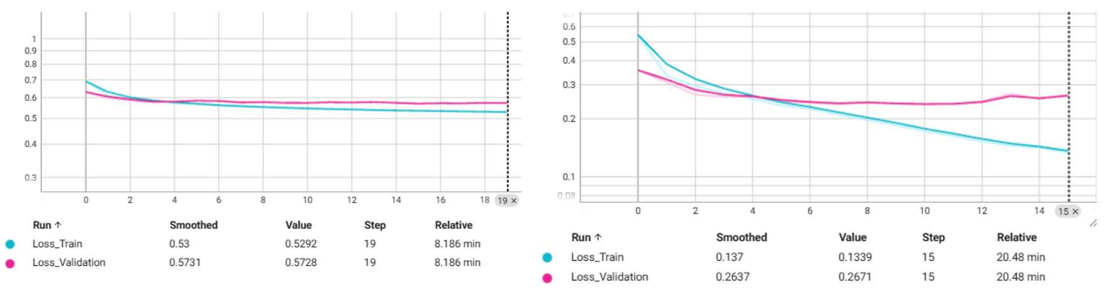
For pooling step, I implemented it by padding $= 1$ as well, other parts are still the same as the original architecture.

Then, I use the method $\_make\_resnet34\_down\_sampling$ in class $ResNet34\_UNet$ to build the layer from $conv2\_x$ to $conv5\_x$. The method will set the first $block$ as down sampling corresponding to architecture of ResNet34 and change the number of channels from $in\_channels$ to $out\_channels$. And build the rest of $block$ with same channels until reaching the n_block which is the number of blocks of this down sampling.

For the $bridge$ to connect the $ResNet34$ and $UNet$, I observe that the network will encode the channels to 512 to 256. So, I use a convolution with 1x1 kernel, batch normalization and ReLU to implement it. This position is the output layer of original $ResNet$ and will use softmax to get the results, but in here needs to be a feature maps for the up sampling operations, so I use similar structure of output layer operation in U-Net (a convolution) to mimic the function of this layer.

For the up sampling in $ResNet34 + UNet$, it only has two differences. One of them is to add batch normalizations in $DoubleConv$ structure that did not exist in original structure of U-Net. I consider the ResNet34 structure have this operation in each convolution operation, so I choose to implement it. Another is that I observe the one of copy and concatenate step in architecture of ResNet34 + UNet have scale its channels numbers from 256 to 512, so it uses a convolution with 1x1 kernel to implement the scaling operation.

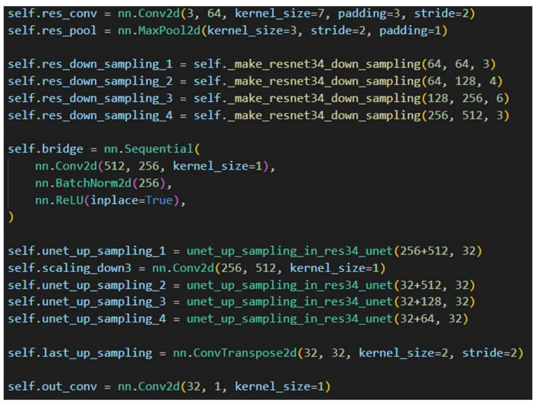
For the output layer, the **original design of ResNet34 + UNet is using convolution with 1x1 kernel, batch normalization and ReLU.** However, ReLU will cut the negative part of logit, it means that while calculating the **loss by Binary cross entropy, the loss will not be able to get down correctly**. For this, I try to rectify it by changing the design into a single convolution with 1x1 kernel to fix this problem.



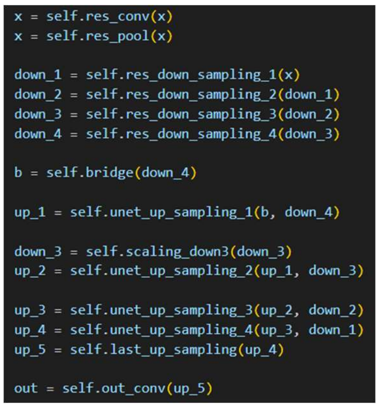
Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.53	0.5292	19	8.186 min
● Loss_Validation	0.5731	0.5728	19	8.186 min

Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.137	0.1339	15	20.48 min
● Loss_Validation	0.2637	0.2671	15	20.48 min

Loss of $ResNet34\_UNet$ with/ without ReLU in output layer



```python
self.res_conv = nn.Conv2d(3, 64, kernel_size=7, padding=3, stride=2)
self.res_pool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

self.res_down_sampling_1 = self._make_resnet34_down_sampling(64, 64, 3)
self.res_down_sampling_2 = self._make_resnet34_down_sampling(64, 128, 4)
self.res_down_sampling_3 = self._make_resnet34_down_sampling(128, 256, 6)
self.res_down_sampling_4 = self._make_resnet34_down_sampling(256, 512, 3)

self.bridge = nn.Sequential(
    nn.Conv2d(512, 256, kernel_size=1),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),
)

self.unet_up_sampling_1 = unet_up_sampling_in_res34_unet(256+512, 32)
self.scaling_down3 = nn.Conv2d(256, 512, kernel_size=1)
self.unet_up_sampling_2 = unet_up_sampling_in_res34_unet(32+512, 32)
self.unet_up_sampling_3 = unet_up_sampling_in_res34_unet(32+128, 32)
self.unet_up_sampling_4 = unet_up_sampling_in_res34_unet(32+64, 32)

self.last_up_sampling = nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2)

self.out_conv = nn.Conv2d(32, 1, kernel_size=1)
```

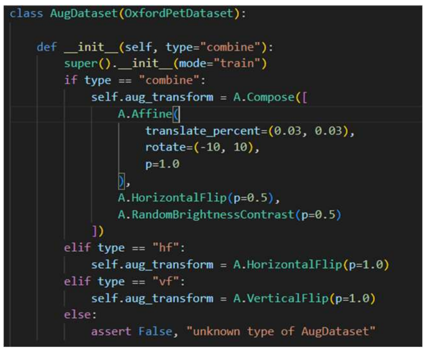


```python
x = self.res_conv(x)
x = self.res_pool(x)

down_1 = self.res_down_sampling_1(x)
down_2 = self.res_down_sampling_2(down_1)
down_3 = self.res_down_sampling_3(down_2)
down_4 = self.res_down_sampling_4(down_3)

b = self.bridge(down_4)

up_1 = self.unet_up_sampling_1(b, down_4)

down_3 = self.scaling_down3(down_3)
up_2 = self.unet_up_sampling_2(up_1, down_3)

up_3 = self.unet_up_sampling_3(up_2, down_2)
up_4 = self.unet_up_sampling_4(up_3, down_1)
up_5 = self.last_up_sampling(up_4)

out = self.out_conv(up_5)
```

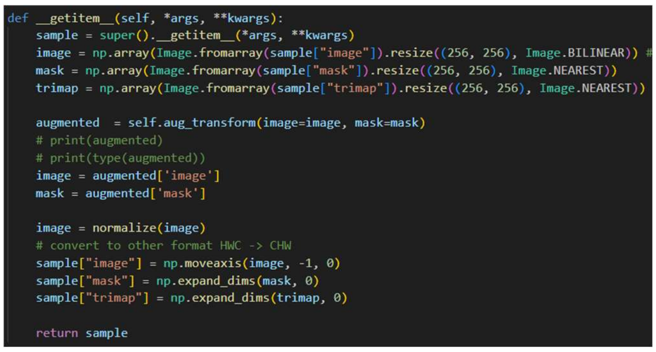
architecture of *ResNet34 + UNet* in code

