HW2
Food Segmentation

資工碩一 M1354024 戴育琪

前置：導入模塊

```
from torch.utils.data import Dataset, DataLoader
import PIL.Image as Image
import os
import numpy as np
import torch
import torch.nn as nn
from torch import optim
import albumentations as A
from albumentations.pytorch import ToTensorV2
from matplotlib import pyplot as plt
import random
import torch.nn.functional as F
from tqdm import tqdm
```

✓ 2.6s

PART1

構建一個自定義數據集並對其進行數據增強、分割與加載，準備好後續訓練過程中使用的圖像和標註數據。

1. make_dataset 函數

```
def make_dataset(root1, root2):
    imgs = []
    lst = []
    for filename in os.listdir(root1):
        img = os.path.join(root1, filename)
        mask = os.path.join(root2, filename.replace('jpg', 'png'))
        imgs.append((img, mask))
    return imgs
```

1. make_dataset 函數

- 功能：生成圖像和標註(mask)文件的路徑對列表。
- 假設圖像以 `.jpg` 格式存儲，而標註文件則以 `.png` 格式存儲，此函數會將文件名從 `.jpg` 轉換為 `.png` 來配對圖片和標註文件。

2. get_labels 和 encode_segmap 函數

```
def get_labels():
    return np.asarray([[0, 0, 0], [1,1,1],])
def encode_segmap(mask):
    mask = mask.astype(int)
    label_mask = np.zeros((mask.shape[0], mask.shape[1]), dtype=np.int16)
    for ii, label in enumerate(get_labels()):
        label_mask[np.where(np.all(mask == label, axis=-1))[:2]] = ii
    label_mask = label_mask.astype(int)
    return label_mask
```

2. get_labels 和 encode_segmap 函數

- `get_labels`:定義了兩種標籤的顏色，`[0, 0, 0]` 和 `[1,1,1]`。
- `encode_segmap`:將 RGB 標註圖像轉換為分類掩碼，這裡每個像素根據顏色被分配到不同的類別(例如，背景為 `0`，對象為 `1`)，為語義分割模型提供二值化的標註掩碼。

3. TrafficDataset 類

```python
class TrafficDataset(Dataset):
    def __init__(self, root1, root2, transform=None):
        imgs = make_dataset(root1, root2)
        self.imgs = imgs
        self.transform = transform


    def __getitem__(self,
        x_path, y_path =
        img = Image.open(x_path)
        mask = Image.open(y_path)
        img = np.array(img)
        mask = np.array(mask)
        if mask.ndim == 3:
            mask = mask[:, :, 2]  # 取出藍色通道作為標籤
        #mask = mask[:,:,2] #get label dim
        mask = np.where(mask>=1, 1,0)
        if self.transform is not None:
            #transformed = transform(image=img, mask=mask)
            transformed = self.transform(image=img, mask=mask)
            img = transformed['image']
            mask = transformed['mask']

        return img, mask

    def __len__(self):
        return len(self.imgs)
```

(variable) x_path: Any

3. TrafficDataset 類

- `TrafficDataset` 類繼承自 `torch.utils.data.Dataset`，用於構建 PyTorch 的自定義數據集。
- `__init__`：初始化數據集，利用 `make_dataset` 生成圖像和標註路徑列表，並設置圖像增強和轉換 `transform`。
- `__getitem__`：加載圖像和標註，並將其轉換為 numpy 陣列格式。當標註有 RGB 通道時，提取藍色通道作為標籤（假設該通道包含標註信息），然後進行二值化處理。若設置了 `transform`，則應用增強和轉換。
- `__len__`：返回數據集的長度。

4. 數據轉換和分割

```
batch_size = 8
w, h = (224, 224)
transform = A.Compose([
    A.Resize(224, 224),
    A.HorizontalFlip(),
    A.VerticalFlip(),
    A.Normalize(),
    ToTensorV2(),
])

train_img_path = '/home/yuchi/AI/food/trainingimages'
train_mask_path = '/home/yuchi/AI/food/traininglabels'

trainset = TrafficDataset(train_img_path, train_mask_path, transform=transform)
valsize = int(len(trainset)*0.2)
trainsize = len(trainset) - valsize
print(trainsize, valsize)
train_set, val_set = torch.utils.data.random_split(trainset, [trainsize, valsize])
```

