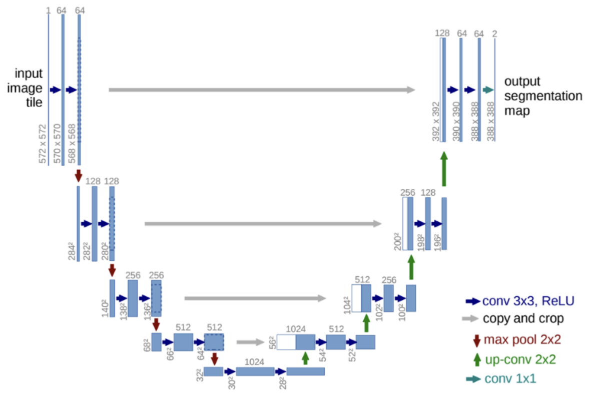
Overview

This project involves the implementation of two deep learning models for image segmentation tasks: a standard U-Net and a ResNet34-based U-Net. Both models are designed to perform pixel-wise classification, which is essential for tasks such as medical image analysis, satellite image segmentation, and more.

Implementation Details

UNet

Architecture of UNet



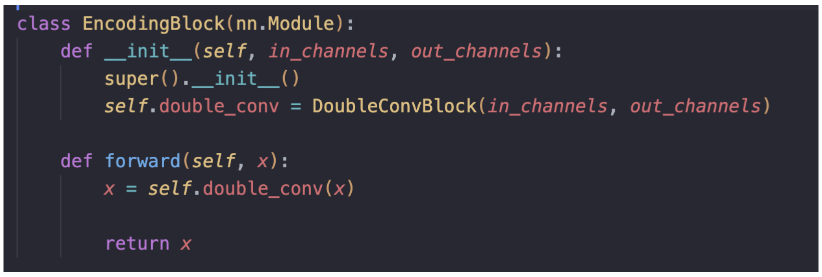
Double Convolution Block

This class is defined because both UNet and ResNet34_UNet frequently use it.

```python
class DoubleConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, kernel_size=kernel_size, stride=1, padding=padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        return x
```

Encoding Block

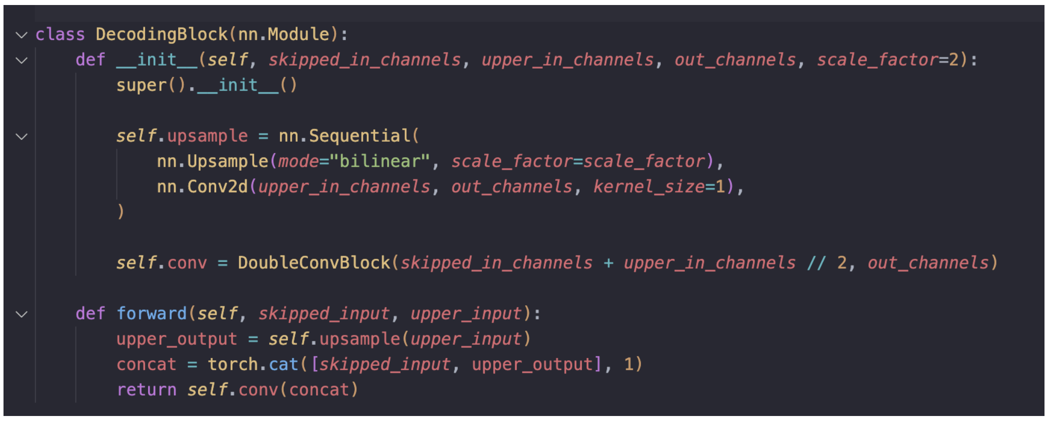
Encoding block simply consists of a double convolution block.



```python
class EncodingBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = DoubleConvBlock(in_channels, out_channels)

    def forward(self, x):
        x = self.double_conv(x)

        return x
```

DecodingBlock

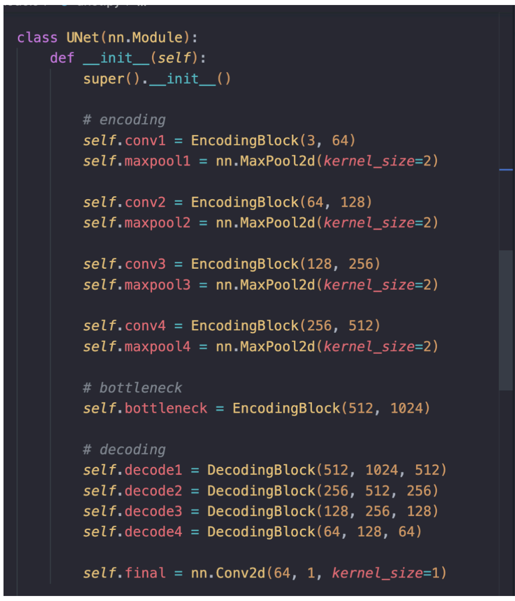
In the decoding block, the input from the upper layer is first upsampled and applied with single convolution.
After the upsampling step, the upsampled input from the upper layer is concatenated with the corresponding part from the encoding phase. This process is known as a skip connection.



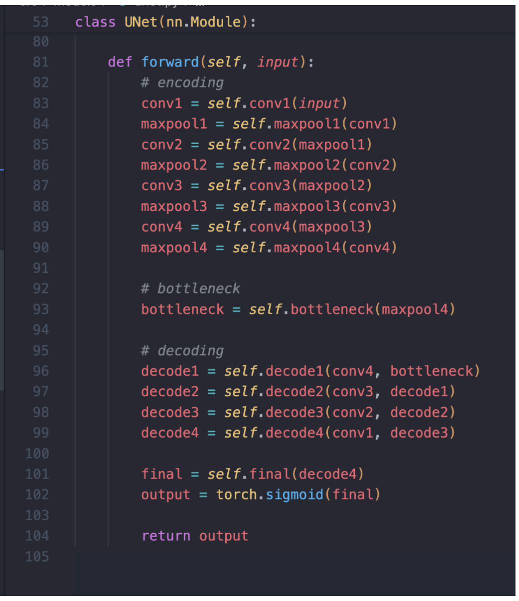
```python
class DecodingBlock(nn.Module):
    def __init__(self, skipped_in_channels, upper_in_channels, out_channels, scale_factor=2):
        super().__init__()

        self.upsample = nn.Sequential(
            nn.Upsample(mode="bilinear", scale_factor=scale_factor),
            nn.Conv2d(upper_in_channels, out_channels, kernel_size=1),
        )

        self.conv = DoubleConvBlock(skipped_in_channels + upper_in_channels // 2, out_channels)

    def forward(self, skipped_input, upper_input):
        upper_output = self.upsample(upper_input)
        concat = torch.cat([skipped_input, upper_output], 1)
        return self.conv(concat)
```

UNet

The encoding part of the UNet is made up of 4 encoding blocks and max pooling layer pairs.
The decoding part of the UNet is made up of 4 decoding blocks defined above.