Data Preprocessing

Beside reshaping the images, I implemented the z-score normalization to promote the efficiency of learning and avoid the effect from magnitude different between images. And I also implemented **2 types of data augmentations by adding some transform on a dataset to dynamically augment the data**. One of them is to add **a little rotation, displacement and flip randomly** to augment the data. This way is like taking an image again from another angle or position. But I also worried about the difference between augmented data and original data are too small, making this operation is like so how "increasing the number of epochs". So, I implemented another method to argument data. To **flip all images in training dataset horizontally or vertically**, this way can avoid the situations that augmented data almost have no difference.



```python
class AugDataset(OxfordPetDataset):

    def __init__(self, type="combine"):
        super().__init__(mode="train")
        if type == "combine":
            self.aug_transform = A.Compose([
                A.Affine(
                    translate_percent=(0.03, 0.03),
                    rotate=(-10, 10),
                    p=1.0
                ),
                A.HorizontalFlip(p=0.5),
                A.RandomBrightnessContrast(p=0.5)
            ])
        elif type == "hf":
            self.aug_transform = A.HorizontalFlip(p=1.0)
        elif type == "vf":
            self.aug_transform = A.VerticalFlip(p=1.0)
        else:
            assert False, "unknown type of AugDataset"
```



```python
def __getitem__(self, *args, **kwargs):
    sample = super().__getitem__(*args, **kwargs)
    image = np.array(Image.fromarray(sample["image"]).resize((256, 256), Image.BILINEAR)) #
    mask = np.array(Image.fromarray(sample["mask"]).resize((256, 256), Image.NEAREST))
    trimap = np.array(Image.fromarray(sample["trimap"]).resize((256, 256), Image.NEAREST))

    augmented  = self.aug_transform(image=image, mask=mask)
    # print(augmented)
    # print(type(augmented))
    image = augmented['image']
    mask = augmented['mask']

    image = normalize(image)
    # convert to other format HWC -> CHW
    sample["image"] = np.moveaxis(image, -1, 0)
    sample["mask"] = np.expand_dims(mask, 0)
    sample["trimap"] = np.expand_dims(trimap, 0)

    return sample
```

Augment data in code

I implement 2 interfaces to use the augmented data, to load the *combine* (with a little rotation, displacement and flip randomly) and *flip* (100% flip) augmented data. n_aug parameter of them means the number of augmented dataset being added, notice that $combine$ has no limit on n_aug. But n_aug in $flip$ have the limitation of 2, means flipping horizontally and vertically.

Analyze the experiment results

For the selections of hyperparameters, I use a strategy that uses **high epochs (50, 60), relatively lower learning rate (1e-3, 1e-4) and the early stop** to run the experiments. This way can balance the proper fit and time saving. The learning rate is from I observed the model can keep the training loss descending stably, which means model can learn the detail in this learning rate, and I chose 1e-3 to try for avoid the model will not stick in local minimal, 1e-4 is to ensure that the model can learn detail property.

And the number of epochs is also from experiments, I observe it the epoch after 30 show the phenomenon of overfitting, but I want to make sure that the validation loss will not cost down anymore. Then, I chose **60 epochs** as the numbers of epochs for time saving and enough quote for model to fit. However, there are seldom experiments that can run until the limitation of epochs because of early stop mechanism.

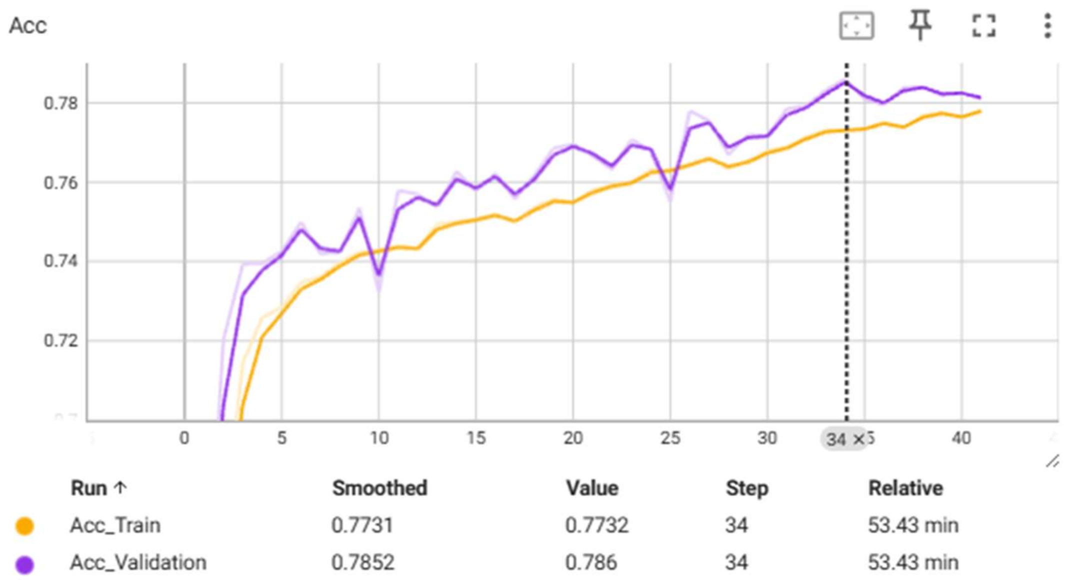
For data augmentation, I try different numbers of $\boldsymbol{n\_aug}$, $\boldsymbol{aug\_type} =$ "$\boldsymbol{combine}$", and $\boldsymbol{aug\_type} =$ "$\boldsymbol{flip}$" and **not augmented data by** $\boldsymbol{n\_aug = 0}$.

For $\boldsymbol{batch\_size}$**, it is fixed at** $\boldsymbol{15}$ because my GPU (4060ti) has the highest training speed in this batch size.

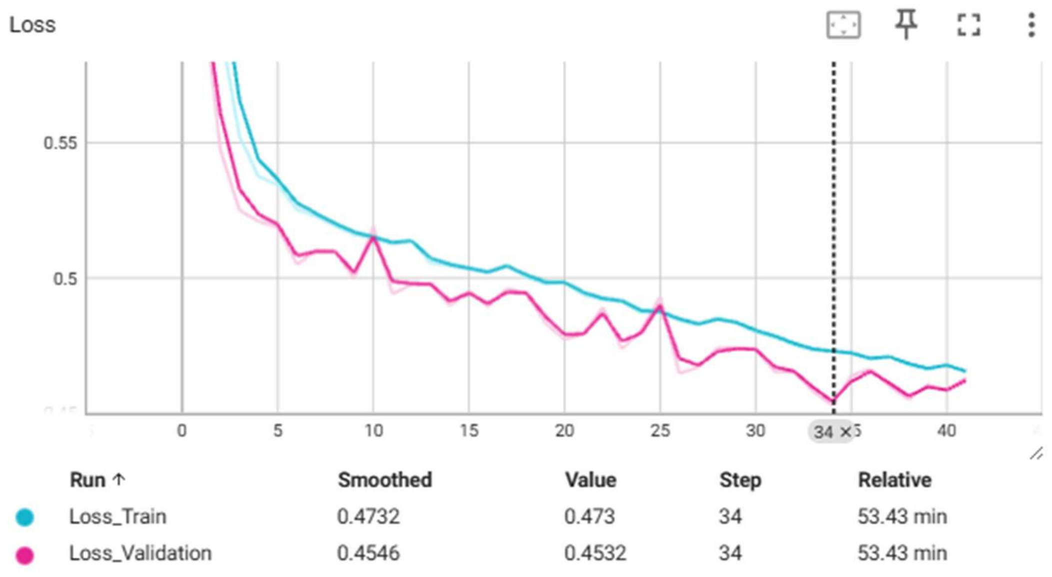
Then, here are the experiment results of each configuration on the $UNet$ and $ResNet34 - UNet$.

I try $lr = 1e - 3$ and with no data augmentation first, the results are below:

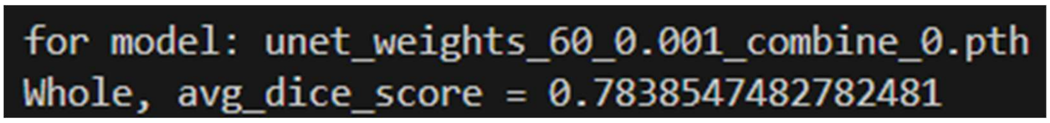
U-Net



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.7731	0.7732	34	53.43 min
● Acc_Validation	0.7852	0.786	34	53.43 min



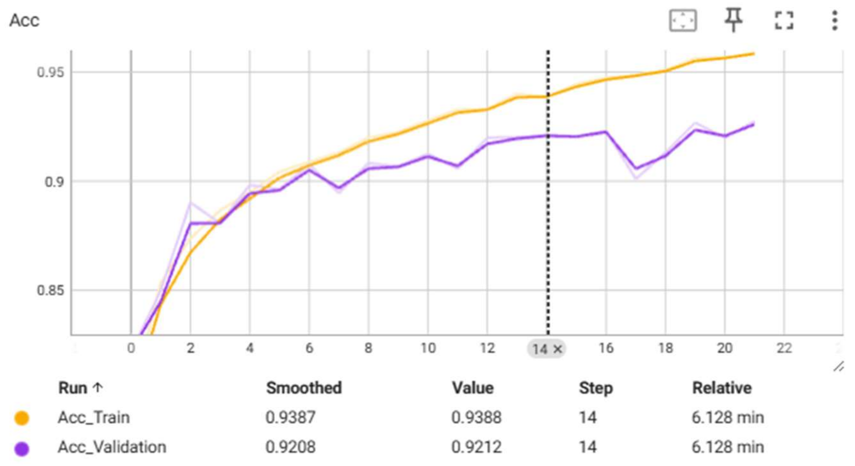
Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.4732	0.473	34	53.43 min
● Loss_Validation	0.4546	0.4532	34	53.43 min



