DL Lab3

Overview

In this lab, I implemented ResNet34_UNet and UNet to train a model for pixel-wise binary semantic segmentation. I implemented unet and resnet34_unet in two .py files and import them when needed. I use the 'load_dataset' function in oxford_pet.py to load the dataset, and train it in train.py.

Implementation Details

A. Details of your training, evaluating, inferencing code

Train

I add an argument to set which model to train. I set default values of all the arguments, so I only need to send lr and model during training.

```python
def get_args():
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')
    parser.add_argument('--data_path', type=str, default="../dataset/oxford-iiit-pet", help='path of the input data')
    parser.add_argument('--epochs', '-e', type=int, default=100, help='number of epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=8, help='batch size')
    parser.add_argument('--learning-rate', '-lr', type=float, default=3e-5, help='learning rate')
    parser.add_argument('--model', '-m', type=str, default='unet', help='model to use', choices=['unet', 'resnet34_unet'])

    return parser.parse_args()
```

Training function. Details in comment.

```python
def train(args):
    # implement the training function here
    # extract arguments
    data_path = args.data_path
    epochs = args.epochs
    batch_size = args.batch_size
    learning_rate = args.learning_rate
    model_n = args.model
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # prepare data, model, and loggers
    train_writer = SummaryWriter('log/'+model_n+'/train_lr' + str(learning_rate))
    valid_writer = SummaryWriter('log/'+model_n+'/valid_lr' + str(learning_rate))
    train_dataset = load_dataset(data_path, mode='train')
    train_data_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    valid_dataset = load_dataset(data_path, mode='valid')
    valid_data_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
    if model_n == 'resnet34_unet':
        model = ResNet34_UNet(in_channels=3, out_channels=1).to(device)
    else:
        model = UNet(in_channels=3, out_channels=1).to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=learning_rate)
```

```python
# start training
for epoch in range(epochs):
    running_loss = 0.0
    for _, sample in enumerate(train_data_loader):
        image = sample['image'].to(device)
        mask = sample['mask'].to(device)
        # zero the gradients
        optimizer.zero_grad()

        # forward + backward + optimize (I already flatten the mask in preprocessing)
        outputs = model(image)
        outputs = outputs.flatten(start_dim=1, end_dim=3)
        loss = criterion(outputs, mask)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    train_writer.add_scalar('training loss', running_loss / len(train_data_loader), epoch)
    print(f'Epoch {epoch + 1}, Loss: {running_loss / len(train_data_loader)}')

    # I originally set it to 2, but eventually I found that I need to evaluate every epoch
    if epoch % 1 == 0:
        train_score = evaluate(model, train_data_loader, device)
        train_writer.add_scalar('dice score', train_score, epoch)
        print(f'Dice score: {train_score}')
        validate_score = evaluate(model, valid_data_loader, device)
        valid_writer.add_scalar('dice score', validate_score, epoch)
        print(f'Validation Dice score: {validate_score}')

    # save the last 5 models
    if(epoch >= epochs - 5):
        torch.save(model.state_dict(), f'../saved_models/{model_n}_{epoch}_{validate_score}.pth')
print('Finished Training')
```

Evaluate

Function to calculate dice score (in utils.py)

```python
def dice_score(pred_mask, gt_mask):
    # implement the Dice score here
    '''
    pred_mask: torch.Tensor, shape=(1, 256, 256), predicted mask
    gt_mask: torch.Tensor, shape=(1, 256, 256), ground truth mask
    '''
    with torch.no_grad():
        sum = 0
        pred_mask = pred_mask > 0.5
        pred_mask = pred_mask.cpu().numpy().astype(np.float32)
        gt_mask = gt_mask.cpu().numpy().astype(np.float32)
        pred = pred_mask.flatten()
        gt = gt_mask
        # intersect = pixel level multiplication
        intersection = np.sum(pred * gt)
        # area of mask = sum of all pixels
        area_pred = np.sum(pred)
        area_gt = np.sum(gt)
        # dice score formula
        sum += 2 * intersection / (area_pred + area_gt)
    return sum
```

