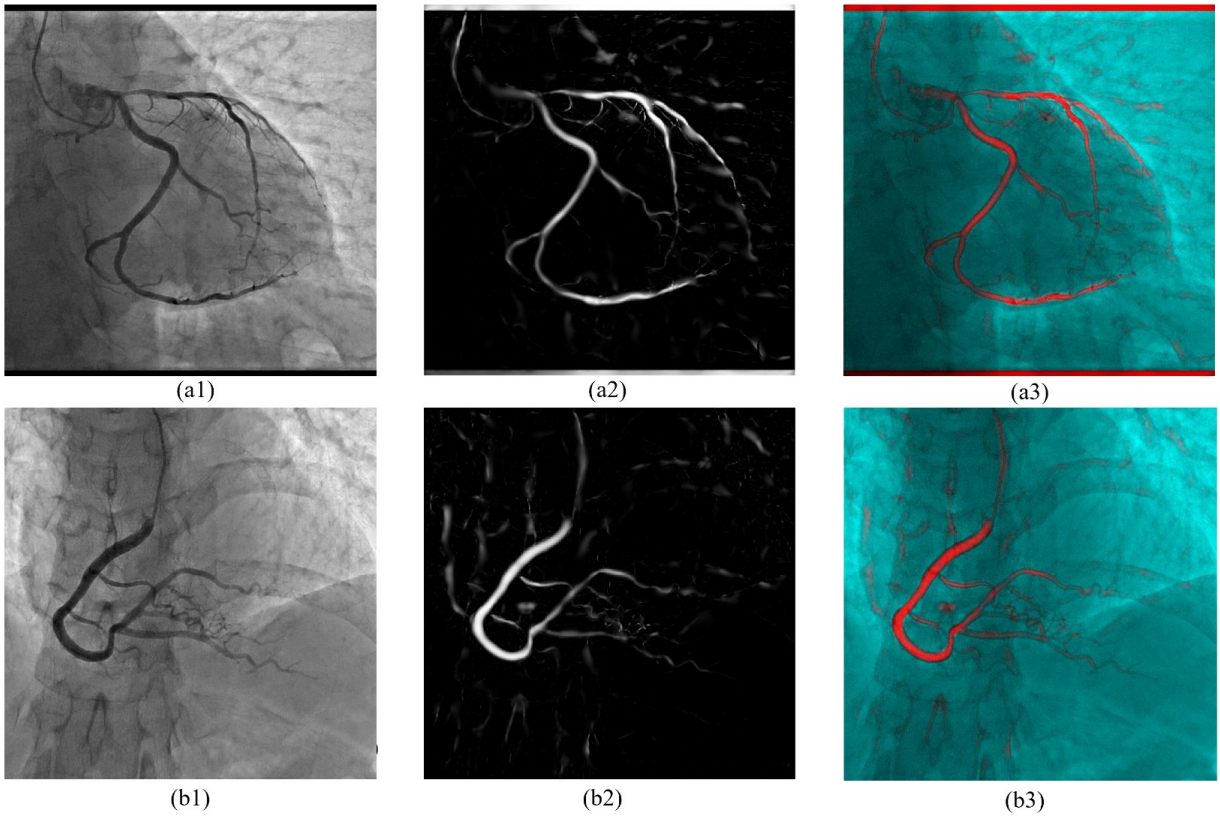
Artery Segment Classification and Stenosis Detection using Unet and Unet++

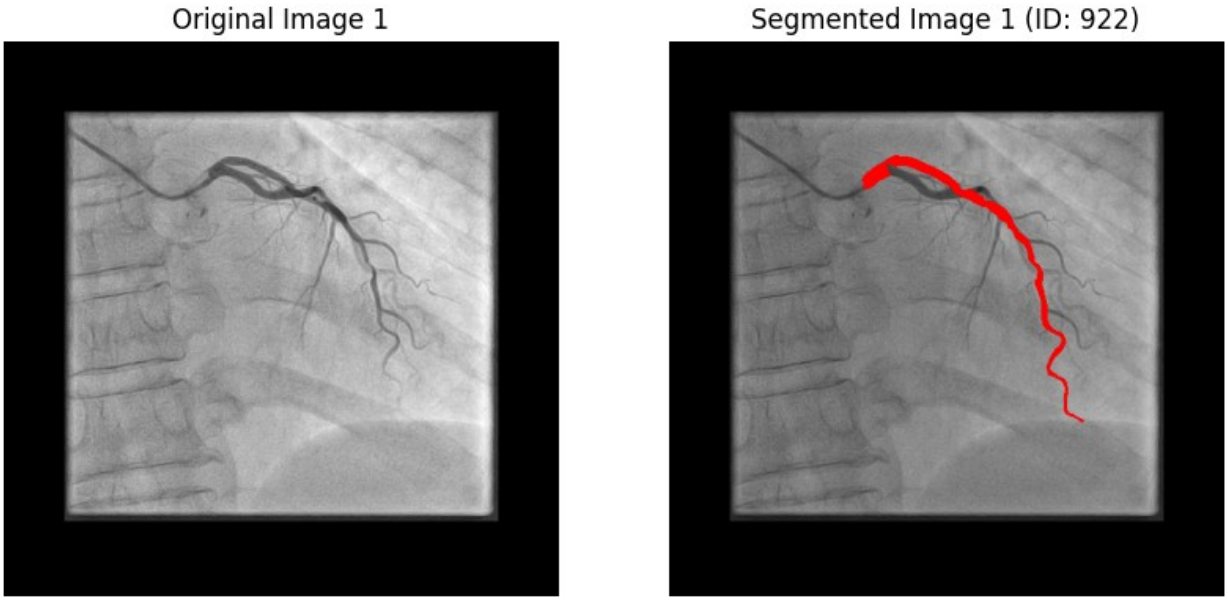



```
import json
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
from pathlib import Path

def load_json_annotations(json_path):
    with open(json_path, 'r') as f:
        data = json.load(f)
    return data

def create_mask(image_shape, segmentation):
    mask = np.zeros(image_shape, dtype=np.uint8)
    points = np.array(segmentation, dtype=np.int32).reshape(-1, 2)
    cv2.fillPoly(mask, [points], color=255)
    return mask
```

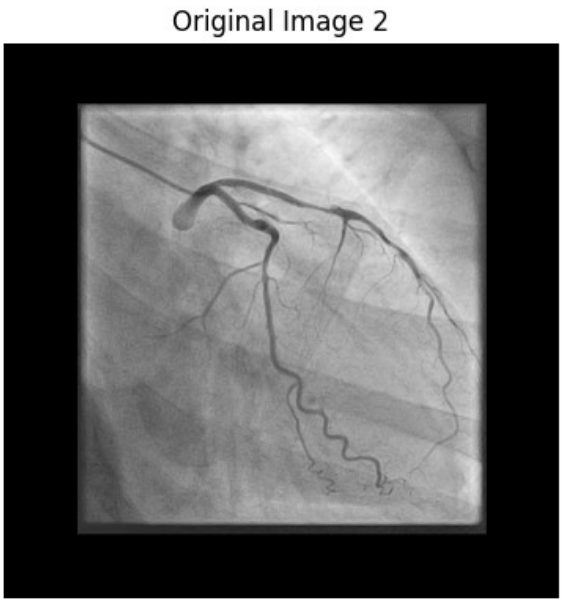
```
def visualize_segmentation(image_path, annotations, categories,
image_id, image_index):
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        raise ValueError(f"Could not load image at {image_path}")

    img_color = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)

    img_annotations = [ann for ann in annotations if ann['image_id']
== image_id]

    mask = np.zeros_like(img, dtype=np.uint8)

    for ann in img_annotations:
        category_id = ann['category_id']
        category_name = next(cat['name'] for cat in categories if
cat['id'] == category_id)
        segmentation = ann['segmentation']

        ann_mask = create_mask(img.shape, segmentation)
        mask = np.maximum(mask, ann_mask)

        img_color[ann_mask == 255] = [255, 0, 0]

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.title(f"Original Image {image_index + 1}")
    plt.imshow(img, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title(f"Segmented Image {image_index + 1} (ID: {image_id})")
    plt.imshow(img_color)
    plt.axis('off')

    plt.show()
    return mask

def main():
    base_path = "/kaggle/input/arcade-dataset/arcade/syntax"
    train_images_path = os.path.join(base_path, "train", "images")
    train_json_path = os.path.join(base_path, "train", "annotations",
"train.json")

    data = load_json_annotations(train_json_path)
    images = data['images']
    annotations = data['annotations']
    categories = data['categories']
```

