from google.colab import drive

drive.mount('/content/drive')

import os, cv2

import numpy as np

import pandas as pd

import random, tqdm

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

import warnings

warnings.filterwarnings("ignore")

import torch

import torch.nn as nn

from torch.utils.data import DataLoader

import albumentations as album

!pip install -q -U segmentation-models-pytorch albumentations > /dev/null

import segmentation\_models\_pytorch as smp

DATA\_DIR = '/content/drive/MyDrive/Dataset'

metadata\_df = pd.read\_csv(os.path.join(DATA\_DIR, 'metadata.csv'))

metadata\_df = metadata\_df[['frame\_id', 'png\_image\_path', 'png\_mask\_path']]

metadata\_df['png\_image\_path'] = metadata\_df['png\_image\_path'].apply(lambda img\_pth: os.path.join(DATA\_DIR, img\_pth))

metadata\_df['png\_mask\_path'] = metadata\_df['png\_mask\_path'].apply(lambda img\_pth: os.path.join(DATA\_DIR, img\_pth))

metadata\_df = metadata\_df.sample(frac=1).reset\_index(drop=True)

val\_df = metadata\_df.sample(frac=0.2, random\_state=42)

tra\_df = metadata\_df.drop(val\_df.index)

len(tra\_df), len(val\_df)

class\_dict = pd.read\_csv(os.path.join(DATA\_DIR, 'class\_dict.csv'))

class\_names = class\_dict['class\_names'].tolist()

class\_rgb\_values = class\_dict[['r','g','b']].values.tolist()

print('All dataset classes and their corresponding RGB values in labels:')

print('Class Names: ', class\_names)

print('Class RGB values: ', class\_rgb\_values)

select\_classes = ['background', 'polyp']

select\_class\_indices = [class\_names.index(cls.lower()) for cls in select\_classes]

select\_class\_rgb\_values =  np.array(class\_rgb\_values)[select\_class\_indices]

print('Selected classes and their corresponding RGB values in labels:')

print('Class Names: ', class\_names)

print('Class RGB values: ', class\_rgb\_values)

def visualize(\*\*images):

    n\_image = len(images)

    plt.figure(figsize=(20,8))

    for idx, (name, image) in enumerate(images.items()):

        plt.subplot(1, n\_image, idx + 1)

        plt.xticks([]);

        plt.yticks([])

        plt.title(name.replace('\_',' ').title(), fontsize=20)

        plt.imshow(image)

    plt.show()

def one\_hot\_encode(label, label\_values):

    semantic\_map = []

    for colour in label\_values:

        equality = np.equal(label, colour)

        class\_map = np.all(equality, axis = -1)

        semantic\_map.append(class\_map)

    semantic\_map = np.stack(semantic\_map, axis=-1)

    return semantic\_map

def reverse\_one\_hot(image):

    x = np.argmax(image, axis = -1)

    return x

def colour\_code\_segmentation(image, label\_values):

    colour\_codes = np.array(label\_values)

    x = colour\_codes[image.astype(int)]

    return x

def get\_bounding\_boxes(mask, target\_class\_idx=1, min\_area=10):

    """

    Extract bounding boxes for a specific class index from a mask.

    mask: numpy array (2D: HxW or 3D: HxWxC)

    target\_class\_idx: class index of the polyp

    min\_area: ignore very small detections

    """

    # If mask is one-hot encoded or has probabilities

    if mask.ndim == 3:

        mask\_gray = np.argmax(mask, axis=-1)  # Get class indices

    else:

        mask\_gray = mask

    # Convert to binary mask for target class

    binary\_mask = (mask\_gray == target\_class\_idx).astype(np.uint8) \* 255

    # Find contours

    contours, \_ = cv2.findContours(binary\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    boxes = []

    for cnt in contours:

        if cv2.contourArea(cnt) >= min\_area:  # filter tiny noise

            x, y, w, h = cv2.boundingRect(cnt)

            boxes.append((x, y, w, h))

    return boxes

def draw\_boxes(image, boxes, color=(0, 255, 0), thickness=2):

    """

    Draw rectangles on image from list of boxes.

