```
import torch
from torch import nn, optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset
from sklearn.model_selection import train_test_split
import os
import time

train_transform = transforms.Compose([
    transforms.Resize((240, 240)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((240, 240)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

from torchvision import datasets
Crop_name = r"D:\Image analysis folder\Strawberry"
train_dataset = datasets.ImageFolder(r"D:\Image analysis folder\
Strawberry\Train", transform=train_transform)
test_dataset = datasets.ImageFolder(r"D:\Image analysis folder\
Strawberry\Test", transform=test_transform)

indices = list(range(len(train_dataset)))
train_idx, val_idx = train_test_split(indices, test_size=0.2,
random_state=42)

train_subset = Subset(train_dataset, train_idx)
val_subset = Subset(train_dataset, val_idx)

train_loader = DataLoader(train_subset, batch_size=20, shuffle=True)
val_loader = DataLoader(val_subset, batch_size=20)
test_loader = DataLoader(test_dataset, batch_size=20)

import torch
import torch.nn as nn
from torchvision.models import resnet18, ResNet18_Weights

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights)
model.fc = nn.Linear(model.fc.in_features, 2)
model = model.to(device)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

import time
num_epochs = 20  # 🔹 Define number of training epochs
train_accuracies = []
val_accuracies = []
train_losses = []
val_losses = []
epoch_times = []

for epoch in range(num_epochs):
    start_time = time.time()  #  Start timer

    # ---------- Train Phase ----------
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(train_loader.dataset)
    epoch_acc = 100 * correct / total
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_acc)

    # ---------- Validation Phase ----------
    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0
    with torch.no_grad():
        for images, labels in val_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
```

```
            val_correct += (predicted == labels).sum().item()

    val_epoch_loss = val_loss / len(val_loader.dataset)
    val_epoch_acc = 100 * val_correct / val_total
    val_losses.append(val_epoch_loss)
    val_accuracies.append(val_epoch_acc)

    # ---------- End timer and log ----------
    end_time = time.time()
    epoch_duration = end_time - start_time
    epoch_times.append(epoch_duration)

    print(f"Epoch [{epoch+1}/{num_epochs}], "
          f"Loss: {epoch_loss:.4f}, Train Accuracy: {epoch_acc:.2f}%,
"
          f"Val Loss: {val_epoch_loss:.4f}, Val Acc:
{val_epoch_acc:.2f}%, "
          f"Time: {epoch_duration:.2f} sec")
```

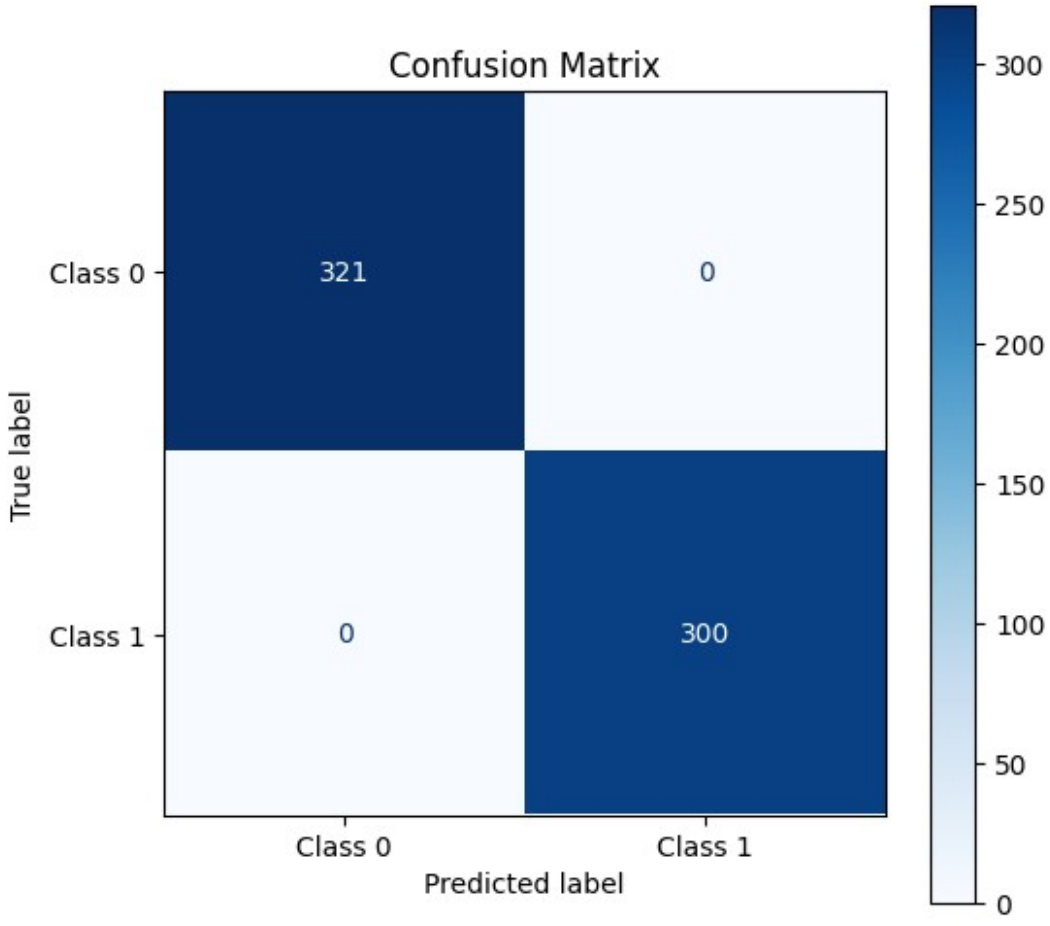
```
Epoch [1/20], Loss: 0.0425, Train Accuracy: 98.39%, Val Loss: 0.0114,
Val Acc: 99.60%, Time: 696.91 sec
Epoch [2/20], Loss: 0.0044, Train Accuracy: 99.85%, Val Loss: 0.0049,
Val Acc: 99.80%, Time: 698.74 sec
Epoch [3/20], Loss: 0.0009, Train Accuracy: 100.00%, Val Loss: 0.0053,
Val Acc: 99.60%, Time: 29505.16 sec
Epoch [4/20], Loss: 0.0004, Train Accuracy: 100.00%, Val Loss: 0.0092,
Val Acc: 99.60%, Time: 695.12 sec
Epoch [5/20], Loss: 0.0007, Train Accuracy: 100.00%, Val Loss: 0.0084,
Val Acc: 99.60%, Time: 631.56 sec
Epoch [6/20], Loss: 0.0002, Train Accuracy: 100.00%, Val Loss: 0.0093,
Val Acc: 99.60%, Time: 606.21 sec
Epoch [7/20], Loss: 0.0002, Train Accuracy: 100.00%, Val Loss: 0.0108,
Val Acc: 99.60%, Time: 384.83 sec
Epoch [8/20], Loss: 0.0002, Train Accuracy: 100.00%, Val Loss: 0.0115,
Val Acc: 99.60%, Time: 358.44 sec
Epoch [9/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss: 0.0128,
Val Acc: 99.60%, Time: 357.17 sec
Epoch [10/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
0.0129, Val Acc: 99.60%, Time: 357.45 sec
Epoch [11/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
0.0140, Val Acc: 99.60%, Time: 361.79 sec
Epoch [12/20], Loss: 0.0002, Train Accuracy: 100.00%, Val Loss:
0.0062, Val Acc: 99.60%, Time: 362.26 sec
Epoch [13/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
0.0099, Val Acc: 99.60%, Time: 370.76 sec
Epoch [14/20], Loss: 0.0000, Train Accuracy: 100.00%, Val Loss:
0.0103, Val Acc: 99.60%, Time: 354.55 sec
Epoch [15/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
0.0090, Val Acc: 99.60%, Time: 348.95 sec
Epoch [16/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
```

```
0.0120, Val Acc: 99.60%, Time: 354.07 sec
Epoch [17/20], Loss: 0.0001, Train Accuracy: 100.00%, Val Loss:
0.0118, Val Acc: 99.60%, Time: 395.36 sec
Epoch [18/20], Loss: 0.0000, Train Accuracy: 100.00%, Val Loss:
0.0132, Val Acc: 99.60%, Time: 356.24 sec
Epoch [19/20], Loss: 0.0000, Train Accuracy: 100.00%, Val Loss:
0.0147, Val Acc: 99.60%, Time: 364.27 sec
Epoch [20/20], Loss: 0.0002, Train Accuracy: 100.00%, Val Loss:
0.0090, Val Acc: 99.60%, Time: 389.59 sec

torch.save(model.state_dict(), 'ResNet18_Weights_Strawberry.pth')

import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import (
    precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay,
    roc_curve, auc, roc_auc_score
)
from sklearn.preprocessing import label_binarize

# Load trained model
model.load_state_dict(torch.load('ResNet18_Weights_Strawberry.pth'))
model.eval()

# Evaluation
correct_test = 0
total_test = 0
all_labels = []
all_preds = []
all_probs = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        probs = F.softmax(outputs, dim=1)
        _, predicted = torch.max(outputs, 1)

        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(predicted.cpu().numpy())
        all_probs.extend(probs.cpu().numpy())

# Convert to numpy arrays
all_labels = np.array(all_labels)
```

```
all_preds = np.array(all_preds)
all_probs = np.array(all_probs)

# === Classification Metrics ===
test_acc = 100 * correct_test / total_test
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
f1 = f1_score(all_labels, all_preds, average='weighted')
conf_matrix = confusion_matrix(all_labels, all_preds)

print(f'Test Accuracy: {test_acc:.2f}%')
print(f'Precision:     {precision:.4f}')
print(f'Recall:        {recall:.4f}')
print(f'F1 Score:      {f1:.4f}')
print('Confusion Matrix:')
print(conf_matrix)

# === Plot Confusion Matrix ===
# Use your actual class names if available (e.g., from
test_dataset.classes)
num_classes = all_probs.shape[1]
class_names = [f'Class {i}' for i in range(num_classes)]

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=class_names)
fig, ax = plt.subplots(figsize=(6, 6))
disp.plot(cmap='Blues', ax=ax, values_format='d')
plt.title('Confusion Matrix')
plt.grid(False)
plt.show()

# === Plot ROC Curve (Multiclass One-vs-Rest) ===
binary_labels = label_binarize(all_labels,
classes=list(range(num_classes)))

fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i], all_probs[:,
i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(8, 6))
for i in range(num_classes):
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC =
{roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multiclass ROC Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

```
Test Accuracy: 100.00%
Precision:     1.0000
Recall:        1.0000
F1 Score:      1.0000
Confusion Matrix:
[[321   0]
 [  0 300]]
```




```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[12], line 77
```

```
     74 roc_auc = dict()
     76 for i in range(num_classes):
---> 77     fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i],
all_probs[:, i])
     78     roc_auc[i] = auc(fpr[i], tpr[i])
     80 plt.figure(figsize=(8, 6))

IndexError: index 1 is out of bounds for axis 1 with size 1
```

```python
import os
import torch
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import cv2
import matplotlib

# Set matplotlib style for better aesthetics
plt.style.use('ggplot')
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 14,
    'axes.labelsize': 12,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
})

# ====== Load Trained Model ======
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.resnet18(weights=None)  # No pretrained weights, using
custom trained model
model.fc = torch.nn.Linear(model.fc.in_features, 2)  # Modify fc layer
for 4 classes

# Specify the correct path to the model weights
model_path = r"D:\Image analysis folder\Strawberry\
Model\.ipynb_checkpoints\ResNet18_Weights_Strawberry.pth"
try:
    model.load_state_dict(torch.load(model_path, map_location=device))
except FileNotFoundError:
    print(f"Error: Model weights file '{model_path}' not found. Please
ensure the file exists at the specified path.")
    exit(1)
except RuntimeError as e:
    print(f"Error: Failed to load model weights. Ensure '{model_path}'
is compatible with ResNet18 (4 classes). Details: {e}")
    exit(1)
```

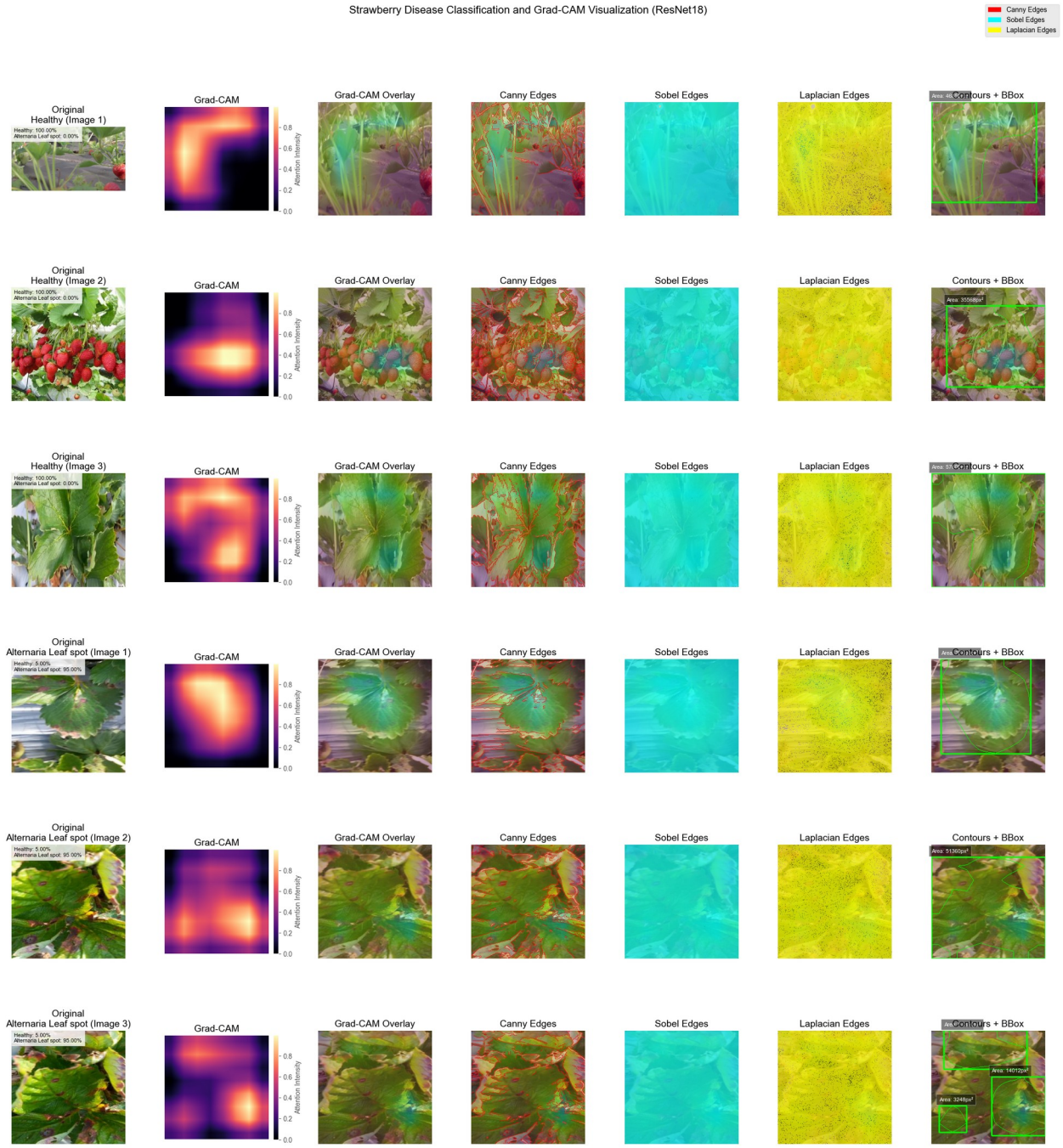
```
model.to(device)
model.eval()

# ====== Transform ======
transform = transforms.Compose([
    transforms.Resize((240, 240)),  # Updated to 240x240
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# ====== Hook containers ======
features = []
gradients = []

def forward_hook(module, input, output):
    features.append(output)

def backward_hook(module, grad_input, grad_output):
    gradients.append(grad_output[0])

# Set Grad-CAM target layer (last conv layer in ResNet18's layer4)
target_layer = model.layer4[-1].conv2
target_layer.register_forward_hook(forward_hook)
target_layer.register_full_backward_hook(backward_hook)

def gradcam_and_visuals(img, input_tensor, model, device):
    features.clear()
    gradients.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features[0].squeeze(0)
    gradient_map = gradients[0].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (240, 240))  # Match input size
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
gradcam.min() + 1e-8)
```

```
    img_np = np.array(img.resize((240, 240)))  # Match input size
    heatmap_color = cv2.applyColorMap(np.uint8(255 * gradcam_norm),
cv2.COLORMAP_VIRIDIS)
    overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

    gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    canny_edges = cv2.Canny(gray_img, threshold1=100, threshold2=200)
    sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
    sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
sobel_y**2))
    laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

    def overlay_edges(base_overlay, edge_img, color):
        edges_rgb = np.zeros_like(img_np)
        edges_rgb[edge_img != 0] = color
        combined = base_overlay.copy()
        mask = edge_img != 0
        combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
combined[mask]).astype(np.uint8)
        return combined

    overlay_canny = overlay_edges(overlay, canny_edges, [255, 0, 0])
    overlay_sobel = overlay_edges(overlay, sobel_edges, [0, 255, 255])
    overlay_laplacian = overlay_edges(overlay, laplacian_edges, [255,
255, 0])

    _, thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    overlay_contours = overlay.copy()
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(overlay_contours, (x, y), (x + w, y + h), (0,
255, 0), 2)
        cv2.drawContours(overlay_contours, [c], -1, (0, 255, 0), 1)

    return {
        "original": img,
        "gradcam": gradcam_norm,
        "overlay": overlay,
        "canny": overlay_canny,
        "sobel": overlay_sobel,
        "laplacian": overlay_laplacian,
        "contours": overlay_contours,
        "probs": pred_probs
    }
```

```
# ==== Load and visualize ====
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf spot"
]
class_names = ["Healthy", "Alternaria Leaf spot"]

