Lab3: Binary Semantic Segmentation

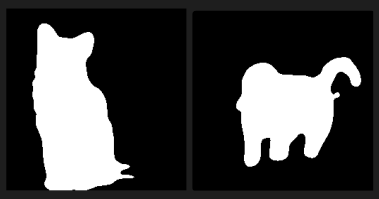
313551055 柯柏旭

Overview

本報告詳述了使用 UNet 和 ResNet34-UNet 進行二元語義分割的實現過程，針對 Oxford Pet 資料集中的寵物圖像進行分割。資料集具有高度多樣性，包括不同品種、顏色、姿勢以及被遮擋的情況，增加了訓練和泛化的難度。

- Oxford Pet example (Input)

- Output example



Implementation Details

Details of your training, evaluating, inferencing code

- Training code

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

    # Data transformations
    training_transform = transforms.Compose([
```

```
        transforms.Lambda(lambda img: Image.fromarray(img)),  # Convert numpy array to PIL image
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(degrees=15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
        transforms.RandomResizedCrop(256, scale=(0.8, 1.0)),
        transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ])

    # Validation and test transformations
    test_transform = transforms.Compose([
        transforms.Lambda(lambda img: Image.fromarray(img)),  # Convert numpy array to PIL image
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ])

    # Load datasets
    train_dataset = SimpleOxfordPetDataset(root=args.data_path, mode='train', transform=training_transform)
    val_dataset = SimpleOxfordPetDataset(root=args.data_path, mode='valid', transform=test_transform)

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False)

    # Initialize model, loss function, and optimizer
    if args.model_type == 'unet':
        model = UNet(in_channels=3, out_channels=1).to(device)
        save_path = '../saved_models/unet_best_model.pth'
```

```
    else:
        model = Res34_UNet(in_channels=3, out_channels=1).to(device)
        save_path = '../saved_models/res34_best_model.pth'

    # Use a combination of BCEWithLogitsLoss and Dice Loss
    bce_loss = nn.BCEWithLogitsLoss()
    def dice_loss(pred, target, smooth=1.):
        pred = torch.sigmoid(pred)
        intersection = (pred * target).sum(dim=(2,3))
        dice = (2. * intersection + smooth) / (pred.sum(dim=(2,3)) +
target.sum(dim=(2,3)) + smooth)
        return 1 - dice.mean()

    def combined_loss(pred, target):
        return bce_loss(pred, target) + dice_loss(pred, target)

    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate,
weight_decay=0.00003)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
'min', patience=2, factor=0.5)

    best_val_loss = float('inf')
    train_losses = []
    val_losses = []

    # Training loop
    for epoch in range(args.epochs):
        model.train()
        train_loss_epoch = 0.0
        for batch in train_loader:
            images = batch['image'].float().to(device)
            masks = batch['mask'].float().to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = combined_loss(outputs, masks)
            loss.backward()
```

```
            optimizer.step()

            train_loss_epoch += loss.item() * images.size(0)

        train_loss_epoch /= len(train_loader.dataset)
        train_losses.append(train_loss_epoch)

        # Validation loop
        model.eval()
        val_loss_epoch = 0.0
        val_dice = 0.0
        with torch.no_grad():
            for batch in val_loader:
                images = batch['image'].float().to(device)
                masks = batch['mask'].float().to(device)

                outputs = model(images)
                loss = combined_loss(outputs, masks)
                val_loss_epoch += loss.item() * images.size(0)

                val_dice += dice_score(outputs, masks).item() *
images.size(0)

        val_loss_epoch /= len(val_loader.dataset)
        val_dice /= len(val_loader.dataset)
        val_losses.append(val_loss_epoch)

        # Print epoch information including learning rate
        current_lr = optimizer.param_groups[0]['lr']
        print(f'Epoch {epoch+1}/{args.epochs}, Train Loss:
{train_loss_epoch:.4f}, Val Loss: {val_loss_epoch:.4f}, Val Dice:
{val_dice:.4f}, Learning Rate: {current_lr:.6f}')

        scheduler.step(val_loss_epoch)

        # Save the model if it has the best validation loss so far
        if val_loss_epoch < best_val_loss:
```

```
            best_val_loss = val_loss_epoch
            torch.save(model.state_dict(), save_path)
            print(f'Saved best model with val loss:
{val_loss_epoch:.4f}')

    # Plot and save the loss graph
    train_loss(train_losses, val_losses,
save_path='unet_loss_plot.png')
```

這個訓練函數的主要步驟包括**數據增強、數據加載、模型初始化、損失函數和優化器設置，然後進行訓練和驗證迴圈，在每個 epoch 之後調整學習率並打印相關信息，最終保存最佳模型並繪製損失圖表**。訓練過程中，模型經過多個 epoch，每個 epoch 都包括訓練和驗證過程。模型學習並更新參數，驗證過程中評估模型性能並根據驗證損失調整學習率，保存最佳模型。每個 epoch 的訓練損失和驗證損失被記錄並用於後續分析。

- Evaluating code & Dice score

