TAICA Deep Learning Lab 2 Report

01. Introduction

- Name: HanPing Wu
- Student ID: 410410042 (in CCU)

0b. Environment

- CPU: Intel(R) Core(TM) i7-14700
- GPU: NVIDIA GeForce RTX 4090
- OS: Ubuntu 22.04.5 LTS
- Python: 3.11.7
- Setup the environment in the lab server and using ssh to work on the lab.

1. Implementation Details

Unet

Both Unet model and resnet34+unet model need a convolution block. The convolution block is a combination of two 3x3 convolution layers with ReLU activation function. The convolution block is used in the encoder part of the Unet model and the resnet34+unet model.

The first question of the implementation to UNet is the up-covolution. The paper said that upsampling of the feature map followed by a 2x2 convolution (up-convolution) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. But it is not clear how to implement the up-convolution. I found that the `nn.ConvTranspose2d` can be used to implement the up-convolution. The `nn.ConvTranspose2d` can upsample the input tensor and then apply a convolution operation. The `nn.ConvTranspose2d` has a parameter `stride` which can be used to control the upsample rate. The `nn.ConvTranspose2d` also has a parameter `output_padding` which can be used to control the output size. The `nn.ConvTranspose2d` can be used to implement the up-convolution in the Unet model.

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,
stride=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1,
stride=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )
```

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

The original U-Net uses unpadded 3×3 convolutions, causing the output (e.g., 388×388) to be smaller than the input (e.g., 572×572), which mismatches the ground truth size (e.g., 1024×1024) and complicates training. I addressed this by switching to padded convolutions (padding=1), ensuring the output matches the input size (e.g., 1024×1024). This practical modification eliminates preprocessing needs, enhancing U-Net's usability while preserving its core functionality.

```
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels, feature_sizes = [64, 128, 256,
512], dropout=0.1):
        super(UNet, self).__init__()
        self.downs = nn.ModuleList()
        self.ups = nn.ModuleList()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        for feature_size in feature_sizes:
            self.downs.append(ConvBlock(in_channels, feature_size))
            in_channels = feature_size
        self.bottleneck = nn.Sequential(
            ConvBlock(feature_sizes[-1], feature_sizes[-1] * 2),
            nn.Dropout2d(p=dropout),
        )
        for feature_size in reversed(feature_sizes):
            self.ups.append(nn.ConvTranspose2d(feature_size * 2, feature_size,
kernel_size=2, stride=2, padding=0))
            self.ups.append(ConvBlock(feature_size * 2, feature_size))

        self.final_conv = nn.Sequential(
            nn.Conv2d(feature_sizes[0], out_channels, kernel_size=1),
            nn.Sigmoid()
        )
        self._initialize_weights()

    def forward(self, x):
        skips = []
        for down in self.downs:
            x = down(x)
            skips.append(x)
            x = self.pool(x)

        x = self.bottleneck(x)
        skips = skips[::-1]

        for i in range(0, len(self.ups), 2):
            x = self.ups[i](x)
            skip = skips[i // 2] # 透過 // 2 取得對應的 skip connection
            x = torch.cat([x, skip], dim=1) # 進行 skip connection
            x = self.ups[i + 1](x) # 進行 convolution
        return self.final_conv(x) # 輸出預測結果
```

```
    def _initialize_weights(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                # 使用 Kaiming initialization
                nn.init.kaiming_normal_(module.weight, mode='fan_out',
nonlinearity='relu')
                if module.bias is not None:
                    nn.init.constant_(module.bias, 0)
            elif isinstance(module, nn.BatchNorm2d):
                nn.init.constant_(module.weight, 1)
                nn.init.constant_(module.bias, 0)


if __name__ == '__main__':
    model = UNet(in_channels=3, out_channels=1)
    x = torch.randn(2, 3, 512, 512)
    print(model(x).shape)
    print(model)
```

After initial training, I observed suboptimal results, likely due to overfitting and unstable feature learning. To address this, I modified the U-Net by adding Dropout (p=0.1) to the bottleneck layer to regularize the network and reduce overfitting. Additionally, I introduced BatchNorm after each convolution to normalize activations, improving training stability and convergence. These common techniques significantly enhanced the model's segmentation performance and robustness. The enhancements are mentioned in the paper of Deep Residual Learning for Image Recognition (ResNet), althought the paper said that dropout is not used in the ResNet model, I still use dropout in the bottleneck to prevent overfitting because the dataset is small.

To make sure we can sovle binary segmentation problem with U-Net, I add a sigmoid function to the output of the model. The sigmoid function can make the output of the model between 0 and 1.

ResNet34+Unet

I don't understand the network artitecture of the ResNet34+Unet model in the lab instruction so I tend to remain the bottleneck and decoder part of the UNet model I implemented before. I use the ResNet34 model as the encoder part of the ResNet34+Unet model.

I follow the structure of ResNet34 to build the ResNet34 model. The ResNet34 model has 5 stages. The first stage is a 7x7 convolution layer with stride 2. The second stage is a maxpooling layer. The third stage is a combination of 3 residual blocks. The fourth stage is a combination of 4 residual blocks. The fifth stage is a combination of 6 residual blocks.

```
class ResSmallBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=None):
        super(ResSmallBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,
stride=2 if downsample is not None else 1, bias=False)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
```

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

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=None):
        super(ResidualBlock, self).__init__()

        if downsample is not None:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2,
bias=False),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.shortcut = nn.Identity()

        self.block = nn.Sequential(
            ResSmallBlock(in_channels, out_channels, downsample),
            ResSmallBlock(out_channels, out_channels),
        )
    def forward(self, x):
        return self.block(x) + self.shortcut(x)
```

