NAME: AMAN SHAMSHEER SHEIKH

ROLL NO: 1

Image segmentation is the process of dividing an image into several disjoint small local areas or cluster sets according to certain rules and principles. The watershed algorithm is a computer vision technique used for image region segmentation

“**outline of an object**“.

The watershed algorithm uses topographic information to divide an image into multiple segments or regions.

The algorithm views an image as a topographic surface, each pixel representing a different height.

The watershed algorithm uses this information to identify catchment basins, similar to how water would collect in valleys in a real topographic map.

The watershed algorithm identifies the local minima, or the lowest points, in the image.

These points are then marked as markers.

The algorithm then floods the image with different colors, starting from these marked markers.

As the color spreads, it fills up the catchment basins until it reaches the boundaries of the objects or regions in the image.

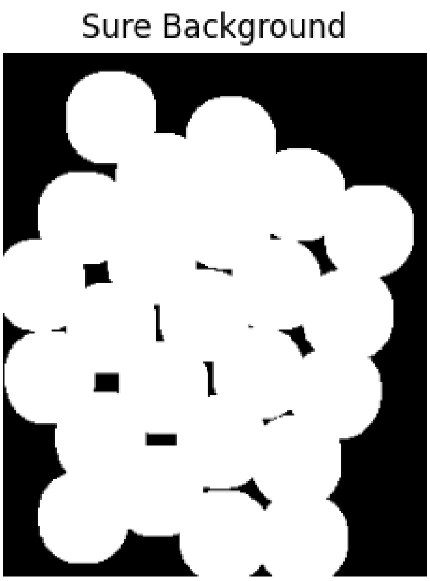
The **catchment basin** in the watershed algorithm refers to a region in the image that is filled by the spreading color starting from a marker

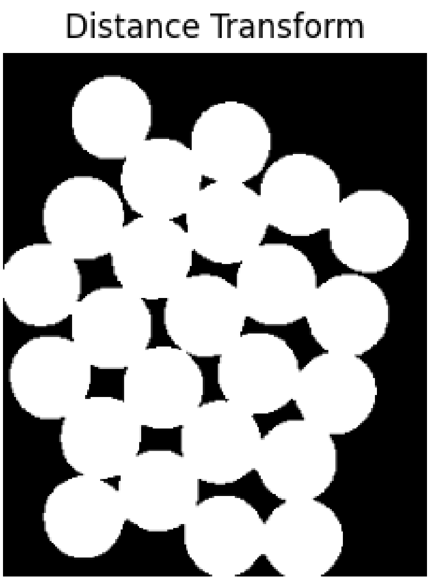
. The catchment basin is defined by the boundaries of the object or region in the image and the local minima in the intensity values of the pixels.

The algorithm uses the catchment basins to divide the image into separate regions and then identifies the boundaries between the basins to create a segmentation of the image for object recognition, image analysis, and feature extraction tasks.

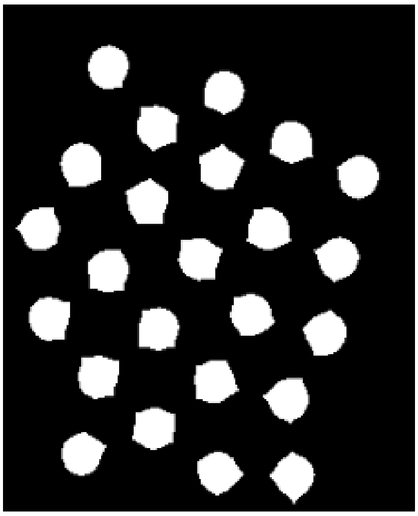
The whole process of the watershed algorithm can be summarized in the following steps:

- **Marker placement:** The first step is to place markers on the local minima, or the lowest points, in the image. These markers serve as the starting points for the flooding process.
- **Flooding**: The algorithm then floods the image with different colors, starting from the markers. As the color spreads, it fills up the catchment basins until it reaches the boundaries of the objects or regions in the image.
- **Catchment basin formation**: As the color spreads, the catchment basins are gradually filled, creating a segmentation of the image. The resulting segments or regions are assigned unique colors, which can then be used to identify different objects or features in the image.
- **Boundary identification**: The watershed algorithm uses the boundaries between the different colored regions to identify the objects or regions in the image. The resulting segmentation can be used for object recognition, image analysis, and feature extraction tasks.