    """

    img\_copy = image.copy()

    for (x, y, w, h) in boxes:

        cv2.rectangle(img\_copy, (x, y), (x + w, y + h), color, thickness)

    return img\_copy

class EndoscopyDataset(torch.utils.data.Dataset):

  def \_\_init\_\_(

            self,

            df,

            class\_rgb\_values=None,

            augmentation=None,

            preprocessing=None,

    ):

        self.image\_paths = df['png\_image\_path'].tolist()

        self.mask\_paths = df['png\_mask\_path'].tolist()

        self.class\_rgb\_values = class\_rgb\_values

        self.augmentation = augmentation

        self.preprocessing = preprocessing

  def \_\_getitem\_\_(self, i):

        # read images and masks

        image = cv2.cvtColor(cv2.imread(self.image\_paths[i]), cv2.COLOR\_BGR2RGB)

        mask = cv2.cvtColor(cv2.imread(self.mask\_paths[i]), cv2.COLOR\_BGR2RGB)

        # one-hot-encode the mask

        mask = one\_hot\_encode(mask, self.class\_rgb\_values).astype('float')

        # apply augmentations

        if self.augmentation:

            sample = self.augmentation(image=image, mask=mask)

            image, mask = sample['image'], sample['mask']

        # apply preprocessing

        if self.preprocessing:

            sample = self.preprocessing(image=image, mask=mask)

            image, mask = sample['image'], sample['mask']

        return image, mask

  def \_\_len\_\_(self):

        # return length of

        return len(self.image\_paths)

dataset = EndoscopyDataset(tra\_df, class\_rgb\_values=select\_class\_rgb\_values)

random\_idx = random.randint(0, len(dataset)-1)

image, mask = dataset[2]

visualize(

    original\_image = image,

    ground\_truth\_mask = colour\_code\_segmentation(reverse\_one\_hot(mask), select\_class\_rgb\_values),

    one\_hot\_encoded\_mask = reverse\_one\_hot(mask)

)

def get\_training\_augmentation():

    train\_transform = [

        album.HorizontalFlip(p=0.5),

    ]

    return album.Compose(train\_transform)

def get\_validation\_augmentation():

    test\_transform = [

        album.PadIfNeeded(min\_height=288, min\_width=384, always\_apply=True, border\_mode=0),

    ]

    return album.Compose(test\_transform)

def to\_tensor(x, \*\*kwargs):

    return x.transpose(2, 0, 1).astype('float32')

def get\_preprocessing(preprocessing\_fn=None):

    \_transform = []

    if preprocessing\_fn:

        \_transform.append(album.Lambda(image=preprocessing\_fn))

    \_transform.append(album.Lambda(image=to\_tensor, mask=to\_tensor))

    return album.Compose(\_transform)

augmented\_dataset = EndoscopyDataset(

    tra\_df,

    augmentation=get\_training\_augmentation(),

    class\_rgb\_values=select\_class\_rgb\_values,

)

random\_idx = random.randint(0, len(augmented\_dataset)-1)

# Different augmentations on image/mask pairs

for idx in range(3):

    image, mask = augmented\_dataset[idx]

    visualize(

        original\_image = image,

        ground\_truth\_mask = colour\_code\_segmentation(reverse\_one\_hot(mask), select\_class\_rgb\_values),

        one\_hot\_encoded\_mask = reverse\_one\_hot(mask)

    )

ENCODER = 'resnet50'

ENCODER\_WEIGHTS = 'imagenet'

CLASSES = select\_classes

ACTIVATION = 'sigmoid'

model = smp.DeepLabV3Plus(

    encoder\_name=ENCODER,

    encoder\_weights=ENCODER\_WEIGHTS,

    classes=len(CLASSES),

    activation=ACTIVATION,

)

preprocessing\_fn = smp.encoders.get\_preprocessing\_fn(ENCODER, ENCODER\_WEIGHTS)

train\_dataset = EndoscopyDataset(

    tra\_df,

    augmentation=get\_training\_augmentation(),

    preprocessing=get\_preprocessing(preprocessing\_fn),

    class\_rgb\_values=select\_class\_rgb\_values,

)

valid\_dataset = EndoscopyDataset(

    val\_df,

    augmentation=get\_validation\_augmentation(),

    preprocessing=get\_preprocessing(preprocessing\_fn),

    class\_rgb\_values=select\_class\_rgb\_values,

)

train\_loader = DataLoader(train\_dataset, batch\_size=16, shuffle=True, num\_workers=4)

valid\_loader = DataLoader(valid\_dataset, batch\_size=16, shuffle=False, num\_workers=4)

