Deep Learning Homework 2 Report

Cheng-Liang Chi

March 23, 2025

Contents

1 Implementation Details

1.1 Model Architecture

All of the following implemention details are implemented using the PyTorch [1] library.

1.1.1 UNet

For the UNet [2] model, I implemented the model in the `UNet` class. The implementation is shown below:

```
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.pool = nn.MaxPool2d(2)
        self.encoder = nn.ModuleList(
            [
                DoubleConv(in_channels, 64),  # 0
                DoubleConv(64, 128),  # 1
                DoubleConv(128, 256),  # 2
                DoubleConv(256, 512),  # 3
                DoubleConv(512, 1024),  # 4
            ]
        )
        self.decoder = nn.ModuleList(
            [
                nn.ConvTranspose2d(1024, 512, 2, stride=2),  # 0
                DoubleConv(1024, 512),  # 1
                nn.ConvTranspose2d(512, 256, 2, stride=2),  # 2
                DoubleConv(512, 256),  # 3
                nn.ConvTranspose2d(256, 128, 2, stride=2),  # 4
                DoubleConv(256, 128),  # 5
                nn.ConvTranspose2d(128, 64, 2, stride=2),  # 6
                DoubleConv(128, 64),  # 7
                nn.Sequential(
                    nn.Conv2d(64, out_channels, 1),
                    nn.Sigmoid(),
                ),  # 8
            ]
        )

    def forward(self, x):
        x1 = self.encoder[0](x)
        x2 = self.encoder[1](self.pool(x1))
        x3 = self.encoder[2](self.pool(x2))
        x4 = self.encoder[3](self.pool(x3))
        x5 = self.encoder[4](self.pool(x4))

        x6 = torch.cat((x4, self.decoder[0](x5)), dim=1)
        x7 = self.decoder[1](x6)
        x8 = torch.cat((x3, self.decoder[2](x7)), dim=1)
```

```
        x9 = self.decoder[3](x8)
        x10 = torch.cat((x2, self.decoder[4](x9)), dim=1)
        x11 = self.decoder[5](x10)
        x12 = torch.cat((x1, self.decoder[6](x11)), dim=1)
        x13 = self.decoder[7](x12)

        ret = self.decoder[8](x13)

        return ret
```

This model consists of an encoder and a decoder. The encoder is a series of convolutional layers with max pooling layers, and the decoder is a series of up-convolutional layers with skip connections from the encoder.

The `DoubleConv` class is a block that consists of two convolutional layers with batch normalization and ReLU activation functions. This component are both used in the encoder and decoder of both the UNet and ResUNet models. The implementation detail will be shown in subsection 1.2.1.

There are some differences between the UNet I implemented and the original UNet [2]:

- The original UNet will receive a 572x572 image and output a 388x388 image. However, I will receive a 256x256 image and output a 256x256 image. This is because the original UNet is designed for EM segmentation challenge at ISBI 2012 [3], which has larger images (512x512) compared to the Oxford Pets dataset we are using in this homework.
- The original UNet didn't use padding in the convolutional layers, which will reduce the size of the image. However, I used padding in the convolutional layers to keep the size of the image the same.
- The original UNet uses "fully convolutional network" [4] by replacing the pooling layers with upsampling layers to increase the resolution of the feature maps. However, I used the up-convolutional layers to upsample the feature maps.
- The original UNet didn't use batch normalization [5], while I used batch normalization in the convolutional layers. This is because in the double convolution block of ResNet [6], batch normalization is used.
- The original UNet used output channels of 2, while I used output channels of 1. The reson will be explained in subsection 1.4.

1.1.2 ResUNet

For the ResUNet [7] model, I implemented the model in the `ResUNet` class. The implementation is shown below:

```
class ResUNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.tail = nn.Sequential(
            nn.Conv2d(in_channels, 64, 7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
```

```
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )  # 0 -> 2

        self.block_nums = [3, 4, 6, 2]
        self.encoder = nn.ModuleList(
            [
                self._make_layer(
                    64,
                    64,
                    self.block_nums[0],
                    stride=1,
                    downsample=False,
                ),  # 2 -> 2
                self._make_layer(64, 128, self.block_nums[1]),  # 2 -> 3
                self._make_layer(128, 256, self.block_nums[2]),  # 3 -> 4
                self._make_layer(256, 512, self.block_nums[3]),  # 4 -> 5
            ]
        )
        self.center = nn.Sequential(
            nn.Conv2d(512, 256, 3, padding="same"),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
        )
        self.decoder = nn.ModuleList(
            [
                nn.Sequential(
                    nn.ConvTranspose2d(256 + 512, 32, kernel_size=2,
                    ↪  stride=2),
                    DoubleConv(32, 32),
                ),
                nn.Sequential(
                    nn.ConvTranspose2d(32 + 256, 32, kernel_size=2,
                    ↪  stride=2),
                    DoubleConv(32, 32),
                ),
                nn.Sequential(
                    nn.ConvTranspose2d(32 + 128, 32, kernel_size=2,
                    ↪  stride=2),
                    DoubleConv(32, 32),
                ),
                nn.Sequential(
                    nn.ConvTranspose2d(32 + 64, 32, kernel_size=2,
                    ↪  stride=2),
                    DoubleConv(32, 32),
                ),
            ],
        )

        self.head = nn.Sequential(
```

```
            nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2),
            nn.Conv2d(32, out_channels, kernel_size=1),
            nn.Sigmoid(),
        )

    def _make_layer(
        self,
        in_channels,
        out_channels,
        num_blocks,
        stride=2,
        downsample=True,
    ):
        layers = [ResNetBlock(in_channels, out_channels, stride,
        ↪  downsample)]
        for _ in range(1, num_blocks):
            layers.append(ResNetBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x1 = self.tail(x)  # 64
        x2 = self.encoder[0](x1)  # 64
        x3 = self.encoder[1](x2)  # 128
        x4 = self.encoder[2](x3)  # 256
        x5 = self.encoder[3](x4)  # 512

        x6 = self.center(x5)  # 256

        x7 = torch.cat((x6, x5), dim=1)  # 256 + 512
        x8 = self.decoder[0](x7)
        x9 = torch.cat((x8, x4), dim=1)  # 32 + 256
        x10 = self.decoder[1](x9)
        x11 = torch.cat((x10, x3), dim=1)  # 32 + 128
        x12 = self.decoder[2](x11)
        x13 = torch.cat((x12, x2), dim=1)  # 32 + 64
        x14 = self.decoder[3](x13)

        return self.head(x14)
```

