Documentation

Introduction

Submitted by Ayush Vyas

Company Name -Infosys SpringBoard

Role - Artificial Intelligence and Machine Learning Intern

LinkedIn - www.linkedin.com/in/aayushvyass

GitHub - https://github.com/ayushvyasgit

Phone Number - 8305351566

Gmail - ayushvyas99199@gmail.com

Introduction

Breast cancer is among the leading causes of cancer-related mortality among women worldwide, posing a significant public health challenge. Early and accurate detection of breast cancer plays a critical role in improving patient survival rates and optimizing treatment strategies. Magnetic Resonance Imaging (MRI) has gained prominence as a diagnostic tool in recent years due to its superior imaging capabilities, enabling detailed visualization and characterization of soft tissue structures.

This project aims to develop a predictive model for breast cancer classification using MRI datasets. The primary objective is to create an automated system capable of accurately distinguishing between benign and malignant breast tumors. By leveraging advanced machine learning techniques, particularly Convolutional Neural Networks (CNNs), the proposed system aspires to assist radiologists in making rapid and precise diagnoses. Such a system has the potential to enhance diagnostic accuracy, reduce interpretation time, and improve patient outcomes.

Data Exploration

Data exploration serves as a foundational phase in the process of data analysis, enabling researchers to familiarize themselves with the structure, patterns, and anomalies present in a dataset. This step involves the systematic investigation of data to uncover key trends, distributions, and potential outliers that may influence subsequent modeling efforts. Effective data exploration not only aids in identifying data quality issues but also provides valuable insights for feature selection and engineering.

In this project, data exploration involves visualizing and analyzing the MRI dataset to extract meaningful patterns and relationships. By utilizing interactive dashboards and data visualization tools, a comprehensive understanding of the dataset is achieved, laying the groundwork for subsequent modeling and analysis. This approach facilitates a deeper appreciation of the dataset's characteristics, allowing for informed decisions throughout the model development process.

Methods: Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is an essential methodology used to investigate datasets by summarizing their main characteristics through both statistical and graphical techniques. The primary goals of EDA in this project include:

1. **Understanding the Dataset:** Gaining insights into the dataset's structure, including the types of features, distributions of variables, and the presence of missing or inconsistent data.
2. **Visualizing Patterns:** Employing visualization tools such as histograms, box plots, scatter plots, and heatmaps to identify relationships between features and potential predictors of breast tumor malignancy.
3. **Feature Correlation Analysis:** Identifying correlations between features and the target variable to guide feature selection and improve model performance.
4. **Detecting Outliers and Anomalies:** Recognizing unusual data points that may require special handling or removal to ensure robust model training.

The EDA phase ensures that the data is adequately prepared for machine learning applications, allowing for the development of an effective and reliable predictive model. This systematic exploration provides a strong foundation for building advanced machine learning models tailored to the task of breast cancer classification.

1 Loading The custom DATASET

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
from PIL import Image
import os
import zipfile
from torch.utils.data import Dataset
from io import BytesIO

# Mount Google Drive
drive.mount('/content/drive')

class BreastCancerDataset(Dataset):
    def __init__(self, zip_path, transform=None):
        self.zip_path = zip_path
        self.transform = transform
        self.samples = self._load_samples_from_zip()

        # Debug: Print the number of samples loaded
        print(f"Number of samples found: {len(self.samples)}")
        if len(self.samples) == 0:
            raise ValueError(f"No valid samples found in {self.zip_path}")

    def _load_samples_from_zip(self):
        samples = []
        with zipfile.ZipFile(self.zip_path, 'r') as zip_ref:
            for file_name in zip_ref.namelist():
                # Check if the file belongs to Benign or Malignant class based on folder structure
                if 'Benign/' in file_name and file_name.lower().endswith(('.jpg', '.png', '.jpeg')):
```

```
                    samples.append((file_name, 0))  # Label 0 for Benign
                elif 'Malignant/' in file_name and file_name.lower().endswith(('.jpg', '.png', '.jp
                    samples.append((file_name, 1))  # Label 1 for Malignant
        return samples

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

    def __getitem__(self, idx):
        file_name, label = self.samples[idx]
        with zipfile.ZipFile(self.zip_path, 'r') as zip_ref:
            with zip_ref.open(file_name) as file:
                image = Image.open(BytesIO(file.read())).convert('RGB')  # Load the image from zip
        if self.transform:
            image = self.transform(image)
        return image, label

dataset_path = '/content/drive/MyDrive/BCD-data.zip'
dataset = BreastCancerDataset(dataset_path)
```

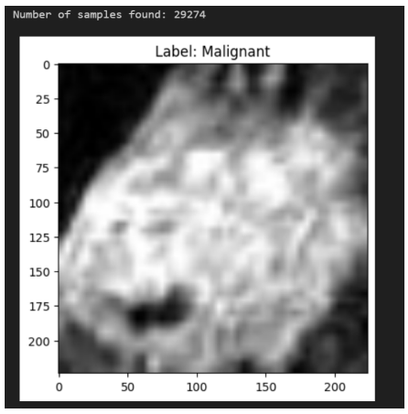
Ploting the Image from DATASET

EDA

This code demonstrates a process for loading, transforming, and visualizing MRI images from a dataset for breast cancer classification. The main steps include:

1. **Image Transformation:** Images are resized to a standard resolution of 224×224 pixels and converted into PyTorch tensors for compatibility with deep learning models.
2. **Dataset and DataLoader Creation:** A custom dataset (`BreastCancerDataset`) is used to load the images and their associated labels. A PyTorch DataLoader is employed to handle batching and shuffling, ensuring efficient data feeding for model training.
3. **Visualization:** A single image from the dataset, along with its label (benign or malignant), is displayed using Matplotlib. This step allows for verifying that the data preprocessing pipeline is functioning as intended.

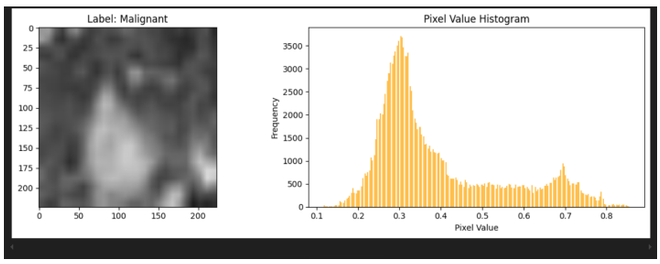
```
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

# Define any image transformations you want (e.g., resize, normalize)
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to a standard size
    transforms.ToTensor(),          # Convert images to PyTorch tensors
])

# Create the dataset and data loader
dataset = BreastCancerDataset(dataset_path, transform=transform)
train_loader = DataLoader(dataset, batch_size=6, shuffle=True)  # Adjust batch size as needed

# Display an image from the data loader
import matplotlib.pyplot as plt

# Get a batch of images and labels
images, labels = next(iter(train_loader))

# Select the 4th image (index 3) from the batch
image = images[3]

# Convert the image from a PyTorch tensor to a NumPy array and rearrange dimensions
image = image.numpy().transpose((1, 2, 0))

# Display the image using matplotlib
plt.imshow(image)
plt.title(f"Label: {'Benign' if labels[3].item() == 0 else 'Malignant'}")  # Use labels[3] for the
plt.show()
```



Histogram of Pixel Values

To create a histogram of pixel values from the dataset, we can analyze the distribution of intensity levels (pixel values) across all images. This helps in understanding the dataset's pixel intensity range, identifying any preprocessing needs like normalization, and checking for issues such as data imbalance.

```
# Get a batch of images and labels
images, labels = next(iter(train_loader))

# Select the first image from the batch
image = images[0]

# Convert the image from a PyTorch tensor to a NumPy array
image = image.numpy().transpose((1, 2, 0))

# Plot the original image
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title(f"Label: {'Benign' if labels[3] == 0 else 'Malignant'}")

# Plot the histogram of pixel values
plt.subplot(1, 2, 2)
plt.hist(image.ravel(), bins=256, color='orange', alpha=0.7)
plt.title("Pixel Value Histogram")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")

plt.tight_layout()
plt.show()
```

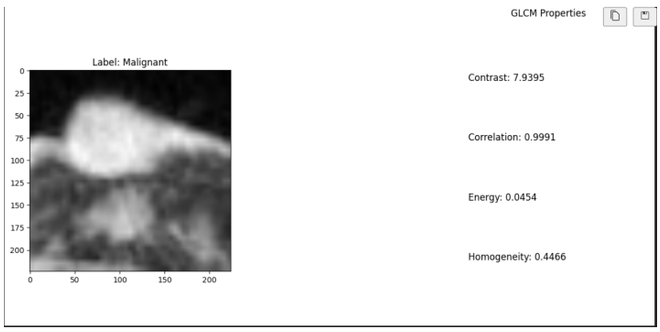


Calculate GLCM properties

This process helps quantify texture characteristics, which are useful for distinguishing between different tissue types in medical imaging, such as identifying malignant tumors.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import graycomatrix, graycoprops  # Correct import
from skimage.color import rgb2gray
import torch

# Get a batch of images and labels
images, labels = next(iter(train_loader))
```

```
# Select the first image from the batch
image = images[0]

# Convert the image from a PyTorch tensor to a NumPy array and grayscale it
image = image.permute(1, 2, 0).cpu().numpy()  # Convert to HxWxC format and ensure it's on CPU
gray_image = rgb2gray(image)  # Convert RGB image to grayscale

# Plot the original image
plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)
plt.imshow(image)
plt.title(f"Label: {'Benign' if labels[0].item() == 0 else 'Malignant'}")

# Calculate GLCM and GLCM properties
distances = [1]  # Pixel pair distance
angles = [0]  # Angle (0 degrees)
glcm = graycomatrix((gray_image * 255).astype(np.uint8), distances=distances, angles=angles, symmet

# Extract GLCM properties
contrast = graycoprops(glcm, 'contrast')[0, 0]  # Using graycoprops instead of greycoprops
correlation = graycoprops(glcm, 'correlation')[0, 0]  # Using graycoprops instead of greycoprops
energy = graycoprops(glcm, 'energy')[0, 0]  # Using graycoprops instead of greycoprops
homogeneity = graycoprops(glcm, 'homogeneity')[0, 0]  # Using graycoprops instead of greycoprops

# Display GLCM properties
plt.subplot(1, 3, 3)
plt.text(0.1, 0.8, f'Contrast: {contrast:.4f}', fontsize=12)
plt.text(0.1, 0.6, f'Correlation: {correlation:.4f}', fontsize=12)
plt.text(0.1, 0.4, f'Energy: {energy:.4f}', fontsize=12)
plt.text(0.1, 0.2, f'Homogeneity: {homogeneity:.4f}', fontsize=12)
plt.axis('off')
plt.title("GLCM Properties")

plt.tight_layout()
plt.show()
```