```
def dice_score(pred_mask, gt_mask):
    # 添加一個很小的數，防止除以0的錯誤
    smooth = 1e-5

    # 將預測的掩碼通過Sigmoid函數轉換為概率值
    pred_mask = torch.sigmoid(pred_mask)

    # 將概率值轉換為二值掩碼，大於0.5的部分設為1，否則設為0
    pred_mask = (pred_mask > 0.5).float()

    # 計算預測掩碼和真實掩碼的交集（乘積）並在寬和高兩個維度上求和
    intersection = (pred_mask * gt_mask).sum(dim=(2, 3))

    # 分別計算預測掩碼和真實掩碼的總和（面積）並在寬和高兩個維度上求和
    union = pred_mask.sum(dim=(2, 3)) + gt_mask.sum(dim=(2, 3))

    # 計算Dice Score，公式為 (2 * 交集 + 平滑項) / (預測掩碼面積 + 真實掩碼面
積 + 平滑項)
    dice = (2. * intersection + smooth) / (union + smooth)

    # 返回整個批次的平均Dice Score
```

```
    return dice.mean()
```

```
def evaluate(model, dataloader, device):
    dice_scores = []
    with torch.no_grad():
        for batch in dataloader:
            images = batch['image'].float().to(device)
            masks = batch['mask'].float().to(device)
            outputs = model(images)
            preds = outputs > 0.5
            dice = dice_score(preds, masks)
            dice_scores.append(dice.item())
    return np.mean(dice_scores)
```

Evaluating 的步驟如下:

1. **初始化**：創建一個空列表 dice_scores 用於存儲每個 batch 的 Dice 分數。
2. **禁用梯度計算**：使用 torch.no_grad() 停用梯度計算，減少內存使用和提高推理速度。
3. **遍歷數據集**：從 dataloader 中獲取批次數據，將圖像和掩碼加載到設備（GPU 或 CPU）。
4. **模型推理**：對輸入圖像進行模型推理，獲取預測結果。
5. **二值化預測結果**：將模型輸出轉換為二值掩碼。 (大於 0.5 為 1, 反之則為 0, 前景與背景)
6. **計算 Dice 分數**：計算每個批次的 Dice 分數並存儲。
7. **返回平均 Dice 分數**：返回所有批次 Dice 分數的平均值，作為模型在整個數據集上的性能指標。

- Inferencing code

```
def predict(model, dataloader, device):
    preds = []
    with torch.no_grad():
        for batch in dataloader:
            images = batch['image'].float().to(device)  # 確保圖像數據是浮點數類型
            outputs = model(images)
```

```
            preds.append(outputs.cpu().numpy())
    return np.vstack(preds)

def save_predictions(predictions, filenames, output_path):
    os.makedirs(output_path, exist_ok=True)
    for pred, filename in zip(predictions, filenames):
        pred_mask = (pred.squeeze() > 0.5).astype(np.uint8) * 255  # 二值化
        pred_image = Image.fromarray(pred_mask)
        pred_image.save(os.path.join(output_path, f"{filename}.png"))
```

predict 函數

這個函數對給定數據集進行推理，返回預測結果。

save_predictions 函數

這個函數將預測結果保存到指定路徑。

總而言之，inference code 通過 predict 函數進行模型推理，並使用 save_predictions 函數將預測結果保存為圖像文件。

Details of your model (UNet & ResNet34_UNet)

UNet

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

        # Contracting path (Encoder)
        self.enc1 = self.conv_block(in_channels, 64)
        self.enc2 = self.conv_block(64, 128)
        self.enc3 = self.conv_block(128, 256)
        self.enc4 = self.conv_block(256, 512)
        self.enc5 = self.conv_block(512, 1024)

        # Expansive path (Decoder)
        self.upconv4 = nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2)
        self.dec4 = self.conv_block(1024, 512)
```

```
        self.upconv3 = nn.ConvTranspose2d(512, 256, kernel_size=2,
stride=2)
        self.dec3 = self.conv_block(512, 256)
        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2,
stride=2)
        self.dec2 = self.conv_block(256, 128)
        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2,
stride=2)
        self.dec1 = self.conv_block(128, 64)

        # Final layer
        self.conv_last = nn.Conv2d(64, out_channels, kernel_size=1)

    def conv_block(self, in_channels, out_channels):
        block = 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 block

    def forward(self, x):
        # Encoder
        enc1 = self.enc1(x)
        enc2 = self.enc2(self.down(enc1))
        enc3 = self.enc3(self.down(enc2))
        enc4 = self.enc4(self.down(enc3))
        enc5 = self.enc5(self.down(enc4))

        # Decoder
        dec4 = self.upconv4(enc5)
        dec4 = torch.cat((dec4, enc4), dim=1)
        dec4 = self.dec4(dec4)
```

```
        dec3 = self.upconv3(dec4)
        dec3 = torch.cat((dec3, enc3), dim=1)
        dec3 = self.dec3(dec3)

        dec2 = self.upconv2(dec3)
        dec2 = torch.cat((dec2, enc2), dim=1)
        dec2 = self.dec2(dec2)

        dec1 = self.upconv1(dec2)
        dec1 = torch.cat((dec1, enc1), dim=1)
        dec1 = self.dec1(dec1)

        return self.conv_last(dec1)

    def down(self, x):
        return F.max_pool2d(x, 2)
```

