Deep Learning

Lab2

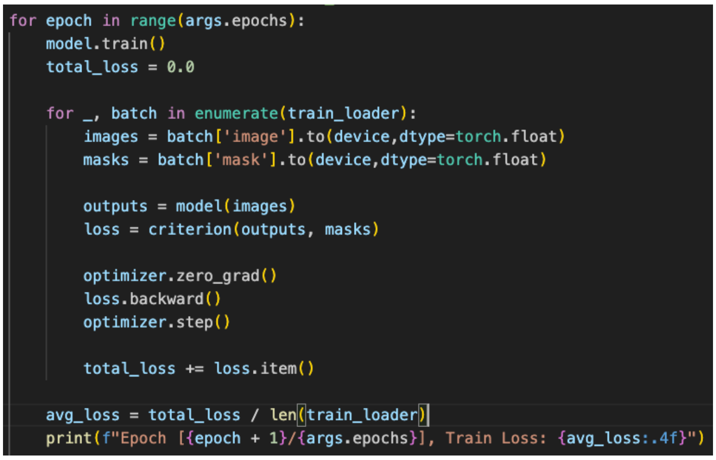
Binary Semantic Segmentation

[110704071] [彭叡楀]

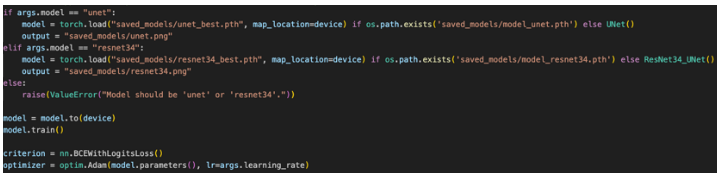
1. Implementation Details:

A. training / validation & inference method:

Training:



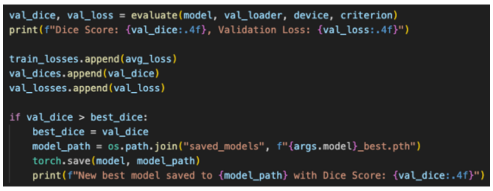
```
for epoch in range(args.epochs):
    model.train()
    total_loss = 0.0

    for _, batch in enumerate(train_loader):
        images = batch['image'].to(device,dtype=torch.float)
        masks = batch['mask'].to(device,dtype=torch.float)

        outputs = model(images)
        loss = criterion(outputs, masks)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch + 1}/{args.epochs}], Train Loss: {avg_loss:.4f}")
```



```
if args.model == "unet":
    model = torch.load("saved_models/unet_best.pth", map_location=device) if os.path.exists('saved_models/model_unet.pth') else UNet()
    output = "saved_models/unet.png"
elif args.model == "resnet34":
    model = torch.load("saved_models/resnet34_best.pth", map_location=device) if os.path.exists('saved_models/model_resnet34.pth') else ResNet34_UNet()
    output = "saved_models/resnet34.png"
else:
    raise(ValueError("Model should be 'unet' or 'resnet34'."))

model = model.to(device)
model.train()

criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
```

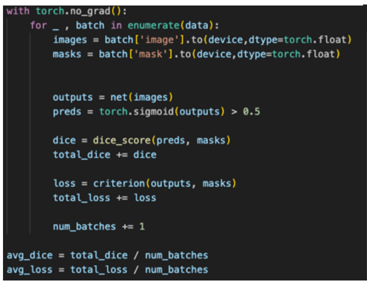
I think this is the main part of the training process, we use the designed model, optimizer, and criterion to train the model based on the images and use the masks to get the loss. In this part we use the back propagation and forward to train the model into a better one.

Validation:

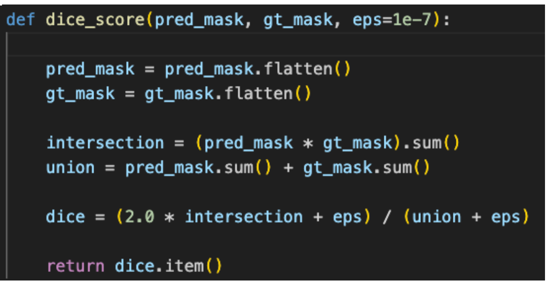


```python
val_dice, val_loss = evaluate(model, val_loader, device, criterion)
print(f"Dice Score: {val_dice:.4f}, Validation Loss: {val_loss:.4f}")

train_losses.append(avg_loss)
val_dices.append(val_dice)
val_losses.append(val_loss)

if val_dice > best_dice:
    best_dice = val_dice
    model_path = os.path.join("saved_models", f"{args.model}_best.pth")
    torch.save(model, model_path)
    print(f"New best model saved to {model_path} with Dice Score: {val_dice:.4f}")
```

We use the validation after the train part of each epoch to calculate it's dice score and accuracy, then save the best model with largest dice score.



```python
with torch.no_grad():
    for _ , batch in enumerate(data):
        images = batch['image'].to(device,dtype=torch.float)
        masks = batch['mask'].to(device,dtype=torch.float)


        outputs = net(images)
        preds = torch.sigmoid(outputs) > 0.5

        dice = dice_score(preds, masks)
        total_dice += dice

        loss = criterion(outputs, masks)
        total_loss += loss

        num_batches += 1

avg_dice = total_dice / num_batches
avg_loss = total_loss / num_batches
```