After the last residual block, I don't use the maxpooling layer and fully connected layer because it is not the output of the model. I use the decoder part of the UNet model to build the ResNet34+Unet model. The decoder part of the UNet model is a combination of up-convolution and convolution block. With the experience of building the UNet model, I also add a dropout in the bottleneck layer and a BatchNorm after each convolution to normalize activations in the ResNet34+Unet model. I also add a sigmoid function to the output of the model to make sure the output of the model is between 0 and 1.

```
from .model_blocks import ResidualBlock, ConvBlock
import torch.nn as nn
import torch

class ResNetEncoder(nn.Module):
    def __init__(self, in_channels=3, layers=[3, 4, 6, 3], feature_sizes=[64, 128,
256, 512]):
        # ResNet34 的 block 數量分別為 [3, 4, 6, 3]
        super(ResNetEncoder, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3,
bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layers = nn.ModuleList()
```

```
        # 64 to 64
        self.layers.append(self._make_layer(ResidualBlock, feature_sizes[0],
feature_sizes[0], layers[0]))
        # 64 to 128
        self.layers.append(self._make_layer(ResidualBlock, feature_sizes[0],
feature_sizes[1], layers[1]))
        # 128 to 256
        self.layers.append(self._make_layer(ResidualBlock, feature_sizes[1],
feature_sizes[2], layers[2]))
        # 256 to 512
        self.layers.append(self._make_layer(ResidualBlock, feature_sizes[2],
feature_sizes[3], layers[3]))


    def _make_layer(self, block, in_channels, out_channels, counts):
        layer = [block(in_channels, out_channels, downsample=True if in_channels
!= out_channels else None)]
        for _ in range(1, counts):
            layer.append(block(out_channels, out_channels))
        return nn.Sequential(*layer)


    def forward(self, x):
        x1 = self.conv1(x)
        x1 = self.maxpool(x1)
        skips = []
        for layer in self.layers:
            x1 = layer(x1)
            skips.append(x1)
        return x1, skips

class UNetDecoder(nn.Module):
    def __init__(self, feature_sizes = [512, 256, 128, 64], out_channels=1,
dropout=0.1):
        super(UNetDecoder, self).__init__()

        # same as UNet upsample path
        self.bottleneck = nn.Sequential(
            ResidualBlock(feature_sizes[0], feature_sizes[0] * 2, True),
            nn.Dropout2d(p=dropout),
        )

        self.ups = nn.ModuleList()
        for feature_size in feature_sizes:
            self.ups.append(nn.ConvTranspose2d(feature_size * 2, feature_size,
kernel_size=2, stride=2))
            self.ups.append(ConvBlock(feature_size * 2, feature_size))

        self.final_conv = nn.Sequential(
            # 放大回原尺寸
            nn.ConvTranspose2d(feature_sizes[-1], feature_sizes[-1],
kernel_size=2, stride=2, bias=False),
            nn.BatchNorm2d(feature_sizes[-1]),
```

```
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(feature_sizes[-1], feature_sizes[-1],
kernel_size=2, stride=2, bias=False),
            nn.BatchNorm2d(feature_sizes[-1]),
            nn.ReLU(inplace=True),
            nn.Conv2d(feature_sizes[-1], out_channels, kernel_size=1),
            nn.Sigmoid()
        )

    def forward(self, x, skips):
        x = self.bottleneck(x)
        skips = skips[::-1]
        for i in range(0, len(self.ups), 2):
            x = self.ups[i](x)
            skip = skips[i // 2] # 透過 // 2 取得對應的 skip connection

            x = torch.cat([x, skip], dim=1) # 進行 skip connection
            x = self.ups[i + 1](x) # 進行 convolution
        return self.final_conv(x) # 輸出預測結果


class ResNet34UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super(ResNet34UNet, self).__init__()
        self.encoder = ResNetEncoder(in_channels=in_channels)
        self.decoder = UNetDecoder(out_channels=out_channels)
        self._initialize_weights()

    def forward(self, x):
        x, skips = self.encoder(x)
        return self.decoder(x, skips)

    def _initialize_weights(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                # 使用 Kaiming initialization
                nn.init.kaiming_normal_(module.weight, mode='fan_out',
nonlinearity='relu')
                if module.bias is not None:
                    nn.init.constant_(module.bias, 0)
            elif isinstance(module, nn.BatchNorm2d):
                nn.init.constant_(module.weight, 1)
                nn.init.constant_(module.bias, 0)

if __name__ == '__main__':
    model = ResNet34UNet()
    x = torch.randn(2, 3, 512, 512)
    print(model(x).shape)
```

Initialization

For both ResNet and U-Net, I initialized the weights using He initialization, which is mentioned in the paper of Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. He initialization is designed for ReLU activation functions, preventing vanishing or exploding gradients and improving convergence speed and performance. I used the default initialization in PyTorch, which implements He initialization for ReLU by default, ensuring optimal weight initialization for both models.

Model Training

```
def train(args):
    # 取出訓練集和驗證集
    train_data = load_dataset(args.data_path, 'train')
    train_data_loader = DataLoader(train_data, batch_size=args.batch_size,
shuffle=True)
    val_data = load_dataset(args.data_path, 'valid')
    val_data_loader = DataLoader(val_data, batch_size=args.batch_size,
shuffle=False)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # 選擇模型
    if args.model == 'unet':
        model = UNet(in_channels=3, out_channels=1).to(device)
    elif args.model == 'resnet':
        model = ResNet34UNet(in_channels=3, out_channels=1).to(device)

    # 優化器新增 L2 正則化項
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate,
weight_decay=args.weight_decay)

    # 使用 BCELoss 作為損失函數
    criterion = nn.BCEWithLogitsLoss()

    writer = SummaryWriter()

    best_score = 0.7
    best_model_path = "../saved_models/" + args.model + '_best_model.pth'

    for epoch in range(args.epochs):
        train_loss = []
        train_score = []
        model.train()
        progress_bar = tqdm.tqdm(enumerate(train_data_loader),
total=len(train_data_loader))
        for i, batch in progress_bar:
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            writer.add_images('images', images, i)
            writer.add_images('masks/true', masks, i)

            predictions = model(images) # 取得模型預測結果
            writer.add_images('masks/pred', predictions, i)
```

```
            loss = criterion(predictions, masks) + (dice_loss(predictions, masks))
# 計算損失值
            train_loss.append(loss.item()) # 將損失值加入 train_loss 中
            optimizer.zero_grad() # 梯度歸零
            loss.backward() # 反向傳播
            optimizer.step() # 更新權重
            with torch.no_grad():
                score = dice_score(predictions, masks)
                train_score.append(score.item())
            progress_bar.set_description(f'Epoch {epoch+1}/{args.epochs}, Loss:
{np.mean(train_loss):.4f}, Dice Score: {np.mean(train_score):.4f}')

            writer.add_scalar('Loss/train', np.mean(train_loss), epoch + 1)
            writer.add_scalar('Dice/train', np.mean(train_score), epoch + 1)

        validation_loss, validation_score = evaluate(model, val_data_loader,
device)
        writer.add_scalar('Loss/valid', validation_loss, epoch + 1)
        writer.add_scalar('Dice/valid', validation_score, epoch + 1)
        if validation_score > best_score: # 如果驗證集的分數比最佳分數還要高
            best_score = validation_score
            torch.save(model.state_dict(), best_model_path) # 儲存模型權重
            print(f"Model saved at {best_model_path}")
    writer.close()
```

I opted for Adam over SGD to leverage its faster convergence and robustness, addressing SGD's limitations in learning rate tuning and gradient adaptability. The training method in the U-Net paper, designed for multi-class segmentation with weighted cross-entropy, does not align with this assignment's binary foreground-background separation task evaluated using Dice score. For this two-class problem, combining BCE with Dice Loss is more suitable, as it directly optimizes for overlap metrics like Dice, improving performance over the paper's approach. I learned dropout and L2 regularization from here. I use L2 regularization to calculate the loss to prevent overfitting.

Most important enhancements are the use of modern initialization (He) and normalization (BatchNorm), which significantly improve training stability and convergence. These enhancements are not present in the original U-Net, designed for biomedical images, but are common in modern architectures like ResNet. By incorporating these techniques, I adapted models to natural image tasks, enhancing their performance and usability.