!pip install -U segmentation-models-pytorch

import segmentation\_models\_pytorch as smp

from segmentation\_models\_pytorch.utils.train import TrainEpoch, ValidEpoch

from segmentation\_models\_pytorch.utils.metrics import IoU

from segmentation\_models\_pytorch.utils.losses import DiceLoss

TRAINING = True

EPOCHS = 15

DEVICE = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

loss = smp.utils.losses.DiceLoss()

metrics = [smp.utils.metrics.IoU(threshold=0.5),]

optimizer = torch.optim.Adam([

    dict(params=model.parameters(), lr=0.00008),

])

lr\_scheduler = torch.optim.lr\_scheduler.CosineAnnealingWarmRestarts(

    optimizer, T\_0=1, T\_mult=2, eta\_min=5e-5,

)

train\_epoch = smp.utils.train.TrainEpoch(

    model,

    loss=loss,

    metrics=metrics,

    optimizer=optimizer,

    device=DEVICE,

    verbose=True,

)

valid\_epoch = smp.utils.train.ValidEpoch(

    model,

    loss=loss,

    metrics=metrics,

    device=DEVICE,

    verbose=True,

)

%%time

if TRAINING:

    best\_iou\_score = 0.0

    train\_logs\_list, valid\_logs\_list = [], []

    for i in range(0, EPOCHS):

        # Perform training & validation

        print('\nEpoch: {}'.format(i))

        train\_logs = train\_epoch.run(train\_loader)

        valid\_logs = valid\_epoch.run(valid\_loader)

        train\_logs\_list.append(train\_logs)

        valid\_logs\_list.append(valid\_logs)

        # Save model if a better val IoU score is obtained

        if best\_iou\_score < valid\_logs['iou\_score']:

            best\_iou\_score = valid\_logs['iou\_score']

            torch.save(model, './best\_model.pth')

            print('Model saved!')

# load best saved model checkpoint from the current run

if os.path.exists('./best\_model.pth'):

    best\_model = torch.load('./best\_model.pth', map\_location=DEVICE ,weights\_only=False)

    print('Loaded UNet model from this run.')

# load best saved model checkpoint from previous commit (if present)

elif os.path.exists('../input/polyp-segmentation-in-colonoscopy-frames-deeplab/best\_model.pth'):

    best\_model = torch.load('../input/polyp-segmentation-in-colonoscopy-frames-deeplab/best\_model.pth', map\_location=DEVICE,weights\_only=False)

    print('Loaded UNet model from a previous commit.')

test\_dataset = EndoscopyDataset(

    val\_df,

    augmentation=get\_validation\_augmentation(),

    preprocessing=get\_preprocessing(preprocessing\_fn),

    class\_rgb\_values=select\_class\_rgb\_values,

)

test\_dataloader = DataLoader(test\_dataset)

test\_dataset\_vis = EndoscopyDataset(

    val\_df,

    class\_rgb\_values=select\_class\_rgb\_values,

)

random\_idx = random.randint(0, len(test\_dataset\_vis)-1)

image, mask = test\_dataset\_vis[random\_idx]

visualize(

    original\_image = image,

    ground\_truth\_mask = colour\_code\_segmentation(reverse\_one\_hot(mask), select\_class\_rgb\_values),

    one\_hot\_encoded\_mask = reverse\_one\_hot(mask)

)

pip install -q segmentation-models-pytorch albumentations gradio grad-cam

!pip install grad-cam --quiet

from pytorch\_grad\_cam import GradCAM

from pytorch\_grad\_cam.utils.model\_targets import SemanticSegmentationTarget

from pytorch\_grad\_cam.utils.image import show\_cam\_on\_image

import torchvision.transforms.functional as TF

# Set model to eval mode

model.eval()

model.to(DEVICE)

if image.shape[-1] == 3:

    input\_tensor = torch.from\_numpy(image).permute(2, 0, 1)

else:

    input\_tensor = torch.from\_numpy(image)

input\_tensor = input\_tensor.unsqueeze(0).float()

\_, \_, h, w = input\_tensor.shape

pad\_h = (16 - h % 16) % 16

pad\_w = (16 - w % 16) % 16

input\_tensor = torch.nn.functional.pad(

    input\_tensor, (0, pad\_w, 0, pad\_h), mode='reflect'