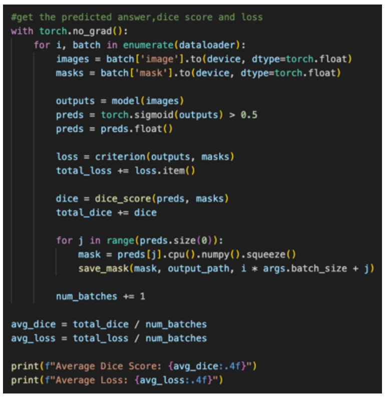


```python
def dice_score(pred_mask, gt_mask, eps=1e-7):

    pred_mask = pred_mask.flatten()
    gt_mask = gt_mask.flatten()

    intersection = (pred_mask * gt_mask).sum()
    union = pred_mask.sum() + gt_mask.sum()

    dice = (2.0 * intersection + eps) / (union + eps)

    return dice.item()
```

In this part we use forward to get the predict mask of the model, then we calculate the dice score and loss with the correct masks, after go through the whole dataset, we return the average dice score and loss. The caculation of the dice score is above and based on the formula.

The dice_score function calculates the Dice Similarity Coefficient, a metric used to evaluate the similarity between two binary masks. First, it flattens both the predicted and ground truth masks into one-dimensional arrays. Then, it computes the intersection by summing the element-wise product of the two masks, while the union is the sum of the elements in both masks. The Dice score is then calculated using the formula: *2×intersection/union* with a small epsilon added to prevent division by zero. Finally, the result is returned as a scalar value.

Inference:

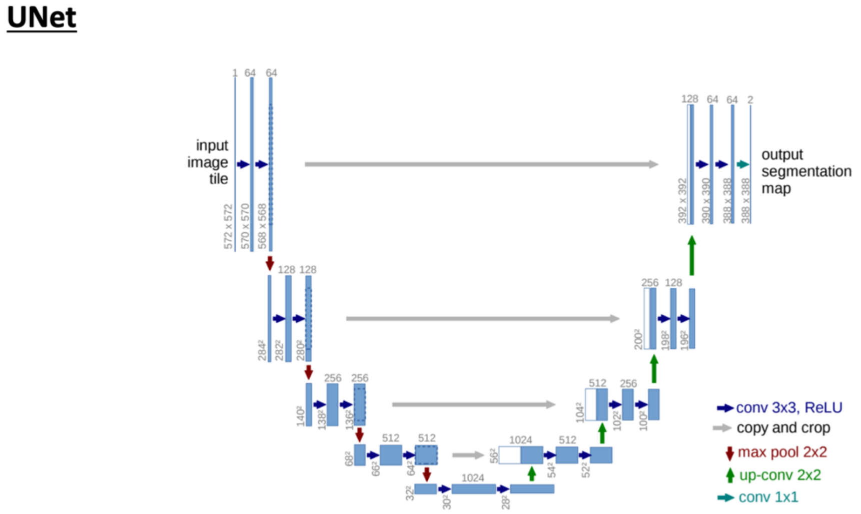


```python
#get the predicted answer,dice score and loss
with torch.no_grad():
    for i, batch in enumerate(dataloader):
        images = batch['image'].to(device, dtype=torch.float)
        masks = batch['mask'].to(device, dtype=torch.float)

        outputs = model(images)
        preds = torch.sigmoid(outputs) > 0.5
        preds = preds.float()

        loss = criterion(outputs, masks)
        total_loss += loss.item()

        dice = dice_score(preds, masks)
        total_dice += dice

        for j in range(preds.size(0)):
            mask = preds[j].cpu().numpy().squeeze()
            save_mask(mask, output_path, i * args.batch_size + j)

        num_batches += 1

avg_dice = total_dice / num_batches
avg_loss = total_loss / num_batches

print(f"Average Dice Score: {avg_dice:.4f}")
print(f"Average Loss: {avg_loss:.4f}")
```

We use the similar way as validation to calculate the predicted masks, dice score and the loss, then saving them to the designed directories.

B. UNet / Resnet34_UNet:

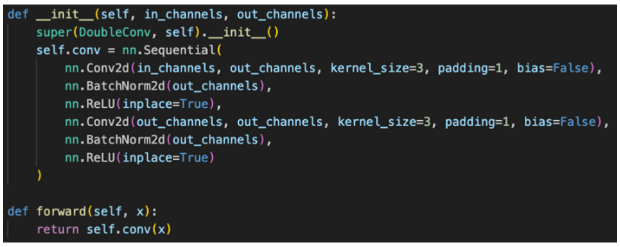
UNet:



Base on the above part, we implement two parts:

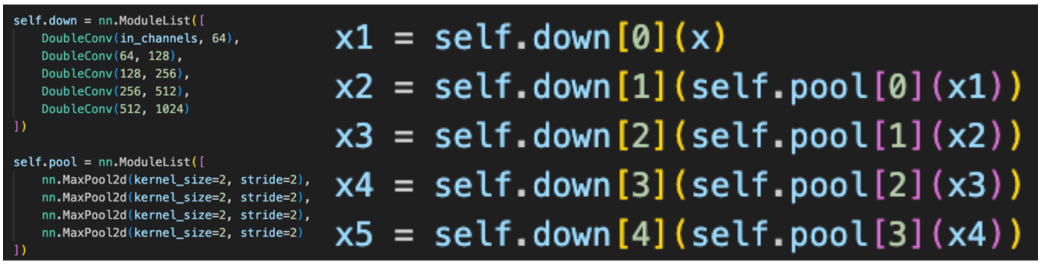
1.Contracting Path (left): The left side of the architecture diagram, as the name suggests, is responsible for down sampling the image and extracting important features into a representation, which is then passed to the later expansive path to decode into a segmentation map.

Down-sampling: Consists of two Convolution operations followed by one Pooling operation, which reduces the size (resolution) of the image while preserving only the features that are important for segmentation. Each down-sampling operation reduces the image size to 1/4 of its original area, while the feature channels are doubled in depth.



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

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

DoubleConv: We use two convolution layers to build it. We also use batch normalization to prevent gradient explosion and vanishing, and then apply ReLU to introduce non-linearity.



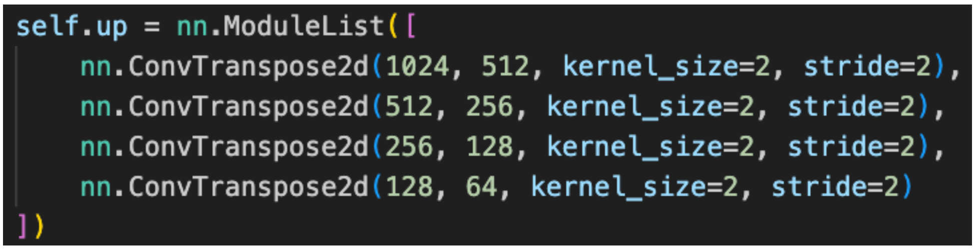
```
self.down = nn.ModuleList([
    DoubleConv(in_channels, 64),
    DoubleConv(64, 128),
    DoubleConv(128, 256),
    DoubleConv(256, 512),
    DoubleConv(512, 1024)
])

self.pool = nn.ModuleList([
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.MaxPool2d(kernel_size=2, stride=2)
])
```

```
x1 = self.down[0](x)
x2 = self.down[1](self.pool[0](x1))
x3 = self.down[2](self.pool[1](x2))
x4 = self.down[3](self.pool[2](x3))
x5 = self.down[4](self.pool[3](x4))
```

So, we make a list of the DoubleCov function and MaxPooling function. Then we put it together to perform the down-sampling.

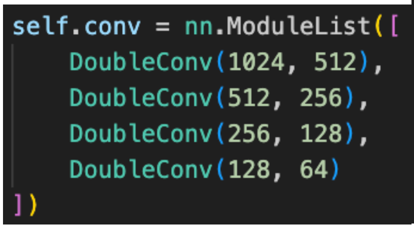
2.Expansive Path(right): This part is like the decoder, which converts the representation extracted by the contracting path back to the original image size.

Up-Sampling: In this part, we use a DoubleCov function and an up-convolution function to perform up-sampling:

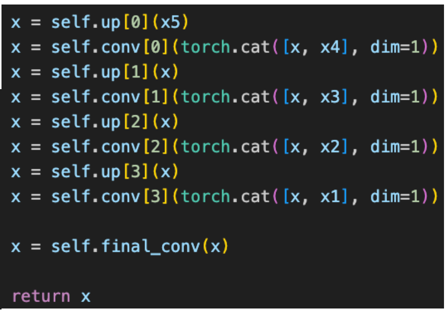


```
self.up = nn.ModuleList([
    nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2),
    nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2),
    nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2),
    nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
])
```

This part I use Transposed Convolution to resize the picture into the original size. It works by expanding the spaces between pixels and adding padding around the original image. It then applies a standard convolution using a learned kernel to fill in the padding and original values with feature-representative values, adjusting the output to the correct size.

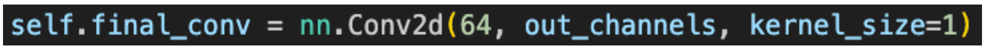


```
self.conv = nn.ModuleList([
    DoubleConv(1024, 512),
    DoubleConv(512, 256),
    DoubleConv(256, 128),
    DoubleConv(128, 64)
])
```

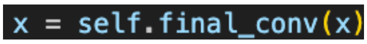


```
x = self.up[0](x5)
x = self.conv[0](torch.cat([x, x4], dim=1))
x = self.up[1](x)
x = self.conv[1](torch.cat([x, x3], dim=1))
x = self.up[2](x)
x = self.conv[2](torch.cat([x, x2], dim=1))
x = self.up[3](x)
x = self.conv[3](torch.cat([x, x1], dim=1))

x = self.final_conv(x)

return x
```

This part I set a list of DoubleConv to handle the photo after upsampling, notify here’s the most important part: concat, we use DoubleConv and torch.cat() to concatenate the features extracted by the Encoder with the unsampled features from the Decoder along the channel dimension, maintaining consistency between the Encoder and Decoder information.