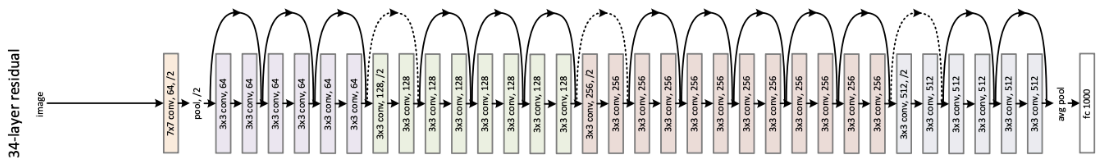
```python
class UNet(nn.Module):
    def __init__(self):
        super().__init__()

        # encoding
        self.conv1 = EncodingBlock(3, 64)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        self.conv2 = EncodingBlock(64, 128)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        self.conv3 = EncodingBlock(128, 256)
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)

        self.conv4 = EncodingBlock(256, 512)
        self.maxpool4 = nn.MaxPool2d(kernel_size=2)

        # bottleneck
        self.bottleneck = EncodingBlock(512, 1024)

        # decoding
        self.decode1 = DecodingBlock(512, 1024, 512)
        self.decode2 = DecodingBlock(256, 512, 256)
        self.decode3 = DecodingBlock(128, 256, 128)
        self.decode4 = DecodingBlock(64, 128, 64)

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



```python
class UNet(nn.Module):

    def forward(self, input):
        # encoding
        conv1 = self.conv1(input)
        maxpool1 = self.maxpool1(conv1)
        conv2 = self.conv2(maxpool1)
        maxpool2 = self.maxpool2(conv2)
        conv3 = self.conv3(maxpool2)
        maxpool3 = self.maxpool3(conv3)
        conv4 = self.conv4(maxpool3)
        maxpool4 = self.maxpool4(conv4)

        # bottleneck
        bottleneck = self.bottleneck(maxpool4)

        # decoding
        decode1 = self.decode1(conv4, bottleneck)
        decode2 = self.decode2(conv3, decode1)
        decode3 = self.decode3(conv2, decode2)
        decode4 = self.decode4(conv1, decode3)

        final = self.final(decode4)
        output = torch.sigmoid(final)

        return output
```

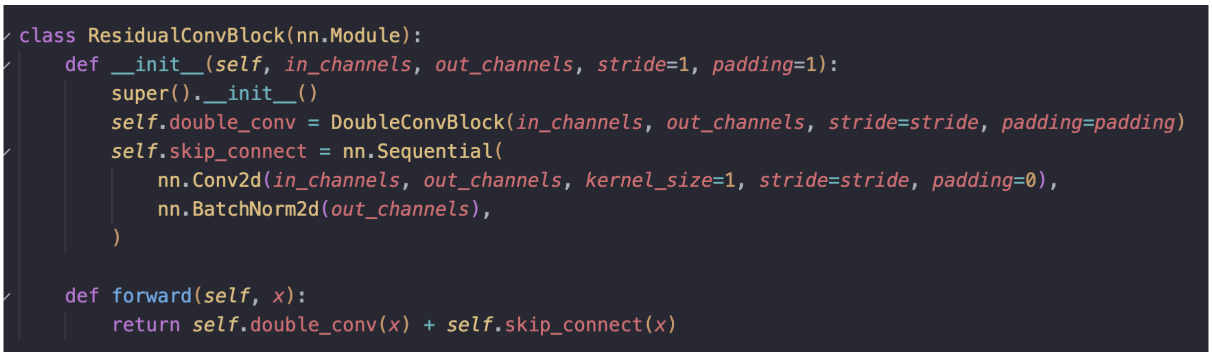
ResNet34_UNet

Architecture of ResNet34



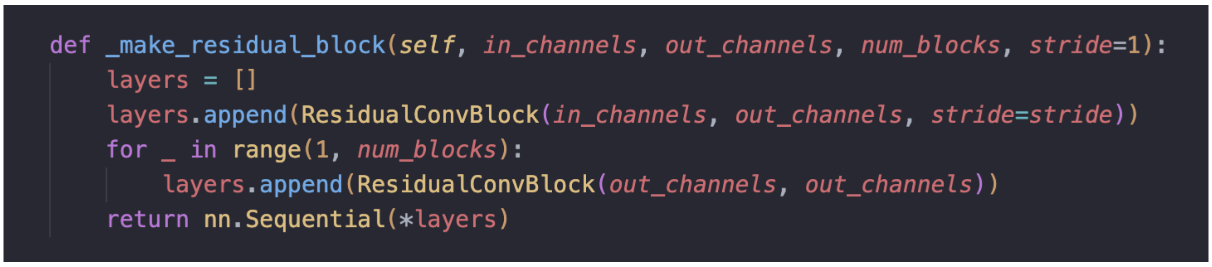
Residual Convolutional Block & Residual Layer

The Residual Convolution Block is made up of a double convolution block defined above and a skipped residual connection.



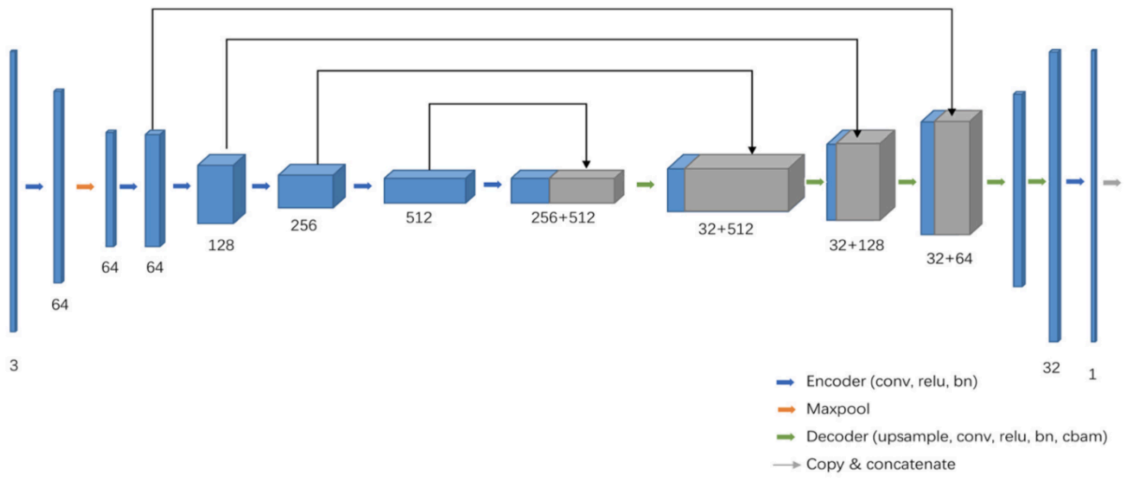
```python
class ResidualConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, padding=1):
        super().__init__()
        self.double_conv = DoubleConvBlock(in_channels, out_channels, stride=stride, padding=padding)
        self.skip_connect = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, padding=0),
            nn.BatchNorm2d(out_channels),
        )