This model is a ResNet [6] model with a UNet [2] decoder. The ResNet model consists of a series of residual blocks, and the UNet decoder is a series of up-convolutional layers with skip connections from the ResNet model.

This model make a lot of use of both `DoubleConv` and `ResNetBlock` classes. The implementation detail for the `DoubleConv` class is shown in subsection 1.2.1. And the implementation detail for the `ResNetBlock` class will be shown in subsection 1.2.2.

The `self._make_layer` function is used to create a series of residual blocks, just as torchvision [8] does.

1.2 Model Components

1.2.1 Double Convolution Block

The `DoubleConv` class is a block that consists of two convolutional layers with batch normalization and ReLU activation functions.

The convolutional layer has a kernel size of 3x3, a stride of 1, and a padding of 1, which will keep the size of the image the same. The batch normalization layer is used to normalize the output of the convolutional layer. The ReLU activation function is used to introduce non-linearity to the model. For memory efficiency, I set the `inplace` parameter of the ReLU activation function to `True` since the output of the ReLU activation function is only used once.

As mentioned in subsection 1.1.1, the `DoubleConv` block contains batch normalization layers directly after the convolutional layers and before the activation functions. This is because in the double convolution block of ResNet [6], batch normalization is used.

The implementation is shown below:

```
23 class DoubleConv(nn.Module):
24     def __init__(self, in_channels, out_channels, stride=1):
25         super().__init__()
26         self.double_conv = nn.Sequential(
27             nn.Conv2d(
28                 in_channels,
29                 out_channels,
30                 kernel_size=3,
31                 stride=stride,
32                 padding=1,
33                 bias=False,
34             ),
35             nn.BatchNorm2d(out_channels),
36             # DyT2d(out_channels),
37             nn.ReLU(inplace=True),
38             nn.Conv2d(
39                 out_channels, out_channels, kernel_size=3, padding="same",
                    ↪ bias=False
40             ),
41             nn.BatchNorm2d(out_channels),
42             # DyT2d(out_channels),
43             nn.ReLU(inplace=True),
44         )
45
46     def forward(self, x):
47         return self.double_conv(x)
```

The `DyT2d` class is a class that implements the DyT [9] layer accept a 4D tensor as input and output. This is discussed in subsection 5.2.

1.2.2 Residual Block

The `ResidualBlock` class is a block with the `DoubleConv` block and a skip connection.

The DoubleConv block is used to extract features from the input. The skip connection is used to pass the input to the output of the DoubleConv block. There are two conditions for the skip connection:

- If the input and output of the residual block have the same size, the skip connection will be the identity mapping.
- If the input and output of the residual block have different sizes, the skip connection will be a convolutional layer with a kernel size of 1x1 and a stride of 2. The reason for why the stride is fixed to 2 is that the size of the image will always be reduced by half after the convolutional layer in the ResNet [6] model.

The implementation is shown below:

```
class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1,
    ↪  downsample=False):
        super().__init__()
        self.block = DoubleConv(in_channels, out_channels, stride)
        if downsample:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
                ↪  stride=2),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.shortcut = nn.Identity()

    def forward(self, x):
        return self.block(x) + self.shortcut(x)
```

1.3 Model Usage

For the both UNet and ResUNet models on the Oxford Pets dataset, I will set the input channel to 3 and the output channel to 1. The input channel is 3 because the images in the Oxford Pets dataset are RGB images. The output channel is 1 because the task is binary segmentation, and I choose to use the sigmoid activation function in the output layer. In such way, I will get a probability map for the segmentation mask, and be able to use a threshold to get the binary mask.

1.4 Loss Function

1.4.1 BCE Loss

Since the task is binary segmentation, I will use the binary cross-entropy loss function. The binary cross-entropy loss function is defined as:

$$\text{BCE}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{1}$$

where y is the ground truth, $\hat{y}$ is the prediction, and N is the number of pixels in the image.

The reason for using the binary cross-entropy loss function is that the output of the model is a probability map, and the task is binary segmentation. The binary cross-entropy loss function is suitable for this task.