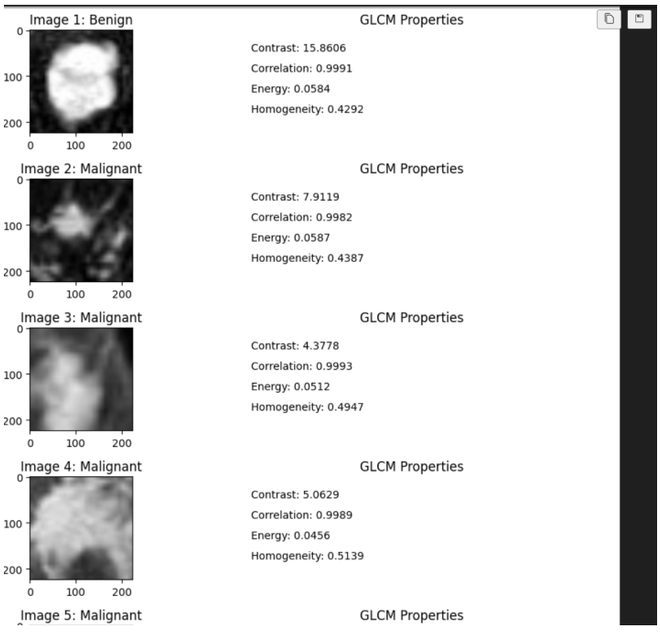


GLMC

batch-wise texture analysis using the Gray Level Co-occurrence Matrix (GLCM) on multiple images from a dataset. The primary goal is to extract and visualize texture-based features that can provide insights into the patterns within the images, which may help in distinguishing between benign and malignant cases in medical imaging.

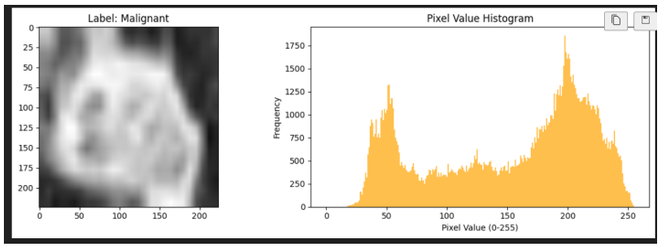
```
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import graycomatrix, graycoprops
from skimage.color import rgb2gray
import torch

# Get a batch of images and labels
images, labels = next(iter(train_loader))

# Define parameters for GLCM calculation
distances = [1]  # Pixel pair distance
angles = [0]  # Angle (0 degrees)
```

```
# Process the first 5 images
num_images = min(5, len(images))
plt.figure(figsize=(10, 10))

for idx in range(num_images):
    # Select the image and label
    image = images[idx]
    label = labels[idx].item()

    # Convert the image from a PyTorch tensor to a NumPy array and grayscale it
    image = image.permute(1, 2, 0).cpu().numpy()  # Convert to HxWxC format and ensure it's on CPU
    gray_image = rgb2gray(image)  # Convert RGB image to grayscale

    # Scale grayscale image to 0-255 for GLCM computation
    gray_image = (gray_image * 255).astype(np.uint8)

    # Calculate GLCM and GLCM properties
    glcm = graycomatrix(gray_image, distances=distances, angles=angles, symmetric=True, normed=True
    contrast = graycoprops(glcm, 'contrast')[0, 0]
    correlation = graycoprops(glcm, 'correlation')[0, 0]
    energy = graycoprops(glcm, 'energy')[0, 0]
    homogeneity = graycoprops(glcm, 'homogeneity')[0, 0]

    # Plot the image and GLCM properties
    plt.subplot(num_images, 2, idx * 2 + 1)
    plt.imshow(image)
    plt.title(f"Image {idx+1}: {'Benign' if label == 0 else 'Malignant'}")

    plt.subplot(num_images, 2, idx * 2 + 2)
    plt.text(0.1, 0.8, f'Contrast: {contrast:.4f}', fontsize=10)
    plt.text(0.1, 0.6, f'Correlation: {correlation:.4f}', fontsize=10)
    plt.text(0.1, 0.4, f'Energy: {energy:.4f}', fontsize=10)
    plt.text(0.1, 0.2, f'Homogeneity: {homogeneity:.4f}', fontsize=10)
    plt.axis('off')
    plt.title("GLCM Properties")

plt.tight_layout()
plt.show()
```

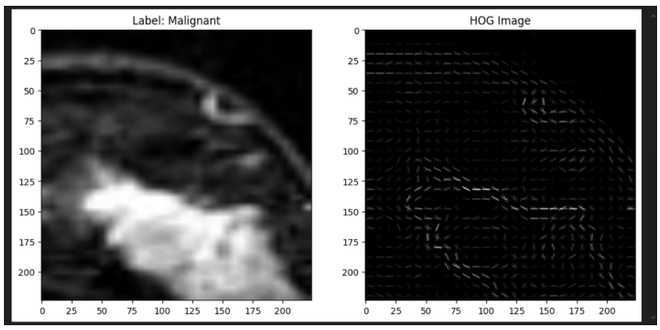



```
import matplotlib.pyplot as plt

# Get a batch of images and labels
images, labels = next(iter(train_loader))

# Adjust the index to a value within the batch size
index = min(5, len(images) - 1)  # Ensure index is within bounds

# Select the image and label
image = images[index]
```

```
label = labels[index].item()

# Convert the image from a PyTorch tensor to a NumPy array and transpose it to HxWxC
image = image.numpy().transpose((1, 2, 0))

# If the image is normalized between 0 and 1, convert it to 0-255 range
image = (image * 255).astype(int)

# Plot the original image
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title(f"Label: {'Benign' if label == 0 else 'Malignant'}")

# Plot the histogram of pixel values
plt.subplot(1, 2, 2)
plt.hist(image.ravel(), bins=256, color='orange', alpha=0.7, range=(0, 255))
plt.title("Pixel Value Histogram")
plt.xlabel("Pixel Value (0-255)")
plt.ylabel("Frequency")

plt.tight_layout()
plt.show()
```

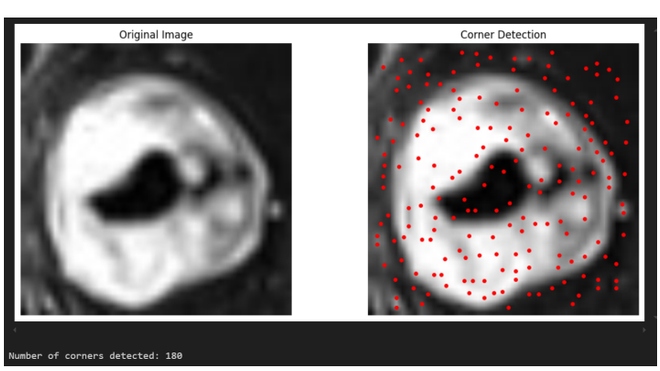



```
image, labels = next(iter(train_loader))  # Get a batch of images and labels
image = image[5]
image = image.numpy().transpose((1, 2, 0))
# Convert the image to grayscale
gray_image = color.rgb2gray(image)

plt.figure(figsize=(12, 5))
# Compute HOG features and the HOG image for visualization
hog_features, hog_image = hog(
    gray_image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    block_norm='L2-Hys',
    visualize=True,
    channel_axis=None
)

# Plot the original image and the HOG image
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title(f"Label: {'Benign' if labels[5] == 0 else 'Malignant'}")

plt.subplot(1, 2, 2)
plt.imshow(hog_image, cmap='gray')
plt.title("HOG Image")
```



Coner dectection

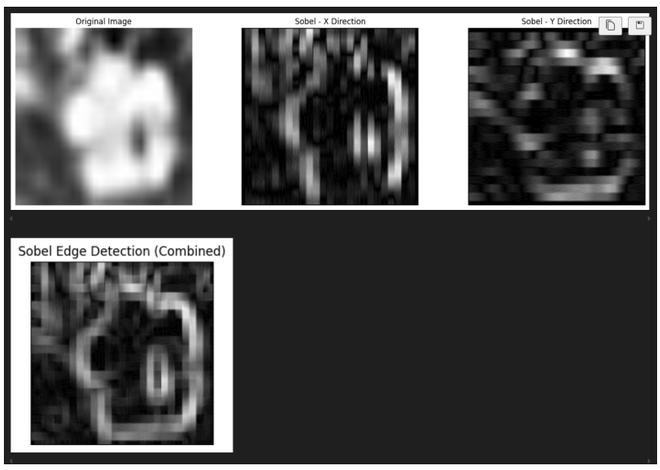
```
#coner dectection
from skimage.feature import hog, corner_harris, corner_peaks # Import corner_harris and corner_peal

image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[5]
image = image.numpy().transpose((1, 2, 0))

gray_image = color.rgb2gray(image)

# Perform Harris corner detection
corners = corner_harris(gray_image)

# Perform corner peak detection
coords = corner_peaks(corners, min_distance=5)

# Plot the original image with detected corners
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(gray_image, cmap='gray')
plt.scatter(coords[:, 1], coords[:, 0], s=15, c='red', marker='o')
plt.title('Corner Detection')
plt.axis('off')

plt.tight_layout()
plt.show()

# Display number of corners detected
print(f'Number of corners detected: {len(coords)}')
```



