Binary Semantic Segmentation
Lab Report # 3

By
312581020
許瀚丰

Deep Learning
Spring 2024
Date Submitted: April 10, 2024

Contents

Listings

1 Overview of your lab 3

1.1 Problem Statement

本次實驗是實作兩個不同的Binary Semantic Segmentation模型，分別為UNet與ResNet34 + UNet，並將其訓練於Oxford-IIIT Pet Dataset上，並期望模型預測出的結果會是一個只含有0與1的矩陣，1代表的是模型預測此位置為物件，0則代表是背景。本次任務的目標是透過計算預測答案與實際答案的Dice Score的方式作為指標，希望計算結果結果越高越好。

2 Implementation Details

2.1 Details of your training,evaluating,inferencing code

2.1.1 Training

Training的過程其實與不同任務的訓練無異，皆是先宣告模型（分別為UNet與ResNet34_UNet），希望使用的Optimizer（在此我使用的是Adam）與Loss Function（Binary Cross Entropy Loss + Dice Loss），與一些額外用來記錄模型訓練情況的工具（Tensorboard，tqdm等）。而在訓練中，就是依照Forward，Calculate Loss，Backward，Update的順序進行，之後分別計算Training與Evaluate的Loss與Dice Score，，並只儲存Evaluate的Dice Score最高時的Model Weights，如程式碼1所示。

```python
def train(args):
    train_data = load_dataset(args.data_path, mode="train")
    train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True)
    val_data = load_dataset(args.data_path, mode="valid")
    val_loader = DataLoader(val_data, batch_size=1, shuffle=False)
    if args.model == "unet":
        model = UNet(3, 1).to(args.device)
    else:
        model = ResNet34_UNet(3, 1).to(args.device)
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
    criterion = nn.BCELoss()
    writer = SummaryWriter(f"runs/{args.model}/")
    best_dice_score = 0.88
```

```python

    for epoch in range(args.epochs):
        train_loss = []
        train_dice_score = []
        model.train()
        progress = tqdm(enumerate(train_loader))
        for i, batch in progress:
            image = batch["image"].to(args.device)
            mask = batch["mask"].to(args.device)
            pred_mask = model(image)
            loss = criterion(pred_mask, mask) + dice_loss(pred_mask, mask)
            train_loss.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            with torch.no_grad():
                train_dice_score.append(dice_score(pred_mask, mask).item())
            progress.set_description((f"Epoch: {epoch + 1}/{args.epochs}, iter: {i + 1}/{len(train_loader)}, Loss: {np.mean(train_loss):.4f}, Dice Score: {np.mean(train_dice_score):.4f}"))
        val_loss, val_dice_score = evaluate(model, val_loader, args.device)

        writer.add_scalars(f"Loss", {"train": np.mean(train_loss), "valid": np.mean(val_loss)}, epoch)
        writer.add_scalars(f"Dice Score", {"train": np.mean(train_dice_score), "valid": np.mean(val_dice_score)}, epoch)
        if np.mean(val_dice_score) > best_dice_score:
            best_dice_score = np.mean(val_dice_score)
            torch.save(model, f"../saved_models/{args.model}.pth")
```

Listing 1: Training

2.1.2 Evaluating

Evaluating的部分基本上與Training Process差異不大，但仍有幾點需要特別設定，首先是需要將Model設定為Eval Mode 來關閉Model的BatchNorm，並且是由於模型不需要做更新，因此我們在過程中也使用no_grad來節省記憶體的使用，如程式碼2所示。

```python
def evaluate(net, data, device):
    val_loss = []
    val_dice_score = []
```

```python
    criterion = nn.BCELoss()
    with torch.no_grad():
        net.eval()
        for batch in data:
            image = batch["image"].to(device)
            mask = batch["mask"].to(device)
            pred_mask = net(image)
            val_loss.append(criterion(pred_mask, mask).item() + dice_loss(pred_mask, mask).item())
            val_dice_score.append(dice_score(pred_mask, mask).item())
        print(f"val losses: {np.mean(val_loss)}, val dice score: {np.mean(val_dice_score)}")
    return val_loss, val_dice_score
```