1. Encoder (Contracting path, 收縮路徑)

- 由五個卷積塊組成，每個卷積塊包含兩個卷積層和 ReLU 激活函數。
- 每個卷積塊之間使用最大池化層來減小特徵圖的尺寸。

2. Decoder (Expansive path)

- 由四個反卷積層（上采樣）和四個卷積塊組成。
- 每個反卷積層之後，與對應層的編碼器輸出進行拼接（跳躍連接），並經過卷積塊處理。

3. Final layer

- 使用一個卷積層將解碼器的輸出轉換為最終的分割結果。

ResNet34_UNet

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1,
downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=stride, padding=1)
        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)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    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 ResNet34Encoder(nn.Module):
    def __init__(self, in_channels):
        super(ResNet34Encoder, self).__init__()
        self.in_channels = in_channels

        self.initial = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2,
padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
```

```
        self.layer1 = self.make_layer(64, 64, 3)
        self.layer2 = self.make_layer(64, 128, 4, stride=2)
        self.layer3 = self.make_layer(128, 256, 6, stride=2)
        self.layer4 = self.make_layer(256, 512, 3, stride=2)

    def make_layer(self, in_channels, out_channels, blocks,
stride=1):
        downsample = None
        if stride != 1 or in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride),
                nn.BatchNorm2d(out_channels),
            )

        layers = []
        layers.append(BasicBlock(in_channels, out_channels, stride,
downsample))
        for _ in range(1, blocks):
            layers.append(BasicBlock(out_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x1 = self.initial(x)  # [batch, 64, H/4, W/4]
        x2 = self.layer1(x1)  # [batch, 64, H/4, W/4]
        x3 = self.layer2(x2)  # [batch, 128, H/8, W/8]
        x4 = self.layer3(x3)  # [batch, 256, H/16, W/16]
        x5 = self.layer4(x4)  # [batch, 512, H/32, W/32]
        return x1, x2, x3, x4, x5

class Res34_UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Res34_UNet, self).__init__()

        # ResNet34 Encoder
        self.encoder = ResNet34Encoder(in_channels)
```

```
        # Expansive path (Decoder)
        self.upconv4 = nn.ConvTranspose2d(512, 256, kernel_size=2,
stride=2)
        self.dec4 = self.conv_block(512, 256)
        self.upconv3 = nn.ConvTranspose2d(256, 128, kernel_size=2,
stride=2)
        self.dec3 = self.conv_block(256, 128)
        self.upconv2 = nn.ConvTranspose2d(128, 64, kernel_size=2,
stride=2)
        self.dec2 = self.conv_block(128, 64)
        self.upconv1 = nn.ConvTranspose2d(64, 64, kernel_size=2,
stride=2)
        self.dec1 = self.conv_block(128, 64)

        # Final layer
        self.conv_last = nn.Conv2d(64, out_channels, kernel_size=1)

        # Additional upsampling to match the input size
        self.final_upsample = nn.Upsample(scale_factor=4,
mode='bilinear', align_corners=False)

    def conv_block(self, in_channels, out_channels):
        block = 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 block

    def forward(self, x):
        # Encoder
        enc1, enc2, enc3, enc4, enc5 = self.encoder(x)
```

```
        # print(f'enc1: {enc1.size()}')
        # print(f'enc2: {enc2.size()}')
        # print(f'enc3: {enc3.size()}')
        # print(f'enc4: {enc4.size()}')
        # print(f'enc5: {enc5.size()}')

        # Decoder
        dec4 = self.upconv4(enc5)
        dec4 = self.center_crop_and_concat(dec4, enc4)
        dec4 = self.dec4(dec4)

        dec3 = self.upconv3(dec4)
        dec3 = self.center_crop_and_concat(dec3, enc3)
        dec3 = self.dec3(dec3)

        dec2 = self.upconv2(dec3)
        dec2 = self.center_crop_and_concat(dec2, enc2)
        dec2 = self.dec2(dec2)

        dec1 = self.upconv1(dec2)
        dec1 = self.center_crop_and_concat(dec1, enc1)
        dec1 = self.dec1(dec1)

        final_output = self.conv_last(dec1)