Sobel operator

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

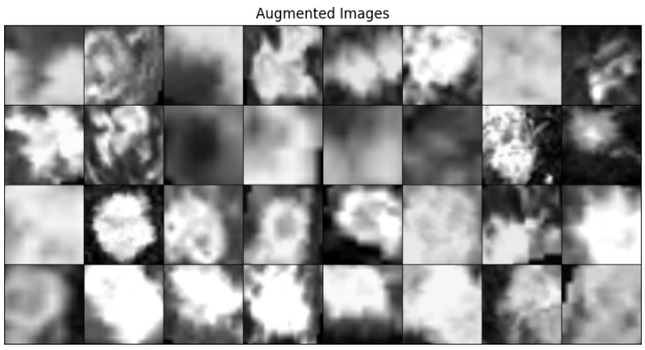
```
from skimage import color, io
from numba import jit, prange

# Define Sobel operator kernels
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

sobel_y = np.array([[-1, -2, -1],
                    [0, 0, 0],
                    [1, 2, 1]])

# Define the custom convolution function with Numba for speedup
@jit(nopython=True, parallel=True)
def convolve(x, h):
    xh, xw = x.shape
    hh, hw = h.shape
    # Kernel radius
    rh, rw = np.array(h.shape)//2
    # Init output
    output = np.zeros(x.shape)
    for n1 in prange(rh, xh-rh):
        for n2 in prange(rw, xw-rw):
            value = 0
            for k1 in prange(hh):
                for k2 in prange(hw):
                    value += h[k1, k2]*x[n1 + k1 - rh, n2 + k2 - rw]
            output[n1, n2] = value
    return output

# Load the image
image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[5]
image = image.numpy().transpose((1, 2, 0))
# Convert the image to grayscale
gray_image = color.rgb2gray(image)

# Apply the custom convolution with the Sobel operator in the x and y directions
gradient_x = convolve(gray_image, sobel_x)
gradient_y = convolve(gray_image, sobel_y)

# Combine the Sobel x and y results
sobel_combined = np.hypot(gradient_x, gradient_y)

# Normalize the combined result to range [0, 1]
sobel_combined /= sobel_combined.max()

# Plot the original image and the Sobel edge-detected images
plt.figure(figsize=(18, 5))

plt.subplot(1, 3, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(np.abs(gradient_x), cmap='gray')
plt.title('Sobel - X Direction')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(np.abs(gradient_y), cmap='gray')
plt.title('Sobel - Y Direction')
plt.axis('off')

plt.figure(figsize=(3, 3))
plt.imshow(sobel_combined, cmap='gray')
plt.title('Sobel Edge Detection (Combined)')
plt.axis('off')

plt.tight_layout()
plt.show()
```

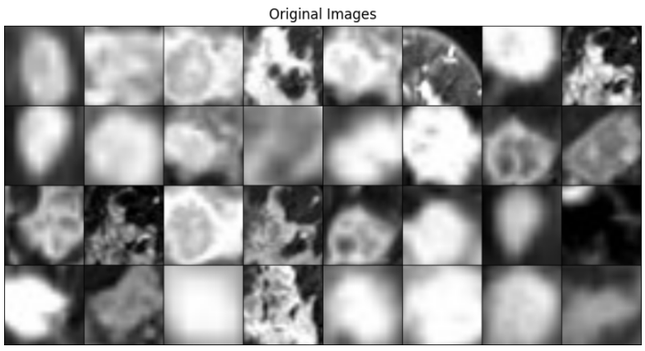


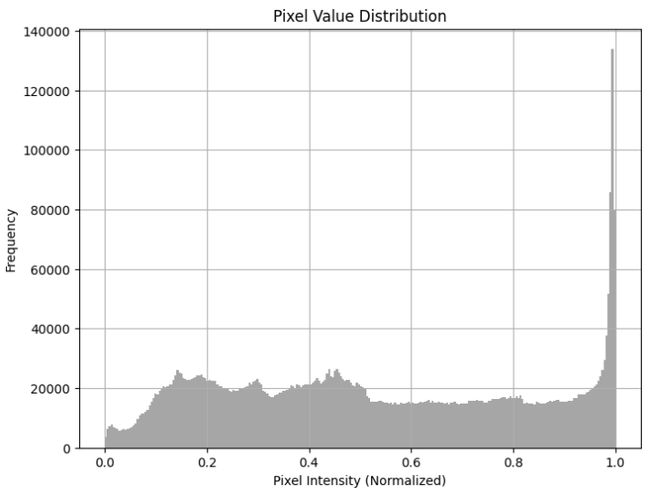
This code focuses on
dataset extraction, augmentation, and pixel intensity analysis for a classification task. It begins by extracting a ZIP file containing the dataset and organizing it into training and validation sets. Key data transformations are applied to the images to improve model performance, including random augmentations like flipping, rotation, and cropping for the training data, while the validation data undergoes standard preprocessing.

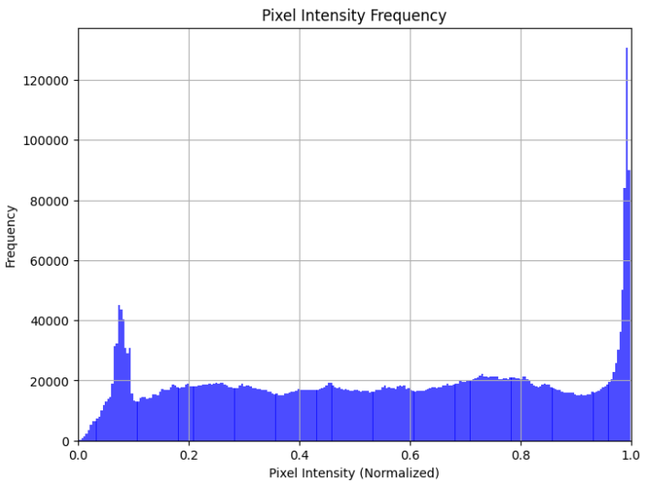
```
import os
import zipfile
import matplotlib.pyplot as plt
import numpy as np
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torchvision

# Step 1: Assume the zip file has been uploaded to Colab manually
dataset_zip = '/content/clasification-roi.zip'  # Path to the uploaded zip file
dataset_dir = '/content/clasification-roi'  # Directory where the dataset will be extracted

# Step 2: Check if the zip file exists and extract it
if os.path.exists(dataset_zip):
    print(f"Found '{dataset_zip}', extracting now...")
    with zipfile.ZipFile(dataset_zip, 'r') as zip_ref:
        zip_ref.extractall(dataset_dir)
    print("Dataset extracted successfully!")
else:
    print(f"File '{dataset_zip}' not found!")

# Step 3: List the contents of the extracted directory to inspect the structure
print("\nListing contents of the extracted directory:")
for root, dirs, files in os.walk(dataset_dir):
    print(root, "contains directories:", dirs, "and files:", files)

# Step 4: Define transformations
augmentations = {
    'train': transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.RandomRotation(20),
        transforms.Resize(256),  # Resize to a larger size before cropping
        transforms.RandomResizedCrop(224),  # Crop to 224x224
        transforms.ColorJitter(brightness=0.1, contrast=0.1),
        transforms.ToTensor(),
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),  # Resize to a larger size before cropping
        transforms.CenterCrop(224),  # Center crop to 224x224
        transforms.ToTensor(),
    ]),
}

# Step 5: Load datasets
dataset_folders = {split: datasets.ImageFolder(os.path.join(dataset_dir, split), augmentations[spli
                   for split in ['train', 'val']}
```

```
# Step 6: Initialize data loaders for both training and validation sets
batch_loaders = {split: DataLoader(dataset_folders[split], batch_size=32, shuffle=True)
                 for split in ['train', 'val']}

# Step 7: Retrieve class names from the training dataset
categories = dataset_folders['train'].classes
print(f"\nDetected Classes: {categories}")

# Step 8: Function to visualize a batch of augmented images
def visualize_augmented_images():
    inputs, classes = next(iter(batch_loaders['train']))
    out = torchvision.utils.make_grid(inputs)
    plt.figure(figsize=(10, 10))
    plt.imshow(np.transpose(out.numpy(), (1, 2, 0)))
    plt.title("Augmented Images")
    plt.axis('off')
    plt.show()

# Step 9: Function to visualize original images from the validation set
def visualize_original_images():
    inputs, classes = next(iter(batch_loaders['val']))
    out = torchvision.utils.make_grid(inputs)
    plt.figure(figsize=(10, 10))
    plt.imshow(np.transpose(out.numpy(), (1, 2, 0)))
    plt.title("Original Images")
    plt.axis('off')
    plt.show()

# Step 10: Visualize images (both augmented and original)
visualize_augmented_images()
visualize_original_images()

# Step 11: Plot pixel value histograms for the images
def plot_pixel_histogram():
    # Collect all pixel values from the validation set for histogram
    inputs, _ = next(iter(batch_loaders['val']))
    pixels = inputs.numpy().ravel()  # Flatten all pixel values

    # Plot histogram
    plt.figure(figsize=(8, 6))
    plt.hist(pixels, bins=256, range=(0, 1), color='gray', alpha=0.7)
    plt.title('Pixel Value Distribution')
    plt.xlabel('Pixel Intensity (Normalized)')
    plt.ylabel('Frequency')
    plt.grid()
    plt.show()

# Step 12: Function to plot pixel intensity frequency
def plot_pixel_frequency():
    inputs, _ = next(iter(batch_loaders['val']))
    pixels = inputs.numpy().ravel()  # Flatten all pixel values

    # Calculate frequency of pixel values
    pixel_freq, bins = np.histogram(pixels, bins=256, range=(0, 1))

    # Plot histogram with pixel intensity on the x-axis and frequency on the y-axis
    plt.figure(figsize=(8, 6))
    plt.bar(bins[:-1], pixel_freq, width=0.004, color='blue', alpha=0.7)  # Bar plot for pixel freq
    plt.title('Pixel Intensity Frequency')
    plt.xlabel('Pixel Intensity (Normalized)')
    plt.ylabel('Frequency')
    plt.grid()
    plt.xlim(0, 1)  # Set x-axis limits
    plt.show()

# Step 13: Display pixel value histogram and pixel frequency
plot_pixel_histogram()
plot_pixel_frequency()
```