Evaluation

```
def evaluate(net, data, device):
    validation_loss = []
    validation_dice = []

    criterion = nn.BCEWithLogitsLoss()
    with torch.no_grad():
        net.eval()
        for batch in data:
            images = batch['image'].to(device)
```

```
            masks = batch['mask'].to(device)

            predictions = net(images)
            loss = criterion(predictions, masks) + dice_loss(predictions, masks)
            validation_loss.append(loss.item())

            score = dice_score(predictions, masks)
            validation_dice.append(score.item())
        print(f'Validation Loss: {np.mean(validation_loss):.4f}, Dice Score:
{np.mean(validation_dice):.4f}')
    return np.mean(validation_loss), np.mean(validation_dice)
```

In the evaluation function, I calculate the loss and dice score of the model on the validation set. The loss is calculated by the BCEWithLogitsLoss and dice loss. The dice score is calculated by the dice score function.

Inference

```
def inference(args):
    if args.model == "unet":
        model = UNet(in_channels=3, out_channels=1)
    else:
        model = ResNet34UNet(in_channels=3, out_channels=1)

    state_dict = torch.load("../saved_models/" + args.model + '_best_model.pth',
weights_only=True)
    model.load_state_dict(state_dict)  # 將參數載入模型
    model.eval()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    data = load_dataset(args.data_path, 'test')
    data_loader = torch.utils.data.DataLoader(data, batch_size=args.batch_size,
shuffle=False)

    dice_scores = []
    progress_bar = tqdm.tqdm(enumerate(data_loader), total=len(data_loader))
    for i, batch in progress_bar:
        images = batch['image'].to(device)
        masks = batch['mask'].to(device)
        predictions = model(images) # 取得模型預測結果
        dice = dice_score(predictions, masks)
        dice_scores.append(dice.item())
    print(f'Model: {args.model}')
    print(f'Dice Score: {np.mean(dice_scores):.4f}')
```

I use the best model to do the inference. The best model is the model with the highest dice score on the validation set. I load the best model and use it to do the inference. I calculate the dice score of the model on the test set. The higher the dice score, the better the model. I only load the weights of the model because the model architecture is the same as the model I trained and for security reasons.

Dice Score

```python
# in utils.py
import torch
import numpy as np

def dice_score(pred_mask, gt_mask):
    """
    Calculate Dice score between predicted mask and ground truth mask.

    Args:
        pred_mask (torch.Tensor): Predicted mask (binary or probabilities)
        gt_mask (torch.Tensor): Ground truth mask (binary)

    Returns:
        torch.Tensor: Dice score as a scalar tensor
    """
    # 確保輸入是 Tensor
    if not isinstance(pred_mask, torch.Tensor):
        pred_mask = torch.tensor(pred_mask, dtype=torch.float32)
    if not isinstance(gt_mask, torch.Tensor):
        gt_mask = torch.tensor(gt_mask, dtype=torch.float32)

    assert pred_mask.max() <= 1 and pred_mask.min() >= 0, "pred_mask out of range"
    assert gt_mask.max() <= 1 and gt_mask.min() >= 0, "gt_mask out of range"

    # 計算交集 (intersection)
    intersection = torch.sum(pred_mask * gt_mask)

    # 計算聯集 (union)
    union = torch.sum(pred_mask) + torch.sum(gt_mask)
    eps = 1e-8 # 避免分母為 0
    # 計算 Dice 分數
    dice_score = 2 * intersection / (union + eps)

    return dice_score

def dice_loss(pred_mask, gt_mask):
    """
    Calculate Dice loss between predicted mask and ground truth mask.

    Args:
        pred_mask (torch.Tensor): Predicted mask (binary or probabilities)
        gt_mask (torch.Tensor): Ground truth mask (binary)

    Returns:
        torch.Tensor: Dice loss as a scalar tensor
    """
    # 計算 Dice Score
    score = dice_score(pred_mask, gt_mask)
    # 將 Dice Score 轉成 Dice Loss
    dice_loss = 1 - score
    return dice_loss
```

I use tensor to calculate the dice score and dice loss. The dice score is the intersection of the predicted mask and the ground truth mask divided by the union of the predicted mask and the ground truth mask. The dice loss is 1 minus the dice score. The dice score is used to evaluate the performance of the model. The higher the dice score, the better the model.

2. Data Preprocessing

```
def load_dataset(data_path, mode):
    # implement the load dataset function here

    # check if the dataset folder has images and annotations
    images_path = os.path.join(data_path, "images")
    annotations_path = os.path.join(data_path, "annotations")
    if not os.path.exists(images_path) or not os.path.exists(annotations_path):
        OxfordPetDataset.download(data_path)

    # 將影像資料轉換成 tensor，並統一尺寸，訓練集增加多種轉換
    if mode == "train":
        transform = A.Compose([
            A.Resize(512, 512),
            A.Affine(
                translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)},
                scale=(0.8, 1.2),
                rotate=(-30, 30),
                p=0.5
            ),
            A.Normalize(mean=(0.481, 0.449, 0.396), std=(0.269, 0.265, 0.273)),
            ToTensorV2()
        ], additional_targets={"mask": "mask"})
    else:
        transform = A.Compose([
            A.Resize(512, 512),
            A.Normalize(mean=(0.481, 0.449, 0.396), std=(0.269, 0.265, 0.273)),
            ToTensorV2()
        ], additional_targets={"mask": "mask"})

    dataset = OxfordPetDataset(data_path, mode=mode, transform=transform)
    return dataset
```

I resize images and masks to 512×512, applies random affine transformations (translation, scaling, rotation) with 50% probability, normalizes the image, and converts both to PyTorch tensors, ensuring synchronized geometric changes for segmentation tasks.

My preprocessing pipeline extends U-Net's elastic deformation with additional augmentations and modern RGB handling (fixed 512×512 size, normalization), suited for natural image tasks. U-Net, designed for biomedical images, uses simpler augmentation and lacks explicit normalization or tensor conversion.

Second, I consider the edges within the images as the foreground class for the segmentation task. Specifically, I process the annotations such that the edge regions—typically representing boundaries between objects and the background—are assigned to the foreground label (value 1). All other regions, including the interior of

objects and the background, are treated as the background class (value 0). This binary classification approach focuses the model's attention on accurately delineating edge structures, which are critical for precise segmentation.

In the final experiment of UNet, I find this webste before I go into ResNet34+Unet. I calculate the mean and std of the dataset and use them to normalize the dataset instead of using the default mean and std in the albumentations library.

```
def calculate_mean_and_std(data_path):
    # implement the calculate mean and std function here
    images_path = os.path.join(data_path, "images")
    means = np.zeros(3)
    stds = np.zeros(3)
    total_pixels = 0
    images = []
    progess_bar = tqdm(os.listdir(images_path))
    for filename in progess_bar:
        image_path = os.path.join(images_path, filename)
        # 確認是否為圖片檔
        if not image_path.endswith(".jpg"):
            continue
        image = np.array(Image.open(image_path).convert("RGB"))
        images.append(image)
        total_pixels += np.prod(image.shape[:2])
        means += np.mean(image, axis=(0, 1))
    means /= len(images)
    progess_bar.set_description("Calculating stds")
    for image in images:
        stds += np.sum((image - means) ** 2, axis=(0, 1))
    stds = np.sqrt(stds / total_pixels)
    means = means / 255
    stds = stds / 255
    return means, stds

if __name__ == "__main__":
    data_path = "../dataset/oxford-iiit-pet"
    means, stds = calculate_mean_and_std(data_path)
    print(f"Means: {means}, Stds: {stds}")
```

3. Analyze the experiment results

In this lab, I use tensorboard to visualize the training process.