The implementation is shown below:

```
class BCELoss:
    def __init__(self):
        self.bce = nn.BCELoss()

    def __call__(self, pred, target):
        return self.bce(pred, target.float())
```

1.4.2 Dice Loss

However, the score function that we will use to evaluate the model is the Dice coefficient. The Dice coefficient is defined as:

$$\text{Dice}(y, \hat{y}) = \frac{2 \sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i} \tag{2}$$

where y is the ground truth, $\hat{y}$ is the prediction, and N is the number of pixels in the image.

Thus, I defined the Dice loss function as:

$$\text{DiceLoss}(y, \hat{y}) = 1 - \text{Dice}(y, \hat{y}) \tag{3}$$

In such way, when we consider the Dice loss as the loss function, the model will be trained to maximize the Dice coefficient.

The implementation is shown below:

```
class DiceLoss:
    def __init__(self):
        self.smooth = 1e-6

    def __call__(self, pred, target):
        pred = pred.contiguous().view(-1)
        target = target.contiguous().view(-1)
        intersection = (pred * target).sum()
        return 1 - (2.0 * intersection + self.smooth) / (
            pred.sum() + target.sum() + self.smooth
        )
```

1.4.3 BCEDiceLoss

As a result, I defined the BCEDiceLoss as the sum of the binary cross-entropy loss and the Dice loss:

$$\text{BCEDiceLoss}(y, \hat{y}) = \text{BCE}(y, \hat{y}) + \text{DiceLoss}(y, \hat{y}) \tag{4}$$

Before then, I had considered using weighted sum of the binary cross-entropy loss and the Dice loss and planned to have two more hyperparameters to tune. However, I'm lazy to tune the hyperparameters, so I decided to just simply sum the two losses.

The implementation is shown below:

```
59 class BCEDiceLoss:
60     def __init__(self):
61         self.bce = BCELoss()
62         self.dice = DiceLoss()
63 
64     def __call__(self, pred, target):
65         return self.bce(pred, target) + self.dice(pred, target)
```

1.5 Optimizer

1.5.1 Adam

I used the Adam optimizer [10] for both the UNet and ResUNet models.

The Adam optimizer is an adaptive learning rate optimization algorithm that is designed to combine the advantages of two other optimization algorithms: AdaGrad and RMSProp.

1.6 Training

Both of my models are trained using the same training loop that trained end-to-end. The training loop is implemented in the `train.py` file.

The training loop consists of the following steps:

- Set the model to training mode.
- Iterate over the training data loader.
- Move the input and target to the device.
- Forward pass the input through the model.
- Calculate the loss.
- Zero the gradients of the optimizer.
- Backward pass the loss through the model.
- Update the parameters of the model.

This process is repeated for each epoch.

The other parts that are not highlighted are the steps that will not effect the model's performance, such as logging, validation, or saving the model weights.

The implementation of the training loop is shown below:

```python
def train(epoch, net, data_loader, criterion, optimizer, args):
    train_loss_hist = []
    train_dice_hist = []

    net.train()

    for data in tqdm(
        data_loader,
        desc=f"Epoch {epoch+1}/{args.epochs}",
        dynamic_ncols=True,
        position=1,
        unit="imgs",
        unit_scale=args.batch_size,
        colour="yellow",
    ):
        img = data["image"].to(args.device)
        gt_mask = data["mask"].to(args.device)

        pred = net(img).squeeze(1)
        loss = criterion(pred, gt_mask.float())
        dice = dice_score(pred, gt_mask)

        train_loss_hist.append(loss.item())
        train_dice_hist.append(dice.item())

        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(net.parameters(), 1.0)
        optimizer.step()

    train_loss = float(np.mean(train_loss_hist))
    train_dice = float(np.mean(train_dice_hist))

    return train_loss, train_dice
```

This train loop only responsible for an epoch of training. The training loop will be called in the `run` function in the `train.py` file. The `run` function will be called in the `main` function in the `train.py` file.

The responsibility of the `main` function is to parse the arguments, set the random seed, set the device, set the output directory. And the responsibility of the `run` function is to set the model, optimizer, loss function, and data loader. It will not only call the training loop for each epoch, and call the evaluation function with the validation dataset loader after each epoch, and save the model weights if the validation Dice coefficient is improved. Not to mention, it will also log the training loss, validation loss, and validation Dice coefficient. You can see the implementation of the `run` function in the `train.py` file for more details.

1.7 Validation

The validation loop consists of the following steps:

- Set the model to evaluation mode.
- Iterate over the validation data loader.
- Move the input and target to the device.
- Forward pass the input through the model.
- Calculate the loss.
- Calculate the Dice coefficient.

In this part, turning off the gradient calculation is important because we don't want to update the model parameters during validation. This is because the model is already trained, and we only want to evaluate the model's performance on the validation dataset. This will save memory and computation time.

The implementation of the validation loop is shown below, it is similar to the training loop except for the gradient calculation and parameter update:

```
def evaluate(net, dataloader, criterion, args, position=1,
↪  save_results=False):
    val_loss = []
    val_dice = []

    net.eval()

    with torch.no_grad():
        for data in tqdm(
            dataloader,
            desc="Evaluate",
            dynamic_ncols=True,
            position=position,
            unit="imgs",
            unit_scale=args.batch_size,
            colour="yellow",
        ):
            img = data["image"].to(args.device)
            gt_mask = data["mask"].to(args.device)

            pred = net(img).squeeze(1)
            loss = criterion(pred, gt_mask.float())
            dice = dice_score(pred, gt_mask)

            val_loss.append(loss.item())
            val_dice.append(dice.item())

            if save_results:
                for i in range(img.size(0)):
                    save_figure(
                        gt_mask[i],
                        pred[i],
                        args.out_dir,
```

```
                        f"{data['image_name'][i]}.png",
                    )

    avg_loss = sum(val_loss) / len(val_loss)
    avg_dice = sum(val_dice) / len(val_dice)
    return avg_loss, avg_dice
```

1.8 Testing

And the testing loop is a way more similar to the validation loop. In fact, they called the same function `evaluate` in the `evaluate.py` file. The only difference is that the testing loop will call the `evaluate` function with the option `save_results=True` to save the results to the output directory. The results will be saved to another directory called `results` under the output directory.