    def forward(self, x):
        return self.double_conv(x) + self.skip_connect(x)
```

`_make_residual_block` is a helper function that return assembled residual blocks.



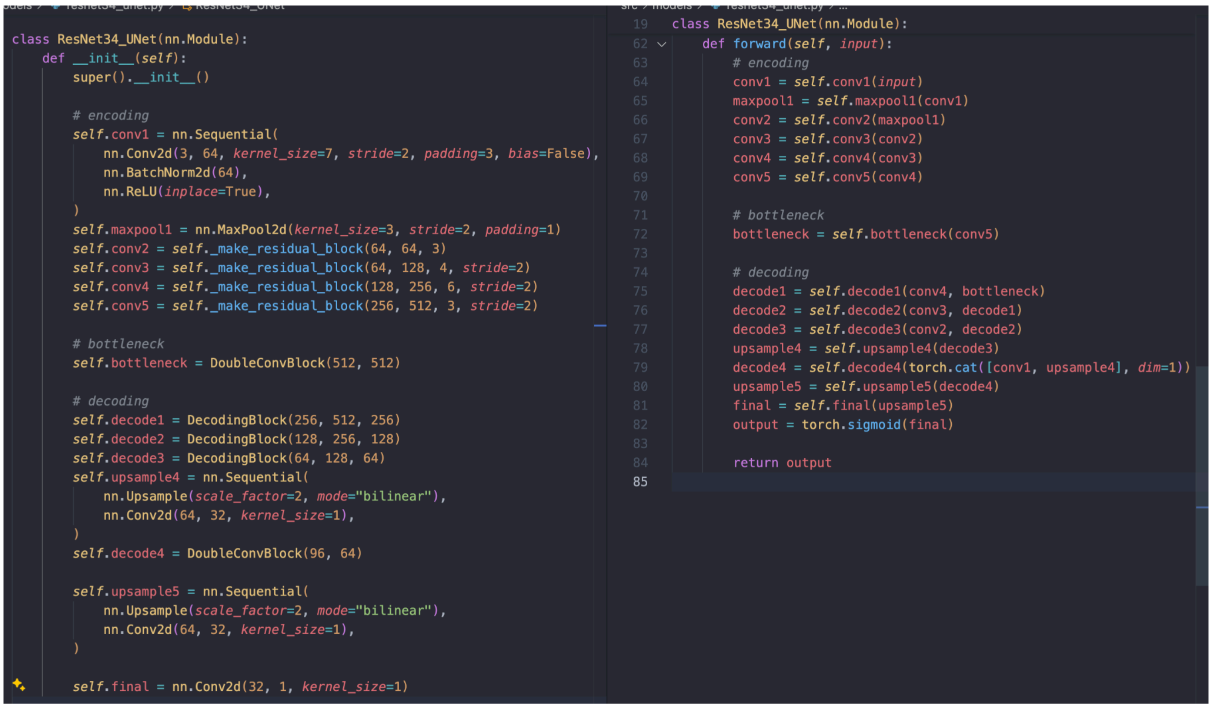
```python
def _make_residual_block(self, in_channels, out_channels, num_blocks, stride=1):
    layers = []
    layers.append(ResidualConvBlock(in_channels, out_channels, stride=stride))
    for _ in range(1, num_blocks):
        layers.append(ResidualConvBlock(out_channels, out_channels))
    return nn.Sequential(*layers)
```

Architecture of ResNet34_UNet



Implementation of ResNet34_UNet

The implementation is mostly identical to UNet, except the encoding block is replaced with multiple residual layers.

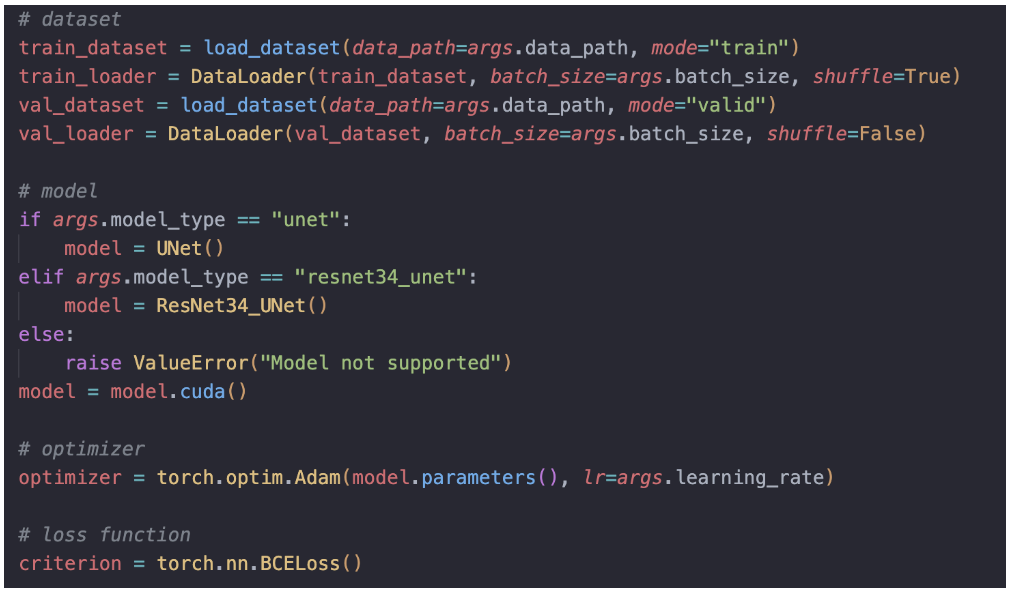


