1. Implementation Details (30%)

Training code:

Initialize the GPU device and the datasets, the datasets are setup via function load_dataset() in oxford_pet.py, it returns a SimpleOxfordPetDataset object, then use the function Dataloader() to define how the datasets are read. For loss function, I use BCEWithLogitsLoss which combines sigmoid function and binary cross-entropy loss. For optimizer, I use AdamW which converges faster than normal Adam. In the training loop, batch size is set to 8, epoch is set to 100, learning rate is set to 0.01, and calculating training loss at every iteration for visualization purpose. At the end of each epoch, the model is tested using validation dataset, to calculate the validation loss and check model performance with these unseen data. In addition, if the validation loss is lower than minimum one, store the model weight. Before returning the model, use matplotlib to visualize the training loss and validation loss to check whether overfitting happen or not.

Evaluation code:

In evaluation, first defining the data loader for given dataset and turning the model into evaluation mode. Next, during model prediction, using the dice score to calculate how the predicted mask aligned with true mask. At the end, calculating the total dice score for every sample of the dataset and returning the average dice score.

Inference code:

In inference code, load the trained model first and setup the torch device, then use load_dataset() in oxford_pet.py again to load test dataset which is unseen to the model. At the end, use the defined model, dataset and device to call evaluate() in evaluation code to calculate the average dice score of the test dataset.

UNet model:

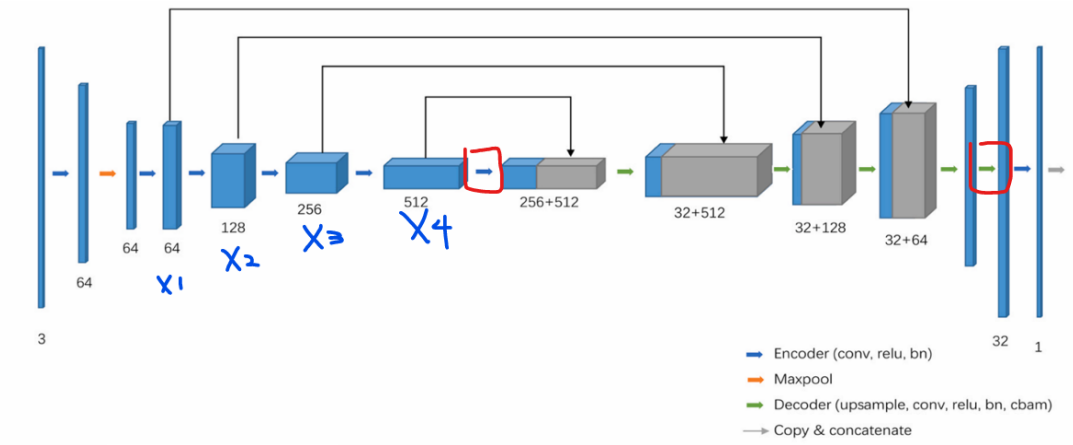
UNet is composed of several components, like double convolution, down sampling, up sampling and output convolution. For the double convolution, it is combined with two sets of convolution layer, batch normalization and ReLU function, kernel sizes are all set to 3, padding sizes set to 1. For down sampling, it is combined with a max pooling and a double convolution. For up sampling, it is combined with a nn.Upsample and a double convolution. For output convolution, it is just one normal convolution layer. To sum up, the whole architecture will be one double convolution first with 64 output channels, then 4 down sampling to encode the image from 64

output channels to 512 output channels, next, 4 up sampling are used to expand the path, besides, taking the skip connection into account and generate 64 output channels. Finally, the output convolution takes 64 input channel to generate 1 output channel, that is, the probability of foreground.

ResUNet model:
The difference is the encoder part to the UNet model, the encoder part of ResUNet is composed of a convolution layer, batch normalization, ReLU, max pooling and 4 residual layers which have different numbers of double convolution layers respectively and down sampling if necessary, that is the encoder part. The decoder part keeps the same as UNet, that is, 4 up sampling. However, there is a little different between the ResUNet architecture of the paper and common ResUNet architecture referred from the Internet[1], first of all, it seems that there is an additional layer in encoder part compared to the common one[1], next, there is a typo in the skip connection of the second decoder layer, the channel of the skip connection should be 256, finally, the all output channels of the decoder are set to 32 which is another difference compared to the common one. To figure out what performance will be between these two architectures, I implement both of them to find out the impact to model performance.

Relevant details:
For the ResUNet model on the paper:

ResNet34 (Encoder) + UNet (Decoder)



It seems that there is another output after x4, but there is only 4 encoders in the architectural reference, as below.

