

NYCU DL

Lab3 – Binary Semantic Segmentation

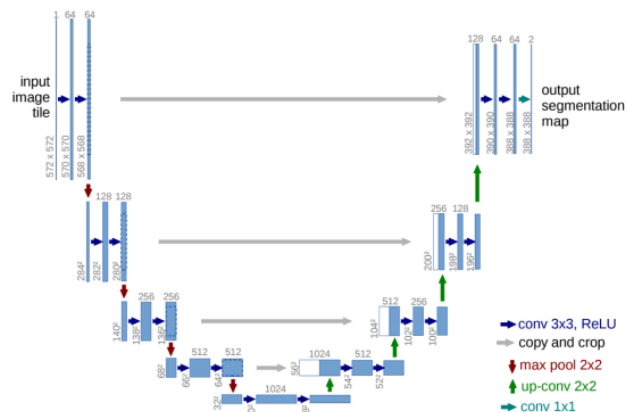
312554006 羅名志

1. Overview of Lab 3

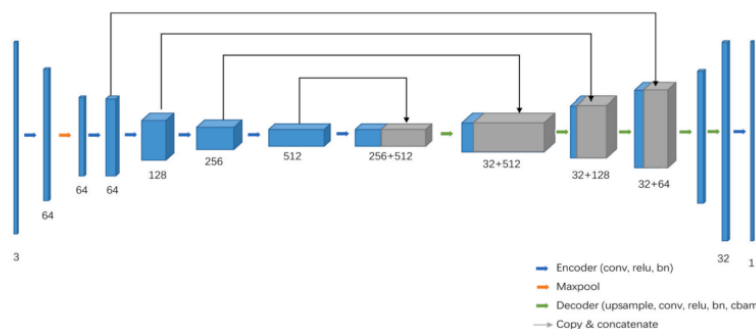
Lab 3 focuses on employing encoder-decoder structures to process image semantic segmentation tasks. In this lab, we will preprocess image data, design the encoder/decoder structure, and train our models to perform well in segmenting image data. Specifically, our designed models need to exhibit a great ability to classify images into foreground and background.

The two structures we will implement are U-Net and ResNet+U-Net:

UNet



ResNet34 (Encoder) + UNet (Decoder)



2. Implementation Details

A. Details of your training, evaluating, inferencing code

(1) Training

After deciding the model to implement, execute the training process as normal training process:

```
# define the model
if args.model == 'Unet':
    model = UNet(in_channels=3, out_channels=1)
else:
    model = ResNet34_Unet([3,4,6,3], 3, 1)
torch.autograd.set_detect_anomaly(True)
model.to(device)
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
criterion = nn.BCELoss()
curr_path = os.getcwd()
pth = os.path.join(curr_path, os.path.join('saved_models', f'{args.model}.pth'))
# train part
for epoch in range(args.epochs):
    model.train()
    running_loss = 0.0
    running_dice = 0.0

    with tqdm(train_loader, desc=f"Epoch {epoch+1}/{args.epochs}", leave = False) as t:
        for sample in t:
            img, mask = sample["image"].to(device), sample["mask"].to(device)
            # print(img.shape, img.dtype)
            # print(mask.shape, mask.dtype)
            # zero the parameters gradients
            optimizer.zero_grad()

            # forward
            outputs = model(img)
            outputs = outputs.squeeze(1)
            mask = mask.squeeze(1)
            # print(outputs[0], mask[0])
            loss = criterion(outputs, mask)
            dice = dice_score(outputs, mask)
            # print(dice)
            # backward
            loss.backward()
            optimizer.step()

            # update the progress bar with current loss
            running_loss += loss.item()*img.size(0)
            running_dice += dice.item()*img.size(0)
            t.set_postfix(loss=loss.item())
```