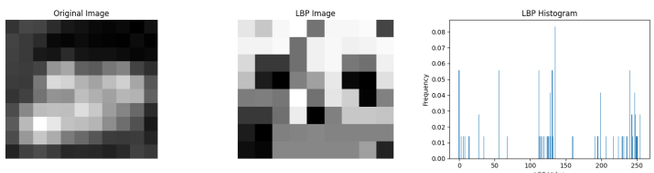


Binary Code

```
import os
import zipfile
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
# Step 1: Assume the zip file has been uploaded to Colab manually
dataset_zip = '/content/clasification-roi.zip'  # Path to the uploaded zip file
dataset_dir = '/content/clasification-roi'  # Directory where the dataset will be extracted

# Step 2: Check if the zip file exists and extract it
if os.path.exists(dataset_zip):
    print(f"Found '{dataset_zip}', extracting now...")
    with zipfile.ZipFile(dataset_zip, 'r') as zip_ref:
        zip_ref.extractall(dataset_dir)
    print("Dataset extracted successfully!")
else:
    print(f"File '{dataset_zip}' not found!")

# Step 3: Define LBP functions
def get_pixel(img, center, x, y):
    """
    Get pixel value at coordinate (x,y) or return 0 if it's outside the image
    """
    new_value = 0
    try:
        if img[x][y] >= center:
            new_value = 1
    except IndexError:
        pass
    return new_value

def calculate_lbp(image, radius=3):
    """
    Calculate the Local Binary Pattern of an image
    """
    height, width = image.shape
    lbp_image = np.zeros((height-2, width-2), np.uint8)
    neighbors = radius*2

    # Iterate through the image pixels (excluding borders)
    for i in range(1, height-1):
        for j in range(1, width-1):
            center = image[i, j]
            binary_code = []

            # Compare with 8 neighboring pixels
            binary_code.append(get_pixel(image, center, i-1, j-1))  # Top-left
            binary_code.append(get_pixel(image, center, i-1, j))    # Top
            binary_code.append(get_pixel(image, center, i-1, j+1))  # Top-right
            binary_code.append(get_pixel(image, center, i, j+1))    # Right
            binary_code.append(get_pixel(image, center, i+1, j+1))  # Bottom-right
            binary_code.append(get_pixel(image, center, i+1, j))    # Bottom
            binary_code.append(get_pixel(image, center, i+1, j-1))  # Bottom-left
            binary_code.append(get_pixel(image, center, i, j-1))    # Left

            # Convert binary code to decimal
            lbp_value = 0
            for idx, value in enumerate(binary_code):
                lbp_value += value * (2 ** idx)

            lbp_image[i-1, j-1] = lbp_value

    # Calculate the histogram
    histogram = np.histogram(lbp_image.ravel(), bins=np.arange(0, 257))[0]

    # Normalize the histogram
    histogram = histogram.astype('float32')
    histogram /= (histogram.sum() + 1e-7)

    return lbp_image, histogram

def visualize_lbp_results(original_image, lbp_image, histogram):
    """
    Visualize the original image, LBP image, and LBP histogram
    """
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 4))

    # Plot original image
```

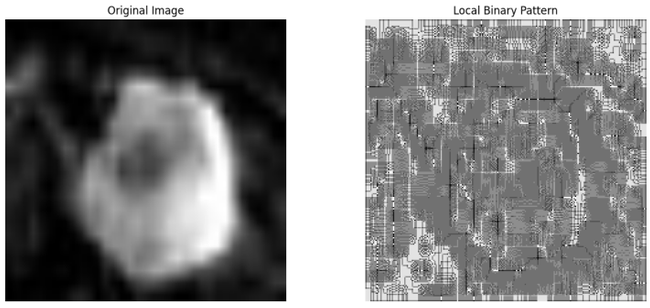
```
    ax1.imshow(original_image, cmap='gray')
    ax1.set_title('Original Image')
    ax1.axis('off')

    # Plot LBP image
    ax2.imshow(lbp_image, cmap='gray')
    ax2.set_title('LBP Image')
    ax2.axis('off')

    # Plot histogram
    ax3.bar(range(256), histogram)
    ax3.set_title('LBP Histogram')
    ax3.set_xlabel('LBP Value')
    ax3.set_ylabel('Frequency')

    plt.tight_layout()
    plt.show()

# Step 4: Process and visualize only one image from the dataset directory
def process_one_image_in_directory(directory):
    """
    Process and visualize only one image from the directory through the LBP algorithm
    """
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(('jpg', 'png', 'jpeg')):
                image_path = os.path.join(root, file)
                print(f"Processing: {image_path}")

                # Read the image
                image = cv2.imread(image_path)
                if len(image.shape) == 3:
                    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

                # Calculate LBP
                lbp_image, histogram = calculate_lbp(image)

                # Visualize the result
                visualize_lbp_results(image, lbp_image, histogram)

                # Only process the first image
                return
# Example usage
if __name__ == "__main__":
    process_one_image_in_directory(dataset_dir)
```



Local Binary Pattern

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import color
from skimage.feature import local_binary_pattern
import torch

# Parameters for LBP
radius = 1  # Radius of the circle
n_points = 8 * radius  # Number of points considered for the LBP

# Get a batch of images and labels
images, labels = next(iter(train_loader))

# Choose an image from the batch (assuming batch size is large enough)
image = images[5]  # Choose the 5th image
```

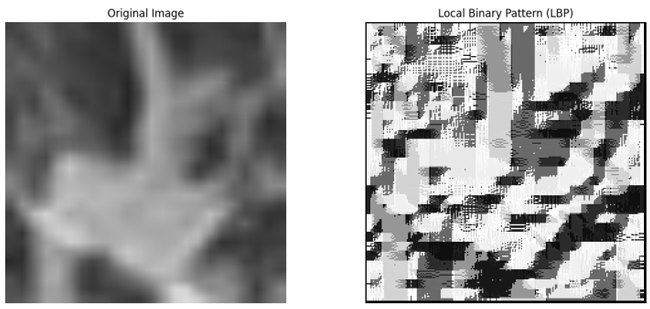
```
# Convert the image from PyTorch tensor to NumPy array and move it to CPU if needed
image = image.permute(1, 2, 0).cpu().numpy()

# Convert the image to grayscale
gray_image = color.rgb2gray(image)

# Apply Local Binary Pattern
lbp_image = local_binary_pattern(gray_image, n_points, radius, method='uniform')

# Plot the original image and the LBP image
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(lbp_image, cmap='gray')
plt.title('Local Binary Pattern')
plt.axis('off')

plt.tight_layout()
plt.show()
```



Local Binary Pattern

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

def lbp_scratch(image):
    """
    Computes the Local Binary Pattern (LBP) for an input image from scratch.
    """
    # Step 0: Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    imgLBP = np.zeros_like(gray_image)  # Initialize the LBP image

    # Define the size of the neighborhood (3x3 window)
    neighboor = 3

    # Iterate over each pixel in the grayscale image, excluding the border pixels
    for ih in range(0, gray_image.shape[0] - neighboor):
        for iw in range(0, gray_image.shape[1] - neighboor):
            # Step 1: Extract a 3x3 region centered on the current pixel
            img = gray_image[ih:ih + neighboor, iw:iw + neighboor]
            center = img[1, 1]  # The center pixel value of the 3x3 region

            # Step 2: Binary operation - Compare each neighbor's value with the center pixel
            img01 = (img >= center) * 1.0  # Threshold the 3x3 region

            # Step 3: Flatten the thresholded image into a vector, excluding the center pixel
            img01_vector = img01.flatten()
            img01_vector = np.delete(img01_vector, 4)  # Remove the center pixel

            # Step 4: Convert the binary pattern to a decimal number
            where_img01_vector = np.where(img01_vector)[0]  # Indices where the value is 1
            if len(where_img01_vector) >= 1:
                num = np.sum(2 ** where_img01_vector)  # Convert binary to decimal
```

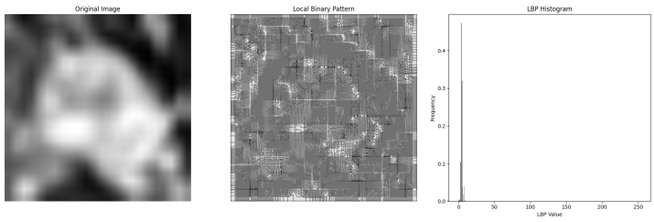
```
            else:
                num = 0  # If all surrounding pixels are smaller, assign 0

            # Step 5: Assign the LBP value to the central pixel of the output image
            imgLBP[ih + 1, iw + 1] = num

    return imgLBP

# Load an example image (substitute with your image)
image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[4]
image = image.numpy().transpose((1, 2, 0))


# Compute LBP using the custom function
lbp_image = lbp_scratch(image)

# Display the original and LBP images
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(lbp_image, cmap='gray')
plt.title('Local Binary Pattern (LBP)')
plt.axis('off')

plt.tight_layout()
plt.show()
```



histogram of LBP

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import color, io
from skimage.feature import local_binary_pattern

