Lab 3 Binary Semantic Segmentation

109550121 温柏萱

Reference Websites: unet-pytorch, Pytorch-UNet, UNet with ResNet34 encoder (Pytorch), 如何用基于resnet的Unet进行图像分割 基于Pytorch-0.5版本, Cook your First U-Net in PyTorch

Overview

This lab aims to implement two structures to perform binary semantic segmentation.

UNet is a symmetric structure aim to solve binary semantic segmentation problems, spotting the resemblance of the U-strucutre from its name, it consists of an encoder and a decoder. The encoder is a typical convolutional neural network with 10 convolutional layers, with a max pooling layer to downsample the feature map every two convolutional layers. Then we expand the feature map with upsampling every two convolution layers, with the help of skip connections passed into the layers to assist precise localization. A key advantage of UNet is its efficiency of learning from a small amount of data, with the minimal need of preprocessing, UNet is able to learn and generalize well on limited datasets.

In a UNet architecture, the encoder is responsible for capturing the context of the input image. By replacing this part of U-Net with ResNet34, more complicated feature extraction and abstract features are enabled, leaveraging its ability to segment and identify more complex patterns.

Implementation Details

Training

Helping Functions:

```
def dice_loss(pred, target, epsilon=1e-6):
    pred = pred.contiguous()
    target = target.contiguous()

    intersection = (pred * target).sum(dim=2).sum(dim=2)

    loss = (1 - ((2. * intersection + epsilon) /
(pred.sum(dim=2).sum(dim=2) + target.sum(dim=2).sum(dim=2) + epsilon)))

    return loss.mean()
```

Dice loss is a metric commonly used to evaluate the model performance on image segmentation tasks. For the model output pred and ground truth target, we first calculate the intersection of each pixel, this can be done by a simple multiplication since the output is binary. The dice loss can be formulated as

$$\mathcal{L}_{\text{dice}} = 1 - \frac{2 \times \text{pred} \times \text{target} + \epsilon}{\text{pred} + \text{target} + \epsilon} \tag{1}$$

```
def calculate_accuracy(pred, target):
    pred = torch.sigmoid(pred)  # Apply sigmoid to get [0,1] range
    preds_binary = (pred > 0.5).float()  # Threshold predictions
    correct = (preds_binary == target).float()  # Correct predictions
    accuracy = correct.sum() / (target.size(0) * target.size(2) *
target.size(3))
    return accuracy
```

The `calculate_accuracy()` function takes the prediction of the model, which is in the form of probability, and transform it into binary. Then compare it with the ground truth `target` and sum up the number of matched pixels. Finally, return the proportion of matched pixels. The result of `calculate_accuracy(pred, target)` is identical to calling `torchmetrics.functional.dice_score(pred, target)`.

`train.py`

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
import numpy as np
import argparse
import os
from torchvision import models
from torch.utils.tensorboard import SummaryWriter
import sys
sys.path.insert(0, '../')
from src_handin.models.resnet34_unet import resnet34_unet
from src_handin.models.unet import UNet
# from src.models.gpt_unet import UNet
# from sample_unet import UNet
from src.oxford_pet import SimpleOxfordPetDataset, load_dataset
from src.utils import dice_loss, calculate_accuracy
# from src.utils import MixedLoss, DiceLoss

# Assume UNet and other required imports are already defined above
```

```
def train_one_epoch(model, dataloader, optimizer, device):
    model.train()
    running_loss = 0.0
    running_accuracy = 0.0
    for batch_idx, batch in enumerate(dataloader):
        images = batch['image'].to(device, dtype=torch.float32)
        masks = batch['mask'].to(device, dtype=torch.float32)


        optimizer.zero_grad()

        outputs = model(images)
        loss = dice_loss(F.sigmoid(outputs), masks)
        accuracy = calculate_accuracy(outputs, masks)

        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        running_accuracy += accuracy.item() * images.size(0)
        if batch_idx % 10 == 0:
            print(f"Train : [{batch_idx *
len(batch['image'])}/{len(dataloader.dataset)}"
      f" ({100. * batch_idx / len(dataloader):.0f}%)]\tLoss:
{loss.item():.6f}")

    epoch_loss = running_loss / len(dataloader.dataset)
    epoch_accuracy = running_accuracy / len(dataloader.dataset)
    return epoch_loss, epoch_accuracy

def validate(model, dataloader, device):
    model.eval()
    running_loss = 0.0
    running_accuracy = 0.0
    with torch.no_grad():
        for batch in dataloader:
            images = batch['image'].to(device, dtype=torch.float32)
            masks = batch['mask'].to(device, dtype=torch.float32)
```

```
            outputs = model(images)
            loss = dice_loss(outputs, masks)
            accuracy = calculate_accuracy(outputs, masks)

            running_loss += loss.item() * images.size(0)
            running_accuracy += accuracy.item() * images.size(0)

    epoch_loss = running_loss / len(dataloader.dataset)
    epoch_accuracy = running_accuracy / len(dataloader.dataset)
    return epoch_loss, epoch_accuracy


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--use_model', help='unet or resnet34_unet')
    parser.add_argument('--log_path', help="Tensorboard log path name", type=str, default=None)
    parser.add_argument('--save_as', help="The name of the saved model", type=str, default=None)
    parser.add_argument('--epoch', help="Number of epochs", type=int, default=None)
    args = parser.parse_args()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if args.log_path is not None:
        writer = SummaryWriter(args.log_path)
    else:
        writer = SummaryWriter('./UNet-1')
    # Parameters
    data_path = "../dataset"
    batch_size = 4
    num_epochs = 50
    if args.epoch is not None:
        num_epochs = args.epoch
    learning_rate = 0.001

    # Model
    if args.use_model == 'unet':
        model = UNet(n_channels=3, n_classes=1).to(device)
```

```
        print('Using UNet')
    else:
        model = resnet34_unet(num_classes=1).to(device)
        print('Using ResNet34 + UNet')

    save_as_model = "unet.pth"
    if args.save_as is not None:
        save_as_model = args.save_as + ".pth"

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

    # Load Data
    train_loader = load_dataset(data_path, mode='train',
batch_size=batch_size)
    valid_loader = load_dataset(data_path, mode='valid',
batch_size=batch_size)

    # Training Loop
    best_valid_acc = 0
    for epoch in range(num_epochs):
        train_loss, train_accuracy = train_one_epoch(model, train_loader,
optimizer, device)
        valid_loss, valid_accuracy = validate(model, valid_loader, device)

        print(f"Epoch {epoch+1}/{num_epochs}, Train Loss:
{train_loss:.4f}, Train Acc: {train_accuracy:.4f}, Valid Loss:
{valid_loss:.4f}, Valid Acc: {valid_accuracy:.4f}")
        writer.add_scalar('Train Accuracy', train_accuracy, epoch)
        writer.add_scalar('Valid Accuracy', valid_accuracy, epoch)
        if valid_accuracy > best_valid_acc:
            best_valid_acc = valid_accuracy
            torch.save(model, save_as_model)
            print(f"Best Valid Acc:{valid_accuracy}, saving to
{save_as_model}")
```

`train.py` takes arguments to determine the model to run on and the logging path for tensorboard and output model.

For every epoch, we train on the train dataset with `dice_loss`, and propagate backwards to modify the weigts and bias of every layer, then validate the model using the validation set, finally printing the train and valid accuracy. When saving the model, only save when its valid accuracy is the highest among all previous epochs to prevent overfitting.

Evaluation