(2) Evaluating

Evaluate the model with the use of BCE loss and dice score.

```
def evaluate(net, data, device):
    # implement the evaluation function here
    net.to(device)
    net.eval()
    criterion = nn.BCELoss()
    total_loss = 0.0
    total_dice = 0.0

    with torch.no_grad():
        for sample in data:
            inputs, masks = sample["image"].to(device), sample["mask"].to(device)
            outputs = net(inputs)
            loss = criterion(outputs, masks)
            dice = dice_score(outputs, masks)

            total_loss += loss.item() * inputs.size(0)
            total_dice += dice.item() * inputs.size(0)

    avg_loss, avg_dice = total_loss/len(data.dataset), total_dice/len(data.dataset)

    return avg_loss, avg_dice
```

(3) Inferencing

In the inference, first load the model weights and call the evaluate function as defined in part (2).

```
# Define the model
if args.model_name == 'Unet':
    model = UNet(in_channels=3, out_channels=1)
else:
    model = ResNet34_Unet([3,4,6,3], 3, 1)

model.load_state_dict(torch.load(path))
train_loss, train_dice = evaluate(model, train_loader, device)
valid_loss, valid_dice = evaluate(model, valid_loader, device)
test_loss, test_dice = evaluate(model, test_loader, device)

# Print the result
print(f"Model Name: {args.model_name}, Model Path: {path}")
print(f"Train Loss: {train_loss:.4f}, Test Dice Score: {train_dice:.4f}")
print(f"Valid Loss: {valid_loss:.4f}, Valid Dice Score: {valid_dice:.4f}")
print(f"Test Loss: {test_loss:.4f}, Test Dice Score: {test_dice:.4f}")
```

Moreover, I also visualize the instance masking output by using `instance_visualize` function defined in `evaluate.py`. Here for instance-wise visualizing, I only take the first sample in test dataset for an example.

```
instance_input, instance_output, instance_truth = instance_visualize(model, test_loader, device)
```

```
def instance_visualize(net, data, device):
    net.to(device)
    net.eval()
    with torch.no_grad():
        for sample in data:
            inputs, masks = sample["image"].to(device), sample["mask"].to(device)

            outputs = net(inputs)
            outputs[0] = (outputs[0] > 0.5)

    return inputs[0], outputs[0], masks[0]
```

B. Details of your model (Unet and ResNet_Unet)

(1) Unet

Following the basic Unet structure, the encoder and decoder structures are shown below:

```
self.encoder = nn.Sequential(  
    DoubleConv(in_channels, 64),  
    nn.Sequential(  
        nn.MaxPool2d(kernel_size=2),  
        DoubleConv(64, 128)  
    ),  
    nn.Sequential(  
        nn.MaxPool2d(kernel_size=2),  
        DoubleConv(128, 256)  
    ),  
    nn.Sequential(  
        nn.MaxPool2d(kernel_size=2),  
        DoubleConv(256, 512)  
    ),  
    nn.Sequential(  
        nn.MaxPool2d(kernel_size=2),  
        DoubleConv(512, 1024)  
    ),  
)
```

```
self.decoder = nn.Sequential(  
    nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2),  
    DoubleConv(1024, 512),  
    nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2),  
    DoubleConv(512, 256),  
    nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2),  
    DoubleConv(256, 128),  
    nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2),  
    DoubleConv(128, 64),  
    nn.Sequential(  
        nn.Conv2d(64, out_channels, kernel_size=1),  
        nn.Sigmoid()  
    ),  
)
```

Where the DoubleConv is the basic part of Unet:

```
self.double_conv = nn.Sequential(  
    nn.Conv2d(in_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)  
)
```

The special part is to add the encoder part with decoder output:

```
x = self.decoder[0](x5)  
x = torch.cat([x, x4], dim=1)  
x = self.decoder[1](x)
```