4. 數據轉換和分割

- 使用 `albumentations` 庫設置數據增強，包括圖像的縮放、水平/垂直翻轉、正則化和轉為 Tensor 格式。
- 將數據集劃分為訓練集和驗證集，其中訓練集占 80%，驗證集占 20%。

5. 可視化示例

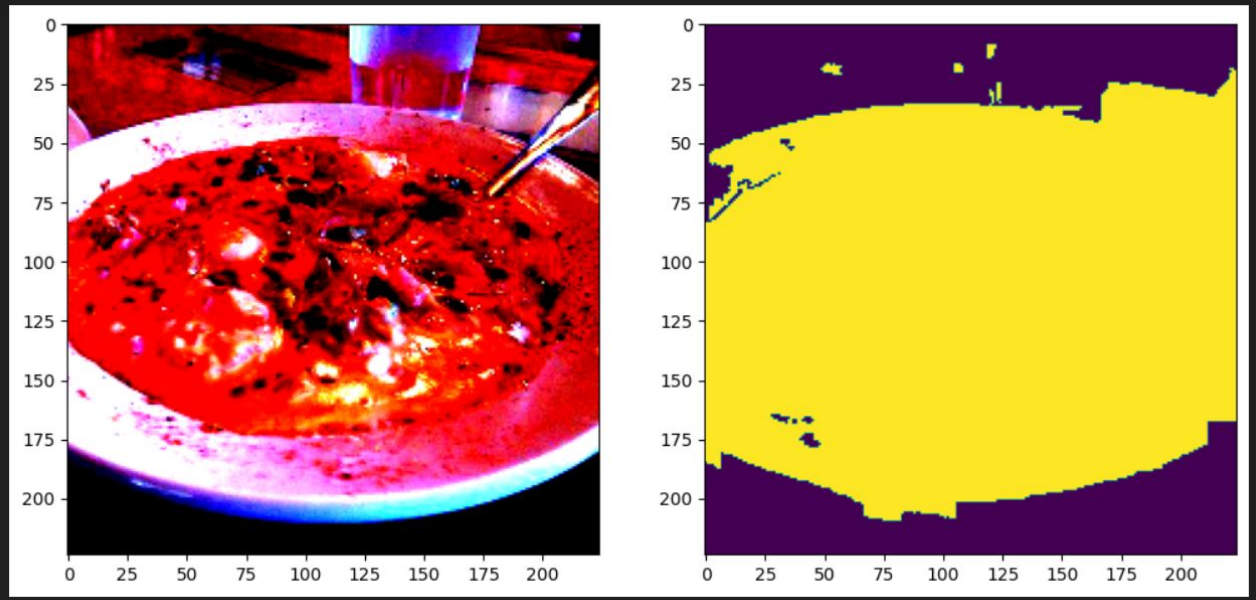
```
l = len(trainset)
print(l)
img, mask = trainset[random.randint(0, l)]
print(img.shape)
print(mask.shape)
img = img.permute(1,2,0).numpy()
mask = mask.numpy()

print(img.shape)
print(mask.shape)

fig = plt.figure(figsize=(12, 12))
fig.add_subplot(1, 2, 1)
plt.imshow(img)
fig.add_subplot(1, 2, 2)
plt.imshow(mask)
plt.show()
```

5. 可視化示例

- 隨機選擇一對圖像和標註並進行可視化，顯示圖像和對應的標註掩碼，以確認數據處理和標註格式是否正確。

5. 可視化示例 -- Output

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
8780 2195
10975
torch.Size([3, 224, 224])
torch.Size([224, 224])
(224, 224, 3)
(224, 224)
```



6.數據加載

```
train_dataloaders = DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=4)
val_dataloaders = DataLoader(val_set, batch_size=batch_size, shuffle=True, num_workers=4)
```

6.數據加載

- 使用 PyTorch 的 `DataLoader` 將訓練集和驗證集分別加載為批次數據，便於訓練過程中的批量處理。

PART2

構建一個基於 U-Net 結構的分割模型，並引入 ResNet 殘差塊來加強特徵提取能力。

1. ResNet Block