The implementation of the testing loop is shown below:

```
def test(args):
    test_data = load_dataset(args.data_path, "test")
    test_loader = DataLoader(test_data, batch_size=args.batch_size,
    ↪  shuffle=False)

    model = None
    match args.model:
        case "unet":
            model = UNet(3, 1).to(args.device)
        case "resunet":
            model = ResUNet(3, 1).to(args.device)
        case _:
            raise ValueError("Invalid model name")

    model.load_state_dict(torch.load(args.weight))

    criterion = BCEDiceLoss()

    print("Evaluating the model on the test set")
    avg_loss, avg_dice = evaluate(
        model,
        test_loader,
        criterion,
        args,
        position=0,
        save_results=True,
    )
    ic(avg_loss, avg_dice)
```

And the implemented of `save_figure` function is shown below:

```python
def save_figure(
    mask: torch.Tensor,
    pred: torch.Tensor,
    save_dir: os.PathLike,
    img_name: str,
):
    dir_path = osp.join(save_dir, "results")
    os.makedirs(dir_path, exist_ok=True)
    np_mask = mask.cpu().numpy()
    np_pred = pred.cpu().numpy()

    result = np.hstack([np_mask, np_pred])
    Image.fromarray((result *
    ↪  255).astype(np.uint8)).save(osp.join(dir_path, img_name))
```

1.9 Useless Features

There are some useless features I have done for this homework, and I will list them below:

1.9.1 Progress Bar

I tuned the arguments of `tqdm` to make the progress bar look better. It has the following features:

- There are two progress bars that show the progress of the epoch and the progress of the batch. It will be shown in the last two lines in the terminal and will not conflict each other even with the training log. That is to say, the training log will be output in the middle of the terminal (the third line counting from the buttom).
- The number of iterations of the progress bar of the current epoch will be shown as the number of images trained in the training dataset. This is because the number of iterations of the progress bar of the current epoch is the number of batches in the training dataset. And the number of images in the training dataset is the number of batches in the training dataset multiplied by the batch size. This is more intuitive for me to know how many images have been trained.
- The progress bar will be reactive to the terminal size. With the default setting of `tqdm`, the progress bar will be fixed to the right side of the terminal. And if I accidentally resize the terminal, the progress bar will be broken and look ugly. However, with the tuned setting, the progress bar will be fixed to the right side of the terminal, but will be reactive to the terminal size. That is to say, the progress bar will be fixed to the right side of the terminal, but will not be broken if I resize the terminal.
- The progress bar will be colorful. Two of the progress bars will be colored with the different colors to avoid confusion.

The following figure shows the appearance of the progress bar:

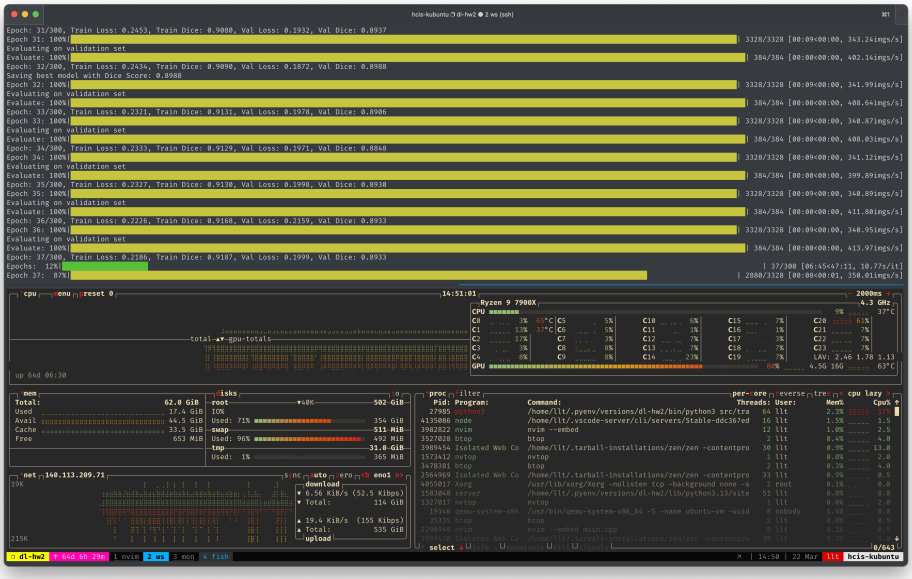

Figure 1: The appearance of the progress bar, the terminal is splited using my configured `tmux` settings. And the bottom plane is a beautiful resouce monitor called `btop++`.