).to(DEVICE)

rgb\_image = np.pad(image, ((0, pad\_h), (0, pad\_w), (0, 0)), mode='reflect')

rgb\_image = (rgb\_image - rgb\_image.min()) / (rgb\_image.max() - rgb\_image.min())

target\_layers = [model.encoder.layer4[-1]]

target\_mask = reverse\_one\_hot(mask).squeeze()

targets = [SemanticSegmentationTarget(1, target\_mask)]

cam = GradCAM(model=model, target\_layers=target\_layers)

grayscale\_cam = cam(input\_tensor=input\_tensor, targets=targets)[0]

if grayscale\_cam.ndim == 3 and grayscale\_cam.shape[0] == 3:

    grayscale\_cam = np.mean(grayscale\_cam, axis=0)

print("rgb\_image shape:", rgb\_image.shape)

print("grayscale\_cam shape:", grayscale\_cam.shape)

cam\_image = show\_cam\_on\_image(rgb\_image, grayscale\_cam, use\_rgb=True)

plt.figure(figsize=(12, 5))

plt.subplot(1, 3, 1)

plt.imshow(rgb\_image)

plt.title("Original Image")

plt.axis('off')

plt.subplot(1, 3, 2)

plt.imshow(target\_mask, cmap='gray')

plt.title("Ground Truth Mask")

plt.axis('off')

plt.subplot(1, 3, 3)

plt.imshow(cam\_image)

plt.title("Grad-CAM on Encoder")

plt.axis('off')

plt.tight\_layout()

plt.show()

def crop\_image(image, true\_dimensions):

    return album.CenterCrop(p=1, height=true\_dimensions[0], width=true\_dimensions[1])(image=image)

sample\_preds\_folder = 'sample\_predictions/'

sample\_boxes\_folder = 'sample\_predictions\_boxes/'

if not os.path.exists(sample\_preds\_folder):

  os.makedirs(sample\_preds\_folder, exist\_ok=True)

  os.makedirs(sample\_boxes\_folder, exist\_ok=True)

for idx in range(len(test\_dataset)):

    image, gt\_mask = test\_dataset[idx]

    image\_vis = test\_dataset\_vis[idx][0].astype('uint8')

    true\_dimensions = image\_vis.shape

    # Predict

    x\_tensor = torch.from\_numpy(image).to(DEVICE).unsqueeze(0)

    pred\_mask = best\_model(x\_tensor)

    pred\_mask = pred\_mask.detach().squeeze().cpu().numpy()

    pred\_mask = np.transpose(pred\_mask, (1, 2, 0))

    # ----- Step 1: Get binary mask for polyp -----

    polyp\_class\_idx = select\_classes.index('polyp')

    binary\_mask = (pred\_mask[:, :, polyp\_class\_idx] > 0.5).astype(np.uint8)  # thresholding

    # Resize / crop binary mask to match image dims

    binary\_mask = crop\_image(binary\_mask, true\_dimensions)['image']

    # ----- Step 2: Get bounding boxes from binary mask -----

    boxes = get\_bounding\_boxes(binary\_mask, target\_class\_idx=1)  # 1 = polyp in binary mask

    boxed\_image = draw\_boxes(image\_vis, boxes)

    # ----- Step 3: Create visualizations -----

    pred\_mask\_colored = crop\_image(

        colour\_code\_segmentation(reverse\_one\_hot(pred\_mask), select\_class\_rgb\_values),

        true\_dimensions

    )['image']

    gt\_mask = np.transpose(gt\_mask, (1, 2, 0))

    gt\_mask = crop\_image(colour\_code\_segmentation(reverse\_one\_hot(gt\_mask), select\_class\_rgb\_values), true\_dimensions)['image']

    # Save triplet: original | GT | prediction

    triplet = np.hstack([image\_vis, gt\_mask, pred\_mask\_colored])

    cv2.imwrite(os.path.join(sample\_preds\_folder, f"sample\_pred\_{idx}.png"), triplet[:, :, ::-1])

    # Save image with bounding boxes

    cv2.imwrite(os.path.join(sample\_boxes\_folder, f"sample\_pred\_boxes\_{idx}.png"), boxed\_image[:, :, ::-1])

    # Optional visualization

    visualize(

        original\_image=image\_vis,

        ground\_truth\_mask=gt\_mask,

        predicted\_mask=pred\_mask\_colored,

        bounding\_box\_image=boxed\_image,

        pred\_polyp\_heatmap=crop\_image(pred\_mask[:, :, polyp\_class\_idx], true\_dimensions)['image']