```
def evaluate(model, dataloader, device):
    model.eval()
    running_loss = 0.0
    running_accuracy = 0.0
    running_dice = 0.0
    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            images = batch['image'].to(device, dtype=torch.float32)
            masks = batch['mask'].to(device, dtype=torch.float32)

            outputs = model(images)
            loss = dice_loss(outputs, masks)
            accuracy = calculate_accuracy(outputs, masks)
            dice = dice_score(outputs, masks)

            running_loss += loss.item() * images.size(0)
            running_accuracy += accuracy.item() * images.size(0)
            # print(dice.shape)
            running_dice += dice.item() * images.size(0)
            if batch_idx % 10 == 0:
                print(f"Eval : [{batch_idx *
len(batch['image'])}/{len(dataloader.dataset)}"
      f" ({100. * batch_idx / len(dataloader):.0f}%)]\tLoss:
{loss.item():.6f}")

    epoch_loss = running_loss / len(dataloader.dataset)
    epoch_accuracy = running_accuracy / len(dataloader.dataset)
    epoch_dice = running_accuracy / len(dataloader.dataset)
    return epoch_loss, epoch_accuracy, epoch_dice
```

The evaluation reads in model, dataloader and device, then uses the model to generate predictions of the image, calculate the loss and accuracy of the image prediction with the functions defined previously.

Inference

```
if __name__=='__main__':
    parser = argparse.ArgumentParser()
    # parser.add_argument('--use_model', help='unet or resnet34_unet')
    parser.add_argument('--pretrained', type=str, help='Path to load 
pretrained model')
    parser.add_argument('--model', help='unet or res')
    args = parser.parse_args()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if args.model == 'unet':
        model = UNet(n_channels=3, n_classes=1)
    else:
        model = resnet34_unet(num_classes=1)
    model.load_state_dict(torch.load(args.pretrained))
    # model = torch.load(args.pretrained).to(device)
    model = model.to(device)
    print("Loaded model:", (args.pretrained))
    print(model)
    data_path = "../dataset"
    batch_size = 4
    test_loader = load_dataset(data_path, mode='test', 
batch_size=batch_size)

    loss, acc, dice = evaluate(model, test_loader, device)
    print(f"Loss: {loss:.4f},  Acc: {acc:.4f}, Dice Score: {dice:.4f}")
```

To inference the model, the file takes in an argument `--pretrained` to load the pretrained model file from the path given in the argument, then load the test dataset and use `evaluate()` to inference the model on the test dataset.

Model

UNet

```
UNet(
  (init_layer): conv_layer(
    (conv_layers): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (down1): conv_layer_down(
    (conv_down_layers): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): conv_layer(
        (conv_layers): Sequential(
          (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
```

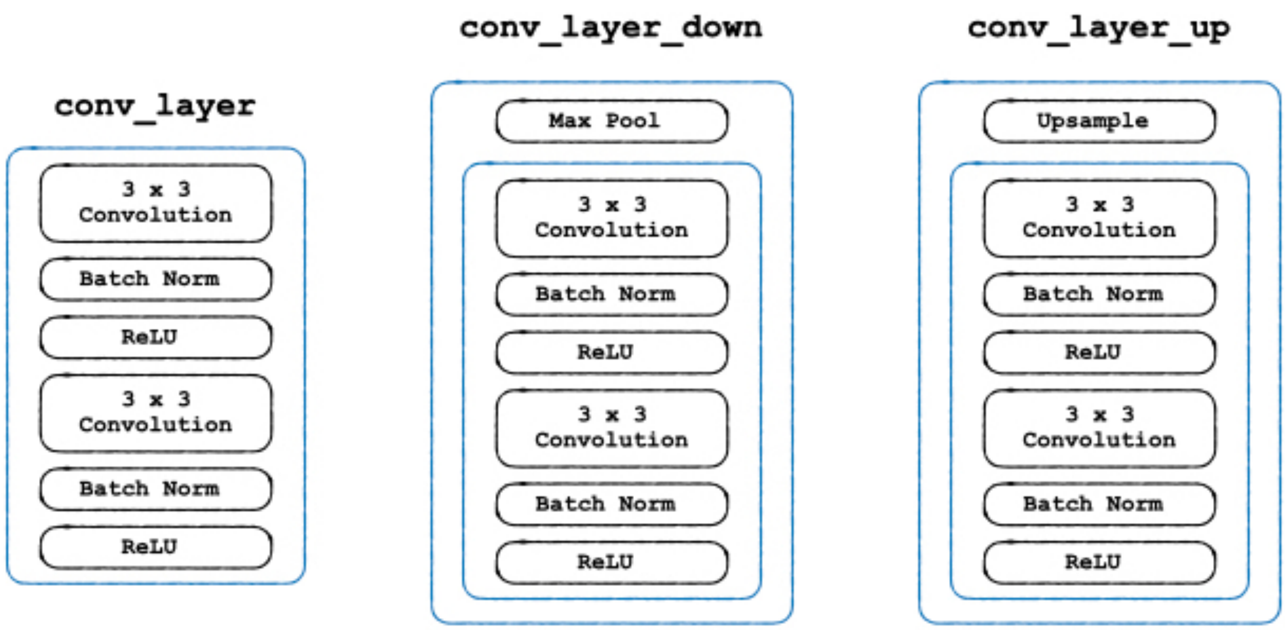
```
  (down2): conv_layer_down(
    (conv_down_layers): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): conv_layer(
        (conv_layers): Sequential(
          (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (down3): conv_layer_down(
    (conv_down_layers): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (1): conv_layer(
        (conv_layers): Sequential(
          (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
```

