NYCU DL Lab3 Binary Semantic Segmentation

313551073 顏琦恩

I. Overview of your lab 3

In this lab, I implemented UNet and ResNet34-UNet from the paper U-Net: Convolutional Networks for Biomedical Image Segmentation and Deep learning-based pelvic levator hiatus segmentation from ultrasound images.

In the training section, I'd trained two models, UNet and ResNet34-UNet. I use the dataset provided by TAs, which is The Oxford-IIIT Pet Dataset, to train my model.

Moreover, I'd implemented some tool functions in file utils.py, which are dice score calculating function, three plotting functions, functions for loading and saving models and loss. Plotting functions are for the loss curve and the predicted masks. The saving function for loss is used when one needs to keep on training the same model. To draw a complete loss curve, this function saves losses into a txt file, one can use the loading function to read the txt file and draw the complete loss curve.

```python
# 畫多張圖、ground truth、predicted mask，一行最多8張圖
def plot_grid(image, mask, pred_mask):
    fig = plt.figure(figsize=(10, 12))
    fig.suptitle('Comparison of Original Image, Mask, and Predicted Mask', fontsize=12)

    fig.add_subplot(3, 1, 1)
    plt.imshow(T.ToPILImage()(torchvision.utils.make_grid(image / 255, nrow=8)))
    plt.axis('off')
    plt.title("Original Image")

    fig.add_subplot(3, 1, 2)
    plt.imshow(T.ToPILImage()(torchvision.utils.make_grid(mask, nrow=8)))
    plt.axis('off')
    plt.title("Ground Truth Masks")

    fig.add_subplot(3, 1, 3)
    plt.imshow(T.ToPILImage()(torchvision.utils.make_grid(pred_mask, nrow=8)))
    plt.axis('off')
    plt.title("Predicted Masks")

    plt.show()
```

Fig. 1 Function for plotting multiple images at the same time

II. Implementation Details

A. Details of your training, evaluating, inferencing code

In my training section, I applied binary cross-entropy as my loss function and Adam as my optimizer. I used binary cross-entropy because this lab only contains two classes: foreground and background. To avoid the error of cuda out of memory, I clean up the gpu

cache after finishing every epoch. Also, to keep training the model, I added a variable retrain_model to keep on training my previous trained model and save losses of training to draw the complete training loss curve. During training, I check if the dice score of the validation dataset gets better, and save the model which has the highest dice score in the process. At the end of the training function, I plot the loss curve of the training section.

```python
def train(args):
    # implement the training function here
    train_dataloader = DataLoader(opData.load_dataset(args.data_path, 'train'), batch_size=args.batch_size, shuffle=True)
    valid_dataloader = DataLoader(opData.load_dataset(args.data_path, 'valid'), batch_size=args.batch_size, shuffle=True)

    if args.model == 'unet':
        model = unet.UNet(3).to(args.device)
    elif args.model == 'resnet':
        model = resnet34_unet.ResNet34_UNet(3).to(args.device)

    criterion = nn.BCELoss()    # 用於只有binary class的cross entropy
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    losses = []
    scores = []
    max_score = 0

    # 如果要繼續train同一個model，可以使用這個部分，loss會寫入txt檔中，所以可以畫完整的loss curve
    if args.retrain_model != '':
        model, optimizer, max_score  = load_model(f'./saved_models/{args.retrain_model}', model, optimizer)
        losses = load_loss(args.model)

    # start training
    last_epoch = len(losses)    # 從上次訓練到的epoch繼續畫loss curve，如果是train from scratch，就從epoch 0開始畫loss curve

    model.train()
    for epoch in range(args.epochs):

        epoch_cost = []
        epoch_score = []
        for sample in tqdm(train_dataloader):
            img = torch.Tensor(sample['image']).float().to(args.device)
            mask = torch.Tensor(sample['mask']).float().to(args.device)
            optimizer.zero_grad()

            pred = model(img)
            loss = criterion(pred, mask)
            loss.backward()
            optimizer.step()

            epoch_cost.append(loss)
            epoch_score.append(dice_score(pred, mask))

        losses.append((sum(epoch_cost)/len(epoch_cost)).item())
        scores.append((sum(epoch_score)/len(epoch_score)).item())
        print(f'[Epoch {epoch+1}] loss = {losses[-1]:.9f}, average dice score = {scores[-1]:.9f}')

        # validation
        valid_score = evaluate(model, valid_dataloader, args.device)
        print(f'Validation dice score = {valid_score}')
        if valid_score > max_score:
            save_model(f'{args.save_path + args.model}.pth', model, optimizer, epoch, valid_score)
            max_score = valid_score
        save_loss(args.model, losses[last_epoch+epoch:])    # 將loss寫入txt檔
        # 避免gpu out of memory
        gc.collect()
        torch.cuda.empty_cache()

    plot_loss(losses, args.model)
```

