Assignment 02 (EE655)

Ques01 **a:**

Approach – Part (a): Foreground Segmentation using Otsu's Method

1. **Load MNIST dataset using PyTorch with tensor transformation.**

2. **Create directories to store grayscale images and their corresponding binary masks.**

3. **For each image:**

    - **Convert the tensor to a 28×28 grayscale NumPy array.**

    - **Scale pixel values to [0, 255].**

    - **Apply Otsu's thresholding to generate a binary mask separating digit (foreground) from background.**

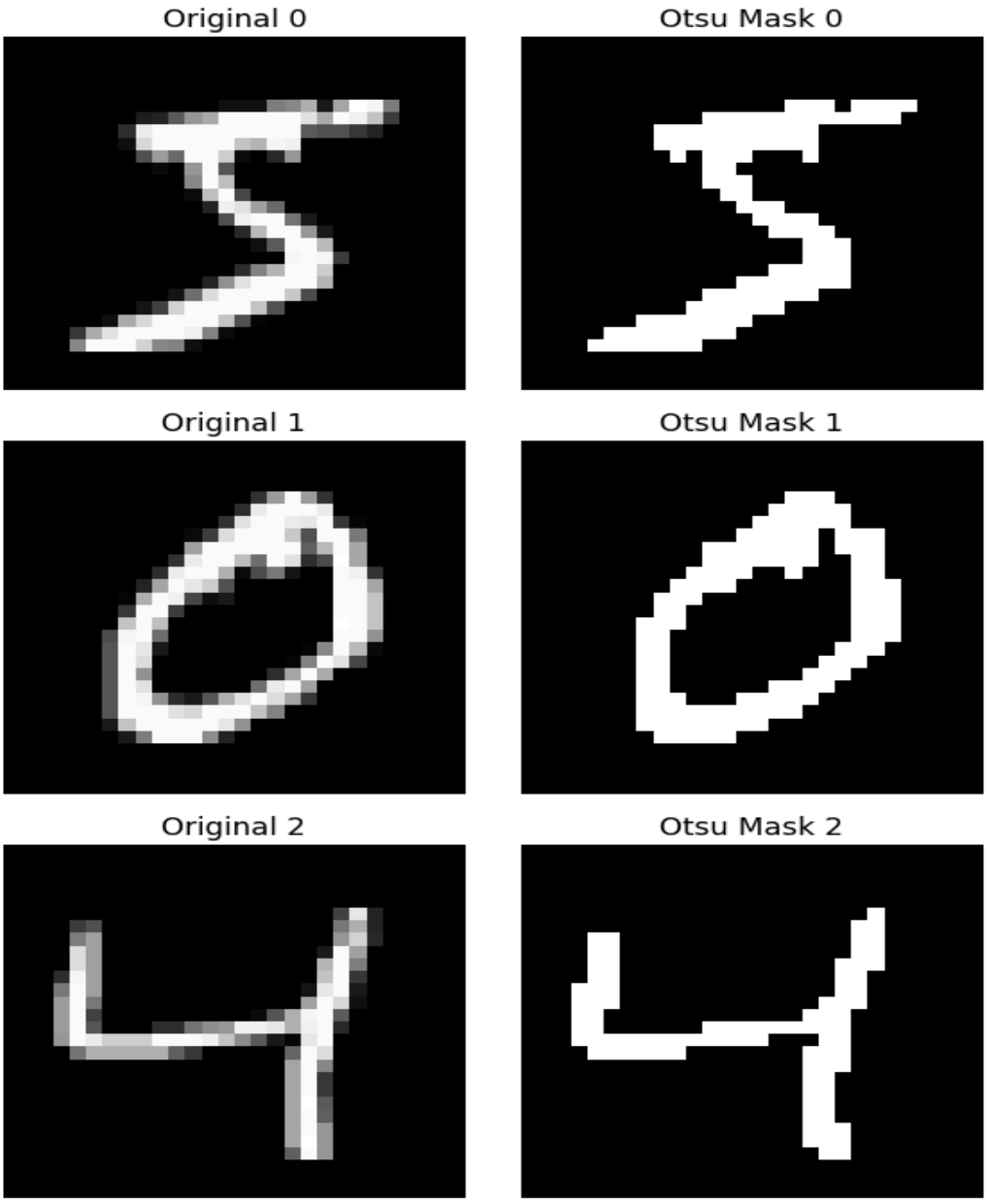
    - **Save both the grayscale image and its mask as PNG files.**