ResNet34

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

In order to make the model architecture more similar to the one on the paper, I add a convolution layer after 4th encoder (red cube on left hand side), which takes 512 input channels and generates 256 output channels. Same as the decoder part, I add an convolution layer after 4th decoder (red cube on right hand side), which takes 32 input channels and generates 32 output channels. The model file name is resnet34_unet_modify.py

During the training, Adam was chosen to be my optimizer initially, but the result showed that the overfitting happened, as a result, weight decay was added to the Adam, however, the result was still not good enough. Therefore, the optimizer was changed to AdamW which has the same idea with Adam but in different implementation. In AdamW, the weight decay is added directly toward the model weight and the level of decay is controlled by the learning rate, that is more intuitive about “decaying” the weight. On the other hand, the weight decay in Adam is added to the gradient, then the weights were be decayed relatively when doing gradient descent. To sum up, AdamW applies weight decay directly and converges faster than Adam.

[1]Common ResUNet implementation reference:
https://blog.csdn.net/mefocus/article/details/129483840

2. Data Preprocessing (25%)

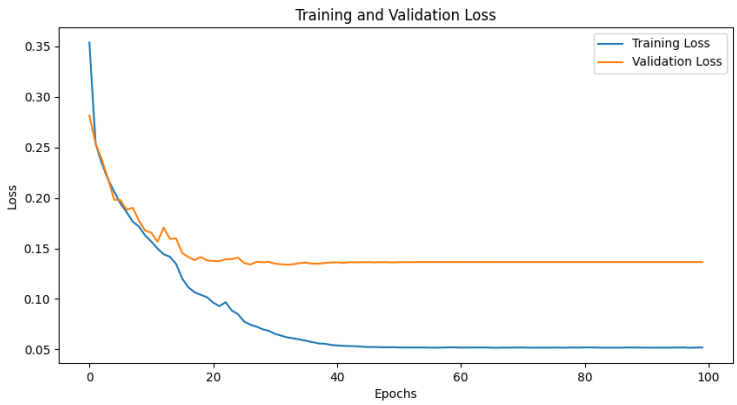
The augmentation is defined in oxford_pet.py, including augment_sample() and adjust_brightness_contrast(). In augment_sample(), after normalization, there is 50% probability to horizontally flip the image, so as to mask to maintain alignment. Besides, the image would be rotated for the angle in a range -15 degree to 15 degree, so as to mask, but the size wasn’t changed and nearest neighbor

interpolation is applied for border handling. In adjust_brightness_contrast(), there are two variables, brightness_factor and contrast_factor, these two variables control the maximum amount of brightness/contrast change. In brightness adjustment, random brightness multiplier ranges from 0.8 to 1.2 will be generated, it multiplies all the pixel values, resulting in those values greater than 1 become brighter, those lower than 1 become darker. In contrast adjustment, first center the values around zero by subtracting 0.5, then applying contrast multiplier to the values, values greater than 1 increase contrast while values lower than 1 decrease contrast, finally add 0.5 to recenter the values. The above approaches create diverse variations of the original training images, which helps the model generalizes well to unseen data. In addition, these preprocessing works well when doing max pooling in model training.

3. Analyze the experiment results (25%)

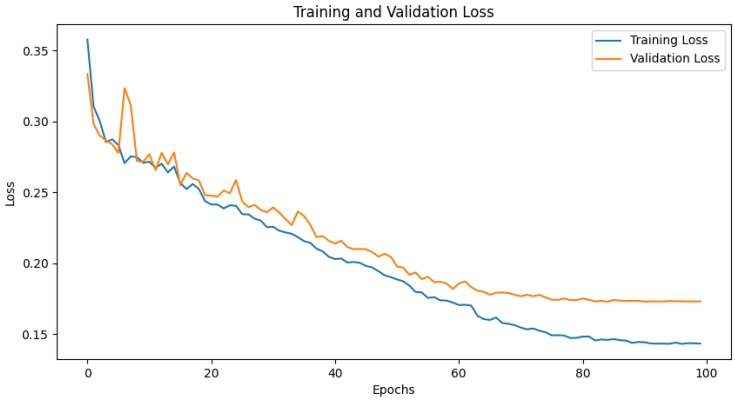
In the model training, the batch size is set to 8, the learning rate is set to 0.01. In the following discussion, the experiment results will cover from the worst one to the best one.