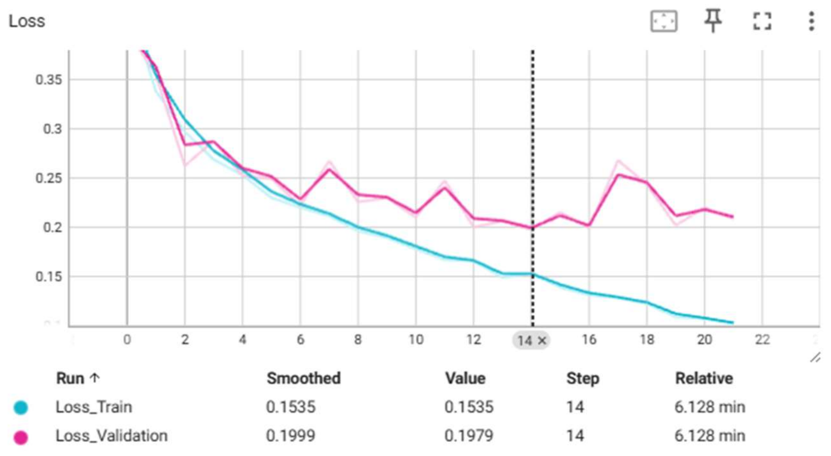
```
for model: unet_weights_60_0.001_combine_0.pth
Whole, avg_dice_score = 0.7838547482782481
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e - 3, no\ aug$) of UNet

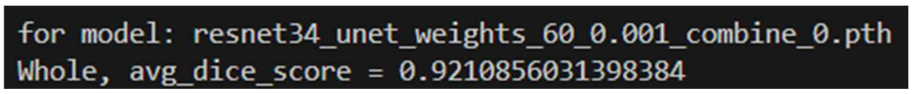
ResNet34_UNet



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.9387	0.9388	14	6.128 min
● Acc_Validation	0.9208	0.9212	14	6.128 min



Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.1535	0.1535	14	6.128 min
● Loss_Validation	0.1999	0.1979	14	6.128 min



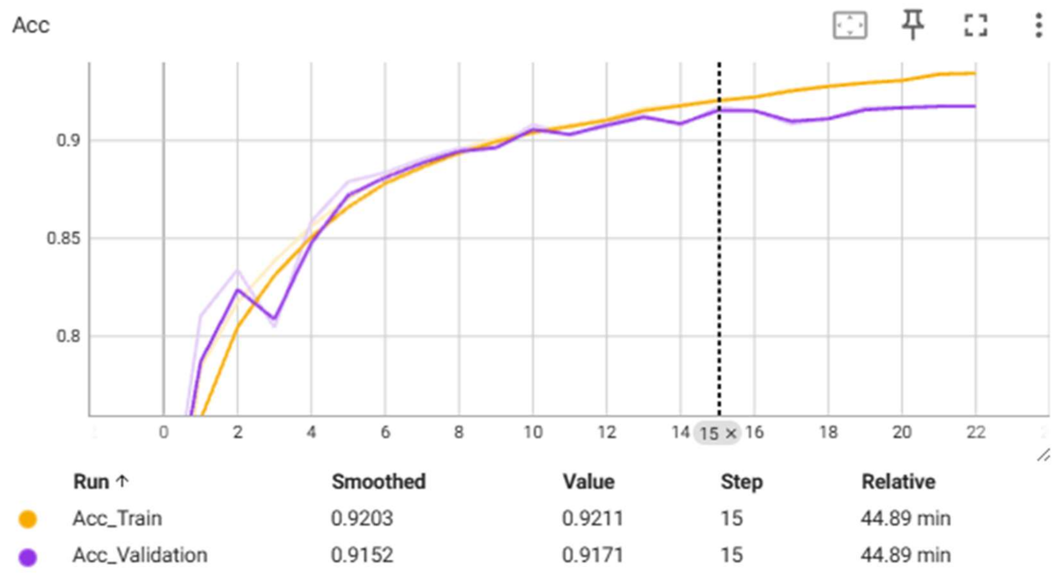
```
for model: resnet34_unet_weights_60_0.001_combine_0.pth
Whole, avg_dice_score = 0.9210856031398384
```

Acc / Loss on training / validation dataset and Dice score on test dataset with $(lr = 1e-3, no\ aug)$ of ResNet34_UNet

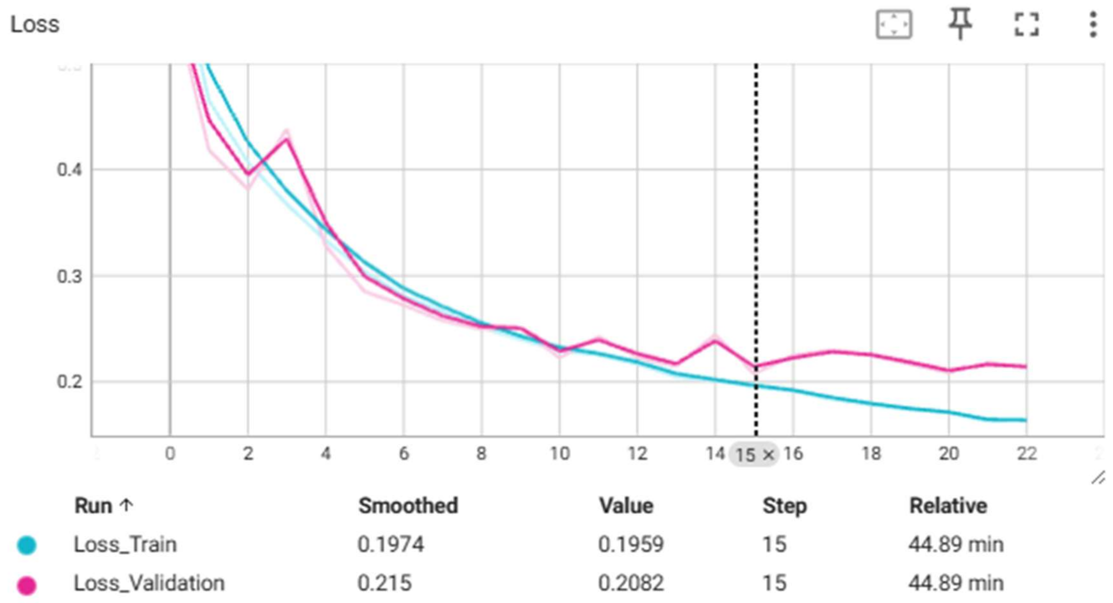
The results show that when $(lr = 1e-3, no\ aug)$, UNet cannot converge properly, the validation loss keeps high and oscillating (0.45) and has a low dice score (0.78). In comparison, ResNet34_UNet has low loss (0.21) and high dice score (0.92). I think it is because the learning rate was too large in this insufficient data. Unlike ResNet34_UNet has a skip connection to stabilize the learning process, **the connection of UNet is for reserving the high-resolution information, not for stablizing. So, while data is insufficient and the learning rate is too large, UNet will have bad performance.**

Then, I try to augment the data by $aug\_type = combine$; $n\_aug = 1$ to increase the amount of data. And the lr keep in $1e - 3$. And get the result below:

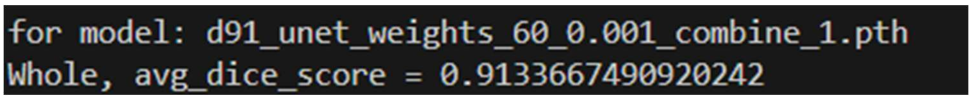
U-Net



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.9203	0.9211	15	44.89 min
● Acc_Validation	0.9152	0.9171	15	44.89 min



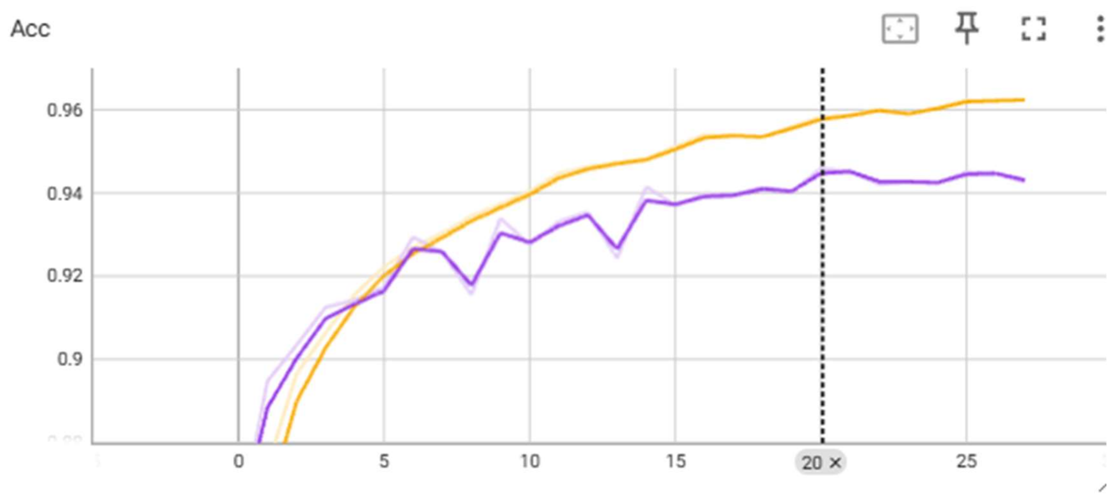
Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.1974	0.1959	15	44.89 min
● Loss_Validation	0.215	0.2082	15	44.89 min



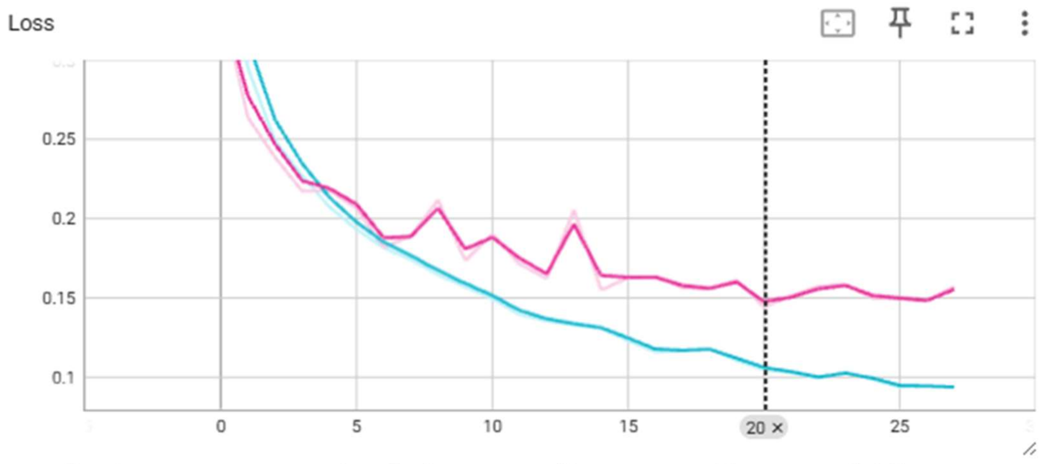
```
for model: d91_unet_weights_60_0.001_combine_1.pth
Whole, avg_dice_score = 0.9133667490920242
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e - 3, aug\_type = combine$; $n\_aug = 1$) of UNet

