Lab 3 Report

Deep Learning

Name: Kai-Jie Lin

Student ID 110652019

April 13, 2024

1 Overview

In this lab, I implemented UNet and ResNet34_Unet architecture with Pytorch. I use the models to do binary semantic segmentation on the Oxford-IIIT Pet dataset. Futhermore, I designed my own dataloader and data preprocessing technique to train the model. Finally, I evaluated the model with the test dataset, calclated the dice score and inferencing the image.

2 Implementation Details

2.1 Details of training, evaluating, inferencing

Training: I trained the model with the training dataset and the model is trained with the cross-entropy loss function and the Adam optimizer. I also used the learning rate scheduler to adjust the learning rate during training. For each epoch, I calculated the dice score that testing on the validation dataset.

```python
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)
critirion = nn.CrossEntropyLoss()
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, mask = sample["image"].to(device), sample["mask"].to(device)
        pred_mask = model(image)
        pred_mask = pred_mask.flatten(start_dim=1)
        loss = critirion(pred_mask, mask)
        train_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)
    model.eval()
    with torch.no_grad():
        sum = 0
        for sample in tqdm(valid_loader):
            image, mask = sample["image"].to(device), sample["mask"].to(device)
            pred_mask = model(image)
            sum += dice_score(pred_mask, mask)
        dice = sum / len(valid_loader)
        print(f"Epoch {args.load_model_epoch + i + 1}, Loss: {train_loss:.4f}, Dice Score: {dice:.4f}, LR: {scheduler.get_last_lr()
    if args.lr_scheduler:
        scheduler.step()
```

Evaluating: I evaluated the model with the test dataset and calculated the dice score.

```python
def evaluate(model, data):
    # implement the evaluation function here
    model.eval()
    with torch.no_grad():
        sum = 0
        for sample in tqdm(data):
            image, mask = sample["image"].to(device), sample["mask"].to(device)
            pred_mask = model(image)
            sum += dice_score(pred_mask, mask)
    return sum / len(data)
```

Inferencing: I inferenced the image with the trained model.

```python
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('outputs_imgs', 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')
```

```python
def preprocess_data(img_path):
    data = Image.open(img_path).convert("RGB")
    data = np.array(data.resize((256, 256), Image.BILINEAR))
    data = torch.tensor(data, dtype=torch.float32)
    data /= 255
    data = torch.permute(data, (2, 0, 1))
    return data

def to_img(data, mask):
    data = data.squeeze(0).cpu().numpy().transpose((1, 2, 0))
    mask = np.stack((mask,)*3, axis=-1)
    data = data * 255
    mask = mask * 255
    mask = mask.astype('uint8')
    mask = Image.fromarray(mask)
    data = Image.fromarray(data.astype('uint8'))
    return Image.blend(data, mask, alpha=0.5)
```

2.2 UNet & ResNet34_UNet

UNet:

Encoder:

```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, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )
        self.pool = nn.MaxPool2d(2, 2)

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

Decoder:

```python
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2, padding=0)
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x, skip):
        x = self.up(x)
        _, _, h, w = x.size()
        _, _, skip_h, skip_w = skip.size()
        diff_h = skip_h - h
        diff_w = skip_w - w
        skip = skip[:, :, diff_h // 2:diff_h // 2 + h, diff_w // 2:diff_w // 2 + w]

        x = torch.cat([skip, x], dim=1)
        x = self.block(x)

        return x
```

Unet:

```python
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()
        self.encoder1 = EncoderBlock(in_channels, 64)
        self.encoder2 = EncoderBlock(64, 128)
        self.encoder3 = EncoderBlock(128, 256)
        self.encoder4 = EncoderBlock(256, 512)
        self.center = nn.Sequential(
            nn.Conv2d(512, 1024, 3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 1024, 3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
        )
        self.decoder4 = DecoderBlock(1024, 512)
        self.decoder3 = DecoderBlock(512, 256)
        self.decoder2 = DecoderBlock(256, 128)
        self.decoder1 = DecoderBlock(128, 64)
        self.decoder_final = nn.Conv2d(64, out_channels, 1)

    def forward(self, x):
        enc1, skip1 = self.encoder1(x)
        enc2, skip2 = self.encoder2(enc1)
        enc3, skip3 = self.encoder3(enc2)
        enc4, skip4 = self.encoder4(enc3)

        center = self.center(enc4)

        dec4 = self.decoder4(center, skip4)
        dec3 = self.decoder3(dec4, skip3)
        dec2 = self.decoder2(dec3, skip2)
        dec1 = self.decoder1(dec2, skip1)

        out = self.decoder_final(dec1)

        return out
```

ResNet34_UNet:
ResNetBlock:

```python
class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super(ResNetBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
        )
        self.down = None
        if stride!=1 or in_channels != out_channels:
            self.down = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x
        x = self.block(x)
        if self.down is not None:
            identity = self.down(identity)
        x += identity
        x = self.relu(x)
        return x
```

Encoder:

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

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

Decoder:

```python
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.block = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x, skip):
        x = torch.cat([skip, x], dim=1)
        x = self.block(self.up(x))
        return x
```

ResNet34_Unet:

```python
class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNet34_UNet, self).__init__()
        self.encoder1 = 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.encoder2 = EncoderBlock(64, 64, 3)
        self.encoder3 = EncoderBlock(64, 128, 4)
        self.encoder4 = EncoderBlock(128, 256, 6)
        self.encoder5 = EncoderBlock(256, 512, 3)

        self.center = nn.Sequential(
            nn.Conv2d(512, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
        )

        self.decoder4 = DecoderBlock(256+512, 32)
        self.decoder3 = DecoderBlock(32+256, 32)
        self.decoder2 = DecoderBlock(32+128, 32)
        self.decoder1 = DecoderBlock(32+64, 32)

        self.output = 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):
    enc1 = self.encoder1(x)
    enc2, _ = self.encoder2(enc1)
    enc3, skip1 = self.encoder3(enc2)
    enc4, skip2 = self.encoder4(enc3)
    enc5, skip3 = self.encoder5(enc4)

    skip4 = enc5
    center = self.center(enc5)

    dec4 = self.decoder4(center, skip4)
    dec3 = self.decoder3(dec4, skip3)
    dec2 = self.decoder2(dec3, skip2)
    dec1 = self.decoder1(dec2, skip1)

    return self.output(dec1)
```

3 Data Preprocessing

3.1 How to preprocess the data?

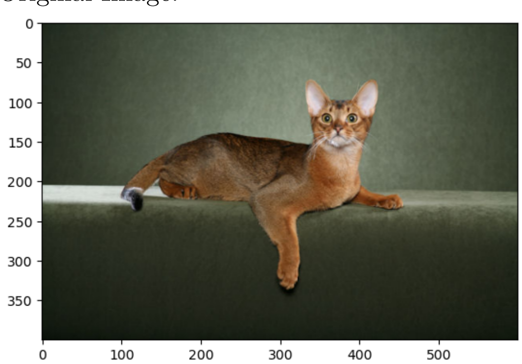
I use some data augmentation techniques to preprocess the data. For example, random vertical or horizontal flip the image and randomly rotate the image.

```
def transform(image, mask, trimap):
    if mode == "train":
        deg = np.random.randint(0, 30)
        deg -= 15
        image = imutils.rotate(image, deg)
        mask = imutils.rotate(mask, deg)
        trimap = imutils.rotate(trimap, deg)