Fig. 2 Details of training code

The structure of evaluating and inferencing code is familiar with each other. The difference between them is that I need to load my trained model for inference, but don't need to load the model in the evaluating section, since I always pass the model to evaluate into the evaluating function.

```python
def evaluate(net, data, device):
    # implement the evaluation function here
    avg_score = []
    net.eval()
    with torch.no_grad():
        for sample in tqdm(data):
            img = torch.tensor(sample['image']).float().to(device)
            mask = torch.tensor(sample['mask']).float().to(device)

            pred = net(img)
            avg_score.append(dice_score(pred, mask))

    return sum(avg_score)/len(avg_score)
```

Fig. 3 Details of evaluating code

```python
if __name__ == '__main__':
    args = get_args()

    dataloader = DataLoader(opData.load_dataset(args.data_path, 'test'), batch_size=args.batch_size, shuffle=True)
    if args.model_to_use == 'unet':
        model = load_model(f'./saved_models/{args.model}', unet.UNet(3))
    elif args.model_to_use == 'resnet':
        model = load_model(f'./saved_models/{args.model}', resnet34_unet.ResNet34_UNet(3))

    avg_score = []
    model.to(args.device)
    model.eval()
    with torch.no_grad():
        for sample in tqdm(dataloader):
            img = torch.tensor(sample['image']).float().to(args.device)
            mask = torch.tensor(sample['mask']).float().to(args.device)

            pred = model(img)
            score = dice_score(pred, mask)
            avg_score.append(score)
            if max(avg_score) == score and args.plot == 'y':
                plot_grid(img, mask, pred)
        print(f'Inference dice score = {sum(avg_score)/len(avg_score)}')
```

Fig. 4 Details of inferencing code

B. Details of your model (UNet & ResNet34_UNet)

The UNet is composed of downsample blocks and upsample blocks. Each downsample block contains a maxpool, two 2d convolutions, relus, and batch normalizations, while the upsample block contains a transposed convolution and two 2d convolutions.

In the downsampling section, poolings are used to capture more abstract and higher resolution data, since they divide features into regions and keep the maximum value. For the kernel size, stride, and padding of convolutions and maxpool, I use the same value as the paper mentioned.

For the upsampling method, I used the transposed convolution, which is used to recover the spatial resolution of the image, the kernel size and stride is also used as the value mentioned in the paper.

```python
# 透過downsample降低spatial resolution，來提取提取high-resolution、low-level的特徵
class DownsampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels) -> None:
        super(DownsampleBlock, self).__init__()

        self.conv = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2),
            Conv2dBlock(in_channels, out_channels, 3)
        )

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

# 恢復spatial resolution，同時skip connection可以維持downsample時提取的特徵
class UpsampleBolck(nn.Module):
    def __init__(self, in_channels, out_channels) -> None:
        super(UpsampleBolck, self).__init__()

        self.upconv =  nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.conv = Conv2dBlock(in_channels, out_channels, kernel_size=3)

    def forward(self, x, prev_x):
        x = self.upconv(x)
        x = torch.cat([prev_x, x], dim=1)   # skip connection

        x = self.conv(x)
        return x
```

Fig. 5 The downsample and upsample block for UNet

My ResNet34_UNet took ResNet as the downsample part, which is the so-called encoder, while UNet did the upsample part, the decoder. The convolution blocks in the encoder are set to the same kernel size, stride, and padding as the ResNet paper.