The evaluate function (loop and calculate dice_score).

```
def evaluate(net, data, device):
    # implement the evaluation function here
    '''
    input: model, dataloader, device
    output: average dice score

    loop through the data and calculate the dice score for each image
    '''
    score = 0
    pic_num = 0
    with torch.no_grad():
        for _, sample in enumerate(data):
            image = sample['image'].to(device)
            mask = sample['mask'].to(device)
            outputs = net(image)
            for i in range(outputs.shape[0]):
                score += dice_score(outputs[i], mask[i])
            pic_num += outputs.shape[0]
    return score / pic_num
```

Inference

Here I support two kinds of inference controlled in the argument. One is loading the testing set and output the average dice score and save the image of the original image overlaid with the mask. The other is to input path of the single image and saves the overlaid image.

```
def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model_path', default='../saved_models/unet.pth', help='path to the stored model weight', choices=['../saved_m
    parser.add_argument('--data_path', type=str, default='../dataset/oxford-iiit-pet', help='path to the input data')
    parser.add_argument('--mode', '-m', type=str, default='txt', help='txt or png', choices=['txt', 'png'])
    return parser.parse_args()
```

Functions to set up model / preprocess / predict

```
def set_model(args):
    model_name = args.model_path.split('/')[-1].split('.')[0]
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if model_name == 'unet':
        model = UNet(in_channels=3, out_channels=1)
    elif model_name == 'resnet34_unet':
        model = ResNet34_UNet(in_channels=3, out_channels=1)

    model.load_state_dict(torch.load(args.model_path))
    model = model.to(device)
    return model

def load_preprocess(image_path):
    data = Image.open(image_path).convert("RGB")
    data = np.array(data.resize((256, 256), Image.BILINEAR))
    data = (np.moveaxis(data, -1, 0) / 255.0).astype(np.float32)
    return data

def predict(model, data):
    prediction = model(data).cpu().detach().numpy().reshape(256, 256)
    prediction = prediction > 0.5
    return prediction
```

Function to inference a single image given image path

```
def inference_png(args):
    path = args.data_path
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = set_model(args)

    os.makedirs('inferenced_image', exist_ok=True)
    data = load_preprocess(path)
    data = torch.tensor(data).unsqueeze(0)
    data = data.to(device)
    prediction = predict(model, data)
    save_name = path.split('/')[-1].split('.')[0]
    new_img = visualize(data.cpu().numpy(), prediction)
    new_img.save('inferenced_image/'+ save_name + "_mask.png")
```

Function to inference from txt

```
def inference_txt(args):
    path = args.data_path
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    dataset = load_dataset(path, mode='test')
    dataset = torch.utils.data.DataLoader(dataset, batch_size=8, shuffle=False)
    model = set_model(args)
    acc = evaluate(model, dataset, device)
    print(f'Dice score: {acc}')

    # I need the file names to save the images
    csv_path = path + '/annotations/test.txt'
    with open(csv_path) as f:
        split_data = f.read().strip("\n").split("\n")
    filenames = [x.split(" ")[0] for x in split_data]

    os.makedirs('inferenced_image', exist_ok=True)
    for file in filenames:
        complete_path = path + '/images/' + file + '.jpg'
        data = load_preprocess(complete_path)
        data = torch.tensor(data).unsqueeze(0)
        data = data.to(device)
        prediction = predict(model, data)
        new_img = visualize(data.cpu().numpy(), prediction)
        new_img.save('inferenced_image/'+ file + "_mask.png")
```

Function to visualize the mask and image

```
def visualize(data, mask):
    data = data.squeeze(0)
    data = np.transpose(data, (1, 2, 0))
    mask = np.stack([mask, mask, mask], axis=-1)
    mask = mask * 255
    data = data * 255
    mask = mask.astype(np.uint8)
    mask = Image.fromarray(mask)
    data = Image.fromarray((data).astype(np.uint8))
    new_img = Image.blend(data, mask, 0.5)
    return new_img
```

B. Details of your model (UNet & ResNet34_UNet)

UNet

Encoder/Decoder block

```python
class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(EncoderBlock, self).__init__()

        self.block = 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),
        )
        self.down = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.block(x)
        output = self.down(x)
        return output, x

class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2, padding=0)

        self.block = 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),
        )

    def forward(self, x, skip_x):
        x = self.upconv(x)
        # crop skip_x to match x
        diff = skip_x.size()[3] - x.size()[3]
        skip_x = skip_x[:, :, diff // 2: x.size()[3] + diff // 2, diff // 2: x.size()[3] + diff // 2]
        x = torch.cat((x, skip_x), dim=1)
        return self.block(x)
```