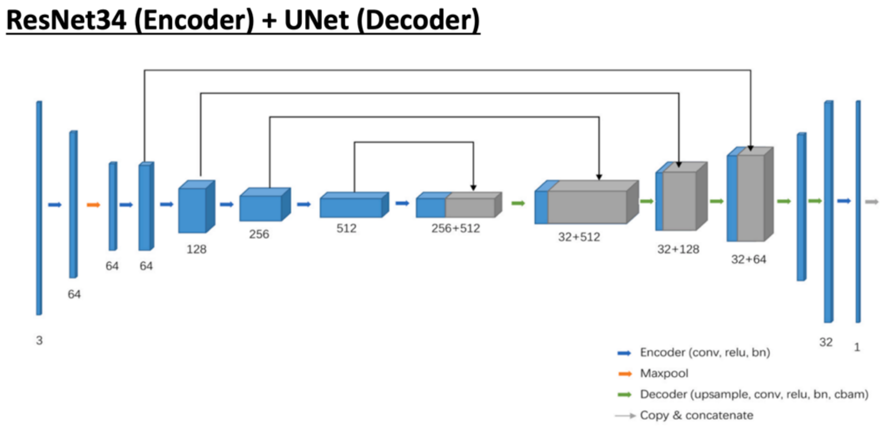
```
self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)
```



```
x = self.final_conv(x)
```

Finally, we use a 1x1 convolution to reduce the final feature map to the desired number of output channels as the prediction.

Resnet34_UNet:



ResNet(Residual Net):

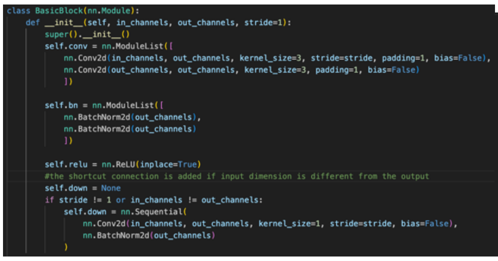
To solve the degradation problem, we use the residual learning technique, which helps in training very deep networks effectively.

Core idea: skip connections/shortcut connections, it allows the input to bypass certain layers and be added directly to the output. This enables the network to learn the residual mapping (the difference between input and output), helping to overcome issues like vanishing gradients and allowing deeper networks to be trained without performance loss.

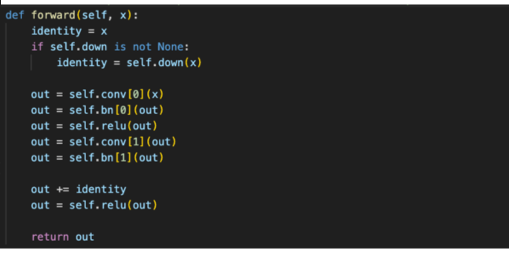
Resnet34-UNet: The CNN use residual mapping. The architecture is similar to the UNet, can get into 2 parts:

1.Encoder (Resnet34): extracting features from the input image and gradually downsampling while retaining important information.

1) Basic block: basic residual block, similar to UNet DoubleConv, however we add the shortcut down, if the input and output dimensions are different, we will add a shortcut connection which directly passes the input data to the output of the convolutional layers, so that the network is able to learn the difference between the input and the output), rather than learning the output itself.



```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv = nn.ModuleList([
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False)
            ])

        self.bn = nn.ModuleList([
            nn.BatchNorm2d(out_channels),
            nn.BatchNorm2d(out_channels)
            ])

        self.relu = nn.ReLU(inplace=True)
        #the shortcut connection is added if input dimension is different from the output
        self.down = None
        if stride != 1 or in_channels != out_channels:
            self.down = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

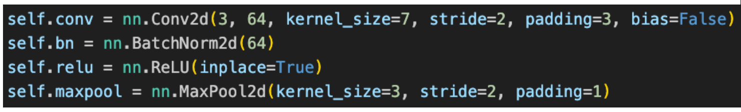


```
def forward(self, x):
    identity = x
    if self.down is not None:
        identity = self.down(x)

    out = self.conv[0](x)
    out = self.bn[0](out)
    out = self.relu(out)
    out = self.conv[1](out)
    out = self.bn[1](out)

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

    return out
```

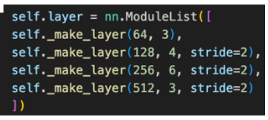
2) Resnet34Encoder

The first is a 7x7 convolutional layer to extract basic features, with Batch Normalization and ReLU.

Then we will build different number layers of the different size. Finally put them together.

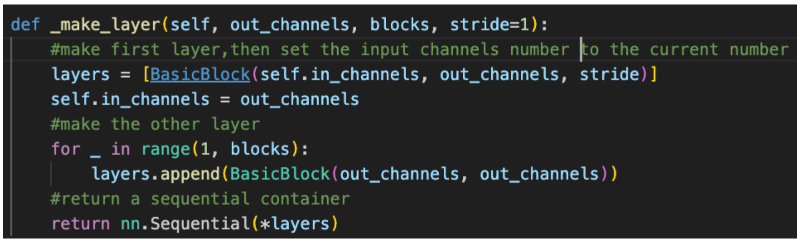