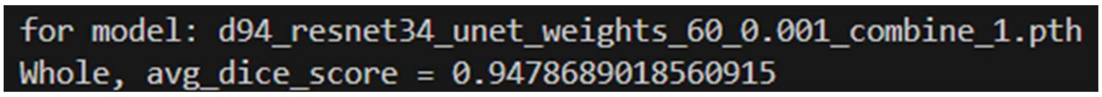
ResNet34_UNet



	Run ↑	Smoothed	Value	Step	Relative
●	Acc_Train	0.9577	0.9583	20	16.22 min
●	Acc_Validation	0.9447	0.9459	20	16.22 min



	Run ↑	Smoothed	Value	Step	Relative
●	Loss_Train	0.1066	0.1049	20	16.22 min
●	Loss_Validation	0.148	0.1445	20	16.22 min



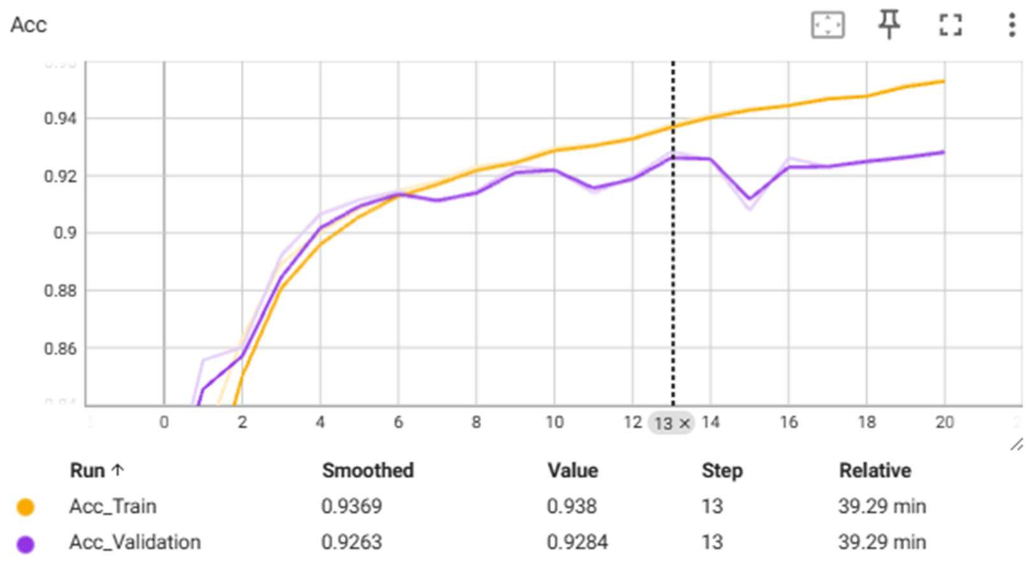
```
for model: d94_resnet34_unet_weights_60_0.001_combine_1.pth
Whole, avg_dice_score = 0.9478689018560915
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e-3, aug\_type = combine;\ n\_aug = 1$) of ResNet34_UNet

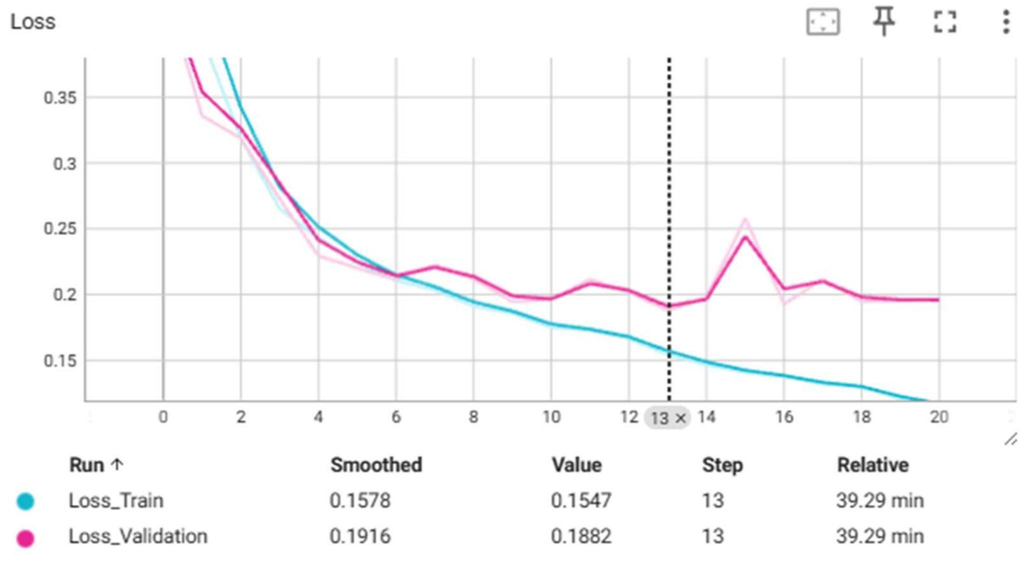
The result shows that there are **a lot of improvements by augmenting the data**. For UNet the validation loss can get down properly (0.2) and has a much higher dice score (0.91) compared to UNet without data augmentation. I think it is because the data is sufficient for UNet to learning the pattern behind the data under this learning rate. As for ResNet34_UNet, it also has some improvements, loss become 0.15 and dice score become 0.94. This shows the power of data augmentation.

After trying the $aug\_type = combine$ to augment data, I also use the different augment type $aug\_type = flip$ to check if it will have better performance.

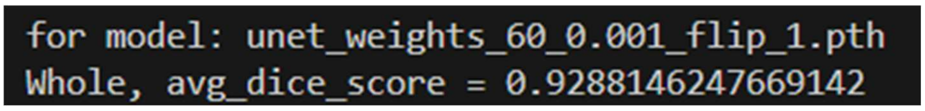
U-Net



	Run ↑	Smoothed	Value	Step	Relative
●	Acc_Train	0.9369	0.938	13	39.29 min
●	Acc_Validation	0.9263	0.9284	13	39.29 min



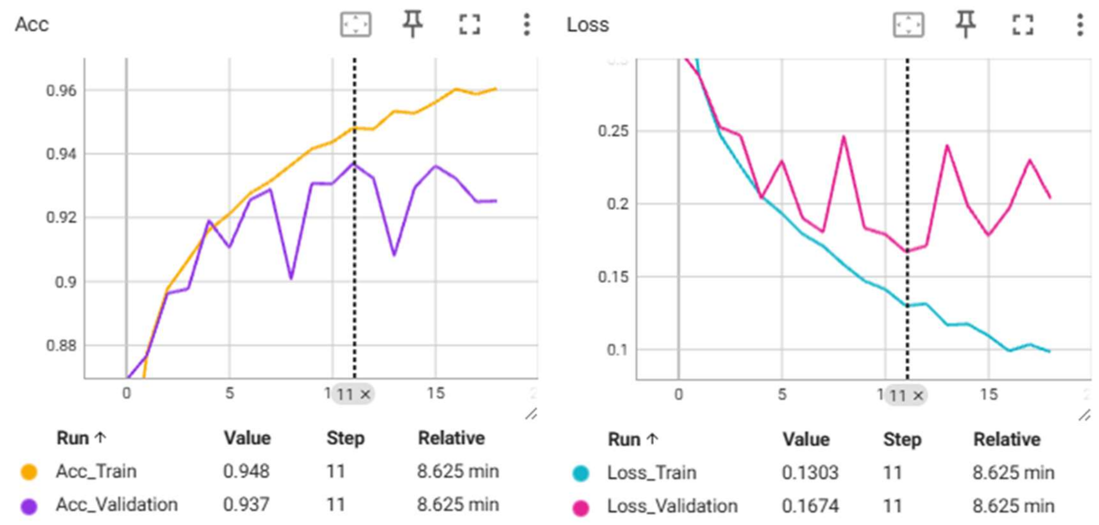
	Run ↑	Smoothed	Value	Step	Relative
●	Loss_Train	0.1578	0.1547	13	39.29 min
●	Loss_Validation	0.1916	0.1882	13	39.29 min