        # Match the output size to the input size
        final_output = F.interpolate(final_output, size=x.size()[2:],
mode='bilinear', align_corners=False)

        return final_output

    def center_crop_and_concat(self, upsampled, bypass):
        _, _, H, W = upsampled.size()
        _, _, H_b, W_b = bypass.size()

        # Resize bypass if necessary
        if H_b != H or W_b != W:
```

```
            bypass = F.interpolate(bypass, size=(H, W),
mode='bilinear', align_corners=False)

        return torch.cat((upsampled, bypass), dim=1)
```

主要組成部分包括 BasicBlock、ResNet34 編碼器和 Res34_UNet 模型。

BasicBlock

BasicBlock 是 ResNet 中的基本構建塊，包括兩個卷積層，每個卷積層後面跟著一個批量歸一化層和 ReLU 激活函數。

ResNet34Encoder

ResNet34Encoder 是 ResNet34 的編碼器部分，由初始卷積層和四個由 BasicBlock 構成的層組成。

Res34_UNet

Res34_UNet 是基於 ResNet34 編碼器和 UNet 解碼器的模型。基本上就是把上面 UNet 的 encoder 部分替換成 ResNet34，但這樣在 skip connection 的時候會有一些尺寸不合的問題因此需要使用該函數: **center_crop_and_concat** 將上采樣的特徵圖與對應的編碼器特徵圖拼接。

Anything more you want to mention

值得一提的是，因為單純的 UNet 難以達到 90% 的 acc，因此我在 training code 做了調整:

1. 防止 overfitting ，在優化器上面添加了 L2 norm 的 weight decay，因為我發現 UNet 比較容易 overfit。
2. 我 combine 了 **Binary Cross Entropy Loss** 和 **Dice Loss**，去當作損失函數，其中的 Dice Loss 為 1 - Dice score。BCE 損失在每個像素級別上進行計算，這意味著它對所有像素都有貢獻，包括前景和背景。Dice 損失專注於前景區域的重疊，更加關注分割的整體形狀和連續性。結合損失同時考慮了像素級別的準確性和區域重疊，這使得模型能夠更加精確地學習分割邊界和形狀。

Data Preprocessing

How you preprocessed your data?

首先是將資料分成 `train` 、 `test` 和 `valid` ，其中 `test` 已經有 `test.txt` 分好了，而剩餘兩者則按照 9 : 1 (train : valid)去分。

接著處理細節 `get_item` 的部分，其中每個 sample 是由 `image`、`trimap`、`mask` 組成，`trimap` 為三元圖，`mask` 為處理過後的 `trimap`，為二元圖，即這次作業的 gt。

然後，`SimpleOxfordPetDataset` 這個函數會繼承 `OxfordPetDataset`，並且將所有圖片 resize 成 256 * 256，還有從從 HWC（高度，高度，通道）轉換為 CHW（通道，高度，寬度），以符合 PyTorch 模型輸入的要求。

最後，分別針對訓練數據和驗證/測試數據集，做資料的 `transformation`，詳細內容在下一個 section。

What makes your method unique?

在上一個 section，主要提到的內容大多為助教已經幫我們實作好的，至於在 data preprocessing 這一塊，自己應用與實作的方法為 data transformation，內容如下。

- Training transform

```
training_transform = transforms.Compose([
    transforms.Lambda(lambda img: Image.fromarray(img)),  # 將 numpy 數組轉換為 PIL 圖像
    transforms.RandomHorizontalFlip(),  # 隨機水平翻轉
    transforms.RandomRotation(degrees=20),  # 隨機旋轉角度在 -20 到 20 度之間
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),  # 隨機調整亮度、對比度和飽和度
    transforms.RandomResizedCrop(256, scale=(0.8, 1.0)),  # 隨機裁剪並調整大小至 256x256，裁剪控制在80%~100%
    transforms.ToTensor(),  # 將 PIL 圖像轉換為張量
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # 標準化，使像素值在 [-1, 1] 範圍內
])
```

- Validating and testing transform