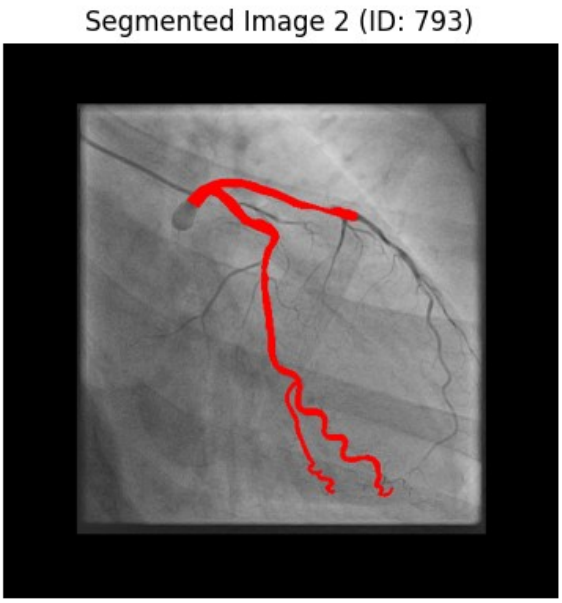
```
    for idx, sample_image in enumerate(images[:5]):
        image_id = sample_image['id']
        image_file = sample_image['file_name']
        image_path = os.path.join(train_images_path, image_file)

        print(f"Processing image: {image_file}")
        visualize_segmentation(image_path, annotations, categories,
image_id, idx)

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

```
Processing image: 922.png
```

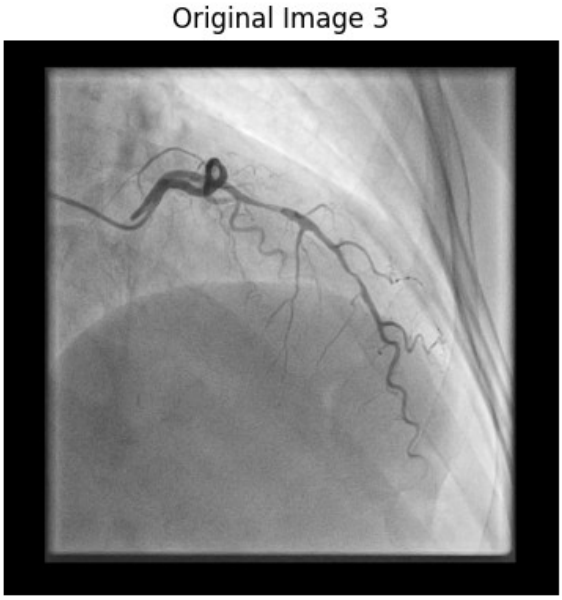


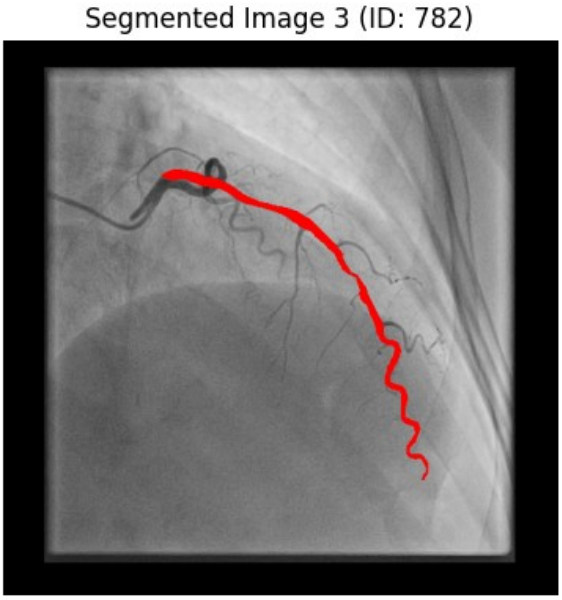

```
Processing image: 793.png
```



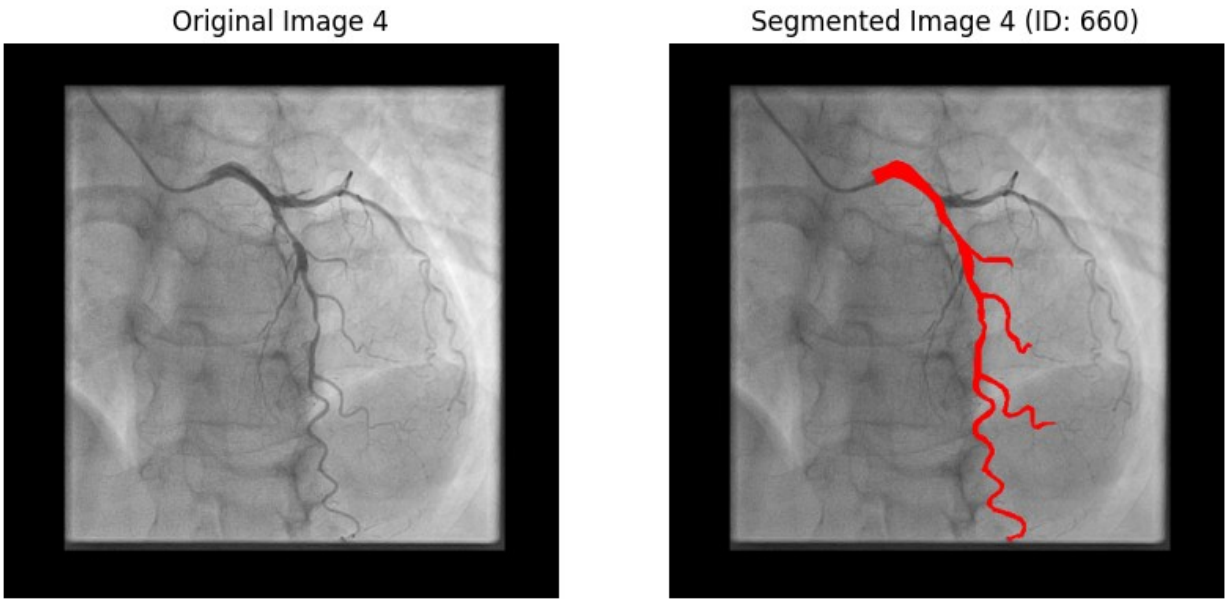



```
Processing image: 782.png
```

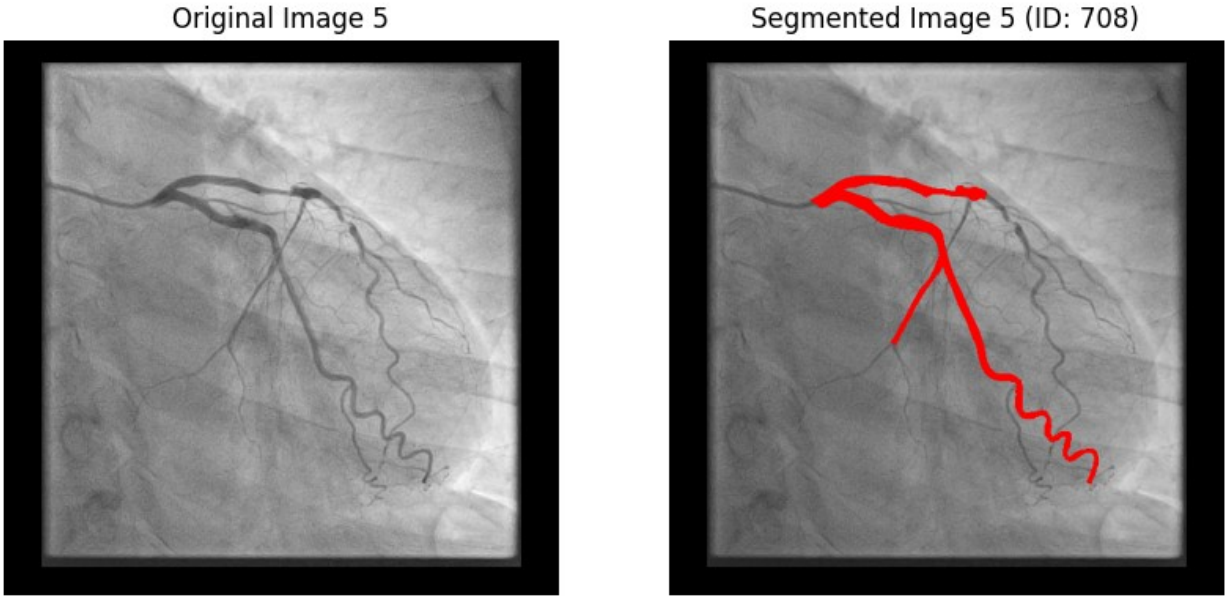





```
Processing image: 660.png
```




```
Processing image: 708.png
```




```
import json
import cv2
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
from pathlib import Path
```

```
import numpy as np
import os

class ARCADE_Dataset(Dataset):
    def __init__(self, images_dir, json_path, img_size=(512, 512),
transform=None):
        self.images_dir = Path(images_dir)
        self.img_size = img_size
        self.transform = transform

        if not self.images_dir.exists():
            raise FileNotFoundError(f"Images directory not found:
{images_dir}")
        if not Path(json_path).exists():
            raise FileNotFoundError(f"JSON annotation file not found:
{json_path}")

        with open(json_path, 'r') as f:
            self.data = json.load(f)

        self.images = self.data['images']
        self.annotations = self.data['annotations']
        self.categories = self.data['categories']

        self.image_annotations_map = {}
        for ann in self.annotations:
            image_id = ann['image_id']
            if image_id not in self.image_annotations_map:
                self.image_annotations_map[image_id] = []
            self.image_annotations_map[image_id].append(ann)

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

    def __getitem__(self, idx):
        img_info = self.images[idx]
        image_id = img_info['id']
        image_path = self.images_dir / img_info['file_name']

        img = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
        if img is None:
            raise ValueError(f"Could not load image at {image_path}.
Check file integrity.")

        original_height, original_width = img.shape[:2]
        img = cv2.resize(img, self.img_size,
interpolation=cv2.INTER_AREA)
        img = img / 255.0

        mask = np.zeros(self.img_size, dtype=np.uint8)
```

```
        img_annotations = self.image_annotations_map.get(image_id, [])


        if not img_annotations:
            pass

        for ann in img_annotations:
            segmentations = ann['segmentation']
            if not isinstance(segmentations, list) or not
all(isinstance(s, list) for s in segmentations):
                segmentations = [segmentations]

            for segmentation_points in segmentations:
                if not segmentation_points:
                    continue
                points = np.array(segmentation_points,
dtype=np.float32).reshape(-1, 2)
                scale_x = self.img_size[0] / original_width
                scale_y = self.img_size[1] / original_height
                points[:, 0] = points[:, 0] * scale_x
                points[:, 1] = points[:, 1] * scale_y
                points = points.astype(np.int32)
                points[:, 0] = np.clip(points[:, 0], 0,
self.img_size[0] - 1)
                points[:, 1] = np.clip(points[:, 1], 0,
self.img_size[1] - 1)
                if points.size > 0:
                    cv2.fillPoly(mask, [points], color=1)

        img_tensor = torch.tensor(img,
dtype=torch.float32).unsqueeze(0)
        mask_tensor = torch.tensor(mask,
dtype=torch.float32).unsqueeze(0)

        if self.transform:
            seed = torch.seed()
            torch.manual_seed(seed)
            img_tensor = self.transform(img_tensor)
            torch.manual_seed(seed)
            mask_tensor = self.transform(mask_tensor)

        return img_tensor, mask_tensor

class DiceLoss(nn.Module):
    def __init__(self):
        super(DiceLoss, self).__init__()

    def forward(self, pred, target):
        pred = torch.sigmoid(pred)
        pred = pred.view(-1)
```

```
        target = target.view(-1)
        intersection = (pred * target).sum()
        dice = (2. * intersection + 1e-6) / (pred.sum() + target.sum()
+ 1e-6)
        return 1 - dice

class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        def conv_block(in_channels, out_channels):
            return nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(out_channels, out_channels, 3, padding=1),
                nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(1, 64)
        self.enc2 = conv_block(64, 128)
        self.enc3 = conv_block(128, 256)
        self.enc4 = conv_block(256, 512)
        self.pool = nn.MaxPool2d(2, 2)

        self.bottleneck = conv_block(512, 1024)

        self.upconv4 = nn.ConvTranspose2d(1024, 512, 2, stride=2)
        self.dec4 = conv_block(1024, 512)
        self.upconv3 = nn.ConvTranspose2d(512, 256, 2, stride=2)
        self.dec3 = conv_block(512, 256)
        self.upconv2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = conv_block(256, 128)
        self.upconv1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = conv_block(128, 64)

        self.final_conv = nn.Conv2d(64, 1, 1)

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        e4 = self.enc4(self.pool(e3))
        b = self.bottleneck(self.pool(e4))
        d4 = self.upconv4(b)
        d4 = torch.cat([d4, e4], dim=1)
        d4 = self.dec4(d4)
        d3 = self.upconv3(d4)
        d3 = torch.cat([d3, e3], dim=1)
        d3 = self.dec3(d3)
        d2 = self.upconv2(d3)
```

```
        d2 = torch.cat([d2, e2], dim=1)
        d2 = self.dec2(d2)
        d1 = self.upconv1(d2)
        d1 = torch.cat([d1, e1], dim=1)
        d1 = self.dec1(d1)
        out = self.final_conv(d1)
        return out

def calculate_iou(pred, target, threshold=0.5):
    pred = (torch.sigmoid(pred) > threshold).float()
    pred_flat = pred.view(pred.shape[0], -1)
    target_flat = target.view(target.shape[0], -1)
    intersection = (pred_flat * target_flat).sum(dim=1)
    union = (pred_flat + target_flat).sum(dim=1) - intersection
    iou = (intersection + 1e-6) / (union + 1e-6)
    return iou.mean().item()

def calculate_dice(pred, target, threshold=0.5):
    pred = (torch.sigmoid(pred) > threshold).float()
    pred_flat = pred.view(pred.shape[0], -1)
    target_flat = target.view(target.shape[0], -1)
    intersection = (pred_flat * target_flat).sum(dim=1)
    dice = (2 * intersection + 1e-6) / (pred_flat.sum(dim=1) +
target_flat.sum(dim=1) + 1e-6)
    return dice.mean().item()

def train_model(model, train_loader, val_loader, num_epochs=15,
device='cuda'):
    model = model.to(device)
    criterion_bce = nn.BCEWithLogitsLoss()
    criterion_dice = DiceLoss()
    optimizer = optim.Adam(model.parameters(), lr=5e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.5, patience=2)

    train_losses, val_losses, val_ious, val_dices = [], [], [], []
    best_val_loss = float('inf')
    best_epoch = -1