    )

test\_epoch = smp.utils.train.ValidEpoch(

    model,

    loss=loss,

    metrics=metrics,

    device=DEVICE,

    verbose=True,

)

valid\_logs = test\_epoch.run(test\_dataloader)

print("Evaluation on Test Data: ")

print(f"Mean IoU Score: {valid\_logs['iou\_score']:.4f}")

print(f"Mean Dice Loss: {valid\_logs['dice\_loss']:.4f}")

def preprocess\_input(image\_np):

    true\_dimensions = image\_np.shape[:2]

    image\_rgb = cv2.cvtColor(image\_np, cv2.COLOR\_BGR2RGB)

    # Apply resizing or padding if required

    padded = get\_validation\_augmentation()(image=image\_rgb)

    image = padded['image']

    # Apply preprocessing

    preprocessed = get\_preprocessing(preprocessing\_fn)(image=image)

    image = preprocessed['image']

    return image, true\_dimensions

def predict\_polyp\_segmentation(image\_input):

    image\_np = np.array(image\_input)

    # Preprocess input

    image, true\_dimensions = preprocess\_input(image\_np)

    x\_tensor = torch.from\_numpy(image).unsqueeze(0).to(DEVICE)

    # Predict segmentation mask

    with torch.no\_grad():

        pred\_mask = best\_model(x\_tensor)

        pred\_mask = pred\_mask.squeeze().cpu().numpy()

    pred\_mask = np.transpose(pred\_mask, (1, 2, 0))

    # ----- Step 1: Get binary mask for polyp -----

    polyp\_class\_idx = select\_classes.index('polyp')

    binary\_mask = (pred\_mask[:, :, polyp\_class\_idx] > 0.5).astype(np.uint8)  # threshold

    binary\_mask = crop\_image(binary\_mask, true\_dimensions)['image']

    # ----- Step 2: Get bounding boxes from binary mask -----

    boxes = get\_bounding\_boxes(binary\_mask, target\_class\_idx=1)  # 1 = polyp

    boxed\_image = draw\_boxes(image\_np, boxes)

    # ----- Step 3: Create heatmap & segmentation visualization -----

    pred\_heatmap = crop\_image(pred\_mask[:, :, polyp\_class\_idx], true\_dimensions)['image']

    pred\_segmentation = crop\_image(

        colour\_code\_segmentation(reverse\_one\_hot(pred\_mask), select\_class\_rgb\_values),

        true\_dimensions

    )['image']

    # Grad-CAM

    input\_tensor = torch.nn.functional.pad(

        x\_tensor, (0, 0, 0, 0), mode='reflect'

    )

    rgb\_image = (image\_np - image\_np.min()) / (image\_np.max() - image\_np.min())

    rgb\_image = rgb\_image.astype(np.float32)

    cam = GradCAM(model=model, target\_layers=[model.encoder.layer4[-1]])

    targets = [SemanticSegmentationTarget(1, reverse\_one\_hot(pred\_mask))]

    grayscale\_cam = cam(input\_tensor=input\_tensor, targets=targets)[0]

    # Resize CAM to match input

    grayscale\_cam\_resized = cv2.resize(grayscale\_cam, (rgb\_image.shape[1], rgb\_image.shape[0]))

    cam\_image = show\_cam\_on\_image(rgb\_image, grayscale\_cam\_resized, use\_rgb=True)

    # Return visualizations

    return [

        image\_np,

        pred\_segmentation,

        pred\_heatmap,

        cam\_image,

        boxed\_image

    ]

model.eval()

model.to(DEVICE)

import gradio as gr

# Interface

interface = gr.Interface(

    fn=predict\_polyp\_segmentation,

    inputs=gr.Image(type="numpy", label="Upload Colonoscopy Frame"),

    outputs=[

        gr.Image(type="numpy", label="Original Image"),

        gr.Image(type="numpy", label="Predicted Segmentation"),

        gr.Image(type="numpy", label="Polyp Heatmap"),

        gr.Image(type="numpy", label="Grad-CAM Overlay"),

        gr.Image(type="numpy", label="Bounding Boxes")

    ],

    title="Polyp Segmentation in Colonoscopy Frames",

    description="Upload an endoscopic image to get polyp segmentation prediction using DeepLabV3+."

)

interface.launch(debug=True, share=True)