4. **Visualize a few samples to verify the segmentation quality.**

```
# Part (a): Foreground Segmentation using Otsu's Method

import numpy as np
import cv2
import os
from torchvision import datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

```
from PIL import Image

# Step 1: Loading MNIST Dataset
transform = transforms.Compose([transforms.ToTensor()])
mnist_data = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)

# Step 2: Creating the folders to store the new dataset
image_dir = "dataset/Q1a/images"
mask_dir = "dataset/Q1a/masks"
os.makedirs(image_dir, exist_ok=True)
os.makedirs(mask_dir, exist_ok=True)

# Step 3: Applying Otsu Thresholding to each image and save
for idx, (image_tensor, label) in enumerate(mnist_data):
    image_array = image_tensor.squeeze().numpy()  # Convert to 28x28 numpy array
    image_gray = (image_array * 255).astype(np.uint8)  # Scale to 0-255
    _, mask = cv2.threshold(image_gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    # Saving the images and corresponding masks
    Image.fromarray(image_gray).save(os.path.join(image_dir, f"{idx}.png"))
    Image.fromarray(mask).save(os.path.join(mask_dir, f"{idx}.png"))

# Step 4: Visualizing a few samples from the newly created dataset
sample_indices = [0, 1, 2]
fig, axes = plt.subplots(len(sample_indices), 2, figsize=(6, 9))

for i, idx in enumerate(sample_indices):
    img = np.array(Image.open(os.path.join(image_dir, f"{idx}.png")))
    msk = np.array(Image.open(os.path.join(mask_dir, f"{idx}.png")))

    axes[i, 0].imshow(img, cmap='gray')
    axes[i, 0].set_title(f"Original {idx}")
    axes[i, 0].axis('off')

    axes[i, 1].imshow(msk, cmap='gray')
    axes[i, 1].set_title(f"Otsu Mask {idx}")
    axes[i, 1].axis('off')

plt.tight_layout()
plt.show()
```

Result :



Ques 01 (b)

Approach – Part (b): Circular Localization using Min Enclosing Circle

1. **Create directories for storing new images and labels.**
2. **For each binary mask from Part (a):**
    - **Extract external contours using OpenCV.**

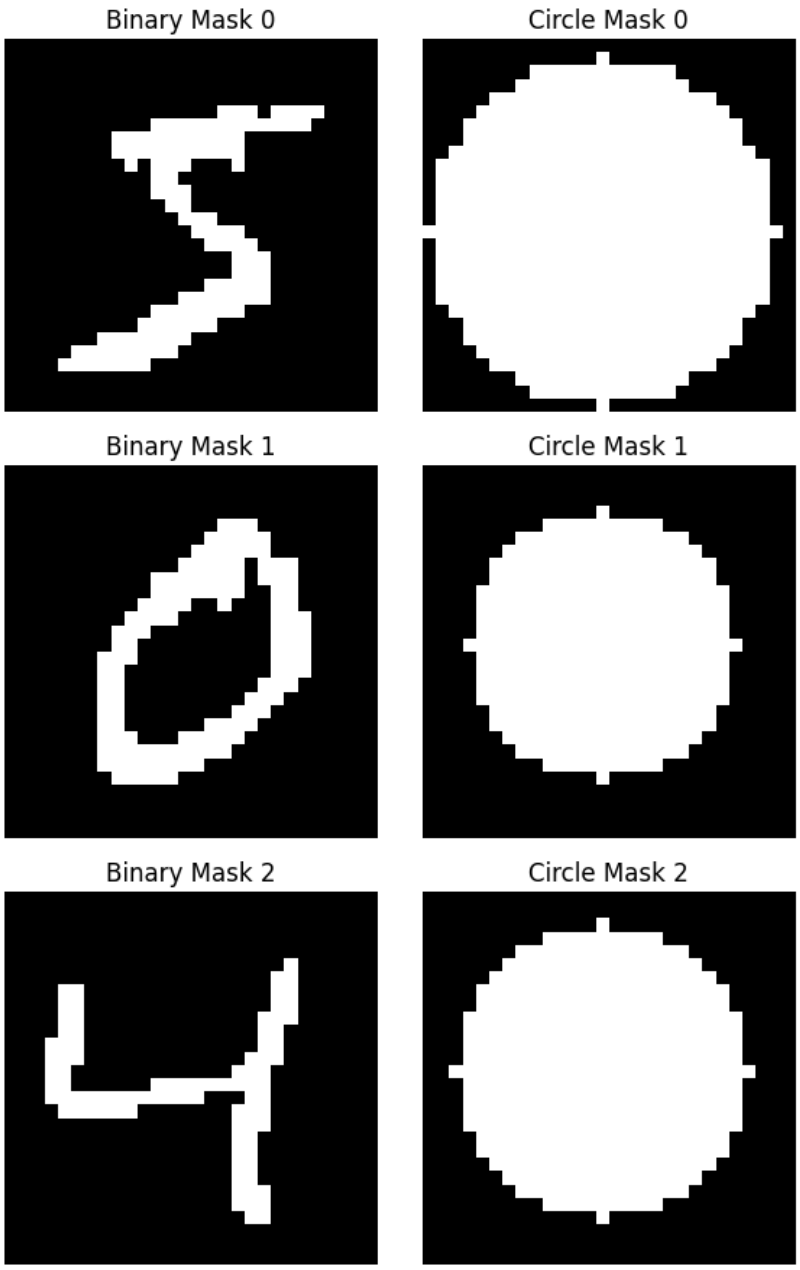
- **Identify the largest contour and compute its minimum enclosing circle.**
- **Draw the circle as a new binary mask.**
- **Save this circle mask as an image.**
- **Save the digit label as a text file.**

3. **Visualize a few samples comparing the original mask with the new circular mask.**