(2) ResNet34&Unet

Following Lab2 ResNet design, using make_layer function to construct each block in the ResNet architecture. Also, combine the Unet decoder same as part (1):

```
self.layer1 = self.make_layer(out_c = 64, layer_num = layer_list[0], stride = 1)
self.layer2 = self.make_layer(out_c = 128, layer_num = layer_list[1], stride = 2)
self.layer3 = self.make_layer(out_c = 256, layer_num = layer_list[2], stride = 2)
self.layer4 = self.make_layer(out_c = 512, layer_num = layer_list[3], stride = 2)

self.layer5 = DoubleConv(1024, 512)
self.layer6 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
self.layer7 = DoubleConv(512, 256)
self.layer8 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
self.layer9 = DoubleConv(256, 128)
self.layer10 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
self.layer11 = DoubleConv(128, 64)

self.bridge = nn.Sequential(
    nn.ConvTranspose2d(64, 32, kernel_size = 2, stride = 2),
    nn.BatchNorm2d(32),
    nn.ReLU(inplace = True),
    nn.ConvTranspose2d(32, 16, kernel_size = 2, stride = 2)
)

self.output = nn.Sequential(
    nn.Conv2d(16, 1, kernel_size=1),
    nn.Sigmoid()
)
```

Since combining 2 different models will cause unmatching figure size, I use the bridge function shown above to resize the image to 256*256

C. Anything you want to mention

The way I bridge the model's output to 256*256 is not referred to original paper, but it works here so I didn't modify it anymore.

3. Data Preprocessing

A. How you preprocessed your data

By calling `load_dataset`, the function will return the simple datasets in CHW format with specific mode.

```
def load_dataset(data_path, mode):  
    # implement the load dataset function here  
  
    # First download the data  
    data_path = os.path.join(data_path, os.path.join('dataset', 'os.oxford-iiit-pet/'))  
    OxfordPetDataset.download(data_path)  
  
    # Process data  
    # transform = transforms.ToTensor()  
  
    # Get the dataset  
  
    return SimpleOxfordPetDataset(root = data_path, mode=mode, transform=None)
```

Here I didn't implement any other data preprocess methods since the original data can already had competitive performance.

B. What makes your method unique?

The method is normal as described in part A. I only use the basic data to complete the lab.

C. Anything more you want to mention

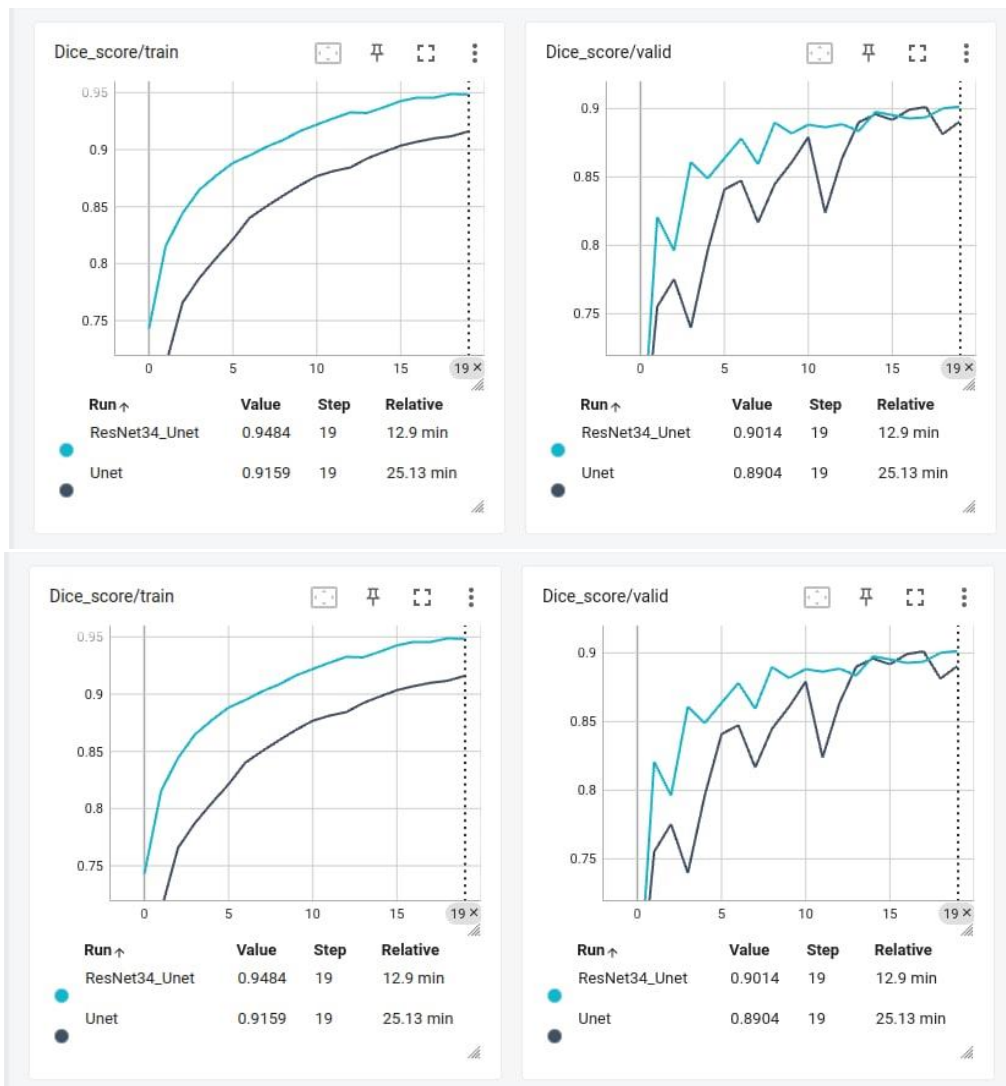
I do have tried normalization and other data augmentation methods. However, the dice score of using these methods is not as good as the one with only using basic data.

```
mean, std = torch.mean(image, dim=(1,2)), torch.std(image, dim=(1,2))  
# print(mean.tolist(), std.tolist())  
# Apply normalization using torchvision transforms (assuming channels are first dimension)  
transform = transforms.Compose([  
    transforms.ToTensor(),  
    transforms.Normalize(mean=mean.tolist(), std=std.tolist())  
)  
img = transform(image)
```