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
        for batch_idx, (images, masks) in enumerate(train_loader):
            images, masks = images.to(device), masks.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss_bce = criterion_bce(outputs, masks)
            loss_dice = criterion_dice(outputs, masks)
            loss = 0.5 * loss_bce + 0.5 * loss_dice
            loss.backward()
```

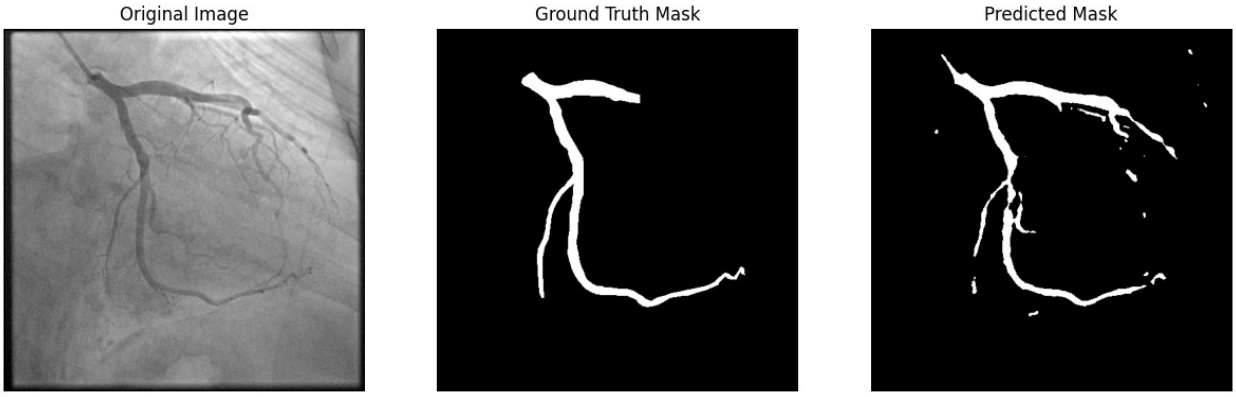
```
            optimizer.step()
            train_loss += loss.item()
            if batch_idx % 50 == 0:
                print(f"Epoch {epoch+1}/{num_epochs}, Batch
{batch_idx}/{len(train_loader)}, Train Loss: {loss.item():.4f}")

        avg_train_loss = train_loss / len(train_loader)
        train_losses.append(avg_train_loss)
        model.eval()
        val_loss, val_iou, val_dice = 0, 0, 0
        with torch.no_grad():
            for images, masks in val_loader:
                images, masks = images.to(device), masks.to(device)
                outputs = model(images)
                loss_bce = criterion_bce(outputs, masks)
                loss_dice = criterion_dice(outputs, masks)
                val_loss += (0.5 * loss_bce + 0.5 * loss_dice).item()
                val_iou += calculate_iou(outputs, masks)
                val_dice += calculate_dice(outputs, masks)

        avg_val_loss = val_loss / len(val_loader)
        avg_val_iou = val_iou / len(val_loader)
        avg_val_dice = val_dice / len(val_loader)
        val_losses.append(avg_val_loss)
        val_ious.append(avg_val_iou)
        val_dices.append(avg_val_dice)

        print(f"Epoch {epoch+1}/{num_epochs}, Train Loss:
{avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}, Val IoU:
{avg_val_iou:.4f}, Val Dice: {avg_val_dice:.4f}")

        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            best_epoch = epoch
            torch.save(model.state_dict(), "best_unet_model.pth")
            print(f"Saved best model at Epoch {epoch+1} with Val Loss:
{best_val_loss:.4f}")

        scheduler.step(avg_val_loss)

    print(f"Training finished. Best model saved from Epoch
{best_epoch+1} with Val Loss: {best_val_loss:.4f}")
    return model, train_losses, val_losses, val_ious, val_dices

def visualize_predictions(model, data_loader, device, num_images=5):
    model.eval()
    with torch.no_grad():
        for idx, (images, masks) in enumerate(data_loader):
            if idx >= num_images:
```

```
                break
            images, masks = images.to(device), masks.to(device)
            outputs = model(images)
            preds = torch.sigmoid(outputs).cpu().numpy() > 0.5
            img = images[0].cpu().numpy().squeeze()
            true_mask = masks[0].cpu().numpy().squeeze()
            pred_mask = preds[0].squeeze()
            print(f"Image {idx+1}: True mask sum: {true_mask.sum()},
Predicted mask sum: {pred_mask.sum()}")
            plt.figure(figsize=(15, 5))
            plt.subplot(1, 3, 1)
            plt.title("Original Image")
            plt.imshow(img, cmap='gray')
            plt.axis('off')
            plt.subplot(1, 3, 2)
            plt.title("Ground Truth Mask")
            plt.imshow(true_mask, cmap='gray')
            plt.axis('off')
            plt.subplot(1, 3, 3)
            plt.title("Predicted Mask")
            plt.imshow(pred_mask, cmap='gray')
            plt.axis('off')
            plt.show()

def main():
    base_path = "/kaggle/input/arcade-dataset/arcade/syntax"
    train_images_path = os.path.join(base_path, "train", "images")
    train_json_path = os.path.join(base_path, "train", "annotations",
"train.json")
    val_images_path = os.path.join(base_path, "val", "images")
    val_json_path = os.path.join(base_path, "val", "annotations",
"val.json")

    transform = transforms.Compose([
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomRotation(10),
    ])

    try:
        train_dataset = ARCADE_Dataset(train_images_path,
train_json_path, transform=transform)
        val_dataset = ARCADE_Dataset(val_images_path, val_json_path)
    except FileNotFoundError as e:
        print(f"Error loading dataset: {e}")
        return
    except Exception as e:
        print(f"An unexpected error occurred during dataset loading:
{e}")
        return
```

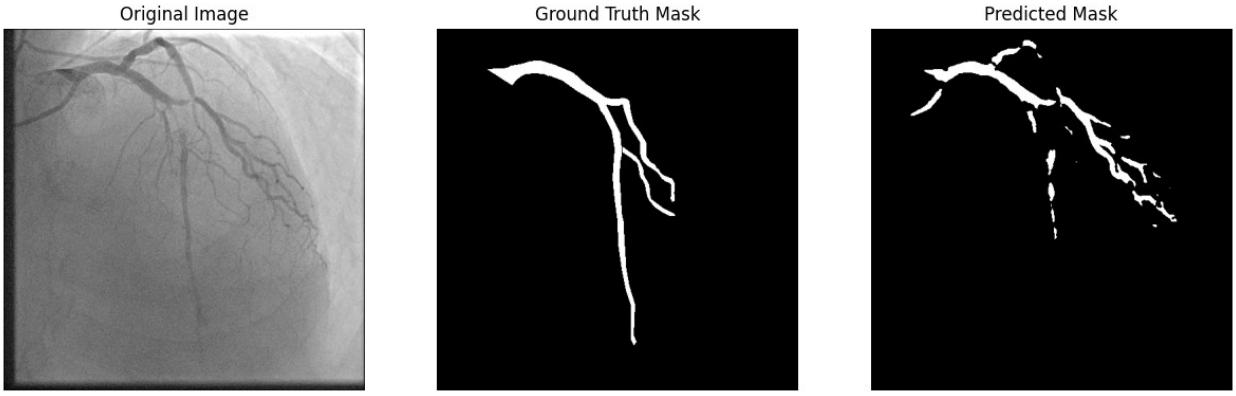
```
    train_loader = DataLoader(train_dataset, batch_size=4,
shuffle=True, num_workers=os.cpu_count() // 2 or 1)
    val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False,
num_workers=os.cpu_count() // 2 or 1)

    print(f"Train dataset size: {len(train_dataset)}")
    print(f"Validation dataset size: {len(val_dataset)}")
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    print(f"Using device: {device}")

    print("Training U-Net...")
    unet_model = UNet()
    trained_unet, train_losses, val_losses, val_ious, val_dices =
train_model(
        unet_model, train_loader, val_loader, num_epochs=3,
device=device
    )

    if Path("best_unet_model.pth").exists():

trained_unet.load_state_dict(torch.load("best_unet_model.pth"))
        print("Loaded best model for visualization.")
    else:
        print("No best model saved. Using the last trained model.")

    print("Visualizing U-Net predictions...")
    visualize_predictions(trained_unet, val_loader, device,
num_images=5)

    plt.figure(figsize=(18, 5))
    plt.subplot(1, 3, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.title('Loss Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 3, 2)
    plt.plot(val_ious, label='Val IoU', color='orange')
    plt.title('IoU Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('IoU')
    plt.legend()
    plt.grid(True)
```

```
    plt.subplot(1, 3, 3)
    plt.plot(val_dices, label='Val Dice', color='green')
    plt.title('Dice Coefficient Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Dice')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

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

```
Train dataset size: 1000
Validation dataset size: 200
Using device: cuda
Training U-Net...
Epoch 1/3, Batch 0/250, Train Loss: 0.8034
Epoch 1/3, Batch 50/250, Train Loss: 0.5405
Epoch 1/3, Batch 100/250, Train Loss: 0.5360
Epoch 1/3, Batch 150/250, Train Loss: 0.5094
Epoch 1/3, Batch 200/250, Train Loss: 0.5085
Epoch 1/3, Train Loss: 0.5482, Val Loss: 0.5805, Val IoU: 0.0021, Val
Dice: 0.0041
Saved best model at Epoch 1 with Val Loss: 0.5805
Epoch 2/3, Batch 0/250, Train Loss: 0.5365
Epoch 2/3, Batch 50/250, Train Loss: 0.4342
Epoch 2/3, Batch 100/250, Train Loss: 0.5065
Epoch 2/3, Batch 150/250, Train Loss: 0.3063
Epoch 2/3, Batch 200/250, Train Loss: 0.4270
Epoch 2/3, Train Loss: 0.4513, Val Loss: 0.3406, Val IoU: 0.3167, Val
Dice: 0.4641
Saved best model at Epoch 2 with Val Loss: 0.3406
Epoch 3/3, Batch 0/250, Train Loss: 0.3498
Epoch 3/3, Batch 50/250, Train Loss: 0.5037
Epoch 3/3, Batch 100/250, Train Loss: 0.2230
Epoch 3/3, Batch 150/250, Train Loss: 0.2289
Epoch 3/3, Batch 200/250, Train Loss: 0.2971
Epoch 3/3, Train Loss: 0.3068, Val Loss: 0.2479, Val IoU: 0.4526, Val
Dice: 0.6121
Saved best model at Epoch 3 with Val Loss: 0.2479
Training finished. Best model saved from Epoch 3 with Val Loss: 0.2479
Loaded best model for visualization.
Visualizing U-Net predictions...
Image 1: True mask sum: 10525.0, Predicted mask sum: 11477
```

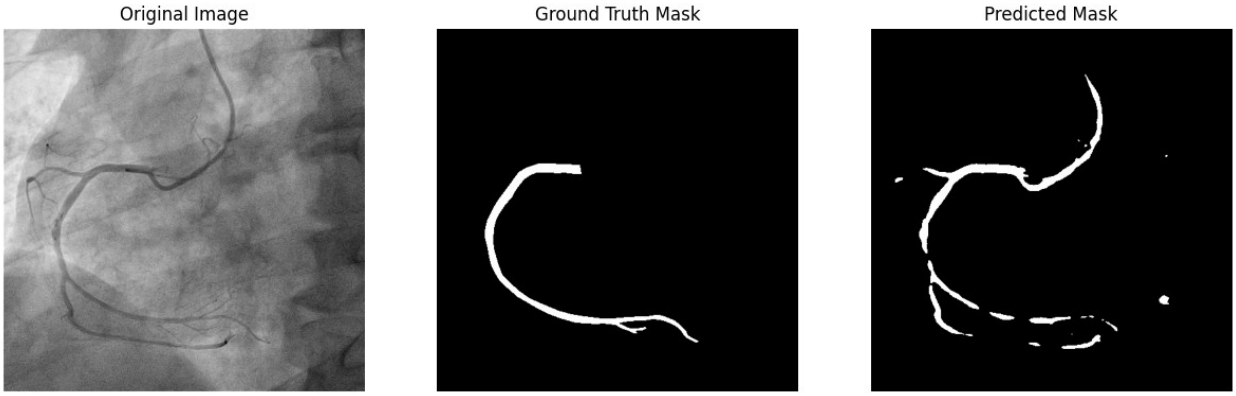



```
Image 2: True mask sum: 7860.0, Predicted mask sum: 7244
```

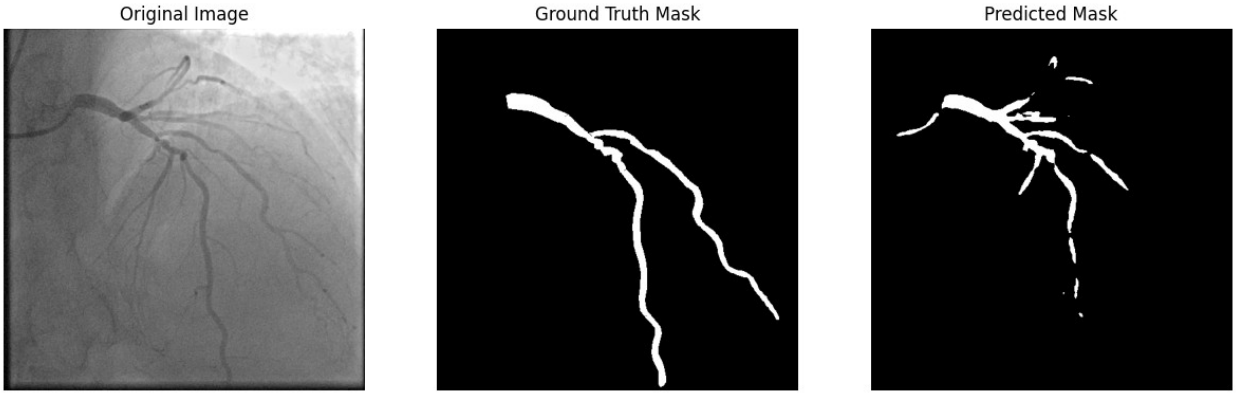


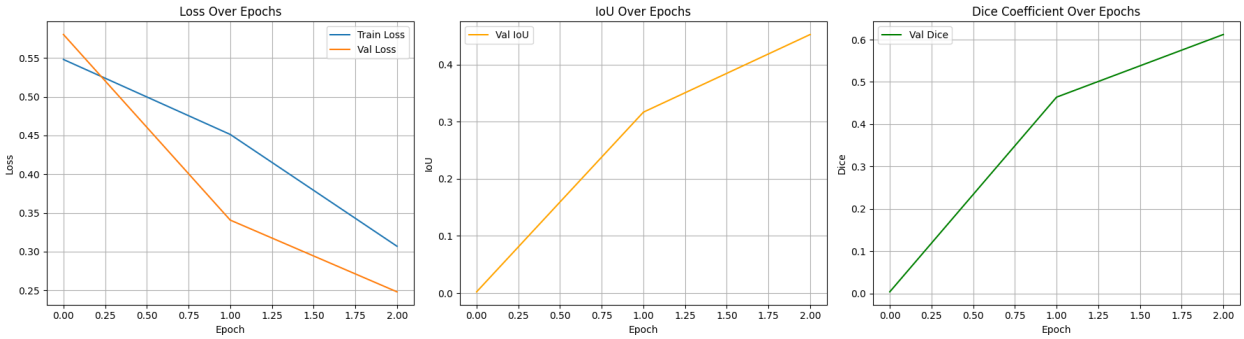

```
Image 3: True mask sum: 6711.0, Predicted mask sum: 6663
```


```
Image 4: True mask sum: 5394.0, Predicted mask sum: 7119
```




```
Image 5: True mask sum: 9790.0, Predicted mask sum: 6954
```






```python
import json
import os
import cv2
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
from pathlib import Path
```

```
class ARCADE_Dataset(Dataset):
    def __init__(self, images_dir, json_path, img_size=(512, 512),
transform=None):
        self.images_dir = Path(images_dir)
        self.img_size = img_size
        self.transform = transform

        if not self.images_dir.exists():
            raise FileNotFoundError(f"Images directory not found:
{images_dir}")
        if not Path(json_path).exists():
            raise FileNotFoundError(f"JSON annotation file not found:
{json_path}")

        with open(json_path, 'r') as f:
            self.data = json.load(f)

        self.images = self.data['images']
        self.annotations = self.data['annotations']
        self.categories = self.data['categories']

        self.image_annotations_map = {}
        for ann in self.annotations:
            image_id = ann['image_id']
            if image_id not in self.image_annotations_map:
                self.image_annotations_map[image_id] = []
            self.image_annotations_map[image_id].append(ann)

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

    def __getitem__(self, idx):
        img_info = self.images[idx]
        image_id = img_info['id']
        image_path = self.images_dir / img_info['file_name']

        img = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
        if img is None:
            raise ValueError(f"Could not load image at {image_path}.
Check file integrity.")

        original_height, original_width = img.shape[:2]
        img = cv2.resize(img, self.img_size,
interpolation=cv2.INTER_AREA)
        img = img / 255.0

        mask = np.zeros(self.img_size, dtype=np.uint8)

        img_annotations = self.image_annotations_map.get(image_id, [])
```

```
        if not img_annotations:
            pass

        for ann in img_annotations:
            segmentations = ann['segmentation']
            if not isinstance(segmentations, list) or not
all(isinstance(s, list) for s in segmentations):
                segmentations = [segmentations]

            for segmentation_points in segmentations:
                if not segmentation_points:
                    continue
                points = np.array(segmentation_points,
dtype=np.float32).reshape(-1, 2)

                scale_x = self.img_size[0] / original_width
                scale_y = self.img_size[1] / original_height
                points[:, 0] = points[:, 0] * scale_x
                points[:, 1] = points[:, 1] * scale_y

                points = points.astype(np.int32)

                points[:, 0] = np.clip(points[:, 0], 0,
self.img_size[0] - 1)
                points[:, 1] = np.clip(points[:, 1], 0,
self.img_size[1] - 1)

                if points.size > 0:
                    cv2.fillPoly(mask, [points], color=1)

        img_tensor = torch.tensor(img,
dtype=torch.float32).unsqueeze(0)
        mask_tensor = torch.tensor(mask,
dtype=torch.float32).unsqueeze(0)

        if self.transform:
            seed = torch.seed()
            torch.manual_seed(seed)
            img_tensor = self.transform(img_tensor)
            torch.manual_seed(seed)
            mask_tensor = self.transform(mask_tensor)

        return img_tensor, mask_tensor

class DiceLoss(nn.Module):
    def __init__(self):
        super(DiceLoss, self).__init__()
```

```
    def forward(self, pred, target):
        pred = torch.sigmoid(pred)
        pred = pred.view(-1)
        target = target.view(-1)
        intersection = (pred * target).sum()
        dice = (2. * intersection + 1e-6) / (pred.sum() + target.sum()
+ 1e-6)
        return 1 - dice

class VGGBlock(nn.Module):
    def __init__(self, in_channels, middle_channels, out_channels):
        super().__init__()
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_channels, middle_channels, 3,
padding=1)
        self.bn1 = nn.BatchNorm2d(middle_channels)
        self.conv2 = nn.Conv2d(middle_channels, out_channels, 3,
padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)
        return out

class NestedUNet(nn.Module):
    def __init__(self, num_classes=1, input_channels=1,
deep_supervision=False):
        super().__init__()
        self.num_classes = num_classes
        self.deep_supervision = deep_supervision
        filters = [32, 64, 128, 256, 512]
        self.pool = nn.MaxPool2d(2, 2)
        self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)

        self.conv0_0 = VGGBlock(input_channels, filters[0],
filters[0])
        self.conv1_0 = VGGBlock(filters[0], filters[1], filters[1])
        self.conv0_1 = VGGBlock(filters[0] + filters[1], filters[0],
filters[0])
        self.conv2_0 = VGGBlock(filters[1], filters[2], filters[2])
        self.conv1_1 = VGGBlock(filters[1] + filters[2], filters[1],
filters[1])
        self.conv0_2 = VGGBlock(filters[0]*2 + filters[1], filters[0],
filters[0])
```

```
        self.conv3_0 = VGGBlock(filters[2], filters[3], filters[3])
        self.conv2_1 = VGGBlock(filters[2] + filters[3], filters[2],
filters[2])
        self.conv1_2 = VGGBlock(filters[1]*2 + filters[2], filters[1],
filters[1])
        self.conv0_3 = VGGBlock(filters[0]*3 + filters[1], filters[0],
filters[0])
        self.conv4_0 = VGGBlock(filters[3], filters[4], filters[4])
        self.conv3_1 = VGGBlock(filters[3] + filters[4], filters[3],
filters[3])
        self.conv2_2 = VGGBlock(filters[2]*2 + filters[3], filters[2],
filters[2])
        self.conv1_3 = VGGBlock(filters[1]*3 + filters[2], filters[1],
filters[1])
        self.conv0_4 = VGGBlock(filters[0]*4 + filters[1], filters[0],
filters[0])

        if self.deep_supervision:
            self.output1 = nn.Conv2d(filters[0], num_classes, 1)
            self.output2 = nn.Conv2d(filters[0], num_classes, 1)
            self.output3 = nn.Conv2d(filters[0], num_classes, 1)
            self.output4 = nn.Conv2d(filters[0], num_classes, 1)
        else:
            self.output = nn.Conv2d(filters[0], num_classes, 1)

    def forward(self, input):
        x0_0 = self.conv0_0(input)
        x1_0 = self.conv1_0(self.pool(x0_0))
        x0_1 = self.conv0_1(torch.cat([x0_0, self.up(x1_0)], 1))
        x2_0 = self.conv2_0(self.pool(x1_0))
        x1_1 = self.conv1_1(torch.cat([x1_0, self.up(x2_0)], 1))
        x0_2 = self.conv0_2(torch.cat([x0_0, x0_1, self.up(x1_1)], 1))
        x3_0 = self.conv3_0(self.pool(x2_0))
        x2_1 = self.conv2_1(torch.cat([x2_0, self.up(x3_0)], 1))
        x1_2 = self.conv1_2(torch.cat([x1_0, x1_1, self.up(x2_1)], 1))
        x0_3 = self.conv0_3(torch.cat([x0_0, x0_1, x0_2,
self.up(x1_2)], 1))
        x4_0 = self.conv4_0(self.pool(x3_0))
        x3_1 = self.conv3_1(torch.cat([x3_0, self.up(x4_0)], 1))
        x2_2 = self.conv2_2(torch.cat([x2_0, x2_1, self.up(x3_1)], 1))
        x1_3 = self.conv1_3(torch.cat([x1_0, x1_1, x1_2,
self.up(x2_2)], 1))
        x0_4 = self.conv0_4(torch.cat([x0_0, x0_1, x0_2, x0_3,
self.up(x1_3)], 1))
        if self.deep_supervision:
            output1 = self.output1(x0_1)
            output2 = self.output2(x0_2)
            output3 = self.output3(x0_3)
            output4 = self.output4(x0_4)
```

```
            return [output1, output2, output3, output4]
        else:
            return self.output(x0_4)

def calculate_iou(pred, target, threshold=0.5):
    pred = (torch.sigmoid(pred) > threshold).float()
    pred_flat = pred.view(pred.shape[0], -1)
    target_flat = target.view(target.shape[0], -1)
    intersection = (pred_flat * target_flat).sum(dim=1)
    union = (pred_flat + target_flat).sum(dim=1) - intersection
    iou = (intersection + 1e-6) / (union + 1e-6)
    return iou.mean().item()

def calculate_dice(pred, target, threshold=0.5):
    pred = (torch.sigmoid(pred) > threshold).float()
    pred_flat = pred.view(pred.shape[0], -1)
    target_flat = target.view(target.shape[0], -1)
    intersection = (pred_flat * target_flat).sum(dim=1)
    dice = (2 * intersection + 1e-6) / (pred_flat.sum(dim=1) +
target_flat.sum(dim=1) + 1e-6)
    return dice.mean().item()

def train_model(model, train_loader, val_loader, num_epochs=15,
device='cuda', deep_supervision=False):
    model = model.to(device)
    criterion_bce = nn.BCEWithLogitsLoss()
    criterion_dice = DiceLoss()
    optimizer = optim.Adam(model.parameters(), lr=5e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.5, patience=2)
    train_losses, val_losses, val_ious, val_dices = [], [], [], []
    best_val_loss = float('inf')
    best_epoch = -1

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
        for batch_idx, (images, masks) in enumerate(train_loader):
            images, masks = images.to(device), masks.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            if deep_supervision:
                loss = 0
                for output in outputs:
                    loss_bce = criterion_bce(output, masks)
                    loss_dice = criterion_dice(output, masks)
                    loss += (0.5 * loss_bce + 0.5 * loss_dice)
                loss /= len(outputs)
            else:
                loss_bce = criterion_bce(outputs, masks)
```

```
                loss_dice = criterion_dice(outputs, masks)
                loss = 0.5 * loss_bce + 0.5 * loss_dice
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
            if batch_idx % 50 == 0:
                print(f"Epoch {epoch+1}/{num_epochs}, Batch
{batch_idx}/{len(train_loader)}, Train Loss: {loss.item():.4f}")
        avg_train_loss = train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        model.eval()
        val_loss, val_iou, val_dice = 0, 0, 0
        with torch.no_grad():
            for images, masks in val_loader:
                images, masks = images.to(device), masks.to(device)
                outputs = model(images)
                if deep_supervision:
                    final_output = outputs[-1]
                    loss_bce = criterion_bce(final_output, masks)
                    loss_dice = criterion_dice(final_output, masks)
                    val_loss += (0.5 * loss_bce + 0.5 *
loss_dice).item()
                    val_iou += calculate_iou(final_output, masks)
                    val_dice += calculate_dice(final_output, masks)
                else:
                    loss_bce = criterion_bce(outputs, masks)
                    loss_dice = criterion_dice(outputs, masks)
                    val_loss += (0.5 * loss_bce + 0.5 *
loss_dice).item()
                    val_iou += calculate_iou(outputs, masks)
                    val_dice += calculate_dice(outputs, masks)
        avg_val_loss = val_loss / len(val_loader)
        avg_val_iou = val_iou / len(val_loader)
        avg_val_dice = val_dice / len(val_loader)
        val_losses.append(avg_val_loss)
        val_ious.append(avg_val_iou)
        val_dices.append(avg_val_dice)

        print(f"Epoch {epoch+1}/{num_epochs}, Train Loss:
{avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}, Val IoU:
{avg_val_iou:.4f}, Val Dice: {avg_val_dice:.4f}")
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            best_epoch = epoch
            torch.save(model.state_dict(), "best_unetpp_model.pth")
            print(f"Saved best model at Epoch {epoch+1} with Val Loss:
{best_val_loss:.4f}")
        scheduler.step(avg_val_loss)
```