Unet architecture

```python
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        # Encoder
        self.encode1 = EncoderBlock(in_channels, 64)
        self.encode2 = EncoderBlock(64, 128)
        self.encode3 = EncoderBlock(128, 256)
        self.encode4 = EncoderBlock(256, 512)

        self.middle = nn.Sequential(
            nn.Conv2d(512, 1024, kernel_size=3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 1024, kernel_size=3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
        )
```

```python
        # Decoder
        self.decode4 = DecoderBlock(1024, 512)
        self.decode3 = DecoderBlock(512, 256)
        self.decode2 = DecoderBlock(256, 128)
        self.decode1 = DecoderBlock(128, 64)

        self.conv_out = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        encode1, skip1 = self.encode1(x)
        encode2, skip2 = self.encode2(encode1)
        encode3, skip3 = self.encode3(encode2)
        encode4, skip4 = self.encode4(encode3)

        middle = self.middle(encode4)

        decode4 = self.decode4(middle, skip4)
        decode3 = self.decode3(decode4, skip3)
        decode2 = self.decode2(decode3, skip2)
        decode1 = self.decode1(decode2, skip1)

        output = self.conv_out(decode1)
        return output
```

ResNet34_UNet

Basic block & encoder block

```python
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)
        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 = None
        if stride!=1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    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

class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, n_blocks):
        super(EncoderBlock, self).__init__()
        self.blocks = [BasicBlock(in_channels, out_channels, stride=2)]
        for _ in range(n_blocks-1):
            self.blocks.append(BasicBlock(out_channels, out_channels))
        self.blocks = nn.Sequential(*self.blocks)

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

Decoder block

```python
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.block = nn.Sequential(
            nn.Conv2d(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):
        x = torch.cat((x, skip_x), dim=1)
        x = self.upconv(x)
        return self.block(x)
```

ResNet34_UNet architecture

```python
class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super(ResNet34_UNet, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_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)
        )

        self.encode1 = EncoderBlock(64, 64, 3)
        self.encode2 = EncoderBlock(64, 128, 4)
        self.encode3 = EncoderBlock(128, 256, 6)
        self.encode4 = EncoderBlock(256, 512, 3)

        self.middle = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, padding='same'),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )

        self.decode1 = DecoderBlock(256+512, 32)
        self.decode2 = DecoderBlock(32+256, 32)
        self.decode3 = DecoderBlock(32+128, 32)
        self.decode4 = DecoderBlock(32+64, 32)

        self.decode5 = nn.Sequential(
            nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2, padding=0),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2, padding=0),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, out_channels, kernel_size=1)
        )
```

```python
def forward(self, x):
    x = self.conv1(x)

    x, _ = self.encode1(x)
    x, skip1 = self.encode2(x)
    x, skip2 = self.encode3(x)
    x, skip3 = self.encode4(x)

    skip4 = x
    x = self.middle(x)

    x = self.decode1(x, skip4)
    x = self.decode2(x, skip3)
    x = self.decode3(x, skip2)
    x = self.decode4(x, skip1)

    output = self.decode5(x)
    return output
```

C. Anything more you want to mention

For the encoder & decoder architecture, I do padding in the convolution layers (this is not shown in the graph). This will make the input and output have the same size, making it easier to evaluate.

Reference: U-Net paper / ResNet34 UNet paper / Github - unet / UNet+ResNet34 in keras / Medium / Github - ResUnet

Data Preprocessing (20%)

A. How you preprocessed your data?

Transform them with random rotation and flipping

```python
def transform(image, mask, trimap, mode):
    # implement the transform function here
    if mode == "train":
        degree = np.random.randint(0, 30)
        degree -= 15
        image = imutils.rotate(image, degree)
        mask = imutils.rotate(mask, degree)
        trimap = imutils.rotate(trimap, degree)

        flip = np.random.choice([True, False])
        if flip:
            image = cv2.flip(image, 1)
            mask = cv2.flip(mask, 1)
            trimap = cv2.flip(trimap, 1)