In the upsample part, the channels of each decoder seem to be wrong, so I adjusted them to the same values as my Unet. Also, the paper of ResNet34-Unet mentioned that they add CBAM in each decoder, so I added it in my network, too. The remaining part of the decoder shared the same structure with the upsample part of UNet.

```python
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, conv_in_channels=0) -> None:
        super(DecoderBlock, self).__init__()
        if conv_in_channels == 0:
            conv_in_channels = in_channels

        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2, bias=False)
        self.unetconv = UnetConv(conv_in_channels, out_channels)
        self.cbam = CBAM(out_channels)

    def forward(self, x, prev_x):
        x = self.upconv(x)
        x = torch.cat([prev_x, x], dim=1)

        x = self.unetconv(x)
        x = self.cbam(x)
        return x
```

```
class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, total_blocks, is_downsample=True) -> None:
        super(EncoderBlock, self).__init__()

        blocks = []
        blocks.append(ResBlock(in_channels, out_channels, kernel_size=3,
                               stride=1+1*is_downsample, is_downsample=is_downsample))
        for _ in range(1, total_blocks):
            blocks.append(ResBlock(out_channels, out_channels, kernel_size=3, stride=1))

        self.layer = nn.Sequential(*blocks)

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

Fig. 6 The decoder and encoder block of ResNet34_UNet

III. Data Preprocessing

A. How you preprocessed your data?

I applied a random horizontal flip to the dataset with a probability of 0.5. The following figure is my flipping function.

