Deep Learning Lab3

312554054 林辰翰

1. Overview of your lab 3

Lab 3 aims to implement two neural networks to classify the pixels are belonging to foreground or background.

The file “oxford-iiit-pet” in the dataset directory is intended to store all training, validation, and testing data. Initially, we must determine the source of the filenames based on the operating mode. If the mode is “test”, filenames are read from "test.txt". For "train" or "valid" modes, filenames are sourced from "trainval.txt".

Each index in the dataset corresponds to specific image and trimap files, which are loaded and converted into RGB format and then into numpy arrays for further processing.

In my code, I have structured workflow into three main phases: data preprocessing, the training phase, and the validation and testing phase.

2. Implementation Details

(A) In the training process, the code copes with data loading, model training, and validation. Each training epoch is coupled with a validation phase to make sure only the best score of the model is saved.

```
def train(args):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    # device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

    train_loader = load_dataset(data_path=args.data_path, mode='train', batch_size=args.batch_size)
    val_loader = load_dataset(data_path=args.data_path, mode='valid', batch_size=args.batch_size)

    # model = UNet(num_classes=1).to(device)
    model = ResNet34Unet(num_classes=1).to(device)
    # criterion = nn.CrossEntropyLoss()
    criterion = nn.BCEWithLogitsLoss()
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    best_dice = 0.0
    model.train()
```

In the epoch loop, I have implemented Binary Cross-Entropy with Logits Loss in optimization.

```
for epoch in range(args.epochs):
    for sample in train_loader:
        images, masks, trimaps = sample["image"], sample["mask"], sample["trimap"]
        images, masks, trimaps = images.to(device), masks.to(device), trimaps.to(device)

        outputs = model(images)
        loss = criterion(outputs, masks)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    avg_dice = evaluate(model, val_loader, device)
    print(f'Epoch [{epoch+1}/{args.epochs}], Loss: {loss.item():.4f}')

    if avg_dice > best_dice:
        best_dice = avg_dice
        torch.save(model.state_dict(), 'saved_models/resner34_unet(2).pth')
        print(f'New best model saved with avg dice: {best_dice:.4f}')
```

```
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=64, help='batch size')
    parser.add_argument('--learning-rate', '-lr', type=float, default=2e-5, help='learning rate')

    return parser.parse_args()

if __name__ == "__main__":
    args = get_args()
    train(args)
```

(B) **Unet:**

In the contracting path, it includes five double convolution modules, each followed by a max pooling layer to reduce the spatial dimensions of the feature maps while increasing depth. In the expanding path, we use a transposed convolutions for upsampling, followed by feature concatenation, and a double convolution module.

The final convolution layer is a 1 by 1 convolution that transforms the number of feature map channels to the number of target classes.

```
def double_convolution(in_channels, out_channels):
    # The reson why implements padding is to make sure that result size is same as input size.
    conv_op = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
        nn.ReLU(inplace=True)
    )
    return conv_op
```

```
class UNet(nn.Module):
    def __init__(self, num_classes):
        super(UNet, self).__init__()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        # Contracting path.
        # Each convolution is applied twice.
        self.down_convolution_1 = double_convolution(3, 64)
        self.down_convolution_2 = double_convolution(64, 128)
        self.down_convolution_3 = double_convolution(128, 256)
        self.down_convolution_4 = double_convolution(256, 512)
        self.down_convolution_5 = double_convolution(512, 1024)
        # Expanding path.
        self.up_transpose_1 = nn.ConvTranspose2d(
            in_channels=1024, out_channels=512,
            kernel_size=2,
            stride=2)
        # Below, `in_channels` again becomes 1024 as we are concatinating.
        self.up_convolution_1 = double_convolution(1024, 512)
        self.up_transpose_2 = nn.ConvTranspose2d(
            in_channels=512, out_channels=256,
            kernel_size=2,
            stride=2)
        self.up_convolution_2 = double_convolution(512, 256)
        self.up_transpose_3 = nn.ConvTranspose2d(
            in_channels=256, out_channels=128,
            kernel_size=2,
            stride=2)
        self.up_convolution_3 = double_convolution(256, 128)
        self.up_transpose_4 = nn.ConvTranspose2d(
            in_channels=128, out_channels=64,
            kernel_size=2,
            stride=2)
        self.up_convolution_4 = double_convolution(128, 64)
        # output => `out_channels` as per the number of classes.
        self.out = nn.Conv2d(
            in_channels=64, out_channels=1,
            kernel_size=1
        )
```