1.9.2 Report Figure Generation

If you had seen the report of my last homework, you may notice that there are up to 10 tables different experiments' comparison. And each of the table has 5 columns and 5 rows, that is 25 cells that need to be filled with the numbers. I'm too lazy to fill the numbers manually, so I wrote some macros in the LaTeX file to read the `accuracy.txt` file in the corresponding directory (according to the hyperparameters) and fill the numbers automatically.

And I also did somthing similar in this homework. However, instead of inventing the wheels again, I used the `pgfplotstable` package to read the `epochs.csv` file and generate the figures automatically. The `epochs.csv` file is generated by the `train.py` file, and it contains the training loss, validation loss, and validation Dice coefficient of each epoch. The `pgfplotstable` package will read the `epochs.csv` file and generate the figures in the LaTeX file. All figures except Figure 1 in this report are generated by the `pgfplotstable` package. You can see the LaTeX source code for more details in both homeworks if you are interested.

Tl;dr, I used the `pgfplotstable` package to generate the figures in the report automatically instead of using `matplotlib` to generate the figures by another python script.

2 Data Preprocessing

For data preprocessing, I first resized the images to 576×576 pixels as Olaf Ronneberger et al. [2] did. However, the model perform poorly no matter how I tuned the hyperparameters. I then resized the images to 256×256 pixels and the model perform much better. The reason might be that our dataset need to upsample the images to 576×576 pixels, which might cause the model to learn on degraded images. And I haven't tried any data augmentation at that time, so the model might not learn invariant features and not generalize well.

However, after I resized the images to 256×256 pixels the model can perform very good while training on the training set but perform poorly on the validation set (cannot exceed 0.9

dice score on the validation set), which indicates that the model is overfitting and the model is not generalizing well.

As Olaf Ronneberger et al. [2] mentioned in their paper, they relies on the strong use of data augmentation to use the available annotated samples more effectively.

Since our task has little training data available (around 3,300 images), I use excessive data augmentation same as the authors of the UNet paper did. This allows the model to learn invariant features and improve the generalization of the model.

2.1 Data Augmentation

After surveying many libraries, I found that the Albumentations library [11] is a very powerful library for image augmentation. It is also suprised that the default code TA provided in `oxford_pet.py` is very similar to the code in the official document of Albumentations. Therefore, I decided to use Albumentations to augment the data.

The augmentation methods I used are as follows:

- Resize: Resize the image to 256×256 pixels.
- HorizontalFlip: Flip the image horizontally.
- VerticalFlip: Flip the image vertically.
- Rotate: Rotate the image by an angle.
- RandomBrightnessContrast: Randomly change brightness and contrast of the image.
- HueSaturationValue: Randomly change hue, saturation and value of the image.
- RGBShift: Randomly shift values for each channel of the image.
- RandomGamma: Randomly change gamma of the image.

The augmentation methods are randomly applied to the images during training. For testing and validation, only the resize operation is applied to the images. This can make validation and testing results more reliable.

The results shows that the model can perform very well on the validation set, which has dice score higher than 0.9 on the validation set. The implementation of the transform using Albumentations is shown below:

```
121
122 def load_dataset(data_path, mode):
123     train_transform = A.Compose(
124         [
125             A.Resize(height=256, width=256),
126             A.HorizontalFlip(p=0.5),
127             A.VerticalFlip(p=0.5),
128             A.Rotate(limit=30, p=0.5),
129             A.RandomBrightnessContrast(p=0.2),
130             A.HueSaturationValue(
131                 hue_shift_limit=20,
132                 sat_shift_limit=30,
```

```
                val_shift_limit=20,
                p=0.2,
            ),
            A.RGBShift(
                r_shift_limit=15,
                g_shift_limit=15,
                b_shift_limit=15,
                p=0.2,
            ),
            A.RandomGamma(gamma_limit=(80, 120), p=0.2),
        ],
        seed=137,
        strict=True,
    )
    transform = A.Compose(
        [
            A.Resize(height=256, width=256),
        ]
    )
    dataset = OxfordPetDataset(
        root=data_path,
        mode=mode,
        transform=transform if mode != "train" else train_transform,
    )
    return dataset
```

3 Analyze and Experiment Results

3.1 Basic Experiment

In the basic experiment, I used experimented two models on the Oxford-IIIT Pet dataset.

For both models, I used the following hyperparameters:

- Batch size: 8
- Epochs: 100
- Optimizer: Adam
- Learning rate: 1×10^{-4}
- Scheduler: None

The training and validation dice score for both models are shown in Figure 2. And the loss curve for both models are shown in Figure 3. We can see that both models have similar performance on the validation set, which has dice score higher than 0.9 on the validation set. The training dice score is higher than the validation dice score, which is expected. The training loss is lower than the validation loss, which is also expected. Note that the loss is described at subsection 1.4. Please refer there for more details.