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

        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)
```

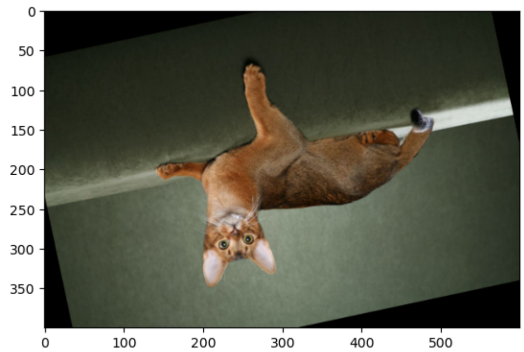
3.2 What makes my method special?

I use the data augmentation technique to increase the diversity of the dataset. This can help the model to learn more features and improve the accuracy.

Original Image:



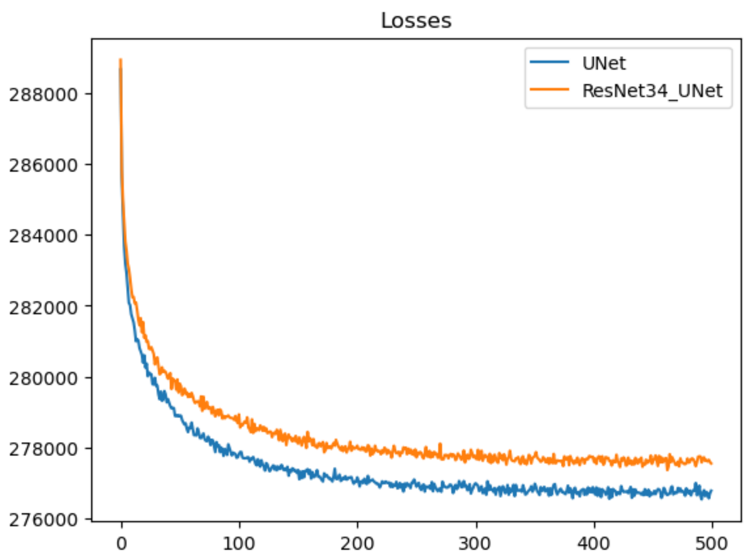
Augmented Image:



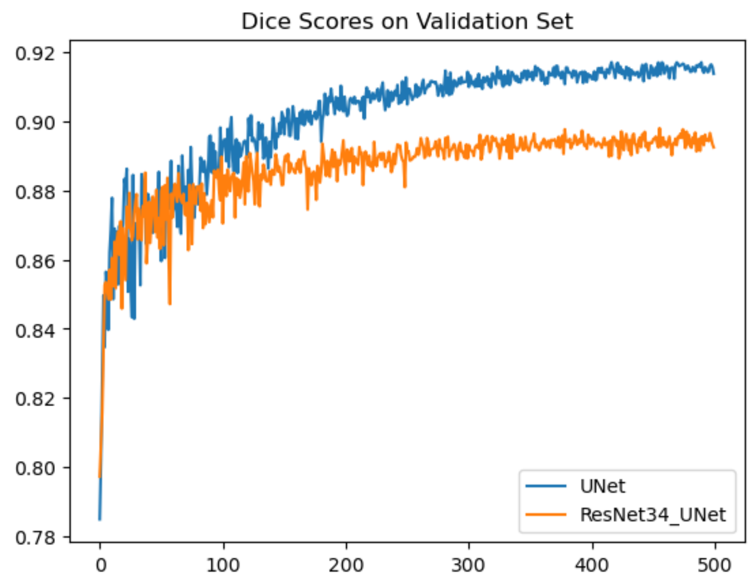
4 Experimental Results

4.1 What did you explore during the training process?

Training loss comparison:



Dice score on validation set:



4.2 Found any characteristics of the data?

A lot of images contain big part of animals and the background is simple. If the model can learn the features of the animals, the model can get a high accuracy.

5 Execution command

5.1 The command and parameters for the training process

Command: python src/train.py –model U –lr 1e-4 –epochs 500
Training model: UNet, learning rate: 1e-4, epochs: 500, Batch size: 8,
Exponential lr Decay $\gamma = 0.99$
Command: python src/train.py –model R –lr 3e-5 –epochs 500
Training model: ResNet34_Unet, learning rate: 3e-5, epochs: 500, Batch size: 8,
Exponential lr Decay $\gamma = 0.99$

5.2 The command and parameters for the inference process

Command: python src/inference.py –model U –batch_size 1 –load_model_epoch 500
Inference model: UNet, batch size: 1, load model epoch: 500
Command: python src/inference.py –model R –batch_size 1 –load_model_epoch 500
Inference model: ResNet34_Unet, batch size: 1, load model epoch: 500

Figure 1: UNet

Figure 2: ResNet34_Unet

6 Discussion

6.1 What architecture may bring better results?

From the inferencing result, we can see that the UNet can get a better result than ResNet34_Unet under same training epochs. The UNet can get a better dice score and the segmentation result is more accurate.

```
Using UNet model
Loading model from saved_models/U/U_epoch_500.pth
Evaluating model
100%|██████████| 3669/3669 [00:52<00:00, 69.69it/s]
Dice Score: 0.9237
```

```
Using ResNet34_UNet model
Loading model from saved_models/R/R_epoch_500.pth
Evaluating model
100%|██████████| 3669/3669 [00:41<00:00, 88.75it/s]
Dice Score: 0.9048
```

Inference Result: See Figure 1 and Figure 2.

6.2 What are the potential research topics in this task?

This dataset is relatively simple and the model can get a high accuracy. What if there are more complex background or the animals are not in the center of the image? We can try to use the more complex model or use the more complex data augmentation technique to improve the accuracy.