In the forwarding pass, data are first passed through the contracting path, undergoing downsampling after each convolution until reaching the bottom layer. Then, it passes through the expanding path, where each upsampling step is followed by feature concatenation and a double convolution.

```
def forward(self, x):
    x = x.float() / 255.0
    down_1 = self.down_convolution_1(x)
    down_2 = self.max_pool2d(down_1)
    down_3 = self.down_convolution_2(down_2)
    down_4 = self.max_pool2d(down_3)
    down_5 = self.down_convolution_3(down_4)
    down_6 = self.max_pool2d(down_5)
    down_7 = self.down_convolution_4(down_6)
    down_8 = self.max_pool2d(down_7)
    down_9 = self.down_convolution_5(down_8)

    up_1 = self.up_transpose_1(down_9)
    x = self.up_convolution_1(torch.cat([down_7, up_1], 1))
    up_2 = self.up_transpose_2(x)
    x = self.up_convolution_2(torch.cat([down_5, up_2], 1))
    up_3 = self.up_transpose_3(x)
    x = self.up_convolution_3(torch.cat([down_3, up_3], 1))
    up_4 = self.up_transpose_4(x)
    x = self.up_convolution_4(torch.cat([down_1, up_4], 1))
    out = self.out(x)
    return out
```

ResNet34_Unet:

The ResNet34_Unet combines the deep feature extraction capabilities of ResNet34 with the precise spatial information recovery function of Unet. The BasicBlock is the basic residual block used in ResNet (compared to the last homework, we used Bottleneck block in ResNet50).

```
class BasicBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride, shortcut=None):
        super(BasicBlock, self).__init__()
        self.basic = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, 3, stride, 1,
                      bias=False),  # need to change the stride
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            nn.Conv2d(outchannel, outchannel, 3, 1, 1,
                      bias=False),  # retain the size of feature map
            nn.BatchNorm2d(outchannel),
        )
        self.shortcut = shortcut

    def forward(self, x):
        out = self.basic(x)
        residual = x if self.shortcut is None else self.shortcut(x)  # compute residual
        out += residual
        return nn.ReLU(inplace=True)(out)
```

The “Conv2dReLU” combines a convolution layer with batch

normalization and ReLU activation function in a sequential manner. The convolution layer applies spatial filters to the input, extracting relevant features while maintaining the depth of the feature maps.

```
class Conv2dReLU(nn.Module):
    def __init__(
            self,
            in_channels,
            out_channels,
            kernel_size,
            padding=0,
            stride=1,
    ):
        super(Conv2dReLU, self).__init__()
        self.conv = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size,
            stride=stride,
            padding=padding,
            bias=False
        )
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

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

DecoderBlock begins by upsampling the input feature maps to increase their spatial dimensions, commonly using techniques like nearest neighbor interpolation to recover the resolution lost during downsampling.

d layer takes the flattened output from the final pooling layer (with an input dimension of 25088) and reduces it to 4096. This is followed by another 4096-unit layer and finally a layer that reduces it to the number of classes (num_classes) for the classification task.

Segmentation is designed to convert complex multi-channel feature maps into a final segmentation map that categorizes each pixel into one of the target classes.