# Parameters for LBP
radius = 1  # Radius of the circle
n_points = 8 * radius  # Number of points considered for the LBP

# Load the image
# Load the image
image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[5]
image = image.numpy().transpose((1, 2, 0))
# Convert the image to grayscale
gray_image = color.rgb2gray(image)

# Apply Local Binary Pattern
lbp_image = local_binary_pattern(gray_image, n_points, radius, method='uniform')


# Compute the histogram of LBP with values ranging from 0 to 255
```

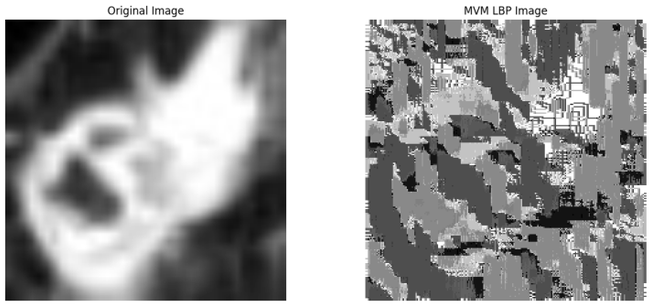
```
n_bins = 256  # Number of bins for the histogram (0 to 255)
hist, bins = np.histogram(lbp_image.ravel(), bins=n_bins, range=(0, 255), density=True)
# Compute the histogram of LBP
#n_bins = int(lbp_image.max() + 1)  # Number of bins in the histogram
#hist, bins = np.histogram(lbp_image.ravel(), bins=n_bins, range=(0, n_bins), density=True)


# Plot the original image, LBP image, and the LBP histogram
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(lbp_image, cmap='gray')
plt.title('Local Binary Pattern')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.bar(bins[:-1], hist, width=0.8, color='gray')
plt.title('LBP Histogram')
plt.xlabel('LBP Value')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



MVM LBP IMAGES

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import color
from skimage.feature import local_binary_pattern
from scipy.ndimage import generic_filter

radius = 1  # Radius of the circle
n_points = 8 * radius  # Number of points considered for the LBP

# Load the image
image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[5]
image = image.numpy().transpose((1, 2, 0))

# Convert the image to grayscale
gray_image = color.rgb2gray(image)

# Function to compute the mean, variance, and median-based LBP
def mvm_lbp(pixel_values):
    central_pixel = pixel_values[len(pixel_values) // 2]
    neighbors = pixel_values[:len(pixel_values) // 2]
    mean_val = np.mean(neighbors)
    variance_val = np.var(neighbors)
    median_val = np.median(neighbors)

    # Compute the LBP using mean, variance, and median as thresholds
    lbp_mean = np.sum((neighbors >= mean_val).astype(int) * (1 << np.arange(len(neighbors))))
    lbp_variance = np.sum((neighbors >= variance_val).astype(int) * (1 << np.arange(len(neighbors)))
    lbp_median = np.sum((neighbors >= median_val).astype(int) * (1 << np.arange(len(neighbors))))
```

```
    # Combine the results by averaging (alternative approaches can be used)
    combined_lbp = (lbp_mean + lbp_variance + lbp_median) / 3
    return combined_lbp

# Define the size of the filter (3x3 for radius=1, 5x5 for radius=2, etc.)
filter_size = (2 * radius + 1, 2 * radius + 1)

# Apply the MVM LBP filter to the image
mvm_lbp_image = generic_filter(gray_image, mvm_lbp, size=filter_size)

# Plot the original image and the MVM LBP image
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(mvm_lbp_image, cmap='gray')
plt.title('MVM LBP Image')
plt.axis('off')

plt.tight_layout()
plt.show()
```

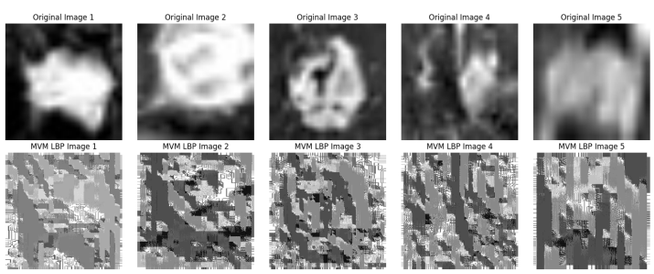


MVM LBP

MVM-LBP, or
Multi-View Multi-Scale Local Binary Patterns, is an advanced texture analysis technique used in image processing and computer vision. It extends the traditional Local Binary Pattern (LBP) method by incorporating multi-view and multi-scale features, allowing for more robust and detailed texture representation.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import color
from scipy.ndimage import generic_filter

radius = 1  # Radius of the circle
n_points = 8 * radius  # Number of points considered for the LBP

# Load a batch of 5 images
images, labels = next(iter(train_loader))  # Get a batch of images and labels

# Function to compute the mean, variance, and median-based LBP
def mvm_lbp(pixel_values):
    neighbors = pixel_values[:len(pixel_values) // 2]
    mean_val = np.mean(neighbors)
    variance_val = np.var(neighbors)
    median_val = np.median(neighbors)

    # Compute the LBP using mean, variance, and median as thresholds
    lbp_mean = np.sum((neighbors >= mean_val).astype(int) * (1 << np.arange(len(neighbors))))
    lbp_variance = np.sum((neighbors >= variance_val).astype(int) * (1 << np.arange(len(neighbors)))
    lbp_median = np.sum((neighbors >= median_val).astype(int) * (1 << np.arange(len(neighbors))))
```

```
    # Combine the results by averaging
    combined_lbp = (lbp_mean + lbp_variance + lbp_median) / 3
    return combined_lbp

# Define the size of the filter (3x3 for radius=1, 5x5 for radius=2, etc.)
filter_size = (2 * radius + 1, 2 * radius + 1)

# Set up the plot for 5 images
plt.figure(figsize=(15, 6))

for i in range(5):
    # Select the ith image from the batch and process it
    image = images[i].numpy().transpose((1, 2, 0))  # Convert to HxWxC format
    gray_image = color.rgb2gray(image)  # Convert to grayscale

    # Apply the MVM LBP filter to the grayscale image
    mvm_lbp_image = generic_filter(gray_image, mvm_lbp, size=filter_size)

    # Plot the original and the MVM LBP image side by side
    plt.subplot(2, 5, i + 1)
    plt.imshow(gray_image, cmap='gray')
    plt.title(f'Original Image {i+1}')
    plt.axis('off')

    plt.subplot(2, 5, i + 6)
    plt.imshow(mvm_lbp_image, cmap='gray')
    plt.title(f'MVM LBP Image {i+1}')
    plt.axis('off')

plt.tight_layout()
plt.show()
```



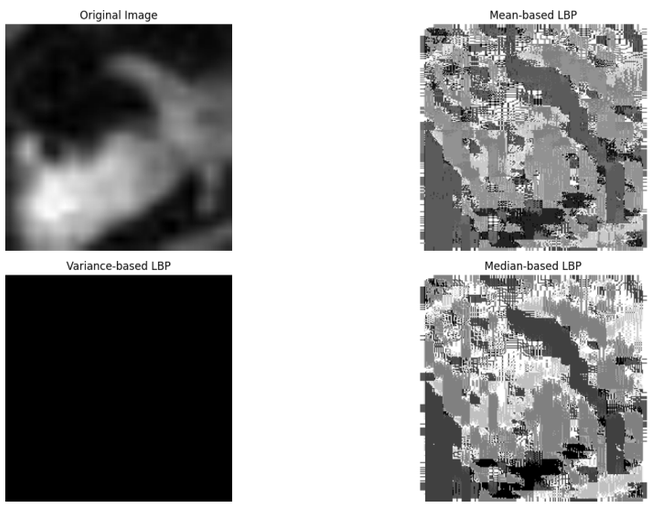
Introduction to Mean LBP (Local Binary Pattern)

Mean Local Binary Pattern (LBP) is a variation of the traditional **Local Binary Pattern** (LBP) used for texture analysis in image processing. LBP is a simple yet powerful method for describing the texture of an image by comparing each pixel with its surrounding neighbors, assigning a binary value (0 or 1) based on whether the neighboring pixel's intensity is greater or smaller than the central pixel. This results in a binary code for each pixel, which can be used to characterize the texture of an image.

Mean LBP specifically refers to computing the **mean** of the binary patterns derived from neighboring pixels over a local region, offering a more averaged and global representation of the texture. This approach helps to reduce sensitivity to noise and small variations, making it more robust for applications in image classification, especially in medical fields like breast cancer detection, where subtle texture differences are crucial for distinguishing between benign and malignant tumors.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import color
from scipy.ndimage import generic_filter

radius = 1  # Radius of the circle
n_points = 8 * radius  # Number of points considered for the LBP

# Load the image
image, labels = next(iter(train_loader))  # Get a batch of images and labels

image = image[5]
image = image.numpy().transpose((1, 2, 0))

# Convert the image to grayscale
gray_image = color.rgb2gray(image)

# Function to compute Mean-based LBP
def mean_based_lbp(pixel_values):
```

```
    neighbors = pixel_values[:len(pixel_values) // 2]
    mean_val = np.mean(neighbors)
    lbp_mean = np.sum((neighbors >= mean_val).astype(int) * (1 << np.arange(len(neighbors))))
    return lbp_mean

# Function to compute Variance-based LBP
def variance_based_lbp(pixel_values):
    neighbors = pixel_values[:len(pixel_values) // 2]
    variance_val = np.var(neighbors)
    lbp_variance = np.sum((neighbors >= variance_val).astype(int) * (1 << np.arange(len(neighbors))
    return lbp_variance

# Function to compute Median-based LBP
def median_based_lbp(pixel_values):
    neighbors = pixel_values[:len(pixel_values) // 2]
    median_val = np.median(neighbors)
    lbp_median = np.sum((neighbors >= median_val).astype(int) * (1 << np.arange(len(neighbors))))
    return lbp_median