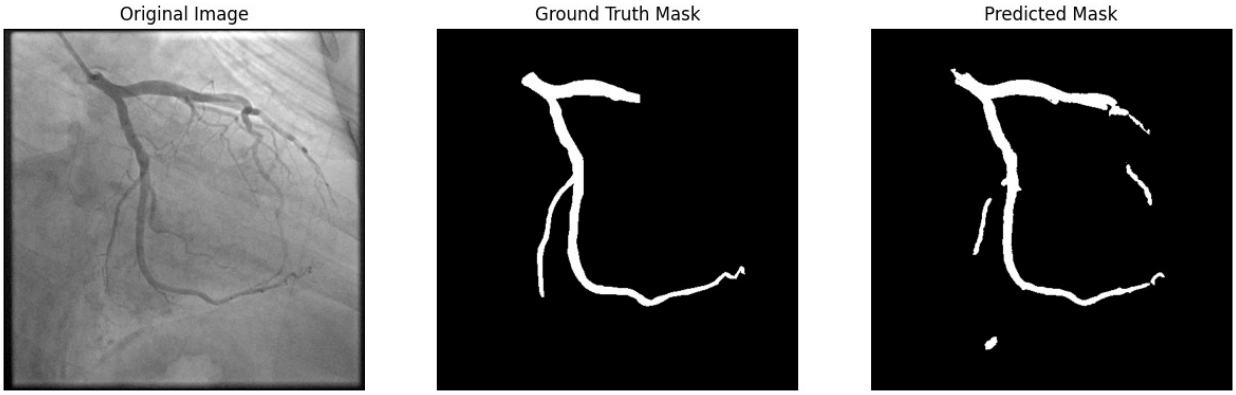
```
    print(f"Training finished. Best model saved from Epoch
{best_epoch+1} with Val Loss: {best_val_loss:.4f}")
    return model, train_losses, val_losses, val_ious, val_dices

def visualize_predictions(model, data_loader, device, num_images=5,
deep_supervision=False):
    model.eval()
    with torch.no_grad():
        for idx, (images, masks) in enumerate(data_loader):
            if idx >= num_images:
                break
            images, masks = images.to(device), masks.to(device)
            outputs = model(images)
            if deep_supervision:
                outputs = outputs[-1]
            preds = (torch.sigmoid(outputs).cpu().numpy() >
0.5).astype(np.uint8)
            img = images[0].cpu().numpy().squeeze()
            true_mask = masks[0].cpu().numpy().squeeze()
            pred_mask = preds[0].squeeze()
            print(f"Image {idx+1}: True mask sum: {true_mask.sum()},
Predicted mask sum: {pred_mask.sum()}")
            plt.figure(figsize=(15, 5))
            plt.subplot(1, 3, 1)
            plt.title("Original Image")
            plt.imshow(img, cmap='gray')
            plt.axis('off')
            plt.subplot(1, 3, 2)
            plt.title("Ground Truth Mask")
            plt.imshow(true_mask, cmap='gray')
            plt.axis('off')
            plt.subplot(1, 3, 3)
            plt.title("Predicted Mask")
            plt.imshow(pred_mask, cmap='gray')
            plt.axis('off')
            plt.show()

def main():
    base_path = "/kaggle/input/arcade-dataset/arcade/syntax"
    train_images_path = os.path.join(base_path, "train", "images")
    train_json_path = os.path.join(base_path, "train", "annotations",
"train.json")
    val_images_path = os.path.join(base_path, "val", "images")
    val_json_path = os.path.join(base_path, "val", "annotations",
"val.json")
    transform =
transforms.Compose([transforms.RandomHorizontalFlip(p=0.5),
transforms.RandomRotation(10)])
    DEEP_SUPERVISION = False
```

```
    try:
        train_dataset = ARCADE_Dataset(train_images_path,
train_json_path, transform=transform)
        val_dataset = ARCADE_Dataset(val_images_path, val_json_path)
    except FileNotFoundError as e:
        print(f"Error loading dataset: {e}")
        return
    except Exception as e:
        print(f"An unexpected error occurred during dataset loading:
{e}")
        return
    train_loader = DataLoader(train_dataset, batch_size=4,
shuffle=True, num_workers=os.cpu_count() // 2 or 1)
    val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False,
num_workers=os.cpu_count() // 2 or 1)
    print(f"Train dataset size: {len(train_dataset)}")
    print(f"Validation dataset size: {len(val_dataset)}")
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    print(f"Using device: {device}")
    print("Training U-Net++...")
    unetpp_model = NestedUNet(num_classes=1, input_channels=1,
deep_supervision=DEEP_SUPERVISION)
    trained_unetpp, train_losses, val_losses, val_ious, val_dices =
train_model(
        unetpp_model, train_loader, val_loader, num_epochs=3,
device=device, deep_supervision=DEEP_SUPERVISION
    )
    if Path("best_unetpp_model.pth").exists():

trained_unetpp.load_state_dict(torch.load("best_unetpp_model.pth"))
        print("Loaded best U-Net++ model for visualization.")
    else:
        print("No best U-Net++ model saved. Using the last trained
model.")
    print("Visualizing U-Net++ predictions...")
    visualize_predictions(trained_unetpp, val_loader, device,
num_images=5, deep_supervision=DEEP_SUPERVISION)
# Plot training metrics
    plt.figure(figsize=(18, 5))

    plt.subplot(1, 3, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.title('Loss Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
```