```python
class DecoderBlock(nn.Module):
    def __init__(
            self,
            in_channels,
            skip_channels,
            out_channels,
    ):
        super().__init__()
        self.conv1 = Conv2dReLU(
            in_channels + skip_channels,
            out_channels,
            kernel_size=3,
            padding=1,
        )
        self.conv2 = Conv2dReLU(
            out_channels,
            out_channels,
            kernel_size=3,
            padding=1,
        )

    def forward(self, x, skip=None):
        x = F.interpolate(x, scale_factor=2, mode="nearest")
        if skip is not None:
            x = torch.cat([x, skip], dim=1)

        x = self.conv1(x)
        x = self.conv2(x)

        return x


class SegmentationHead(nn.Sequential):
    def __init__(self, in_channels, out_channels, kernel_size=3, upsampling=1):
        conv2d = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, padding=kernel_size // 2)
        upsampling = nn.UpsamplingBilinear2d(scale_factor=upsampling) if upsampling > 1 else nn.Identity()
        super(SegmentationHead, self).__init__(conv2d, upsampling)
```

(C) The structure of the Unet model is little bit different from the original one in paper. In the double convolution layer, we should add padding is because to make sure the result size is same as the input size.

3. Data Preprocessing

(A) Data processing can be divided into three parts: data split reading, data loading and transformation, and the last resizing images and format conversion.

(B) The adapted input sizing and normalization, custom mask processing, and the decoder blocks in the model. The implementation considers the integration of skip connections from the contracting path to the expansive path in the U-Net architecture.

4. Analyze on the experiment results

(A) In the training process, it is initially not possible to calculate the performance score due to the lack of sufficient iterations for the model to

learn distinguishing features effectively. However, once the training commences in earnest, the score typically rises quickly, reaching a peak rapidly. Despite this swift improvement, surpassing a score of 0.87 proves to be challenging. This plateau often indicates that the model has reached its learning capacity under the current configuration and training data. Overcoming this threshold may require adjustments such as tuning hyperparameters, enhancing the training dataset, or modifying the network architecture to better capture the complexities of the data.