UNet

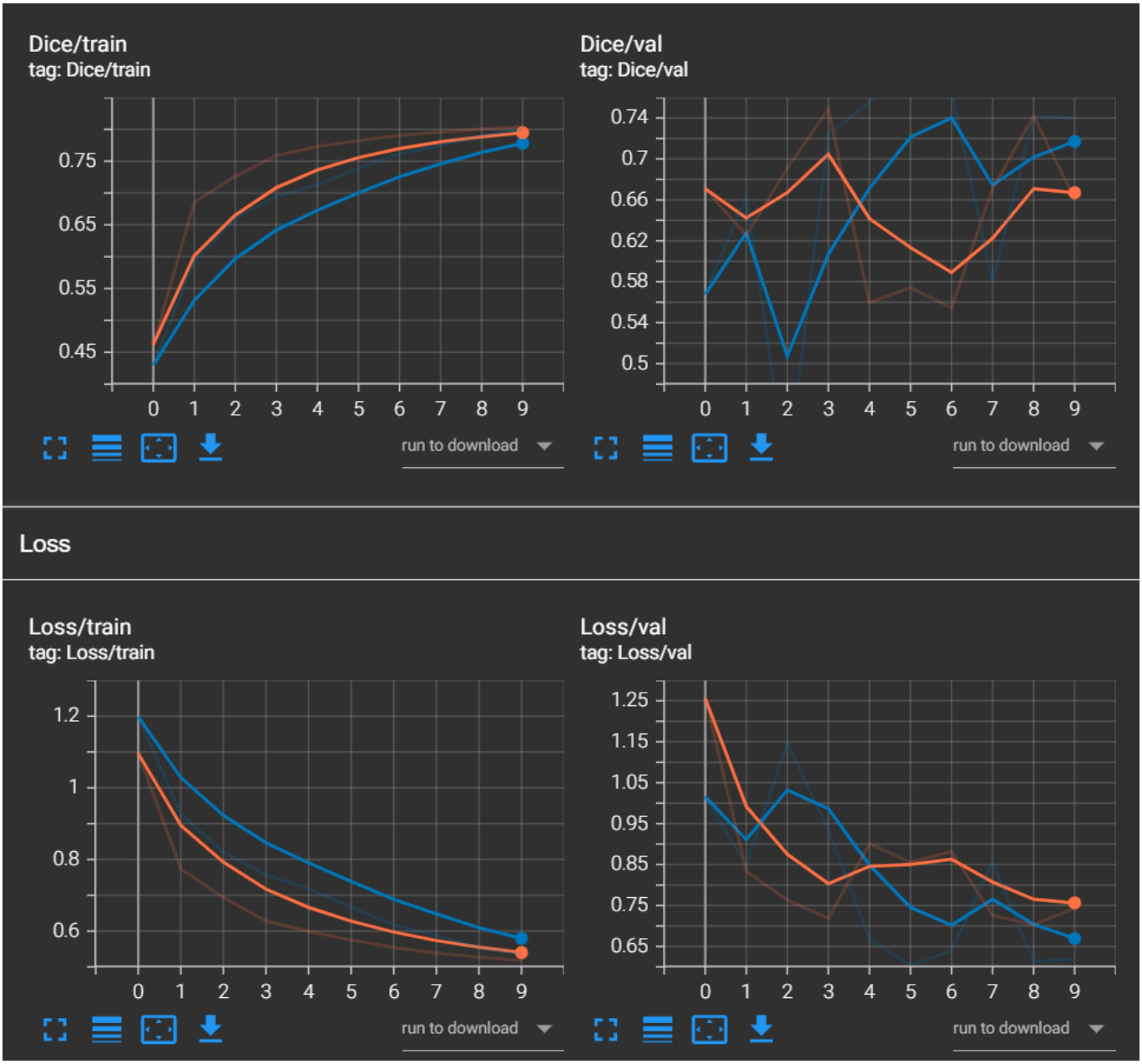
I first train the UNet model with the following hyperparameters:

- Batch size: 1
- Learning rate: 0.001
- Weight decay: 0.0002

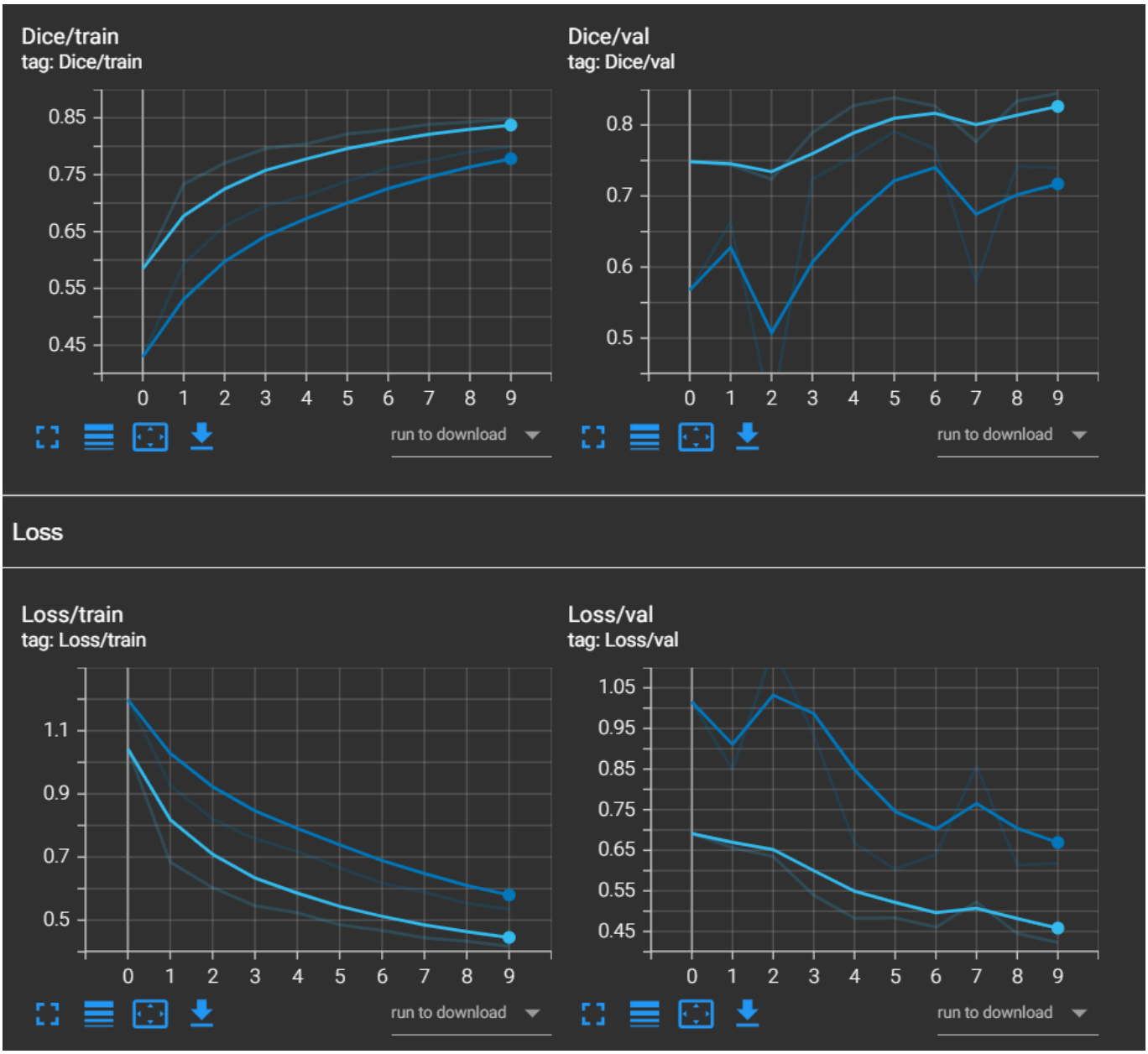
- Epochs: 10

The training loss and validation loss are shown in the following figure.

The orange line is showing the performance of the one with original initialization, and the blue line is showing the performance of the one with He initialization. The model with He initialization has a lower accuracy at the beginning, but it can converge quickly and achieve a better performance on the validation set after 10 epochs. The model with original initialization has a higher accuracy at the beginning, but it has a little bit overfitting problem.



After adjusting the learning rate to 0.0001, the training loss and validation loss are shown in the following figure.



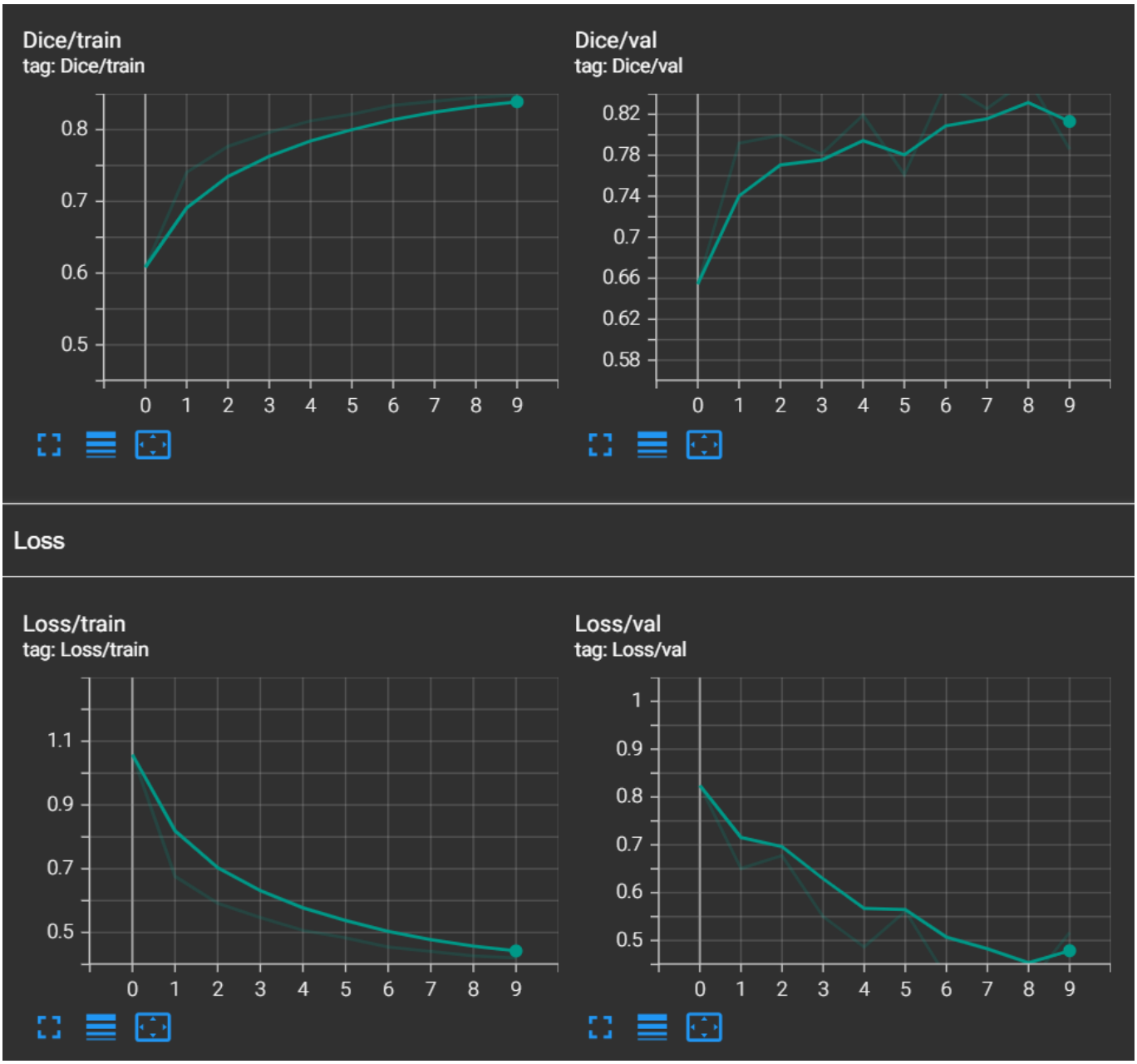
The light blue line is showing the performance of the one with learning rate 0.0001, and the dark blue line is showing the performance of the one with learning rate 0.001.

After I adjust the learning rate to 0.0001, the model can reach the baseline (Dice score 0.8448) after 10 epochs. The model with learning rate 0.001 can not reach the baseline after 10 epochs.

