EE655: Computer Vision & Deep Learning Assignment II

Written by: Tanvi Pooranmal Meena, 231085

1 Perform the following tasks on the MNIST dataset to build three new datasets:

(a) Obtain foreground segmentation masks for the images in the MNIST dataset using the Otsu thresholding method. This will provide rough ground truth masks required to build a new foreground segmentation dataset.

```
    import cv2
import numpy as np
from pathlib import Path

def generate_otsu_masks(input_dir, output_dir):
    input_dir = Path(input_dir)
    output_dir = Path(output_dir)

    for class_dir in input_dir.glob("*"):
        if class_dir.is_dir():
            output_class_dir = output_dir / class_dir.name
            output_class_dir.mkdir(parents=True, exist_ok=True)

            for image_path in class_dir.glob("*.jpg"):
                # Read grayscale image
                img = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
                # Apply Gaussian blur
                blur = cv2.GaussianBlur(img, (5, 5), 0)
                # Otsu's thresholding
                _, mask = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.
                                        THRESH_OTSU)

                # Save the mask
                output_path = output_class_dir / image_path.name
                cv2.imwrite(str(output_path), mask)

# Usage:
generate_otsu_masks(
    Path("MNIST_Dataset_JPG_format/MNIST_JPG_training"),
    Path("MNIST_Masks_Training")
)

generate_otsu_masks(
    Path("MNIST_Dataset_JPG_format/MNIST_JPG_testing"),
    Path("MNIST_Masks_Testing")
)
```

OUTPUT:

Two datasets (MNIST_Masks_Testing & MNIST_Masks_Training) available on Github.

(b) Obtain tight ground truth circles around the foreground segmentation masks obtained in (a).
This will create a new dataset of 10 classes for performing classification with circlization (circular localization). You may use existing libraries to generate the tight circles

```
    import cv2
import numpy as np
from pathlib import Path

def generate_circle_masks(mask_dir, output_dir):
    mask_dir = Path(mask_dir)
    output_dir = Path(output_dir)

    for class_dir in mask_dir.glob("*"):
        if class_dir.is_dir():
            output_class_dir = output_dir / class_dir.name
            output_class_dir.mkdir(parents=True, exist_ok=True)

            for mask_path in class_dir.glob("*.jpg"):
                mask = cv2.imread(str(mask_path), cv2.IMREAD_GRAYSCALE)
                contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.
                                               CHAIN_APPROX_SIMPLE)

                circle_mask = np.zeros_like(mask)
                if contours:
                    # Use largest contour
                    largest = max(contours, key=cv2.contourArea)
                    (x, y), radius = cv2.minEnclosingCircle(largest)
                    center = (int(x), int(y))
                    radius = int(radius)
                    cv2.circle(circle_mask, center, radius, 255, -1)

                output_path = output_class_dir / mask_path.name
                cv2.imwrite(str(output_path), circle_mask)

# Usage:
generate_circle_masks(
    Path("MNIST_Masks_Training"),
    Path("MNIST_CircleMasks_Training")
)

generate_circle_masks(
    Path("MNIST_Masks_Testing"),
    Path("MNIST_CircleMasks_Testing")
)
```

OUTPUT:

Two datasets (MNIST_CircleMasks_Testing & MNIST_CircleMasks_Training) available on Github.

(c) Spatially concatenate randomly chosen 4 images and their corresponding ground truths obtained in (a) in a 2x2 manner to develop new images and semantic segmentation ground truths, respectively. This will result in a new dataset of 10 classes for performing semantic segmentation.

```
    import cv2
import numpy as np
import random
from pathlib import Path

def generate_concat_2x2(image_dir, mask_dir, output_image_dir, output_mask_dir,
                                            num_samples=1000):
    image_dir = Path(image_dir)
    mask_dir = Path(mask_dir)
    output_image_dir = Path(output_image_dir)
    output_mask_dir = Path(output_mask_dir)
    output_image_dir.mkdir(parents=True, exist_ok=True)
    output_mask_dir.mkdir(parents=True, exist_ok=True)

    classes = [str(i) for i in range(10)]
    image_paths = {cls: list((image_dir / cls).glob("*.jpg")) for cls in classes}
    mask_paths = {cls: list((mask_dir / cls).glob("*.jpg")) for cls in classes}

    for i in range(num_samples):
        chosen_classes = random.choices(classes, k=4)
        imgs, masks = [], []

        for cls in chosen_classes:
            idx = random.randint(0, len(image_paths[cls]) - 1)
            img = cv2.imread(str(image_paths[cls][idx]), cv2.IMREAD_GRAYSCALE)
            msk = cv2.imread(str(mask_paths[cls][idx]), cv2.IMREAD_GRAYSCALE)
            imgs.append(img)
            masks.append(msk)

        top_img = np.hstack((imgs[0], imgs[1]))
        bottom_img = np.hstack((imgs[2], imgs[3]))
        full_img = np.vstack((top_img, bottom_img))

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

        cv2.imwrite(str(output_image_dir / f"{i:05d}.jpg"), full_img)
        cv2.imwrite(str(output_mask_dir / f"{i:05d}_mask.jpg"), full_mask)

# Usage:
generate_concat_2x2(
    image_dir=Path("MNIST_Dataset_JPG_format/MNIST_JPG_training"),
    mask_dir=Path("MNIST_Masks_Training"),
    output_image_dir=Path("MNIST_Concat2x2_Images"),
    output_mask_dir=Path("MNIST_Concat2x2_Masks"),
    num_samples=1000
)
```

OUTPUT:

Two datasets (MNIST_Concat2×2_Images & MNIST_Concat2×2_Masks) available on Github.

2 Train a DL network from scratch to perform foreground extraction on the new dataset obtained in Q1(a). Report your test performance using the IoU metric. [2 Marks]

```
    import os
import cv2
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torch
import torch.nn as nn
import torch.nn.functional as F
from tqdm import tqdm
from sklearn.metrics import jaccard_score
import matplotlib.pyplot as plt

# ------------------------------
# Dataset
# ------------------------------
class MNISTSegmentationDataset(Dataset):
    def __init__(self, img_dir, mask_dir, transform=None):
        self.img_paths = []
        self.mask_paths = []
        for label in os.listdir(img_dir):
            img_subdir = os.path.join(img_dir, label)
            mask_subdir = os.path.join(mask_dir, label)
            for fname in os.listdir(img_subdir):
                self.img_paths.append(os.path.join(img_subdir, fname))
                self.mask_paths.append(os.path.join(mask_subdir, fname))
        self.transform = transform

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

    def __getitem__(self, idx):
        img = cv2.imread(self.img_paths[idx], cv2.IMREAD_GRAYSCALE)
        mask = cv2.imread(self.mask_paths[idx], cv2.IMREAD_GRAYSCALE)

        img = img.astype(np.float32) / 255.0
        mask = (mask > 0).astype(np.float32)

        img = np.expand_dims(img, axis=0)
        mask = np.expand_dims(mask, axis=0)

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

        return torch.tensor(img, dtype=torch.float32), torch.tensor(mask, dtype=
                                        torch.float32)

# ------------------------------
# U-Net-like CNN (minimal)
# ------------------------------
class SimpleSegNet(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(),
            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.Conv2d(16, 1, 1),
            nn.Sigmoid()
        )

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

# ------------------------------
# IoU Metric
# ------------------------------
def compute_iou(preds, masks):
    preds_bin = (preds > 0.5).float()
    ious = []
    for p, m in zip(preds_bin, masks):
        p = p.view(-1).cpu().numpy()
        m = m.view(-1).cpu().numpy()
        if np.sum(m) == 0 and np.sum(p) == 0:
            ious.append(1.0)
        elif np.sum(m) == 0 or np.sum(p) == 0:
            ious.append(0.0)
        else:
            ious.append(jaccard_score(m, p))
    return np.mean(ious)

# ------------------------------
# Training and Evaluation
# ------------------------------
def train_model():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    train_set = MNISTSegmentationDataset(
        img_dir="../MNIST_Dataset_JPG_format/MNIST_JPG_training",
        mask_dir="../q1/q1a/MNIST_Masks_Training"
    )
    test_set = MNISTSegmentationDataset(
        img_dir="../MNIST_Dataset_JPG_format/MNIST_JPG_testing",
        mask_dir="../q1/q1a/MNIST_Masks_Testing"
    )

    train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_set, batch_size=64)

    model = SimpleSegNet().to(device)
    criterion = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(20):
        model.train()
        running_loss = 0.0
```

```
        for imgs, masks in tqdm(train_loader):
            imgs, masks = imgs.to(device), masks.to(device)
            optimizer.zero_grad()
            outputs = model(imgs)
            loss = criterion(outputs, masks)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}: Loss = {running_loss / len(train_loader):.4f}")

    # Evaluation
    model.eval()
    all_preds, all_masks = [], []
    with torch.no_grad():
        for imgs, masks in test_loader:
            imgs = imgs.to(device)
            preds = model(imgs).cpu()
            all_preds.append(preds)
            all_masks.append(masks)
    preds_cat = torch.cat(all_preds)
    masks_cat = torch.cat(all_masks)

    iou = compute_iou(preds_cat, masks_cat)
    print(f"Test IoU: {iou:.4f}")

if __name__ == "__main__":
    train_model()
```

OUTPUT:

```
Epoch 1: Loss = 0.3643
Epoch 2: Loss = 0.3025
Epoch 3: Loss = 0.2985
Epoch 4: Loss = 0.2968
Epoch 5: Loss = 0.2958
Epoch 6: Loss = 0.2952
Epoch 7: Loss = 0.2947
Epoch 8: Loss = 0.2943
Epoch 9: Loss = 0.2940
Epoch 10: Loss = 0.2938
Epoch 11: Loss = 0.2935
Epoch 12: Loss = 0.2933
Epoch 13: Loss = 0.2931
Epoch 14: Loss = 0.2930
Epoch 15: Loss = 0.2928
Epoch 16: Loss = 0.2927
Epoch 17: Loss = 0.2926
Epoch 18: Loss = 0.2925
Epoch 19: Loss = 0.2924
Epoch 20: Loss = 0.2923
Test IoU: 0.5439
```

Code Listing 1: bash version

3 Train a DL network from scratch to perform classification with circlization on the new dataset obtained in Q1(b). Report your test performance using the IoU metric. [2 Marks]
Note: If the classification is wrong, consider the IoU metric score as zero.

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

# ==== Step 1: Custom Dataset ====
class CircleMNISTDataset(Dataset):
    def __init__(self, image_root, circle_mask_root, transform=None):
        self.image_paths = []
        self.labels = []
        self.circles = []
        self.transform = transform

        for digit in range(10):
            digit_path = os.path.join(image_root, str(digit))
            mask_path = os.path.join(circle_mask_root, str(digit))
            for filename in os.listdir(digit_path):
                if filename.endswith('.jpg'):
                    img_path = os.path.join(digit_path, filename)
                    mask_img = Image.open(os.path.join(mask_path, filename)).
                                        convert('L')
                    np_mask = np.array(mask_img)

                    # Extract circle from white pixels
                    y, x = np.where(np_mask > 0)
                    if len(x) == 0 or len(y) == 0:
                        continue
                    xc = (x.min() + x.max()) / 2 / 28
                    yc = (y.min() + y.max()) / 2 / 28
                    r = max((x.max() - x.min()), (y.max() - y.min())) / 2 / 28

                    self.image_paths.append(img_path)
                    self.labels.append(digit)
                    self.circles.append([xc, yc, r])

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

    def __getitem__(self, idx):
        img = Image.open(self.image_paths[idx]).convert('L')
        label = self.labels[idx]
        circle = torch.tensor(self.circles[idx], dtype=torch.float32)

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

```
        return img, label, circle

# ==== Step 2: Network ====
class CircleNet(nn.Module):
    def __init__(self):
        super(CircleNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        )
        self.flatten = nn.Flatten()
        self.classifier = nn.Sequential(
            nn.Linear(64*7*7, 128), nn.ReLU(), nn.Linear(128, 10)
        )
        self.circle_regressor = nn.Sequential(
            nn.Linear(64*7*7, 128), nn.ReLU(), nn.Linear(128, 3), nn.Sigmoid()
        )

    def forward(self, x):
        x = self.features(x)
        x_flat = self.flatten(x)
        class_logits = self.classifier(x_flat)
        circle_pred = self.circle_regressor(x_flat)
        return class_logits, circle_pred

# ==== Step 3: IoU Function ====
def circle_iou(circle1, circle2):
    x1, y1, r1 = circle1
    x2, y2, r2 = circle2
    d = math.sqrt((x1 - x2)**2 + (y1 - y2)**2)
    if d >= r1 + r2:
        return 0.0
    if d <= abs(r1 - r2):
        return (min(r1, r2)**2) / (max(r1, r2)**2)
    r1_sq = r1 ** 2
    r2_sq = r2 ** 2
    part1 = r1_sq * math.acos((d**2 + r1_sq - r2_sq) / (2 * d * r1))
    part2 = r2_sq * math.acos((d**2 + r2_sq - r1_sq) / (2 * d * r2))
    part3 = 0.5 * math.sqrt((-d + r1 + r2) * (d + r1 - r2) * (d - r1 + r2) * (d +
                                        r1 + r2))
    inter = part1 + part2 - part3
    union = math.pi * (r1_sq + r2_sq) - inter
    return inter / union

# ==== Step 4: Evaluation ====
def evaluate(model, dataloader, device):
    model.eval()
    total_iou = 0.0
    total_samples = 0
    with torch.no_grad():
        for images, labels, circles in dataloader:
            images = images.to(device)
            labels = labels.to(device)
            circles = circles.to(device)

            class_logits, circle_preds = model(images)
            pred_classes = torch.argmax(class_logits, dim=1)

            for i in range(images.size(0)):
```

```
                pred_class = pred_classes[i].item()
                true_class = labels[i].item()
                pred_circle = circle_preds[i].cpu().numpy()
                true_circle = circles[i].cpu().numpy()

                if pred_class == true_class:
                    iou = circle_iou(pred_circle, true_circle)
                else:
                    iou = 0.0
                total_iou += iou
                total_samples += 1
    mean_iou = total_iou / total_samples
    print(f"\n      Mean IoU on Test Set: {mean_iou:.4f}")
    return mean_iou

# ==== Step 5: Train & Run ====
def train_and_eval():
    train_root = "MNIST_Dataset_JPG_format/MNIST_JPG_training"
    test_root = "MNIST_Dataset_JPG_format/MNIST_JPG_testing"
    circle_train = "q1/q1b/MNIST_CircleMasks_Training"
    circle_test = "q1/q1b/MNIST_CircleMasks_Testing"

    transform = transforms.Compose([
        transforms.Resize((28, 28)),
        transforms.ToTensor()
    ])

    train_set = CircleMNISTDataset(train_root, circle_train, transform)
    test_set = CircleMNISTDataset(test_root, circle_test, transform)

    train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_set, batch_size=64)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = CircleNet().to(device)

    criterion_class = nn.CrossEntropyLoss()
    criterion_circle = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(1, 11):
        model.train()
        total_loss = 0.0
        for images, labels, circles in train_loader:
            images, labels, circles = images.to(device), labels.to(device),
                                      circles.to(device)
            optimizer.zero_grad()
            class_logits, circle_preds = model(images)
            loss_class = criterion_class(class_logits, labels)
            loss_circle = criterion_circle(circle_preds, circles)
            loss = loss_class + loss_circle
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch}: Loss = {total_loss:.4f}")

    evaluate(model, test_loader, device)

if __name__ == "__main__":
```

```
    train_and_eval()
```

OUTPUT:

```
Epoch 1: Loss = 162.4205
Epoch 2: Loss = 48.5278
Epoch 3: Loss = 33.3905
Epoch 4: Loss = 23.5051
Epoch 5: Loss = 20.5727
Epoch 6: Loss = 14.8037
Epoch 7: Loss = 12.4969
Epoch 8: Loss = 9.8469
Epoch 9: Loss = 9.3188
Epoch 10: Loss = 6.9169

Mean IoU on Test Set: 0.9698
```

Code Listing 2: bash version

4 Train a DL network from scratch to perform semantic segmentation on the new dataset obtained in Q1(c). Report your test performance using the Dice Coefficient. [2 Marks]

```
    import os
import cv2
import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torch.nn.functional as F
import torch.nn as nn
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import matplotlib.pyplot as plt

# --- Constants ---
IMAGE_DIR = 'q1/q1c/MNIST_Concat2x2_Images'
MASK_DIR = 'q1/q1c/MNIST_Concat2x2_Masks'
IMG_SIZE = 112  # since 28x2 = 56, and 2x = 112
BATCH_SIZE = 16
EPOCHS = 15
NUM_CLASSES = 10
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Dataset ---
class MNISTSegmentationDataset(Dataset):
    def __init__(self, image_paths, mask_paths, transform=None):
        self.image_paths = image_paths
        self.mask_paths = mask_paths
        self.transform = transform

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

    def __getitem__(self, idx):
        img = cv2.imread(self.image_paths[idx], cv2.IMREAD_GRAYSCALE)
        mask = cv2.imread(self.mask_paths[idx], cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
        mask = cv2.resize(mask, (IMG_SIZE, IMG_SIZE), interpolation=cv2.
                                        INTER_NEAREST)

        img = img.astype(np.float32) / 255.0
        mask = mask.astype(np.int64)

        mask = mask // 255

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

        return torch.from_numpy(img).unsqueeze(0), torch.from_numpy(mask)

# --- Model: Simple U-Net ---
class UNet(nn.Module):
    def __init__(self, num_classes=10):
        super(UNet, self).__init__()
        def CBR(in_c, out_c): return nn.Sequential(
```

```
            nn.Conv2d(in_c, out_c, 3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(out_c, out_c, 3, padding=1), nn.ReLU(inplace=True)
        )

        self.enc1 = CBR(1, 64)
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = CBR(64, 128)
        self.pool2 = nn.MaxPool2d(2)

        self.bottleneck = CBR(128, 256)

        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = CBR(256, 128)
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = CBR(128, 64)

        self.out_conv = nn.Conv2d(64, num_classes, 1)

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool1(e1))
        b = self.bottleneck(self.pool2(e2))
        d2 = self.dec2(torch.cat([self.up2(b), e2], dim=1))
        d1 = self.dec1(torch.cat([self.up1(d2), e1], dim=1))
        return self.out_conv(d1)

# --- Dice Coefficient Metric ---
def dice_coef(pred, target, smooth=1.0):
    pred = F.one_hot(pred, NUM_CLASSES).permute(0, 3, 1, 2).float()
    target = F.one_hot(target, NUM_CLASSES).permute(0, 3, 1, 2).float()
    intersection = (pred * target).sum(dim=(2,3))
    union = pred.sum(dim=(2,3)) + target.sum(dim=(2,3))
    dice = (2. * intersection + smooth) / (union + smooth)
    return dice.mean().item()

# --- Load Data ---
img_files = sorted([os.path.join(IMAGE_DIR, f) for f in os.listdir(IMAGE_DIR)])
mask_files = sorted([os.path.join(MASK_DIR, f) for f in os.listdir(MASK_DIR)])

train_imgs, test_imgs, train_masks, test_masks = train_test_split(img_files,
                                        mask_files, test_size=0.2, random_state=
                                        42)

train_ds = MNISTSegmentationDataset(train_imgs, train_masks)
test_ds = MNISTSegmentationDataset(test_imgs, test_masks)

train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=1)

# --- Training ---
model = UNet(num_classes=NUM_CLASSES).to(DEVICE)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

for epoch in range(EPOCHS):
    model.train()
    total_loss = 0
    for imgs, masks in tqdm(train_loader, desc=f"Epoch {epoch+1}/{EPOCHS}"):
        imgs, masks = imgs.to(DEVICE), masks.to(DEVICE)
```

```
        preds = model(imgs)
        loss = criterion(preds, masks)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Train Loss: {total_loss / len(train_loader):.4f}")

# --- Evaluation ---
model.eval()
dice_scores = []
with torch.no_grad():
    for imgs, masks in test_loader:
        imgs, masks = imgs.to(DEVICE), masks.to(DEVICE)
        preds = model(imgs)
        preds = torch.argmax(preds, dim=1)
        dice_scores.append(dice_coef(preds.cpu(), masks.cpu()))

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

```
Epoch 1/15:
Train Loss: 0.5083

Epoch 2/15:
Train Loss: 0.1482

Epoch 3/15:
Train Loss: 0.1425

Epoch 4/15:
Train Loss: 0.1409

Epoch 5/15:
Train Loss: 0.1393

Epoch 6/15:
Train Loss: 0.1372

Epoch 7/15:
Train Loss: 0.1360

Epoch 8/15:
Train Loss: 0.1352

Epoch 9/15:
Train Loss: 0.1349

Epoch 10/15:
Train Loss: 0.1343

Epoch 11/15:
Train Loss: 0.1340

Epoch 12/15:
Train Loss: 0.1336

Epoch 13/15:
Train Loss: 0.1336
```

```
Epoch 14/15:
Train Loss: 0.1332

Epoch 15/15:
Train Loss: 0.1334

Test Dice Coefficient: 0.9474
```

Code Listing 3: bash version

5 Apply one of the background subtraction algorithms to the given video and use appropriate post-processing techniques to generate a new video with the provided image as the background. The entire process should be implemented using a program. [2 Marks]

```
    import cv2
import numpy as np
from tqdm import tqdm

# Paths
video_path = 'q5/denis_walk.avi'
background_image_path = 'q5/bg.png'
output_path = 'q5/output_with_new_bg.avi'

# Load and resize background image
bg_image = cv2.imread(background_image_path)

# Load video
cap = cv2.VideoCapture(video_path)
frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
frame_w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps = cap.get(cv2.CAP_PROP_FPS)

bg_image = cv2.resize(bg_image, (frame_w, frame_h))

# Estimate static background using average of initial N frames
N = 30
avg_bg = np.zeros((frame_h, frame_w, 3), dtype=np.float32)

for _ in range(N):
    ret, frame = cap.read()
    if not ret:
        break
    avg_bg += frame.astype(np.float32)

avg_bg /= N
avg_bg = avg_bg.astype(np.uint8)

# Reset video to start
cap.set(cv2.CAP_PROP_POS_FRAMES, 0)

# Setup video writer
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter(output_path, fourcc, fps, (frame_w, frame_h))

# Morphology kernel
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
dilate_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))

# Process video
for _ in tqdm(range(frame_count)):
    ret, frame = cap.read()
    if not ret:
        break
```

```
    # Step 1: Absolute difference from average background
    diff = cv2.absdiff(frame, avg_bg)
    gray = cv2.cvtColor(diff, cv2.COLOR_BGR2GRAY)

    # Step 2: Otsu's threshold
    _, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    # Step 3: Clean + THICKEN mask (this solves shredding)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
    mask = cv2.dilate(mask, dilate_kernel, iterations=1)

    # Convert to 3 channel
    mask_3ch = cv2.merge([mask] * 3)
    inv_mask_3ch = cv2.merge([cv2.bitwise_not(mask)] * 3)

    # Foreground and background parts
    fg = cv2.bitwise_and(frame, mask_3ch)
    bg = cv2.bitwise_and(bg_image, inv_mask_3ch)
    final = cv2.add(fg, bg)

    out.write(final)

cap.release()
out.release()
print("    Output saved to:", output_path)
```

OUTPUT:

The generated video is attached with the submitted zip file.