```
test_transform = transforms.Compose([
    transforms.Lambda(lambda img: Image.fromarray(img)),  # 將 numpy 數組轉換為 PIL 圖像
    transforms.ToTensor(),  # 將 PIL 圖像轉換為張量
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # 標準化，使像素值在 [-1, 1] 範圍內
])
```

除了以上 4 個 data augmenatation 外， `ToTensor` 和 `Normalize` 是PyTorch 圖像數據預處理中的常用操作，能夠確保數據以合適的格式和範圍輸入到模型中，提高模型的訓練效率和性能。其中 `ToTensor` 會將圖像的像素值從 [0, 255] 範圍內的整數轉換為 [0, 1] 範圍內的浮點數，這樣更符合深度學習模型的輸入要求。 `Normalize` 標準化數據可以加快梯度下降收斂速度，並有助於模型更快、更好地學習數據特徵。

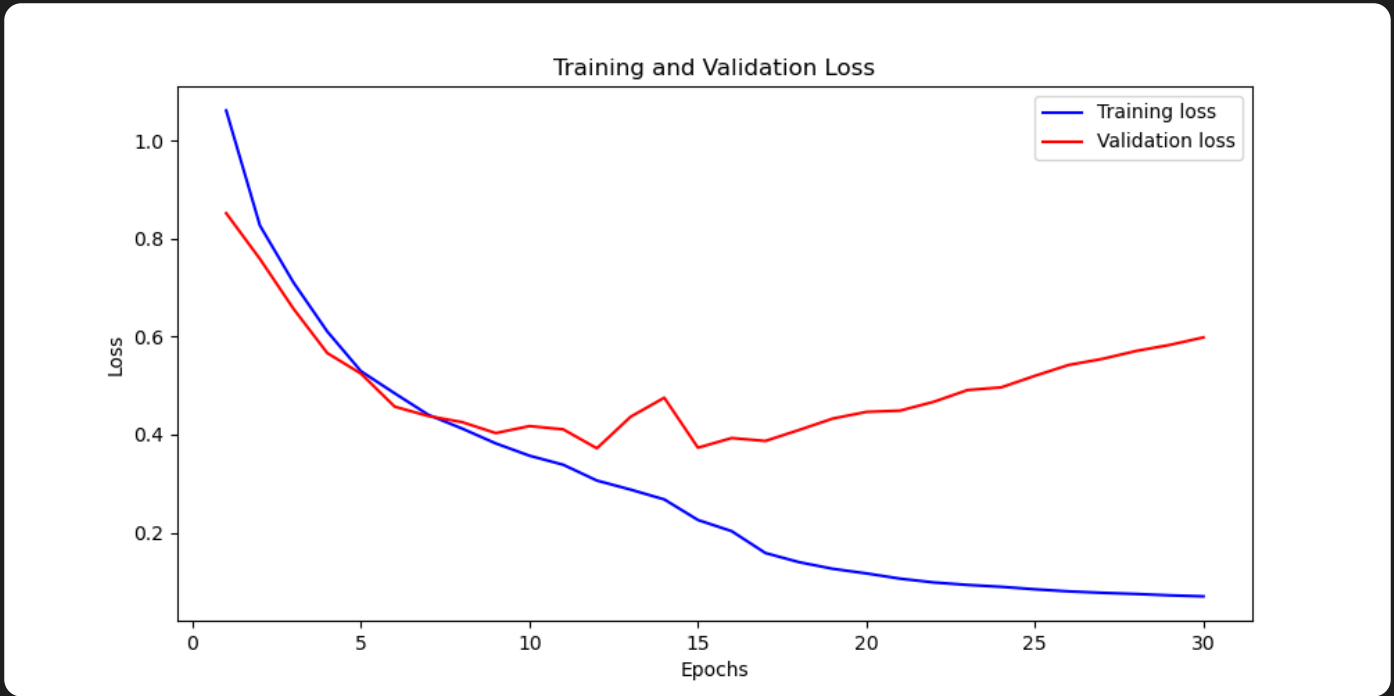
Anything more you want to mention

Analyze on the experiment results