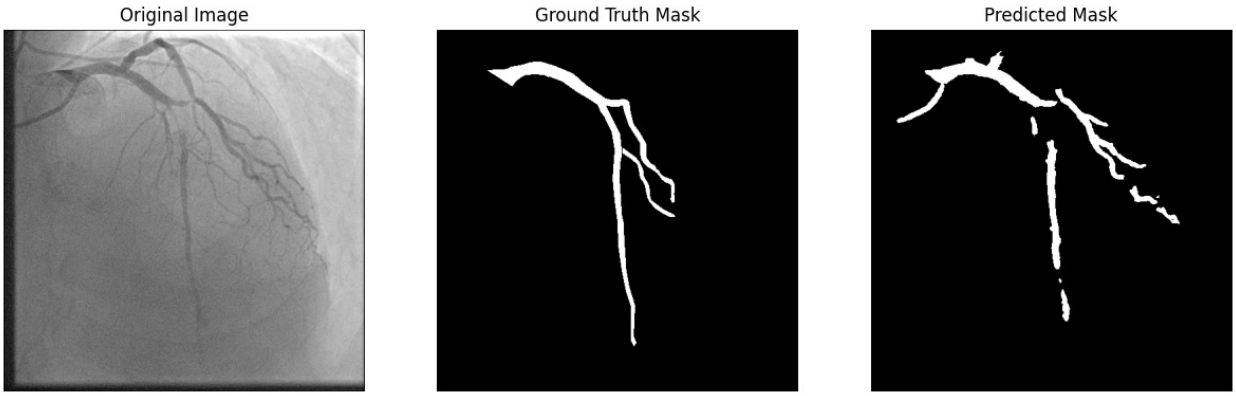
```
    plt.subplot(1, 3, 2)
    plt.plot(val_ious, label='Val IoU', color='orange')
    plt.title('IoU Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('IoU')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 3, 3)
    plt.plot(val_dices, label='Val Dice', color='green')
    plt.title('Dice Coefficient Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Dice')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

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

```
Train dataset size: 1000
Validation dataset size: 200
Using device: cuda
Training U-Net++...
Epoch 1/3, Batch 0/250, Train Loss: 0.7473
Epoch 1/3, Batch 50/250, Train Loss: 0.5526
Epoch 1/3, Batch 100/250, Train Loss: 0.5397
Epoch 1/3, Batch 150/250, Train Loss: 0.4325
Epoch 1/3, Batch 200/250, Train Loss: 0.3739
Epoch 1/3, Train Loss: 0.4901, Val Loss: 0.3513, Val IoU: 0.5101, Val
Dice: 0.6675
Saved best model at Epoch 1 with Val Loss: 0.3513
Epoch 2/3, Batch 0/250, Train Loss: 0.3237
Epoch 2/3, Batch 50/250, Train Loss: 0.2864
Epoch 2/3, Batch 100/250, Train Loss: 0.2430
Epoch 2/3, Batch 150/250, Train Loss: 0.2012
Epoch 2/3, Batch 200/250, Train Loss: 0.1692
Epoch 2/3, Train Loss: 0.2683, Val Loss: 0.2525, Val IoU: 0.4891, Val
Dice: 0.6394
Saved best model at Epoch 2 with Val Loss: 0.2525
Epoch 3/3, Batch 0/250, Train Loss: 0.2399
Epoch 3/3, Batch 50/250, Train Loss: 0.2251
Epoch 3/3, Batch 100/250, Train Loss: 0.2284
Epoch 3/3, Batch 150/250, Train Loss: 0.1292
Epoch 3/3, Batch 200/250, Train Loss: 0.1686
Epoch 3/3, Train Loss: 0.1980, Val Loss: 0.1931, Val IoU: 0.5769, Val
Dice: 0.7228
```

```
Saved best model at Epoch 3 with Val Loss: 0.1931
Training finished. Best model saved from Epoch 3 with Val Loss: 0.1931
Loaded best U-Net++ model for visualization.
Visualizing U-Net++ predictions...
Image 1: True mask sum: 10525.0, Predicted mask sum: 11672
```

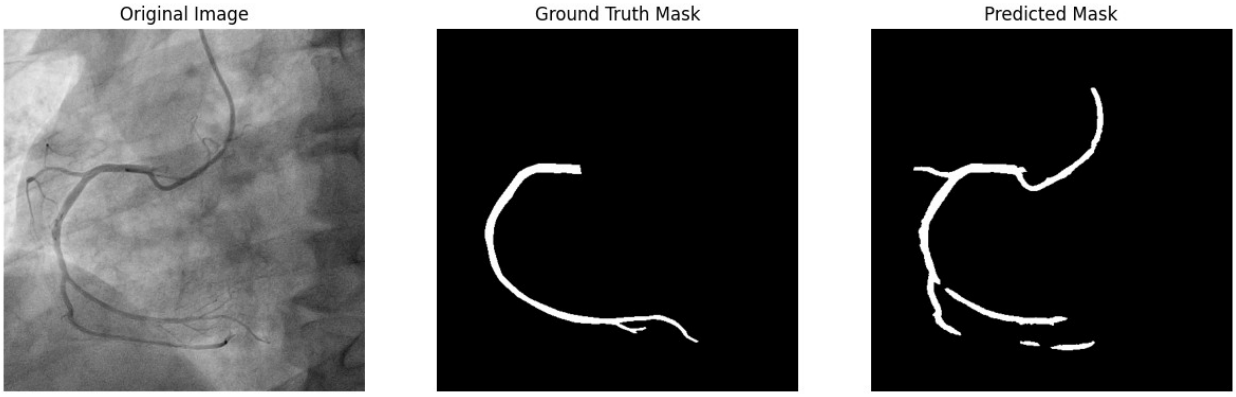



```
Image 2: True mask sum: 7860.0, Predicted mask sum: 9467
```




```
Image 3: True mask sum: 6711.0, Predicted mask sum: 4799
```

```
Image 4: True mask sum: 5394.0, Predicted mask sum: 8229
```




```
Image 5: True mask sum: 9790.0, Predicted mask sum: 11479
```