```python
class ResNet34_UNet(nn.Module):
    def __init__(self):
        super().__init__()

        # encoding
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
        )
        self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = self._make_residual_block(64, 64, 3)
        self.conv3 = self._make_residual_block(64, 128, 4, stride=2)
        self.conv4 = self._make_residual_block(128, 256, 6, stride=2)
        self.conv5 = self._make_residual_block(256, 512, 3, stride=2)

        # bottleneck
        self.bottleneck = DoubleConvBlock(512, 512)

        # decoding
        self.decode1 = DecodingBlock(256, 512, 256)
        self.decode2 = DecodingBlock(128, 256, 128)
        self.decode3 = DecodingBlock(64, 128, 64)
        self.upsample4 = nn.Sequential(
            nn.Upsample(scale_factor=2, mode="bilinear"),
            nn.Conv2d(64, 32, kernel_size=1),
        )
        self.decode4 = DoubleConvBlock(96, 64)

        self.upsample5 = nn.Sequential(
            nn.Upsample(scale_factor=2, mode="bilinear"),
            nn.Conv2d(64, 32, kernel_size=1),
        )

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

```python
class ResNet34_UNet(nn.Module):
    def forward(self, input):
        # encoding
        conv1 = self.conv1(input)
        maxpool1 = self.maxpool1(conv1)
        conv2 = self.conv2(maxpool1)
        conv3 = self.conv3(conv2)
        conv4 = self.conv4(conv3)
        conv5 = self.conv5(conv4)

        # bottleneck
        bottleneck = self.bottleneck(conv5)

        # decoding
        decode1 = self.decode1(conv4, bottleneck)
        decode2 = self.decode2(conv3, decode1)
        decode3 = self.decode3(conv2, decode2)
        upsample4 = self.upsample4(decode3)
        decode4 = self.decode4(torch.cat([conv1, upsample4], dim=1))
        upsample5 = self.upsample5(decode4)
        final = self.final(upsample5)
        output = torch.sigmoid(final)

        return output

```

Training Code

- Initialize the dataloader of training dataset and validation dataset.
- Initialize the model according to the CLI arguments.
- Initialize the optimizer `Adam`.
- Initialize the loss function `BCELoss`. Since `softmax` is already applied in the final layer of both models, `BCELoss` is adopted instead of `BCEWithLogitsLoss`.

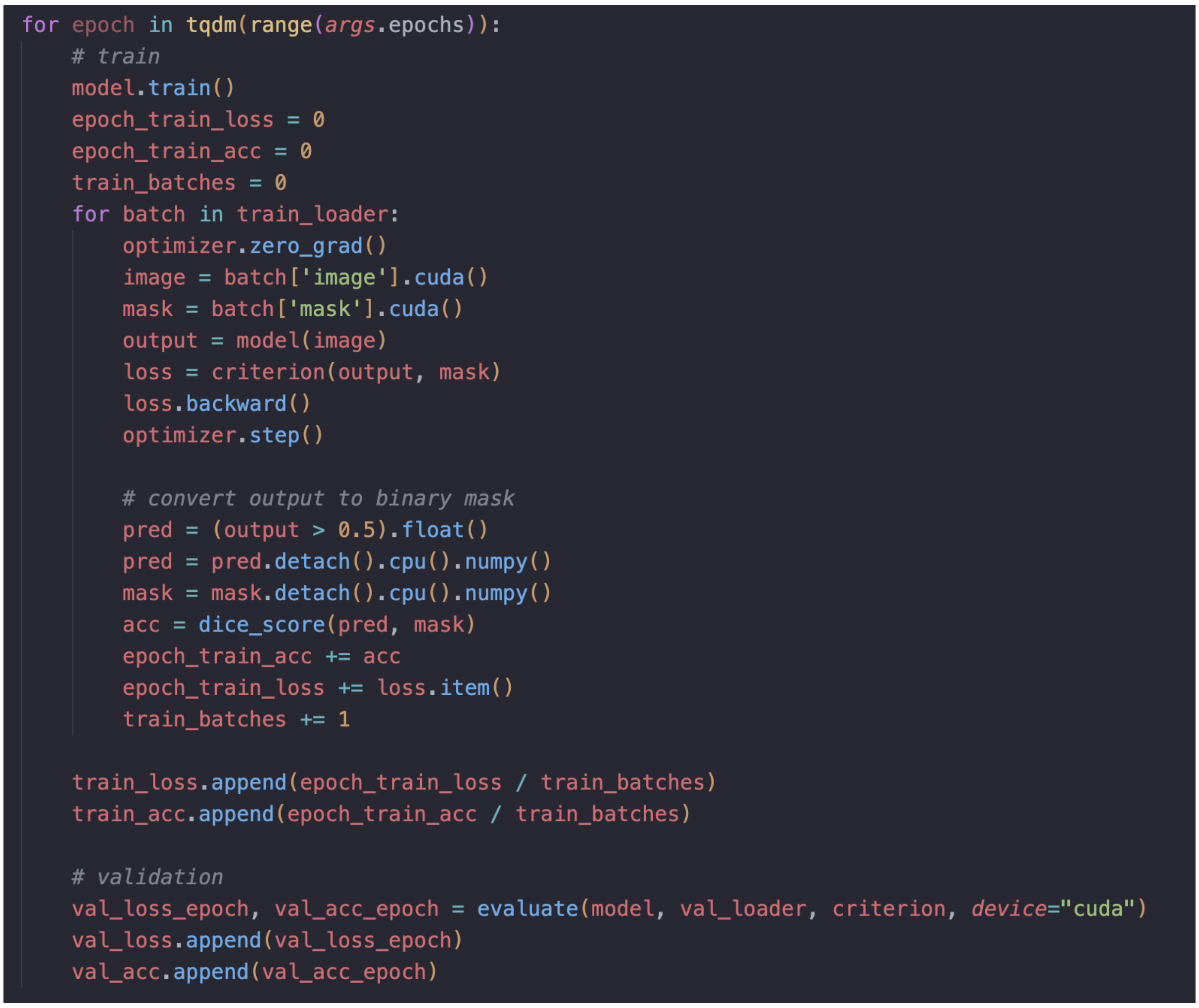


```python
# dataset
train_dataset = load_dataset(data_path=args.data_path, mode="train")
train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
val_dataset = load_dataset(data_path=args.data_path, mode="valid")
val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False)

# model
if args.model_type == "unet":
    model = UNet()
elif args.model_type == "resnet34_unet":
    model = ResNet34_UNet()
else:
    raise ValueError("Model not supported")
model = model.cuda()

# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)

# loss function
criterion = torch.nn.BCELoss()
```