What did you explore during the training process?

Loss curve

- UNet (改善前)

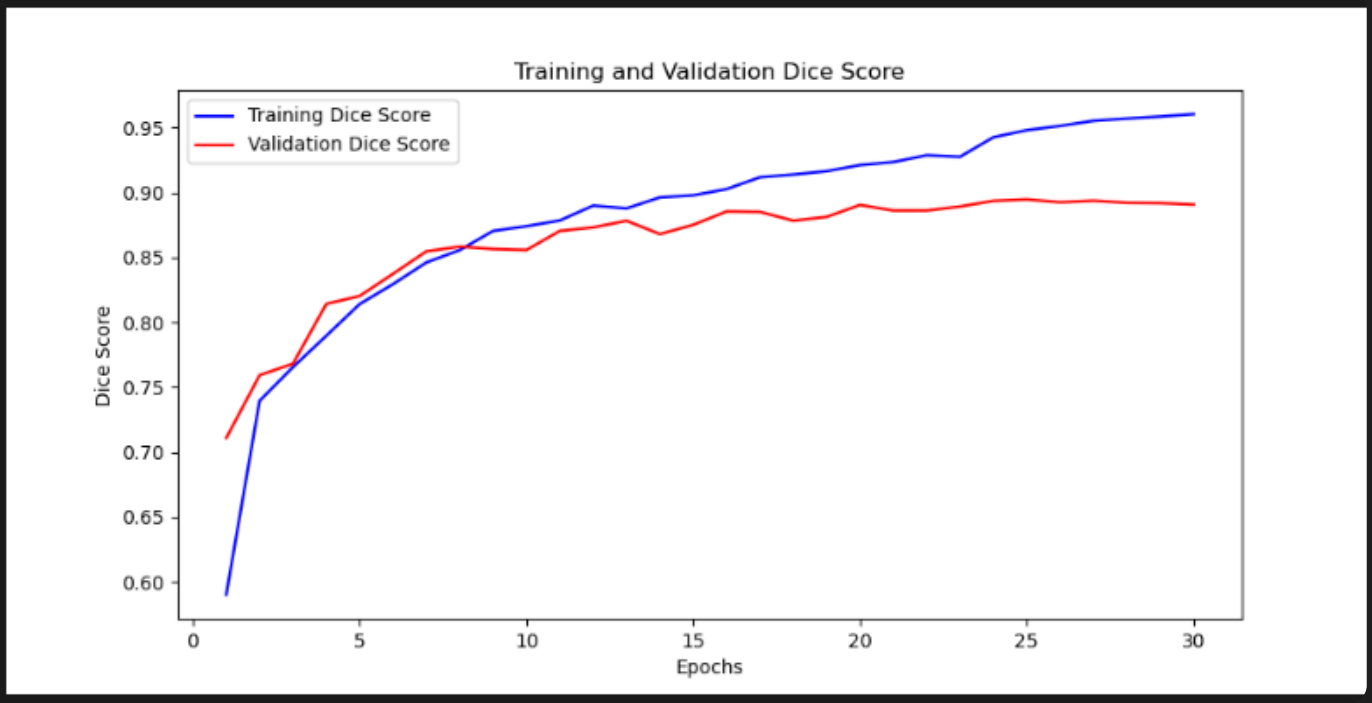


- UNet (改善後)

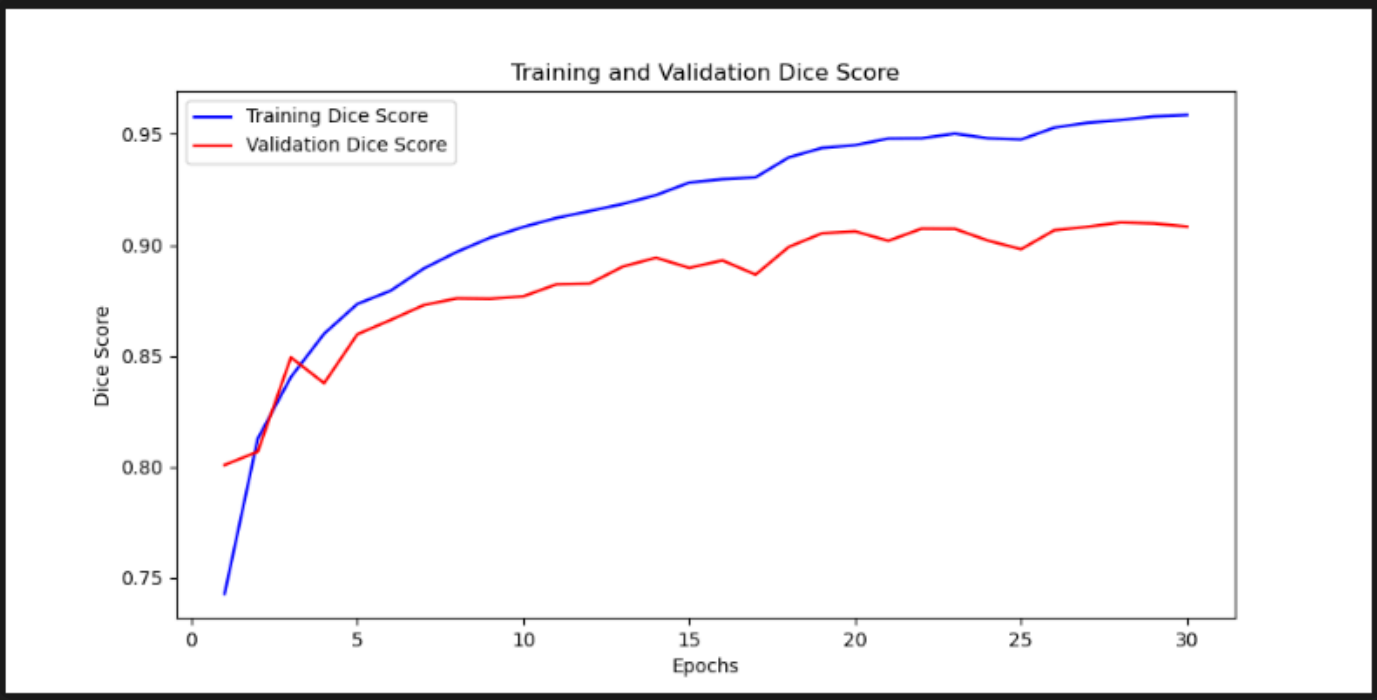
- ResNet34-UNet

Accuracy curve

- UNet (final acc = 0.9036)



- ResNet34-UNet (final acc = 0.9103)



因為這次有 validation dataset，所以我在前幾輪的訓練中，發現不管是 UNet 還是 ResNet34-UNet，都很容易 overfitting (training loss一直都有在持續下降，但到大概15 epochs 後 validation loss 就不降了)。然而，雖然都會 overfitting，但是 ResNet34-UNet 的機體性能非常好，和 UNet 在同樣的訓練環境下，它很快地就可以達成 90% accuracy 的任務。

在接下來，為了讓 UNet 一樣能夠達到 90% 的 accuracy，我首先針對 overfitting 的問題做了簡單的調整，我在 Adam 的優化器上新增了 weight decay 的參數:

```
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate,
weight_decay=0.00001)
```

接著讓 learning rate 能夠動態調整，以免 gradient 卡在區域最小值:

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
patience=2, factor=0.5)
```

最後也將 data augmentation 從 2 種新增到 4 種來提昇資料的多樣性 (參見 Data preprocessing 的 What makes your method unique?)

有了以上這些改動後，UNet 也成功上升到了 90%，並且可以從改善前與改善後的圖發現 **validation 和 training 的 loss 曲線更加靠近了一些**，代表我的訓練策略有成功...。

Found any characteristics of the data?

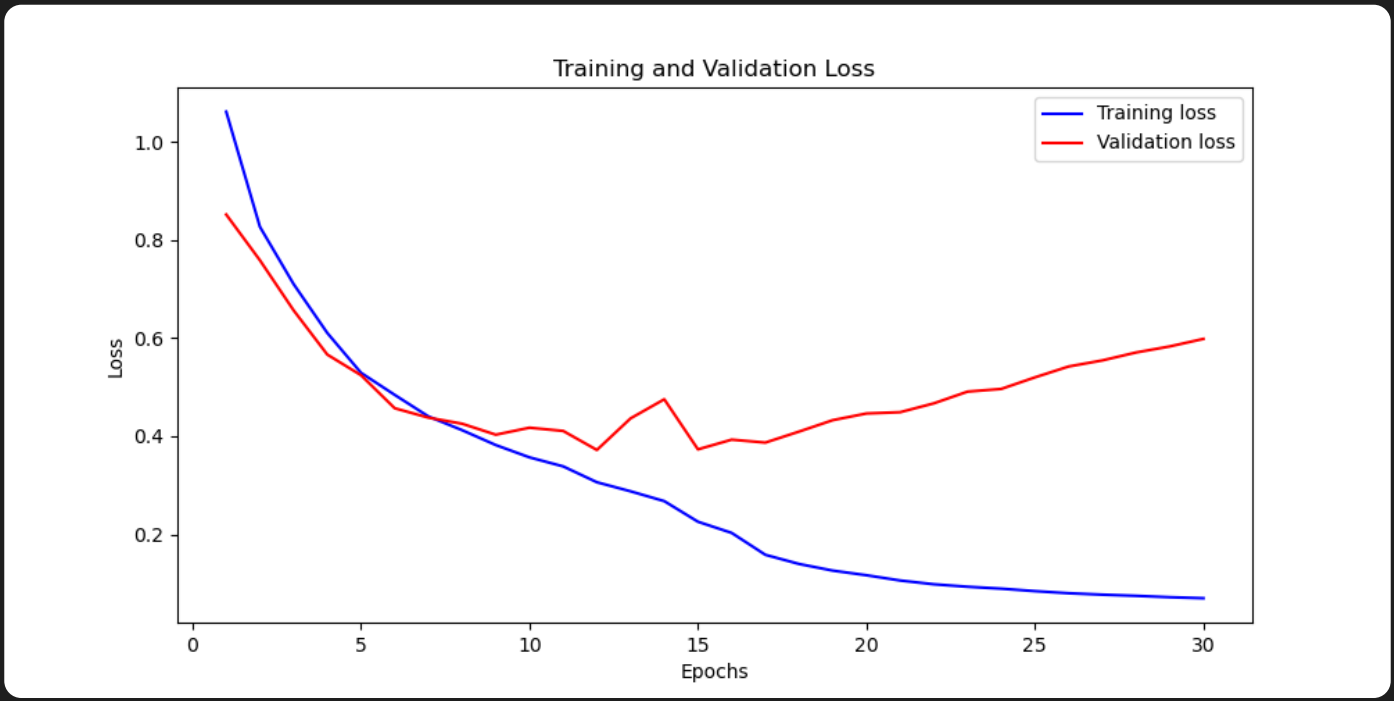
1. **多樣性高的寵物圖片**：
   - 各種品種的貓和狗。
   - 不同的顏色和花紋。
   - 各種不同的姿勢和角度。
2. **遮擋和背景干擾**：
   - 寵物可能被各種物品部分遮擋，如棉被、家具等。
   - 圖片背景各異，可能包含草地、室內地板、家具等。
   - 有些圖片中的寵物可能與背景顏色相似，增加了分割難度。
3. **光照變化**：
   - 圖片拍攝於不同的光照條件下，包括室內和室外，光線強度和方向可能不同。
   - 有些圖片可能存在陰影，影響分割精度。
4. **不同的拍攝距離**：
   - 圖片中寵物的大小和位置各異，有些圖片中的寵物可能佔據整個畫面，有些則僅佔據一部分。
5. **分辨率和清晰度變化**：
   - 圖片的分辨率和清晰度可能不同，有些圖片可能模糊。