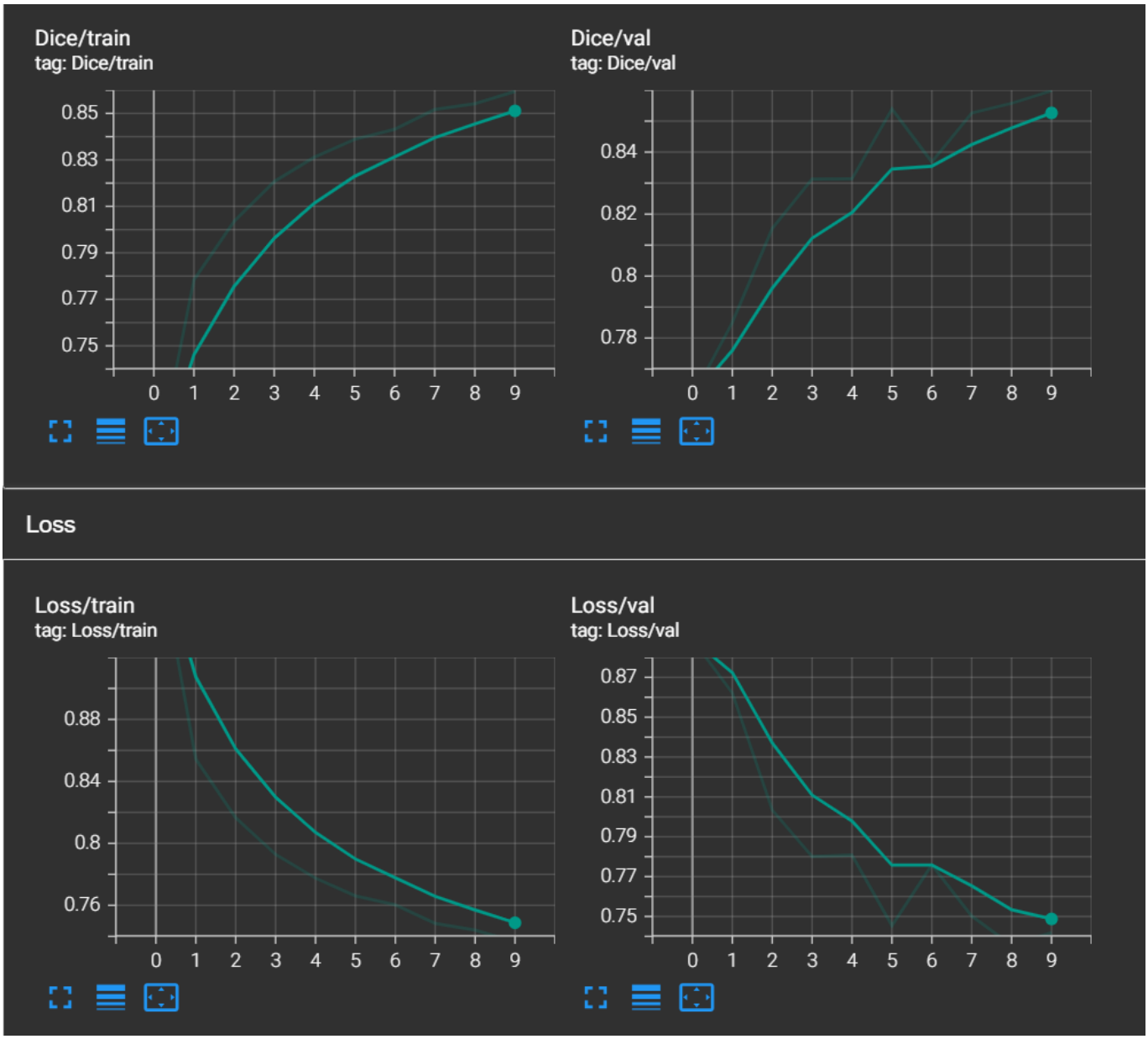
```
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$ python inference.py
100%| | 459/459 [00:48<00:00,  9.43it/s]
Model: unet
Dice Score: 0.8448
```

In the final experiment, I adjust the mean and std of the dataset to normalize the dataset. The idea is from this webste. I use the function `calculate_mean_and_std` in `oxford_pet.py` of `src` directory to calculate the mean and std of the dataset. The mean and std are [0.481, 0.449, 0.396] and [0.269, 0.265, 0.273]. I use these mean and std to normalize the dataset. Also, I figure out that I should add a sigmoid function to the output of the model to make sure the output of the model is between 0 and 1 in stead of binarize the output of the model. The training loss and validation loss are shown in the following figure with dark green line.

This figure shows that If I don't the sigmoid function to the output of the model, the model is good but not good enough.



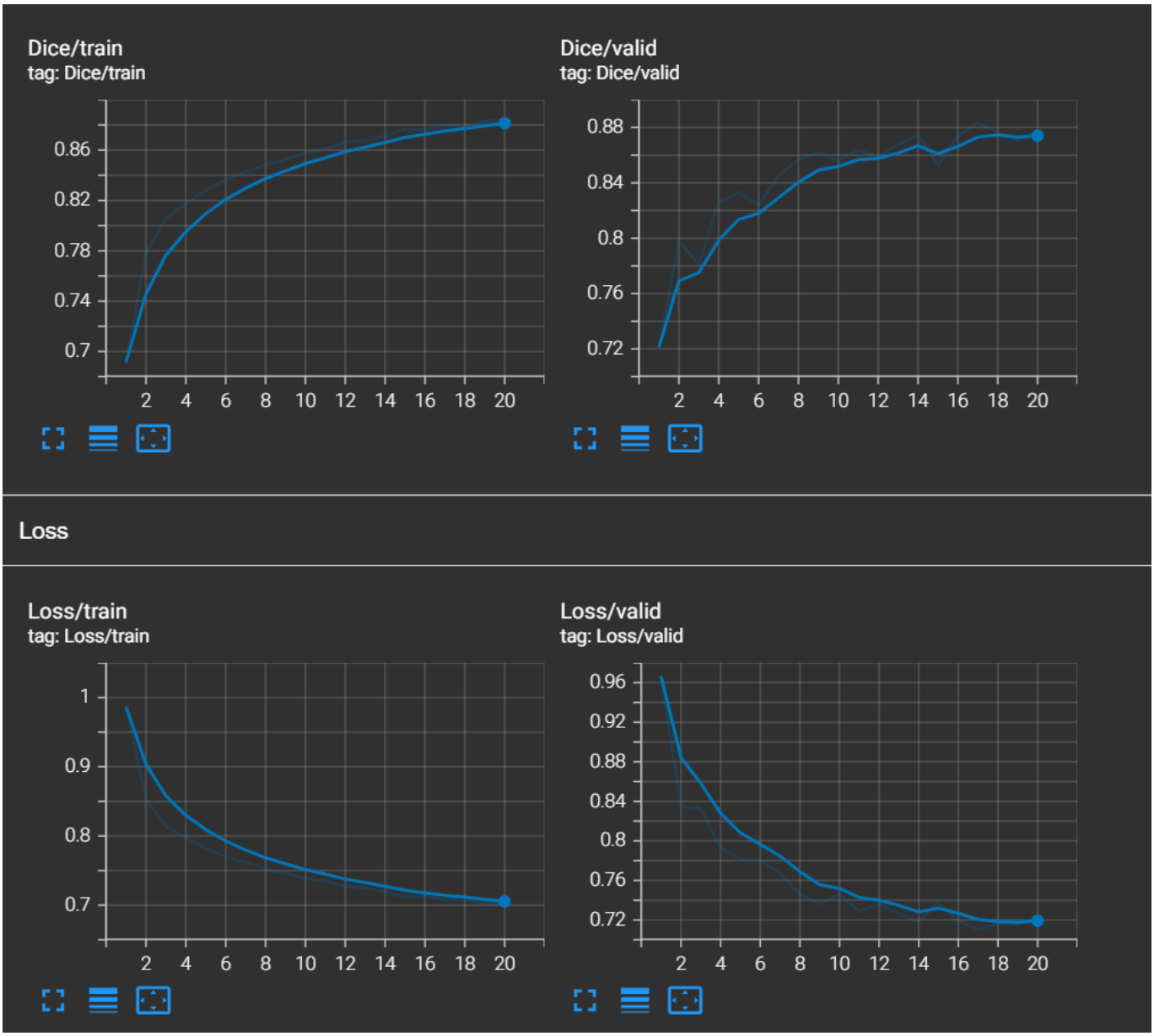
If I add the sigmoid function to the output of the model, the model can reach the baseline after 10 epochs. The model with sigmoid function has a better performance than the model without sigmoid function.