```python
# Define ResNet block
class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.relu(out)
        out = self.conv2(out)
        out += residual
        out = self.relu(out)
        return out
```

1. ResNet Block

- **功能**：ResNet 殘差塊包含兩個卷積層，並使用跳躍連接將輸入直接加到輸出中，這樣可以緩解深層網路中的梯度消失問題。
- **流程**：輸入先經過第一個卷積層和 ReLU 激活，再通過第二個卷積層後與原始輸入相加，再經過 ReLU 激活輸出。

2. U-Net 模型

```python
# Define UNet model
class Unet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Unet, self).__init__()
        # Encoder
        self.conv1 = nn.Conv2d(in_channels, 64, kernel_size=3, padding=1)
        self.block1 = ResNetBlock(64, 64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.block2 = ResNetBlock(128, 128)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.block3 = ResNetBlock(256, 256)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.block4 = ResNetBlock(512, 512)

        # Decoder
        self.upconv3 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.conv5 = nn.Conv2d(512, 256, kernel_size=3, padding=1)  # Match concatenated channels

        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.conv6 = nn.Conv2d(256, 128, kernel_size=3, padding=1)  # Match concatenated channels

        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.conv7 = nn.Conv2d(128, 64, kernel_size=3, padding=1)   # Match concatenated channels

        self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)  # Output layer
```

```python
def forward(self, x):
    # Encoder
    x1 = self.conv1(x)
    x1 = self.block1(x1)
    x2 = self.pool1(x1)
    x2 = self.conv2(x2)
    x2 = self.block2(x2)
    x3 = self.pool2(x2)
    x3 = self.conv3(x3)
    x3 = self.block3(x3)
    x4 = self.pool3(x3)
    x4 = self.conv4(x4)
    x4 = self.block4(x4)

    # Decoder
    x = self.upconv3(x4)
    x = torch.cat((x, x3), dim=1)  # Concatenate with encoder output
    x = self.conv5(x)

    x = self.upconv2(x)
    x = torch.cat((x, x2), dim=1)
    x = self.conv6(x)

    x = self.upconv1(x)
    x = torch.cat((x, x1), dim=1)
    x = self.conv7(x)

    x = self.final_conv(x)
    return x
```

2. U-Net 模型

- **功能**：U-Net 結構包含編碼器 (Encoder) 和解碼器 (Decoder)，適合用於語義分割任務。
- **Encoder 部分**：逐層提取特徵，並使用 ResNetBlock 強化特徵表示。每層之間用 `MaxPool2d` 進行下採樣。
- **Decoder 部分**：逐步上採樣並使用 `ConvTranspose2d` 進行反卷積，並與對應層的編碼器輸出拼接，這樣能保留低層次的細節特徵。
- **Concatenate (拼接)**：在解碼器中，每個反卷積層的輸出與編碼器中相對應層的輸出進行拼接，使得模型能夠更好地融合上下文信息。

2. U-Net 模型 – Output

```
Unet(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (block1): ResNetBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu): ReLU(inplace=True)
  )
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (block2): ResNetBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu): ReLU(inplace=True)
  )
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (block3): ResNetBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu): ReLU(inplace=True)
  )
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv4): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (block4): ResNetBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
...
  (conv6): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (upconv1): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
  (conv7): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (final_conv): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
)
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

3. 權重初始化

```
# Initialize weights
def init_weights(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        nn.init.xavier_uniform_(m.weight)
```

3. 權重初始化

- 使用 `Xavier` 初始化方法來隨機初始化卷積層和反卷積層的權重，使網路中的權重值具有更好的分布。

4.模型實例化

```
in_channels = 3
out_channels = 1
model = Unet(in_channels, out_channels)
model.apply(init_weights)
```

4. 模型實例化

- **功能**：創建一個 U-Net 模型實例，設置輸入通道數（`in_channels = 3` 表示 RGB 圖像）和輸出通道數（`out_channels = 1` 表示二分類掩碼），並應用自定義的權重初始化函數 `init_weights`。

PART3

此訓練循環將 U-Net 模型訓練一個 epoch，使用 BCE with Logits Loss 計算損失，並在損失最小時儲存模型。

1. 訓練設定與初始化

```
num_epochs = 1
batch_size = 4
device = torch.device("cpu")
in_channel, out_channel = 3, 1
model = Unet(in_channel, out_channel).to(device)
optimizer = optim.Adam(model.parameters(), lr=0.0008)
# Binary Cross Entropy Loss (適合二元分割)
criterion = nn.BCEWithLogitsLoss()  # 使用 logit 的 BCE 適合在最後一層未加 sigmoid 的情況
best_val_loss = float('inf')  # 初始化最佳驗證損失為無限大
```