```
for model: unet_weights_60_0.001_flip_1.pth
Whole, avg_dice_score = 0.9288146247669142
```

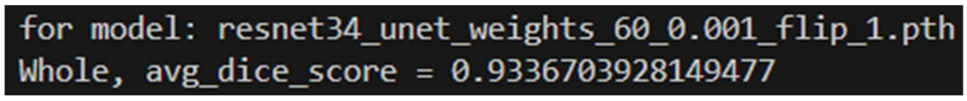
Acc / Loss on training / validation dataset and Dice score on test dataset with $(lr = 1e-3, aug\_type = flip;\ n\_aug = 1)$ of UNet

ResNet34_UNet



Run ↑	Value	Step	Relative
● Acc_Train	0.948	11	8.625 min
● Acc_Validation	0.937	11	8.625 min

Run ↑	Value	Step	Relative
● Loss_Train	0.1303	11	8.625 min
● Loss_Validation	0.1674	11	8.625 min



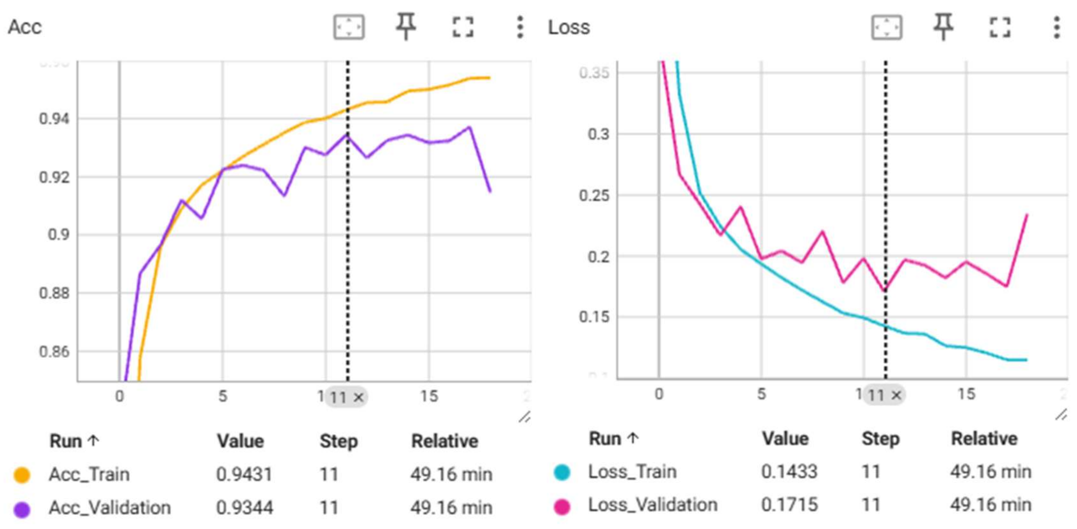
```
for model: resnet34_unet_weights_60_0.001_flip_1.pth
Whole, avg_dice_score = 0.9336703928149477
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e-3, aug\_type = flip;\ n\_aug = 1$) of ResNet34_UNet

The result of using $\boldsymbol{aug\_type = flip}$ to augment data shows similar dice score to situations that using $aug\_type = combine$. However, t**he loss of them is relatively higher and is oscillating for ResNet34_UNet, which means that the model might be less stable**. For UNet, the oscillating phenomenon was not so obvious. I think it is because the **flipping data augmentation changes the image a lot that the network with deep layers like ResNet34_UNet is hard to fit the varying data easily**. But **UNet has a shallower structure** compared to ResNet34_UNet, so **it can fit those varying of feature more easily**. By this experiment, I know the flipping data augmentation might cause some unstable of model because the too much change on feature.

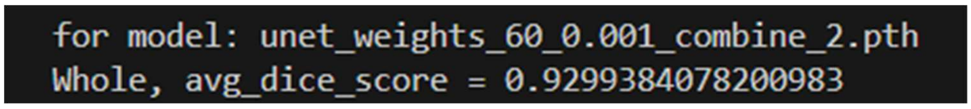
The previous experiment shows both $combine$ and $flip$ can improve the accuracy of models, but models on $combine$ augmented dataset show more stable. So, I try to increase the n_aug to get better model performance with $n_aug = 2, aug_type = combine$m and lr remain $1e-3$

U-Net



Run ↑	Value	Step	Relative
● Acc_Train	0.9431	11	49.16 min
● Acc_Validation	0.9344	11	49.16 min