```
self.conv = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
self.bn = nn.BatchNorm2d(64)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```



```
self.layer = nn.ModuleList([
self._make_layer(64, 3),
self._make_layer(128, 4, stride=2),
self._make_layer(256, 6, stride=2),
self._make_layer(512, 3, stride=2)
])
```

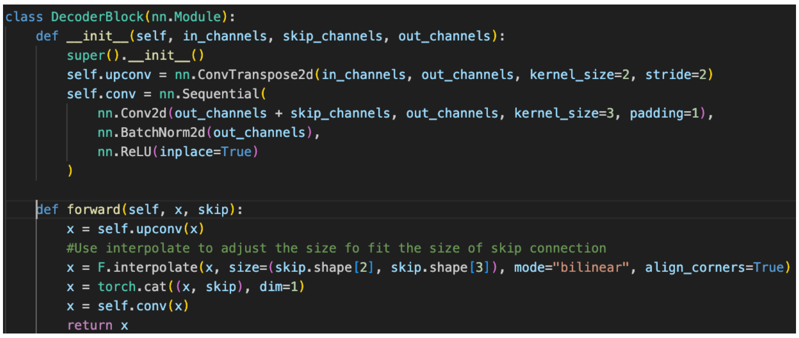
This function creating a series of residual blocks, then return a container that layers are executed sequentially, one after other.



```python
def _make_layer(self, out_channels, blocks, stride=1):
    #make first layer,then set the input channels number to the current number
    layers = [BasicBlock(self.in_channels, out_channels, stride)]
    self.in_channels = out_channels
    #make the other layer
    for _ in range(1, blocks):
        layers.append(BasicBlock(out_channels, out_channels))
    #return a sequential container
    return nn.Sequential(*layers)
```

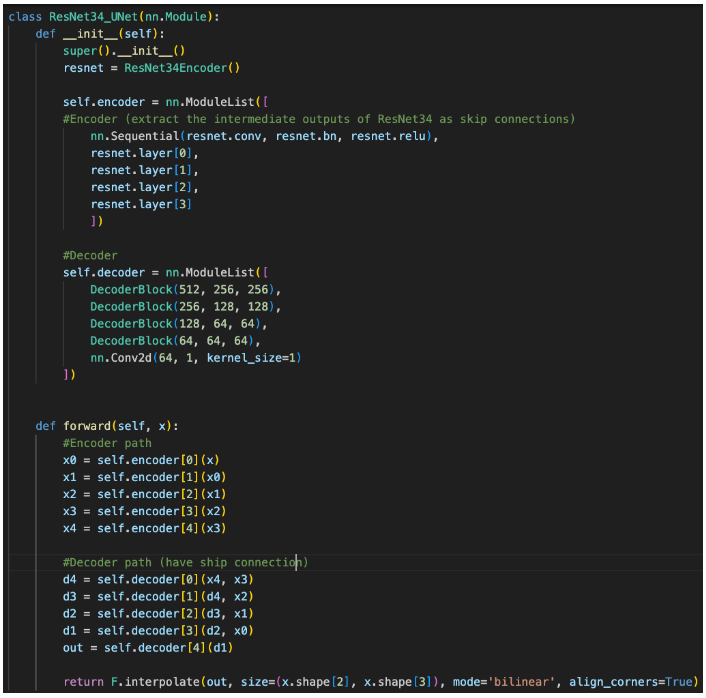
2.Decoder (UNet):

The structure is just like the decoder in UNet, however to fit the size of the skip connection, I use F.interpolate() here to fits the dimension of skip connection and use torch.cat() to merge dimensions, since I need to do the interpolate first. So I put the torch.cat() in the forward function this part.



```python
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, skip_channels, out_channels):
        super().__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.conv = nn.Sequential(
            nn.Conv2d(out_channels + skip_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip):
        x = self.upconv(x)
        #Use interpolate to adjust the size fo fit the size of skip connection
        x = F.interpolate(x, size=(skip.shape[2], skip.shape[3]), mode="bilinear", align_corners=True)
        x = torch.cat((x, skip), dim=1)
        x = self.conv(x)
        return x
```