1. 訓練設定與初始化

- **功能**：設置訓練參數和初始化模型。將模型移動到指定的設備（如 CPU 或 GPU），並使用 Adam 優化器進行參數更新，損失函數選擇 BCE with Logits Loss。`best_val_loss` 用於儲存最佳的驗證損失，以便於模型存儲。

2. 訓練循環

```python
# 训练循环
for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}/{num_epochs}")
    model.train()
    total_loss = 0.0
    for images, masks in train_dataloaders:
        images, masks = images.to(device), masks.to(device)
        #masks = masks.to(device=device, dtype=torch.long)  # Ensure masks are LongTensor
        masks = masks.unsqueeze(1)
        masks = masks.float()

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

        total_loss += loss.item()
```

2. 訓練循環

- `model.train()`:將模型設置為訓練模式，以便於啟用 dropout 等訓練專用層。
- `for images, masks in train_dataloaders`: 遍歷訓練數據集，每次從 DataLoader 中提取一批圖像和相應的標籤。
- `masks.unsqueeze(1)`:為分割標籤增加一個通道維度，因為模型的輸出形狀為 `[batch_size, 1, height, width]`。
- `optimizer.zero_grad()`:清空梯度，以便於計算新的梯度。
- `outputs = model(images)`:將圖像輸入模型，得到輸出結果。
- `loss.backward()`:計算梯度，將損失向後傳播。
- `optimizer.step()`:根據計算出的梯度更新模型參數。

3. 模型儲存 &損失打印

```python
if total_loss < best_val_loss:
    best_val_loss = total_loss
    torch.save(model.state_dict(), '/home/yuchi/AI/best_model.pth')
    print(f"Model saved with validation loss: {total_loss:.4f}")

print(f"Epoch {epoch+1}/{num_epochs}, Loss: {total_loss/len(train_dataloaders)}")
```

3. 模型儲存 &損失打印

- 如果當前 epoch 的訓練損失小於先前最佳驗證損失，則將當前模型的參數儲存為最佳模型，以便在訓練過程中保存表現最好的模型。
- 每個 epoch 結束時，打印平均損失，以便於觀察訓練過程中的損失變化。

3. 模型儲存 &損失打印

```
Epoch 1/1
Model saved with validation loss: 541.5685
Epoch 1/1, Loss: 0.4932318235898495 7
```

PART4

U-Net 模型在測試集上的預測，並將生成的二值化分割掩碼保存到指定的路徑。

1. 模型初始化與權重加載

```
test_img_path = '/home/yuchi/AI/food/testimages'
test_mask_path = '/home/yuchi/AI/food/testlabels'
model_path = "/home/yuchi/AI/best_model.pth"

# 初始化模型並載入權重
in_channel, out_channel = 3, 1
model = Unet(in_channel, out_channel).to(device)
model.load_state_dict(torch.load(model_path))
model.eval()
```

1. 模型初始化與權重加載

- 初始化一個 U-Net 模型，並載入已訓練的最佳模型權重（在之前訓練過程中儲存的 `best_model.pth`）。`model.eval()` 將模型設置為評估模式，以確保 dropout 層等僅在訓練模式下啟用的層不再作用。

2. 定義圖像轉換

```
w, h = (128, 128)
transform = A.Compose([
    A.Resize(128, 128),
    A.HorizontalFlip(),
    A.VerticalFlip(),
    A.Normalize(),
    ToTensorV2(),
],is_check_shapes=False)
```

2. 定義圖像轉換

- 這段代碼定義了圖像轉換（如大小調整、水平與垂直翻轉、標準化等），並應用於每張測試影像。設定 `is_check_shapes=False` 用於避免對形狀檢查，讓測試時的形狀變化更靈活。