# Define the size of the filter (3x3 for radius=1, 5x5 for radius=2, etc.)
filter_size = (2 * radius + 1, 2 * radius + 1)

# Apply the Mean, Variance, and Median LBP filters to the image
mean_lbp_image = generic_filter(gray_image, mean_based_lbp, size=filter_size)
variance_lbp_image = generic_filter(gray_image, variance_based_lbp, size=filter_size)
median_lbp_image = generic_filter(gray_image, median_based_lbp, size=filter_size)

# Plot the original image and the Mean, Variance, and Median LBP images
plt.figure(figsize=(16, 8))

plt.subplot(2, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 2, 2)
plt.imshow(mean_lbp_image, cmap='gray')
plt.title('Mean-based LBP')
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(variance_lbp_image, cmap='gray')
plt.title('Variance-based LBP')
plt.axis('off')

plt.subplot(2, 2, 4)
plt.imshow(median_lbp_image, cmap='gray')
plt.title('Median-based LBP')
plt.axis('off')

plt.tight_layout()
plt.show()
```



GLMC for texture analysis

```
test_transforms = transforms.Compose([
    transforms.Resize(224),                    # Resize the image to 256x256
    transforms.CenterCrop(224),                # Center crop to 224x224
    transforms.ToTensor(),                     # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Normalize with Ir
])
```

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

```
from torch.utils.data import Dataset
from PIL import Image
import os
class BreastCancerDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_paths = [os.path.join(root_dir, file) for file in os.listdir(root_dir)]

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

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = Image.open(image_path).convert("RGB")  # Ensure 3-channel RGB image
        if self.transform:
            image = self.transform(image)
        return image
```

Training the Model

Loading Vgg16

```
import torch

import torchvision.models as models

vgg16 = models.vgg16(pretrained=True)

vgg16.eval()

print(vgg16)
```

MODEL Training

```
import torch
import torch.nn as nn
import torchvision.models as models

class customVGG16(nn.Module):
    def __init__(self, num_classes=2):
        super(customVGG16, self).__init__()

        # Load the pre-trained VGG16 model
        vgg16 = models.vgg16(pretrained=True)

        # Extract features and avgpool layers
        self.features = vgg16.features
        self.avgpool = vgg16.avgpool
```

```
        # Define a new classifier
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),  # Linear layer with input size 512 7 7 and output size 4
            nn.ReLU(),                    # ReLU activation function
            nn.Dropout(),                 # Dropout
            nn.Linear(4096, 4096),         # Another linear layer with input size 4096 and output s
            nn.ReLU(),                    # ReLU activation
            nn.Dropout(),                 # Dropout layer
            nn.Linear(4096, num_classes)  # Final Linear layer with output size equal to number of
        )
    # Forward Method (Make sure this is outside the __init__ method)
    def forward(self, x):
        # Pass input through the features layer
        x = self.features(x)
        # Pass through the AVGpool layer
        x = self.avgpool(x)
        # Reshape output to a 2D tensor
        x = torch.flatten(x, 1)
        # Pass through the classifier
        x = self.classifier(x)
        return x
```

```
model = customVGG16()
print(model)
```

Train Function

```
import torch
from tqdm import tqdm
import torch.nn.functional as F
def train(epoch, model, num_epochs, loader, criterion, l2_decay):
    learing_rate = max(lr*(0.1**(epoch//10)),1e-5)
    optimizer = torch.optim.SGD(model.parameters(), lr= learing_rate, momentum=0.9, weight_decay=l2
    model.train()
    correct = 0
    for data, label in tqdm(loader, desc=f'Epoch {epoch+1}/{num_epochs}', unit='batch'):
        data = data.float().cuda()

        label = label.long().cuda()

        output = model(data)
        optimizer.zero_grad()
        loss = F.nll_loss(F.log_softmax(output, dim=1), label)
        loss.backward()
        optimizer.step()

        pred = output.data.max(1)[1]
        correct += pred.eq(label.data.view_as(pred)).cpu().sum()

    print(f'train accuracy: {100. * correct / len(loader.dataset)}%')
```

Function to evaluates a trained PyTorch model on a test dataset and calculates key metrics like loss, accuracy, and probabilities for further analysis.

```
import torch
import torch.nn.functional as F
import numpy as np
from sklearn.metrics import roc_auc_score, roc_curve

def test(model, test_dataloader):
    name = 'test'  # test dataset identifier
    len_test_dataloader = len(test_dataloader.dataset)  # length of test dataset
    model.eval()
    test_loss = 0
    correct = 0
    possibilities = None
    all_predictions = []
    labels = ['benign', 'malignant']  # class labels
```

```
    for data, target in test_dataloader:  # use test_dataloader instead of val_dataloader
        if torch.cuda.is_available():
            data, target = data.cuda(), target.cuda()

        test_output = model(data)  # changing variable name to test_output
        test_loss += F.nll_loss(F.log_softmax(test_output, dim=1), target, reduction='sum').item()

        pred = test_output.data.max(1)[1]  # get the index of the max log-probability
        all_predictions.append(pred.cpu().numpy())

        possibility = F.softmax(test_output, dim=1).cpu().data.numpy()  # get probabilities
        if possibilities is None:
            possibilities = possibility
        else:
            possibilities = np.concatenate((possibilities, possibility), axis=0)

        correct += pred.eq(target.data.view_as(pred)).cpu().sum()  # count correct predictions

    # Flatten the list of predictions
    all_predictions = [i for item in all_predictions for i in item]

    # Return the accuracy and loss
    accuracy = correct / len_test_dataloader
    avg_loss = test_loss / len_test_dataloader
    return avg_loss, accuracy, all_predictions, possibilities
```

Validation function

Evaluate the model on unseen data (validation set).
Compute key metrics:
Loss: Average loss across all validation samples.
Accuracy: Percentage of correctly predicted samples.
AUC (Area Under the Curve): A performance metric that shows how well the model distinguishes between classes.
Specificity and Sensitivity: Quantify the true negative rate and true positive rate, respectively.
Confusion Matrix: A matrix showing the true positives, false positives, true negatives, and false negative

```
from sklearn import metrics
from sklearn.metrics import roc_auc_score, roc_curve, auc
import torch
import torch.nn.functional as F
import numpy as np

def validation(model, val_loader):
    name = 'validation'
    len_val_dataloader = len(val_loader.dataset)
    model.eval()  # Set model to evaluation mode
    test_loss = 0
    correct = 0
    all_predictions = []  # Corrected typo: 'all_prediction' to 'all_predictions'
    possibilities = None  # Corrected typo: 'possibiltys' to 'possibilities'

    for data, target in val_loader:
        if torch.cuda.is_available():
            data, target = data.cuda(), target.cuda()

        val_output = model(data)

        # Calculate test loss
        test_loss += F.nll_loss(F.log_softmax(val_output, dim=1), target, reduction='sum').item()

        # Get predictions
        pred = val_output.data.max(1)[1]
        all_predictions.append(pred.cpu().numpy())  # Collect all predictions

        # Calculate probabilities for AUC
        possibility = F.softmax(val_output, dim=1).cpu().detach().numpy()
        if possibilities is None:
            possibilities = possibility
        else:
            possibilities = np.concatenate((possibilities, possibility), axis=0)
```

```
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    # Flatten predictions
    all_predictions = [i for item in all_predictions for i in item]

    # Confusion matrix
    cm = metrics.confusion_matrix(target.cpu().numpy(), all_predictions)

    # One-hot encode the labels for AUC computation
    num_classes = val_output.shape[1]
    label_onehot = np.eye(num_classes)[np.array(target.cpu().numpy()).astype(int).tolist()]

    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(label_onehot.ravel(), possibilities.ravel())
    auc_value = auc(fpr, tpr)

    # Test loss
    test_loss /= len_val_dataloader  # Average loss per sample

    print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(1 - fpr[0], tpr[0], auc_va
    print('\nTest set: Average loss: {:.4f}, Accuracy: {:.2f}%\n'.format(test_loss, 100. * correct

    return test_loss, 100. * correct / len_val_dataloader, test_loss, auc_value
```

```
total_epochs = 50
lr = 0.01
momentum = 0.9
no_cuda = False
num_classes=2
log_interval = 10
l2_decay = 0.01
model = customVGG16(num_classes=num_classes)
model = model.to(device)
customVGG16(num_classes=num_classes)
```

comprehensive training loop for a PyTorch model, incorporating features like weight decay (L2 regularization), validation, model checkpointing, and early stopping. Here's a breakdown of its functionality:

Early Stopping function

```
import numpy as np
import torch

class EarlyStopping:
    def __init__(self, patience=7, verbose=False, delta=0, path='checkpoint.pt', trace_func=print)
        """
        Args:
            patience (int): How long to wait after last time validation loss improved.
                            Default: 7
            verbose (bool): If True, prints a message for each validation loss improvement.
                            Default: False
            delta (float): Minimum change in the monitored quantity to qualify as an improvement.
                            Default: 0
            path (str): Path for the checkpoint to save the best model.
                            Default: 'checkpoint.pt'
            trace_func (function): Function to print trace messages.
                            Default: print
        """
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf  # Set to infinity to ensure the first comparison
        self.delta = delta
        self.path = path
        self.trace_func = trace_func

    def __call__(self, val_loss, model):
        """Check if early stopping condition is met and save the model if there's improvement."""
        score = -val_loss
```

```
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            self.trace_func(f"EarlyStopping counter: {self.counter} out of {self.patience}")
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        """Saves model when validation loss decreases."""
        if self.verbose:
            self.trace_func(f"Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}
        torch.save(model.state_dict(), self.path)
        self.val_loss_min = val_loss
```

****script sets up a data pipeline for training a deep learning model in PyTorch using the torchvision library and a dataset stored in Google Drive.****

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')

# Define transformations for data augmentation and normalization
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resizing to match VGG input size
    transforms.ToTensor(),          # Convert to Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # VGG normalizatio
])

# Load training dataset from Google Drive
train_dataset = datasets.ImageFolder(root='/content/drive/MyDrive/BCD-data/train', transform=transf

# Set batch size
batch_size = 32

# Initialize DataLoader for training
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

VGG16 Moddel