總而言之，由於**數據的多樣性高**，模型容易**過擬合於訓練數據中的特定特徵**，導致在驗證和測試數據上的泛化能力較差。

Anything more you want to mention

前往 accuracy 90% 之路:

- Augmentation 上升3% -> 81 -> 84%
- Loss: only bceloss -> bceloss + dice loss -> 上升2%
- lr 上升2% -> 86 -> 88% (lr=1e-5 -> lr = 1e-4)
- unet比較難train，用同樣和res34_unet的架構, res34可以達到90, unet只有88, 而且unet比較容易overfit



上圖為unet, 30epoch, 沒weight decay

因此之後做了 `What did you explore during the training process?` 這個 section 中的改進，成功讓 UNet 也達到 90%。

Execution command

The command and parameters for the training process

- For UNet

```
python3 train.py --model_type unet --data_path ../dataset/oxford-iiit-pet/ -lr 5e-4
```

- For ResNet34-UNet

```
python3 train.py --data_path ../dataset/oxford-iiit-pet/ --model_type
res34  -lr 1e-4
```

The command and parameters for the inference process

Evaluate.py

- For UNet

```
python3 evaluate.py --data_path ../dataset/oxford-iiit-pet/ --model
../saved_models/DL_Lab3_UNet_313551055_柯柏旭.pth --model_type unet
```

- For ResNet34-UNet

```
python3 evaluate.py --data_path ../dataset/oxford-iiit-pet/ --model
../saved_models/DL_Lab3_ResNet34_UNet_313551055_柯柏旭.pth --
model_type res34
```

Inference.py

- For UNet

```
python3 inference.py --data_path ../dataset/oxford-iiit-pet/ --model
../saved_models/DL_Lab3_UNet_313551055_柯柏旭.pth --model_type unet
```

- For ResNet34-UNet

```
python3 inference.py --data_path ../dataset/oxford-iiit-pet/ --model
../saved_models/DL_Lab3_ResNet34_UNet_313551055_柯柏旭.pth --
model_type res34
```

Discussion

What architecture may bring better results?

Ans: **ResNet34-UNet**

單純從實驗結果(`Analyze on the experiment results`這個 section)上來看便有很明顯的差距了，在同樣都會 overfitting 的情況下，ResNet34-UNet 甚至還可以輕鬆達到 90% 的 accuracy。

我推測 ResNet34-UNet 會比較優的原因如下:

1. **ResNet34 更深的結構**：ResNet34 是一個相對較深的網絡，擁有更多的層數，這使得它能夠提取出更深層次、更豐富的特徵。這對於捕捉圖像中的細微結構和高級信息非常重要。
2. **殘差連接（Residual Connections）**：ResNet34 使用殘差連接來解決深層網絡的梯度消失問題，這使得訓練更深層的網絡變得更加容易，並且能夠更有效地學習到圖像中的細節特徵。

反之，單純的 UNet 編碼器結構相對簡單，可能不足以提取複雜的特徵，特別是在處理具有高多樣性和複雜背景的數據集時。(Oxford-pet 其實還挺複雜的，上面有說明。)

What are the potential research topics in this task?

針對二元分割任務的潛在研究主題，這邊提出三個:

1. **模型架構改進**：探索更深層或更複雜的模型架構，如 U-Net++、Attention U-Net 或 Transformer-based segmentation 模型，以提高分割性能。
2. **多尺度特徵融合**(Oxford-Pet 就蠻需要的)：

- 開發多尺度特徵融合技術，將不同尺度的特徵進行有效結合，以增強模型對不同大小目標的識別能力。
- 探討 pyramid pooling 模塊或特徵金字塔網絡（FPN）在二元分割中的應用。

3. **損失函數設計**：研究新的損失函數，如基於 IoU（Intersection over Union）的損失函數，來提高模型對邊界區域的敏感度。

Anything more you want to mention

Reference

- UNet 介紹: https://tomohiroliu22.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92paper%E7%B3%BB%E5%88%97-05-u-net-41be7533c934
- 理解反捲積: https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/%E5%8F%8D%E6%8D%B2%E7%A9%8D-deconvolution-%E4%B8%8A%E6%8E%A1%E6%A8%A3-unsampling-%E8%88%87%E4%B8%8A%E6%B1%A0%E5%8C%96-unpooling-%E5%B7%AE%E7%95%B0-feee4db49a00
- 理解 Oxford-pet dataset: https://www.robots.ox.ac.uk/~vgg/data/pets/
- ChatGPT