Listing 2: Evaluating

2.1.3 Inferencing

在Inferecning中，首先我們需要將Model給Load進來，之後便與Evaluating類似，但不需要再計算Loss，只需要計算Dice Score即可。

```python
def inference(args):
    model = torch.load("../saved_models/{args.model}.pth")
    model.eval()
    model.to(args.device)
    data = load_dataset(args.data_path, mode="test")
    dataloader = torch.utils.data.DataLoader(data, batch_size=args.batch_size, shuffle=False)
    dice_scores = []
    for i, batch in tqdm(enumerate(dataloader)):
        image = batch["image"].to(args.device)
        mask = batch["mask"].to(args.device)
        pred_mask = model(image)
        dice = dice_score(pred_mask, mask)
        dice_scores.append(dice.item())
    print(f"Mean Dice Score: {np.mean(dice_scores)}")
```

Listing 3: Inferencing

2.2 Details of your model (UNet ResNet34_UNet)

2.2.1 UNet

在UNet的架構中，每一個Block都是由兩個Conv組合而成，因此首先我先定義了一個由兩個Conv的架構所組合的DoubleConv，如程式碼4所示。

```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

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

Listing 4: DoubleConv

UNet的架構非常單純，就是由四個Down block與四個Up block所組合而成，而中間還有一個bottleneck用來做升維，並將所有Up Blcok的輸出與對應的Up Blcok的輸入做concatenate，最後在輸出結果時在透過一個Conv與一個sigmoid來將其變為一個與輸入影像大小相同（只有H * W），但介於0到1的矩陣，如程式碼5所示。

```
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels, features=[64, 128, 256, 512]):
        super(UNet, self).__init__()
        self.downs = nn.ModuleList()
        self.ups = nn.ModuleList()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        for feature in features:
            self.downs.append(DoubleConv(in_channels, feature))
            in_channels = feature
        self.bottleneck = DoubleConv(features[-1], features[-1]*2)
        self.last = nn.Sequential(
            nn.Conv2d(features[0], out_channels, kernel_size=1),
            nn.Sigmoid()
        )
```

```
        for feature in reversed(features):
            self.ups.append(nn.ConvTranspose2d(feature*2, feature, kernel_size=2,
    stride=2))
            self.ups.append(DoubleConv(feature*2, feature))

    def forward(self, x):
        skips = []
        for down in self.downs:
            x = down(x)
            skips.append(x)
            x = self.pool(x)
        x = self.bottleneck(x)
        for up in self.ups:
            if isinstance(up, nn.ConvTranspose2d):
                x = up(x)
                x = torch.cat((x, skips.pop()), dim=1)
            else:
                x = up(x)
        x = self.last(x)
        return x
```

Listing 5: UNet

2.2.2 ResNet34_UNet

在ResBet34_UNet中，首先ResNet34就是由ConvBlcok與ResidualBlock所組合而成，ConvBlcok的其實就是Conv+batchNorm+ReLU，比較需要注意的是由於在ResNet34中可能會有需要Downsampling的部分，因此需要特別設定Conv的stride為何。而在ResidualBlock中，由於每個blcok的輸入是由上個block而來，因此需要特別注意在做Dowmsample時，skip connection時維度會不相同，因此就有前半部shortcut的部分，若需要Downsample時就需要一個額外的Conv來做降維。而其餘的部分就是一般的Residual的作法，輸出的結果就是原始輸入（可能有經過shortcut）再加上經過兩個ConvBlock的結果，如程式碼6所示。

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down = False, **kwargs):
        super(ConvBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size= 3, padding= 1,
                              stride= 2 if down else 1, padding_mode="reflect", **
    kwargs)
```

```
        self.bn = nn.BatchNorm2d(out_channels)
        self.act = nn.ReLU(inplace=True)

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

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down = False):
        super(ResidualBlock, self).__init__()
        if down:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.shortcut = nn.Identity()
        self.block = nn.Sequential(
            ConvBlock(in_channels, out_channels, down=down),
            ConvBlock(out_channels, out_channels, down=False),
        )
    def forward(self, x):
        return self.shortcut(x) + self.block(x)
```

