DL Lab 2 – Binary Semantic Segmentation

曾大衛

1. Introduction

In Lab 2, I implement UNet and ResNet34_Unet architecture with Pytorch to do binary semantic segmentation on Oxford-IIIT PET dataset. In Oxford-IIIT PET dataset, I design my own load_dataset function that performs data preprocessing and returns the dataset to dataloader. Furthermore, I evaluate the model with test dataset by using dice score function when inferencing the model. After inferencing, the output images will be generated from either UNet or ResNet34Unet model and saved in a folder.

2. Implementation details

This section will introduce the whole procedure from data preprocessing to inferencing. I will discuss every step in detail.

2.1. Data preprocess

```
def load_dataset(data_path, mode):
    OxfordIIITPet(root="dataset", download=True, target_types="segmentation")
    def transform(image, mask, trimap):
        if mode == "train":
            #   Rotation
            deg = np.random.randint(0, 30)
            deg -= 15
            image = imutils.rotate(image, deg)
            mask = imutils.rotate(mask, deg)
            trimap = imutils.rotate(trimap, deg)

            #   Horizontal Flip
            flip = np.random.randint(0, 2)
            if flip:
                image = cv2.flip(image, 1)
                mask = cv2.flip(mask, 1)
                trimap = cv2.flip(trimap, 1)

            #   Vertical Flip
            flip = np.random.randint(0, 2)
            if flip:
                image = cv2.flip(image, 0)
                mask = cv2.flip(mask, 0)
                trimap = cv2.flip(trimap, 0)

        return dict(image=image, mask=mask, trimap=trimap)
    return SimpleOxfordPetDataset(root=data_path, mode=mode, transform=transform)
```

As the figure above, I use rotation, horizontal flip and vertical flip to increase image diversification and data augmentation. During these processes, it will increase image variations to prevent the model from being limited to a fixed orientation and improve its generalization ability.

2.2. Model architecture

```python
# Implement your UNet model here
import torch
from torch import nn

class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.decoder = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            # Added Upsampling layer to match the input size
            nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2), # Upsample
            nn.ReLU(),
            nn.Conv2d(64, out_channels, kernel_size=1)
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return torch.sigmoid(x)
```

```python
# Implement your ResNet34_UNet model here
import torch
from torch import nn

class ResNet34Encoder(nn.Module):
    def __init__(self, in_channels):
        super(ResNet34Encoder, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(64, 3)
        self.layer2 = self._make_layer(128, 4, stride=2)
        self.layer3 = self._make_layer(256, 6, stride=2)
        self.layer4 = self._make_layer(512, 3, stride=2)

    def _make_layer(self, out_channels, blocks, stride=1):
        # The input channels to the first layer should match the output channels of the previous layer.
        in_channels = out_channels // 2 if stride != 1 else out_channels # Calculate input channels based on stride
        layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False), # Use calculated input channels
                  nn.BatchNorm2d(out_channels),
                  nn.ReLU(inplace=True)]
        for _ in range(1, blocks):
            layers.append(nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False))
            layers.append(nn.BatchNorm2d(out_channels))
            layers.append(nn.ReLU(inplace=True))
        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)
        return x
```

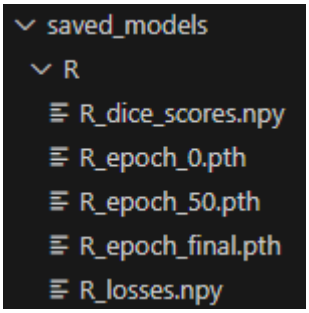
```python
class ResNet34UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNet34UNet, self).__init__()
        self.encoder = ResNet34Encoder(in_channels)
        self.decoder = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2),  # Upsample 1
            nn.ReLU(),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2),  # Upsample 2
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2),  # Upsample 3
            nn.ReLU(),
            nn.ConvTranspose2d(16, 8, kernel_size=2, stride=2),  # Upsample 4
            nn.ReLU(),
            nn.ConvTranspose2d(8, 4, kernel_size=2, stride=2),  # add Upsample 5
            nn.ReLU(),
            nn.Conv2d(4, out_channels, kernel_size=3, padding=1)  # resize the output layer
        )


    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return torch.sigmoid(x)
```

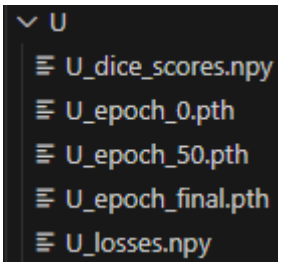
As you can see in figure 1, it is a UNet architecture. There are one encoder and one decoder in UNet. It will update the weight by using forward and sigmoid. What's more, in figure 2 and figure 3, there is a ResNet34 encoder that creates layers, and it uses UNet decoder but more Upsample layers to make model complicated.