```
def transform(image, mask, trimap):
    if np.random.rand() < 0.5:
        image = np.fliplr(image).copy()
        mask = np.fliplr(mask).copy()
        trimap = np.fliplr(trimap).copy()
    return {'image': image, 'mask': mask, 'trimap': trimap}
```

Fig. 7 The preprocess function

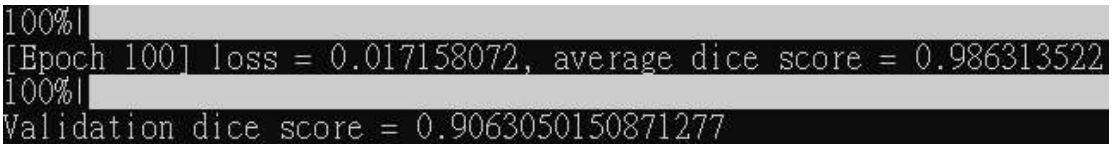
B. What makes your method unique?

Although just flipping images horizontally is a quite common way of data augmentation, it is an appropriate method in this lab. Since it could be too much if we applied many kinds of methods at the same time, which may decrease the resolution of images.

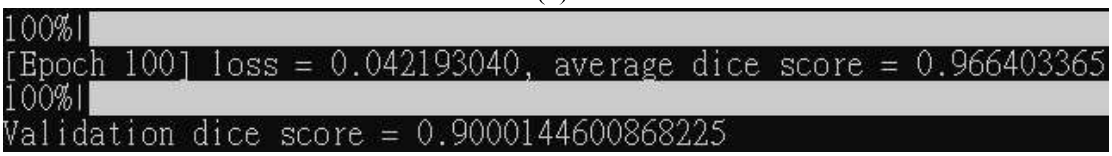
IV. Analyze on the experiment results

A. What did you explore during the training process?

Both of the networks converge very fast on the training dataset, especially the UNet network, they also achieve good performance on validation and testing dataset.



(a)



(b)

Fig. 8 The dice score of UNet (a) and ResNet34_UNet (b) in the last epoch. (Both networks trained for 100 epochs with batch size 2 and learning rate 0.00001)

From the following figures, we can see that though both of the networks achieved a very low loss, the UNet only needs 20 epochs to converge, while the ResNet34_UNet needs 30 epochs or more. I think that may be because of the size of the network. Larger or deeper networks usually need more time to converge than smaller networks, since they have more weights to train, which may cause gradient vanishing or exploding problems.