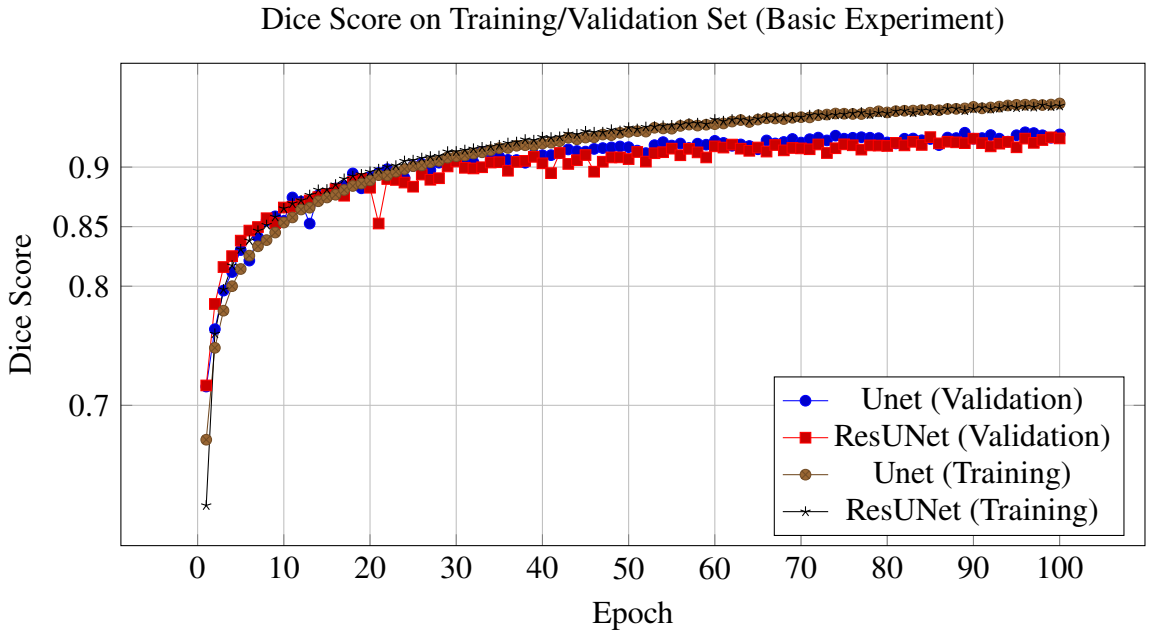

Figure 2: Dice score on training and validation sets for both models (Basic Experiment)

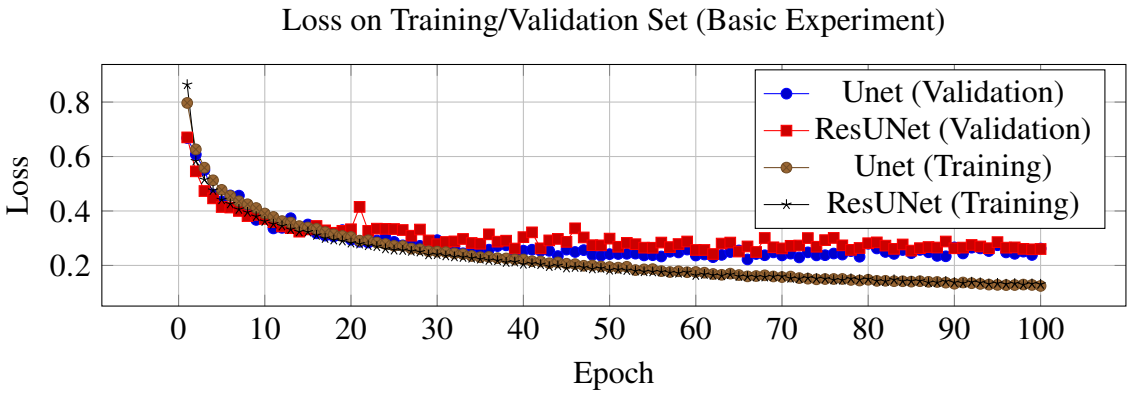

Figure 3: Loss on training and validation sets for both models (Basic Experiment)

4 Execution Steps

4.1 Training

For training the UNet model, you can use the following command:

```
python3 src/train.py -b 8 -e 200 -o u1 -lr 0.0001 -m unet --data_path
↪  dataset/oxford-iiit-pet
```

The command above will train the model with a batch size of 8, 200 epochs, Adam optimizer (by default), and a learning rate of 1×10^{-4}.

For training the ResUNet model, you can use the following command:

```
python3 src/train.py -b 8 -e 200 -o ru1 -lr 0.0001 -m resunet --data_path
↪ dataset/oxford-iiit-pet
```

Note that the `-o` option is used to specify the output directory for saving the model and logs. You can change the output directory to any directory you want. But mentioned that the output directory should will be prefixed with `log/`.

4.2 Inference

For inference, you can use the following command:

```
python3 src/inference.py -m unet -o u1 -w best_model.pth --data_path
↪ dataset/oxford-iiit-pet
python3 src/inference.py -m resunet -o ru1 -w best_model.pth --data_path
↪ dataset/oxford-iiit-pet
```

5 Discussion

5.1 Double Convolution Without Batch Normalization

Since the UNet model I implemented use the double convolution block with batch normalization that Olaf Ronneberger et al. [2] didn't use, I decided to implement the double convolution block without batch normalization to see how it affects the performance of the model.

The result of the model with double convolution block without batch normalization is shown in Figure 4 and Figure 5.