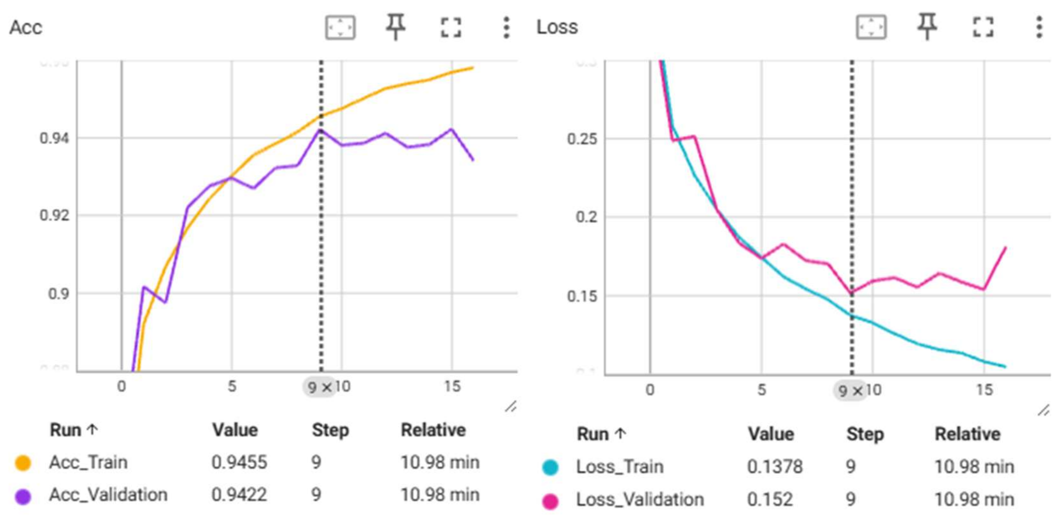
Run ↑	Value	Step	Relative
● Loss_Train	0.1433	11	49.16 min
● Loss_Validation	0.1715	11	49.16 min



```
for model: unet_weights_60_0.001_combine_2.pth
Whole, avg_dice_score = 0.9299384078200983
```

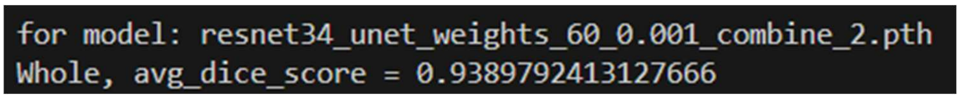
Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e-3, aug\_type = combine$; $n\_aug = 2$) of UNet

ResNet34_UNet



Run ↑	Value	Step	Relative
● Acc_Train	0.9455	9	10.98 min
● Acc_Validation	0.9422	9	10.98 min

Run ↑	Value	Step	Relative
● Loss_Train	0.1378	9	10.98 min
● Loss_Validation	0.152	9	10.98 min



```
for model: resnet34_unet_weights_60_0.001_combine_2.pth
Whole, avg_dice_score = 0.9389792413127666
```

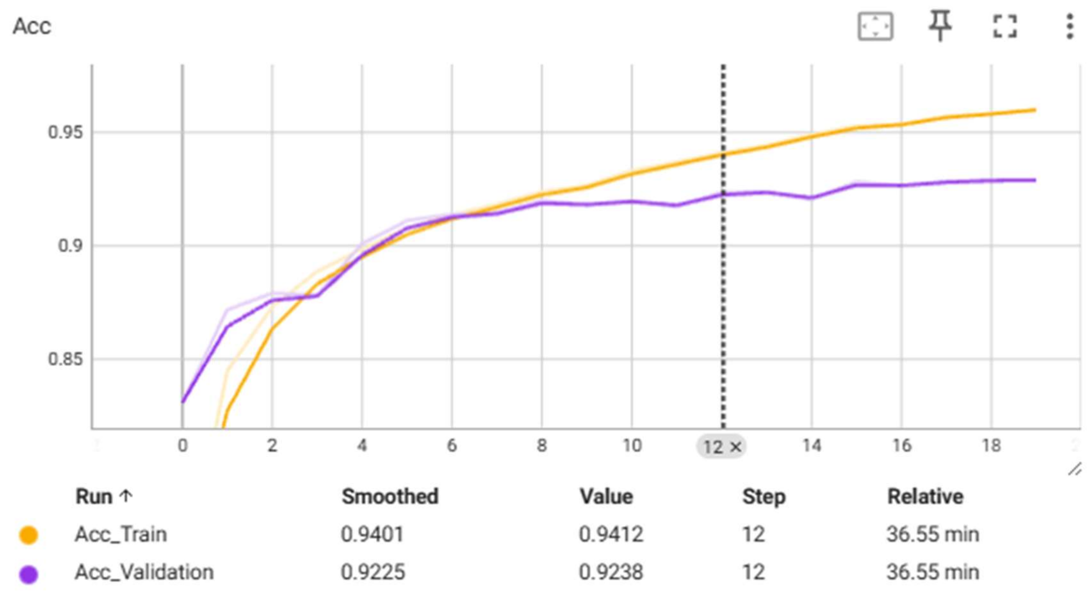
Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1e-3, aug\_type = combine;\ n\_aug = 2$) of ResNet34_UNet

After **using more data augmentation, the benefit of augmenting data becomes smaller for ResNet34_UNet.** ResNet34_UNet have no improvement both on loss and score. But UNet have prove its loss to 0.17 and prove its dice score to 0.93 compared to UNet with $n\_aug = 1$ (loss: 0.20, dice score: 0.91). I think this phenomenon is because the **UNet is extracting directly from images, so once there are more images, UNet can learn more. But ResNet34_UNet extract features by its deep convolutions structure, so the effect of augmenting more data might be diluted.**

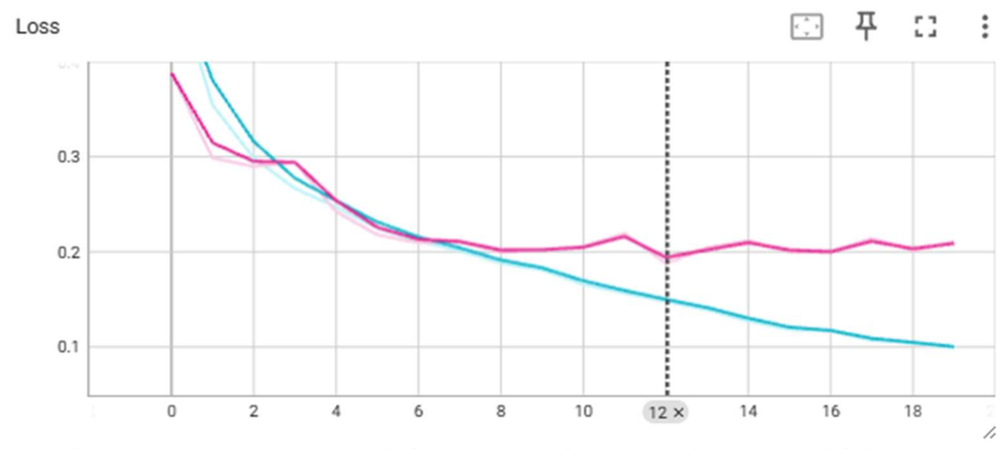
After testing the different augmented data, I try smaller learning rates to check if better results can be reached by using smaller learning rate to learn the detail of data. The experiment below has fixed $aug\_type = combine$ and $n\_aug = 1$.

For lr=1e-4,

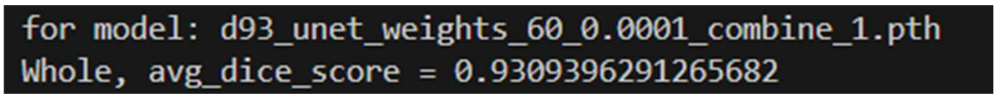
U-Net



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.9401	0.9412	12	36.55 min
● Acc_Validation	0.9225	0.9238	12	36.55 min



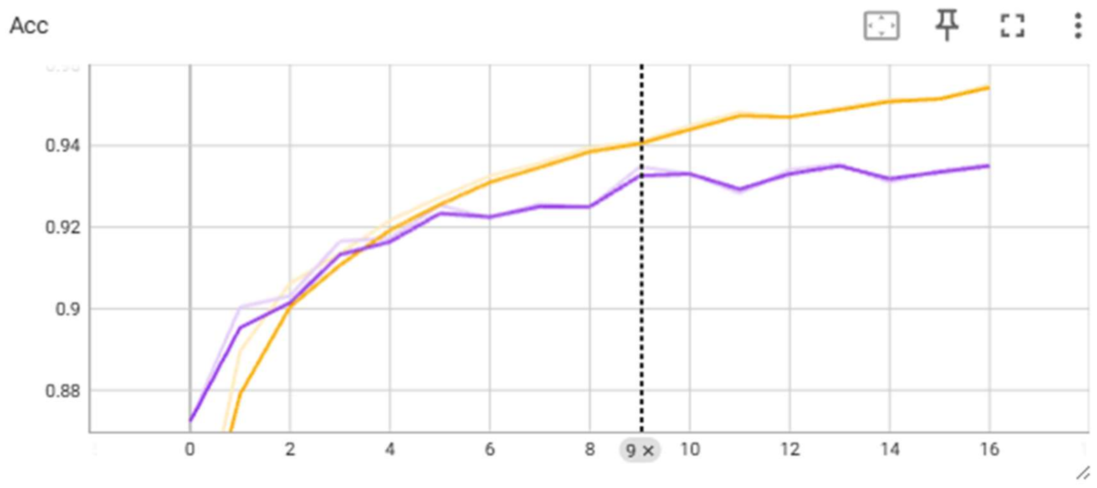
Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.1507	0.1481	12	36.55 min
● Loss_Validation	0.1946	0.1883	12	36.55 min