For each epoch of the training loop, the following lines are executed to train the model:

- `model.train()`: switches the model to training mode, enabling gradient calculation and back propagation.
- `optimizer.zero_grad()`: resets the gradients of the model parameters.
- `loss = criterion(outputs, labels)`: computes the loss between the model's predictions outputs and the actual labels labels.
- `loss.backward()`: performs backpropagation, calculates the gradients, and stores the calculated gradients in the neural network.
- `optimizer.step()`: updates the model's weights using the calculated gradients and the given learning rate.
- `pred = (output > 0.5).float()` to `train_batches += 1`: convert the prediction to a binary mask and calculate the dice score.
- `val_loss_epoch, val_acc_epoch = evaluate(model, val_loader, criterion, device="cuda”)`: call the validation function defined as the following.

Finally, the accuracy and loss of the training and the validation parts are stored.

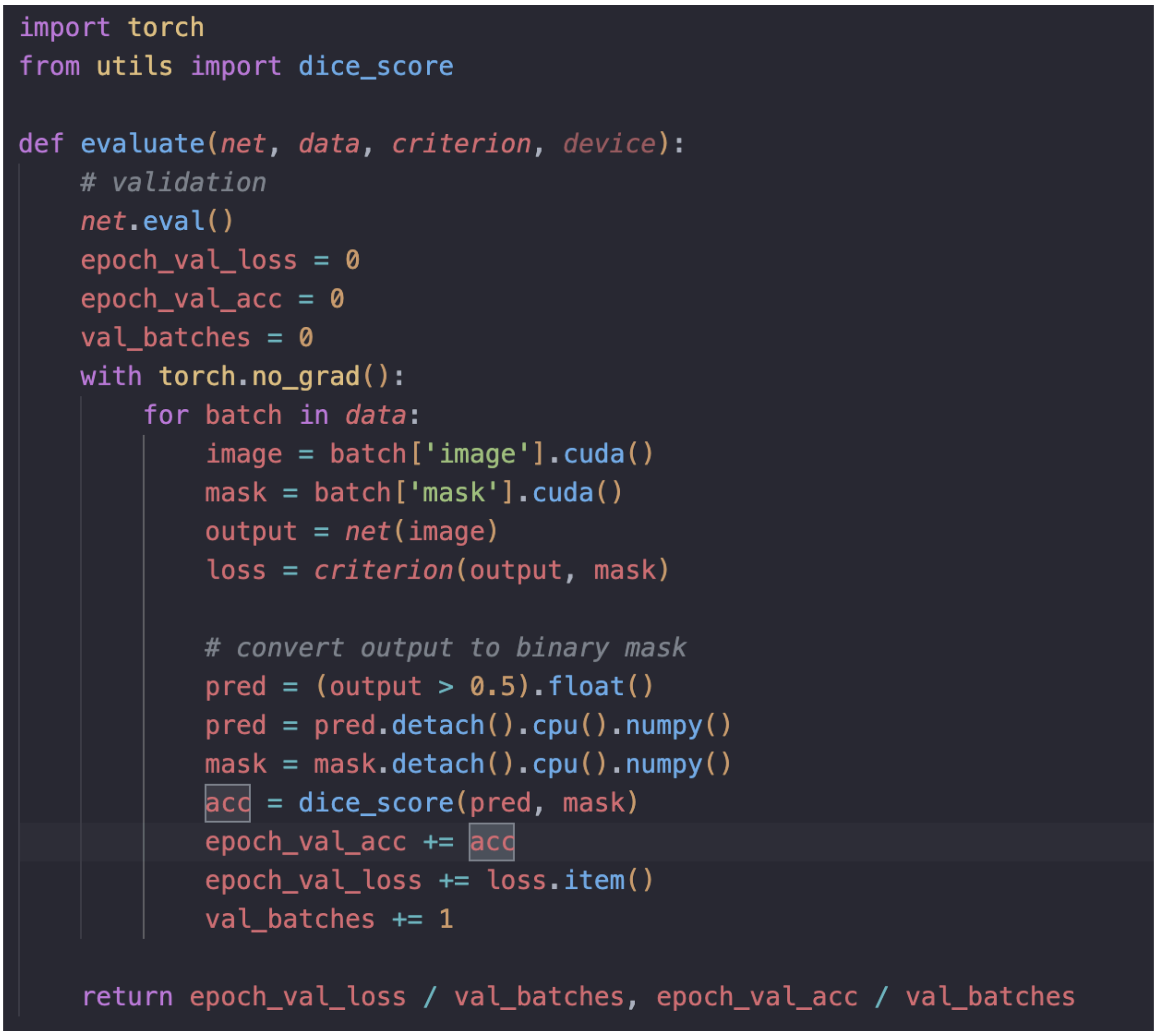


```
for epoch in tqdm(range(args.epochs)):
    # train
    model.train()
    epoch_train_loss = 0
    epoch_train_acc = 0
    train_batches = 0
    for batch in train_loader:
        optimizer.zero_grad()
        image = batch['image'].cuda()
        mask = batch['mask'].cuda()
        output = model(image)
        loss = criterion(output, mask)
        loss.backward()
        optimizer.step()

        # convert output to binary mask
        pred = (output > 0.5).float()
        pred = pred.detach().cpu().numpy()
        mask = mask.detach().cpu().numpy()
        acc = dice_score(pred, mask)
        epoch_train_acc += acc
        epoch_train_loss += loss.item()
        train_batches += 1

    train_loss.append(epoch_train_loss / train_batches)
    train_acc.append(epoch_train_acc / train_batches)

    # validation
    val_loss_epoch, val_acc_epoch = evaluate(model, val_loader, criterion, device="cuda")
    val_loss.append(val_loss_epoch)
    val_acc.append(val_acc_epoch)
```

Evaluating Code

- `model.eval()`: switches the model to evaluation mode, disabling certain layers like dropout.
- `with torch.no_grad()`: disables gradient calculation, reducing memory usage and speeding up computations during inference.