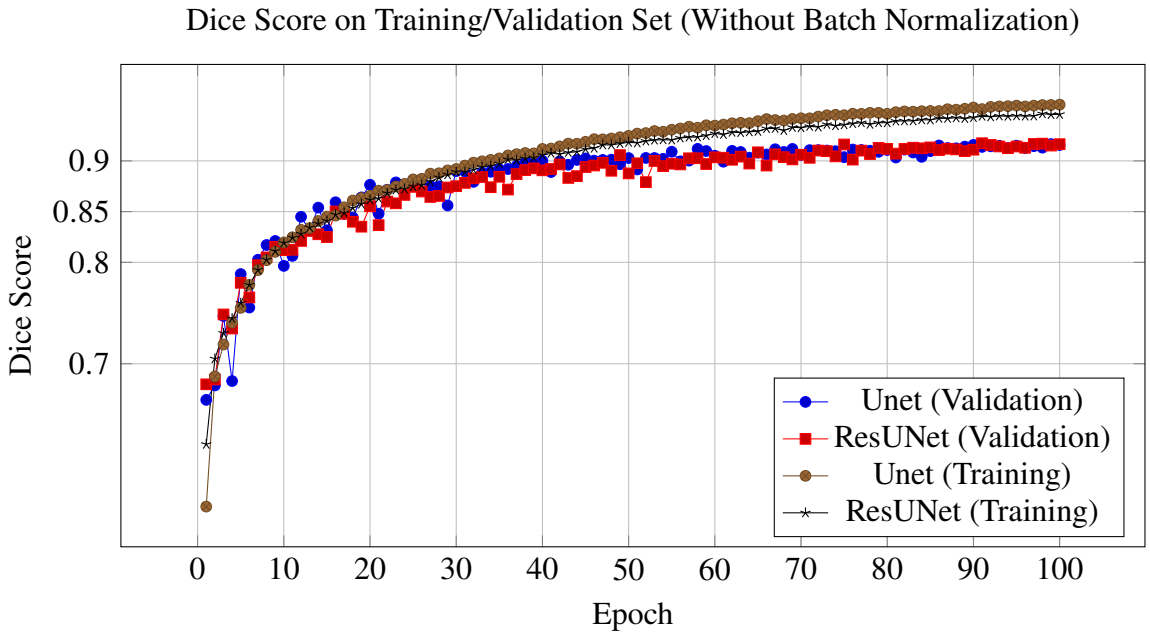

Figure 4: Dice score on training and validation sets for both models (Without Batch Normalization)

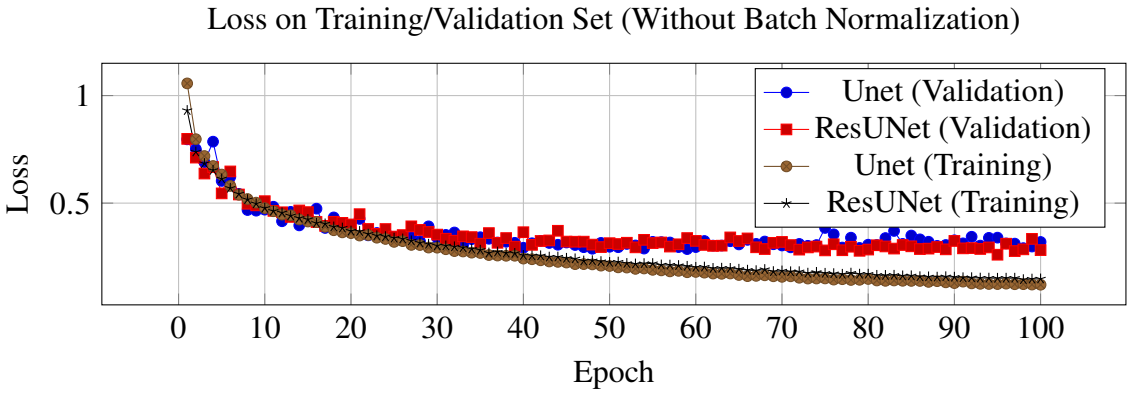

Figure 5: Loss on training and validation sets for both models (Without Batch Normalization)

We can see that the model with double convolution block without batch normalization has not only a slightly lower performance than the model with batch normalization, but it is also not as stable as the model with batch normalization.

5.2 DyT: Replace Batch Normalization with DyT

However, the double convolution block without batch normalization is not the only way to replace batch normalization. Jiachen Zhu et al. [9] proposed that Transformers without normalization can achieve similar or even better performance with a remakably simple technique called DyT (Dynamic Tanh). DyT is a simple technique that replaces the normalization layer with parameterized tanh, which is an element-wise operation that scales the input tensor by a learnable parameter.

The implementation of DyT is quite simple, it is shown below:

```
class DyT2d(nn.Module):
    def __init__(self, num_features):
        super().__init__()
        self.alpha = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))
        self.gamma = nn.Parameter(torch.ones(num_features))

    def forward(self, x):
        B, C, H, W = x.size()
        x = x.view(B, C, -1)
        x = x * self.alpha.view(1, C, 1)
        x = torch.tanh(x)
        x = x * self.gamma.view(1, C, 1)
        x = x + self.beta.view(1, C, 1)
        x = x.view(B, C, H, W)
        return x
```

The result of the model with DyT is shown in Figure 6 and Figure 7. We can see that the model with DyT has a similar performance to the model without batch normalization, which is slightly lower than the model with batch normalization, but still have a dice score higher than 0.9 on the validation set for the last few epochs. And the figure also shows that it is even not as stable as the model without batch normalization.

This might be because the DyT is not suitable for the double convolution block, since it is designed for the Transformer model. It is worth to investigate more on why DyT doesn't work well on the double convolution block and how to replace the batch normalization layer with DyT in the double convolution block.