```
# Paths from Part (a)
mask_dir_a = "dataset/Q1a/masks"
img_dir_b = "dataset/Q1b/images"
label_dir_b = "dataset/Q1b/labels"

# Make new directories for Part (b)
os.makedirs(img_dir_b, exist_ok=True)
os.makedirs(label_dir_b, exist_ok=True)

circle_masks = []

# Loop through all Part (a) masks
for idx in range(len(mnist_data)):
    mask_path = os.path.join(mask_dir_a, f"{idx}.png")
    mask = np.array(Image.open(mask_path))
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        (x, y), radius = cv2.minEnclosingCircle(largest_contour)
        circle_mask = np.zeros_like(mask)
        cv2.circle(circle_mask, (int(x), int(y)), int(radius), 255, -1)
    else:
        circle_mask = np.zeros_like(mask)  # fallback

    # Save mask and label
    Image.fromarray(circle_mask).save(os.path.join(img_dir_b, f"{idx}.png"))
    with open(os.path.join(label_dir_b, f"{idx}.txt"), "w") as f:
        f.write(str(mnist_data[idx][1]))  # digit class
```

```
    circle_masks.append(circle_mask)

# Visualize a few samples
sample_indices = [0, 1, 2]
fig, axes = plt.subplots(len(sample_indices), 2, figsize=(6, 9))

for i, idx in enumerate(sample_indices):
    original_mask = np.array(Image.open(os.path.join(mask_dir_a, f"{idx}.png")))
    circ_mask = np.array(Image.open(os.path.join(img_dir_b, f"{idx}.png")))

    axes[i, 0].imshow(original_mask, cmap='gray')
    axes[i, 0].set_title(f"Binary Mask {idx}")
    axes[i, 0].axis('off')

    axes[i, 1].imshow(circ_mask, cmap='gray')
    axes[i, 1].set_title(f"Circle Mask {idx}")
    axes[i, 1].axis('off')

plt.tight_layout()
plt.show()
```

Result :



<u>Ques01</u> (c)
Create folders to store composite images and masks.

For each sample:

- **Randomly select 4 different MNIST images.**

- **Apply Otsu's thresholding to each to generate binary masks.**

- **Arrange them in a 2×2 grid to create a composite image and corresponding mask.**

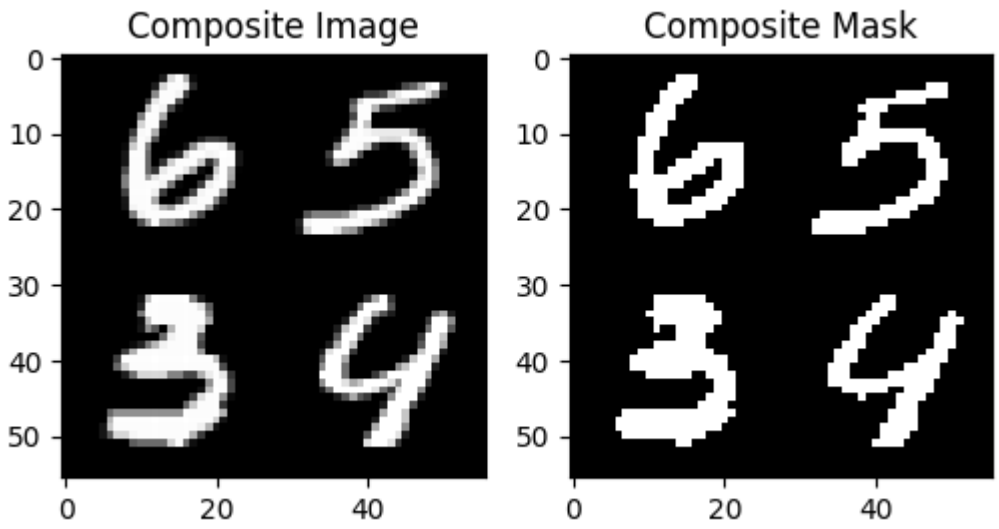
Save the composite image and mask.

Visualize a random composite sample.

```
# Paths to save the new dataset
save_image_dir = "dataset/Q1c/images"
save_mask_dir = "dataset/Q1c/masks"

# Creating directories if not exist
os.makedirs(save_image_dir, exist_ok=True)
os.makedirs(save_mask_dir, exist_ok=True)



composite_images = []
composite_masks = []

# Creating N composite samples
num_samples = 100  # WE can increase this number as needed

for sample_idx in range(num_samples):
   indices = random.sample(range(len(mnist_data)), 4)

   images = []
   seg_masks = []

   for idx in indices:
       image_array = mnist_data[idx][0].squeeze().numpy()
       image_gray = (image_array * 255).astype(np.uint8)

       # Otsu's thresholding for mask
       _, binary_mask = cv2.threshold(image_gray, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)

       images.append(image_gray)
       seg_masks.append(binary_mask)
```

```
    # Composing images and masks into 2x2 layout
    top_img = np.hstack((images[0], images[1]))
    bottom_img = np.hstack((images[2], images[3]))
    composite_image = np.vstack((top_img, bottom_img))

    top_mask = np.hstack((seg_masks[0], seg_masks[1]))
    bottom_mask = np.hstack((seg_masks[2], seg_masks[3]))
    composite_mask = np.vstack((top_mask, bottom_mask))

    # Saving image and mask to dataset directory
    Image.fromarray(composite_image).save(os.path.join(save_image_dir,
f"{sample_idx}.png"))
    Image.fromarray(composite_mask).save(os.path.join(save_mask_dir,
f"{sample_idx}.png"))

    composite_images.append(composite_image)
    composite_masks.append(composite_mask)

# Visualizing a random sample
rand_idx = random.randint(0, num_samples - 1)
plt.figure(figsize=(6, 3))
plt.subplot(1, 2, 1)
plt.title("Composite Image")
plt.imshow(composite_images[rand_idx], cmap='gray')

plt.subplot(1, 2, 2)
plt.title("Composite Mask")
plt.imshow(composite_masks[rand_idx], cmap='gray')
plt.show()
```

Result:



Ques2:

Approach: Training a DL network for Foreground Segmentation

1. **Custom Dataset Class:**
    - **Loads grayscale images and corresponding binary masks from `Q1a` dataset.**
    - **Applies `ToTensor()` transformation.**
2. **CNN Architecture (SimpleSegNet):**
    - **Encoder: Two convolutional layers with ReLU + MaxPooling.**
    - **Decoder: Two transposed convolution layers with ReLU and final Sigmoid.**
3. **Training Setup:**
    - **Dataset split: 80% train, 20% test.**

- Loss: Binary Cross Entropy.
- Optimizer: Adam.
- Training for 5 epochs using batches of size 64.

4. Evaluation:
   - Predictions are thresholded at 0.5.
   - Mean IoU (Intersection over Union) is computed using `sklearn` for test set.

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import numpy as np
from torchvision import transforms
from sklearn.metrics import jaccard_score
import matplotlib.pyplot as plt

# ================================
# Dataset Class
# ================================
class SegmentationDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_paths = sorted([os.path.join(image_dir, f) for f in
os.listdir(image_dir)])
        self.mask_paths = sorted([os.path.join(mask_dir, f) for f in
os.listdir(mask_dir)])
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
```

```
        image = Image.open(self.image_paths[idx]).convert("L")
        mask = Image.open(self.mask_paths[idx]).convert("L")
        if self.transform:
            image = self.transform(image)
            mask = self.transform(mask)
        return image, mask

# Defining Simple CNN Model

class SimpleSegNet(nn.Module):
    def __init__(self):
        super(SimpleSegNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(16, 8, 2, stride=2), nn.ReLU(),
            nn.ConvTranspose2d(8, 1, 2, stride=2), nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x


# Preparing the  Data

transform = transforms.Compose([transforms.ToTensor()])

dataset = SegmentationDataset("dataset/Q1a/images", "dataset/Q1a/masks", transform)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_data, test_data = torch.utils.data.random_split(dataset, [train_size,
test_size])
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)


# Training the Model

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleSegNet().to(device)
criterion = nn.BCELoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)

EPOCHS = 5
for epoch in range(EPOCHS):
    model.train()
    running_loss = 0
    for images, masks in train_loader:
        images, masks = images.to(device), masks.to(device)
        preds = model(images)
        loss = criterion(preds, masks)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(train_loader):.4f}")


model.eval()
ious = []

with torch.no_grad():
    for images, masks in test_loader:
        images, masks = images.to(device), masks.to(device)
        preds = model(images)
        preds = (preds > 0.5).float()
        for p, m in zip(preds, masks):
            iou = jaccard_score(m.cpu().view(-1).numpy(), p.cpu().view(-1).numpy(),
average='binary')
            ious.append(iou)

print(f"Mean IoU on Test Set: {np.mean(ious):.4f}")
```

Result:

```
Epoch 1, Loss: 0.2206
Epoch 2, Loss: 0.0588
Epoch 3, Loss: 0.0489
Epoch 4, Loss: 0.0450
Epoch 5, Loss: 0.0427
Mean IoU on Test Set: 0.9822
```

Ques03 :

Approach:

Dataset Construction:

- **Uses circle masks from Q1b as both image and mask inputs.**
- **Loads digit class labels (0–9) from corresponding `.txt` files.**
- **Applies `ToTensor()` transform.**

Model Architecture – `CirclizationCNN`:

- **Shared encoder for feature extraction.**
- **Two parallel heads:**
  - **Classifier (digit class): Fully connected layer (output = 10).**
  - **Decoder: Transposed convolution layers to predict circular masks (output = 28×28).**

Training Objective:

- **Loss = CrossEntropyLoss (classification) + BCELoss (mask reconstruction).**
- **Trained for 5 epochs using Adam optimizer.**

Evaluation Strategy:

- IoU (Intersection over Union) for predicted vs. actual circle masks.
- IoU only included if class prediction is correct, otherwise counted as `0.0`.
- Final metric: Mean IoU across test set.

```
import os
import numpy as np
from PIL import Image
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torch.optim as optim
import matplotlib.pyplot as plt

# Dataset class for Q1b
class CircleMaskClassificationDataset(Dataset):
    def __init__(self, image_dir, mask_dir, label_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.label_dir = label_dir
        self.transform = transform
        self.filenames = sorted(os.listdir(image_dir), key=lambda x:
int(x.split('.')[0]))

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        img_name = self.filenames[idx]
        image = Image.open(os.path.join(self.image_dir, img_name)).convert("L")
        mask = Image.open(os.path.join(self.mask_dir, img_name)).convert("L")
        label_path = os.path.join(self.label_dir, img_name.replace(".png", ".txt"))
        with open(label_path, "r") as f:
            label = int(f.read())

        if self.transform:
            image = self.transform(image)
            mask = self.transform(mask)
```

```
        return image, mask, label

# Paths (ensure Q1b created dataset in these dirs)
image_dir = "dataset/Q1b/images"
mask_dir = "dataset/Q1b/images"  # using same image for circlization masks
label_dir = "dataset/Q1b/labels"

# Transforms
transform = transforms.Compose([
    transforms.ToTensor(),
])

# Dataset and Dataloader
dataset = CircleMaskClassificationDataset(image_dir, mask_dir, label_dir,
transform=transform)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_ds, test_ds = torch.utils.data.random_split(dataset, [train_size, test_size])
train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=1)

# Model
class CirclizationCNN(nn.Module):
    def __init__(self):
        super(CirclizationCNN, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),  # 14x14
            nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),  # 7x7
        )
        self.fc = nn.Linear(32 * 7 * 7, 10)  # 10 classes

        # Decoder for mask prediction
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2),  # 14x14
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, kernel_size=2, stride=2),  # 28x28
            nn.Sigmoid()
        )

    def forward(self, x):
        x_enc = self.encoder(x)
        x_flat = x_enc.view(x.size(0), -1)
```

```
        class_out = self.fc(x_flat)
        mask_out = self.decoder(x_enc)
        return class_out, mask_out

# IoU Metric
def compute_iou(pred_mask, true_mask):
    pred_mask = (pred_mask > 0.5).float()
    intersection = (pred_mask * true_mask).sum()
    union = ((pred_mask + true_mask) >= 1).float().sum()
    return intersection / union if union != 0 else 0.0

# Training setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CirclizationCNN().to(device)
criterion_class = nn.CrossEntropyLoss()
criterion_mask = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(5):
    model.train()
    total_loss = 0
    for img, mask, label in train_loader:
        img, mask, label = img.to(device), mask.to(device), label.to(device)
        optimizer.zero_grad()
        out_class, out_mask = model(img)
        loss_class = criterion_class(out_class, label)
        loss_mask = criterion_mask(out_mask, mask)
        loss = loss_class + loss_mask
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss / len(train_loader):.4f}")

# Evaluation
# Evaluation
model.eval()
iou_scores = []
with torch.no_grad():
    for img, mask, label in test_loader:
        img, mask, label = img.to(device), mask.to(device), label.to(device)
        out_class, out_mask = model(img)
        pred_class = torch.argmax(out_class, dim=1)
        if pred_class == label:
            iou = compute_iou(out_mask.squeeze(), mask.squeeze())
```

```
            iou_scores.append(iou.item())  # tensor -> float
        else:
            iou_scores.append(0.0)  # already float

mean_iou = np.mean(iou_scores)
print(f"\nTest Mean IoU with classification: {mean_iou:.4f}")
```

Result:

Epoch 1, Loss: 0.6963
Epoch 2, Loss: 0.5320
Epoch 3, Loss: 0.5106
Epoch 4, Loss: 0.4965
Epoch 5, Loss: 0.4897

Test Mean IoU with classification: 0.9817

Ques04:

Implementation Details:

- **Dataset:**
    - **Images and corresponding binary masks are loaded from `dataset/Q1c/images` and `dataset/Q1c/masks`.**
    - **`ToTensor()` is applied to convert them into PyTorch tensors.**
- **Model – `SimpleUNet`:**
    - **A lightweight U-Net with:**
        - **Encoder: 2 convolutional layers with ReLU and MaxPooling.**

        - **Decoder: 2 transpose convolutional layers ending with a Sigmoid activation.**

    - **Input/output image size: 28×28 pixels, grayscale.**

- **Loss Function: Binary Cross Entropy (`BCELoss`) between predicted and true masks.**

- **Optimizer: Adam with learning rate `0.001`.**

```
import os
import numpy as np
from PIL import Image
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torch.optim as optim

# Dice Coefficient Function
def dice_coeff(pred, target, epsilon=1e-6):
    pred = (pred > 0.5).float()
    target = (target > 0.5).float()
    intersection = (pred * target).sum()
    union = pred.sum() + target.sum()
    dice = (2. * intersection + epsilon) / (union + epsilon)
    return dice.item()

# Custom Dataset
class SegmentationDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.filenames = sorted(os.listdir(image_dir), key=lambda x:
int(os.path.splitext(x)[0]))
        self.transform = transform

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

```
    def __getitem__(self, idx):
        image_path = os.path.join(self.image_dir, self.filenames[idx])
        mask_path = os.path.join(self.mask_dir, self.filenames[idx])
        img = Image.open(image_path).convert("L")
        mask = Image.open(mask_path).convert("L")

        if self.transform:
            img = self.transform(img)
            mask = self.transform(mask)

        return img, mask

# Simple U-Net
class SimpleUNet(nn.Module):
    def __init__(self):
        super(SimpleUNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 2, stride=2), nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 2, stride=2), nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Paths
image_dir = "dataset/Q1c/images"
mask_dir = "dataset/Q1c/masks"

# Transforms
transform = transforms.Compose([
    transforms.ToTensor(),
])

# Dataset and Dataloaders
dataset = SegmentationDataset(image_dir, mask_dir, transform=transform)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size,
test_size])
```

```
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1)

# Model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleUNet().to(device)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training
for epoch in range(5):  # Increase this for better training
    model.train()
    total_loss = 0
    for img, mask in train_loader:
        img, mask = img.to(device), mask.to(device)
        optimizer.zero_grad()
        pred = model(img)
        loss = criterion(pred, mask)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1} Loss: {total_loss/len(train_loader):.4f}")

# Evaluation
model.eval()
dice_scores = []
with torch.no_grad():
    for img, mask in test_loader:
        img, mask = img.to(device), mask.to(device)
        pred = model(img)
        dice = dice_coeff(pred, mask)
        dice_scores.append(dice)

print(f"\n Mean Dice Coefficient on Test Set: {np.mean(dice_scores):.4f}")
```

Result:

Epoch 1 Loss: 0.6498
Epoch 2 Loss: 0.5240
Epoch 3 Loss: 0.3501
Epoch 4 Loss: 0.2276
Epoch 5 Loss: 0.1852

🧪 **Mean Dice Coefficient on Test Set: 0.9765**

Ques05:

Implementation Details:

- **Tools: OpenCV, NumPy**
- **Method:**
    1. **Background Subtraction using `cv2.createBackgroundSubtractorMOG2()` to isolate moving foreground (i.e., the walking person).**
    2. **Thresholding & Morphological Filtering to refine the binary foreground mask.**
    3. **Masking & Merging:**
        - **Foreground from the original frame is extracted using the binary mask.**
        - **Background from the replacement image is extracted using the inverse mask.**
        - **Combined via `cv2.add()` to produce the final composited frame.**

⚙️ **Steps:**

- **Video and background image resized to 512×512.**
- **Foreground extracted and composited per frame.**
- **Output video saved using `cv2.VideoWriter`.**

```python
import cv2
import numpy as np

# Paths
video_path = "/Users/shyampremi/Desktop/denis_walk.avi"
bg_image_path = "/Users/shyampremi/Desktop/bg.png"
output_path = "/Users/shyampremi/Desktop/output_replaced.avi"

# Load background image
bg_img = cv2.imread(bg_image_path)
bg_img = cv2.resize(bg_img, (512, 512))

# Initialize video
cap = cv2.VideoCapture(video_path)
fourcc = cv2.VideoWriter_fourcc(*'XVID')
fps = int(cap.get(cv2.CAP_PROP_FPS))
frame_width = int(cap.get(3))
frame_height = int(cap.get(4))

out = cv2.VideoWriter(output_path, fourcc, fps, (frame_width, frame_height))

# Background subtractor
fgbg = cv2.createBackgroundSubtractorMOG2(detectShadows=True)

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Resize background to match frame
    bg_resized = cv2.resize(bg_img, (frame.shape[1], frame.shape[0]))

    # Apply background subtraction
    fgmask = fgbg.apply(frame)

    # Clean the mask
    _, mask = cv2.threshold(fgmask, 200, 255, cv2.THRESH_BINARY)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((3,3), np.uint8))
    mask = cv2.medianBlur(mask, 5)

    # Create inverse mask
    mask_inv = cv2.bitwise_not(mask)

    # Extract foreground and background
```

```
    fg = cv2.bitwise_and(frame, frame, mask=mask)
    bg = cv2.bitwise_and(bg_resized, bg_resized, mask=mask_inv)

    # Combine both
    combined = cv2.add(fg, bg)

    out.write(combined)

# Release resources
cap.release()
out.release()
cv2.destroyAllWindows()

print("✅ Background replaced video saved to:", output_path)
```