The final Dice score of the UNet model is 0.8641.

```
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$ python inference.py
100%|| 459/459 [00:48<00:00,  9.43it/s]
Model: unet
Dice Score: 0.8641
```

I train the UNet model with more epochs, the model can reach the baseline after 20 epochs. The final dice score of the UNet model is 0.8885.



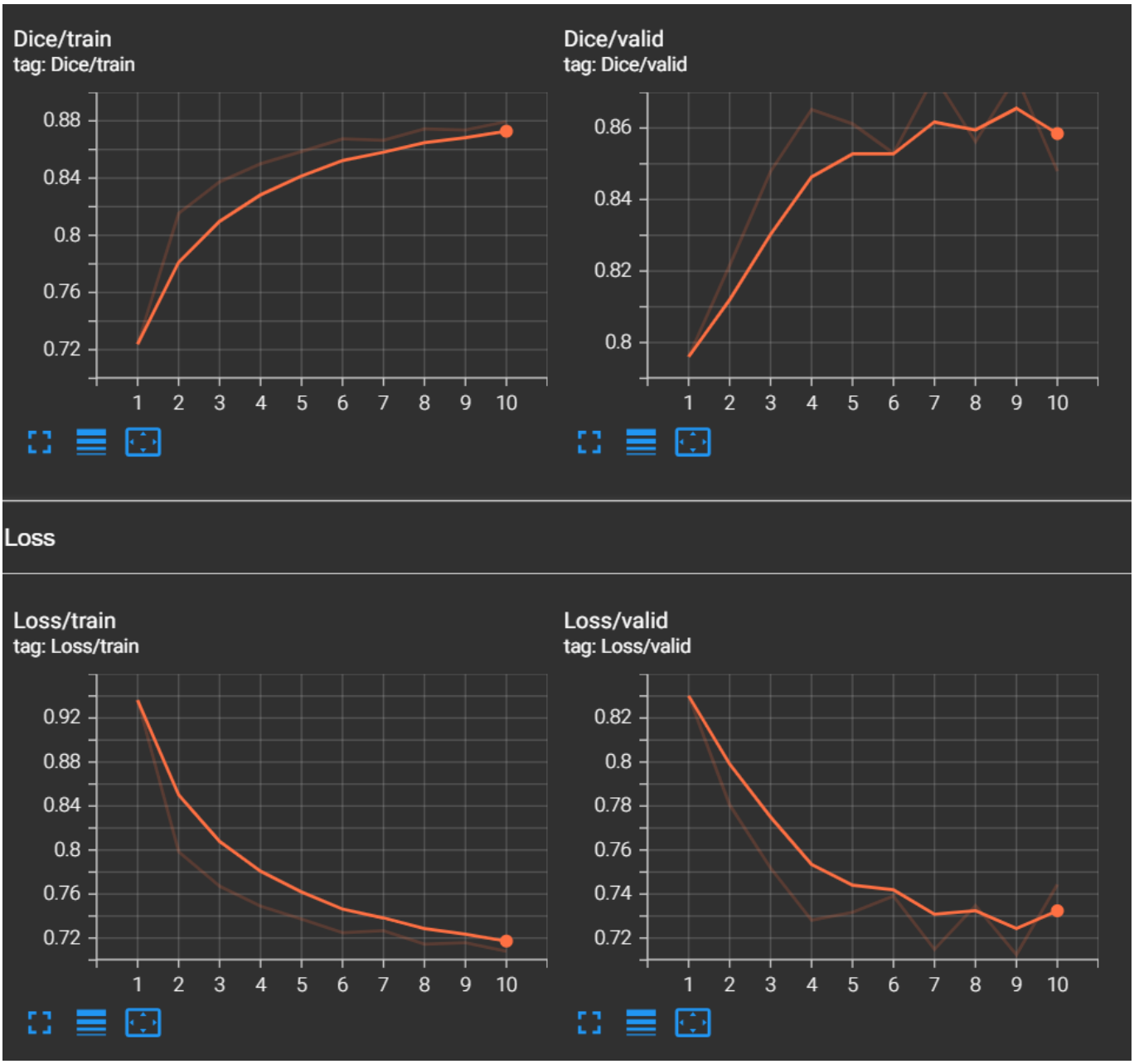

```
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$ python inference.py -m unet
100%|                                                                                | 459/459 [00:48<00:00,  9.42it/s]
Model: unet
Dice Score: 0.8885
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$
```