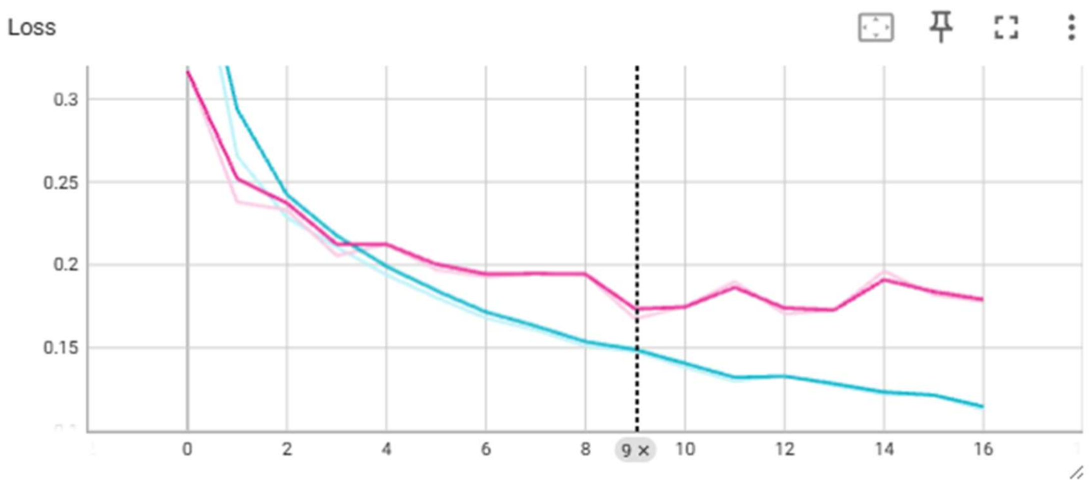
```
for model: d93_unet_weights_60_0.0001_combine_1.pth
Whole, avg_dice_score = 0.9309396291265682
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr =$ 1e-4, $aug\_type = combine$; $n\_aug = 1$) of UNet

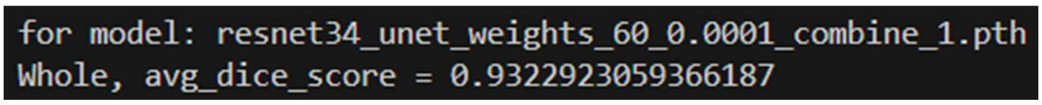
ResNet34_UNet



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.9405	0.9411	9	7.481 min
● Acc_Validation	0.9327	0.9348	9	7.481 min



	Run ↑	Smoothed	Value	Step	Relative
●	Loss_Train	0.1494	0.1481	9	7.481 min
●	Loss_Validation	0.1737	0.1678	9	7.481 min



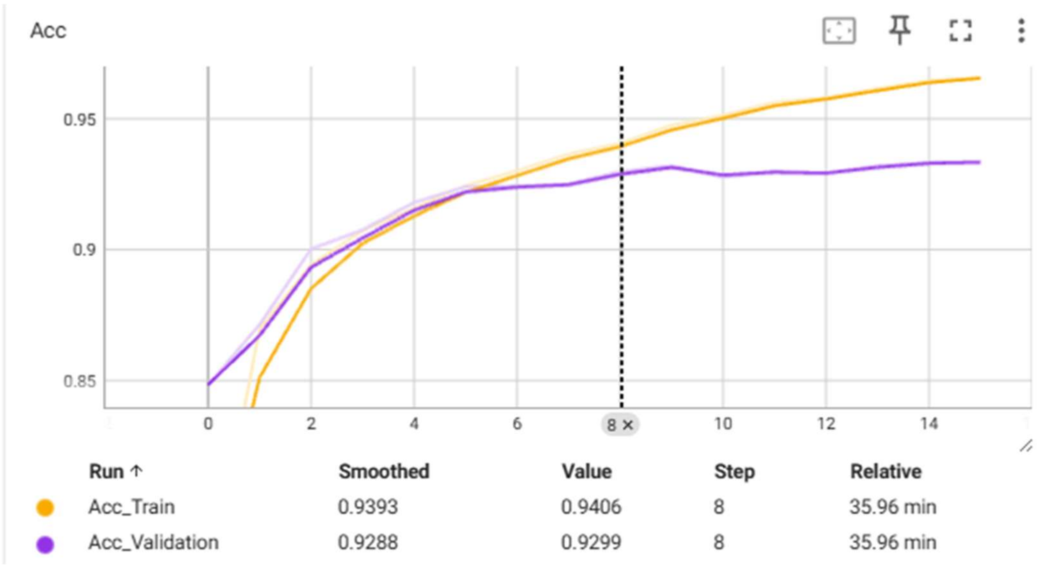
```
for model: resnet34_unet_weights_60_0.0001_combine_1.pth
Whole, avg_dice_score = 0.9322923059366187
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr =$ 1e-4, $aug\_type = combine$; $n\_aug = 1$) of ResNet34_UNet

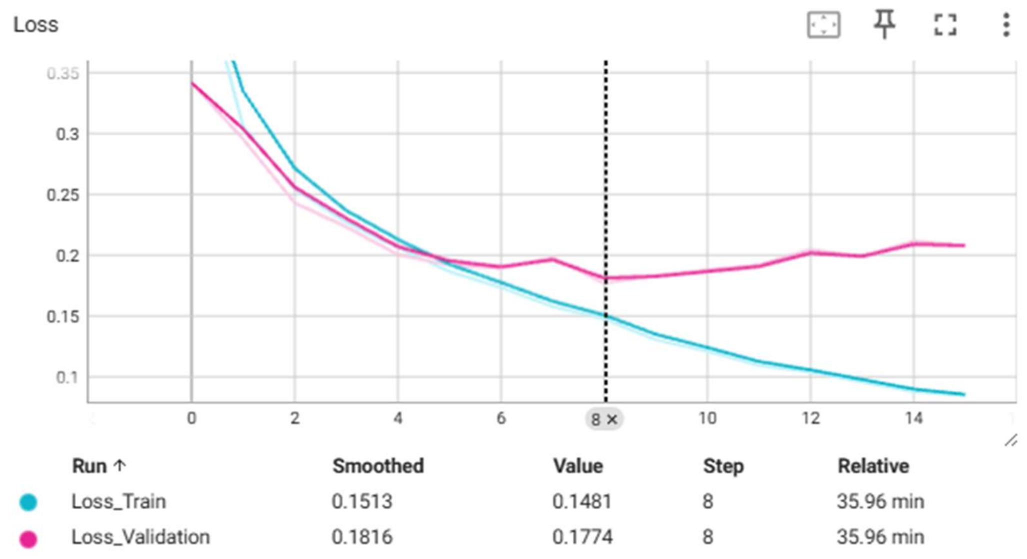
These figures show there are almost **no improvement for ResNet34_UNet** both for loss and dice scores. And **for UNet, the loss (0.18) goes down compared and dice score (0.93) increase compared to UNet with lr=1e-3.** (loss: 0.20 to 0.18. dice: 0.91 to 0.93). I think it is because the original **UNet doesn't have bactch normalize and has a shallow structure, which means it is more likely to oscillate if the learning rate is high**. But for **ResNet34_UNet, it has batch normalize and deep structure, so using smaller learning rate cannot improve its performance.**

For the experiment above, I find there are high chances to improve performance of UNet model by using more data augmentation and lower learning rate. Then, I use ($lr = 1\text{e-}4, aug\_type = combine;\ n\_aug = 2$) to train a UNet model.