Initially, the training doesn't apply the self-defined preprocessing and the learning rate scheduler is used, in that scheduler, factor is set to 0.5 , which will multiplies the learning rate if adjusting, and the patience is set to 2, which means if the learning rate doesn't reduce for at most 2 epochs, the learning rate will be adjusted. Take ResUNet as an example, here is the training result:

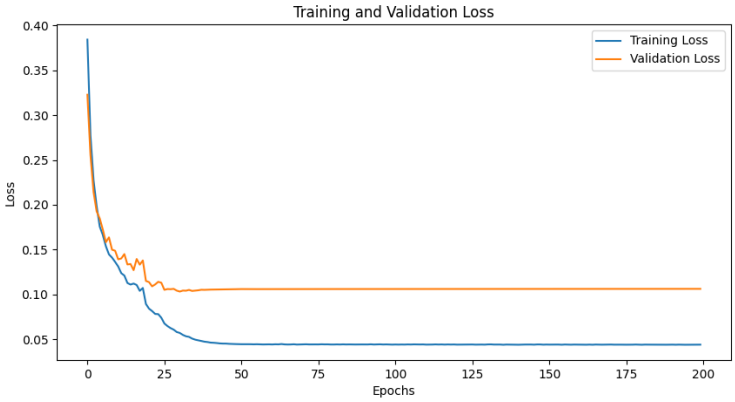


Clearly, this tells that the generalization gap is too large, the overfitting problem may occur. Next, looking for some solutions to deal with this.

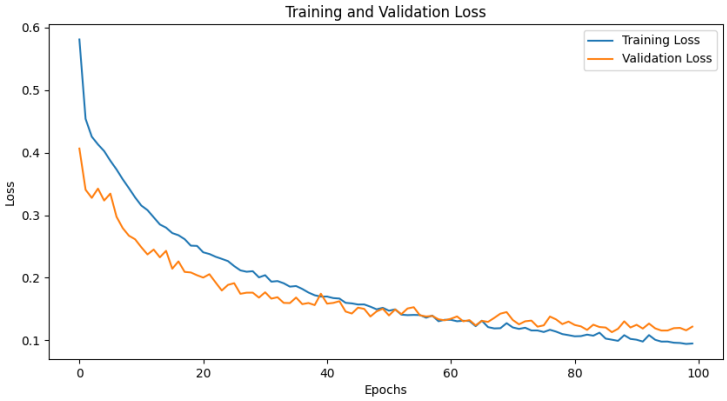
In order to deal with the overfitting problem, I try to use weight decay on Adam optimizer. Here is the result:



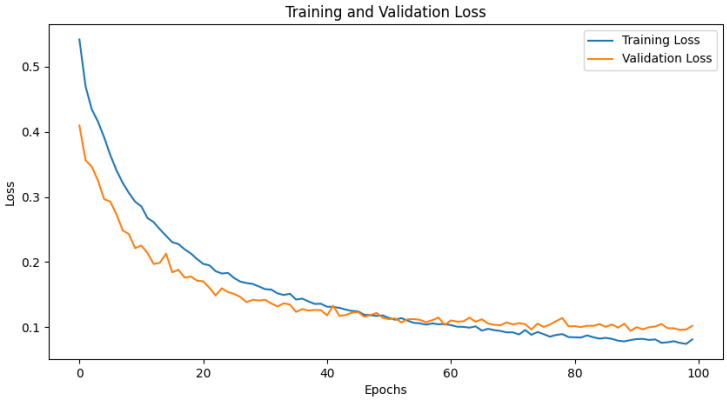
This is much greater than last one, but some how the dice score is still bad.
This time I try to adjust the hyperparameters, epoch set to 200 and learning rate set to 1e-4.
Unfortunately, the result gets worst again, as below:



The overfitting problem shows again.
After some consideration, I want to try another different loss function and there is a question come to my mind. Since the goal is to maximize the dice score, why not use 1-dice score as my loss function to achieve the maximum dice score? To do that, I define a function, dice_loss, which is nearly the same as dice_score function but returns 1-dice score, here is the result of ResUNet:



And the UNet result is also good, as below:



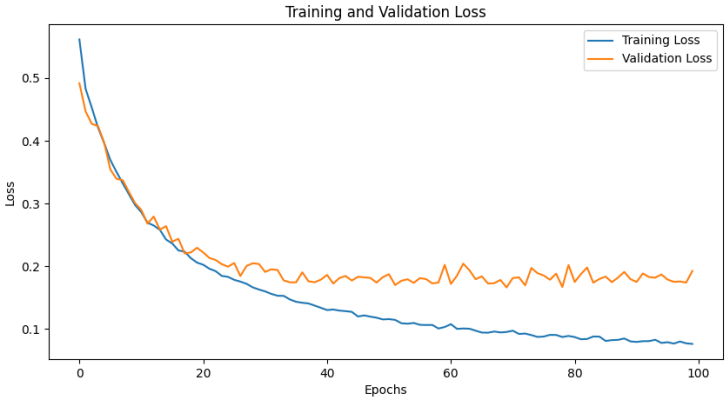
These two results show that the training and validation curve are more flatten than before.
Besides, the dice scores for both models are above 0.9.

But I am not sure that whether it's kind of cheating or not, so I still implement the original loss function, that is, BCEWithLogitsLoss. And this time, the self-defined preprocessing is used. The result is showed below:
This is the ResUNet architecture.