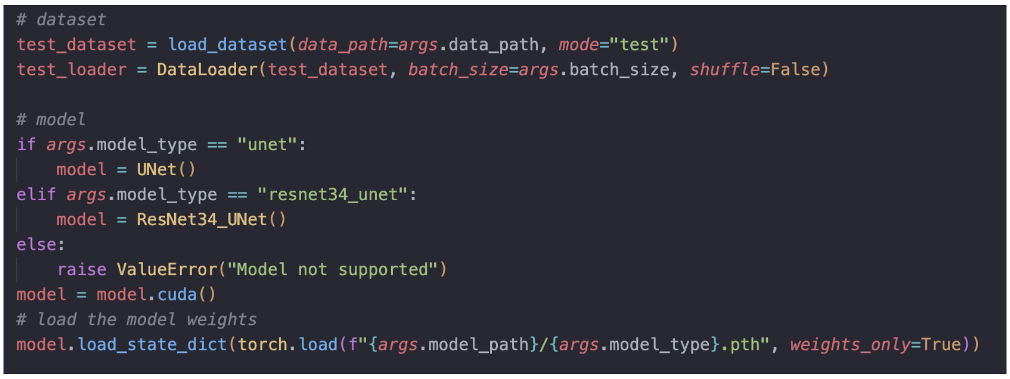
For each batch of the data, the model makes a prediction on the validation dataset, the output is converted into binary mask, and the dice score is calculated as the training session. Finally, the loss and accuracy of the validation part is returned.



```python
import torch
from utils import dice_score

def evaluate(net, data, criterion, device):
    # validation
    net.eval()
    epoch_val_loss = 0
    epoch_val_acc = 0
    val_batches = 0
    with torch.no_grad():
        for batch in data:
            image = batch['image'].cuda()
            mask = batch['mask'].cuda()
            output = net(image)
            loss = criterion(output, mask)

            # convert output to binary mask
            pred = (output > 0.5).float()
            pred = pred.detach().cpu().numpy()
            mask = mask.detach().cpu().numpy()
            acc = dice_score(pred, mask)
            epoch_val_acc += acc
            epoch_val_loss += loss.item()
            val_batches += 1

    return epoch_val_loss / val_batches, epoch_val_acc / val_batches
```

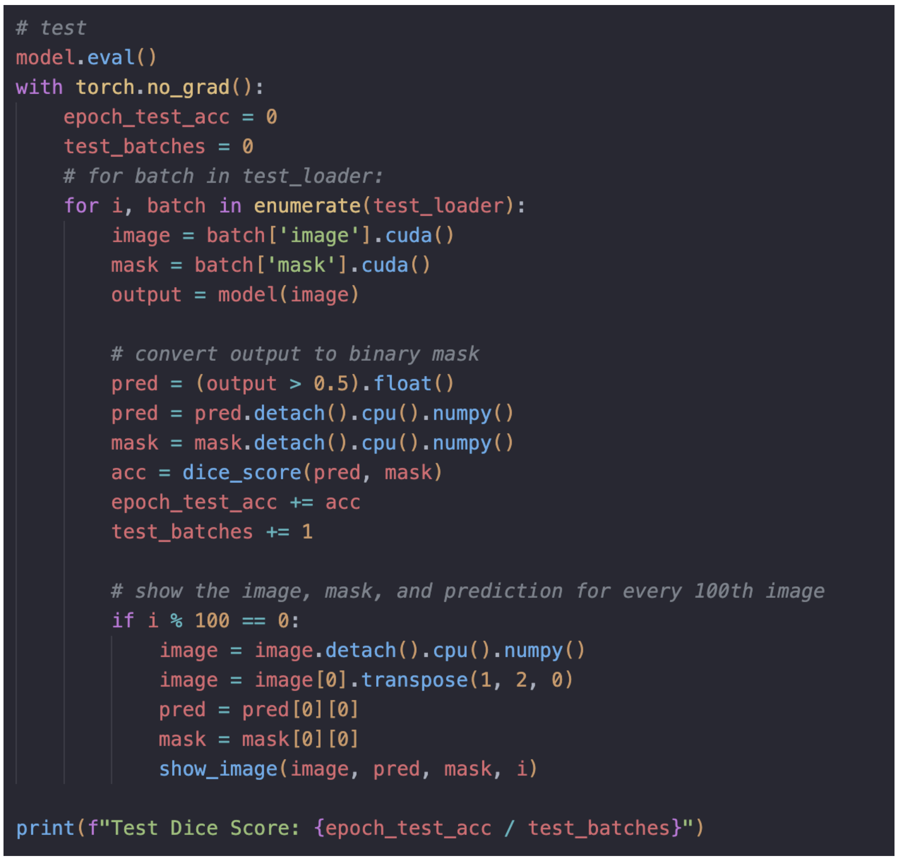
Inferencing Code

Initialize the dataset and load the model weight accordingly.



```python
# dataset
test_dataset = load_dataset(data_path=args.data_path, mode="test")
test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

# model
if args.model_type == "unet":
    model = UNet()
elif args.model_type == "resnet34_unet":
    model = ResNet34_UNet()
else:
    raise ValueError("Model not supported")
model = model.cuda()
# load the model weights
model.load_state_dict(torch.load(f"{args.model_path}/{args.model_type}.pth", weights_only=True))
```

- `model.eval()`: switches the model to evaluation mode, disabling certain layers like dropout.
- `with torch.no_grad()`: disables gradient calculation, reducing memory usage and speeding up computations during inference.
- `pred = (output > 0.5).float()` to `test_batches += 1`: convert the prediction to a binary mask and calculate the dice score.