    return dict(image=image, mask=mask, trimap=trimap)
```

B. What makes your method unique?
   The rotated and transposed images can be viewed as the image of same label. With data augmentation, the datasize increased significantly, making the robust to small rotations. This will reduce overfitting and make the model more generalized.

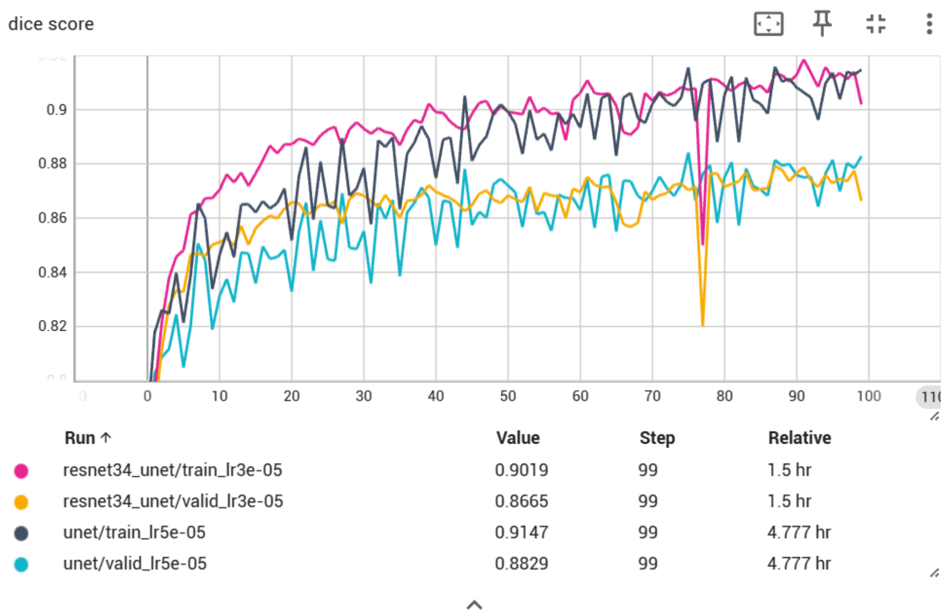
C. Anything more you want to mention
   I tried to normalize the images using torchvision.transform but the result shows that it makes the model converges slower and the accuracy also grow slower. Reference: Image Preprocessing Techniques | Medium python-imutils

Analyze on the experiment results

A. What did you explore during the training process?
   The accuracy grows substantially in the first epoch. Before I start the training, I evaluated it and the average dice score is 0.02. After 1 epoch, the average dice score increases to 0.8. I also tuned several learning rates, and it turns out that the one I'm now using converges the fastest.
   I also tried using BCEWithLogitsLoss for the loss function. Acordding to PyTorch Forums, using BCEWithLogitsLoss is more suitable. I experiment on both losses, and they perform approximately the same. The only difference is that the value the loss of CrossEntropyLoss (about 2e+5) is way higher than BCEWithLogitsLoss (less than 1). Since I have to check whether my implementation of the models are correct, so I choose the one with an obvious drop of loss, which is CE.
B. Found any characteristics of the data?
   The images are quite big. We reshape it to 256*256 in preprocessing, but this will make the new image have a size way smaller than the original one. Maybe we can cut them into half in both width and height if some of them is bigger than 512. This can reduce the error during of interpolation during resize.

C. Anything more you want to mention

Learning curve using dice score



The result is smoother when the background is clean

Execution command

A. The command and parameters for the training process

python train.py -m unet -lr 5e-5

python train.py -m resnet34_unet -lr 3e-5

other arg: --data_path, --epochs, --batch_size

B. The command and parameters for the inference process python inference.py --model_path ../saved_models/DL_Lab3_UNet_110705013_沈昱宏.pth

python inference.py --model_path ../saved_models/DL_Lab3_ResNet34_UNet_110705013_沈昱宏.pth

other arg: --mode(mentioned in implementation), --data_path

Discussion

A. What architecture may bring better results?

ResNet34_UNet modified the UNet encoder, making the encoder able to learn more complex features. In this lab, the task is binary semantic segmentation, so the task is not complex enough to make UNet perform worse. According to the learning curve, they do not show much difference, but I believe that the performance of UNet will drop when the task becomes multi-class segmentation, and changing resnet34 to resnet50 or higher will help learning more complex features.

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

Semantic segmentation is often used in the field of self-driving cars. Strong semantic segmentation models can support real-time inference and help the self-driving car to detect the lines or crossing. Additionally, exploration of interactive segmentation methods (Segment Anything), and multi-modal segmentation(Multimodal Material Segmentation) further advances the capabilities of semantic segmentation across diverse fields.