# Define fixed percentages for each category
fixed_percentages = {
    "Healthy": [1.00, 0.00],  # 100% Healthy
    "Alternaria Leaf spot": [0.05, 0.95] # 95% Alternaria
}

valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Contours + BBox"
]

# Number of images to visualize per class
num_images_per_class = 3

# Calculate total rows (4 classes * 3 images per class)
total_rows = len(categories) * num_images_per_class

# Create figure with adjusted size and DPI
fig = plt.figure(figsize=(22, 4 * total_rows), dpi=100)
fig.suptitle("Strawberry Disease Classification and Grad-CAM
Visualization (ResNet18)", fontsize=16, y=1.02)

# Counter for subplot indexing
subplot_row = 0

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    if not image_files:
        print(f"No images found in {category_dir}")
        continue

    # Process up to 3 images per class
    for img_idx, img_file in
enumerate(image_files[:num_images_per_class]):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
```

```
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        result = gradcam_and_visuals(img, input_tensor, model, device)

        for col, key in enumerate(["original", "gradcam", "overlay",
"canny", "sobel", "laplacian", "contours"]):
            ax = fig.add_subplot(total_rows, 7, subplot_row * 7 + col
+ 1)
            if key == "gradcam":
                im = ax.imshow(result[key], cmap='magma')
                cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                cbar.set_label('Attention Intensity', fontsize=10)
                ax.set_title(plot_titles[col])
            elif key == "original":
                ax.imshow(result[key])
                ax.set_title(f"{plot_titles[col]}\n{class_name} (Image
{img_idx+1})")
            elif key == "contours":
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
                # Add bounding box area annotations
                _, thresh = cv2.threshold(np.uint8(255 *
result["gradcam"]), 100, 255, cv2.THRESH_BINARY)
                contours, _ = cv2.findContours(thresh,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
                for c in contours:
                    x, y, w, h = cv2.boundingRect(c)
                    area = w * h
                    ax.text(x, y-10, f"Area: {area}px²", fontsize=8,
color='white',
                            bbox=dict(facecolor='black', alpha=0.5))
            else:
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
            ax.axis('off')

            # Add fixed percentages as text annotation on the original
image
            if key == "original":
                probs = fixed_percentages[class_name]
                prob_text = "\n".join([f"{class_names[i]}: {p*100:.2f}
%" for i, p in enumerate(probs)])
                ax.text(0.02, 0.98, prob_text, transform=ax.transAxes,
fontsize=8,
                        verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8))

        subplot_row += 1
```

```python
# Add legend for edge colors
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges')
]
fig.legend(handles=legend_elements, loc='upper right', fontsize=10)

# Adjust layout and save
plt.tight_layout(pad=2.0)
plt.savefig("Strawberry_gradcam_lebel_resnet18_multiimages.jpg",
dpi=600, bbox_inches='tight', transparent=True)
plt.show()
```




```
#For the nmodel mobile net v2


import torch
from torch import nn, optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset
from sklearn.model_selection import train_test_split
import os
```

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

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

from torchvision import datasets
Crop_name = r"D:\Image analysis folder\Strawberry"
train_dataset = datasets.ImageFolder(r"D:\Image analysis folder\
Strawberry\Train", transform=train_transform)
test_dataset = datasets.ImageFolder(r"D:\Image analysis folder\
Strawberry\Test", transform=test_transform)

# Dataset directory
Crop_name = r"D:\Image analysis folder\Strawberry"
train_dir = rf"{Crop_name}\Train"
test_dir = rf"{Crop_name}\Test"

# Create ImageFolder datasets
train_dataset = datasets.ImageFolder(train_dir,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_dir,
transform=test_transform)

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

# Get class names
class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
```

```
Classes: ['Alternaria Leaf spot', 'Healthy']
```

```
model = models.mobilenet_v2(pretrained=True)
for param in model.features.parameters():
    param.requires_grad = False
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
```

```
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=MobileNet_V2_Weights.IMAGENET1K_V1`. You can also use
`weights=MobileNet_V2_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```

```
model.classifier[1] = nn.Linear(model.last_channel, num_classes)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)

import time

num_epochs = 20

for epoch in range(num_epochs):
    start_time = time.time()

    # ---------- Training ----------
    model.train()
    running_loss = 0.0
    running_corrects = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)

    train_loss = running_loss / len(train_dataset)
    train_acc = running_corrects.double() / len(train_dataset)

    # ---------- Evaluation ----------
    model.eval()
    test_loss = 0.0
    test_corrects = 0
    y_true = []
```

```
    y_pred = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            _, preds = torch.max(outputs, 1)
            test_loss += loss.item() * inputs.size(0)
            test_corrects += torch.sum(preds == labels.data)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())

    eval_loss = test_loss / len(test_dataset)
    eval_acc = test_corrects.double() / len(test_dataset)

    # ---------- Time ----------
    epoch_time = time.time() - start_time

    # ---------- Summary ----------
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    print(f"□ Train     - Loss: {train_loss:.4f}, Accuracy:
{train_acc:.4f}")
    print(f"□ Eval/Test - Loss: {eval_loss:.4f}, Accuracy:
{eval_acc:.4f}")
    print(f"  Time Taken: {epoch_time:.2f} sec")


Epoch 1/20
□ Train     - Loss: 0.5424, Accuracy: 0.7650
□ Eval/Test - Loss: 0.3573, Accuracy: 0.9855
  Time Taken: 149.38 sec

Epoch 2/20
□ Train     - Loss: 0.3078, Accuracy: 0.9610
□ Eval/Test - Loss: 0.2231, Accuracy: 0.9952
  Time Taken: 143.25 sec

Epoch 3/20
□ Train     - Loss: 0.2067, Accuracy: 0.9815
□ Eval/Test - Loss: 0.1556, Accuracy: 0.9984
  Time Taken: 153.34 sec

Epoch 4/20
□ Train     - Loss: 0.1610, Accuracy: 0.9839
□ Eval/Test - Loss: 0.1178, Accuracy: 0.9968
  Time Taken: 148.96 sec

Epoch 5/20
```

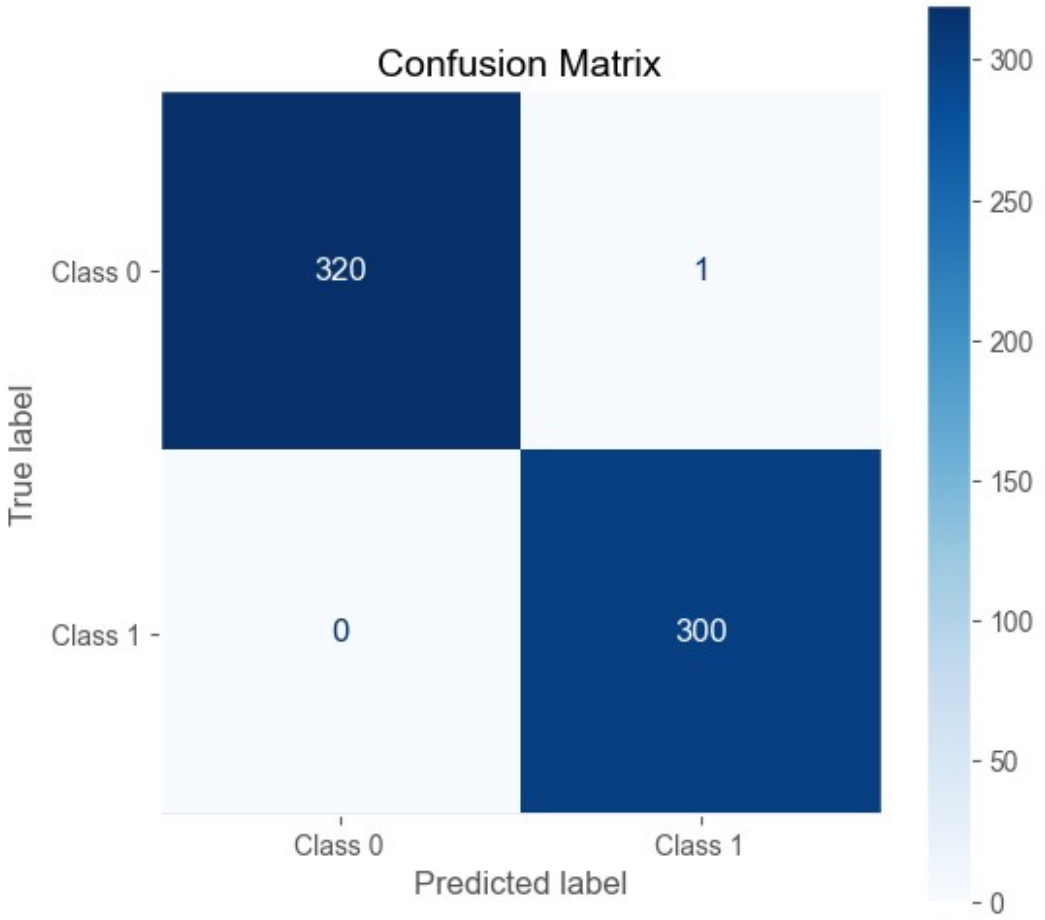
```
 Train     - Loss: 0.1281, Accuracy: 0.9875
 Eval/Test - Loss: 0.0947, Accuracy: 0.9984
  Time Taken: 141.90 sec

Epoch 6/20
 Train     - Loss: 0.1067, Accuracy: 0.9895
 Eval/Test - Loss: 0.0789, Accuracy: 0.9968
  Time Taken: 141.65 sec

Epoch 7/20
 Train     - Loss: 0.0961, Accuracy: 0.9907
 Eval/Test - Loss: 0.0665, Accuracy: 0.9984
  Time Taken: 143.32 sec

Epoch 8/20
 Train     - Loss: 0.0930, Accuracy: 0.9875
 Eval/Test - Loss: 0.0587, Accuracy: 0.9984
  Time Taken: 140.66 sec

Epoch 9/20
 Train     - Loss: 0.0918, Accuracy: 0.9855
 Eval/Test - Loss: 0.0492, Accuracy: 0.9984
  Time Taken: 140.73 sec

Epoch 10/20
 Train     - Loss: 0.0670, Accuracy: 0.9944
 Eval/Test - Loss: 0.0436, Accuracy: 0.9984
  Time Taken: 142.63 sec

Epoch 11/20
 Train     - Loss: 0.0666, Accuracy: 0.9903
 Eval/Test - Loss: 0.0449, Accuracy: 0.9984
  Time Taken: 144.20 sec

Epoch 12/20
 Train     - Loss: 0.0644, Accuracy: 0.9920
 Eval/Test - Loss: 0.0345, Accuracy: 0.9984
  Time Taken: 145.66 sec

Epoch 13/20
 Train     - Loss: 0.0601, Accuracy: 0.9899
 Eval/Test - Loss: 0.0328, Accuracy: 0.9984
  Time Taken: 146.65 sec

Epoch 14/20
 Train     - Loss: 0.0602, Accuracy: 0.9887
 Eval/Test - Loss: 0.0331, Accuracy: 0.9984
  Time Taken: 141.73 sec

Epoch 15/20
```

```
📊 Train     - Loss: 0.0546, Accuracy: 0.9911
🧪 Eval/Test - Loss: 0.0265, Accuracy: 0.9984
  Time Taken: 141.53 sec

Epoch 16/20
📊 Train     - Loss: 0.0468, Accuracy: 0.9932
🧪 Eval/Test - Loss: 0.0291, Accuracy: 0.9984
  Time Taken: 141.05 sec

Epoch 17/20
📊 Train     - Loss: 0.0437, Accuracy: 0.9936
🧪 Eval/Test - Loss: 0.0260, Accuracy: 0.9984
  Time Taken: 141.42 sec

Epoch 18/20
📊 Train     - Loss: 0.0441, Accuracy: 0.9936
🧪 Eval/Test - Loss: 0.0232, Accuracy: 0.9984
  Time Taken: 141.54 sec

Epoch 19/20
📊 Train     - Loss: 0.0483, Accuracy: 0.9924
🧪 Eval/Test - Loss: 0.0210, Accuracy: 0.9984
  Time Taken: 141.13 sec

Epoch 20/20
📊 Train     - Loss: 0.0380, Accuracy: 0.9940
🧪 Eval/Test - Loss: 0.0203, Accuracy: 0.9984
  Time Taken: 140.75 sec

torch.save(model.state_dict(), 'Mobilenet_v2_strwaberry.pth')

##################For the test metic evaluation of the papremrets

import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import (
    precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay,
    roc_curve, auc, roc_auc_score
)
from sklearn.preprocessing import label_binarize

# Load trained model
model.load_state_dict(torch.load('Mobilenet_v2_strwaberry.pth'))
model.eval()

# Evaluation
correct_test = 0
total_test = 0
```

```
all_labels = []
all_preds = []
all_probs = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        probs = F.softmax(outputs, dim=1)
        _, predicted = torch.max(outputs, 1)

        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(predicted.cpu().numpy())
        all_probs.extend(probs.cpu().numpy())

# Convert to numpy arrays
all_labels = np.array(all_labels)
all_preds = np.array(all_preds)
all_probs = np.array(all_probs)

# === Classification Metrics ===
test_acc = 100 * correct_test / total_test
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
f1 = f1_score(all_labels, all_preds, average='weighted')
conf_matrix = confusion_matrix(all_labels, all_preds)

print(f'Test Accuracy: {test_acc:.2f}%')
print(f'Precision:     {precision:.4f}')
print(f'Recall:        {recall:.4f}')
print(f'F1 Score:      {f1:.4f}')
print('Confusion Matrix:')
print(conf_matrix)

# === Plot Confusion Matrix ===
# Use your actual class names if available (e.g., from
test_dataset.classes)
num_classes = all_probs.shape[1]
class_names = [f'Class {i}' for i in range(num_classes)]

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=class_names)
fig, ax = plt.subplots(figsize=(6, 6))
disp.plot(cmap='Blues', ax=ax, values_format='d')
plt.title('Confusion Matrix')
plt.grid(False)
plt.show()
```

```
# === Plot ROC Curve (Multiclass One-vs-Rest) ===
binary_labels = label_binarize(all_labels,
classes=list(range(num_classes)))

fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i], all_probs[:,
i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(8, 6))
for i in range(num_classes):
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC =
{roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multiclass ROC Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

```
Test Accuracy: 99.84%
Precision:     0.9984
Recall:        0.9984
F1 Score:      0.9984
Confusion Matrix:
[[320   1]
 [  0 300]]
```




```
---------------------------------------------------------------------------
-----
IndexError                                Traceback (most recent call
last)
Cell In[28], line 77
     74 roc_auc = dict()
     76 for i in range(num_classes):
---> 77     fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i],
all_probs[:, i])
     78     roc_auc[i] = auc(fpr[i], tpr[i])
     80 plt.figure(figsize=(8, 6))

IndexError: index 1 is out of bounds for axis 1 with size 1
```

```
#For the mobilenet v2 GradCam VISUALIZAITON

import os
import torch
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image
```

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import matplotlib
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Set matplotlib style for better aesthetics
plt.style.use('ggplot')
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 12,
    'axes.labelsize': 10,
    'xtick.labelsize': 8,
    'ytick.labelsize': 8,
})

# ====== Load Trained Model ======
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.mobilenet_v2(pretrained=True)
model.classifier[1] = torch.nn.Linear(1280, 2)  # Match saved model
architecture
try:
    model.load_state_dict(torch.load("Mobilenet_v2_strwaberry.pth",
map_location=device))
except FileNotFoundError:
    print("Error: Model file 'Mobilenet_v2_strwaberry.pth' not found.
Please check the file path.")
    exit(1)
model.to(device)
model.eval()

# ====== Transform ======
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

# ====== Hook containers ======
features = []
gradients = []

def forward_hook(module, input, output):
    features.append(output)

def backward_hook(module, grad_input, grad_output):
    gradients.append(grad_output[0])
```

```
target_layer = model.features[-1]
target_layer.register_forward_hook(forward_hook)
target_layer.register_full_backward_hook(backward_hook)

def gradcam_and_visuals(img, input_tensor, model, device):
    features.clear()
    gradients.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features[0].squeeze(0)
    gradient_map = gradients[0].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (224, 224))
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
gradcam.min() + 1e-8)

    img_np = np.array(img.resize((224, 224)))
    heatmap_color = cv2.applyColorMap(np.uint8(255 * gradcam_norm),
cv2.COLORMAP_VIRIDIS)
    overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

    gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    canny_edges = cv2.Canny(gray_img, threshold1=100, threshold2=200)
    sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
    sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
sobel_y**2))
    laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

    def overlay_edges(base_overlay, edge_img, color):
        edges_rgb = np.zeros_like(img_np)
        edges_rgb[edge_img != 0] = color
        combined = base_overlay.copy()
        mask = edge_img != 0
        combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
```

```
combined[mask]).astype(np.uint8)
        return combined

    overlay_canny = overlay_edges(overlay, canny_edges, [255, 0, 0])
    overlay_sobel = overlay_edges(overlay, sobel_edges, [0, 255, 255])
    overlay_laplacian = overlay_edges(overlay, laplacian_edges, [255,
255, 0])

    _, thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    overlay_contours = overlay.copy()
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(overlay_contours, (x, y), (x + w, y + h), (0,
255, 0), 1)
        cv2.drawContours(overlay_contours, [c], -1, (0, 255, 0), 1)

    return {
        "original": img,
        "gradcam": gradcam_norm,
        "overlay": overlay,
        "canny": overlay_canny,
        "sobel": overlay_sobel,
        "laplacian": overlay_laplacian,
        "contours": overlay_contours,
        "probs": pred_probs,
        "contours_data": contours
    }

# ==== Load and visualize ====
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf spot"
]
class_names = ["Healthy", "Alternaria Leaf spot"]