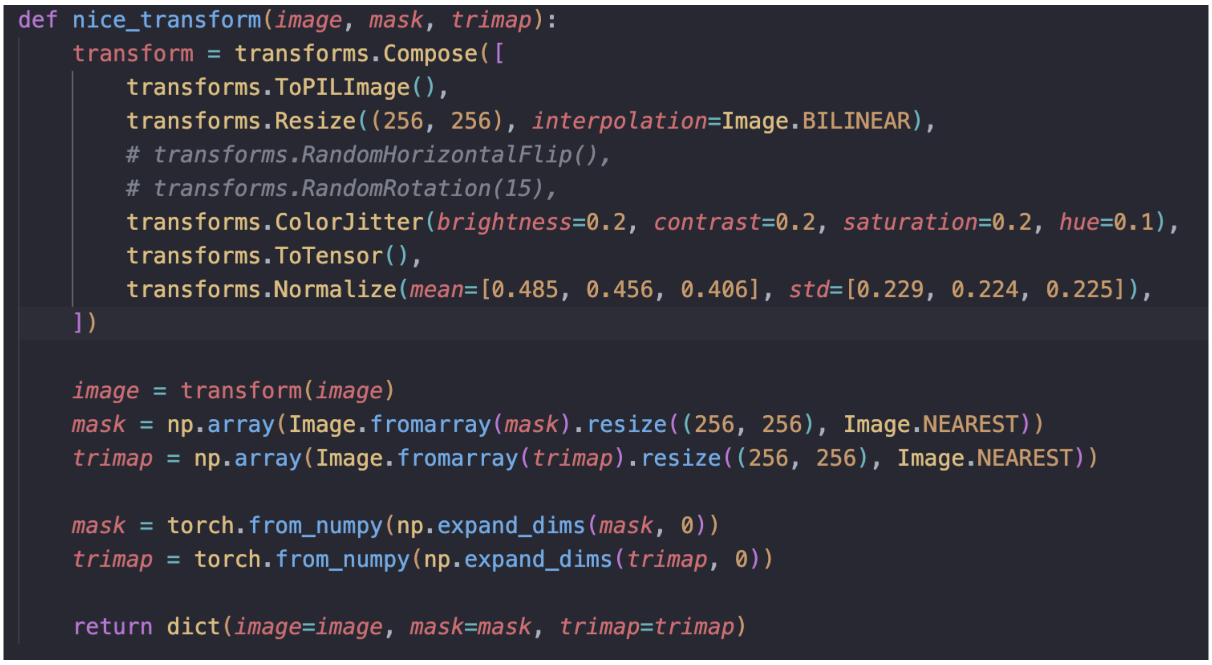


```python
# test
model.eval()
with torch.no_grad():
    epoch_test_acc = 0
    test_batches = 0
    # for batch in test_loader:
    for i, batch in enumerate(test_loader):
        image = batch['image'].cuda()
        mask = batch['mask'].cuda()
        output = model(image)

        # convert output to binary mask
        pred = (output > 0.5).float()
        pred = pred.detach().cpu().numpy()
        mask = mask.detach().cpu().numpy()
        acc = dice_score(pred, mask)
        epoch_test_acc += acc
        test_batches += 1

        # show the image, mask, and prediction for every 100th image
        if i % 100 == 0:
            image = image.detach().cpu().numpy()
            image = image[0].transpose(1, 2, 0)
            pred = pred[0][0]
            mask = mask[0][0]
            show_image(image, pred, mask, i)

print(f"Test Dice Score: {epoch_test_acc / test_batches}")
```

Data Preprocessing

In the `oxford_pet.py` file, the dataset preprocessing involves a series of transformations applied to each image.
The `transforms.Compose` function is used to create a pipeline that

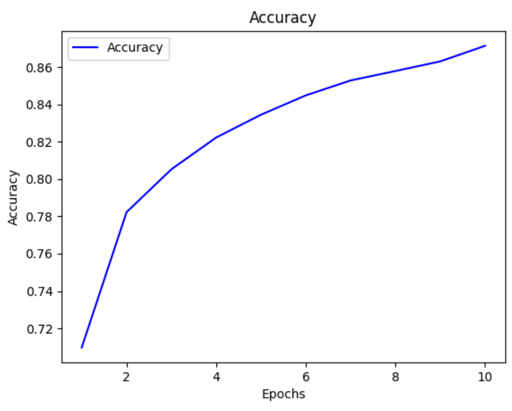
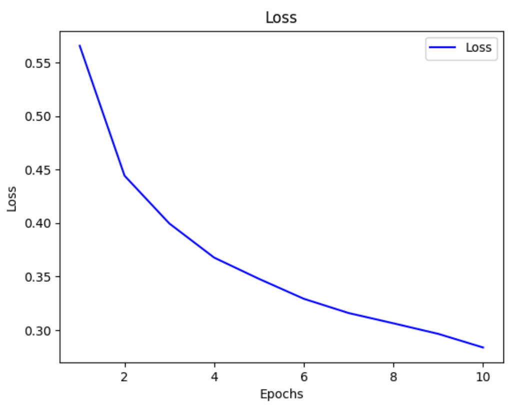
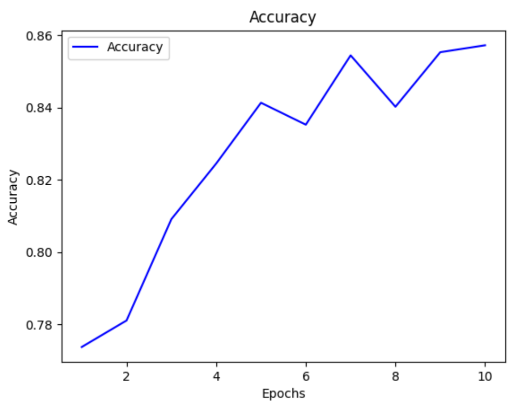
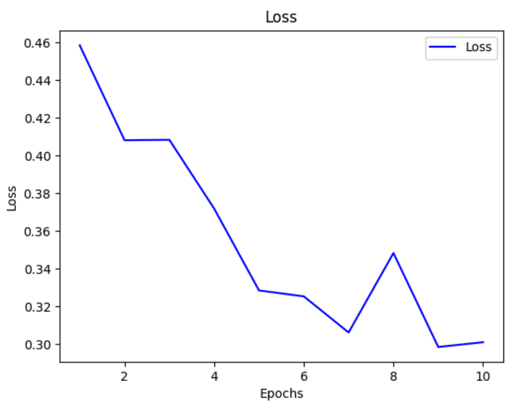
1. converts the image to a PIL format
2. resizes it to 256x256 pixels using bilinear interpolation
3. applies color jittering to randomly adjust the brightness, contrast, saturation, and hue, which help the model to gain more generalizability
4. converts to tensor
5. normalizes using predefined mean and standard deviation values for more stable data input



```python
def nice_transform(image, mask, trimap):
    transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize((256, 256), interpolation=Image.BILINEAR),
        # transforms.RandomHorizontalFlip(),
        # transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    image = transform(image)
    mask = np.array(Image.fromarray(mask).resize((256, 256), Image.NEAREST))
    trimap = np.array(Image.fromarray(trimap).resize((256, 256), Image.NEAREST))

    mask = torch.from_numpy(np.expand_dims(mask, 0))
    trimap = torch.from_numpy(np.expand_dims(trimap, 0))

    return dict(image=image, mask=mask, trimap=trimap)
```

