**Gradcam.py**

import torch

import numpy as np

import cv2

from PIL import Image

def get\_gradcam\_outputs(model, input\_tensor, class\_idx):

    model.eval()

    features = []

    gradients = []

    def forward\_hook(module, input, output):

        features.append(output)

    def backward\_hook(module, grad\_in, grad\_out):

        gradients.append(grad\_out[0])

    handle\_fwd = model.features[-1].register\_forward\_hook(forward\_hook)

    handle\_bwd = model.features[-1].register\_backward\_hook(backward\_hook)

    output = model(input\_tensor)

    model.zero\_grad()

    class\_score = output[0, class\_idx]

    class\_score.backward()

    handle\_fwd.remove()

    handle\_bwd.remove()

    grads = gradients[0].cpu().detach().numpy()[0]

    fmap = features[0].cpu().detach().numpy()[0]

    weights = np.mean(grads, axis=(1, 2))

    cam = np.zeros(fmap.shape[1:], dtype=np.float32)

    for i, w in enumerate(weights):

        cam += w \* fmap[i, :, :]

    cam = np.maximum(cam, 0)

    cam = cv2.resize(cam, (224, 224))

    cam = cam - np.min(cam)

    cam = cam / np.max(cam)

    heatmap = (cam \* 255).astype(np.uint8)

    heatmap\_img = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)

    overlay = input\_tensor.squeeze().permute(1, 2, 0).cpu().numpy()

    overlay = (overlay \* 0.5 + 0.5) \* 255

    overlay = overlay.astype(np.uint8)

    overlay\_img = cv2.addWeighted(overlay, 0.6, heatmap\_img, 0.4, 0)

    heatmap\_pil = Image.fromarray(heatmap\_img)

    overlay\_pil = Image.fromarray(overlay\_img)

    return heatmap\_pil, overlay\_pil

**model.py**

import torch

import torch.nn as nn

import torchvision.models as models

def build\_model(model\_path, device):

    model = models.efficientnet\_b0()

    model.classifier[1] = nn.Linear(model.classifier[1].in\_features, 2)

    model.load\_state\_dict(torch.load(model\_path, map\_location=device))

    model.to(device)

    model.eval()

    return model

**requirements.txt**

torch

torchvision

pillow

numpy

opencv-python

**script.py**

import argparse

import torch

import torchvision.transforms as transforms

from PIL import Image

import numpy as np

import cv2

import os

from model import build\_model

from utils import preprocess\_image

from gradcam import get\_gradcam\_outputs

LABELS = {0: "Sinus Rhythm", 1: "Atrial Fibrillation"}

def main(image\_path, model\_path="/var/www/script/Final\_Model.pth",filename="output1s.png"):

    if not os.path.isfile(image\_path):

        print(f"Image not found: {image\_path}")

        return

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

    model = build\_model(model\_path, device)

    with open(image\_path, "rb") as f:

        image\_bytes = f.read()

    input\_tensor, pil\_image = preprocess\_image(image\_bytes)

    input\_tensor = input\_tensor.to(device)

    with torch.no\_grad():

        output = model(input\_tensor)

        probs = torch.softmax(output, dim=1).cpu().numpy()[0]

        pred\_class = int(np.argmax(probs))

        confidence = float(np.max(probs))

    label = LABELS[pred\_class]

    print(f"Prediction:start {label}end;")

    print(f"Confidence:start {confidence \* 100:.2f}%end;")

    heatmap, overlay = get\_gradcam\_outputs(model, input\_tensor, pred\_class)

    # pil\_image.save("original.png")

    overlay\_path = f"/var/www/html/afsense.com/storage/app/public/overlay/{filename}"

    overlay.save(overlay\_path)

    print("Saved: original.png, /var/www/html/afsense.com/storage/app/public/overlay/{filename}")

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="ECG Classification CLI")

    parser.add\_argument("--image", type=str, required=True, help="Path to ECG image")

    parser.add\_argument("--model", type=str, default="Final\_Model.pth", help="Path to model .pth file")

    parser.add\_argument("--filename", type=str, required=True, help="filename")

    args = parser.parse\_args()

    main(args.image, args.model,args.filename)

**utils.py**

import torchvision.transforms as transforms

from PIL import Image

import io

import numpy as np

import cv2

def sharpen\_image(img, sigma=1.0):

    blurred = cv2.GaussianBlur(img, (5, 5), sigma)

    return cv2.addWeighted(img, 1.5, blurred, -0.5, 0)

def binarize\_image(img):

    img = np.uint8(img)

    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

    img\_clahe = clahe.apply(img)

    return img\_clahe

def pad\_image(image):

    size = max(image.shape[:2])

    top = (size - image.shape[0]) // 2

    bottom = size - image.shape[0] - top

    left = (size - image.shape[1]) // 2

    right = size - image.shape[1] - left

    return cv2.copyMakeBorder(image, top, bottom, left, right, borderType=cv2.BORDER\_CONSTANT, value=[255,255,255])

def denoise\_image(img):

    return np.uint8(cv2.fastNlMeansDenoising(img, None, h=8, templateWindowSize=11, searchWindowSize=21))

def erode\_image(img, kernel\_size=(3, 3), iterations=1):

    kernel = np.ones(kernel\_size, np.uint8)

    return np.uint8(cv2.erode(img, kernel, iterations=iterations))

def preprocess\_image(image\_bytes):

    pil = Image.open(io.BytesIO(image\_bytes)).convert("RGB")

    img = np.array(pil)

    if len(img.shape) == 3:

        img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    img = pad\_image(img)

    img = denoise\_image(img)

    img = erode\_image(img)

    img = sharpen\_image(img)

    img = np.clip(img, 0, 255).astype(np.uint8)

    img = cv2.resize(img, (224, 224))

    img = Image.fromarray(img).convert("RGB")

    tensor = transforms.ToTensor()(img).unsqueeze(0)

    tensor = transforms.Normalize(mean=[0.5], std=[0.5])(tensor)

    return tensor, img