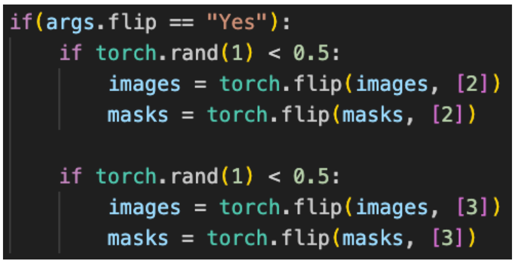
Then put them together, however it seems like the size is smaller than I need, so I use the F.interpolate() again to resize into the normal one.



```python
class ResNet34_UNet(nn.Module):
    def __init__(self):
        super().__init__()
        resnet = ResNet34Encoder()

        self.encoder = nn.ModuleList([
        #Encoder (extract the intermediate outputs of ResNet34 as skip connections)
            nn.Sequential(resnet.conv, resnet.bn, resnet.relu),
            resnet.layer[0],
            resnet.layer[1],
            resnet.layer[2],
            resnet.layer[3]
            ])

        #Decoder
        self.decoder = nn.ModuleList([
            DecoderBlock(512, 256, 256),
            DecoderBlock(256, 128, 128),
            DecoderBlock(128, 64, 64),
            DecoderBlock(64, 64, 64),
            nn.Conv2d(64, 1, kernel_size=1)
        ])


    def forward(self, x):
        #Encoder path
        x0 = self.encoder[0](x)
        x1 = self.encoder[1](x0)
        x2 = self.encoder[2](x1)
        x3 = self.encoder[3](x2)
        x4 = self.encoder[4](x3)

        #Decoder path (have ship connection)
        d4 = self.decoder[0](x4, x3)
        d3 = self.decoder[1](d4, x2)
        d2 = self.decoder[2](d3, x1)
        d1 = self.decoder[3](d2, x0)
        out = self.decoder[4](d1)

        return F.interpolate(out, size=(x.shape[2], x.shape[3]), mode='bilinear', align_corners=True)
```

2. Data Preprocessing:

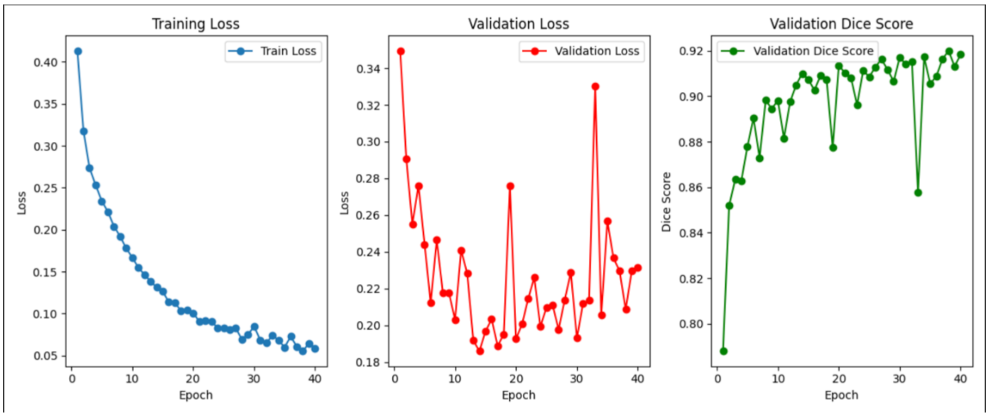
In this part, I try to flip the photo to get more different data to increase the data variability. To make the model more robust and capable of generalizing better to unseen data. This helps the model learn more diverse patterns and improves its performance on different types of input images.



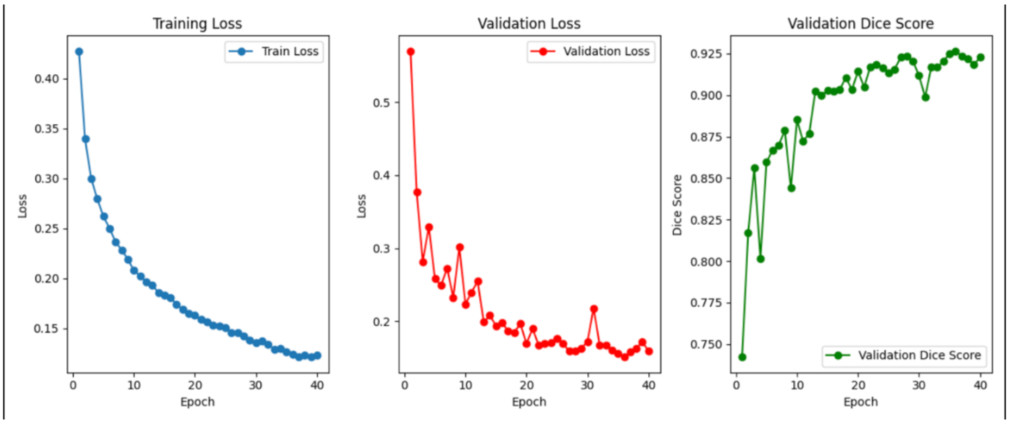
```
if(args.flip == "Yes"):
    if torch.rand(1) < 0.5:
        images = torch.flip(images, [2])
        masks = torch.flip(masks, [2])

    if torch.rand(1) < 0.5:
        images = torch.flip(images, [3])
        masks = torch.flip(masks, [3])
```

In this part, I use the torch.rand(1) to randomly get a number to decide whether I should perform horizontal flip or vertical flip.