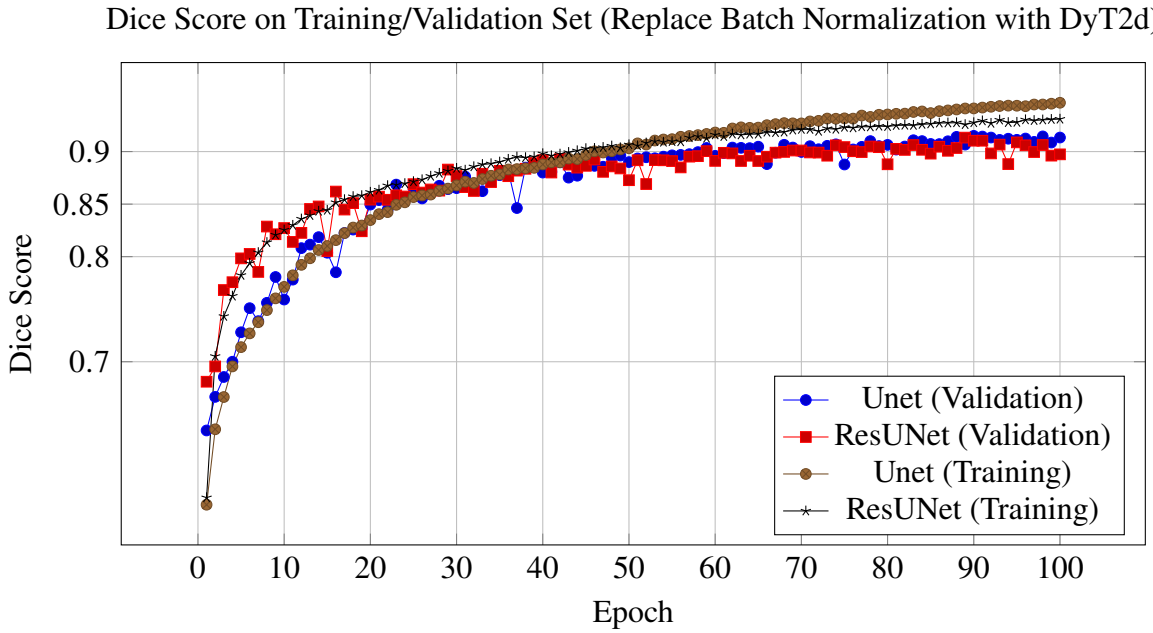

Figure 6: Dice score on training and validation sets for both models (Replace Batch Normalization with DyT2d)

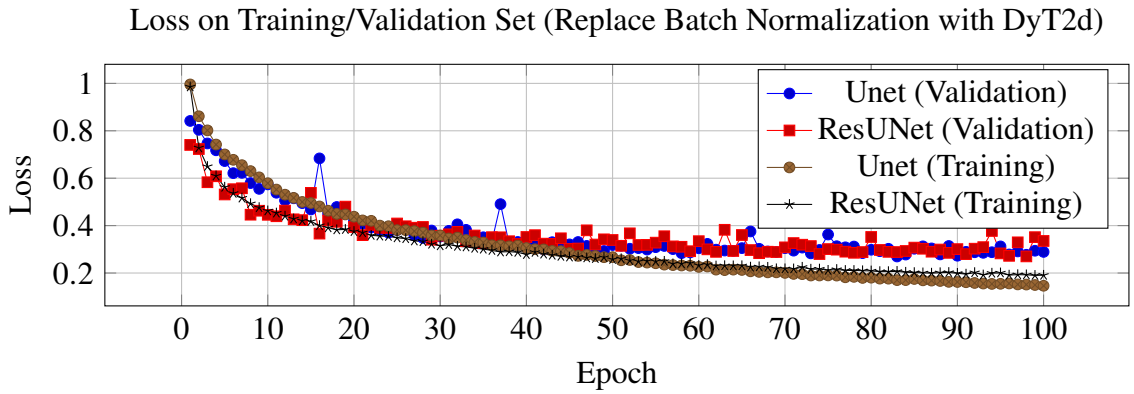

Figure 7: Loss on training and validation sets for both models (Replace Batch Normalization with DyT2d)

References

[1] J. Ansel, E. Yang, H. He, *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf.

[2] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. arXiv: 1505.04597. [Online]. Available: http://arxiv.org/abs/1505.04597.

[3] I. Arganda-Carreras, S. C. Turaga, D. R. Berger, and et al., "Crowdsourcing the creation of image segmentation algorithms for connectomics," *Frontiers in Neuroanatomy*, vol. 9, p. 142, 2015. DOI: 10.3389/fnana.2015.00142.

[4] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014. arXiv: 1411.4038. [Online]. Available: http://arxiv.org/abs/1411.4038.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 448–456.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385.

[7] Z. Huang, E. Qu, Y. Meng, *et al.*, "Deep learning-based pelvic levator hiatus segmentation from ultrasound images," *European Journal of Radiology Open*, vol. 9, p. 100 412, Mar. 2022. DOI: 10.1016/j.ejro.2022.100412.

[8] T. maintainers and contributors, *Torchvision: Pytorch's computer vision library*, https://github.com/pytorch/vision, 2016.

[9] J. Zhu, X. Chen, K. He, Y. LeCun, and Z. Liu, "Transformers without normalization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6628106.

[11] A. Buslaev, A. Parinov, E. Khvedchenya, V. I. Iglovikov, and A. A. Kalinin, "Albumentations: fast and flexible image augmentations," *ArXiv e-prints*, 2018. eprint: 1809.06839.