3.遍歷測試影像並生成二 值化掩碼&保存生成的掩碼

```python
# 遍歷每張測試影像並生成掩碼
threshold = 0.5
# 遍歷每張測試影像並生成二值化掩碼
with torch.no_grad():
    for img_name in tqdm(os.listdir(test_img_path), desc="Generating masks"):
        # 讀取並處理影像
        img_path = os.path.join(test_img_path, img_name)
        image = Image.open(img_path).convert("RGB")
        image = np.array(image)  # 將 PIL 圖像轉為 numpy 格式

        # 應用轉換
        transformed = transform(image=image)
        image = transformed["image"].unsqueeze(0).to(device)  # 增加批次維度

        # 預測掩碼
        output = model(image)
        pred = torch.sigmoid(output).squeeze().cpu().numpy()  # 使用 Sigmoid 激活並移除多餘的維度
        pred = (pred > threshold).astype(np.uint8) * 255  # 閾值化為 0 和 255，轉為二值掩碼

        # 保存二值化掩碼
        mask = Image.fromarray(pred)  # 轉為 PIL 格式
        mask_name = img_name.replace('.jpg', '.png')  # 修改檔名為掩碼格式
        mask.save(os.path.join(test_mask_path, mask_name))

print("所有測試掩碼已成功生成並儲存至:", test_mask_path)
```

✓ 3m 17.8s

3. 遍歷測試影像並生成二 值化掩碼&保存生成的掩碼

- `for img_name in tqdm(...)`：遍歷所有測試影像，並顯示進度條。
- `torch.sigmoid(output)`：使用 Sigmoid 激活函數將模型輸出映射至 [0, 1] 範圍，適合二元分割。
- `(pred > threshold).astype(np.uint8) * 255`：將預測結果二值化，使用閾值化為 0 或 255。
- 將二值化掩碼轉為 PIL 圖片格式並保存到指定路徑。通過替換影像的副檔名來生成對應的掩碼檔名（如 `.jpg` 轉 `.png`），並將結果存儲在 `test_mask_path` 下。

3.遍歷測試影像並生成二 值化掩碼&保存生成的掩碼

```
Generating masks: 100%|██████████| 1636/1636 [03:17<00:00,  8.27it/s]
所有測試掩碼已成功生成並儲存至: /home/yuchi/AI/food/testlabels
```

PART 5

在測試集上評估訓練好的 U-Net 模型，並計算測試損失。

1. 資料與轉換設定

```
w, h = (128, 128)
transform = A.Compose([
    A.Resize(height=128, width=128),
    A.Normalize(),
],is_check_shapes=False)

testset = TrafficDataset(test_img_path, test_mask_path, transform=transform)
bs = batch_size
test_dataloaders = DataLoader(testset, batch_size=bs, shuffle=False, num_workers=4)
```

1. 資料與轉換設定

- 定義了圖像轉換(128x128大小和標準化處理)，並將其應用於 `testset`。
  `DataLoader` 將 `testset` 以批次大小 `bs` 加載，便於批次處理。

2. 模型與損失函數設定

```
device = torch.device('cpu')
in_channel, out_channel = 3, 1
model = Unet(in_channel, out_channel).to(device)
model.load_state_dict(torch.load(model_path))
model.eval()

epoch_loss=0
gt = np.empty([0, 1, 128, 128])
result = np.empty([0, 1, 128, 128])
criterion = nn.BCEWithLogitsLoss()
```

2. 模型與損失函數設定

- 將已訓練的模型載入並設為評估模式（`model.eval()`），以確保 dropout 和 batch normalization 等層保持不變。使用 `BCEWithLogitsLoss` 作為損失函數，適合二元分割。

3. 進度條與測試迴圈

```
pbar = tqdm(total = len(test_dataloaders)+1, ncols=150)
c = 0
epoch_loss=0
for x, y in test_dataloaders:
    inputs = x.permute(0, 3, 1, 2).to(device).float()
    labels = y.to(device).float().unsqueeze(dim=1)
    outputs = model(inputs)

    loss = criterion(outputs, labels)
    epoch_loss += loss.item()

    outputs = F.sigmoid(outputs)
    result = np.concatenate((result, outputs.cpu().detach().numpy()), axis=0)
    gt = np.concatenate((gt, labels.cpu().detach().numpy()), axis=0)

    pbar.update(1)
```

3. 進度條與測試迴圈

- 這段代碼遍歷測試資料，將模型輸出的 logits 經過 Sigmoid 函數轉為概率值。`epoch_loss` 累加每批數據的損失，以最終計算平均損失。
- `inputs = x.permute(0, 3, 1, 2)`：將影像從 `NHWC`（批次大小、寬度、高度、通道）格式轉為 `NCHW`，符合 PyTorch 的模型輸入需求。
- `outputs = F.sigmoid(outputs)`：將模型輸出經過 Sigmoid，得出每個像素點的二元分割概率。
- `np.concatenate`：將當前批次的預測結果 `outputs` 和標籤 `labels` 分別累加到 `result` 和 `gt` 中，便於後續進行更詳細的結果分析。