```
  )
  (down4): conv_layer_down(
    (conv_down_layers): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): conv_layer(
        (conv_layers): Sequential(
          (0): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
  )
  (up1): conv_layer_up(
    (up): ConvTranspose2d(1024, 512, kernel_size=(2, 2), stride=(2, 2))
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up2): conv_layer_up(
```

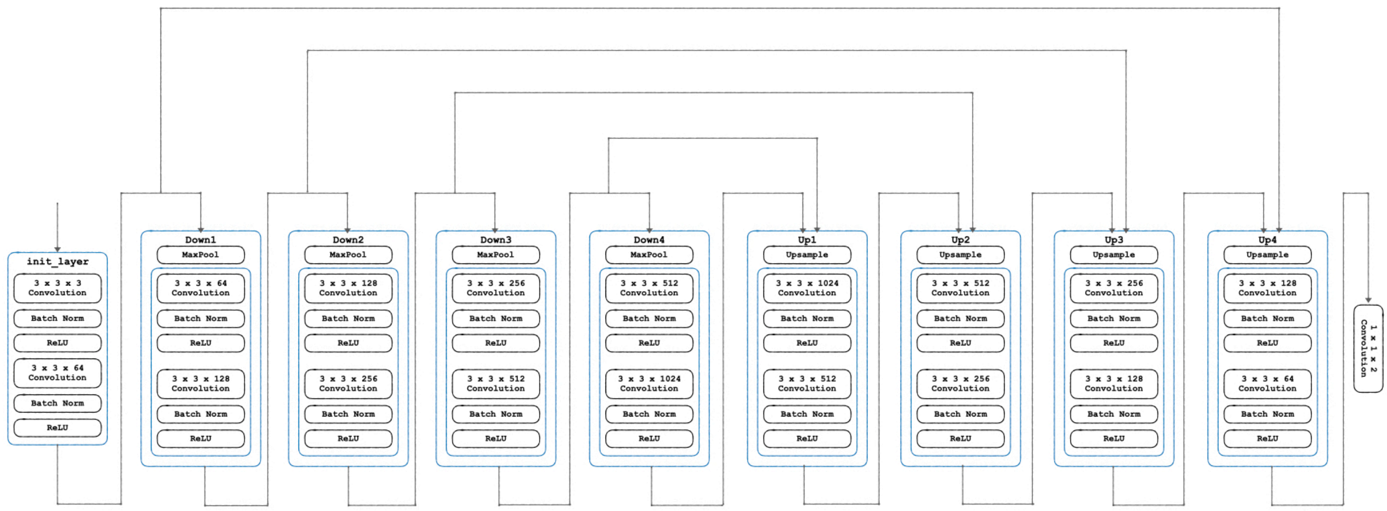
```
    (up): ConvTranspose2d(512, 256, kernel_size=(2, 2), stride=(2, 2))
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up3): conv_layer_up(
    (up): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up4): conv_layer_up(
    (up): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
```

```
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (output_layer): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
)
```

To simplify the implementation, I constructed the `conv_layer` class to wrap up two convolution layers, batch normalization and `ReLU` activation. The `conv_layer_down` combines max-pooling with `conv_layer` as the basic building block for UNet in the encoding phase. The `conv_layer_up` combines upsampling with `conv_layer` as the decoder building block. The three structure can be visualized in the following figure.



With the basic building blocks, we can construct UNet following the structure below.




```
import torch
import torch.nn as nn
import torch.nn.functional as F

class conv_layer(nn.Module):

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.conv_layers = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1,
bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.conv_layers(x)


class conv_layer_down(nn.Module):

    def __init__(self, in_channels, out_channels):
```

```
        super().__init__()
        self.conv_down_layers = nn.Sequential(
            nn.MaxPool2d(2),
            conv_layer(in_channels, out_channels)
        )

    def forward(self, x):
        return self.conv_down_layers(x)


class conv_layer_up(nn.Module):

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
            self.conv = conv_layer(in_channels, out_channels, in_channels
// 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
kernel_size=2, stride=2)
            self.conv = conv_layer(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)


class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
```

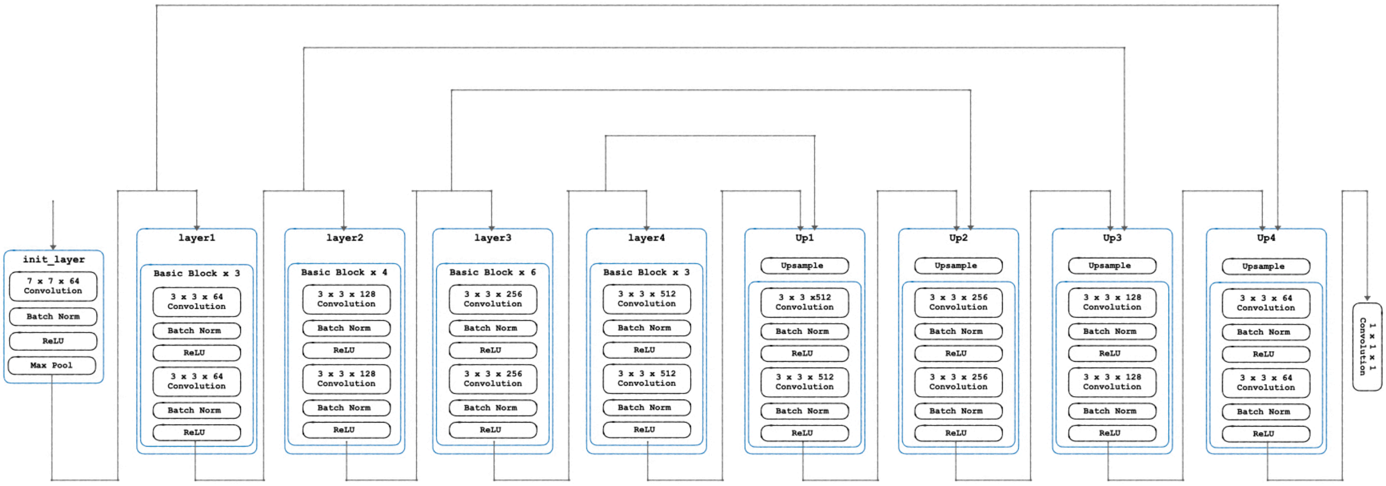
```python
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.init_layer = (conv_layer(n_channels, 64))
        self.down1 = (conv_layer_down(64, 128))
        self.down2 = (conv_layer_down(128, 256))
        self.down3 = (conv_layer_down(256, 512))
        factor = 2 if bilinear else 1
        self.down4 = (conv_layer_down(512, 1024 // factor))
        self.up1 = (conv_layer_up(1024, 512 // factor, bilinear))
        self.up2 = (conv_layer_up(512, 256 // factor, bilinear))
        self.up3 = (conv_layer_up(256, 128 // factor, bilinear))
        self.up4 = (conv_layer_up(128, 64, bilinear))
        self.output_layer = nn.Conv2d(64, n_classes, kernel_size=1)

    def forward(self, x):
        x1 = self.init_layer(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.output_layer(x)
        return logits
    def use_checkpointing(self):
        self.init_layer = torch.utils.checkpoint(self.init_layer)
        self.down1 = torch.utils.checkpoint(self.down1)
        self.down2 = torch.utils.checkpoint(self.down2)
        self.down3 = torch.utils.checkpoint(self.down3)
        self.down4 = torch.utils.checkpoint(self.down4)
        self.up1 = torch.utils.checkpoint(self.up1)
        self.up2 = torch.utils.checkpoint(self.up2)
        self.up3 = torch.utils.checkpoint(self.up3)
        self.up4 = torch.utils.checkpoint(self.up4)
        self.outc = torch.utils.checkpoint(self.outc)
```

ResNet34 + UNet

The model replaces the encoder with ResNet34, with the initial layers passed to the `Up1` layer in UNet, three layers of stacked Basic Blocks passed into `Up2, Up3, Up4`. The structure can be visualized as the following figure.