Perhaps the normalization methods I used in the lab is not suitable.

4. Analyze on the experiment results



A. What did you explore during the training process



1. After 12 epochs, the training loss/dice score start to converge under both settings
2. We can tell that the validation set oscillates more dramatically when using the pure Unet architecture.
3. During the training part, I found that both models can reach accurate performance within 5 epochs, making the task easier to train.

B. Found any characteristics of the data

Instance visualization:

Model	Visualization
Unet	
ResNet34 +Unet	

Training results:

```
Model Name: Unet, Model Path: /home/max/DLP/La
Train Loss: 0.1690, Test Dice Score: 0.9138
Valid Loss: 0.2185, Valid Dice Score: 0.8907
Test Loss: 0.2168, Test Dice Score: 0.8911
```

```
Model Name: ResNet34 Unet, Model Path: /home/
Train Loss: 0.0951, Test Dice Score: 0.9535
Valid Loss: 0.2238, Valid Dice Score: 0.9010
Test Loss: 0.2166, Test Dice Score: 0.9061
```

1. The original datasets seem to have already been shuffled, as the model's performance does not drop significantly without shuffling.
2. The distribution of testing datasets seems to be similar to train/valid datasets, since there is no apparent overfitting problem and the testing results is similar to validation.

C. Anything more you want to mention

Dice score is a good evaluation metric when judging the model's accuracy on segmentation task!

5. Execution command

Default: All codes are implemented under Lab3-Binary_Semantic_Segmentation directory

A. The command and parameters for the training process

```
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=20, help='number of epochs')
parser.add_argument('--batch_size', '-b', type=int, default=16, help='batch size')
parser.add_argument('--learning_rate', '-lr', type=float, default=1e-3, help='learning rate')
parser.add_argument('--model', type=str, default='ResNet34_Unet', help='model selection')
return parser.parse_args()
```

Sample:

```
python ./src/train.py --epochs 20 --batch_size 16 --
learning_rate 0.001 --model 'ResNet34_Unet'
```

```
python ./src/train.py --epochs 20 --batch_size 16 --
learning_rate 0.001 --model 'Unet'
```