Without data augmentation:



With data augmentation:



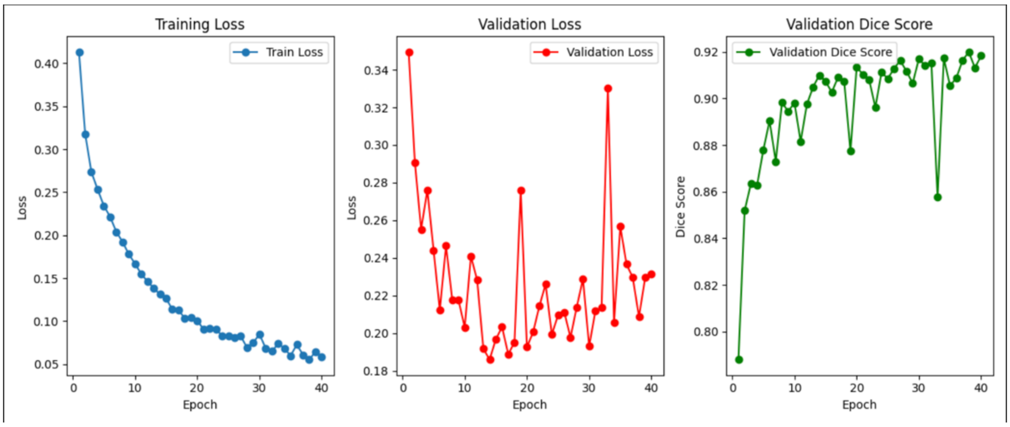
The most unique aspect of my method is that the training loss and dice score trends are similar for both the data with and without data augmentation.

However, in terms of validation loss, we can see that the loss fluctuates and even shows an upward trend in the latter stages. On the other hand, the data with data augmentation demonstrates a more stable decreasing trend. This is because, by adding image flipping and creating different combinations of the dataset in each epoch, the model can learn a broader range of features, which helps reduce the risk of overfitting during training. As a result, the model performs more stably when encountering new, unseen data and avoids getting stuck in local minima or unstable learning phases during training.

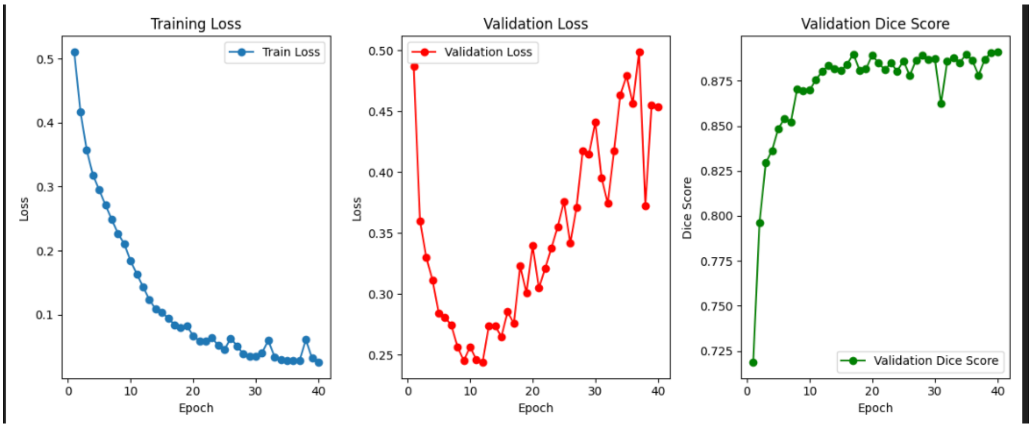
3. Analyze the experiment results:

A. The learning rate effect: In this part, I try to find what if I use a higher learning rate would affect the training process:

lr = 0.0001



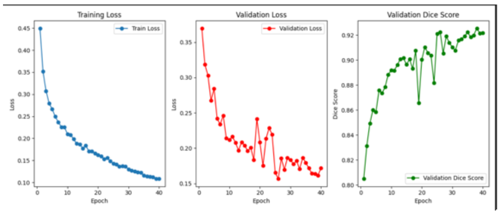
lr = 0.015:

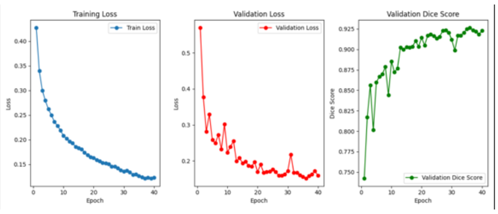


It seems that the validation loss fluctuates more dramatically after increasing the learning rate because a high learning rate can make the model unstable during parameter updates. When the learning rate is set too high, the step size in the gradient descent process becomes too large, which can cause the model to oscillate around the minimum value without converging smoothly. This happens because with a large learning rate, the model jumps around the minimum during each parameter update, preventing stable convergence.

B. the relation of dice score and validation loss overfitting:

UNet: Resnet34_UNet





Here, I noticed that overfitting can sometimes occur. When the training loss decreases to a very low value, the validation loss still fluctuates, and excessive fluctuations may lead to corresponding changes in the dice score. Therefore, to achieve a higher dice score, one can focus on making the validation loss more stable first. This also means that overfitting should be properly handled, such as through data augmentation I use now, and early stopping are really good ideas.