Listing 6: ResidualBlck

而整體ResNet34_UNet架構的部分可以分為前半的Encoder與後半的Decoder。前半部分就是直接使用ResNet34的架構，並在中間的Conv2至Conv5將輸出結果儲存下來，而在建立時使用build_layer來建立，需要特別注意除了Conv2外其餘的Conv在第一個ResidualBlcok皆會需要設定Down為True來做降維，其餘的部分只需要使用迴圈一一建立即可。至於BottleNeck的部分，我使用了一個ResidualBock來建立。而之後Decoder的部分與將結果轉為與輸入影像大小相同的矩陣的部分與UNet後半部分完全相同，如程式碼7所示。

```
class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels, out_channels, features=[64, 128, 256, 512],
    num_block=[3, 4, 6, 3]):
        super(ResNet34_UNet, self).__init__()
        self.init = 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.downs = nn.ModuleList()
        self.downs.append(self.build_layer(features[0], features[0], num_block[0], False))
        self.downs.append(self.build_layer(features[0], features[1], num_block[0], True))
        self.downs.append(self.build_layer(features[1], features[2], num_block[0], True))
        self.downs.append(self.build_layer(features[2], features[3], num_block[0], True))

        self.bottleneck = ResidualBlock(features[-1], features[-1] * 2, True)

        self.ups = nn.ModuleList()
        for feature in reversed(features):
            self.ups.append(nn.ConvTranspose2d(feature*2, feature, kernel_size=2, stride=2))
            self.ups.append(DoubleConv(feature*2, feature))

        self.last = nn.Sequential(
            nn.ConvTranspose2d(features[0], features[0], kernel_size=2, stride=2),
            nn.BatchNorm2d(features[0]),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(features[0], features[0], kernel_size=2, stride=2),
            nn.BatchNorm2d(features[0]),
            nn.ReLU(inplace=True),
            nn.Conv2d(features[0], out_channels, kernel_size=1),
            nn.Sigmoid()
        )
    def build_layer(self, in_channels, out_channels, num_block, down):
        layer = [ResidualBlock(in_channels, out_channels, down)]
        for _ in range(1, num_block):
            layer.append(ResidualBlock(out_channels, out_channels, False))
        return nn.Sequential(*layer)

    def forward(self, x):
        x = self.init(x)
        skips = []
```

```python
        for down in self.downs:
            x = down(x)
            skips.append(x)
        x = self.bottleneck(x)
        for up in self.ups:
            if isinstance(up, nn.ConvTranspose2d):
                x = up(x)
                x = torch.cat((x, skips.pop()), dim=1)
            else:
                x = up(x)
        x = self.last(x)
        return x
```

Listing 7: ResNet34_UNet

2.3 Dice Loss

在Loss Function的部分，經過實驗我發現使用BCELoss加上Dice Loss的結果是最好的，Dice Loss的部分實作了一個Dice Loss來作為其中一個Objective，其與Dice Score最大的差別主要是比較注重於畫面Object（class = 1）的部分，且與Dice Score相比並不需要兩個predict class與actual class完全相同，而是直接用相乘的方式，我認為此方式對於在做Backpropagation也會更有幫助，如程式碼8所示。

```python
def dice_loss(pred_mask, gt_mask, eps=1e-8):
    import torch
    intersection = torch.sum(gt_mask * pred_mask) + eps
    union = torch.sum(gt_mask) + torch.sum(pred_mask) + eps
    loss = 1 - (2 * intersection / union)
    return loss
```

Listing 8: Dice Loss

3 Data Preprocessing

3.1 How you preprocessed your data?

在Data Preprocessing的部分，我使用Albumentations這個套件，此套件最主要的優勢是可以同時對Image與Mask做一樣的操作，而我在Training時對每張照片皆做以下

操作，如程式碼9所示：

- Resize成256×256
- 隨機翻轉
- 隨機擷取畫面一部分並Resize成256×256
- 隨機平移與旋轉±30度
- 隨機調整畫面的HSI
- 依照ImageNet 影像的統計結果來對資料做Normalization

```python
def load_dataset(data_path, mode):
    import albumentations as A
    from albumentations.pytorch import ToTensorV2
    train_transform = A.Compose(
        [
            A.Resize(256, 256),
            A.Flip(),
            A.RandomResizedCrop(size=(256, 256), scale=(0.8, 1)),
            A.ShiftScaleRotate(shift_limit=0.2, scale_limit=0.2, rotate_limit=30, p=0.5),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2, p=0.5),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ],
    )
    transform = A.Compose(
        [
            A.Resize(256, 256),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ],
    )
    if mode == "train":
        dataset = OxfordPetDataset(root=data_path, mode="train", transform=train_transform)
    elif mode == "valid":
        dataset = OxfordPetDataset(root=data_path, mode="valid", transform=transform)
```

```python
    elif mode == "test":
        dataset = OxfordPetDataset(root=data_path, mode="test", transform=transform)
    return dataset
```