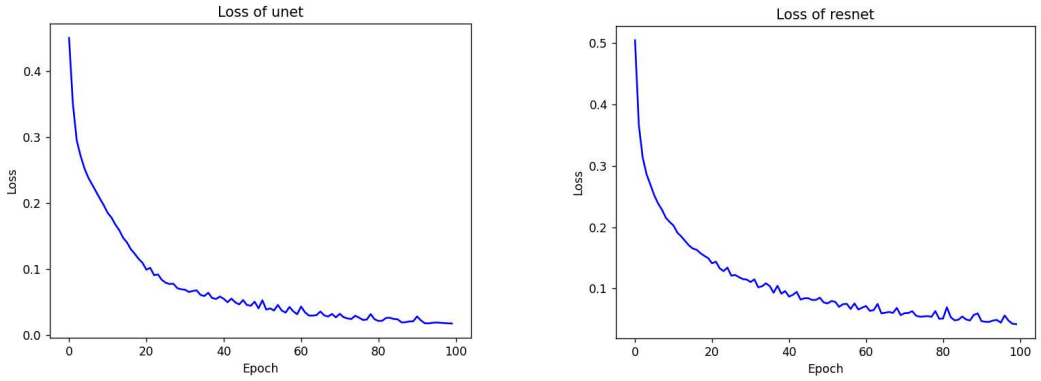
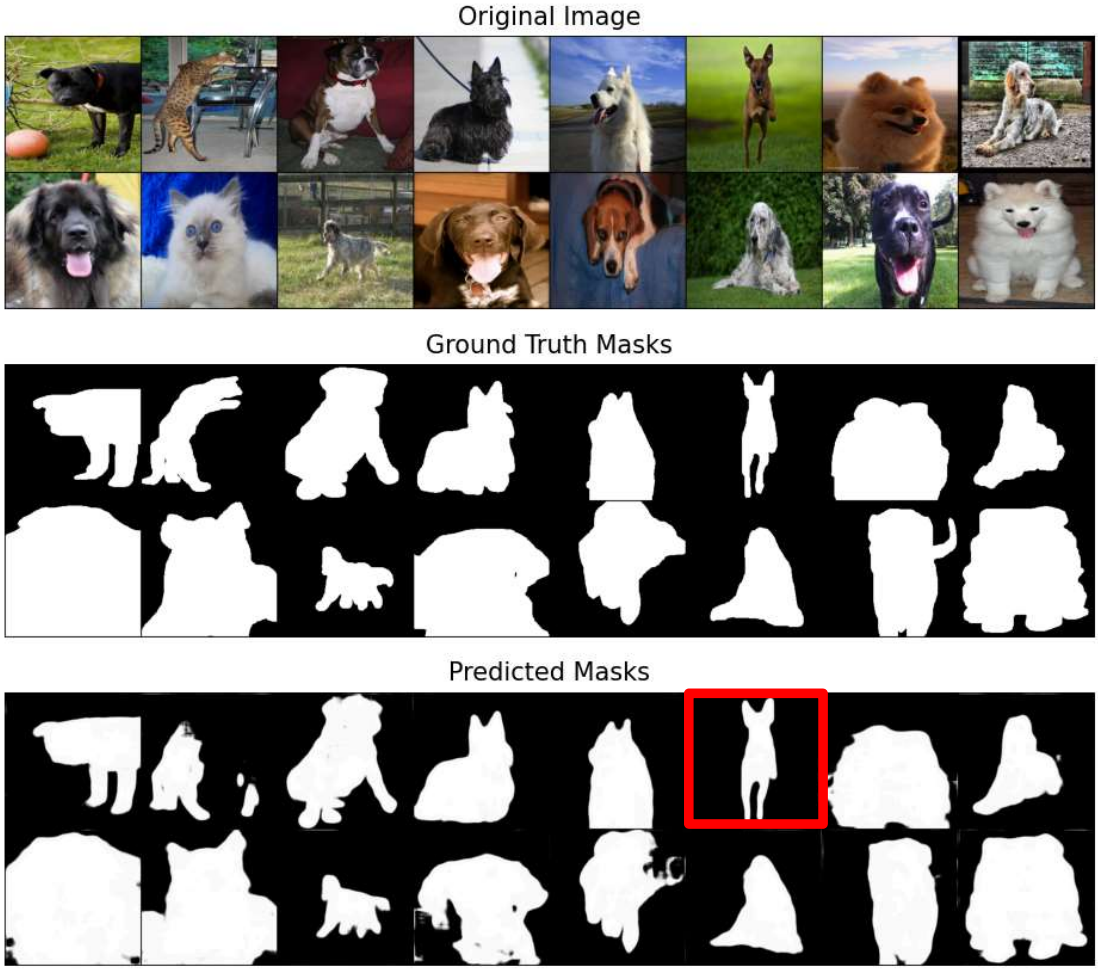
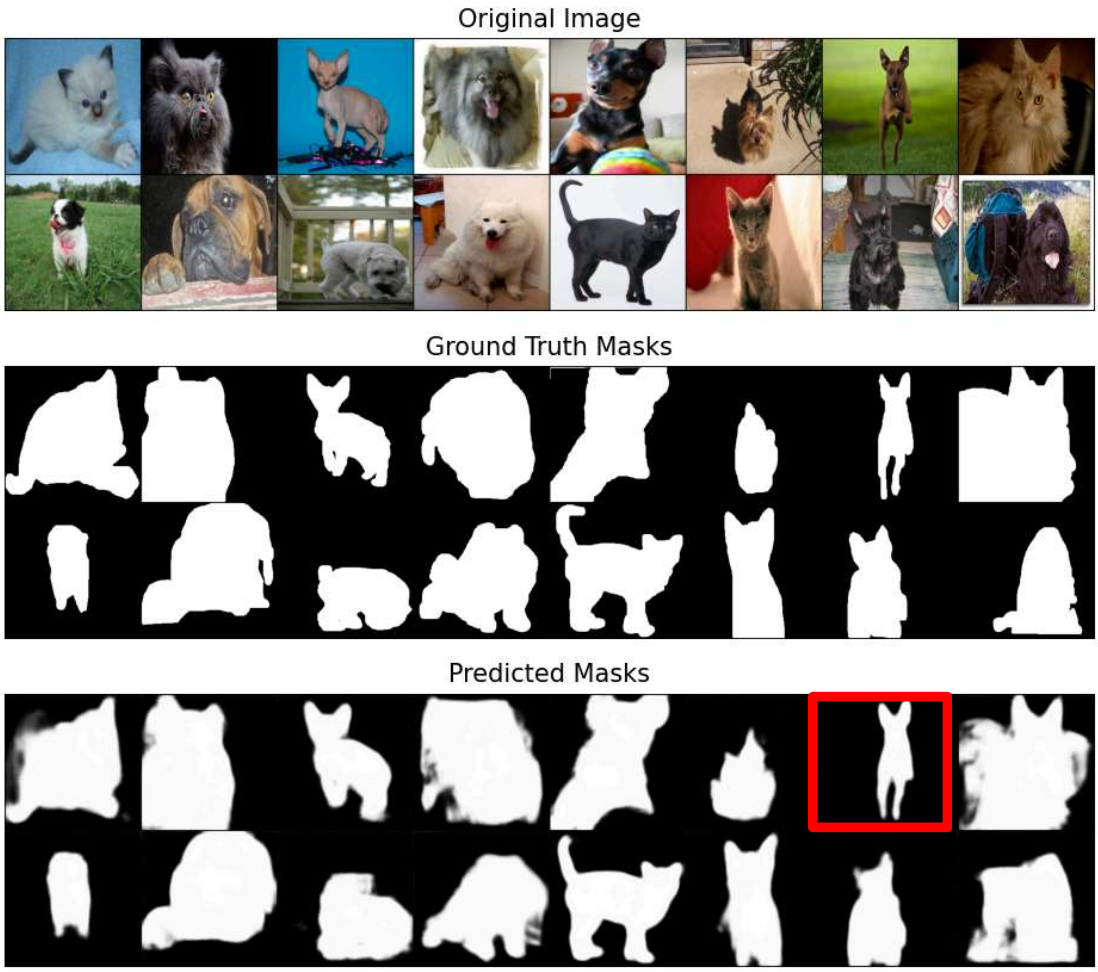


Fig. 9 Loss curve of UNet (left) and ResNet34_UNet (right). The loss of UNet can be converged to about 0.01, while ResNet34_UNet converges to 0.04

Moreover, I observed that results of UNet (Fig.10-a) are sharper than the ResNet34_UNet (Fig.10-b). In the below figure, we can see that although two networks predicted well on these images, the prediction of ResNet34_UNet seems to have some blur.



(a)



(b)

Fig. 10 Predicted Masks of UNet (a) and ResNet34_UNet (b). We can apparently observe that UNet predicts a sharper mask.

B. Found any characteristics of the data?

Objects in the images have large variations in pose, lighting, and scale, this characteristic allows networks to learn a more general feature to the foreground class, thus can achieve a similar score in the training and inferencing section.

V. Execution command

For an anaconda environment, if you are at the root of the folder, please use the following command to train the model.

A. The command and parameters for the training process

Model	Command
UNet	python src/train.py -m unet
ResNet34-UNet	python src/train.py -m resnet

Using the command above will train 50 epochs with the batch size of 2.

Parameters	Description	Default Value
-e	number of training epochs	50
-b	batch size	2
-lr	learning rate	1e-5
-m	model to train (options: unet/resnet) **(required)**	resnet