B. The command and parameters for the inference process

```
parser = argparse.ArgumentParser(description='Predict masks from input images')
parser.add_argument('--model', default='saved_models/ResNet34_Unet.pth', help='path to the stored model weoght')
# parser.add_argument('--data_path', type=str, help='path to the input data')
parser.add_argument('--batch_size', '-b', type=int, default=16, help='batch size')
parser.add_argument('--model_name', '-m', type=str, default='ResNet34_Unet', help='batch size')
```

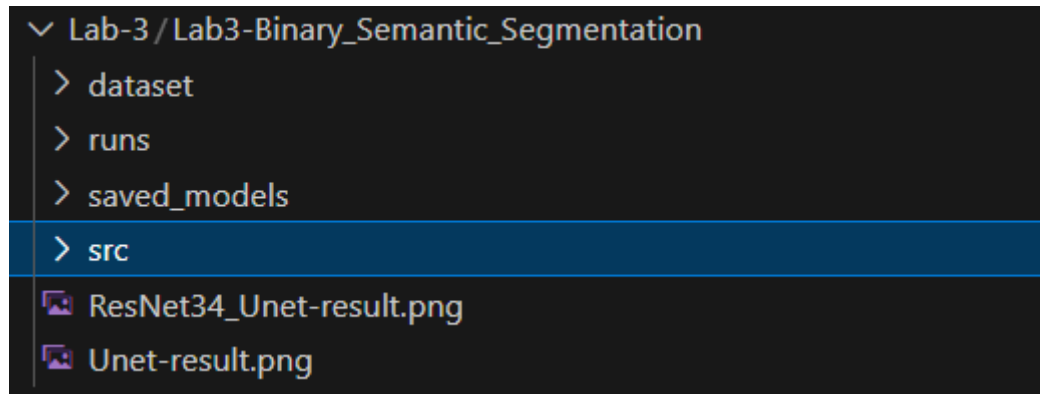
Sample:

```
python ./src/inference.py --model 'saved_models/Unet.pth' --
batch_size 16 --model_name 'Unet'
```

```
python ./src/inference.py --model 'saved_models/ResNet34_Unet.pth' --
batch_size 16 --model_name 'ResNet34_Unet'
```

C. Others

Since I use the TensorBoard to visualize the results in this lab and generate the mask outputs from the models, there are some differences between the directory structure outlined in the spec and mine. (As shown in the follow pic)



6. Discussion

A. What architecture may bring better results?

	Train Dice Score	Valid Dice Score	Test Dice Score
Unet	0.9138	0.8907	0.8911
ResNet34+Unet	0.9535	0.9010	0.9061

As shown in the experiment part, the structure with ResNet34 encoder and Unet decoder yields better result compared to pure Unet architecture. This improvement is likely due to the enhanced feature extraction capabilities on images offered by the ResNet34 encoder.

B. What are the potential research topics in this task.

Since the task is restricted to binary case, where we can only get foreground and background class of a picture. We can further extend to multi-class task. For example, classify a given picture to human, animals, buildings, ornaments, etc.

C. Anything more you want to mention

The architecture of ResNet34 with Unet is both challenging and interesting, as they represent different model structures. Combining them may encounter size matching problems. The method I employed, as discussed in the previous section, may be considered crude, leaving room for improvement in future tasks.

7. Citation

1. ChatGPT: <https://chat.openai.com/chat>
2. Dice score calculation: <https://oecd.ai/en/catalogue/metrics/dice-score>
3. Unet architecture: <https://arxiv.org/abs/1505.04597v1>
4. Unet sample code: <https://github.com/milesial/Pytorch-UNet>
5. Unet sample code: <https://github.com/zhixuhao/unet>
6. ResNet_Unet architecture:
[https://www.researchgate.net/publication/359463249 Deep learning-based pelvic levator hiatus segmentation from ultrasound images](https://www.researchgate.net/publication/359463249_Deep_learning-based_pelviclevatorhiatussegmentationfromultrasoundimages)
7. ResNet_Unet sample code: <https://github.com/GohVh/resnet34-unet>
8. ConvTanspose2d: <https://blog.csdn.net/zbzcDZF/article/details/87881076>