4. 計算並輸出損失

```
epoch_loss /= len(test_dataloaders)
pbar.update(1)
pbar.close()
print(epoch_loss, 1-epoch_loss)
```

4. 計算並輸出損失

- 計算測試集中每批次損失的平均值，並輸出損失以及（1 - 損失）作為模型在測試集上的表現指標。

```
100%|██████████| 410/410 [02:57<00:00,  2.31it/s]
0.37545727838222437 0.6245427216177757
```

PART6

模型的分割結果轉換為 Run-Length Encoding (RLE) 格式，並將其儲存為 CSV 文件。

1. 轉換分割結果為二 值化格式

```
threthold = 0.5
pred = np.where(result>=threthold,1,0)
print(pred[0].shape, pred[0].dtype)
print(np.unique(pred[0]))
```

1. 轉換分割結果為二 值化格式

- 將 `result` (模型預測的浮點數概率) 按照閾值 0.5 進行二值化處理。大於等於 0.5 的像素被設為 1 (前景)，小於 0.5 的像素設為 0 (背景)。
- 使用 `np.unique(pred[0])` 查看 `pred` 的唯一值，以確保二值化成功。

2. RLE 編碼函數 rle_encoding

```
def rle_encoding(x):
    dots = np.where(x.flatten()==1)[0]
    run_lengths = []
    prev = -2
    for b in dots:
        if (b>prev+1): run_lengths.extend((b+1, 0))
        run_lengths[-1] += 1
        prev = b
    return run_lengths
```

2. RLE 編碼函數 rle_encoding

此函數接收二值化掩碼 x，將前景區域 (像素值為 1 的位置) 進行 RLE 編碼。

- `np.where(x.flatten() == 1)[0]` 將掩碼展平並找到前景像素的位置。
- `run_lengths` 儲存了每段前景的開始位置和長度，以方便壓縮存儲。
- 若當前前景像素 `b` 與上個前景像素的索引 `prev` 不連續 (`b > prev + 1`)，則添加新的編碼段。

3. 將所有掩碼轉為 RLE 並寫入 CSV

```
def convert_into_rle(mask, pred_root):
    encoding = []
    for i, m in enumerate(mask[:]):
        encoding.append(rle_encoding(m))

    print(len(encoding))

    with open(pred_root, 'w', ) as csvfile:
        csvfile.write("ImageId,EncodedPixels\n")
        for i, m in enumerate(encoding):
            conv = lambda l: ' '.join(map(str, l))
            subject, img = 1, 1
            text = '{},{}'.format(i,conv(encoding[i]))+'\n'
            csvfile.write(text)
```

3. 將所有掩碼轉為 RLE 並寫入 CSV

此函數將所有影像的掩碼（已經二值化的 `pred`）轉換為 RLE 格式，並儲存到指定路徑的 CSV 文件。

- 對 `mask` 的每個掩碼應用 `rle_encoding`，將結果存入 `encoding` 列表。
- 建立 CSV 文件，並寫入 `ImageId` 和 `EncodedPixels` 的標題。每行對應一個影像的 RLE 編碼。

4. 執行 convert_into_rle

```
convert_into_rle(pred, '/home/yuchi/AI/pred_08_1.csv')
```

4. 執行 convert_into_rle

- 將生成的二值化掩碼 `pred` 經過 `convert_into_rle` 函數轉換為 RLE 格式，並存入 `/home/yuchi/AI/pred_08_1.csv` 文件。

4. 執行 convert_into_rle – Output

```
[0.00641152 0.00689651 0.0069556  ... 0.98224741 0.98278332 0.98340732]
(1, 128, 128) int64
[0 1]
1636
```

PART6

Conclusion

Conclusion

這次實驗讓我發現，在深度學習專案裡，每個步驟都環環相扣。同時，也學到怎麼把結果可視化，這樣更方便了解模型的表現。

總之，這次經歷不但讓我更了解圖像分割任務，也提升了我解決實際問題的能力。希望以後能把這些經驗更好地應用到未來的專案中