Listing 9: Data Preprocessing

3.2 What makes your method unique?

其中我認為較為特別的是我加上了ColorJitter與參考了ImageNet所統計出來的mean與std來做Normalization。前者可以在不改變畫面內容物位置的情況下透過調整畫面的HSI來增加資料的多樣性，而後者與直接將資料Normalize成-1到1的方法更能符合實際影像上的結果，期望因此能提升模型的泛化能力。

4 Analyze on the experiment results

4.1 Hyperparameter settings

以下為本次實驗的超參數設定：

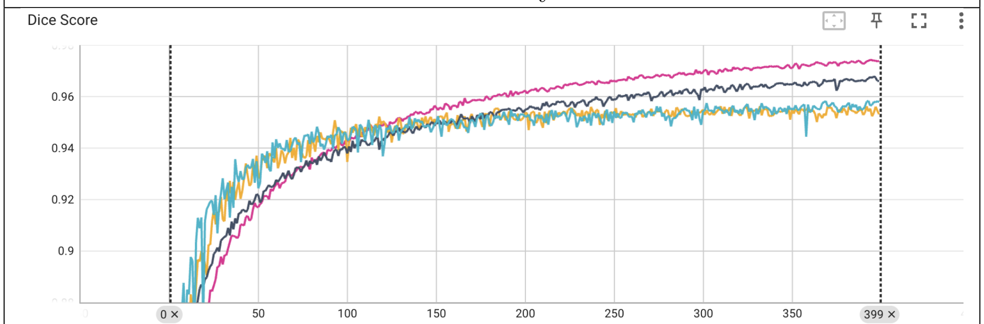
- **Batch Size**: 32
- **Loss Function**: CrossEntropyLoss + Dice Loss
- **Optimizer**: Adam
- **Epoch**: 400
- **Learning Rate**: $1 * 10^{-3}$

4.2 What did you explore during the training process

訓練結果的Comparison figure如表1所示，可以觀察到ResNet34_UNet相比於UNet在前期更容易提升，可見使用ResNet34的架構對於模型是更容易訓練的，然而在訓練後期UNet在Training Dataset的Mean Dice Score是更高的。另外我們也可以觀察到，

兩者訓練到最後Training的Mean Dice Score雖然只相差1%左右，然而在Validation上兩者則相差無幾，可見ResNet34_UNet的架構泛化能力可能是更好的。

Accuracy



Run ↑	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
resnet34_unet_bce_dice/Dice Score_train	0.7307	0.9679	0.7307	0.966	↑0.2353	↑32%	0	399
resnet34_unet_bce_dice/Dice Score_valid	0.7723	0.9583	0.7723	0.9578	↑0.1855	↑24%	0	399
unet_bce_dice/Dice Score_train	0.6692	0.9743	0.6692	0.9739	↑0.3046	↑46%	0	399
unet_bce_dice/Dice Score_valid	0.7322	0.9563	0.7322	0.9531	↑0.2209	↑30%	0	399

Table 1: Comparison figure

4.3 Found any characteristics of the data?

在常見的Image Semantic Segmentation任務上，我們想要擷取的Object往往是畫面中的一小部分，因此物件與背景在畫面中的比例往往是不均衡的，因此在訓練上可能會傾向於使用Focal Loss來加强對於物件的訓練。然而對於此資料集來說，物件在畫面中的比例相較起來是更大的，且再加上Crop等Image Augmentation便能使物體在畫面的比例變大，因此在我的實驗中，使用BCE Loss與Dice Loss的效果是比使用Focal Loss更好的。

4.4 Analyze on testing results

4.4.1 testing results

實驗結果如表2所示，我們可以發現兩者在Testing dataset上的結果差異不大，Mean Dice Score都有約0.95左右，而ResNet34_UNet在Testing略好一些。

UNet Mean Dice Score	ResNet34_UNet Mean Dice Score
inference on UNet Mean Dice Score: 0.9527780944042357	inference on ResNet34_UNet Mean Dice Score: 0.9566519927770574