(B) Most of the images in the Oxford-IIIT Pet dataset are well-balanced, with each image containing only a single pet subject against various backgrounds. The dataset exhibits remarkably high diversity across multiple aspects. Additionally, the lighting conditions captured in the dataset span a broad spectrum, from well-lit scenes to low-light situations, further enhancing the variety of scenarios represented. This rich diversity in pet species, backgrounds, and lighting conditions provides a comprehensive and challenging testbed for developing robust binary semantic segmentation models capable of generalizing effectively across a multitude of real-world scenarios.

5. Execution command

(A) In the training process, the ultimate model parameters of “Unet”, I’ve set the learning rate as 4e-5, batch_size as 64, and epoch as 100. As for the combination of “ResNet34” and “Unet”, I’ve set the learning rate as 5e-5, batch_size as 100, and epoch as 150(But it has achieved the 0.87 score within 100 epochs).

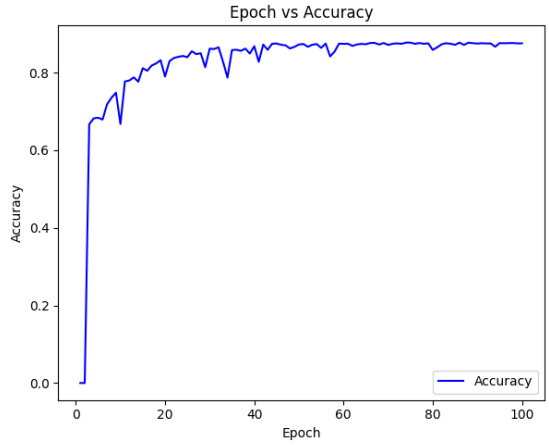
Command for training process: python train.py --epochs (epoch) --batch_size (batch_size) --learning_rate (learning rate)

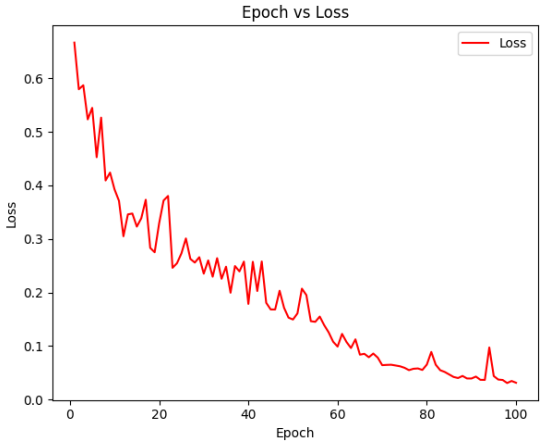
(B) In the inference process, the last scores of Unet and ResNet34_Unet are 0.8731 and 0.8777 respectively.

Command for inference file: python inference.py --model (0: zero for Unet, 1: one for ResNet34_Unet) --model_path ‘../saved_models/unet.pth (or resnet34_unet.pth)’ --batch_size 64(100 for ResNet34)

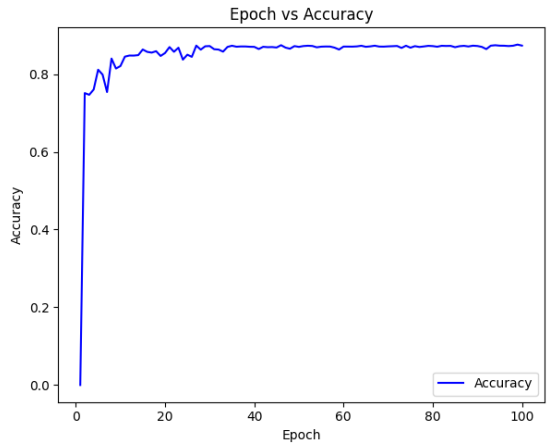
6. Discussion

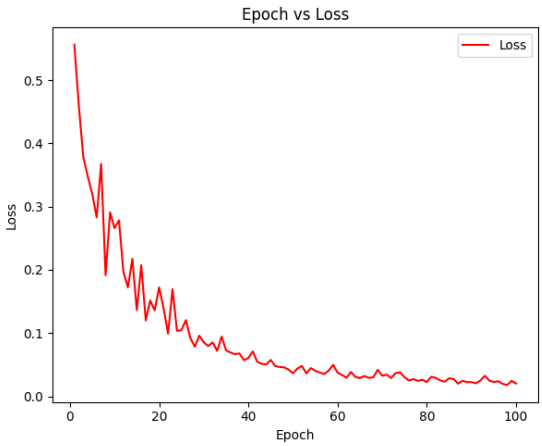
(A) **Unet:**





ResNet34_Unet:





The fusion of ResNet34 and U-Net indeed yields superior performance. ResNet facilitates efficient feature transfer and context propagation across diverse layers, while U-Net excels in precise spatial recovery. This hybrid architecture maximizes the benefits of both networks, leveraging their strengths to enhance overall performance.

(B) Potential topics could be like 1. Real-Time Segmentation (applications such as autonomous driving and video surveillance). 2. Cross-Modal Segmentation (Exploring models that can utilize information from various modalities (such as visual, auditory, and textual cues) to perform segmentation) 3. Defense on adversarial attacks (robust models resistant to such attacks is crucial for security-sensitive applications)

7. Reference

1. https://www.researchgate.net/publication/359463249_Deep_learning-based_pelvic_levator_hiatus_segmentation_from_ultrasound_images

2. https://oecd.ai/en/catalogue/metrics/dice-score

3. https://medium.com/the-research-nest/calculating-the-s%C3%B8rensen-dice-coefficient-a-simplified-guide-3b59a1829a82

4. https://pyimagesearch.com/2021/11/08/u-net-training-image-segmentation-models-in-pytorch/

5. https://blog.csdn.net/m0_37477175/article/details/83861678

6. https://github.com/sidml/Image-Segmentation-Challenge-Kaggle/blob/master/.ipynb_checkpoints/unet-resnet34-in-keras-checkpoint.ipynb

7. https://blog.csdn.net/a_piece_of_ppx/article/details/125960098

8. https://blog.csdn.net/weixin_54255111/article/details/125625991