And the dice score for ResUNet is:

```
(DL) johnlol@wnlab-4080:~/lab2$ pyt
Using device: cuda
Dice score: 0.9154646992683411
```

And this is the UNet architecture:

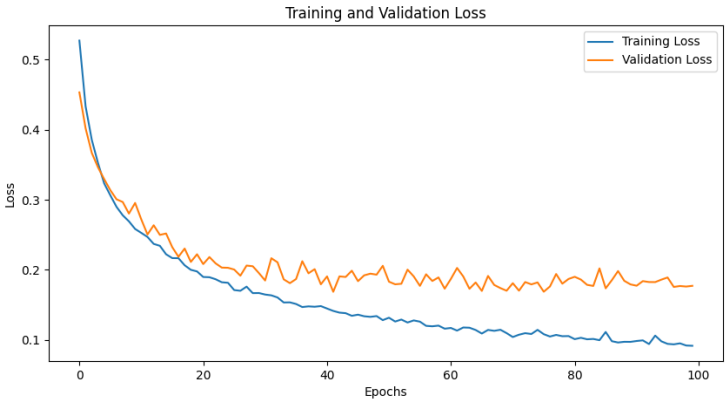


The dice score for UNet is:

```
(DL) johnlol@wnlab-4080:~/lab2$ py
Using device: cuda
Dice score: 0.9265546202659607
```

Both architectures have good dice score and the generalization gap significantly reduces.

The ResUNet discussed above is the common implementation referred to [1], as for the implementation of the ResUNet on the paper, the model performance is showed below, self-defined preprocessing is also used.



The dice score of ResUNet implementation on the paper is :

```
(DL) johnlol@wnlab-4080:~/lab2$ pytho
Using device: cuda
Dice score: 0.9204326868057251
```

From the above results, they can tell that how important the augmentation of data is, not only reduce the generalization gap but also increase the dice score to above 0.9. Making image datasets transpose, flip, brighten and more contrast really have significant impact to model training.

4. Execution steps (0%)

First, create a python virtual environment.
'python3 -m venv DL'
Then activate the virtual environment.
'source ./DL/bin/activate'

In training, the command is 'python3 ./src/train.py --data_path ./dataset/oxford-iiit-pet/ --epochs 100 -b 8 -lr 0.01'.
Besides, in the training code, please ensure the specific model object is correctly invoked, as attached below.

```
41      # Initialize model
42      # model for unet
43      model = unet.UNet(n_channels=3, n_classes=1, bilinear=True)
44      # model for common ResUNet
45      # model = resnet34_unet.ResUNet()
46      # model for ResUNet on paper
47      # model = resnet34_unet_modify.ResUNet()
48      model.to(device)
49
```

And the output model weight will be stored at saved_models directory, the file name will be output_{epoch}_{val_loss}.pth, the latest one refers to the best one.

```
126         if val_loss < best_val_loss:
127             best_val_loss = val_loss
128             torch.save(model.state_dict(), f'./saved_models/output_{epoch}_{val_loss}.pth')
129             print(f'Checkpoint saved! Val Loss: {val_loss:.4f}')
130
131     plot(train_losses, val_losses)
```

In inference, the command is 'python3 ./src/inference.py --model ./path_to_the_model_weight –data_path ./dataset/oxford-iiit-pet/ -b 8'.

Like in the training code, please ensure to load the correct model.

```
    # model for UNet
    model = unet.UNet(n_channels=3, n_classes=1)
    # model for common ResUNet
    # model = resnet34_unet.ResUNet()
    # model for ResUNet on paper
    # model = resnet34_unet_modify.ResUNet()
    model.load_state_dict(torch.load(model_path, map_location=device))
```

And the result will show the dice score of the loaded model.

5. Discussion (20%)

There are several alternative architectures that can potentially yield better result.

1. DeepLabV3+:

DeepLabV3+ incorporates dilated convolutions and an effective encoder-decoder structure. The dilated convolutions increase the receptive field without losing resolution, which is particularly valuable for capturing both fine details and global context in segmentation tasks.

2. Transformer-based Architectures (SETR, SegFormer):

Vision transformers have shown impressive performance in segmentation tasks. Models like SETR (Segmentation Transformer) or SegFormer leverage the self-attention mechanism to capture long-range dependencies in images. This could be particularly helpful for understanding the global structure of pets in the dataset, leading to more coherent segmentations.

Next, there are some potential research directions:

1. Efficient Attention Mechanisms for Segmentation:

Exploring efficient attention mechanisms that can capture global context without the computational overhead of standard transformers.

2. Domain Adaptation for Segmentation Models:

Researching techniques to adapt segmentation models trained on one dataset (like Oxford Pets) to perform well on other datasets with different characteristics (like medical images). This direction addresses the common challenge of domain shift in practical applications.