-d	device (option: cuda/cpu)	cuda
-p	save path of model checkpoints	./saved_models/
-r	name of model to keep on training (ex. unet.pth)	empty string

B. The command and parameters for the inference process

Model	Command and parameters
UNet	python src/inference.py -m DL_Lab3_UNet_313551073_顏琦恩.pth -u unet -b 16
ResNet34-UNet	python src/inference.py -m DL_Lab3_ResNet34_UNet_313551073_顏琦恩.pth -u resnet -b 16

Using the command above will inference with the batch size of 16.

Parameters	Description	Default Value
-m	path to the stored model weight (ex. resnet.pth)	DL_Lab3_UNet_313551073_顏琦恩.pth
-b	batch size	2
-d	device (option: cuda/cpu)	cuda
-u	model to inference (options: unet/resnet) **(required)**	resnet

-p	plot during inferencing or not (option: y/n), it will show the batch which has the highest score before the batch inferencing	n

VI. Discussion

A. What architecture may bring better results?

In my experiment, the UNet architecture has better results than the ResNet34-UNet, but I think if there is more training data, or the testing data is more complicated, the ResNet34-UNet would get a better result, since the network is more deeper to learn a more complicated feature.

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

I think we can investigate the impact and effectiveness of attention mechanisms, such as CBAM, on image segmentation. Additionally, we can explore whether adding attention mechanisms into UNet can help improve accuracy.