```
resnet34_unet(
  (resnet34_encoder): RN34(
    (init_layers): Sequential(
      (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    )
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
```

```
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
```

```
      (3): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (4): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (5): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
```

```
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu2): ReLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1, bias=True)
  )
```

```
  (up1): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up2): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(640, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up3): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
```

```
        (0): Conv2d(320, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up4): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(192, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up5): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
```

```
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (up6): conv_layer_up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): conv_layer(
      (conv_layers): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
  (output_layer): Conv2d(32, 1, kernel_size=(1, 1), stride=(1, 1))
)
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        nn.init.xavier_uniform_(self.conv1.weight)  # Xavier
initialization
```

```
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
        nn.init.xavier_uniform_(self.conv2.weight)  # Xavier
initialization
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu2 = nn.ReLU(inplace=True)
        self.stride = stride
        self.in_channels = in_channels
        self.out_channels = out_channels

    def forward(self, x):
        residual = x

        output = self.conv1(x)
        output = self.bn1(output)
        output = self.relu1(output)

        output = self.conv2(output)
        output = self.bn2(output)

        # residual shortcut
        if self.stride != 1 or self.in_channels != self.out_channels:
            conv_layer = nn.Conv2d(self.in_channels, self.out_channels,
kernel_size=1, stride=self.stride, bias=False).to(residual.device)
            residual = conv_layer(residual)

            bn_layer =
nn.BatchNorm2d(self.out_channels).to(residual.device)
            residual = bn_layer(residual)

        output += residual
        output = self.relu2(output)

        return output


class RN34(nn.Module):
```

```
    def __init__(self, num_classes=1000):
        super(RN34, self).__init__()

        self.init_layers = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7,
stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        # ResNet layers
        self.layer1 = self.layer_init(in_channels=64, out_channels=64,
blocks=3, stride=1)
        self.layer2 = self.layer_init(in_channels=64, out_channels=128,
blocks=4, stride=2)
        self.layer3 = self.layer_init(in_channels=128, out_channels=256,
blocks=6, stride=2)
        self.layer4 = self.layer_init(in_channels=256, out_channels=512,
blocks=3, stride=2)

        # Final layers
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def layer_init(self, in_channels, out_channels, blocks, stride):
        layers = []
        layers.append(BasicBlock(in_channels, out_channels, stride))
        for _ in range(1, blocks):
            layers.append(BasicBlock(out_channels, out_channels, stride))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
```

```
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

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

    def forward(self, x):
        return self.conv_layers(x)


class conv_layer_up(nn.Module):
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
            self.conv = conv_layer(in_channels, out_channels)
        else:
            self.up = nn.ConvTranspose2d(in_channels , in_channels // 2,
kernel_size=2, stride=2)
            self.conv = conv_layer(in_channels, out_channels)
```

```python
    def forward(self, x1, x2=None):
        if x2 is None:
            x = self.up(x1)
            x = self.conv(x)
            return x
        x2 = self.up(x2)
        diffY = x1.size()[2] - x2.size()[2]
        diffX = x1.size()[3] - x2.size()[3]
        x2 = F.pad(x2, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x1, x2], dim=1)
        return self.conv(x)

class resnet34_unet(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()

        self.resnet34_encoder = RN34(num_classes)

        self.up1 = conv_layer_up(768, 512)
        self.up2 = conv_layer_up(640, 256)
        self.up3 = conv_layer_up(320, 128)
        self.up4 = conv_layer_up(192, 64)
        self.up5 = conv_layer_up(64, 32)
        self.up6 = conv_layer_up(32, 32)
        self.output_layer = nn.Conv2d(32, 1, kernel_size=1)

    def forward(self, x):
        x1 = self.resnet34_encoder.init_layers(x)
        x2 = self.resnet34_encoder.layer1(x1)
        x3 = self.resnet34_encoder.layer2(x2)
        x4 = self.resnet34_encoder.layer3(x3)
        x5 = self.resnet34_encoder.layer4(x4)

        x6 = self.up1(x4, x5)
        x7 = self.up2(x3, x6)
        x8 = self.up3(x2, x7)
        x9 = self.up4(x1, x8)
```

```
        x10 = self.up5(x9)
        x11 = self.up6(x10)

        return self.output_layer(x11)
```

Data Preprocess

For data preprocess, I applied random horizontal flip and vertical flip to augment the dataset and enhance the generalize ability of the model. The comparision of data preprocess will be presented in the discussion section.

Below is the data preprocess done for training dataset.

```
 transform = transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.RandomVerticalFlip(p=0.5),
            transforms.ToTensor()  # Convert images and masks to PyTorch
tensors
        ])
```

`transforms.Resize((256, 256))` resizes the image to 256×256, and apply random horizontal and vertical flip with probability of 0.5, finally, `transforms.ToTensor()` will convert the `PIL`image to tensor and project the values from $[0, 255]$ to $[0, 1]$.

Since validation is to verify against the original data, we only apply resizing and convert the image to tensor.

```
transform = transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ToTensor()  # Convert images and masks to PyTorch
tensors
        ])
```

Also for the modification to the original `oxford_pet.py`, I removed the transformations such as moving axis from $256 \times 256 \times 3$ to $3 \times 256 \times 256$, resizing to 256×256 and projection to $[0, 1]$ done in `numpy` arrays in the sample code, since in the sample code, passing `numpy` arrays into `torchvision.transforms` will cause the following error:

```
Binary_Semantic_Segmentation/src/../src/oxford_pet.py", line 44, in
__getitem__
    sample = self.transform(**sample)
TypeError: __call__() got an unexpected keyword argument 'image'
```

All of the tasks in the following code that was in the providied sample code

```
image = np.array(Image.fromarray(sample["image"]).resize((256, 256),
Image.BILINEAR), dtype=np.float32)
mask = np.array(Image.fromarray(sample["mask"]).resize((256, 256),
Image.NEAREST), dtype=np.float32)
trimap = np.array(Image.fromarray(sample["trimap"]).resize((256, 256),
Image.NEAREST), dtype=np.float32)
sample["image"] = np.moveaxis(image, -1, 0)
sample["mask"] = np.expand_dims(mask, 0)
sample["trimap"] = np.expand_dims(trimap, 0)
```

can be done by

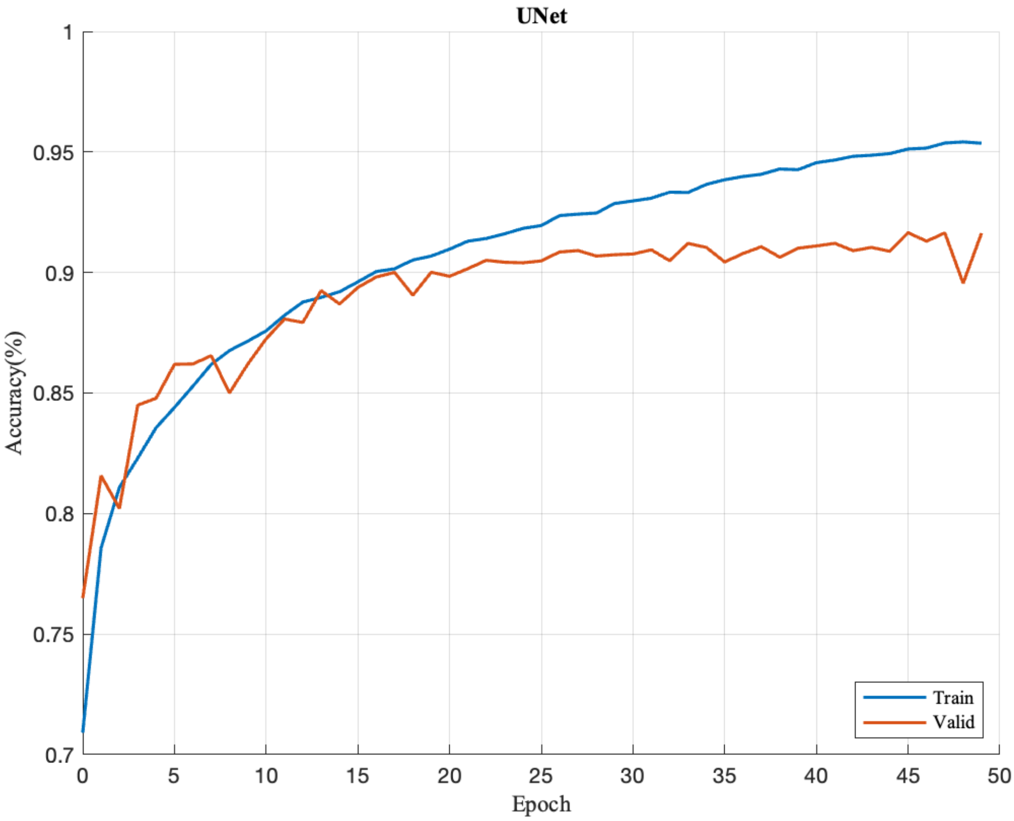
```
transform = transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ToTensor()
        ])
```

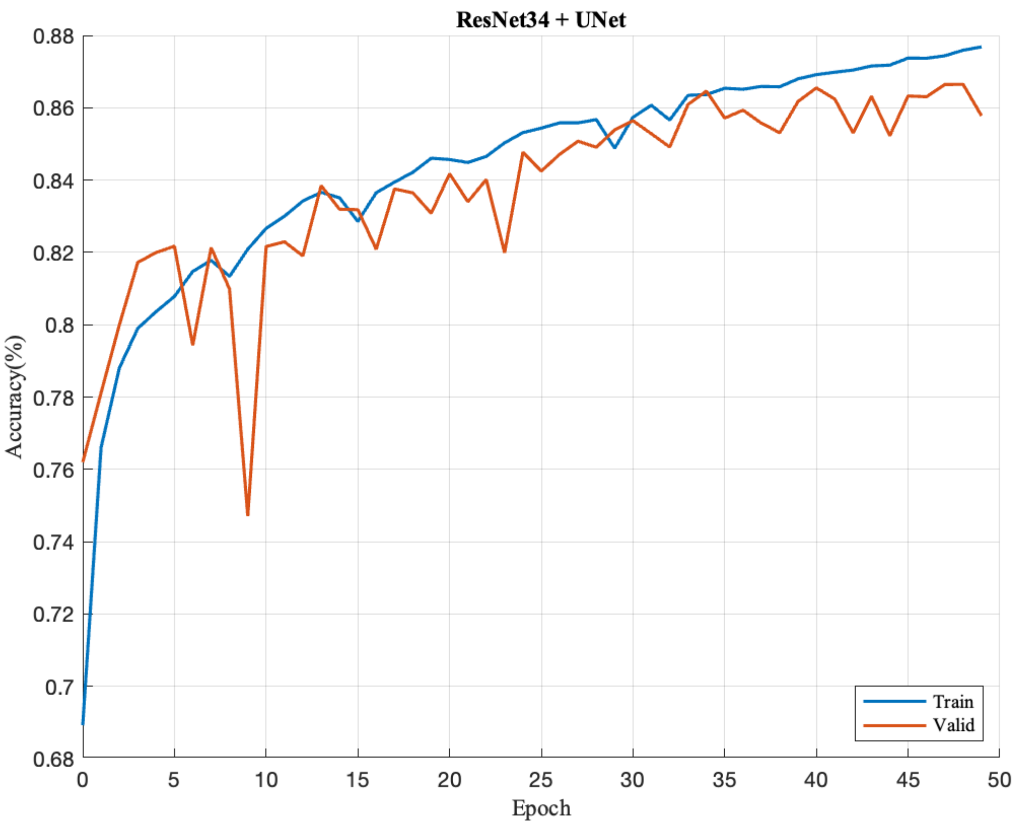
Hence I removed the reduendant code and simplified the implementation.

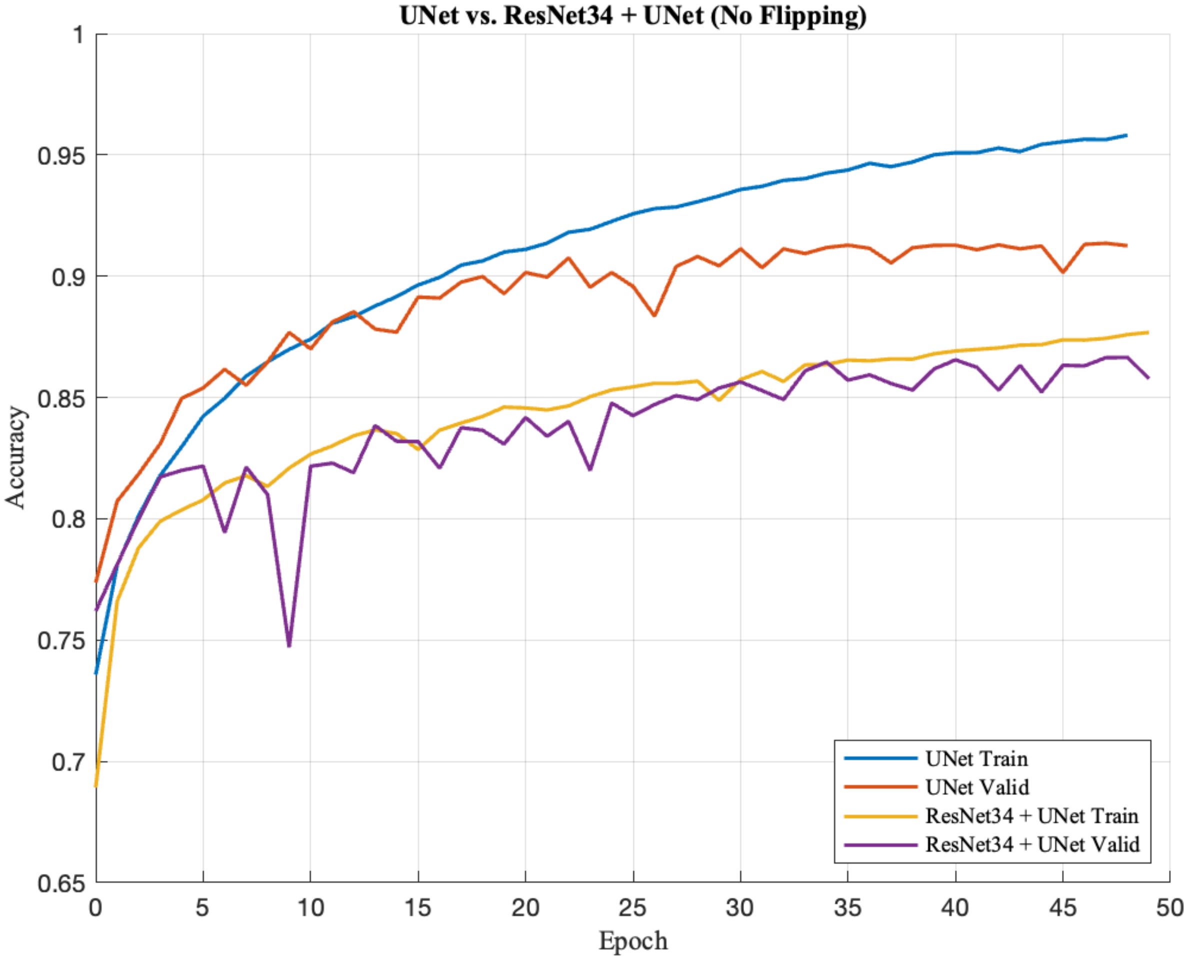
Experiment Results Analysis

	UNet	ResNet34 + UNet
No Flipping (50 epochs)	0.9206	0.8687
Flipping (50 epochs)	0.8112	0.8095
Flipping (100 epochs)	0.8301	0.8133

No Flipping


UNet
Accuracy(%)
Epoch
Train
Valid
1
0.95
0.9
0.85
0.8
0.75
0.7
0
5
10
15
20
25
30
35
40
45
50

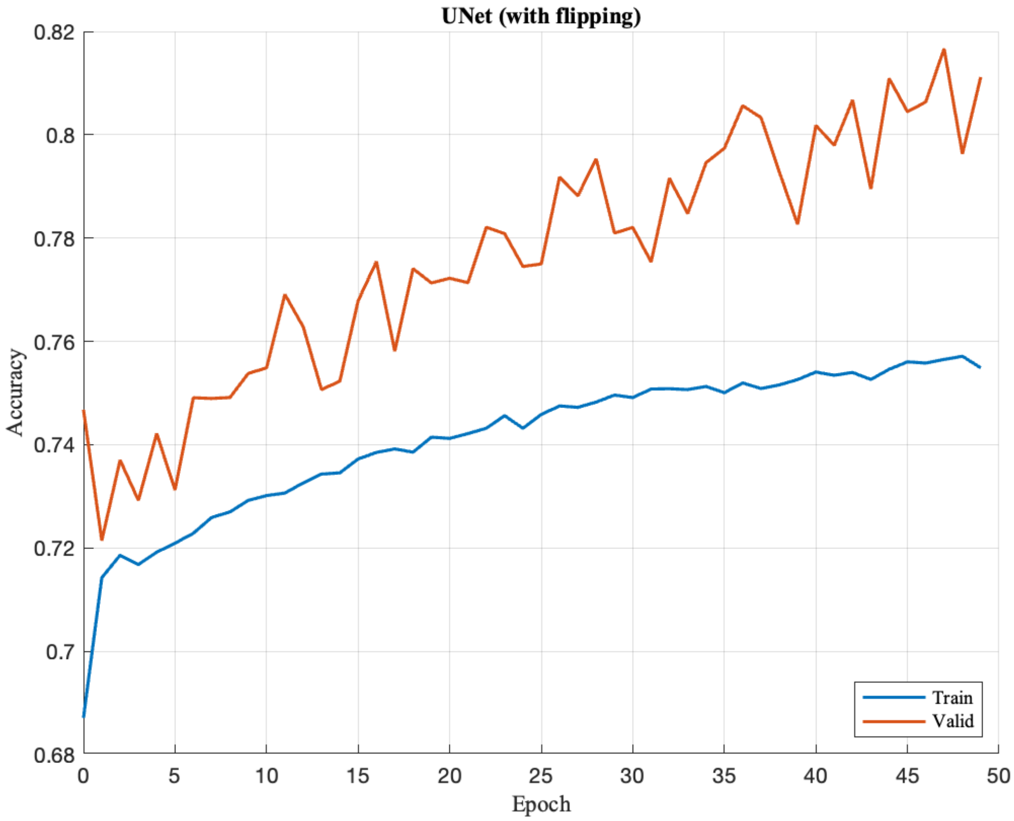

ResNet34 + UNet
Accuracy(%)
Epoch
Train
Valid
0.88
0.86
0.84
0.82
0.8
0.78
0.76
0.74
0.72
0.7
0.68
0
5
10
15
20
25
30
35
40
45
50



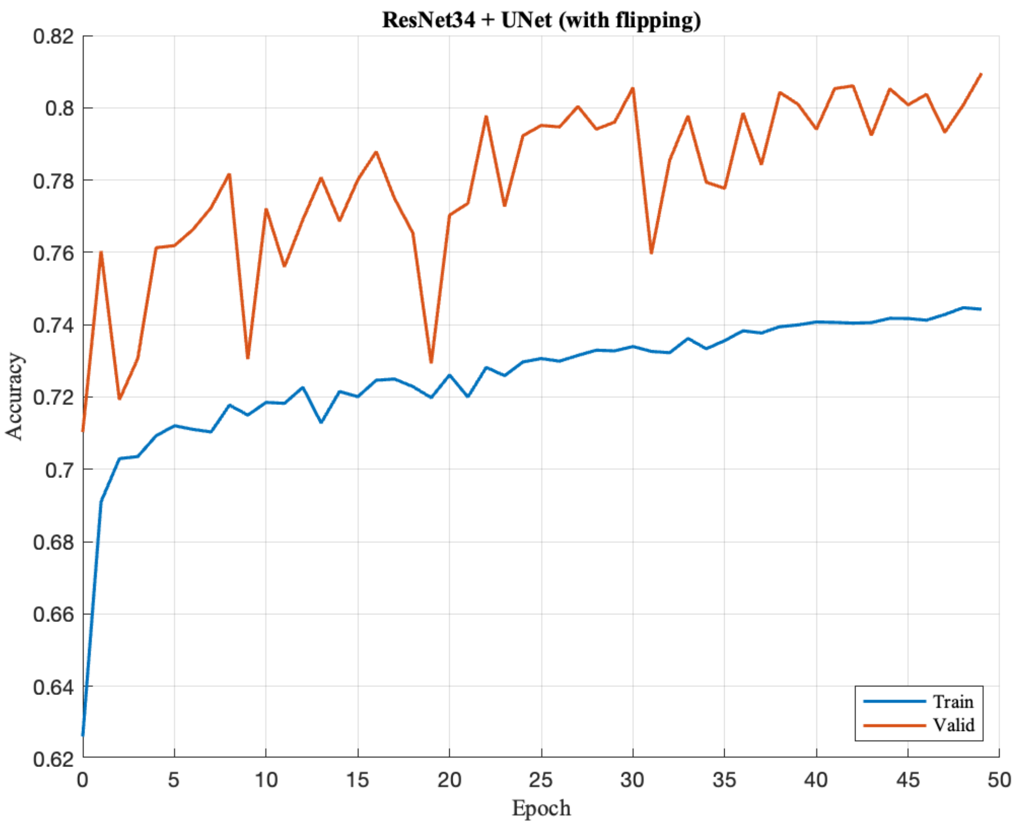
Without random horizontal or vertical flip, the accuracy converged really fast with only 50 epochs, and the training accuracy is higher than validation, which is reasonable since the model is trained on the training dataset.

Flipping

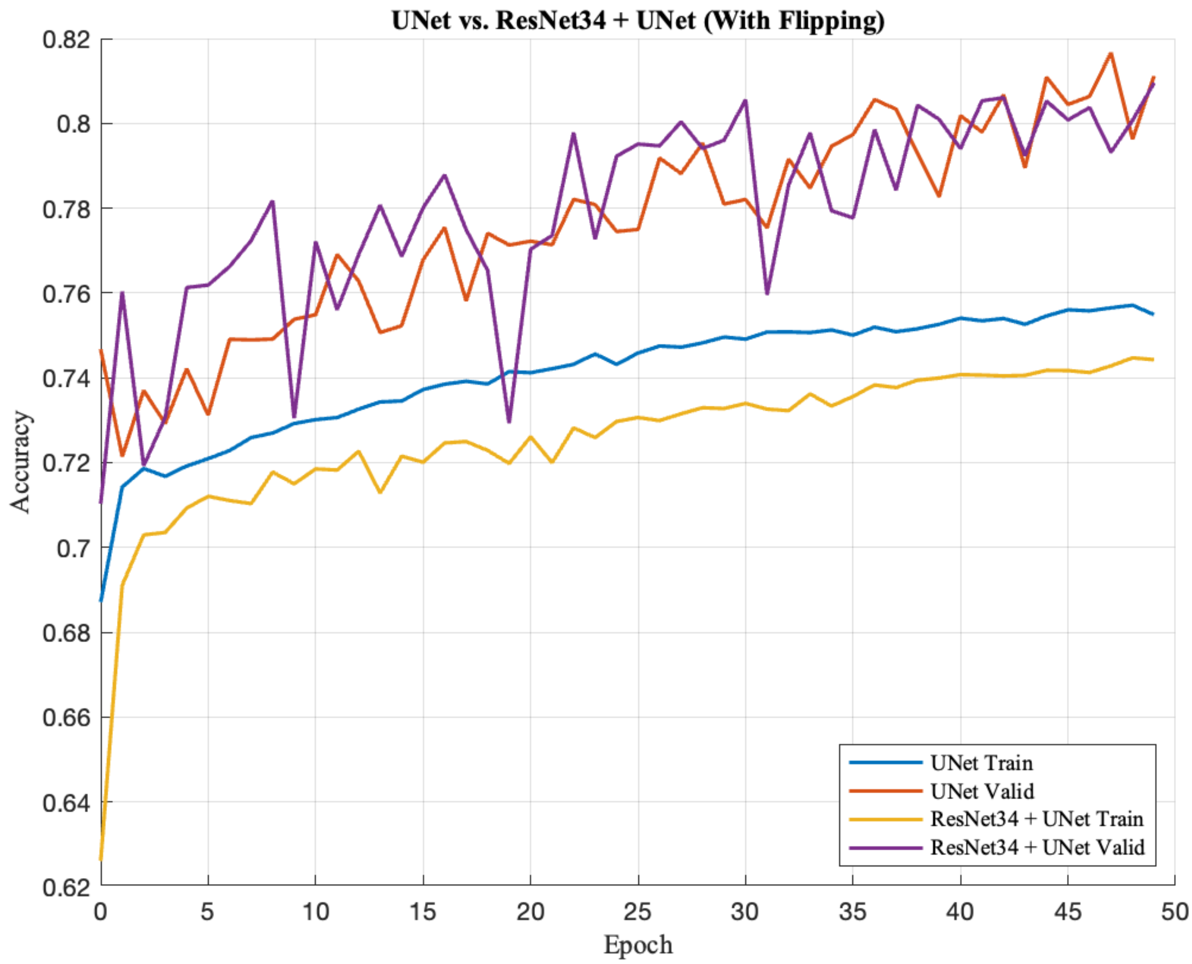
By flipping horizontally and vertically, the training accuracy was decreased and even dropped below the validation accuracy, the phenomenon indicates the dataset do not have many upside down images, therefore, since the validation set was not flipped, the accuracy exceeds training. The following are the figures of training for 50 epochs with horizontal and vertical flipping.



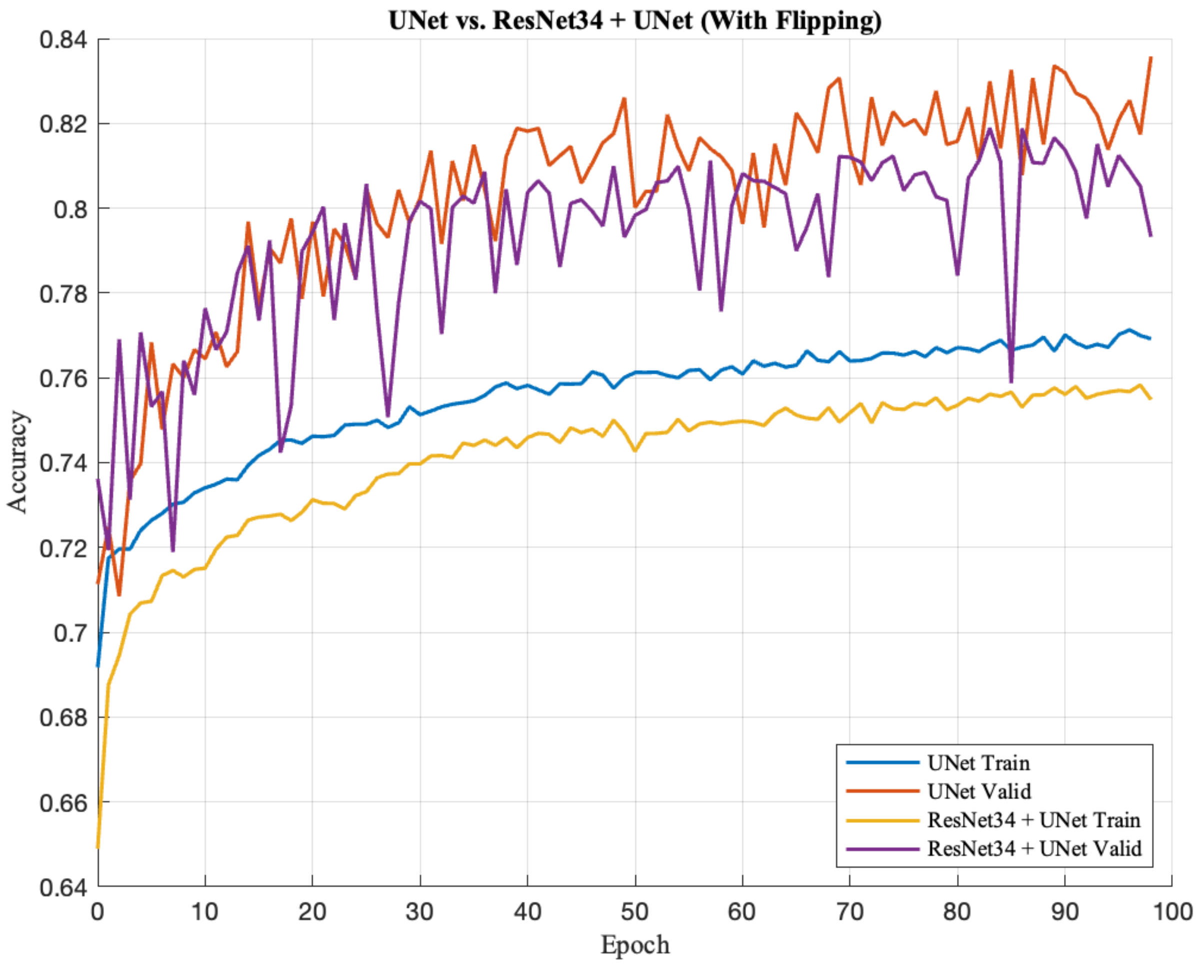
UNet (with flipping)
Accuracy
Epoch
0.82
0.8
0.78
0.76
0.74
0.72
0.7
0.68
0
5
10
15
20
25
30
35
40
45
50
Train
Valid



ResNet34 + UNet (with flipping)
Accuracy
Epoch
0.82
0.8
0.78
0.76
0.74
0.72
0.7
0.68
0.66
0.64
0.62
0
5
10
15
20
25
30
35
40
45
50
Train
Valid



To further witness the possibility of higher accuracy with flipping, I also experimented with running for 100 epochs, but the increase in accuracy was minor.



Another presumption is that cats and dogs have varied poses, and certain positions may be more common or natural in one direction. For instance, a dog might typically curve its tail in a specific direction. Flipping such an image may create an unusual appearance that does not occur frequently in the actual data distribution.

Execution Command

Train

```
# UNet
python train.py --use_model unet --log_path ./Unet-1 --save_as Unet-1
# ResNet34 + UNet
python train.py --use_model resnet34_unet --log_path ./Res-Unet-1 --
save_as Res-Unet-1
# Arguments
  --use_model USE_MODEL unet or resnet34_unet
  --log_path LOG_PATH   Tensorboard log path name
  --save_as SAVE_AS     The name of the saved model
  --epoch EPOCH         Number of epochs
```

Inference

```
python inference.py --pretrained ../saved_models/UNet-2.pth --model unet
  --pretrained PRETRAINED
                        Path to load pretrained model
  --model MODEL         unet or res
```

Discussion

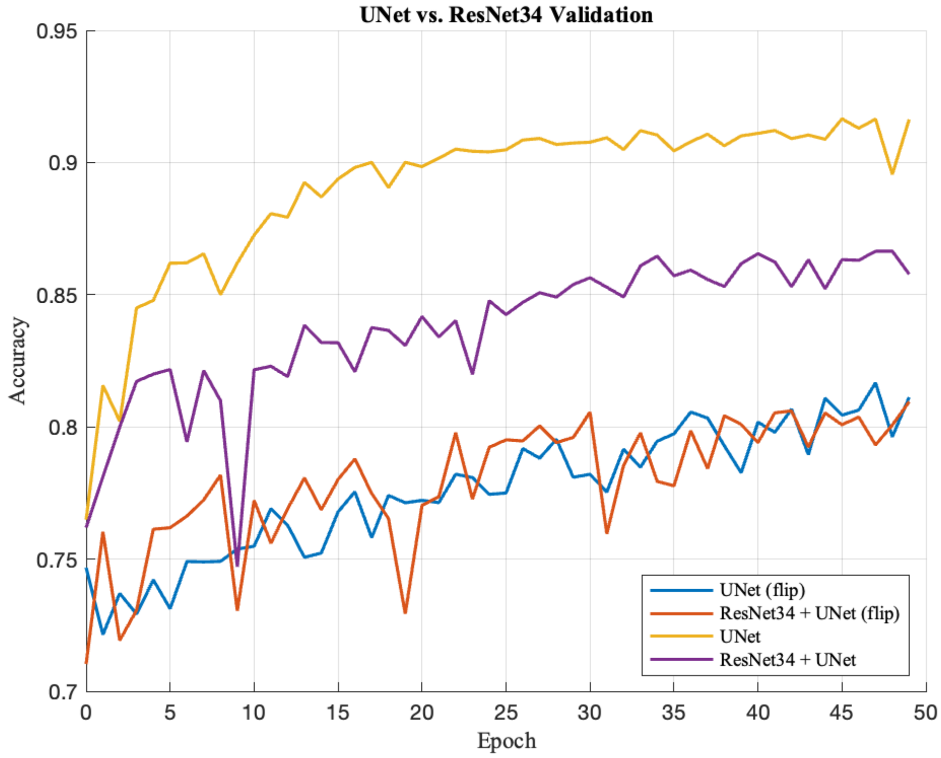
UNet vs. ResNet34 + UNet

UNet performs better on whethear or not we flip the image, while ResNet34 itself performs well in image classification tasks and have outstanding ability to identify complex patterns, the task is binary semantic segmentation, which may not need and elaborate structure like ResNet34 to identify the patters, the traditional UNet is enough and could even out-perform the combination with ResNet34.

Another possible reason is that the residual connections in ResNet34 promote the flow of gradients during training, which can be advantageous in deep networks. However, if the task or the data does not require such deep feature hierarchies, the original U-Net's architecture could offer a more balanced approach to feature extraction and localization.

Flipping the image

As shown in the experiment result analysis section, flipping the image in the training phase hindered the learning and lowered the validation accuracy compared to those without flipping.



Therefore, by presumption, the dataset does not have many upside-down images even in the validation or test set.

Potential Experiments

Different Encoders

Combining the experiences from Lab2, we could replace the ResNet34 encoder with the VGG family encoders to analyze the impact resulted from the structure in terms of accuracy, training time and model loss.

Data Augmentation Strategies

Study the effect of various data augmentation techniques (beyond simple flipping, such as rotations, scaling, color jittering) on the model's ability to generalize and perform on the binary semantic segmentation task. Analyze the trade-offs between augmentation complexity and performance improvement.

Different Loss functions

The performance of semantic segmentatioin can also be evaluated by other loss functions such as focal loss, Jaccard loss and different combinations of the basic loss functions, further analysis can evaluate the influence of different loss functions in terms of accuracy and training time.