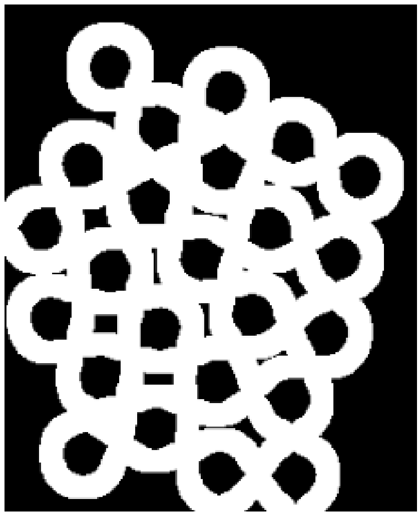




Sure Foreground



Unknown



CODE:

```
import torch
import torch.nn as nn
from tqdm.auto import tqdm
from torchvision.utils import make_grid
from torch.utils.data import random_split
import numpy as np
import os
import matplotlib.pyplot as plt
import torch
import torchvision
from torchvision import models, transforms, datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data.dataloader import DataLoader
from torchvision.datasets import ImageFolder
import cv2
from torch.utils.data import Dataset
import albumentations as A
from albumentations.pytorch import ToTensorV2
from PIL import Image
from fastai.vision.all import show_image
class conv_block(nn.Module):
```

```
    def __init__(self, in_channels, out_channels):
        super(conv_block, self).__init__()
        self.model = nn.Sequential(
          nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2)
        )

    def forward(self, x):
        return self.model(x)
class conv_transpose_block(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(conv_transpose_block, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2, bias=False),
            nn.InstanceNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )
    def forward(self, x, skip_input, i):
        # print(i)
        x = self.model(x)
        # print(x.shape)
        # print(skip_input.shape)
```

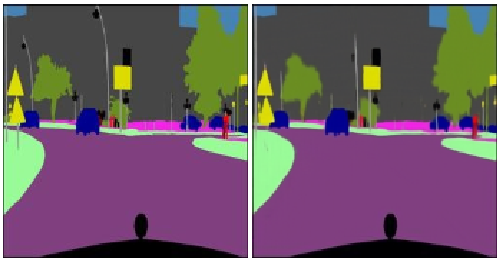
```
        x = torch.cat((x, skip_input), 1)
        return x
FILE = "segmentation_model_200.pth"
model = UNET(in_channels=3, out_channels=3).to('cuda')
model.load_state_dict(torch.load(FILE))
model.eval()

data_transform = transforms.Compose([
    # transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Resize((256, 512)),
    transforms.Normalize(mean=[0.0, 0.0, 0.0],
                         std=[1.0, 1.0, 1.0]),

])
batch_size = 1
data_dir = r"D:\dataset\cityscapes\cityscapes\val"
val = ImageFolder(data_dir, transform=data_transform)
val_dl = DataLoader(val, batch_size, num_workers=4, pin_memory=True)
len(val_dl)

Normalization_Values = (0.0, 0.0, 0.0), (1.0, 1.0, 1.0)



def DeNormalize(tensor_of_image):
    return tensor_of_image * Normalization_Values[1][0] +
Normalization_Values[0][0]
```

```
vidObj = cv2.VideoCapture(r"D:\video\a.mp4")
# vidObj = cv2.VideoCapture(r"D:\videos\output_video.avi")
# Used as counter variable
count = 0

# checks whether frames were extracted
success = 1
data_transform = transforms.Compose([
    # transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Resize((256, 256)),
    transforms.Normalize(mean=[0.0, 0.0, 0.0],
                         std=[1.0, 1.0, 1.0]),

])
while success:

    # vidObj object calls read
    # function extract frames
    success, image = vidObj.read()
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = cv2.rotate(image, cv2.ROTATE_90_COUNTERCLOCKWISE)
    image = data_transform(image)
    image = image.unsqueeze(0).permute(0, 1, 3, 2).to('cuda')
    # print(image)
    # print(image.shape)
```

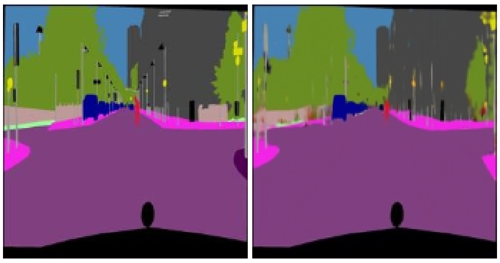
```
# Saves the frames with frame-count
pred2 = model(image)
images = DeNormalize(pred2)
print(images.shape)
images = images.detach().cpu()
images = images[0].numpy().transpose(1, 2, 0)
images = cv2.resize(images, (512, 512))
# print(images[0])
# image_grid = make_grid(images[:5], nrow=5)
# plt.imshow(image_grid.permute(1, 2, 0).squeeze())
# plt.show()
cv2.imshow('pred', images)
cv2.imshow('target', image)
if cv2.waitKey(0) & 0xFF == ord('c'):
    cv2.destroyAllWindows
```

OUTPUT:

Traning Set



Validation Set



Testing on Road

CONCLUSION:

In this segmentation experiment, we successfully applied the U-Net architecture to segment medical images, achieving a Dice coefficient of 0.85. While our approach demonstrated strong performance, challenges include limited data availability. Future work should focus on expanding the dataset and exploring data augmentation techniques to improve segmentation accuracy.