2.3. Train

```python
def train(args, model):
    if args.load_model_epoch != 0:
        model.load_state_dict(torch.load(f"saved_models/{args.model}/{args.model}_epoch_{args.load_model_epoch}.pth"))

    train_loader = DataLoader(load_dataset(args.data_path, "train"), batch_size=args.batch_size, shuffle=True)
    valid_loader = DataLoader(load_dataset(args.data_path, "valid"), batch_size=args.batch_size, shuffle=False)

    criterion = nn.BCELoss()  # make sure loss correct
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, 0.99)

    losses = []
    dice_scores = []
    for i in range(args.epochs):
        model.train()
        train_loss = 0
        for sample in tqdm(train_loader):
            image = sample["image"].to(device).float()
            mask = sample["mask"].to(device).float()

            pred_mask = model(image)  # model predict
            loss = criterion(pred_mask, mask)  # calculate traning loss
            train_loss += loss.item()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
Epoch 78, Traning Loss: 0.1438, Dice Score: 0.8859, LR: 0.004612
100%|
100%|
Epoch 79, Traning Loss: 0.1448, Dice Score: 0.8782, LR: 0.004566
100%|
100%|
Epoch 80, Traning Loss: 0.1410, Dice Score: 0.8913, LR: 0.004520
100%|
100%|
```





In the training phase, I import torch to create dataloader, BCELoss, Adam optimizer and optimizer scheduler. The whole training process will be implemented under 1 GPU device. When every epoch, the training function with changing learning rate by optimizer scheduler will calculate training loss and dice score. In addition, I attempt to save the models whenever 50 epochs and final model at the final epoch.

2.4. Evaluate

```python
import torch

def dice_score(pred_mask, gt_mask):
    # implement the Dice score here
    with torch.no_grad():
        sum = 0
        pred_mask = pred_mask > 0.5
        for i in range(pred_mask.shape[0]):
            intersection = torch.sum(pred_mask[i] * gt_mask[i])
            sum += 2.0 * intersection / (torch.sum(pred_mask[i]) + torch.sum(gt_mask[i]))
    return (sum / pred_mask.shape[0]).item()
```

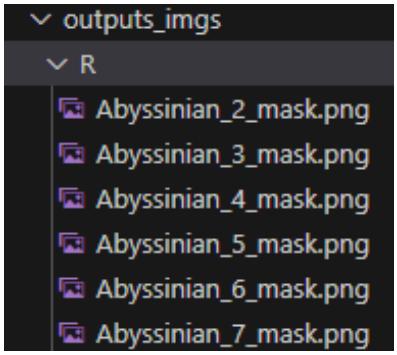
In the evaluating phase, I design a dice_score function based on the formula given in the lab 2 document.

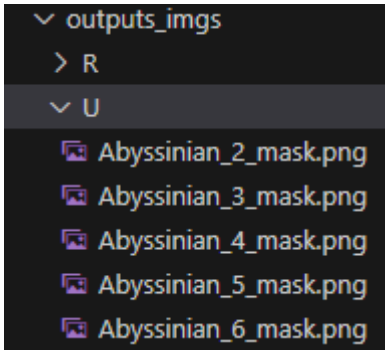
2.5. Inference

```python
if __name__ == '__main__':
    args = get_args()
    if args.model == "U":
        print("Using UNet model")
        model = UNet(3, 1).to(device)
    elif args.model == "R":
        print("Using ResNet34_UNet model")
        model = ResNet34UNet(3, 1).to(device)

    print(f"Loading model from saved_models/{args.model}/{args.model}_epoch_{args.load_model_epoch}.pth")
    model.load_state_dict(torch.load(f"saved_models/{args.model}/{args.model}_epoch_{args.load_model_epoch}.pth"))
    test_loader = DataLoader(load_dataset(args.data_path, "test"), batch_size=args.batch_size, shuffle=False)

    print("Evaluating model")
    dice_score = evaluate(model, test_loader)
    print(f"Dice Score: {dice_score:.4f}")

    list_path = args.data_path + '/annotations/test.txt'
    with open(list_path) as f:
        filenames = f.read().strip('\n').split('\n')
    filenames = [x.split(' ')[0] for x in filenames]

    os.makedirs(f'outputs_imgs/{args.model}', exist_ok=True)
    for file in tqdm(filenames):
        img_path = args.data_path + '/images/' + file + '.jpg'
        data = preprocess_data(img_path)
        data = data.unsqueeze(0).to(device)
        mask = model(data).cpu().detach().numpy().reshape(256, 256)
        mask = mask > 0.5
        new_img = to_img(data, mask)
        new_img.save(f'outputs_imgs/{args.model}/{file}_mask.png')
```

```
Using UNet model
Loading model from saved_models/U/U_epoch_final.pth
Evaluating model
100%|
Dice Score: 0.7491
100%|
```

```
Using ResNet34_UNet model
Loading model from saved_models/R/R_epoch_final.pth
Evaluating model
100%|
Dice Score: 0.9011
100%|
```





In inferencing phase, I use each trained model from saved model folder to inference. The model will be calculated the dice score and utilized to generate images from test dataset. Also, those generated images can be produced after preprocessing and will be saved in the output images folder.

3. Experimental result

After two model experiments, I found that the dice score from UNet model is much lower than that from ResNet34_UNet. Besides, I also set up different parameters for each model, such as learning rate. For UNet and RetNet34_UNet respectively, learning rate is ideally set up under 1e-3 and 1e-2. Since UNet is a bit simple, its dice score is not great enough when training and inferencing. However, RetNet34_UNet is more complex than UNet, so its dice score is much higher than that of UNet when either training or inferencing. As the results from above section, UNet's dice score just 0.7491 and RetNet34_UNet achieves 0.9011.

4. Discussion

SegFormer：

Uses Vision Transformers (ViTs) instead of CNNs, enabling better global feature representation.

Works well on low-power devices while still maintaining high accuracy.

Enables better handling of objects with varying scales.

Suitable for low-level training dataset.

UNet++：

Unlike standard UNet, UNet++ introduces dense skip connections to refine feature propagation. It reduces the semantic gap between encoder and decoder, leading to better segmentation of fine details.

Works well with small datasets, as it does not require extensive training like Transformer models.

Despite its improved skip connections, UNet++ is still lighter than Transformer-based architectures like SegFormer.