Table 2: Testing Mean Dice Score

4.4.2 testing analyze

實驗結果如3所示，我發現兩者模型有些各的優缺點，UNet的結果在不規則的情況可能表現較好，如左上，但也因此更容易產生出不規則且破碎的狀況，如左下。而ResNet34_UNet在不規則的情況小表現就不如UNet，如右上，而對於整體較圓滑的物件效果就不錯，如右下。

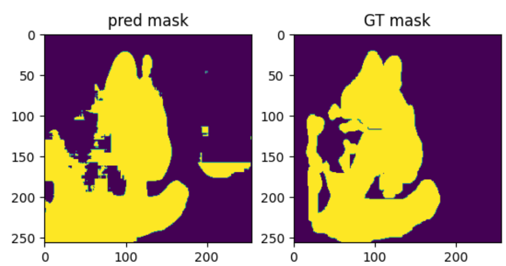
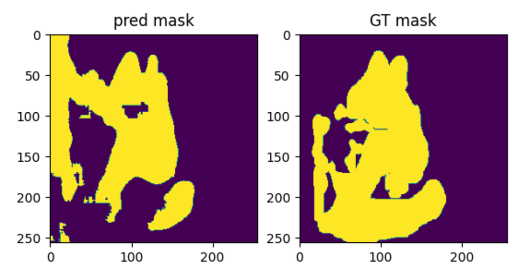
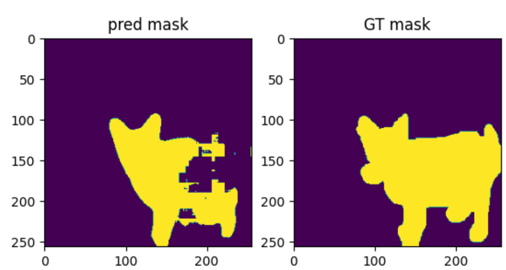
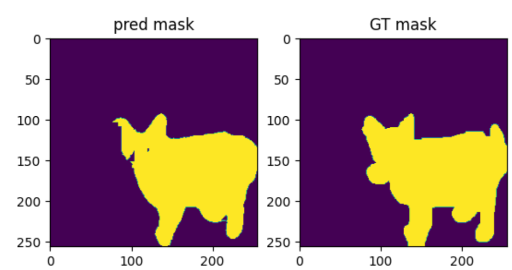
Result on UNet	Result on ResNet_UNet
	
	

Table 3: Testing Accuracy on Pretrained Model

5 Execution command

5.1 The command and parameters for the training process

若需要訓練，有一些簡單的parameters可以設定，如下所示：

- model: unet or resnet34_unet
- device: cpu or cuda
- data_path: "../dataset/oxford-iiit-pet/"(default)
- epoch: 400(default)
- batch_size: 32(default)
- learning-rate: 1e-3(default)

```
python train.py \
  --model unet \
  --device cuda \
  --data_path "../dataset/oxford-iiit-pet/" \
  --epoch 400 \
  --batch_size 32 \
  --learning-rate 1e-3
```

Listing 10: train script

5.2 The command and parameters for the evaluate process

若需要evaluate，基本上與train的參數類似，如下所示：

- model: unet or resnet34_unet
- device: cpu or cuda
- data_path: "../dataset"(default)
- batch_size: 1(default)

```
python inference.py \
  --model unet \
  --device cuda \
  --data_path "../dataset/" \
  --batch_size 32
```

Listing 11: evaluate script

6 Discussion

6.1 What architecture may bring better results?

6.1.1 Ensemble learning

我認為不管是UNet與ResNet34_UNet在不同任務上的表現各有優劣，因此若能夠使用多個模型一起做預測，並透過Ensemble learning的方式進行預測，相信在結果上會更加準確。

6.1.2 Object Detection

對於困難的資料集或者物體與背景比例差異很大的影像，或許我們能夠先採用Object Detection的方式，先將包含物件的範圍從影像中擷取出來，再透過UNet或ResNet34_UNet來做預測，我認為此方式能夠去除畫面中的Outlier，就更加有利於模型的預測與訓練。

6.2 What are the potential research topics in this task?

我認為Image Semantic Segmentation這項Task非常適合當一些影像處理上的Downstream task，例如尋找ROI或者是希望只針對影像的某個物件做操作例如針對某個Object的Style Transfer、Image Manipulation等，都可以先透過找出影相中Object的Mask後，只需要對Mask內的目標做操作即可，以此就可以減少需要操作的影像大小，進而減少不必要的計算量。