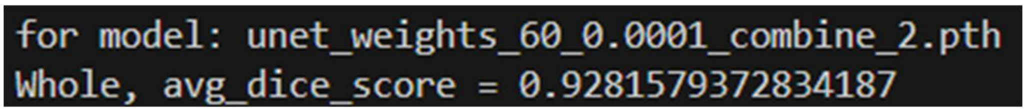
U-Net



Run ↑	Smoothed	Value	Step	Relative
● Acc_Train	0.9393	0.9406	8	35.96 min
● Acc_Validation	0.9288	0.9299	8	35.96 min



Run ↑	Smoothed	Value	Step	Relative
● Loss_Train	0.1513	0.1481	8	35.96 min
● Loss_Validation	0.1816	0.1774	8	35.96 min



```
for model: unet_weights_60_0.0001_combine_2.pth
Whole, avg_dice_score = 0.9281579372834187
```

Acc / Loss on training / validation dataset and Dice score on test dataset with ($lr = 1\text{e-}4, aug\_type = combine;\ n\_aug = 2$) of UNet

These results show that the UNet didn't perform better compare to the situations that use smaller lr or more data augmentation independently. I think that might be because as **using more data augmentation, it is increasing the complexity of data. And using the smaller learning rate at this time might face the problem that sticks in the local minimum. So, the UNet doesn't perform better.**

Overall, we can have the highest dice score by the models with the hyperparameters below:

U-Net: 0.9309

- $epochs = 60$
- $lr =$ 1e-4
- $aug\_type = combine$
- $n\_aug = 1$

ResNet34_UNet: 0.9478

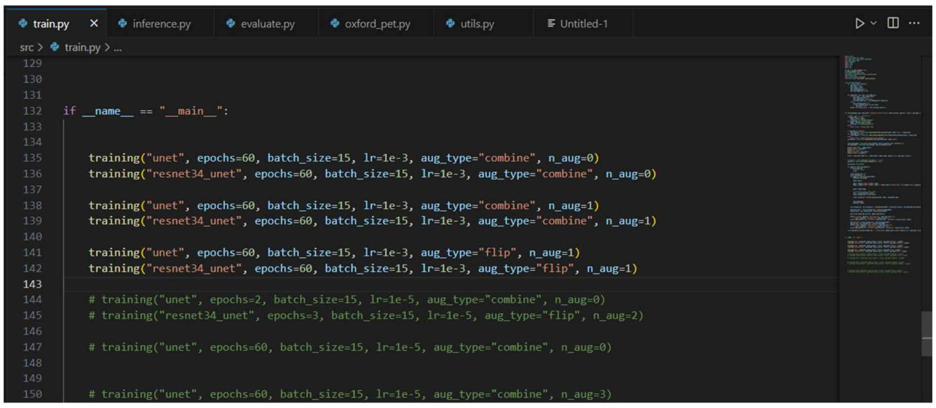
- $epochs = 60$
- $lr = 1e - 3$
- $aug\_type = combine$
- $n\_aug = 1$

Execution steps

Python version: 3.12.6

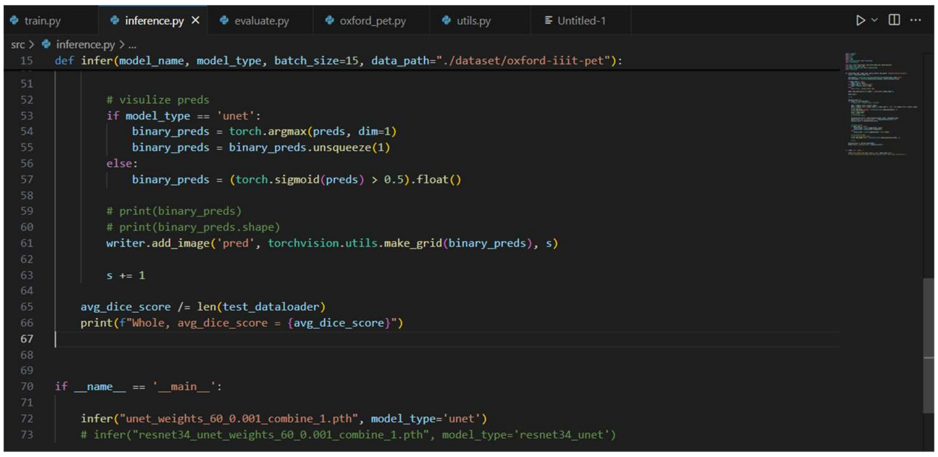
1. Open the **Lab2_Binary_Semantic_Segmentation** folder by VS code
2. Install **requirement.txt** by pip
3. Download data by executing **oxford_pet.py** directly
4. **Training** model by executing **train.py** after entering the model want to train and hyperparameters of the $train(\dots)$ function in $if\ \_name\_\ ==\ \_main\_$: statement.

   *PS: after training, you can use tensorboard to visualize fitting process



```python
if __name__ == "__main__":


    training("unet", epochs=60, batch_size=15, lr=1e-3, aug_type="combine", n_aug=0)
    training("resnet34_unet", epochs=60, batch_size=15, lr=1e-3, aug_type="combine", n_aug=0)

    training("unet", epochs=60, batch_size=15, lr=1e-3, aug_type="combine", n_aug=1)
    training("resnet34_unet", epochs=60, batch_size=15, lr=1e-3, aug_type="combine", n_aug=1)

    training("unet", epochs=60, batch_size=15, lr=1e-3, aug_type="flip", n_aug=1)
    training("resnet34_unet", epochs=60, batch_size=15, lr=1e-3, aug_type="flip", n_aug=1)

    # training("unet", epochs=2, batch_size=15, lr=1e-5, aug_type="combine", n_aug=0)
    # training("resnet34_unet", epochs=3, batch_size=15, lr=1e-5, aug_type="flip", n_aug=2)

    # training("unet", epochs=60, batch_size=15, lr=1e-5, aug_type="combine", n_aug=0)


    # training("unet", epochs=60, batch_size=15, lr=1e-5, aug_type="combine", n_aug=3)
```

5. **Inference** model by executing **inference.py** after entering the name of saved model in $saved\_models$ folder and its type (unet or resnet34_unet) of $infer(\ldots)$ function in $if\ \_name\_ == \_main\_:$ statement.



```python
def infer(model_name, model_type, batch_size=15, data_path="./dataset/oxford-iiit-pet"):

            # visulize preds
            if model_type == 'unet':
                binary_preds = torch.argmax(preds, dim=1)
                binary_preds = binary_preds.unsqueeze(1)
            else:
                binary_preds = (torch.sigmoid(preds) > 0.5).float()

            # print(binary_preds)
            # print(binary_preds.shape)
            writer.add_image('pred', torchvision.utils.make_grid(binary_preds), s)

            s += 1

    avg_dice_score /= len(test_dataloader)
    print(f"Whole, avg_dice_score = {avg_dice_score}")



if __name__ == '__main__':

    infer("unet_weights_60_0.001_combine_1.pth", model_type='unet')
    # infer("resnet34_unet_weights_60_0.001_combine_1.pth", model_type='resnet34_unet')
```

6. To reproduce the result of experiment, please use the trained model in $saved\_models$ folder. The prefix $d(n)$ means the model has n dice score. *
7. Here is the hyperparameter to re-train the models with highest dice scores

U-Net: 0.9309

- $epochs = 60$
- $lr = 1\text{e-}4$
- $aug\_type = combine$
- $n\_aug = 1$

ResNet34_UNet: 0.9478

- $epochs = 60$
- $lr = 1e - 3$
- $aug\_type = combine$

- $n\_aug = 1$

Discussion

For data, I think the label of data might have some error in it, there are some pictures of cats and dogs grappled by humans or blocked by some obstacle. This deformation might let the data set have some noise, I think if took those pictures off, it has chance to improve the performance of model.

For alternative architecture, I think if combined the dice loss in loss function might have chance to improve the accuracy of models. Since the ability of the models are evaluated by dice score, if there are dice loss exist in loss function while training the model, the models will have more specific direction to improve. And for $bridge$ structure of $ResNet34\_UNet$, it reduces the channels of feature maps while encoding step, I think this operation will cause some loss of information in feature maps (same size of feature maps, but with less channels to store information), like blurring the images. So, I think that part can be changed to the operation of original $UNet$.

For potential research directions, I think it is the semantic segmentation on medical images. My major is Biomedical Engineering, and I have many chances to listen to what clinic doctors often complain about. They think labelling lesions are an annoying thing. And think about the aging population and declining birth rate, with the additional problem of a shortage of healthcare workers, Taiwan's medical capacity is expected to face serious shortages. At this point, if models involving semantic segmentation can reduce the workload of doctors or medical technicians, it would be a highly promising direction for development.

Reference

1. Papers on Lab 2 spec.
2. https://www.youtube.com/watch?v=HS3Q_90hnDg
3. https://medium.com/biased-algorithms/batch-normalization-in-cnn-81c0bd832c63