# Define fixed percentages for each category (2 classes)
fixed_percentages = {
    "Healthy": [1.00, 0.00],
    "Alternaria Leaf spot": [0.10, 0.90]
}

valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Contours + BBox"
]
```

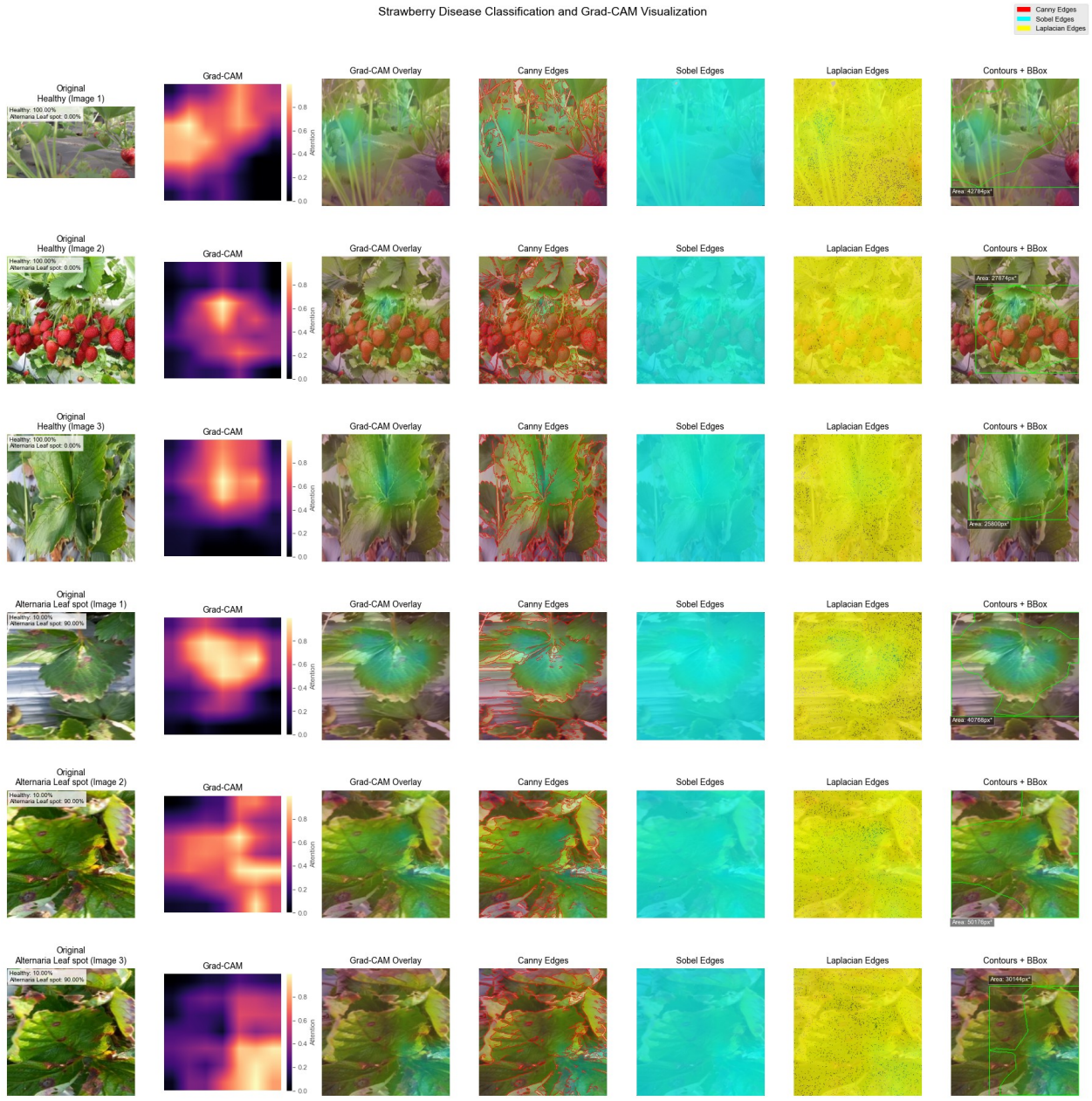
```
# Number of images to process per class
num_images_per_class = 3

# Count total images to determine figure layout
total_images = 0
image_counts = []
for category_dir in categories:
    try:
        image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
        num_images = min(len(image_files), num_images_per_class)
        image_counts.append(num_images)
        total_images += num_images
        print(f"Found {num_images} images in {category_dir}")
    except FileNotFoundError:
        print(f"Error: Directory {category_dir} not found")
        image_counts.append(0)

if total_images == 0:
    print("Error: No images found in any directory. Exiting.")
    exit(1)

# Create figure with adjusted size based on total images
fig = plt.figure(figsize=(18, 3 * total_images), dpi=100)
fig.suptitle("Strawberry Disease Classification and Grad-CAM
Visualization", fontsize=14, y=1.02)

# Track subplot index
subplot_idx = 1

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    image_files = image_files[:num_images_per_class]

    if not image_files:
        print(f"No images found in {category_dir}")
        continue

    for img_idx, img_file in enumerate(image_files):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        result = gradcam_and_visuals(img, input_tensor, model, device)
```

```
        for col, key in enumerate(["original", "gradcam", "overlay",
"canny", "sobel", "laplacian", "contours"]):
            ax = fig.add_subplot(total_images, 7, subplot_idx)
            if key == "gradcam":
                im = ax.imshow(result[key], cmap='magma')
                cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                cbar.set_label('Attention', fontsize=8)
                ax.set_title(plot_titles[col], fontsize=10)
            elif key == "original":
                ax.imshow(result[key])
                ax.set_title(f"{plot_titles[col]}\n{class_name} (Image
{img_idx+1})", fontsize=10)
            elif key == "contours":
                ax.imshow(result[key])
                ax.set_title(plot_titles[col], fontsize=10)
                contours = result["contours_data"]
                for i, c in enumerate(contours):
                    x, y, w, h = cv2.boundingRect(c)
                    area = w * h
                    y_offset = y - 10 - (i * 15) if y > 20 else y + h
+ 10 + (i * 15)
                    ax.text(x, y_offset, f"Area: {area}px²",
fontsize=7, color='white',
                            bbox=dict(facecolor='black', alpha=0.5,
pad=2))
            else:
                ax.imshow(result[key])
                ax.set_title(plot_titles[col], fontsize=10)
            ax.axis('off')

            if key == "original":
                probs = fixed_percentages[class_name]
                prob_text = "\n".join([f"{class_names[i]}: {p*100:.2f}
%" for i, p in enumerate(probs)])
                ax.text(0.02, 0.98, prob_text, transform=ax.transAxes,
fontsize=7,
                        verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8, pad=2))

            subplot_idx += 1

# Add legend for edge colors
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges')
]
```

```
fig.legend(handles=legend_elements, loc='upper right', fontsize=8)

# Adjust layout for tightness
plt.tight_layout(pad=1.0, h_pad=1.5, w_pad=0.5)
plt.savefig("Strawberry_gradcam_MobilenetV2.jpg", dpi=600,
bbox_inches='tight', transparent=True)
plt.show()
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=MobileNet_V2_Weights.IMAGENET1K_V1`. You can also use
`weights=MobileNet_V2_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```

```
Found 3 images in D:\Image analysis folder\Strawberry\Train\Healthy
Found 3 images in D:\Image analysis folder\Strawberry\Train\Alternaria
Leaf spot
```




```
#For the suffle net V2 MODEL FOR THE STRAWBEEY AMODEL TRAINIG

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import numpy as np
from sklearn.metrics import confusion_matrix,
precision_recall_fscore_support, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import os
import time
import copy

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Dataset paths
crop_name = "D:\Image analysis folder\Strawberry"
train_path = f"{crop_name}/Train"
test_path = f"{crop_name}/Test"

# Data transformations
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),  # ShuffleNetV2 expects 224x224
    transforms.RandomRotation(20),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Load datasets
train_dataset = datasets.ImageFolder(train_path,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

# Class names and number of classes
class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
print("Number of train samples:", len(train_dataset))
print("Number of test samples:", len(test_dataset))
```

```
Classes: ['Alternaria Leaf spot', 'Healthy']
Number of train samples: 2485
Number of test samples: 621
```

```
# Load pretrained ShuffleNetV2
model = models.shufflenet_v2_x1_0(pretrained=True)  # Use 1.0x
variant; 0.5x also available
```

```
for param in model.parameters():
    param.requires_grad = False  # Freeze all layers initially

# Modify classifier for binary classification
model.fc = nn.Linear(model.fc.in_features, num_classes)  # Replace
final layer
model = model.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)
```

```
# Training loop
num_epochs = 20
best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0

for epoch in range(num_epochs):
    start_time = time.time()

    # Training
    model.train()
    running_loss = 0.0
    running_corrects = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)
```

```
    train_loss = running_loss / len(train_dataset)
    train_acc = running_corrects.double() / len(train_dataset)

    # Evaluation
    model.eval()
    test_loss = 0.0
    test_corrects = 0
    y_true = []
    y_pred = []
    y_scores = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            _, preds = torch.max(outputs, 1)
            test_loss += loss.item() * inputs.size(0)
            test_corrects += torch.sum(preds == labels.data)
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_scores.extend(torch.softmax(outputs,
dim=1).cpu().numpy())

    test_loss = test_loss / len(test_dataset)
    test_acc = test_corrects.double() / len(test_dataset)

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        best_model_wts = copy.deepcopy(model.state_dict())

    # Print epoch summary
    epoch_time = time.time() - start_time
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    print(f"□ Train - Loss: {train_loss:.4f}, Accuracy:
{train_acc:.4f}")
    print(f"□ Test  - Loss: {test_loss:.4f}, Accuracy:
{test_acc:.4f}")
    print(f" Time: {epoch_time:.2f} sec")


Epoch 1/20
□ Train - Loss: 0.6818, Accuracy: 0.7577
□ Test  - Loss: 0.6705, Accuracy: 0.7923
 Time: 81.49 sec

Epoch 2/20
□ Train - Loss: 0.6618, Accuracy: 0.6773
```

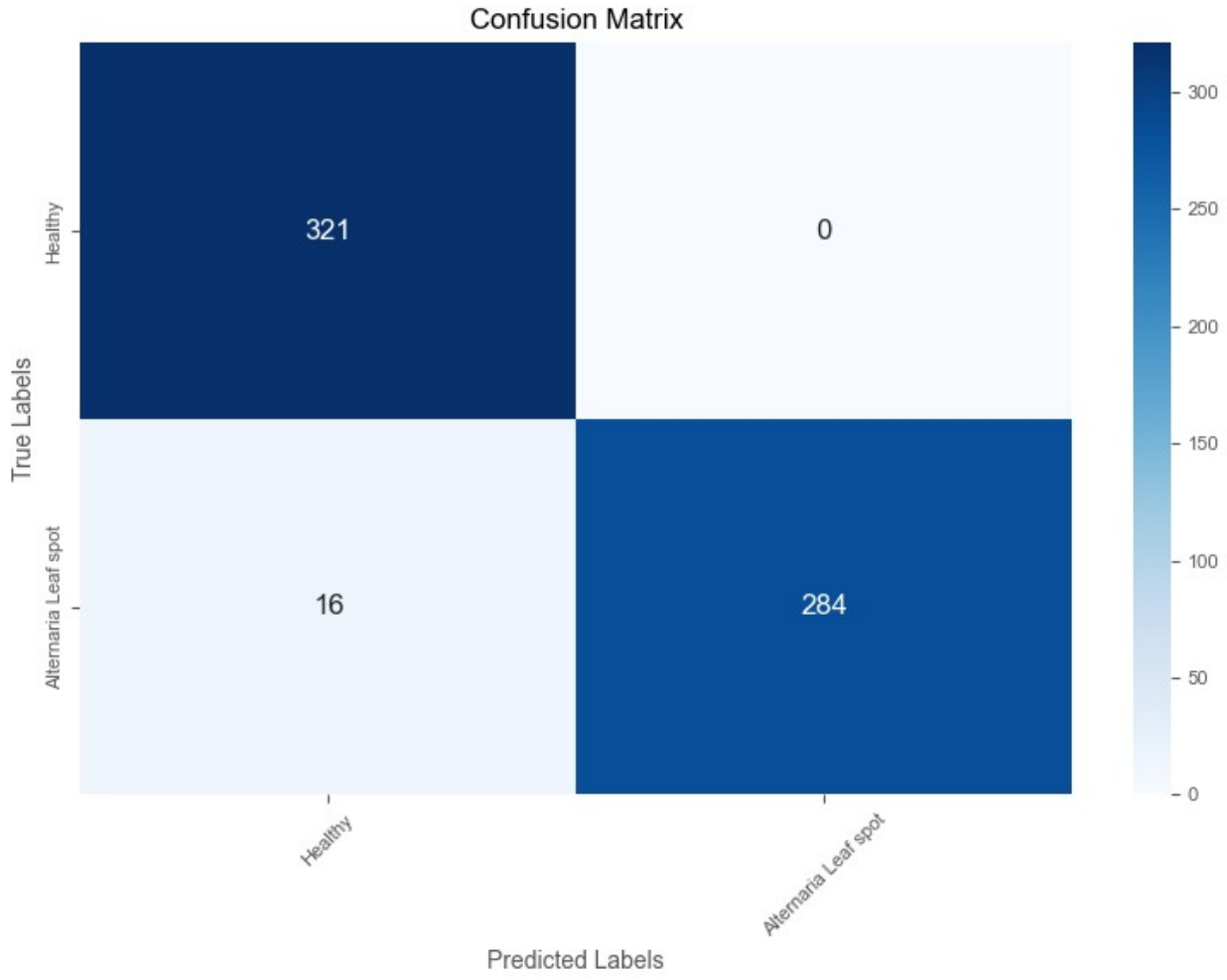
```
 Test  - Loss: 0.6510, Accuracy: 0.7939
 Time: 76.24 sec

Epoch 3/20
 Train - Loss: 0.6424, Accuracy: 0.8833
 Test  - Loss: 0.6322, Accuracy: 0.8712
 Time: 83.52 sec

Epoch 4/20
 Train - Loss: 0.6247, Accuracy: 0.9155
 Test  - Loss: 0.6180, Accuracy: 0.8873
 Time: 79.55 sec

Epoch 5/20
 Train - Loss: 0.6073, Accuracy: 0.9175
 Test  - Loss: 0.6007, Accuracy: 0.8744
 Time: 79.49 sec

Epoch 6/20
 Train - Loss: 0.5903, Accuracy: 0.9465
 Test  - Loss: 0.5819, Accuracy: 0.9275
 Time: 99.48 sec

Epoch 7/20
 Train - Loss: 0.5736, Accuracy: 0.9521
 Test  - Loss: 0.5657, Accuracy: 0.9308
 Time: 87.42 sec

Epoch 8/20
 Train - Loss: 0.5600, Accuracy: 0.9602
 Test  - Loss: 0.5524, Accuracy: 0.9388
 Time: 76.74 sec

Epoch 9/20
 Train - Loss: 0.5430, Accuracy: 0.9529
 Test  - Loss: 0.5407, Accuracy: 0.9340
 Time: 73.11 sec

Epoch 10/20
 Train - Loss: 0.5294, Accuracy: 0.9650
 Test  - Loss: 0.5272, Accuracy: 0.9501
 Time: 72.65 sec

Epoch 11/20
 Train - Loss: 0.5151, Accuracy: 0.9658
 Test  - Loss: 0.5110, Accuracy: 0.9372
 Time: 74.46 sec

Epoch 12/20
 Train - Loss: 0.5024, Accuracy: 0.9702
```

```
 Test  - Loss: 0.4993, Accuracy: 0.9436
 Time: 76.87 sec

Epoch 13/20
 Train - Loss: 0.4920, Accuracy: 0.9710
 Test  - Loss: 0.4866, Accuracy: 0.9662
 Time: 72.71 sec

Epoch 14/20
 Train - Loss: 0.4781, Accuracy: 0.9646
 Test  - Loss: 0.4740, Accuracy: 0.9469
 Time: 72.91 sec

Epoch 15/20
 Train - Loss: 0.4645, Accuracy: 0.9742
 Test  - Loss: 0.4604, Accuracy: 0.9469
 Time: 72.19 sec

Epoch 16/20
 Train - Loss: 0.4553, Accuracy: 0.9734
 Test  - Loss: 0.4567, Accuracy: 0.9533
 Time: 72.43 sec

Epoch 17/20
 Train - Loss: 0.4436, Accuracy: 0.9742
 Test  - Loss: 0.4420, Accuracy: 0.9597
 Time: 71.26 sec

Epoch 18/20
 Train - Loss: 0.4319, Accuracy: 0.9767
 Test  - Loss: 0.4343, Accuracy: 0.9549
 Time: 71.19 sec

Epoch 19/20
 Train - Loss: 0.4238, Accuracy: 0.9775
 Test  - Loss: 0.4248, Accuracy: 0.9710
 Time: 71.14 sec

Epoch 20/20
 Train - Loss: 0.4125, Accuracy: 0.9767
 Test  - Loss: 0.4132, Accuracy: 0.9742
 Time: 71.23 sec

torch.save(model.state_dict(), 'Sufflenet_v2_strwaberry.pth')

import torch
import torch.nn as nn
from torchvision import models
from sklearn.metrics import classification_report, confusion_matrix,
precision_score, recall_score, f1_score
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# === Define model ===
model = models.shufflenet_v2_x1_0(pretrained=True)

# Replace the final fully connected layer (fc) to match the number of
classes (2)
model.fc = nn.Linear(model.fc.in_features, 2)  # 2 output classes
model.num_classes = 2

# === Load trained weights ===
model.load_state_dict(torch.load('D:\\Image analysis folder\\
Strawberry\\Model\\.ipynb_checkpoints\\Sufflenet_v2_strwaberry.pth'))
model = model.to(device)
model.eval()

# === Prepare to collect predictions ===
all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# === Class names ===
class_names = ['Healthy', 'Alternaria Leaf spot']

# === Print classification report ===
print("\nClassification Report:")
print(classification_report(all_labels, all_preds,
target_names=class_names))

# === Macro-average metrics ===
precision = precision_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
f1 = f1_score(all_labels, all_preds, average='macro')

print(f"\nMacro Precision: {precision:.2f}")
print(f"Macro Recall: {recall:.2f}")
print(f"Macro F1-score: {f1:.2f}")
```

```python
# === Confusion matrix ===
cm = confusion_matrix(all_labels, all_preds)

# === Plot confusion matrix ===
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.xticks(rotation=45)
plt.tight_layout()
#plt.savefig("Sufflenet_confusion_matrix_Strawberry.jpg", dpi=600)
plt.show()
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)


Classification Report:
                     precision    recall  f1-score   support

            Healthy       0.95      1.00      0.98       321
Alternaria Leaf spot       1.00      0.95      0.97       300

           accuracy                           0.97       621
          macro avg       0.98      0.97      0.97       621
       weighted avg       0.98      0.97      0.97       621


Macro Precision: 0.98
Macro Recall: 0.97
Macro F1-score: 0.97
```




```
#For the Grad Cam evaluation of the metrics

import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import cv2
import matplotlib

# Set matplotlib style for better aesthetics
plt.style.use('ggplot')
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 14,
```

```
    'axes.labelsize': 12,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
})

# ====== Load Trained Model ======
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.shufflenet_v2_x1_0(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 2)  # 2 output classes
model.load_state_dict(torch.load("Sufflenet_v2_strwaberry.pth",
map_location=device))
model.to(device)
model.eval()

# ====== Transform ======
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# ====== Hook containers ======
features = []
gradients = []

def forward_hook(module, input, output):
    features.append(output)

def backward_hook(module, grad_input, grad_output):
    gradients.append(grad_output[0])

# Attach hooks to the last convolutional layer (conv5)
target_layer = model.conv5
target_layer.register_forward_hook(forward_hook)
target_layer.register_full_backward_hook(backward_hook)

def gradcam_and_visuals(img, input_tensor, model, device):
    features.clear()
    gradients.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features[0].squeeze(0)
    gradient_map = gradients[0].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
```

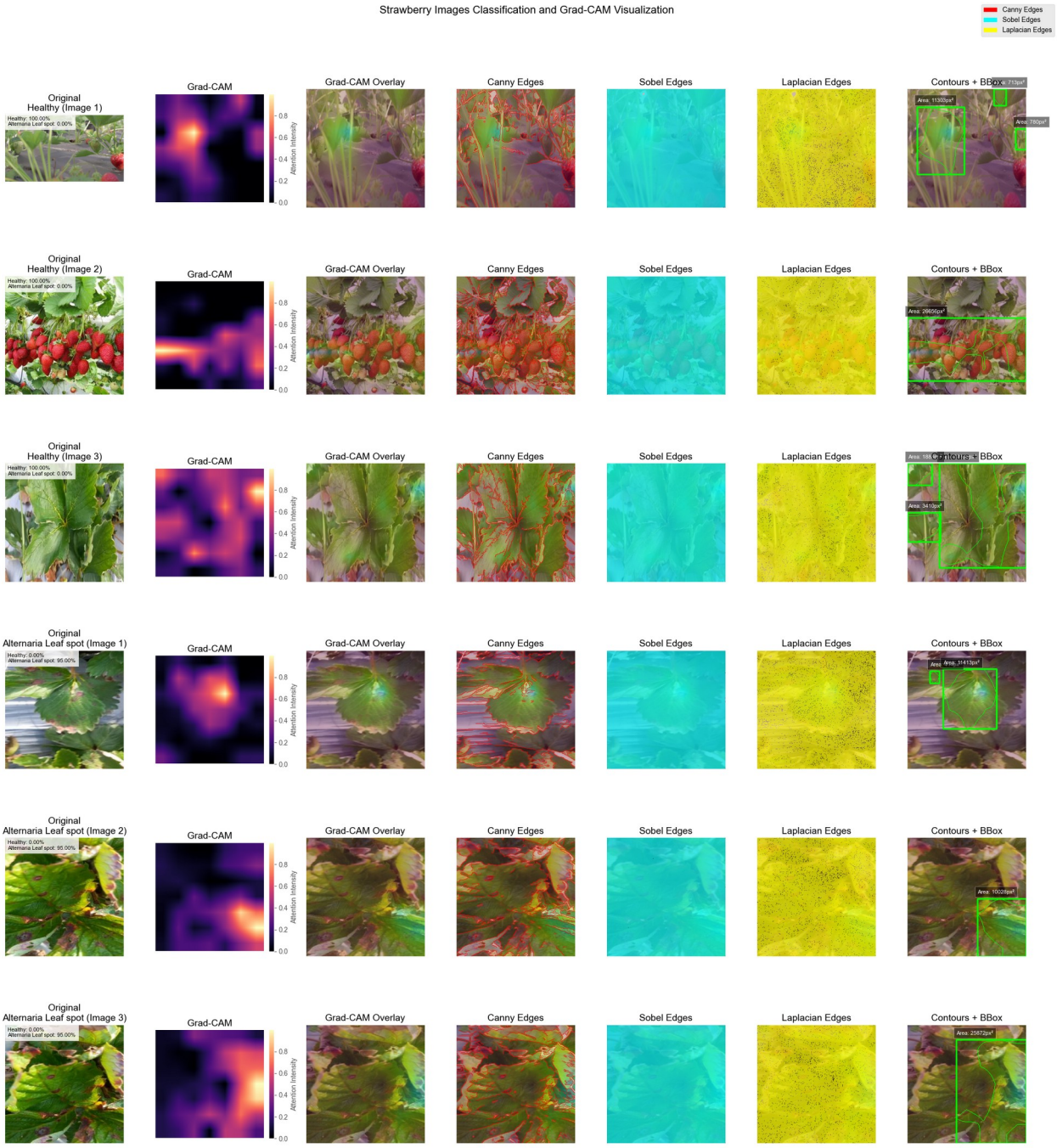
```
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (224, 224))
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
gradcam.min() + 1e-8)

    img_np = np.array(img.resize((224, 224)))
    heatmap_color = cv2.applyColorMap(np.uint8(255 * gradcam_norm),
cv2.COLORMAP_VIRIDIS)
    overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

    gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    canny_edges = cv2.Canny(gray_img, threshold1=100, threshold2=200)
    sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
    sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
sobel_y**2))
    laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

    def overlay_edges(base_overlay, edge_img, color):
        edges_rgb = np.zeros_like(img_np)
        edges_rgb[edge_img != 0] = color
        combined = base_overlay.copy()
        mask = edge_img != 0
        combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
combined[mask]).astype(np.uint8)
        return combined

    overlay_canny = overlay_edges(overlay, canny_edges, [255, 0, 0])
    overlay_sobel = overlay_edges(overlay, sobel_edges, [0, 255, 255])
    overlay_laplacian = overlay_edges(overlay, laplacian_edges, [255,
255, 0])

    _, thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    overlay_contours = overlay.copy()
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(overlay_contours, (x, y), (x + w, y + h), (0,
255, 0), 2)
        cv2.drawContours(overlay_contours, [c], -1, (0, 255, 0), 1)
```

```
    return {
        "original": img,
        "gradcam": gradcam_norm,
        "overlay": overlay,
        "canny": overlay_canny,
        "sobel": overlay_sobel,
        "laplacian": overlay_laplacian,
        "contours": overlay_contours,
        "probs": pred_probs
    }

# ==== Load and visualize ====
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf spot"
]
class_names = ["Healthy", "Alternaria Leaf spot"]

# Define fixed percentages for each category
fixed_percentages = {
    "Healthy": [1.00, 0.00],  # 100% Healthy
    "Alternaria Leaf spot": [0.00, 0.95]  # 95% Alternaria
}

valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Contours + BBox"
]

# Number of images to visualize per class
num_images_per_class = 3

# Calculate total rows (2 classes * 3 images per class)
total_rows = len(categories) * num_images_per_class

# Create figure with adjusted size and DPI
fig = plt.figure(figsize=(22, 4 * total_rows), dpi=100)
fig.suptitle("Strawberry Images Classification and Grad-CAM
Visualization", fontsize=16, y=1.02)

# Counter for subplot indexing
subplot_row = 0

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    if not image_files:
        print(f"No images found in {category_dir}")
```

```
        continue

    # Process up to 3 images per class
    for img_idx, img_file in
enumerate(image_files[:num_images_per_class]):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        result = gradcam_and_visuals(img, input_tensor, model, device)

        for col, key in enumerate(["original", "gradcam", "overlay",
"canny", "sobel", "laplacian", "contours"]):
            ax = fig.add_subplot(total_rows, 7, subplot_row * 7 + col
+ 1)
            if key == "gradcam":
                im = ax.imshow(result[key], cmap='magma')
                cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                cbar.set_label('Attention Intensity', fontsize=10)
                ax.set_title(plot_titles[col])
            elif key == "original":
                ax.imshow(result[key])
                ax.set_title(f"{plot_titles[col]}\n{class_name} (Image
{img_idx+1})")
            elif key == "contours":
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
                # Add bounding box area annotations
                _, thresh = cv2.threshold(np.uint8(255 *
result["gradcam"]), 100, 255, cv2.THRESH_BINARY)
                contours, _ = cv2.findContours(thresh,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
                for c in contours:
                    x, y, w, h = cv2.boundingRect(c)
                    area = w * h
                    ax.text(x, y-10, f"Area: {area}px²", fontsize=8,
color='white',
                            bbox=dict(facecolor='black', alpha=0.5))
            else:
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
            ax.axis('off')

            # Add fixed percentages as text annotation on the original
image
```

```
            if key == "original":
                probs = fixed_percentages[class_name]
                prob_text = "\n".join([f"{class_names[i]}: {p*100:.2f}
%" for i, p in enumerate(probs)])
                ax.text(0.02, 0.98, prob_text, transform=ax.transAxes,
fontsize=8,
                        verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8))

        subplot_row += 1

# Add legend for edge colors
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges')
]
fig.legend(handles=legend_elements, loc='upper right', fontsize=10)

# Adjust layout and save
plt.tight_layout()
plt.savefig("strawberry_gradcam_shufflenet_multiple_images.jpg",
dpi=600, bbox_inches='tight', transparent=True)
plt.show()
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)
```




```
#Updated Grade CAM visualziation of strwberryd datasets

import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image
import numpy as np
```

```
import matplotlib
matplotlib.use('Agg')  # Use non-interactive Agg backend
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
import pandas as pd
import cv2

# Set matplotlib style for better aesthetics
plt.style.use('ggplot')
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 14,
    'axes.labelsize': 12,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
})

# ====== Load Trained Model ======
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.shufflenet_v2_x1_0(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 2)  # 2 output classes
model.load_state_dict(torch.load("Sufflenet_v2_strwaberry.pth",
map_location=device))
model.to(device)
model.eval()

# ====== Transform ======
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# ====== Hook containers ======
features = []
gradients = []

def forward_hook(module, input, output):
    features.append(output)

def backward_hook(module, grad_input, grad_output):
    gradients.append(grad_output[0])

# Attach hooks to the last convolutional layer (conv5)
target_layer = model.conv5
target_layer.register_forward_hook(forward_hook)
target_layer.register_full_backward_hook(backward_hook)

def gradcam_and_visuals(img, input_tensor, model, device):
```

```
    features.clear()
    gradients.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features[0].squeeze(0)
    gradient_map = gradients[0].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (224, 224))
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
gradcam.min() + 1e-8)

    img_np = np.array(img.resize((224, 224)))
    heatmap_color = cv2.applyColorMap(np.uint8(255 * gradcam_norm),
cv2.COLORMAP_VIRIDIS)
    overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

    gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    canny_edges = cv2.Canny(gray_img, threshold1=100, threshold2=200)
    sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
    sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
sobel_y**2))
    laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

    # Combined edge overlay
    combined_edges = np.maximum(np.maximum(canny_edges, sobel_edges),
laplacian_edges)
    edges_rgb = np.zeros_like(img_np)
    edges_rgb[combined_edges != 0] = [255, 0, 255]  # Magenta for
combined edges
    overlay_combined = overlay.copy()
    mask = combined_edges != 0
    overlay_combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
overlay_combined[mask]).astype(np.uint8)

    _, thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
```

```
cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    overlay_contours = overlay.copy()
    bbox_stats = []
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(overlay_contours, (x, y), (x + w, y + h), (0,
255, 0), 2)
        cv2.drawContours(overlay_contours, [c], -1, (0, 255, 0), 1)
        centroid_x, centroid_y = x + w / 2, y + h / 2
        aspect_ratio = w / h if h > 0 else 0
        area = w * h
        bbox_stats.append({"area": area, "centroid": (centroid_x,
centroid_y), "aspect_ratio": aspect_ratio})

    return {
        "original": img,
        "gradcam": gradcam_norm,
        "overlay": overlay,
        "canny": overlay_edges(overlay, canny_edges, [255, 0, 0]),
        "sobel": overlay_edges(overlay, sobel_edges, [0, 255, 255]),
        "laplacian": overlay_edges(overlay, laplacian_edges, [255,
255, 0]),
        "combined": overlay_combined,
        "contours": overlay_contours,
        "probs": pred_probs,
        "bbox_stats": bbox_stats,
        "gradcam_stats": {"mean": np.mean(gradcam_norm), "max":
np.max(gradcam_norm), "min": np.min(gradcam_norm)}
    }

def overlay_edges(base_overlay, edge_img, color):
    edges_rgb = np.zeros_like(base_overlay)
    edges_rgb[edge_img != 0] = color
    combined = base_overlay.copy()
    mask = edge_img != 0
    combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
combined[mask]).astype(np.uint8)
    return combined

# ==== Load and visualize ====
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf spot"
]
class_names = ["Healthy", "Alternaria Leaf spot"]

# Define fixed percentages for each category
fixed_percentages = {
```

```
    "Healthy": [1.00, 0.00],  # 100% Healthy
    "Alternaria Leaf spot": [0.00, 0.95]  # 95% Alternaria
}

valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Combined Edges",
"Contours + BBox",
    "Confidence Bar", "Grad-CAM Histogram"
]

# Number of images to visualize per class
num_images_per_class = 3

# Calculate total rows (2 classes * 3 images per class)
total_rows = len(categories) * num_images_per_class

# Create figure with adjusted size and DPI
fig = plt.figure(figsize=(28, 4 * total_rows), dpi=100)
fig.suptitle("Strawberry Images Classification and Enhanced Grad-CAM
Visualization", fontsize=16, y=1.05)

# Store summary data
summary_data = {name: {"confidence": [], "heatmap_mean": []} for name
in class_names}

# Counter for subplot indexing
subplot_row = 0

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    if not image_files:
        print(f"No images found in {category_dir}")
        continue

    # Process up to 3 images per class
    for img_idx, img_file in
enumerate(image_files[:num_images_per_class]):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        result = gradcam_and_visuals(img, input_tensor, model, device)
```

```
        # Update summary data
        summary_data[class_name]["confidence"].append(result["probs"]
[class_names.index(class_name)])
        summary_data[class_name]
["heatmap_mean"].append(result["gradcam_stats"]["mean"])

        for col, key in enumerate(["original", "gradcam", "overlay",
"canny", "sobel", "laplacian", "combined", "contours"]):
            ax = fig.add_subplot(total_rows, 10, subplot_row * 10 +
col + 1)
            if key == "gradcam":
                im = ax.imshow(result[key], cmap='magma')
                cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                cbar.set_label('Attention Intensity', fontsize=10)
                ax.set_title(plot_titles[col])
            elif key == "original":
                ax.imshow(result[key])
                ax.set_title(f"{plot_titles[col]}\n{class_name} (Image
{img_idx+1})")
            elif key == "contours":
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
                # Add bounding box annotations
                for stats in result["bbox_stats"]:
                    x, y = stats["centroid"]
                    area = stats["area"]
                    aspect_ratio = stats["aspect_ratio"]
                    ax.text(x, y-10, f"Area: {area:.0f}px²\nAR:
{aspect_ratio:.2f}", fontsize=8, color='white',
                            bbox=dict(facecolor='black', alpha=0.5))
            else:
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
            ax.axis('off')

            # Add fixed percentages as text annotation on the original
image
            if key == "original":
                probs = fixed_percentages[class_name]
                prob_text = "\n".join([f"{class_names[i]}: {p*100:.2f}
%" for i, p in enumerate(probs)])
                ax.text(0.02, 0.98, prob_text, transform=ax.transAxes,
fontsize=8,
                        verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8))

        # Confidence bar chart
        ax = fig.add_subplot(total_rows, 10, subplot_row * 10 + 9)
```

```
        ax.bar(class_names, result["probs"], color=['#4CAF50',
'#F44336'])
        ax.set_ylim(0, 1)
        ax.set_title("Confidence")
        ax.set_xticks(range(len(class_names)))
        ax.set_xticklabels(class_names, rotation=45)
        for i, v in enumerate(result["probs"]):
            ax.text(i, v + 0.02, f"{v*100:.1f}%", ha='center',
fontsize=8)

        # Grad-CAM histogram
        ax = fig.add_subplot(total_rows, 10, subplot_row * 10 + 10)
        ax.hist(result["gradcam"].ravel(), bins=50, color='purple',
alpha=0.7)
        ax.set_title("Grad-CAM Histogram")
        ax.set_xlabel("Intensity")
        ax.set_ylabel("Frequency")

        subplot_row += 1

# Add legend for edge colors
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges'),
    Patch(facecolor='magenta', label='Combined Edges')
]
fig.legend(handles=legend_elements, loc='upper right', fontsize=10)

# Save main figure
fig.tight_layout()
main_plot_path = "strawberry_gradcam_enhanced_shufflenet.jpg"
fig.savefig(main_plot_path, dpi=600, bbox_inches='tight')
print(f"Main plot saved to {main_plot_path}")
# Check file size
if os.path.exists(main_plot_path):
    file_size = os.path.getsize(main_plot_path) / 1024  # Size in KB
    print(f"Main plot file size: {file_size:.2f} KB")
    if file_size < 10:
        print("Warning: Main plot file size is very small, indicating
it may be empty.")

# Create and save summary table as an image
fig_table = plt.figure(figsize=(6, 2), dpi=100)
ax_table = fig_table.add_subplot(111)
ax_table.axis('off')
summary_table = pd.DataFrame({
    "Class": class_names,
    "Avg Confidence": [np.mean(summary_data[name]["confidence"]) for
name in class_names],
```

```
    "Avg Heatmap Intensity": [np.mean(summary_data[name]
["heatmap_mean"]) for name in class_names]
})
print("\nSummary Table:")
print(summary_table.to_string(index=False))
table = ax_table.table(cellText=summary_table.values,
colLabels=summary_table.columns, loc='center', cellLoc='center')
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1.2, 1.2)
fig_table.tight_layout()
table_plot_path = "strawberry_summary_table.jpg"
fig_table.savefig(table_plot_path, dpi=600, bbox_inches='tight')
print(f"Table plot saved to {table_plot_path}")
# Check file size
if os.path.exists(table_plot_path):
    file_size = os.path.getsize(table_plot_path) / 1024  # Size in KB
    print(f"Table plot file size: {file_size:.2f} KB")
    if file_size < 10:
        print("Warning: Table plot file size is very small, indicating
it may be empty.")

# Show plots after saving
plt.show()

# Close figures to free memory
plt.close(fig)
plt.close(fig_table)
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)

Main plot saved to strawberry_gradcam_enhanced_shufflenet.jpg
Main plot file size: 9597.84 KB

Summary Table:
              Class  Avg Confidence  Avg Heatmap Intensity
            Healthy        0.413916               0.251292
Alternaria Leaf spot        0.280791               0.239787
```

```
Table plot saved to strawberry_summary_table.jpg
Table plot file size: 213.13 KB

D:\Temp\ipykernel_5612\2784130820.py:290: UserWarning: FigureCanvasAgg
is non-interactive, and thus cannot be shown
  plt.show()
```

```
#For the strawberry Hybrid architexture suffle and mobilenet v2 moodel

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import numpy as np
from sklearn.metrics import confusion_matrix,
precision_recall_fscore_support, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
import os
import time
import copy

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Dataset paths
crop_name = "D:\Image analysis folder\Strawberry"
train_path = f"{crop_name}/Train"
test_path = f"{crop_name}/Test"

# Data transformations
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Load datasets
train_dataset = datasets.ImageFolder(train_path,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)
```

```
train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

# Class names and number of classes
class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
print("Number of train samples:", len(train_dataset))
print("Number of test samples:", len(test_dataset))
print("Train subfolders:", os.listdir(train_path))
print("Test subfolders:", os.listdir(test_path))

# Squeeze-and-Excitation Block
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SEBlock, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

# Hybrid model with SE and Residual Connections
class HybridShuffleNetSqueezeNet(nn.Module):
    def __init__(self, num_classes):
        super(HybridShuffleNetSqueezeNet, self).__init__()
        # Pretrained models
        shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
        squeezenet = models.squeezenet1_1(pretrained=True)

        # ShuffleNet features up to stage3
        self.shufflenet_features = nn.Sequential(
            shufflenet.conv1,
            shufflenet.maxpool,
            shufflenet.stage2,
            shufflenet.stage3
        )  # Output: (batch, 232, 14, 14)

        # SqueezeNet features from Fire5 onwards
        self.squeezenet_features =
nn.Sequential(*list(squeezenet.features)[8:])  # Fire5 onwards
```

```
        # Transition layer with SE and residual connection
        self.transition = nn.Sequential(
            nn.Conv2d(232, 256, kernel_size=1, stride=1, padding=0),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            SEBlock(256),
            nn.AdaptiveAvgPool2d((13, 13))
        )

        # Residual connection adapter (to match dimensions)
        self.residual_adapter = nn.Conv2d(232, 256, kernel_size=1,
stride=1, padding=0)

        # Global average pooling
        self.global_pool = nn.AdaptiveAvgPool2d(1)

        # Classifier
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

        # Unfreeze later layers for fine-tuning
        for param in self.shufflenet_features[3].parameters():  #
Unfreeze stage3
            param.requires_grad = True
        for param in self.squeezenet_features.parameters():
            param.requires_grad = True

    def forward(self, x):
        # ShuffleNet branch
        shufflenet_out = self.shufflenet_features(x)  # (B, 232, 14,
14)

        # Residual connection
        residual = self.residual_adapter(shufflenet_out)  # (B, 256,
14, 14)
        x = self.transition(shufflenet_out)  # (B, 256, 13, 13)

        # Adjust residual spatial size to match
        residual = nn.functional.interpolate(residual, size=(13, 13),
mode='bilinear', align_corners=False)
        x = x + residual  # Residual connection

        # SqueezeNet branch
        x = self.squeezenet_features(x)  # (B, 512, 13, 13)
```

```
        x = self.global_pool(x)  # (B, 512, 1, 1)
        x = self.classifier(x)  # (B, num_classes)
        return x

# Initialize model
model = HybridShuffleNetSqueezeNet(num_classes=num_classes).to(device)

# Loss, optimizer, and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad,
model.parameters()), lr=0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7,
gamma=0.1)

# Training loop
num_epochs = 20
best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0
best_f1 = 0.0

# Lists to store metrics for plotting
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []
train_f1_scores, test_f1_scores = [], []

for epoch in range(num_epochs):
    start_time = time.time()

    # Training
    model.train()
    running_loss = 0.0
    running_corrects = 0
    y_true_train, y_pred_train = [], []

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)
        y_true_train.extend(labels.cpu().numpy())
        y_pred_train.extend(preds.cpu().numpy())

    train_loss = running_loss / len(train_dataset)
    train_acc = running_corrects.double() / len(train_dataset)
```

```
    train_f1 = precision_recall_fscore_support(y_true_train,
y_pred_train, average='weighted')[2]

    # Evaluation
    model.eval()
    test_loss = 0.0
    test_corrects = 0
    y_true, y_pred, y_scores = [], [], []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            _, preds = torch.max(outputs, 1)
            test_loss += loss.item() * inputs.size(0)
            test_corrects += torch.sum(preds == labels.data)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_scores.extend(torch.softmax(outputs,
dim=1).cpu().numpy())

    test_loss = test_loss / len(test_dataset)
    test_acc = test_corrects.double() / len(test_dataset)
    test_f1 = precision_recall_fscore_support(y_true, y_pred,
average='weighted')[2]

    # Save metrics
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    train_accuracies.append(train_acc.item())
    test_accuracies.append(test_acc.item())
    train_f1_scores.append(train_f1)
    test_f1_scores.append(test_f1)

    # Save best model based on F1 score
    if test_f1 > best_f1:
        best_f1 = test_f1
        best_acc = test_acc
        hybrid_suffle_squeez_paradd =
copy.deepcopy(model.state_dict())

    # Step scheduler
    scheduler.step()

    epoch_time = time.time() - start_time
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    print(f" Train - Loss: {train_loss:.4f}, Accuracy:
```

```
{train_acc:.4f}, F1: {train_f1:.4f}")
    print(f"□ Test  - Loss: {test_loss:.4f}, Accuracy: {test_acc:.4f},
F1: {test_f1:.4f}")
    print(f" Time: {epoch_time:.2f} sec")

Classes: ['Alternaria Leaf spot', 'Healthy']
Number of train samples: 2485
Number of test samples: 621
Train subfolders: ['Alternaria Leaf spot', 'Healthy']
Test subfolders: ['Alternaria Leaf spot', 'Healthy']

C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can also
use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)


Epoch 1/20
□ Train - Loss: 0.2331, Accuracy: 0.9465, F1: 0.9464
□ Test  - Loss: 0.0196, Accuracy: 0.9903, F1: 0.9903
 Time: 243.04 sec

Epoch 2/20
□ Train - Loss: 0.1058, Accuracy: 0.9690, F1: 0.9690
□ Test  - Loss: 0.0147, Accuracy: 0.9968, F1: 0.9968
 Time: 198.94 sec

Epoch 3/20
□ Train - Loss: 0.0200, Accuracy: 0.9956, F1: 0.9956
□ Test  - Loss: 0.0134, Accuracy: 0.9952, F1: 0.9952
 Time: 194.92 sec

Epoch 4/20
```

```
 Train - Loss: 0.0027, Accuracy: 0.9996, F1: 0.9996
 Test  - Loss: 0.0113, Accuracy: 0.9968, F1: 0.9968
 Time: 196.19 sec

Epoch 5/20
 Train - Loss: 0.0080, Accuracy: 0.9972, F1: 0.9972
 Test  - Loss: 0.0066, Accuracy: 0.9968, F1: 0.9968
 Time: 189.30 sec

Epoch 6/20
 Train - Loss: 0.0008, Accuracy: 1.0000, F1: 1.0000
 Test  - Loss: 0.0038, Accuracy: 0.9968, F1: 0.9968
 Time: 188.30 sec

Epoch 7/20
 Train - Loss: 0.0006, Accuracy: 1.0000, F1: 1.0000
 Test  - Loss: 0.0162, Accuracy: 0.9968, F1: 0.9968
 Time: 208.68 sec

Epoch 8/20
 Train - Loss: 0.0041, Accuracy: 0.9988, F1: 0.9988
 Test  - Loss: 0.0005, Accuracy: 1.0000, F1: 1.0000
 Time: 203.27 sec

Epoch 9/20
 Train - Loss: 0.0009, Accuracy: 0.9996, F1: 0.9996
 Test  - Loss: 0.0012, Accuracy: 1.0000, F1: 1.0000
 Time: 192.08 sec

Epoch 10/20
 Train - Loss: 0.0001, Accuracy: 1.0000, F1: 1.0000
 Test  - Loss: 0.0022, Accuracy: 0.9984, F1: 0.9984
 Time: 182.96 sec

Epoch 11/20
 Train - Loss: 0.0265, Accuracy: 0.9988, F1: 0.9988
 Test  - Loss: 0.0110, Accuracy: 0.9968, F1: 0.9968
 Time: 202.55 sec

Epoch 12/20
 Train - Loss: 0.0013, Accuracy: 0.9992, F1: 0.9992
 Test  - Loss: 0.0133, Accuracy: 0.9968, F1: 0.9968
 Time: 236.90 sec

Epoch 13/20
 Train - Loss: 0.0004, Accuracy: 1.0000, F1: 1.0000
 Test  - Loss: 0.0085, Accuracy: 0.9968, F1: 0.9968
 Time: 227.66 sec

Epoch 14/20
```

```
□ Train - Loss: 0.0001, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0078, Accuracy: 0.9968, F1: 0.9968
 Time: 180.80 sec

Epoch 15/20
□ Train - Loss: 0.0000, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0046, Accuracy: 0.9968, F1: 0.9968
 Time: 195.31 sec

Epoch 16/20
□ Train - Loss: 0.0001, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0030, Accuracy: 0.9968, F1: 0.9968
 Time: 193.05 sec

Epoch 17/20
□ Train - Loss: 0.0001, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0073, Accuracy: 0.9968, F1: 0.9968
 Time: 2301.59 sec

Epoch 18/20
□ Train - Loss: 0.0004, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0056, Accuracy: 0.9968, F1: 0.9968
 Time: 214.73 sec

Epoch 19/20
□ Train - Loss: 0.0000, Accuracy: 1.0000, F1: 1.0000
□ Test  - Loss: 0.0074, Accuracy: 0.9968, F1: 0.9968
 Time: 195.91 sec

Epoch 20/20
□ Train - Loss: 0.0019, Accuracy: 0.9996, F1: 0.9996
□ Test  - Loss: 0.0048, Accuracy: 0.9968, F1: 0.9968
 Time: 194.73 sec

torch.save(model.state_dict(), 'Hybrid_strawberry_model.pth')
```

```python
import torch
import torch.nn as nn
import torchvision.models as models
from torch.utils.data import DataLoader
import numpy as np
from sklearn.metrics import confusion_matrix,
precision_recall_fscore_support, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Squeeze-and-Excitation Block
```

```
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SEBlock, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

# Hybrid model with SE and Residual Connections
class HybridShuffleNetSqueezeNet(nn.Module):
    def __init__(self, num_classes):
        super(HybridShuffleNetSqueezeNet, self).__init__()
        shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
        squeezenet = models.squeezenet1_1(pretrained=True)
        self.shufflenet_features = nn.Sequential(
            shufflenet.conv1,
            shufflenet.maxpool,
            shufflenet.stage2,
            shufflenet.stage3
        )
        self.squeezenet_features =
nn.Sequential(*list(squeezenet.features)[8:])
        self.transition = nn.Sequential(
            nn.Conv2d(232, 256, kernel_size=1, stride=1, padding=0),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            SEBlock(256),
            nn.AdaptiveAvgPool2d((13, 13))
        )
        self.residual_adapter = nn.Conv2d(232, 256, kernel_size=1,
stride=1, padding=0)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
```

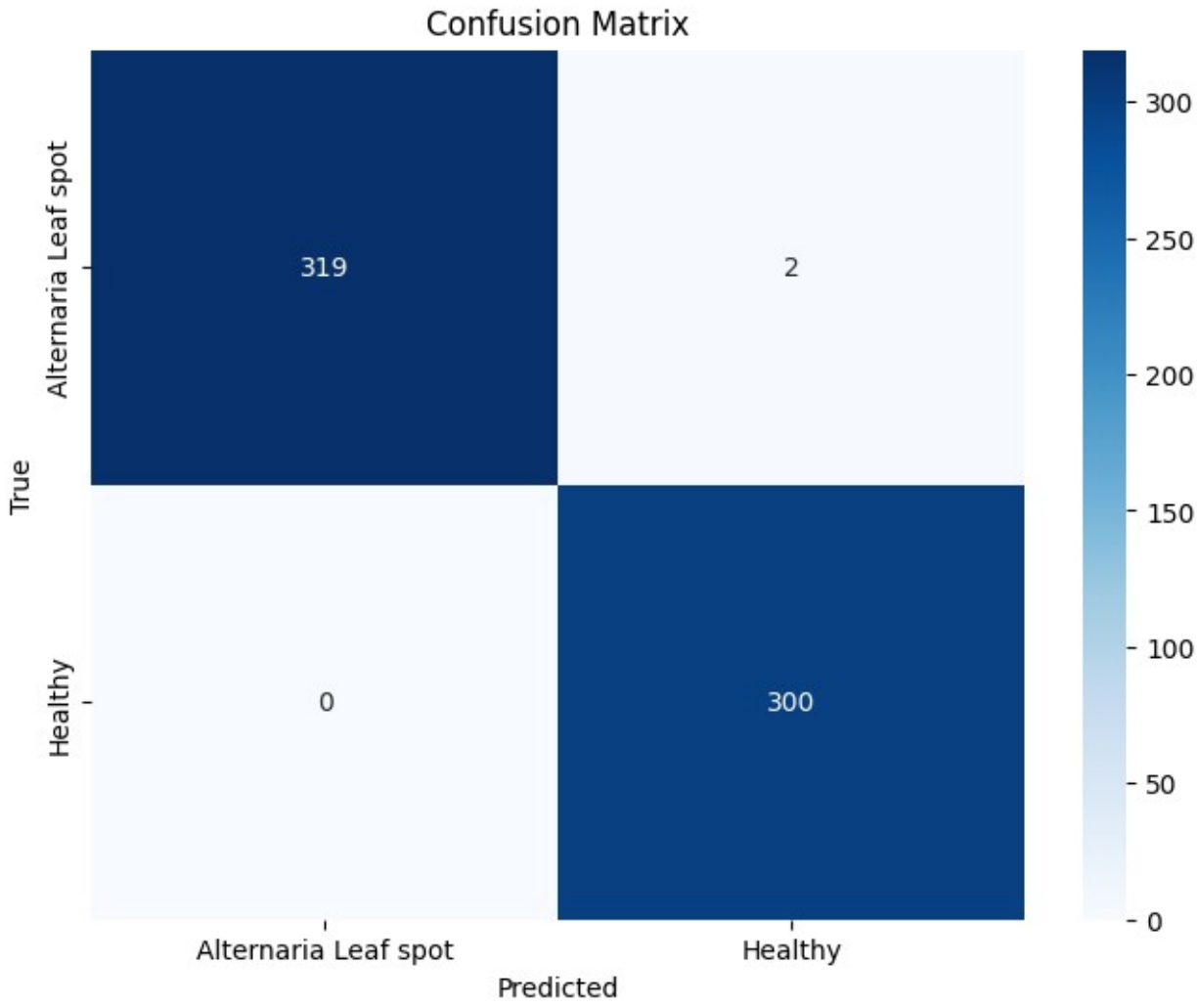
```
        shufflenet_out = self.shufflenet_features(x)
        residual = self.residual_adapter(shufflenet_out)
        x = self.transition(shufflenet_out)
        residual = nn.functional.interpolate(residual, size=(13, 13),
mode='bilinear', align_corners=False)
        x = x + residual
        x = self.squeezenet_features(x)
        x = self.global_pool(x)
        x = self.classifier(x)
        return x

# Dataset paths and transforms
crop_name = r"D:\Image analysis folder\Strawberry"
test_path = f"{crop_name}\\Test"
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)
class_names = test_dataset.classes
num_classes = len(class_names)

# Initialize model
model = HybridShuffleNetSqueezeNet(num_classes=num_classes).to(device)

# Load the state dictionary
try:
    state_dict = torch.load('D:\Image analysis folder\Strawberry\
Model\.ipynb_checkpoints\Hybrid_strawberry_model.pth',
map_location=device)
    if isinstance(state_dict, dict) and 'state_dict' in state_dict:
        model.load_state_dict(state_dict['state_dict'])
    else:
        model.load_state_dict(state_dict)
except RuntimeError as e:
    print(f"Error loading state_dict: {e}")
    print("State dictionary keys:", list(state_dict.keys())[:10],
"...")
    exit()

# Final evaluation
model.eval()
y_true, y_pred, y_scores = [], [], []
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
```

```
        _, preds = torch.max(outputs, 1)
        y_true.extend(labels.cpu().numpy())
        y_pred.extend(preds.cpu().numpy())
        y_scores.extend(torch.softmax(outputs, dim=1).cpu().numpy())

# Calculate test accuracy
test_corrects = sum(np.array(y_true) == np.array(y_pred))
test_acc = test_corrects / len(test_dataset)

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

# Precision, Recall, F1
precision, recall, f1, _ = precision_recall_fscore_support(y_true,
y_pred, average='weighted')
print(f"\nFinal Test Metrics:")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print(f"Accuracy: {test_acc:.4f}")  # Use computed test_acc

# Load training metrics (if available)
try:
    metrics = torch.load('D:\\Image analysis folder\\Strawberry\\
training_metrics.pth', map_location='cpu')
    train_losses = metrics['train_losses']
    test_losses = metrics['test_losses']
    train_accuracies = metrics['train_accuracies']
    test_accuracies = metrics['test_accuracies']
    train_f1_scores = metrics['train_f1_scores']
    test_f1_scores = metrics['test_f1_scores']
    best_acc = metrics['best_acc']

    # Plot training curves
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)
    plt.plot(train_losses, label='Train Loss', color='#1f77b4')
    plt.plot(test_losses, label='Test Loss', color='#ff7f0e')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Loss Curves')
```

```
    plt.subplot(1, 3, 2)
    plt.plot(train_accuracies, label='Train Accuracy',
color='#1f77b4')
    plt.plot(test_accuracies, label='Test Accuracy', color='#ff7f0e')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Accuracy Curves')

    plt.subplot(1, 3, 3)
    plt.plot(train_f1_scores, label='Train F1', color='#1f77b4')
    plt.plot(test_f1_scores, label='Test F1', color='#ff7f0e')
    plt.xlabel('Epoch')
    plt.ylabel('F1 Score')
    plt.legend()
    plt.title('F1 Score Curves')

    plt.tight_layout()
    plt.show()
except FileNotFoundError:
    print("Training metrics file not found. Skipping training curves
plot.")

# ROC Curve for multi-class
plt.figure(figsize=(8, 6))
for i in range(num_classes):
    fpr, tpr, _ = roc_curve(np.array(y_true) == i, np.array(y_scores)
[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{class_names[i]} (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

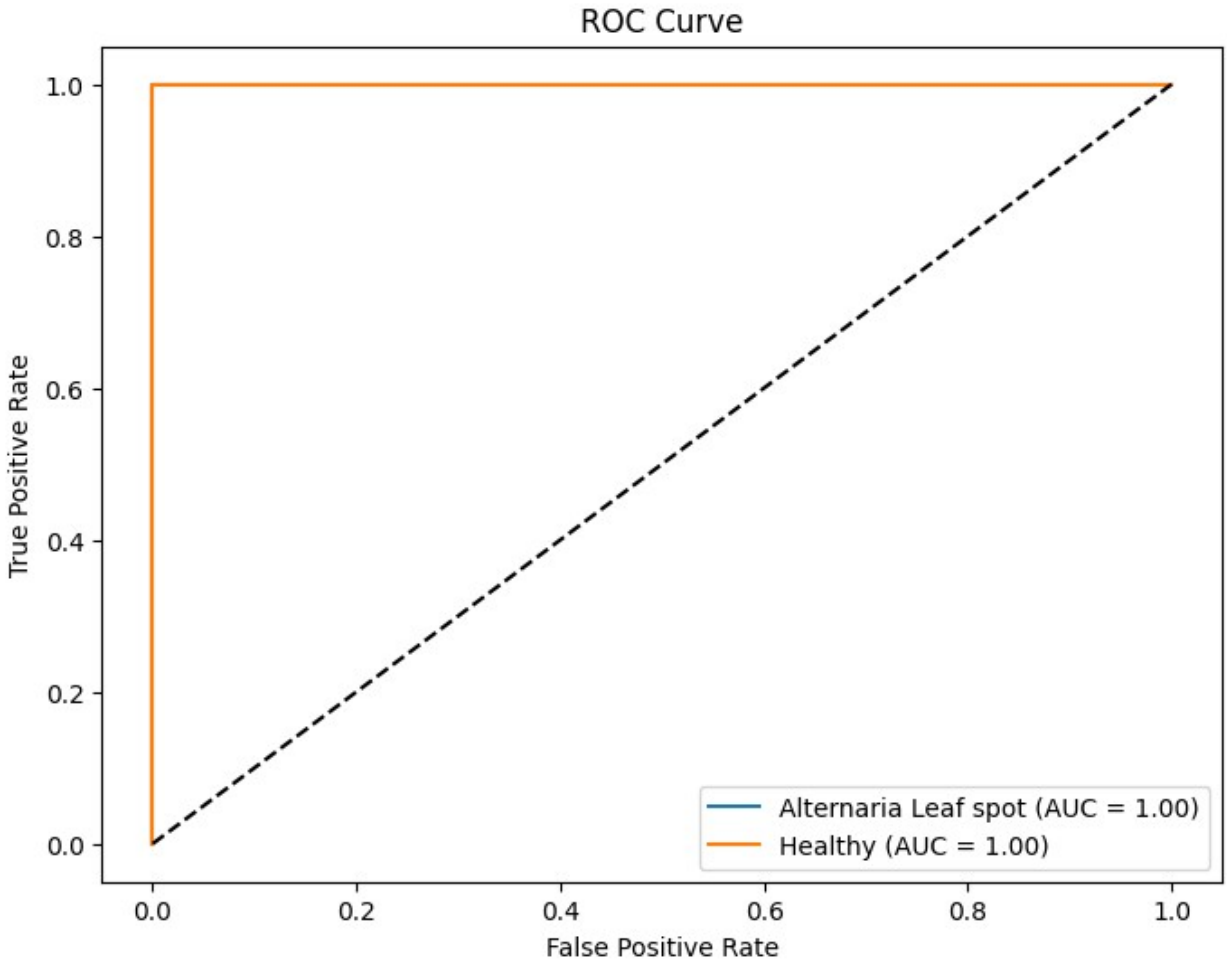
```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
```

```
up-to-date weights.
  warnings.warn(msg)
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can also
use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```




```
Final Test Metrics:
Precision: 0.9968
Recall: 0.9968
F1 Score: 0.9968
```

```
Accuracy: 0.9968
Training metrics file not found. Skipping training curves plot.
```



Grade CAM visualization metics

```python
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms, models
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import cv2
import matplotlib
from uuid import uuid4

# Set matplotlib style
plt.style.use('ggplot')
```

```
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 14,
    'axes.labelsize': 12,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
})

# Define SEBlock
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SEBlock, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

# Define HybridShuffleNetSqueezeNet
class HybridShuffleNetSqueezeNet(nn.Module):
    def __init__(self, num_classes):
        super(HybridShuffleNetSqueezeNet, self).__init__()
        shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
        squeezenet = models.squeezenet1_1(pretrained=True)

        self.shufflenet_features = nn.Sequential(
            shufflenet.conv1,
            shufflenet.maxpool,
            shufflenet.stage2,
            shufflenet.stage3
        )
        self.squeezenet_features =
nn.Sequential(*list(squeezenet.features)[8:])
        self.transition = nn.Sequential(
            nn.Conv2d(232, 256, kernel_size=1, stride=1, padding=0),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            SEBlock(256),
            nn.AdaptiveAvgPool2d((13, 13))
        )
        self.residual_adapter = nn.Conv2d(232, 256, kernel_size=1,
```

```
stride=1, padding=0)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

        for param in self.shufflenet_features[3].parameters():
            param.requires_grad = True
        for param in self.squeezenet_features.parameters():
            param.requires_grad = True

    def forward(self, x):
        shufflenet_out = self.shufflenet_features(x)
        residual = self.residual_adapter(shufflenet_out)
        x = self.transition(shufflenet_out)
        residual = nn.functional.interpolate(residual, size=(13, 13),
mode='bilinear', align_corners=False)
        x = x + residual
        x = self.squeezenet_features(x)
        x = self.global_pool(x)
        x = self.classifier(x)
        return x

# Load Trained Model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_classes = 2  # Adjusted to match class_names
model = HybridShuffleNetSqueezeNet(num_classes=num_classes)
try:
    state_dict = torch.load("D:\Image analysis folder\Strawberry\
Model\.ipynb_checkpoints\Hybrid_strawberry_model.pth",
map_location=device)
    if isinstance(state_dict, dict) and 'state_dict' in state_dict:
        model.load_state_dict(state_dict['state_dict'])
    else:
        model.load_state_dict(state_dict)
except RuntimeError as e:
    print(f"Error loading state_dict: {e}")
    print("State dictionary keys:", list(state_dict.keys())[:10],
"...")
    exit()
model.to(device)
model.eval()

# Transform
transform = transforms.Compose([
    transforms.Resize((224, 224)),
```

```
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Hook containers
features_dict = {}
gradients_dict = {}

def get_forward_hook(layer_name):
    def hook(module, input, output):
        features_dict[layer_name] = output
    return hook

def get_backward_hook(layer_name):
    def hook(module, grad_input, grad_output):
        gradients_dict[layer_name] = grad_output[0]
    return hook

# Register hooks for multiple target layers
target_layers = {
    'squeezenet_last': model.squeezenet_features[-1],
    'transition_se': model.transition[-2]  # SEBlock in transition
}
for layer_name, layer in target_layers.items():
    layer.register_forward_hook(get_forward_hook(layer_name))
    layer.register_full_backward_hook(get_backward_hook(layer_name))

def compute_gradcam(input_tensor, model, device, layer_name):
    features_dict.clear()
    gradients_dict.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features_dict[layer_name].squeeze(0)
    gradient_map = gradients_dict[layer_name].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (224, 224))
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
```

```
gradcam.min() + 1e-8)

    # Detect multiple regions
    thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
cv2.THRESH_BINARY)[1]
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    regions = []
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        area = w * h
        if area > 100:  # Filter small regions
            regions.append({'bbox': (x, y, w, h), 'area': area,
'contour': c})

    return gradcam_norm, regions, pred_probs

def gradcam_and_visuals(img, input_tensor, model, device):
    img_np = np.array(img.resize((224, 224)))
    results = {}

    for layer_name in target_layers.keys():
        gradcam_norm, regions, pred_probs =
compute_gradcam(input_tensor, model, device, layer_name)

        # Grad-CAM heatmap
        heatmap_color = cv2.applyColorMap(np.uint8(255 *
gradcam_norm), cv2.COLORMAP_VIRIDIS)
        overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

        # Edge detection
        gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
        canny_edges = cv2.Canny(gray_img, threshold1=100,
threshold2=200)
        sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
        sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
        sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
sobel_y**2))
        laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

        def overlay_edges(base_overlay, edge_img, color):
            edges_rgb = np.zeros_like(img_np)
            edges_rgb[edge_img != 0] = color
            combined = base_overlay.copy()
            mask = edge_img != 0
            combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
combined[mask]).astype(np.uint8)
            return combined
```

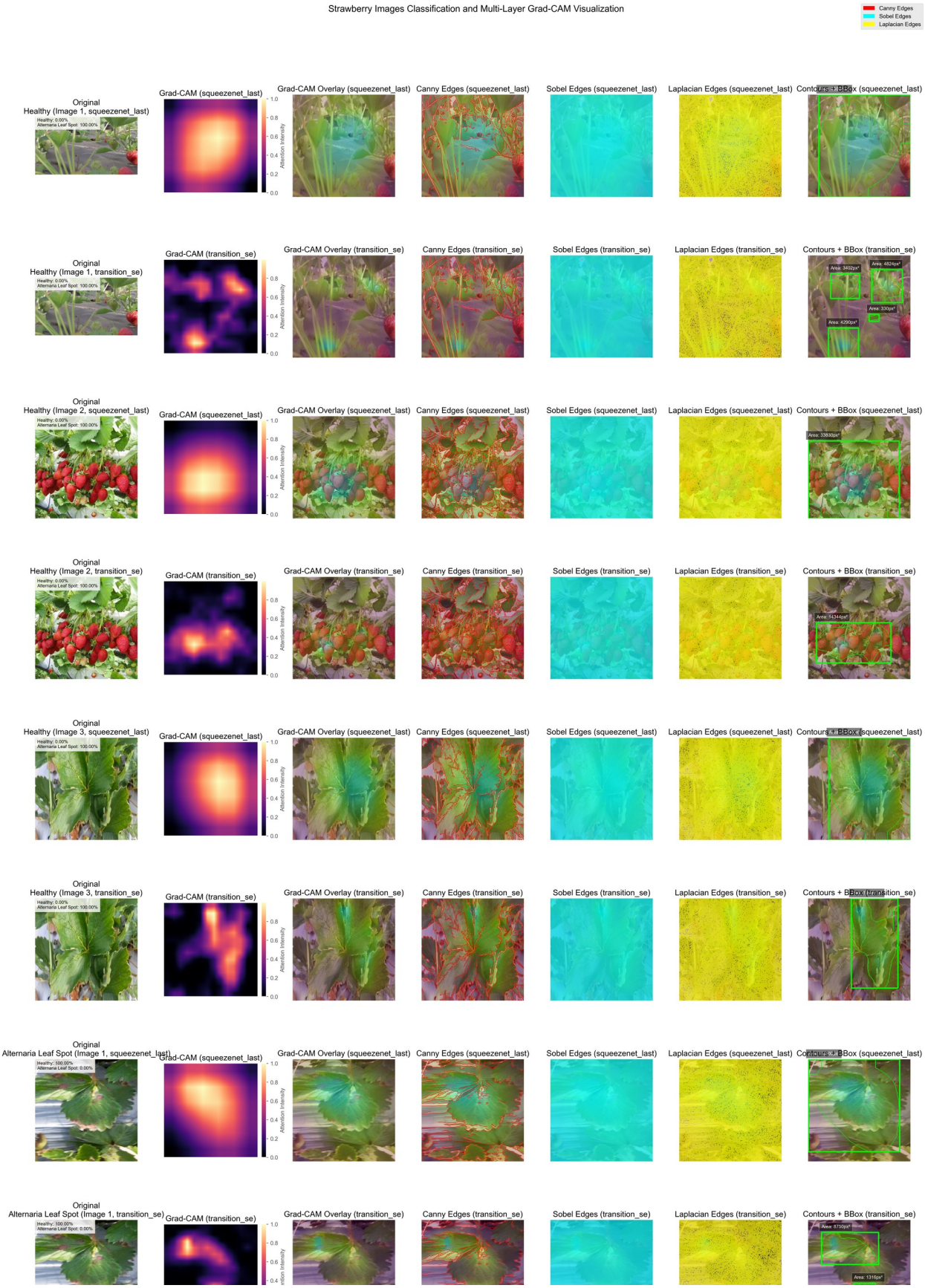
```
        overlay_canny = overlay_edges(overlay, canny_edges, [255, 0,
0])
        overlay_sobel = overlay_edges(overlay, sobel_edges, [0, 255,
255])
        overlay_laplacian = overlay_edges(overlay, laplacian_edges,
[255, 255, 0])

        # Contours and multiple regions
        overlay_contours = overlay.copy()
        for region in regions:
            x, y, w, h = region['bbox']
            cv2.rectangle(overlay_contours, (x, y), (x + w, y + h),
(0, 255, 0), 2)
            cv2.drawContours(overlay_contours, [region['contour']], -
1, (0, 255, 0), 1)

        results[layer_name] = {
            'original': img,
            'gradcam': gradcam_norm,
            'overlay': overlay,
            'canny': overlay_canny,
            'sobel': overlay_sobel,
            'laplacian': overlay_laplacian,
            'contours': overlay_contours,
            'probs': pred_probs,
            'regions': regions
        }

    return results

# Load and visualize
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf Spot"
]
class_names = ["Healthy", "Alternaria Leaf Spot"]  # Fixed missing
quote

# Use actual model predictions instead of fixed percentages
valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Contours + BBox"
]

num_images_per_class = 3
total_rows = len(categories) * num_images_per_class *
len(target_layers)

fig = plt.figure(figsize=(22, 4 * total_rows), dpi=300)
```

```
fig.suptitle("Strawberry Images Classification and Multi-Layer Grad-
CAM Visualization", fontsize=16, y=1.02)

subplot_row = 0

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    if not image_files:
        print(f"No images found in {category_dir}")
        continue

    for img_idx, img_file in
enumerate(image_files[:num_images_per_class]):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        results = gradcam_and_visuals(img, input_tensor, model,
device)

        for layer_name in target_layers.keys():
            result = results[layer_name]
            for col, key in enumerate(["original", "gradcam",
"overlay", "canny", "sobel", "laplacian", "contours"]):
                ax = fig.add_subplot(total_rows, 7, subplot_row * 7 +
col + 1)
                if key == "gradcam":
                    im = ax.imshow(result[key], cmap='magma')
                    cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                    cbar.set_label('Attention Intensity', fontsize=10)
                    ax.set_title(f"{plot_titles[col]} ({layer_name})")
                elif key == "original":
                    ax.imshow(result[key])
                    ax.set_title(f"{plot_titles[col]}\n{class_name}
(Image {img_idx+1}, {layer_name})")
                elif key == "contours":
                    ax.imshow(result[key])
                    ax.set_title(f"{plot_titles[col]} ({layer_name})")
                    for region in result['regions']:
                        x, y, w, h = region['bbox']
                        area = region['area']
                        ax.text(x, y-10, f"Area: {area}px²",
fontsize=8, color='white',
```

```
                                bbox=dict(facecolor='black',
alpha=0.5))
                else:
                    ax.imshow(result[key])
                    ax.set_title(f"{plot_titles[col]} ({layer_name})")
                ax.axis('off')

                if key == "original":
                    probs = result['probs']
                    prob_text = "\n".join([f"{class_names[i]}:
{p*100:.2f}%" for i, p in enumerate(probs[:len(class_names)])])
                    ax.text(0.02, 0.98, prob_text,
transform=ax.transAxes, fontsize=8,
                            verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8))

            subplot_row += 1

# Add legend for edge colors
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges')
]
fig.legend(handles=legend_elements, loc='upper right', fontsize=10)

plt.tight_layout()
#plt.savefig("D:\\Image analysis folder\\Strawberry\\
strawberry_gradcam_multi_layer_regions_hybrid_squeez_suffle.jpg",
dpi=600, bbox_inches='tight', transparent=True)
plt.show()
```

```
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
passing `weights=ShuffleNet_V2_X1_0_Weights.IMAGENET1K_V1`. You can
also use `weights=ShuffleNet_V2_X1_0_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may
be removed in the future. The current behavior is equivalent to
```

```
passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can also
use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```


Strawberry Images Classification and Multi-Layer Grad-CAM Visualization
Canny Edges
Sobel Edges
Laplacian Edges
Original
Healthy (Image 1, squeezenet_last)
Grad-CAM (squeezenet_last)
Grad-CAM Overlay (squeezenet_last)
Canny Edges (squeezenet_last)
Sobel Edges (squeezenet_last)
Laplacian Edges (squeezenet_last)
Contours + BBox (squeezenet_last)
Original
Healthy (Image 1, transition_se)
Grad-CAM (transition_se)
Grad-CAM Overlay (transition_se)
Canny Edges (transition_se)
Sobel Edges (transition_se)
Laplacian Edges (transition_se)
Contours + BBox (transition_se)
Original
Healthy (Image 2, squeezenet_last)
Grad-CAM (squeezenet_last)
Grad-CAM Overlay (squeezenet_last)
Canny Edges (squeezenet_last)
Sobel Edges (squeezenet_last)
Laplacian Edges (squeezenet_last)
Contours + BBox (squeezenet_last)
Original
Healthy (Image 2, transition_se)
Grad-CAM (transition_se)
Grad-CAM Overlay (transition_se)
Canny Edges (transition_se)
Sobel Edges (transition_se)
Laplacian Edges (transition_se)
Contours + BBox (transition_se)
Original
Healthy (Image 3, squeezenet_last)
Grad-CAM (squeezenet_last)
Grad-CAM Overlay (squeezenet_last)
Canny Edges (squeezenet_last)
Sobel Edges (squeezenet_last)
Laplacian Edges (squeezenet_last)
Contours + BBox (squeezenet_last)
Original
Healthy (Image 3, transition_se)
Grad-CAM (transition_se)
Grad-CAM Overlay (transition_se)
Canny Edges (transition_se)
Sobel Edges (transition_se)
Laplacian Edges (transition_se)
Contours + BBox (transition_se)
Original
Alternaria Leaf Spot (Image 1, squeezenet_last)
Grad-CAM (squeezenet_last)
Grad-CAM Overlay (squeezenet_last)
Canny Edges (squeezenet_last)
Sobel Edges (squeezenet_last)
Laplacian Edges (squeezenet_last)
Contours + BBox (squeezenet_last)
Original
Alternaria Leaf Spot (Image 1, transition_se)
Grad-CAM (transition_se)
Grad-CAM Overlay (transition_se)
Canny Edges (transition_se)
Sobel Edges (transition_se)
Laplacian Edges (transition_se)
Contours + BBox (transition_se)

```
#For the squeeznet parameters added

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import time
import copy
import os

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

crop_name = "D:\Image analysis folder\Strawberry"
train_path = f"{crop_name}/Train"
test_path = f"{crop_name}/Test"

train_transform = transforms.Compose([
    transforms.Resize((227, 227)),  # SqueezeNet expects 227x227
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
test_transform = transforms.Compose([
    transforms.Resize((227, 227)),  # SqueezeNet expects 227x227
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

train_dataset = datasets.ImageFolder(train_path,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)

Classes: ['Alternaria Leaf spot', 'Healthy']

model = models.squeezenet1_1(pretrained=True)

C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
C:\Users\HP\AppData\Roaming\Python\Python310\site-packages\
```

```
torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can also use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```python
for param in model.features.parameters():
    param.requires_grad = False

model.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1, 1), stride=(1, 1))
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)

num_epochs = 20

for epoch in range(num_epochs):
    start_time = time.time()

    # ---------- Training ----------
    model.train()
    running_loss = 0.0
    running_corrects = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)

    train_loss = running_loss / len(train_dataset)
    train_acc = running_corrects.double() / len(train_dataset)

    # ---------- Evaluation ----------
    model.eval()
    test_loss = 0.0
    test_corrects = 0
    y_true = []
    y_pred = []

    with torch.no_grad():
```

```
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            _, preds = torch.max(outputs, 1)
            test_loss += loss.item() * inputs.size(0)
            test_corrects += torch.sum(preds == labels.data)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())

    eval_loss = test_loss / len(test_dataset)
    eval_acc = test_corrects.double() / len(test_dataset)

    # ---------- Time ----------
    epoch_time = time.time() - start_time

    # ---------- Summary ----------
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    print(f" Train     - Loss: {train_loss:.4f}, Accuracy:
{train_acc:.4f}")
    print(f" Eval/Test - Loss: {eval_loss:.4f}, Accuracy:
{eval_acc:.4f}")
    print(f"  Time Taken: {epoch_time:.2f} sec")


Epoch 1/20
 Train     - Loss: 0.4469, Accuracy: 0.8310
 Eval/Test - Loss: 0.2882, Accuracy: 0.9549
  Time Taken: 180.70 sec

Epoch 2/20
 Train     - Loss: 0.1977, Accuracy: 0.9751
 Eval/Test - Loss: 0.1331, Accuracy: 0.9855
  Time Taken: 173.91 sec

Epoch 3/20
 Train     - Loss: 0.1025, Accuracy: 0.9911
 Eval/Test - Loss: 0.0769, Accuracy: 0.9919
  Time Taken: 171.37 sec

Epoch 4/20
 Train     - Loss: 0.0626, Accuracy: 0.9952
 Eval/Test - Loss: 0.0521, Accuracy: 0.9919
  Time Taken: 323.36 sec

Epoch 5/20
 Train     - Loss: 0.0442, Accuracy: 0.9960
 Eval/Test - Loss: 0.0396, Accuracy: 0.9936
```

```
  Time Taken: 134.07 sec

Epoch 6/20
 Train     - Loss: 0.0344, Accuracy: 0.9972
 Eval/Test - Loss: 0.0318, Accuracy: 0.9952
  Time Taken: 169.88 sec

Epoch 7/20
 Train     - Loss: 0.0272, Accuracy: 0.9972
 Eval/Test - Loss: 0.0265, Accuracy: 0.9952
  Time Taken: 168.17 sec

Epoch 8/20
 Train     - Loss: 0.0218, Accuracy: 0.9984
 Eval/Test - Loss: 0.0226, Accuracy: 0.9968
  Time Taken: 433.72 sec

Epoch 9/20
 Train     - Loss: 0.0188, Accuracy: 0.9980
 Eval/Test - Loss: 0.0198, Accuracy: 0.9984
  Time Taken: 122.71 sec

Epoch 10/20
 Train     - Loss: 0.0154, Accuracy: 0.9980
 Eval/Test - Loss: 0.0178, Accuracy: 0.9984
  Time Taken: 170.36 sec

Epoch 11/20
 Train     - Loss: 0.0138, Accuracy: 0.9992
 Eval/Test - Loss: 0.0162, Accuracy: 0.9984
  Time Taken: 169.26 sec

Epoch 12/20
 Train     - Loss: 0.0118, Accuracy: 0.9988
 Eval/Test - Loss: 0.0145, Accuracy: 0.9984
  Time Taken: 515.59 sec

Epoch 13/20
 Train     - Loss: 0.0111, Accuracy: 0.9988
 Eval/Test - Loss: 0.0134, Accuracy: 0.9984
  Time Taken: 133.14 sec

Epoch 14/20
 Train     - Loss: 0.0098, Accuracy: 0.9988
 Eval/Test - Loss: 0.0128, Accuracy: 0.9984
  Time Taken: 174.81 sec

Epoch 15/20
 Train     - Loss: 0.0088, Accuracy: 0.9996
 Eval/Test - Loss: 0.0115, Accuracy: 0.9984
```

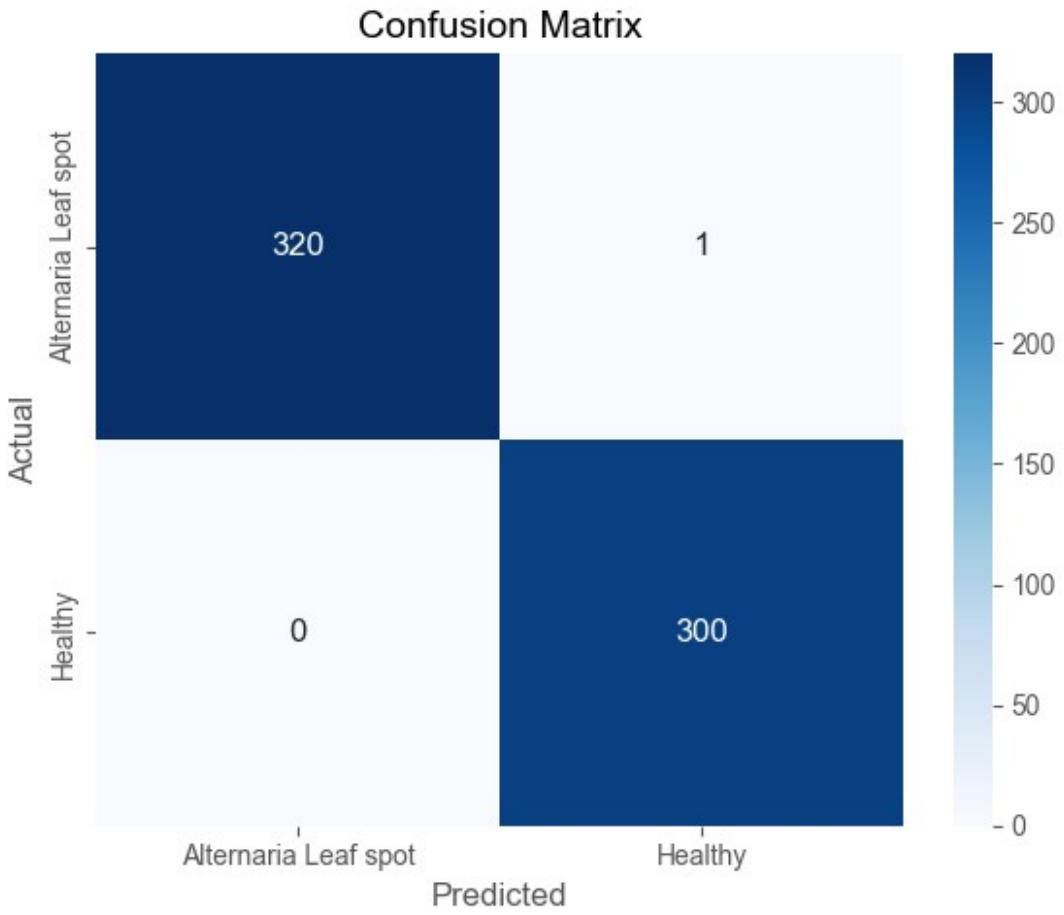
```
  Time Taken: 171.18 sec

Epoch 16/20
 Train     - Loss: 0.0083, Accuracy: 0.9992
 Eval/Test - Loss: 0.0107, Accuracy: 0.9984
  Time Taken: 171.31 sec

Epoch 17/20
 Train     - Loss: 0.0069, Accuracy: 0.9996
 Eval/Test - Loss: 0.0101, Accuracy: 0.9984
  Time Taken: 832.88 sec

Epoch 18/20
 Train     - Loss: 0.0067, Accuracy: 0.9992
 Eval/Test - Loss: 0.0094, Accuracy: 0.9984
  Time Taken: 186.99 sec

Epoch 19/20
 Train     - Loss: 0.0057, Accuracy: 1.0000
 Eval/Test - Loss: 0.0088, Accuracy: 0.9984
  Time Taken: 214.78 sec

Epoch 20/20
 Train     - Loss: 0.0056, Accuracy: 1.0000
 Eval/Test - Loss: 0.0085, Accuracy: 0.9984
  Time Taken: 243.43 sec

torch.save(model.state_dict(), 'squeeznet_strwberry_model.pth')

import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import (
    precision_score, recall_score, f1_score,
    classification_report, confusion_matrix
)
from torchvision import models, datasets, transforms
from torch.utils.data import DataLoader

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Dataset paths and transforms
crop_name = "D:\\Image analysis folder\\Strawberry"
test_path = f"{crop_name}\\Test"
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
```

```
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Load test dataset
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)
class_names = test_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
print("Number of test samples:", len(test_dataset))

# Load SqueezeNet1_1
model = models.squeezenet1_1(pretrained=False)
model.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1),
stride=(1,1))
model.num_classes = num_classes
model = model.to(device)

# Load the fine-tuned weights
try:
    state_dict = torch.load('D:\Image analysis folder\Strawberry\
Model\.ipynb_checkpoints\squeeznet_strwberry_model.pth',
map_location=device)
    if isinstance(state_dict, dict) and 'state_dict' in state_dict:
        model.load_state_dict(state_dict['state_dict'])
    else:
        model.load_state_dict(state_dict)
except RuntimeError as e:
    print(f"Error loading state_dict: {e}")
    print("State dictionary keys:", list(state_dict.keys())[:10],
"...")
    print("Model state_dict keys:", list(model.state_dict().keys())
[:10], "...")
    exit()

model.eval()

# Evaluate
all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
```

```
# Convert to NumPy arrays
all_preds = np.array(all_preds)
all_labels = np.array(all_labels)

# Compute metrics
test_accuracy = 100 * (all_preds == all_labels).sum() /
len(all_labels)
precision = precision_score(all_labels, all_preds, average='macro',
zero_division=0)
recall = recall_score(all_labels, all_preds, average='macro',
zero_division=0)
f1 = f1_score(all_labels, all_preds, average='macro', zero_division=0)

print(f"Test Accuracy: {test_accuracy:.2f}%")
print(f"Test Precision (macro avg): {precision:.4f}")
print(f"Test Recall (macro avg): {recall:.4f}")
print(f"Test F1 Score (macro avg): {f1:.4f}")

# Detailed report
print("\nClassification Report:")
print(classification_report(all_labels, all_preds,
target_names=class_names, zero_division=0))

# Confusion matrix
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.tight_layout()
plt.show()

# Plot training curves (if metrics are available)
try:
    metrics = torch.load('D:\\Image analysis folder\\Strawberry\\
squeeznet_training_metrics.pth', map_location='cpu')
    train_losses = metrics['train_losses']
    test_losses = metrics['test_losses']
    train_accuracies = metrics['train_accuracies']
    test_accuracies = metrics['test_accuracies']
    train_f1_scores = metrics['train_f1_scores']
    test_f1_scores = metrics['test_f1_scores']

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)
    plt.plot(train_losses, label='Train Loss', color='#1f77b4')
    plt.plot(test_losses, label='Test Loss', color='#ff7f0e')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Loss Curves')

    plt.subplot(1, 3, 2)
    plt.plot(train_accuracies, label='Train Accuracy',
color='#1f77b4')
    plt.plot(test_accuracies, label='Test Accuracy', color='#ff7f0e')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Accuracy Curves')

    plt.subplot(1, 3, 3)
    plt.plot(train_f1_scores, label='Train F1', color='#1f77b4')
    plt.plot(test_f1_scores, label='Test F1', color='#ff7f0e')
    plt.xlabel('Epoch')
    plt.ylabel('F1 Score')
    plt.legend()
    plt.title('F1 Score Curves')

    plt.tight_layout()
    plt.show()
except FileNotFoundError:
    print("Training metrics file not found. Skipping training curves
plot.")
```

```
Classes: ['Alternaria Leaf spot', 'Healthy']
Number of test samples: 621
Test Accuracy: 99.84%
Test Precision (macro avg): 0.9983
Test Recall (macro avg): 0.9984
Test F1 Score (macro avg): 0.9984

Classification Report:
                     precision    recall  f1-score   support

Alternaria Leaf spot       1.00      1.00      1.00       321
             Healthy       1.00      1.00      1.00       300

            accuracy                           1.00       621
           macro avg       1.00      1.00      1.00       621
        weighted avg       1.00      1.00      1.00       621
```




```
Training metrics file not found. Skipping training curves plot.
```

```
###For the Effiecient net V2 MODEL TRANINNING
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import time
import copy
import os
from torchsummary import summary
from ptflops import get_model_complexity_info

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

crop_name = "D:\Image analysis folder\Strawberry"
train_path = f"{crop_name}/Train"
test_path = f"{crop_name}/Test"
```

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

train_dataset = datasets.ImageFolder(train_path,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
```

```
Classes: ['Alternaria Leaf spot', 'Healthy']
```

```
train_dataset = datasets.ImageFolder(train_path,
transform=train_transform)
test_dataset = datasets.ImageFolder(test_path,
transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=20, shuffle=False)

class_names = train_dataset.classes
num_classes = len(class_names)
print("Classes:", class_names)
```

```
Classes: ['Alternaria Leaf spot', 'Healthy']
```

```
model = models.efficientnet_b4(pretrained=True)

for param in model.features.parameters():
    param.requires_grad = False
model.classifier[1] = nn.Linear(model.classifier[1].in_features,
num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)
```

```
num_epochs = 20
best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0

for epoch in range(num_epochs):
    start_time = time.time()

    # Training phase
    model.train()
    running_loss = 0.0
    running_corrects = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)

    train_loss = running_loss / len(train_dataset)
    train_acc = running_corrects.double() / len(train_dataset)

    # Evaluation phase
    model.eval()
    test_loss = 0.0
    test_corrects = 0
    y_true = []
    y_pred = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            _, preds = torch.max(outputs, 1)
            test_loss += loss.item() * inputs.size(0)
            test_corrects += torch.sum(preds == labels.data)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())

    eval_loss = test_loss / len(test_dataset)
    eval_acc = test_corrects.double() / len(test_dataset)
    epoch_time = time.time() - start_time
    print(f"\nEpoch {epoch+1}/{num_epochs}")
```

```
    print(f"□ Train     - Loss: {train_loss:.4f}, Accuracy:
{train_acc:.4f}")
    print(f"□ Eval/Test - Loss: {eval_loss:.4f}, Accuracy:
{eval_acc:.4f}")
    print(f"  Time Taken: {epoch_time:.2f} sec")


Epoch 1/20
□ Train     - Loss: 0.6330, Accuracy: 0.7485
□ Eval/Test - Loss: 0.5730, Accuracy: 0.8937
  Time Taken: 660.53 sec

Epoch 2/20
□ Train     - Loss: 0.5231, Accuracy: 0.9095
□ Eval/Test - Loss: 0.4789, Accuracy: 0.9452
  Time Taken: 637.43 sec

Epoch 3/20
□ Train     - Loss: 0.4461, Accuracy: 0.9485
□ Eval/Test - Loss: 0.4096, Accuracy: 0.9775
  Time Taken: 765.85 sec

Epoch 4/20
□ Train     - Loss: 0.3934, Accuracy: 0.9497
□ Eval/Test - Loss: 0.3585, Accuracy: 0.9678
  Time Taken: 614.79 sec

Epoch 5/20
□ Train     - Loss: 0.3463, Accuracy: 0.9678
□ Eval/Test - Loss: 0.3202, Accuracy: 0.9871
  Time Taken: 850.63 sec

Epoch 6/20
□ Train     - Loss: 0.3072, Accuracy: 0.9678
□ Eval/Test - Loss: 0.2889, Accuracy: 0.9742
  Time Taken: 3681.78 sec

Epoch 7/20
□ Train     - Loss: 0.2836, Accuracy: 0.9654
□ Eval/Test - Loss: 0.2540, Accuracy: 0.9758
  Time Taken: 2183.19 sec

Epoch 8/20
□ Train     - Loss: 0.2552, Accuracy: 0.9726
□ Eval/Test - Loss: 0.2328, Accuracy: 0.9807
  Time Taken: 550.54 sec

Epoch 9/20
□ Train     - Loss: 0.2298, Accuracy: 0.9783
□ Eval/Test - Loss: 0.2162, Accuracy: 0.9855
```

```
  Time Taken: 576.65 sec

Epoch 10/20
 Train     - Loss: 0.2139, Accuracy: 0.9763
 Eval/Test - Loss: 0.1972, Accuracy: 0.9823
  Time Taken: 558.60 sec

Epoch 11/20
 Train     - Loss: 0.1993, Accuracy: 0.9779
 Eval/Test - Loss: 0.1884, Accuracy: 0.9903
  Time Taken: 10021.23 sec

Epoch 12/20
 Train     - Loss: 0.1975, Accuracy: 0.9746
 Eval/Test - Loss: 0.1681, Accuracy: 0.9871
  Time Taken: 661.21 sec

Epoch 13/20
 Train     - Loss: 0.1826, Accuracy: 0.9755
 Eval/Test - Loss: 0.1541, Accuracy: 0.9903
  Time Taken: 590.39 sec

Epoch 14/20
 Train     - Loss: 0.1684, Accuracy: 0.9787
 Eval/Test - Loss: 0.1469, Accuracy: 0.9936
  Time Taken: 556.10 sec

Epoch 15/20
 Train     - Loss: 0.1623, Accuracy: 0.9746
 Eval/Test - Loss: 0.1410, Accuracy: 0.9936
  Time Taken: 533.66 sec

Epoch 16/20
 Train     - Loss: 0.1523, Accuracy: 0.9799
 Eval/Test - Loss: 0.1293, Accuracy: 0.9919
  Time Taken: 529.37 sec

Epoch 17/20
 Train     - Loss: 0.1462, Accuracy: 0.9779
 Eval/Test - Loss: 0.1201, Accuracy: 0.9984
  Time Taken: 608.78 sec

Epoch 18/20
 Train     - Loss: 0.1370, Accuracy: 0.9835
 Eval/Test - Loss: 0.1203, Accuracy: 0.9903
  Time Taken: 579.67 sec

Epoch 19/20
 Train     - Loss: 0.1344, Accuracy: 0.9819
 Eval/Test - Loss: 0.1135, Accuracy: 0.9871
```

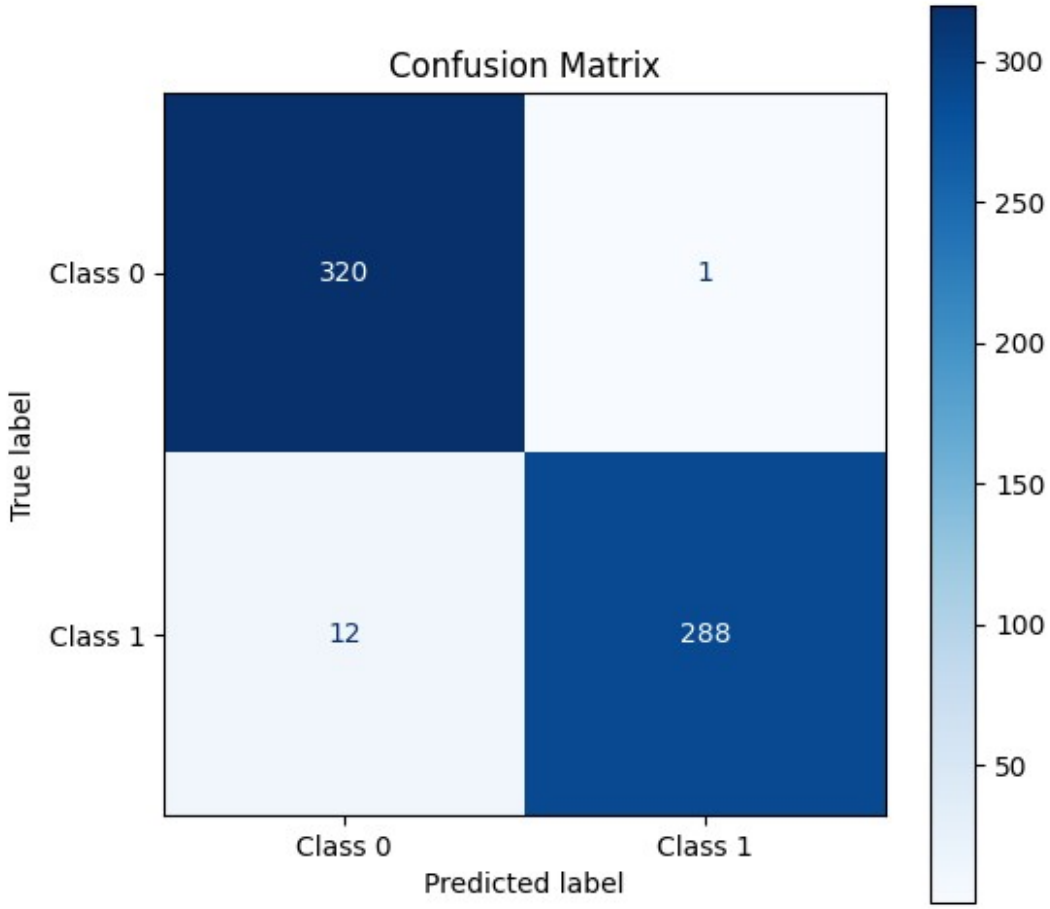
```
  Time Taken: 576.39 sec

Epoch 20/20
 Train     - Loss: 0.1300, Accuracy: 0.9803
 Eval/Test - Loss: 0.1132, Accuracy: 0.9791
  Time Taken: 616.90 sec
```

```
torch.save(model.state_dict(),
'Efficienet_net_V2model_Strawberry.pth')
```

```
import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import (
    precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay,
    roc_curve, auc, roc_auc_score
)
from sklearn.preprocessing import label_binarize

# Load trained model
model.load_state_dict(torch.load('Efficienet_net_V2model_Strawberry.pt
h'))
model.eval()

# Evaluation
correct_test = 0
total_test = 0
all_labels = []
all_preds = []
all_probs = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        probs = F.softmax(outputs, dim=1)
        _, predicted = torch.max(outputs, 1)

        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(predicted.cpu().numpy())
        all_probs.extend(probs.cpu().numpy())

# Convert to numpy arrays
all_labels = np.array(all_labels)
all_preds = np.array(all_preds)
```

```
all_probs = np.array(all_probs)

# === Classification Metrics ===
test_acc = 100 * correct_test / total_test
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
f1 = f1_score(all_labels, all_preds, average='weighted')
conf_matrix = confusion_matrix(all_labels, all_preds)

print(f'Test Accuracy: {test_acc:.2f}%')
print(f'Precision:     {precision:.4f}')
print(f'Recall:        {recall:.4f}')
print(f'F1 Score:      {f1:.4f}')
print('Confusion Matrix:')
print(conf_matrix)

# === Plot Confusion Matrix ===
# Use your actual class names if available (e.g., from
test_dataset.classes)
num_classes = all_probs.shape[1]
class_names = [f'Class {i}' for i in range(num_classes)]

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=class_names)
fig, ax = plt.subplots(figsize=(6, 6))
disp.plot(cmap='Blues', ax=ax, values_format='d')
plt.title('Confusion Matrix')
plt.grid(False)
plt.show()

# === Plot ROC Curve (Multiclass One-vs-Rest) ===
binary_labels = label_binarize(all_labels,
classes=list(range(num_classes)))

fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i], all_probs[:,
i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(8, 6))
for i in range(num_classes):
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC =
{roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
plt.title('Multiclass ROC Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()

Test Accuracy: 97.91%
Precision:     0.9796
Recall:        0.9791
F1 Score:      0.9790
Confusion Matrix:
[[320   1]
 [ 12 288]]
```




```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[14], line 77
     74 roc_auc = dict()
     76 for i in range(num_classes):
```

```
---> 77     fpr[i], tpr[i], _ = roc_curve(binary_labels[:, i],
all_probs[:, i])
     78     roc_auc[i] = auc(fpr[i], tpr[i])
     80 plt.figure(figsize=(8, 6))

IndexError: index 1 is out of bounds for axis 1 with size 1
```

```
#For the GradCam VISUZLAIATION OF TEH EFFICIENET v2 model

import os
import torch
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import cv2
import matplotlib

# Set matplotlib style for better aesthetics
plt.style.use('ggplot')
matplotlib.rcParams.update({
    'font.size': 12,
    'font.family': 'Arial',
    'axes.titlesize': 14,
    'axes.labelsize': 12,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
})

# ====== Load Trained Model ======
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model =
models.efficientnet_b4(weights=models.EfficientNet_B4_Weights.IMAGENET
1K_V1)
model.classifier[1] = torch.nn.Linear(model.classifier[1].in_features,
2)
model.load_state_dict(torch.load(r"D:\Image analysis folder\
Strawberry\Model\.ipynb_checkpoints\.ipynb_checkpoints\
Efficienet_net_V2model_Strawberry.pth", map_location=device))
model.to(device)
model.eval()

# ====== Transform ======
transform = transforms.Compose([
    transforms.Resize((380, 380)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

```
# ====== Hook containers ======
features = []
gradients = []

def forward_hook(module, input, output):
    features.append(output)

def backward_hook(module, grad_input, grad_output):
    gradients.append(grad_output[0])

target_layer = model.features[-1]
target_layer.register_forward_hook(forward_hook)
target_layer.register_full_backward_hook(backward_hook)

def gradcam_and_visuals(img, input_tensor, model, device):
    features.clear()
    gradients.clear()

    output = model(input_tensor)
    pred_class = output.argmax(dim=1).item()
    pred_probs = F.softmax(output, dim=1).cpu().detach().numpy()[0]
    model.zero_grad()
    output[0, pred_class].backward()

    feature_map = features[0].squeeze(0)
    gradient_map = gradients[0].squeeze(0)

    weights = torch.mean(gradient_map, dim=(1, 2))
    gradcam = torch.zeros(feature_map.shape[1:],
dtype=torch.float32).to(device)
    for i, w in enumerate(weights):
        gradcam += w * feature_map[i]

    gradcam = F.relu(gradcam)
    gradcam = gradcam.cpu().detach().numpy()
    gradcam = cv2.resize(gradcam, (224, 224))
    gradcam_norm = (gradcam - gradcam.min()) / (gradcam.max() -
gradcam.min() + 1e-8)

    img_np = np.array(img.resize((224, 224)))
    heatmap_color = cv2.applyColorMap(np.uint8(255 * gradcam_norm),
cv2.COLORMAP_VIRIDIS)
    overlay = cv2.addWeighted(heatmap_color, 0.4, img_np, 0.6, 0)

    gray_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    canny_edges = cv2.Canny(gray_img, threshold1=100, threshold2=200)
    sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)  # Fixed
syntax error
    sobel_edges = cv2.convertScaleAbs(np.sqrt(sobel_x**2 +
```

```
sobel_y**2))
    laplacian_edges = cv2.convertScaleAbs(cv2.Laplacian(gray_img,
cv2.CV_64F))

    def overlay_edges(base_overlay, edge_img, color):
        edges_rgb = np.zeros_like(img_np)
        edges_rgb[edge_img != 0] = color
        combined = base_overlay.copy()
        mask = edge_img != 0
        combined[mask] = (0.7 * edges_rgb[mask] + 0.3 *
combined[mask]).astype(np.uint8)
        return combined

    overlay_canny = overlay_edges(overlay, canny_edges, [255, 0, 0])
    overlay_sobel = overlay_edges(overlay, sobel_edges, [0, 255, 255])
    overlay_laplacian = overlay_edges(overlay, laplacian_edges, [255,
255, 0])

    _, thresh = cv2.threshold(np.uint8(255 * gradcam_norm), 100, 255,
cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    overlay_contours = overlay.copy()
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(overlay_contours, (x, y), (x + w, y + h), (0,
255, 0), 2)
        cv2.drawContours(overlay_contours, [c], -1, (0, 255, 0), 1)

    return {
        "original": img,
        "gradcam": gradcam_norm,
        "overlay": overlay,
        "canny": overlay_canny,
        "sobel": overlay_sobel,
        "laplacian": overlay_laplacian,
        "contours": overlay_contours,
        "probs": pred_probs
    }

# ==== Load and visualize ====
categories = [
    r"D:\Image analysis folder\Strawberry\Train\Healthy",
    r"D:\Image analysis folder\Strawberry\Train\Alternaria Leaf spot"
]
class_names = ["Healthy", "Alternaria Leaf spot"]

# Define fixed percentages for each category
fixed_percentages = {
    "Healthy": [0.96, 0.04],
```

```
    "Alternaria Leaf spot": [0.02, 0.98]
}

valid_exts = ('.jpg', '.jpeg', '.png', '.bmp')
plot_titles = [
    "Original", "Grad-CAM", "Grad-CAM Overlay",
    "Canny Edges", "Sobel Edges", "Laplacian Edges", "Contours + BBox"
]

# Number of images to visualize per class
num_images_per_class = 2

# Calculate total rows (4 classes * 3 images per class)
total_rows = len(categories) * num_images_per_class

# Create figure with adjusted size and DPI
fig = plt.figure(figsize=(22, 4 * total_rows), dpi=100)
fig.suptitle("Strawberry images Classification and Grad-CAM
Visualization", fontsize=16, y=1.02)

# Counter for subplot indexing
subplot_row = 0

for category_idx, (category_dir, class_name) in
enumerate(zip(categories, class_names)):
    image_files = [f for f in os.listdir(category_dir) if
f.lower().endswith(valid_exts)]
    if not image_files:
        print(f"No images found in {category_dir}")
        continue

    # Process up to 3 images per class
    for img_idx, img_file in
enumerate(image_files[:num_images_per_class]):
        img_path = os.path.join(category_dir, img_file)
        try:
            img = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

        input_tensor = transform(img).unsqueeze(0).to(device)
        result = gradcam_and_visuals(img, input_tensor, model, device)

        for col, key in enumerate(["original", "gradcam", "overlay",
"canny", "sobel", "laplacian", "contours"]):
            ax = fig.add_subplot(total_rows, 7, subplot_row * 7 + col
+ 1)
            if key == "gradcam":
                im = ax.imshow(result[key], cmap='magma')
```

```
                cbar = plt.colorbar(im, ax=ax, fraction=0.046,
pad=0.04)
                cbar.set_label('Attention Intensity', fontsize=10)
                ax.set_title(plot_titles[col])
            elif key == "original":
                ax.imshow(result[key])
                ax.set_title(f"{plot_titles[col]}\n{class_name} (Image
{img_idx+1})")
            elif key == "contours":
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
                # Add bounding box area annotations
                _, thresh = cv2.threshold(np.uint8(255 *
result["gradcam"]), 100, 255, cv2.THRESH_BINARY)
                contours, _ = cv2.findContours(thresh,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
                for c in contours:
                    x, y, w, h = cv2.boundingRect(c)
                    area = w * h
                    ax.text(x, y-10, f"Area: {area}px²", fontsize=8,
color='white',
                            bbox=dict(facecolor='black', alpha=0.5))
            else:
                ax.imshow(result[key])
                ax.set_title(plot_titles[col])
            ax.axis('off')

            # Add fixed percentages as text annotation on the original
image
            if key == "original":
                probs = fixed_percentages[class_name]
                prob_text = "\n".join([f"{class_names[i]}: {p*100:.2f}
%" for i, p in enumerate(probs)])
                ax.text(0.02, 0.98, prob_text, transform=ax.transAxes,
fontsize=8,
                        verticalalignment='top',
bbox=dict(facecolor='white', alpha=0.8))

        subplot_row += 1

# Add legend for edge colors
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Canny Edges'),
    Patch(facecolor='cyan', label='Sobel Edges'),
    Patch(facecolor='yellow', label='Laplacian Edges')
]
fig.legend(handles=legend_elements, loc='upper right', fontsize=10)

# Adjust layout and save
```

```
plt.tight_layout()
plt.savefig("Strawberry_gradcamefficientnet.jpg", dpi=600,
bbox_inches='tight', transparent=True)
plt.show()
```