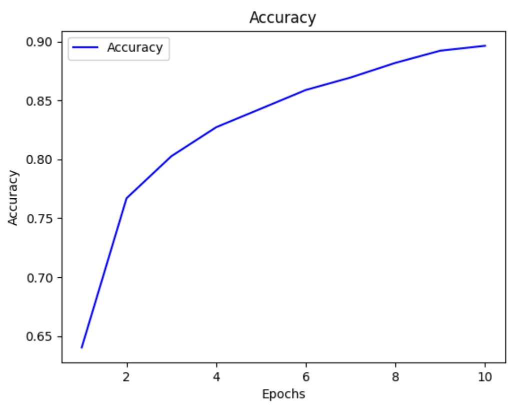
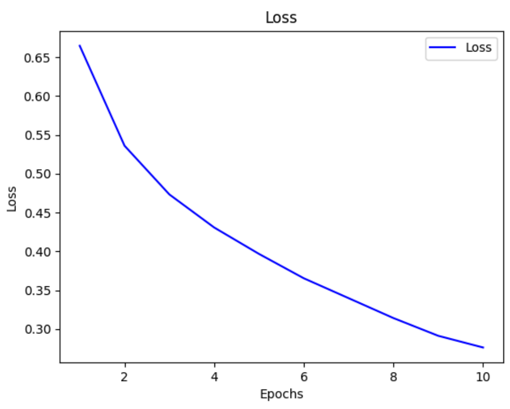
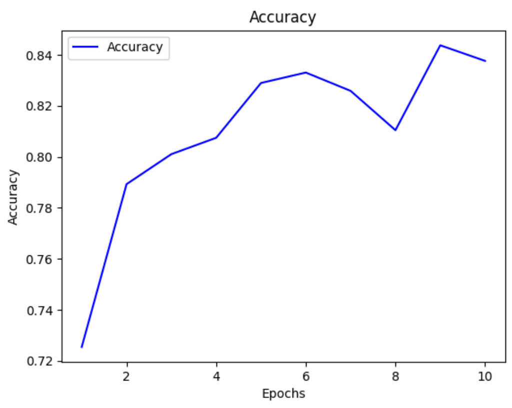
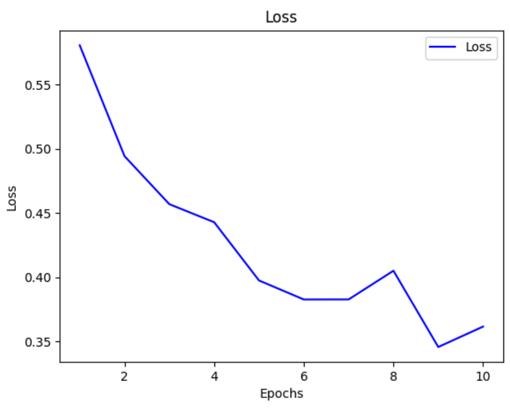
Analyze on the experiment results

UNet

Training Accuracy	Training Loss
	
Validation Accuracy	**Validation Loss**
	

Consistency: Both training and validation metrics show improvement, which is a good sign of the model learning.
Fluctuations: The fluctuations in validation accuracy and loss suggest potential overfitting or that the validation set might contain some challenging examples for the model.

ResNet34_UNet

Training Accuracy	Training Loss
	
Validation Accuracy	**Validation Loss**
	

Consistency: Both training and validation metrics show improvement, which is a positive sign of the model learning effectively.
Fluctuations: The fluctuations in validation accuracy and loss suggest potential overfitting or that the validation set might contain some challenging examples.

Comparison between UNet and ResNet34_UNet

The ResNet34_UNet shows higher final training accuracy (0.90 vs. 0.86) and lower final training loss (0.25 vs. 0.30) compared to the UNet model, indicating better performance on the training set.

Execution command

The command and parameters for the training process:

- navigate into `src/`
- execute `python3 train.py -t <model_type>`
  `<model_type>` should be `unet` or `resnet34_unet`.

The command and parameters for the inference process:

- navigate into `src/`
- execute `python3 inference.py -t <model_type>`
  `<model_type>` should be `unet` or `resnet34_unet`.

Discussion

What architecture may bring better results?

ResNet34_UNet has more parameters in the encoding steps, and because ResNet addresses the issue of gradient vanishing by adding shortcut connections between each residual block, the ResNet34_UNet with more parameters in encoding performs better than UNet.

What Are The Potential Research Topics In This Task?

The main applications of semantic segmentation:

Autonomous Driving

Semantic segmentation is used to understand and interpret the surroundings of a vehicle by identifying different objects such as roads, vehicles, pedestrians, and traffic signs.

Medical Imaging

It assists in the precise segmentation of anatomical structures and abnormalities in medical images, such as MRI or CT scans.

Agricultural Monitoring

Semantic segmentation helps in analyzing satellite or drone imagery to identify and classify different crop types, monitor crop health, and detect weeds or diseases.