```
class CustomVGG16(nn.Module):
    def __init__(self, num_classes=2):
        super(CustomVGG16, self).__init__()
        # Load the pre-trained VGG16 model
        vgg16 = models.vgg16(pretrained=True)
        # Extract the features and avgpool layers
        self.features = vgg16.features
        self.avgpool = vgg16.avgpool
        # Define a new classifier
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096), # Adjust input size for VGG16's avgpool output
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes)
        )
    def forward(self, x):
```

```
        # Pass through feature layers
        x = self.features(x)
        # Pass through avgpool layer
        x = self.avgpool(x)
        # Reshape the output to a 2D tensor (batch_size, 512*7*7)
        x = torch.flatten(x, 1)  # Flatten from (batch_size, 512, 7, 7) to (batch_size, 512*7*7)
        # Pass through the custom classifier
        x = self.classifier(x)
        return x
```

```
model=CustomVGG16(num_classes=2)
print(model)
```

Train Data set using pretrained Vgg16 Model

```
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

# Image dimensions and data directories
img_size = 224
train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

# Data transformations for augmentation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Load datasets
train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

def create_vgg16_model(num_classes):
    model = models.vgg16(pretrained=True)

    # Freeze the convolutional layers
    for param in model.features.parameters():
        param.requires_grad = False

    # Modify the classifier to match the number of classes
    model.classifier[6] = nn.Linear(model.classifier[6].in_features, num_classes)

    return model
```

```
def train_vgg16(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, patie
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch + 1}/{num_epochs}")

        # 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)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

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

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects.double() / len(train_loader.dataset)

        print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_running_corrects = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Track statistics
                val_running_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_running_corrects += torch.sum(preds == labels.data)

        val_loss = val_running_loss / len(val_loader.dataset)
        val_acc = val_running_corrects.double() / len(val_loader.dataset)

        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

        # Save the model if it has the best accuracy so far
        if val_acc > best_accuracy:
            best_accuracy = val_acc
            best_model_wts = model.state_dict()
            early_stop_counter = 0  # Reset counter when validation improves
        else:
            early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break
```

```
    # Load the best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'best_vgg16_model.pth')
    print(f"Best Validation Accuracy: {best_accuracy:.4f}")

def test_vgg16(model, test_loader, device):
    model.eval()
    test_corrects = 0
    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)
            test_corrects += torch.sum(preds == labels.data)

    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")

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

# Create model, define criterion and optimizer
num_classes = len(train_dataset.classes)
model = create_vgg16_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)

# Train the model with early stopping
train_vgg16(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patience=

# Load the best model for testing
model.load_state_dict(torch.load('best_vgg16_model.pth'))

# Test the model
test_vgg16(model, test_loader, device)
```

Train dataset using pretrained ResNet50 model

```
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

# Image dimensions and data directories
img_size = 224
train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

# Data transformations for augmentation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
```

```
}

# Load datasets
train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Function to create ResNet-50 model
def create_resnet50_model(num_classes):
    model = models.resnet50(pretrained=True)

    # Freeze the convolutional layers
    for param in model.parameters():
        param.requires_grad = False

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

    return model

# Training function for ResNet-50
def train_model(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, patie
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch + 1}/{num_epochs}")

        # 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)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

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

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects.double() / len(train_loader.dataset)

        print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_running_corrects = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
```

```
                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Track statistics
                val_running_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_running_corrects += torch.sum(preds == labels.data)

        val_loss = val_running_loss / len(val_loader.dataset)
        val_acc = val_running_corrects.double() / len(val_loader.dataset)

        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

        # Save the model if it has the best accuracy so far
        if val_acc > best_accuracy:
            best_accuracy = val_acc
            best_model_wts = model.state_dict()
            early_stop_counter = 0  # Reset counter when validation improves
        else:
            early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break

    # Load the best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'best_resnet50_model.pth')
    print(f"Best Validation Accuracy: {best_accuracy:.4f}")

# Test the model
def test_model(model, test_loader, device):
    model.eval()
    test_corrects = 0
    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)
            test_corrects += torch.sum(preds == labels.data)

    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")

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

# Create ResNet-50 model, define criterion and optimizer
num_classes = len(train_dataset.classes)
model = create_resnet50_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)

# Train the model with early stopping
train_model(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patience=
# Load the best model for testing
model.load_state_dict(torch.load('best_resnet50_model.pth'))

# Test the model
test_model(model, test_loader, device)
```

Train dataset using pretrained ResNet18 model

```
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
```

```
# Image dimensions and data directories
img_size = 224a
train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

# Data transformations for augmentation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Load datasets
train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Function to create ResNet-18 model
def create_resnet18_model(num_classes):
    model = models.resnet18(pretrained=True)

    # Freeze the convolutional layers
    for param in model.parameters():
        param.requires_grad = False

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

    return model

# Training function for ResNet-18
def train_model(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, patie
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch + 1}/{num_epochs}")

        # 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)

            # Zero the parameter gradients
            optimizer.zero_grad()
```

```
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

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

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects.double() / len(train_loader.dataset)

        print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_running_corrects = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Track statistics
                val_running_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_running_corrects += torch.sum(preds == labels.data)

        val_loss = val_running_loss / len(val_loader.dataset)
        val_acc = val_running_corrects.double() / len(val_loader.dataset)

        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

        # Save the model if it has the best accuracy so far
        if val_acc > best_accuracy:
            best_accuracy = val_acc
            best_model_wts = model.state_dict()
            early_stop_counter = 0  # Reset counter when validation improves
        else:
            early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break

    # Load the best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'best_resnet18_model.pth')
    print(f"Best Validation Accuracy: {best_accuracy:.4f}")

# Test the model
def test_model(model, test_loader, device):
    model.eval()
    test_corrects = 0
    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)
            test_corrects += torch.sum(preds == labels.data)

    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")
```

```
# Set device (use GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Create ResNet-18 model, define criterion and optimizer
num_classes = len(train_dataset.classes)
model = create_resnet18_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)

# Train the model with early stopping
train_model(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patience=

# Load the best model for testing
model.load_state_dict(torch.load('best_resnet18_model.pth'))

# Test the model
test_model(model, test_loader, device)
```

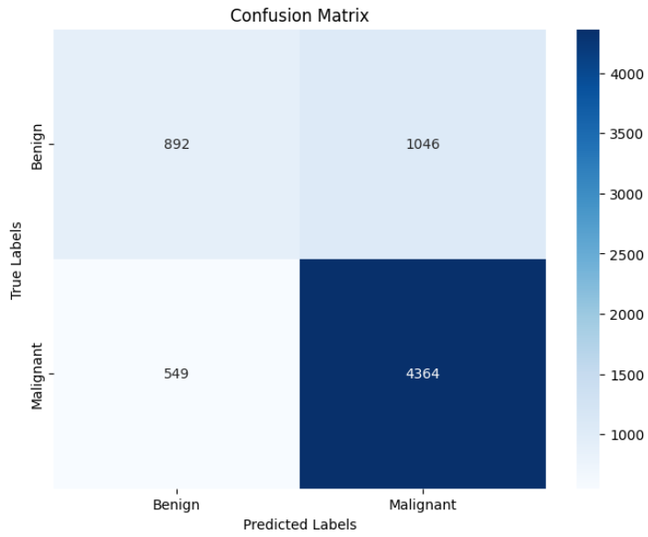
VGG16 (AUC,CONFUSION_MATRIX)

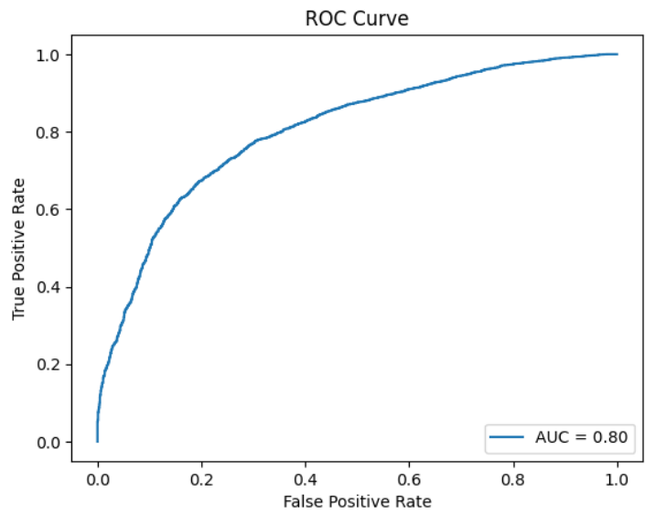
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve
import seaborn as sns

# Helper functions
def plot_auc(y_true, y_scores):
    """Plot the ROC AUC curve."""
    auc_score = roc_auc_score(y_true, y_scores)
    print(f"ROC AUC Score: {auc_score:.4f}")
    fpr, tpr, _ = roc_curve(y_true, y_scores)
    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes):
    """Plot the confusion matrix."""
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Model creation function
def create_vgg16_model(num_classes):
    model = models.vgg16(pretrained=True)

    # Freeze the convolutional layers
    for param in model.features.parameters():
        param.requires_grad = False

    # Modify the classifier to match the number of classes
    model.classifier[6] = nn.Linear(model.classifier[6].in_features, num_classes)

    return model

# Training function
def train_vgg16(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, patie
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0
```

```
for epoch in range(num_epochs):
    print(f"Epoch {epoch + 1}/{num_epochs}")

    # 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)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

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

    epoch_loss = running_loss / len(train_loader.dataset)
    epoch_acc = running_corrects.double() / len(train_loader.dataset)

    print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

    # Validation phase
    model.eval()
    val_running_loss = 0.0
    val_running_corrects = 0

    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Track statistics
            val_running_loss += loss.item() * inputs.size(0)
            _, preds = torch.max(outputs, 1)
            val_running_corrects += torch.sum(preds == labels.data)

    val_loss = val_running_loss / len(val_loader.dataset)
    val_acc = val_running_corrects.double() / len(val_loader.dataset)

    print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

    # Save the model if it has the best accuracy so far
    if val_acc > best_accuracy:
        best_accuracy = val_acc
        best_model_wts = model.state_dict()
        early_stop_counter = 0  # Reset counter when validation improves
    else:
        early_stop_counter += 1

    if early_stop_counter >= patience:
        print("Early stopping triggered.")
        break

# Load the best model weights
model.load_state_dict(best_model_wts)
torch.save(model.state_dict(), 'best_vgg16_model.pth')
print(f"Best Validation Accuracy: {best_accuracy:.4f}")
```

```
# Testing function
def test_vgg16(model, test_loader, device, class_names):
    """Test the model and plot metrics."""
    model.eval()
    test_corrects = 0
    all_labels = []
    all_preds = []
    all_probs = []

    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)
            probs = torch.softmax(outputs, dim=1)[:, 1]  # Assuming binary classification

            test_corrects += torch.sum(preds == labels.data)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
            all_probs.extend(probs.cpu().numpy())

    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")

    # Plot the confusion matrix
    plot_confusion_matrix(all_labels, all_preds, class_names)

    # Plot the ROC AUC curve
    plot_auc(all_labels, all_probs)

# Main workflow
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
img_size = 224

train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

num_classes = len(train_dataset.classes)
class_names = train_dataset.classes
model = create_vgg16_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)
```

```
train_vgg16(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patience=
model.load_state_dict(torch.load('best_vgg16_model.pth'))
test_vgg16(model, test_loader, device, class_names)
```