Final hyperparameters

- Batch size: 8
- Learning rate: 0.0001
- Weight decay: 0.0002
- Epochs: 20

ResNet34+UNet

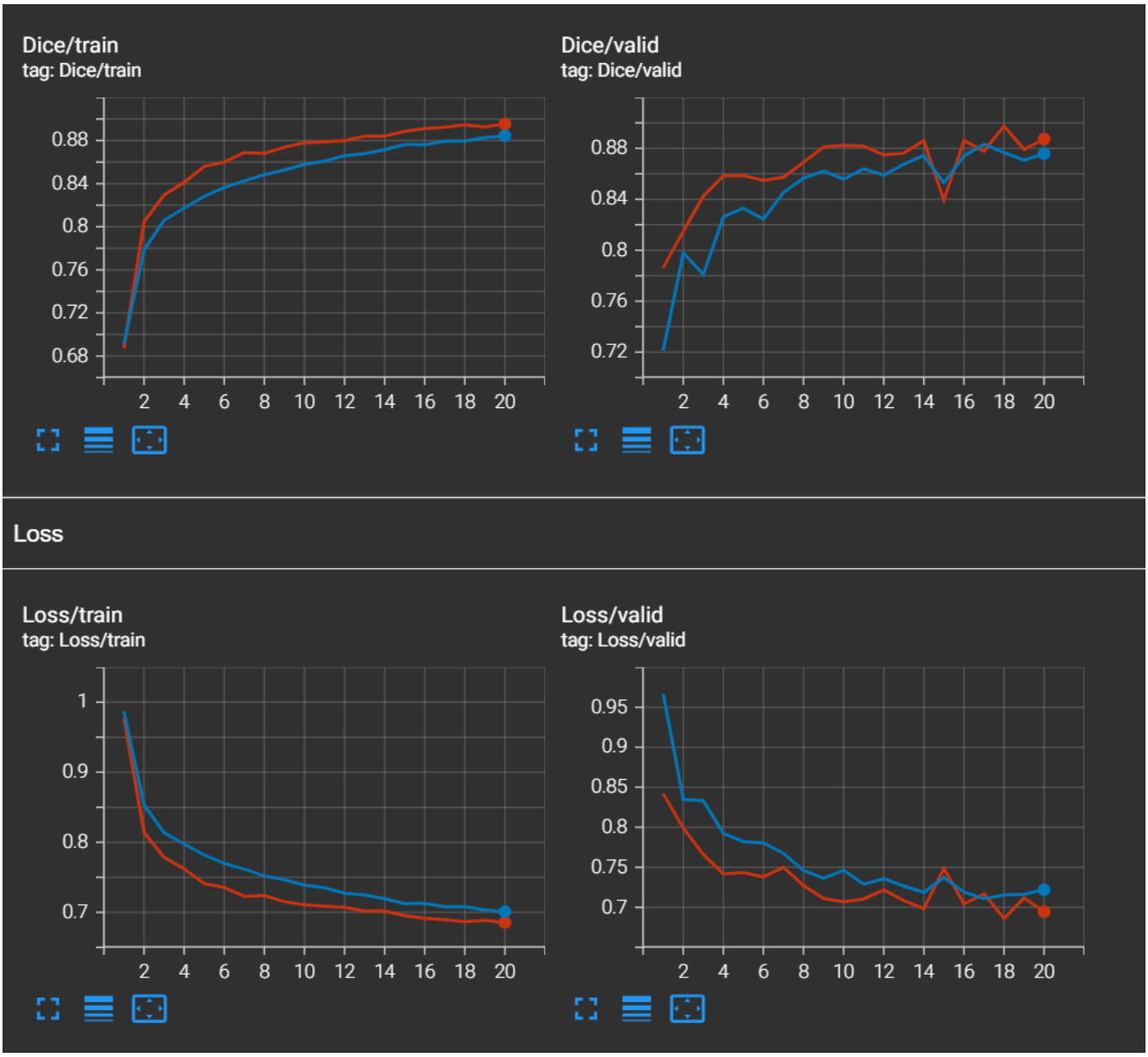
I have the experience of training the UNet model, so I use the same initialization method to train the ResNet34+Unet model. The training loss and validation loss are shown in the following figure.



The dice score of the ResNet34+Unet model is 0.8775 in 10 epochs.

```
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$ python inference.py -m resnet
100%|                                                                                | 459/459 [00:19<00:00, 23.55it/s]
Model: resnet
Dice Score: 0.8775
```

Compare to the UNet model (blue line) in 20 epochs, the ResNet34+Unet (red line) model has a better performance. The ResNet34+Unet model is always better than the UNet model in the training process.



The dice score of the ResNet34+Unet model is 0.9085 in 20 epochs.

```
(.venv) (base) jason@MS-7D96:~/Desktop/TAICA_DL_Labs/Lab2/src$ python inference.py -m resnet
100%|                                                                                | 459/459 [00:19<00:00, 23.64it/s]
Model: resnet
Dice Score: 0.9085
```

Final hyperparameters

- Batch size: 8
- Learning rate: 0.0001
- Weight decay: 0.0002
- Epochs: 20

4. Execution steps

First, go to the `src` directory.

```
cd src
```

Unet

To train the Unet model, run the following command.

```
python train.py --model unet --batch_size 8 --lr 0.0001 --weight_decay 0.0002 --
epochs 20
```

To test the Unet model, run the following command.

```
python inference.py --model unet
```

To train the Resnet34+Unet model, run the following command.

```
python train.py --model resnet --batch_size 8 --lr 0.0001 --weight_decay 0.0002 --
epochs 20
```

To test the Resnet34+Unet model, run the following command.

```
python inference.py --model resnet
```

5. Discussion

In the last lab, I practice building two simple fully connected neural networks. In this lab, I face many difficulties in building the UNet model and the ResNet34+Unet model. I learn a lot from building these two models.

In the data preprocessing phase, I learn that adding transformations to the training set can improve the performance of the model. The transformations can help the model to learn the features of the dataset better with different angles, scales, and rotations. The transformations can also help the model to prevent overfitting. I also learn that the mean and std of the dataset are important for the normalization of the dataset. The normalization of the dataset can help the model to learn the features of the dataset better and faster.

In the model building phase, I learn that the initialization method is very important for the performance of the model. The He initialization is designed for ReLU activation functions, preventing vanishing or exploding gradients and improving convergence speed and performance. I also learn that the dropout and BatchNorm are important for the performance of the model. The dropout can prevent overfitting and the BatchNorm can normalize activations, improving training stability and convergence. I also learn that the sigmoid function is important for the output of the model. The sigmoid function can make the output of the model between 0 and 1.

I am a newbie in deep learning, so I don't have much experience in building deep learning models. I learn a lot from building the UNet model and the ResNet34+Unet model. L2 regularization is a new concept for me. I think L2 regularization is important for the performance of the model. This method add a penalty term to the loss function to prevent overfitting.

Adding dice loss to the loss function also plays an important role in the performance of the model. Because our goal is to solve the binary segmentation problem and evaluate the model with the dice score, adding dice loss to the loss function can help the model to learn what is a good segmentation result.

In a same hyperparameters setting, the ResNet34UNet model training time is less than the UNet model. The ResNet34UNet model converges faster than the UNet model. The ResNet34UNet model has a better performance than the UNet model.

From the experiment results, I think I learning a lot from building such a residual network. It did well compared to a only Unet model. I think the ResNet34+Unet model is a good choice for the binary segmentation problem between the UNet model and the ResNet34+Unet model. It shows that the residual network can improve the performance of deep learning models.

6. Reference

1. 【调教神经网络咋这么难？【白话DeepSeek03】】 https://www.bilibili.com/video/BV1RqXRYDEe2/?share_source=copy_web&vd_source=8eb0208b6e349b456c095c16067fb3af
2. Unet https://arxiv.org/abs/1505.04597
3. ResNet https://arxiv.org/abs/1512.03385
4. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification https://arxiv.org/abs/1502.01852
5. PyTorch計算dataset的mean和std https://yanwei-liu.medium.com/pytorch%E8%A8%88%E7%AE%97dataset%E7%9A%84mean%E5%92%8Cstd-ecadb63420ca
6. Why Pytorch officially use mean=[0.485, 0.456, 0.406] and std=[0.229, 0.224, 0.225] to normalize images? https://stackoverflow.com/questions/58151507/why-pytorch-officially-use-mean-0-485-0-456-0-406-and-std-0-229-0-224-0-2
7. 【PyTorch教程】P7. TensorBoard的使用（一） https://www.youtube.com/watch?v=d_1wVaRsCDE&t=1s
8. 【PyTorch教程】P7. TensorBoard的使用（二） https://www.youtube.com/watch?v=udAfnTvRD4A