4. Execution steps:

Install: pip install -r requirements.txt

Train(unet):

python src/train.py --model unet --data_path dataset/oxford-iiit-pet --epochs 40 --batch_size 20 --learning-rate 0.0001

Train(resnet34_unet):

python src/train.py --model resnet34 --data_path dataset/oxford-iiit-pet --epochs 40 --batch_size 5 --learning-rate 0.0001

Inference (unet):

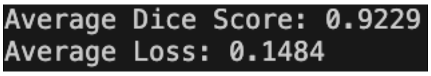
python src/inference.py --model unet --data_path dataset/oxford-iiit-pet --

batch_size 1

Inference(resnet34_unet):

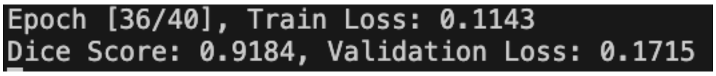
python src/inference.py --model resnet34 --data_path dataset/oxford-iiit-pet --batch_size 1

Inference output:



```
Average Dice Score: 0.9229
Average Loss: 0.1484
```

The train output will output the train loss and validation result of each epoch. After finishing the whole training process, we will get picture of the curve of the train loss/validation loss/dice score of each epoch.



```
Epoch [36/40], Train Loss: 0.1143
Dice Score: 0.9184, Validation Loss: 0.1715
```

If you do not want to use data augmentation, you can add following to your command:

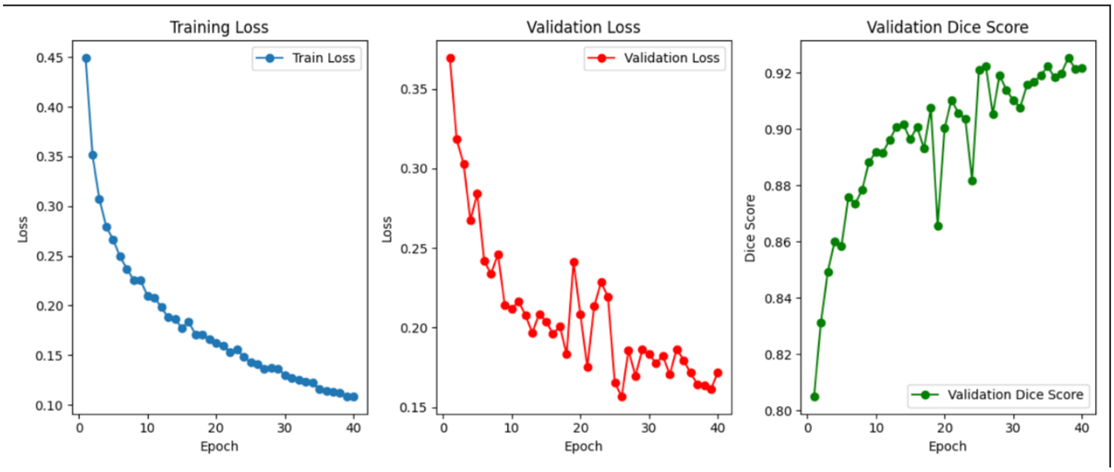
--flip No

5.experiment:

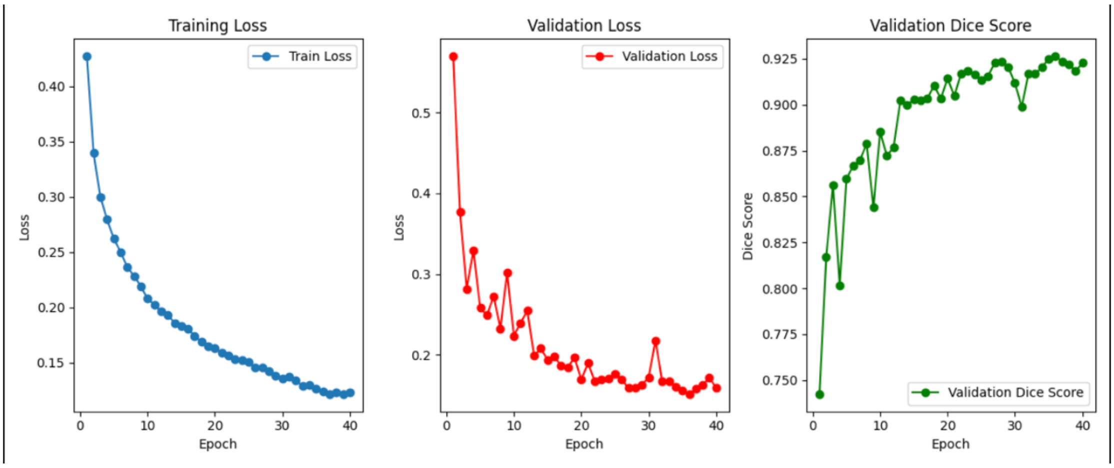
1. What architectures could potentially yield better results? :

We use the data_augmentation and the learning rate = 0.0001,and use Adam Optimizer for 40 epochs:

UNet:



Resnet34_UNet:



We can see that in these two graphs, the training loss curves of both models decrease in a very similar trend. However, ResNet34_UNet experiences less fluctuation in the validation loss and eventually converges to a lower value. On the other hand, although UNet reaches a higher dice score than ResNet34_UNet after just a few epochs, its validation loss exhibits more significant fluctuations, which means it is more prone to overfitting compared to ResNet34_UNet. Additionally, UNet takes more time to complete one epoch during training. Therefore, I believe ResNet34_UNet is the better and more stable model overall.

B. Potential Research Directions:

1.Data Augmentation: