DLP Lab3 Report

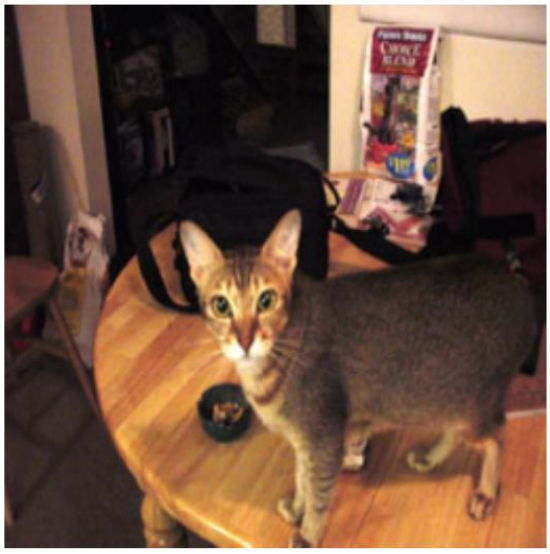
Binary Semantic Segmentation

Name: 丁祐承

ID: 0812212

1. Introduction

The task of the lab is to let us implement the binary semantic segmentation to obtain the mask of the dog and cat dataset. For segmentation, it is basically the pixel-wise classification. We need to implement the famous UNet architecture and its variation ResNet34-UNet architecture.



Image

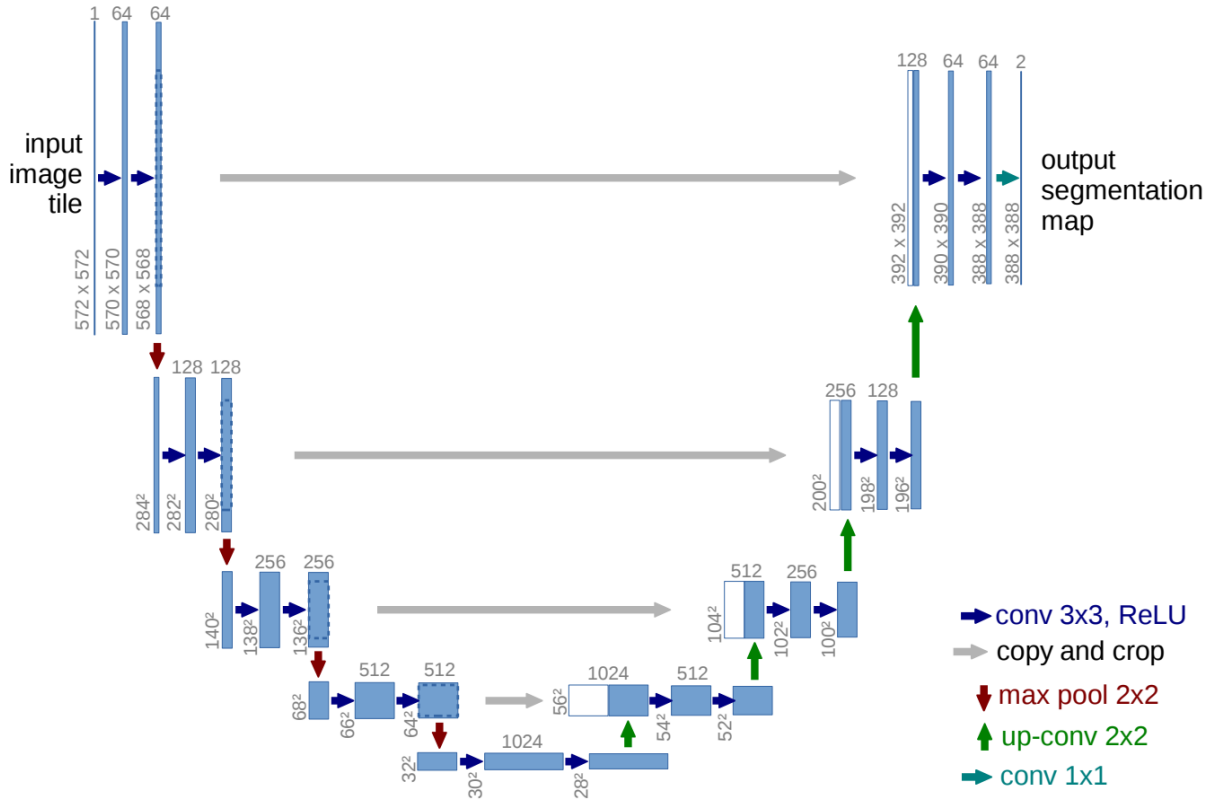
GT mask

2. Implementation Details

A. Model achitecture

UNet:

UNet is a CNN based model and it is originally being used as biomedical image segmentation. It has two main parts, the first part is the encoder and the second part is the decoder. For the encoder part, it downsamples the input data to small size, high dimension latent features. For the decoder part, it upsamples the latent features back to the desired size and dimension. For this lab, we upsamples the features back to its original size to get the mask.



UNet architecture

For encoder, it is constructed with many convolutional blocks. Each convolutional block has two convolutional layers with relu activation function. After each convolutional block, we add a max pooling layer to achieve the downsampling. For decoder, it is also constructed with many convolutional blocks. The difference between encoder and decoder is that the each decoder's block, we change the max pooling layer to up convolutional layer to achieve the upsampling. Also, we will add a connection between encoder and decoder to make sure the

network learn both the feature and original data to keep the background the the foreground of the mask. The following screenshots are my implementation for UNet.

Since our task is to do the binary semantic segmentation, we add a sigmoid function at the end of the network.

```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

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

class Down(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

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

```
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

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

```python
class Up(nn.Module):
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels , in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)


    def forward(self, x1, x2):
        x1 = self.up(x1)
        # print(x1.size())
        # print(x2.size())
        # data (N,C,H,W)
        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)
```

```python
class UNet(nn.Module):
    def __init__(self, config=[64*2**x for x in range(5)], bilinear=True, input_class=3, out_class=1):
        super(UNet, self).__init__()
        self.inc = DoubleConv(input_class, config[0])
        self.down1 = Down(config[0], config[1])
        self.down2 = Down(config[1], config[2])
        self.down3 = Down(config[2], config[3])
        factor = 2 if bilinear else 1
        self.down4 = Down(config[3], config[4] // factor)
        self.up1 = Up(config[4], config[3] // factor, bilinear)
        self.up2 = Up(config[3], config[2] // factor, bilinear)
        self.up3 = Up(config[2], config[1] // factor, bilinear)
        self.up4 = Up(config[1], config[0], bilinear)
        self.outc = OutConv(config[0], out_class)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x1 = self.inc(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.outc(x)
        logits = self.sigmoid(logits)
        return logits
```

ResNet34-UNet:

For ResNet34-UNet, it changes the encoder part of the UNet to ResNet34 to do the downsampling. By using ResNet34 as the backbone, the network benefits from deep and robust feature extraction capabilities, essential for complex segmentation tasks where identifying subtle features is crucial. The residual connections in ResNet34 help in stabilizing the training of deep networks by addressing the vanishing gradient problem, making it feasible to leverage deeper architectures for better performance. The following screenshots are my implementation on ResNet34-UNet.

```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

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

        return out
```

```python
def make_layer(block, in_channels, out_channels, blocks, stride=1):
    downsample = None
    if stride != 1 or in_channels != out_channels * block.expansion:
        downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels * block.expansion, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(out_channels * block.expansion),
        )

    layers = []
    layers.append(block(in_channels, out_channels, stride, downsample))
    in_channels = out_channels * block.expansion
    for _ in range(1, blocks):
        layers.append(block(in_channels, out_channels))

    return nn.Sequential(*layers)
```

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

    def forward(self, x, skip):
        x = self.up(x)
        # Handling spatial size mismatch
        diffY = skip.size()[2] - x.size()[2]
        diffX = skip.size()[3] - x.size()[3]

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

```python
class ResNet34UNet(nn.Module):
    def __init__(self):
        super(ResNet34UNet, self).__init__()
        self.initial = nn.Sequential(nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
                                     nn.BatchNorm2d(64),
                                     nn.ReLU(True),
                                     nn.Conv2d(64, 64, kernel_size=3, padding=1, bias=False),
                                     nn.BatchNorm2d(64),
                                     nn.ReLU(True))

        # Encoder (ResNet34)
        self.layer1 = make_layer(BasicBlock, 64, 64, 3)
        self.layer2 = make_layer(BasicBlock, 64, 128, 4, stride=2)
        self.layer3 = make_layer(BasicBlock, 128, 256, 6, stride=2)
        self.layer4 = make_layer(BasicBlock, 256, 512, 3, stride=2)

        # Decoder
        self.up1 = DecoderBlock(512, 256, 256)
        self.up2 = DecoderBlock(256, 128, 128)
        self.up3 = DecoderBlock(128, 64, 64)
        self.up4 = DecoderBlock(64, 64, 64)

        self.outc = nn.Conv2d(64, 1, kernel_size=1)
        self.bn = nn.BatchNorm2d(1)
        self.sigmoid = nn.Sigmoid()
```

B. Training, evaluation, inferencing code:

```python
if args.model == 'unet':
    model = UNet()
else:
    model = ResNet34UNet()

model = model.cuda()
criterion = nn.BCELoss()
optimizer = RAdam(model.parameters(), lr=args.learning_rate)

train_loss_his, train_score_his, valid_score_his, valid_loss_his = [], [], [], []
best_score = 0
```

```python
print('Training')
for epoch in range(1, args.epochs+1):
    # if epoch == 25:
    #     optimizer = Adam(model.parameters(), lr=args.learning_rate/10)
    model.train()
    train_loss = 0
    train_dice = 0
    total = 0
    for data in tqdm(train_loader, desc=f'Epoch {epoch}'):
        x = data['image'].cuda()
        y = data['mask'].cuda()

        optimizer.zero_grad()
        # Foward
        pred_mask = model(x)
        # Compute BCE loss
        loss = criterion(pred_mask, y)
        # Backward
        loss.backward()
        # Update
        optimizer.step()

        train_loss += loss.item() * x.size(0)
        total += x.size(0)
        train_dice += dice_score(pred_mask, y) * x.size(0)

    # Validation
    val_loss, val_dice = evaluate(model, valid_loader, criterion)

    # Save model for best score
    if (val_dice) > best_score:
        best_score = val_dice
        print(f'Best valid score: {best_score}, model saved')
        torch.save(model, f'../saved_models/{args.model}.pth')

    print(f'Training loss: {train_loss/total:.4f}    Training dice score: {train_dice/total:.4f}    Valid loss: {val_loss:.4f}    Valid dice score: {val_dice:.4f}')
```

For training code, my loss function is Binary Cross Entropy and my optimizer is RAdam. I stored the training loss, score, validation loss and score in a list so that I can plot the learning curve after training. During training, if my validation score is higher than previous best score, I will save the model.

```python
# Validation
def evaluate(net, dataloader, criterion):
    # implement the evaluation function here
    net.eval()
    with torch.no_grad():
        val_loss, val_dice, total = 0, 0, 0
        for data in tqdm(dataloader, desc='Valid'):
            x = data['image'].cuda()
            y = data['mask'].cuda()
            pred_mask = net(x)
            loss = criterion(pred_mask, y)
            val_loss += loss.item() * x.size(0)
            total += x.size(0)
            val_dice += dice_score(pred_mask, y) * x.size(0)

    return val_loss/total, val_dice/total
```

For evaluation code, I set the model to eval() mode for evaluation. I stored the loss and the score for plotting the learning curve.

```
args = get_args()

test_dataset = load_dataset(args.data_path, mode='test', transform={'image':transforms.Compose([transforms.Resize((256,256)),
                                                                                                transforms.ToTensor(),
                                                                                                transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])]),
                                                                    'mask':transforms.Compose([transforms.Resize((256,256)),
                                                                                               transforms.ToTensor()])})
test_loader = DataLoader(test_dataset)
model = torch.load(args.model).cuda()

# Evaluate the test dataset
test_score = test(model, test_loader)
print(f'Experiment results: Dice score {test_score:.4f}')

# Plot random example
plot_example(model, test_dataset, plot=True)
```

```
# Testing
def test(net, dataloader):
    # implement the evaluation function here
    net.eval()
    with torch.no_grad():
        val_dice, total = 0, 0
        for data in tqdm(dataloader, desc='Test'):
            x = data['image'].cuda()
            y = data['mask'].cuda()
            pred_mask = net(x)
            total += x.size(0)
            val_dice += dice_score(pred_mask, y) * x.size(0)

    return val_dice/total
```

For inferencing code, I calculated the dice score and plot the example to visualize our result.

Plotting example

C. Dice Score

In the lab, we use dice score to evaluate our result. Mathematically, the Dice score is computed as:

Dice score = 2 * (number of common pixels) / (predicted img size + groud truth img size)

Dice score quantifies how closely a predicted segmentation mask aligns with the ground truth segmentation mask in image segmentation. Its range spans from 0 (indicating no overlap) to 1 (representing perfect alignment).

```python
def dice_score(pred_mask, gt_mask):
    # implement the Dice score here
    # Data size: (N,1,H,W)
    assert pred_mask.size() == gt_mask.size(), 'Predict mask should be the same size as the ground truth mask.'
    num_batch = pred_mask.size(0)
    # Data size: (N,H*W)
    pred = pred_mask.view(num_batch, -1)
    gt = gt_mask.view(num_batch, -1)
    # Round to 0, 1
    pred = torch.round(pred)

    intersection = torch.sum((pred == gt), dim=1)
    score = torch.mean(2 * intersection / (pred.size(-1) + gt.size(-1))).item()
    return score
```

Since the model output a 0 ~ 1 range value for a (N * 1 * H * W) tensor, so I round my output to make it a 0, 1 tensor. After rounding my output, I calculate the intersection by using " ==" operator and then calculate the final result.

3. Data Preprocessing

For data preprocessing, I modified the given code "oxford_pet.py". Basically I changed the __getitem__ function in SimpleOxfordPetDataset class to make sure the random seed is the same. I used "torchvision.transforms" to do the data preprocessing and the data augmentation. For training data, I divide the transforms into two parts. The first one is to deal with the original data and the second one is to deal with the mask. I resized the data and turn it into tensor and normalize it. I used random vertical, horizontal flip and random rotation for data augmentation. For testing and validation data, I only do the preprocessing procedure which are resize, totenor and normalize.

```
T = {'image':transforms.Compose([transforms.Resize((256,256)),
                                 transforms.RandomVerticalFlip(),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])]),
     'mask':transforms.Compose([transforms.Resize((256,256), interpolation=Image.NEAREST),
                                transforms.RandomVerticalFlip(),
                                transforms.RandomHorizontalFlip(),
                                transforms.ToTensor()])}
```

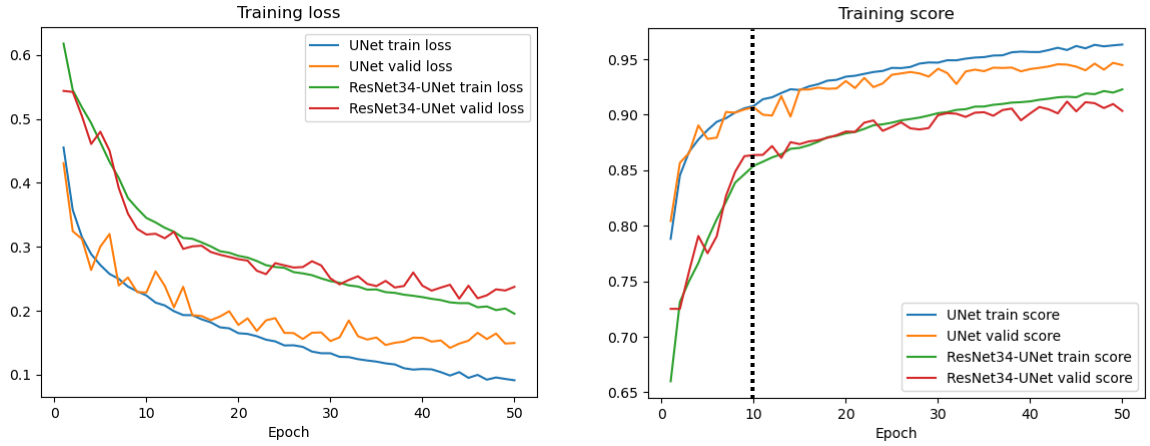
```
class SimpleOxfordPetDataset(OxfordPetDataset):
    def __getitem__(self, *args, **kwargs):

        sample = super().__getitem__(*args, **kwargs)

        # load image and mask
        image = Image.fromarray(sample["image"])
        mask = Image.fromarray(sample["mask"])
        trimap = np.array(Image.fromarray(sample["trimap"]).resize((256, 256), Image.NEAREST))

        # Transform
        seed = np.random.randint(2147483647)
        if self.transform is not None:
            torch.manual_seed(seed)
            sample['image'] = self.transform['image'](image)
            torch.manual_seed(seed)
            sample['mask'] = self.transform['mask'](mask)

        sample['mask'] = (sample['mask'] > 0).float()
        sample = dict(image=sample['image'], mask=sample['mask'])
        return sample
```

By implementing data augmentation, such as random flip, I think it can help increase the model' performance. Also, normalizing the data can help model converge more easily and quickly.

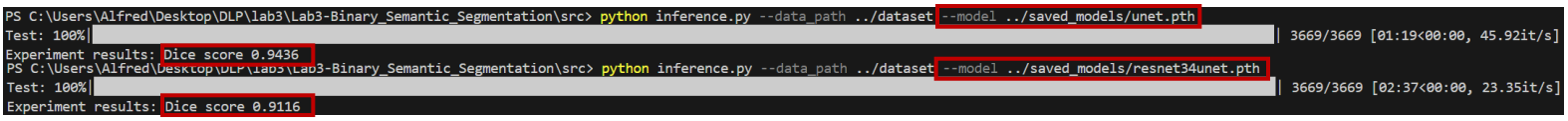
4. Analyze on the experiment results

I trained both the UNet and ResNet34-UNet for 50 with no learning decay. From the training loss history figure we can see that both the networks almost have the same decreasing curve except ResNet34-UNet is about 0.15 higher than UNet. The loss curves keep going down after 50 epochs, so I do not know if it will keep converge after 50 epoch. From the training score history figure we can see that it is quite the same as the loss history. The score curves look quite the same

except ResNet34-UNet is about 0.05 lower than UNet. From the score history, we can know that the model almost only need about 10 epoch to pass the baseline of this lab, which is 0.87 dice score. For UNet, it already hit 0.9 dice score at 10 epoch and ResNet34-UNet hit 0.85 dice score.

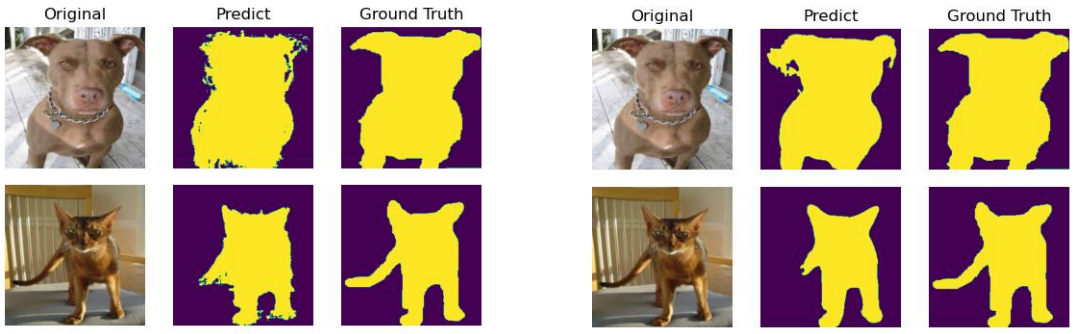


Training history



Result

For testing phase, UNet reached 0.9436 and ResNet-UNet reached 0.9116 as our experiment expected.
The following are few examples of the predict mask and their comparison with the ground truth.



Left: ResNet34-UNet demonstration

Right: UNet demonstration

We can found that output generated by ResNet34-UNet has many discrete points around the object and for output generated by UNet does not have that feature. Besides, it was very clear that the quality generated by UNet was better than ResNet34-UNet.

5. Execution command

A. The command and parameters for the training process

```
def get_args():
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')
    parser.add_argument('--data_path', type=str, help='path of the input data')
    parser.add_argument('--epochs', '-e', type=int, default=5, help='number of epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=1, help='batch size')
    parser.add_argument('--learning_rate', '-lr', type=float, default=1e-5, help='learning rate')
    parser.add_argument('--model', '-m', type=str, help='choose the model')

    return parser.parse_args()
```

Command: python train.py --data_path [ ] --epochs [ ] --batch_size [ ] --learning_rate [ ] –model [ ]
Where each [ ] fill your arguments.

```
python train.py --data_path ../dataset --epochs 50 --learning_rate 5e-4 --batch_size 8 --model unet
```

Command for training example

B. The command and parameters for the inference process

```
def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model', '-m', default='MODEL.pth', help='path to the stored model weoght')
    parser.add_argument('--data_path', '-d', type=str, help='path to the input data')
    parser.add_argument('--batch_size', '-b', type=int, default=1, help='batch size')
    parser.add_argument('--print', type=int, default=1, help='Print the result')
    parser.add_argument('--plot', type=int, default=1, help='Plot example')

    return parser.parse_args()
```

Command: python inference.py --data_path [ ] --model [ ] --print [ ] --plot [ ]
Where each [ ] fill your arguments.

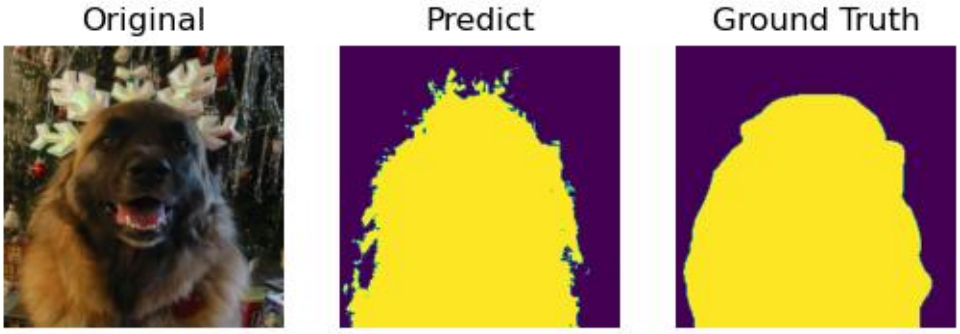
```
python inference.py --data_path ../dataset --model ../saved_models/unet.pth --print 1 --plot 0
```

Command for inference example

6. Discussion

Theoretically ResNet34-UNet should have better performance than UNet since it is its variation and have better architecture with residual connection. However, from my experiment, I do not get the desired result and the result is quite opposite. My UNet has way better performance than ResNet34-UNet and the predicted mask from visually seen are also better.

For dice score, I think it is not a very good metric to evaluate the output. For example, obviously, this is not a good pred and however, its dice score is 0.9374.



ResNet34-UNet bad prediction mask