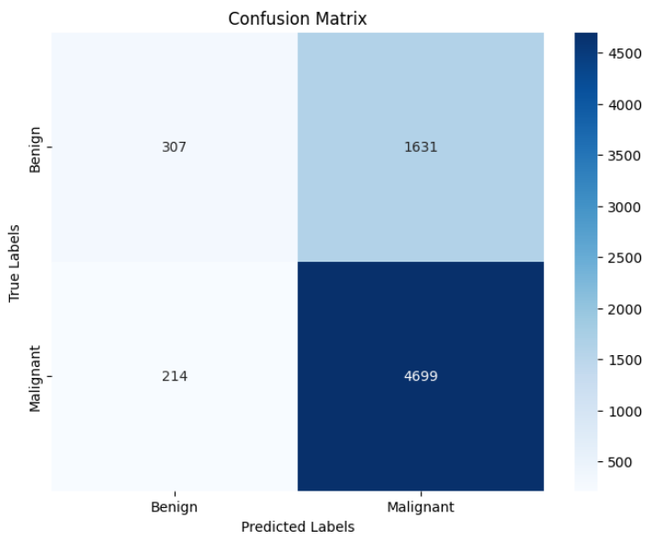
Resnet50 (AUC,CONFUSION_MATRIX)

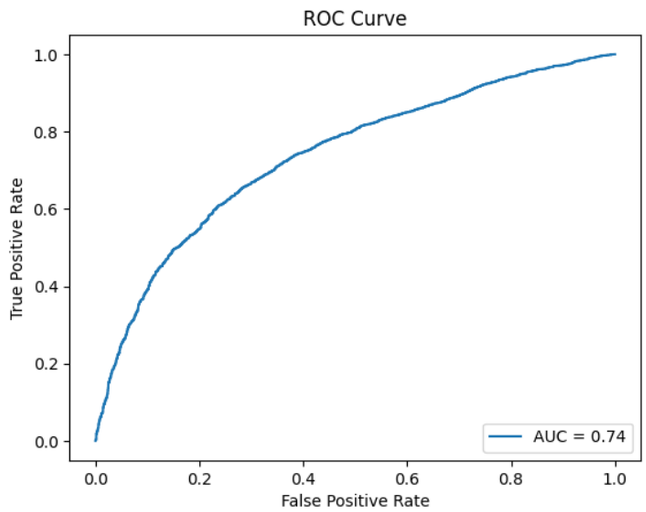
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve
import seaborn as sns

# Helper functions (same as provided)
def plot_auc(y_true, y_scores):
    """Plot the ROC AUC curve."""
    auc_score = roc_auc_score(y_true, y_scores)
    print(f"ROC AUC Score: {auc_score:.4f}")
    fpr, tpr, _ = roc_curve(y_true, y_scores)
    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes):
    """Plot the confusion matrix."""
    cm = confusion_matrix(y_true, y_pred)
```

```
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Model creation function for ResNet-50
def create_resnet50_model(num_classes):
    model = models.resnet50(pretrained=True)

    # Freeze the convolutional layers
    for param in model.parameters():
        param.requires_grad = False

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

    return model

# Training function
def train_resnet50(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, pa
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch + 1}/{num_epochs}")

        # 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)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

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

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects.double() / len(train_loader.dataset)

        print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_running_corrects = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Track statistics
```

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

        val_loss = val_running_loss / len(val_loader.dataset)
        val_acc = val_running_corrects.double() / len(val_loader.dataset)

        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

        # Save the model if it has the best accuracy so far
        if val_acc > best_accuracy:
            best_accuracy = val_acc
            best_model_wts = model.state_dict()
            early_stop_counter = 0  # Reset counter when validation improves
        else:
            early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break

    # Load the best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'best_resnet50_model.pth')
    print(f"Best Validation Accuracy: {best_accuracy:.4f}")

# Testing function
def test_resnet50(model, test_loader, device, class_names):
    """Test the model and plot metrics."""
    model.eval()
    test_corrects = 0
    all_labels = []
    all_preds = []
    all_probs = []

    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)
            probs = torch.softmax(outputs, dim=1)[:, 1]  # Assuming binary classification

            test_corrects += torch.sum(preds == labels.data)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
            all_probs.extend(probs.cpu().numpy())

    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")

    # Plot the confusion matrix
    plot_confusion_matrix(all_labels, all_preds, class_names)

    # Plot the ROC AUC curve
    plot_auc(all_labels, all_probs)

# Main workflow
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
img_size = 224

train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
```

```
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

num_classes = len(train_dataset.classes)
class_names = train_dataset.classes
model = create_resnet50_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)

train_resnet50(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patie
model.load_state_dict(torch.load('best_resnet50_model.pth'))
test_resnet50(model, test_loader, device, class_names)
```



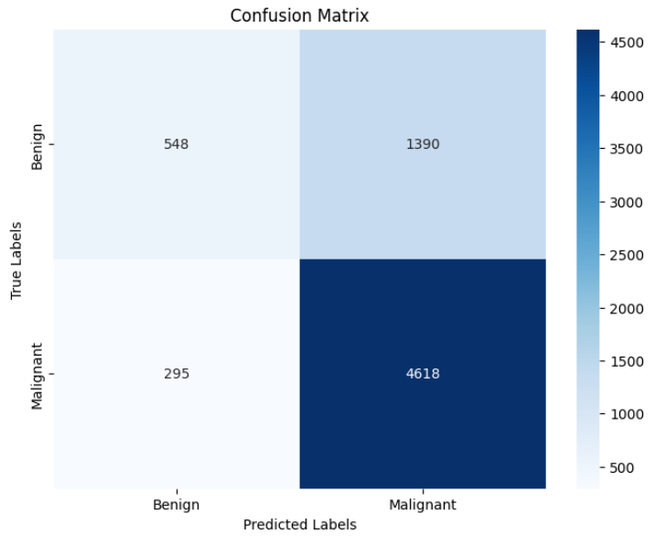


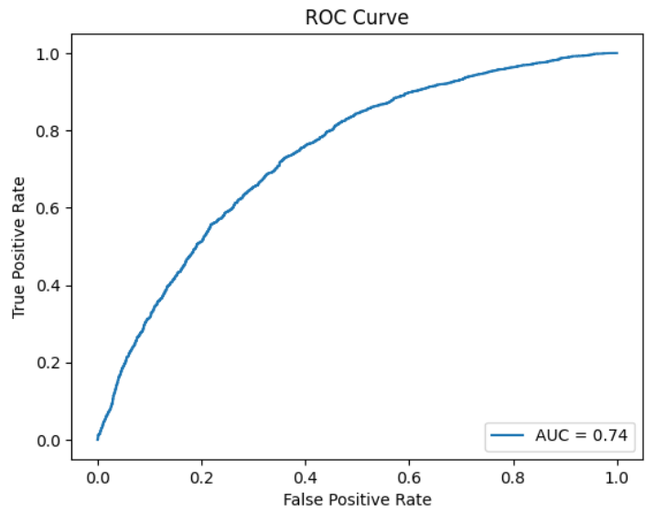
Resnet18 (AUC,CONFUSION_MATRIX)

```
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
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve
import seaborn as sns

# Helper functions (same as provided)
def plot_auc(y_true, y_scores):
    """Plot the ROC AUC curve."""
    auc_score = roc_auc_score(y_true, y_scores)
    print(f"ROC AUC Score: {auc_score:.4f}")
    fpr, tpr, _ = roc_curve(y_true, y_scores)
    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes):
    """Plot the confusion matrix."""
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Model creation function for ResNet-18
def create_resnet18_model(num_classes):
    model = models.resnet18(pretrained=True)

    # Freeze the convolutional layers
    for param in model.parameters():
        param.requires_grad = False

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

    return model

# Training function
def train_resnet18(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=10, pa
    model = model.to(device)
    best_accuracy = 0.0
    best_model_wts = model.state_dict()
    early_stop_counter = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch + 1}/{num_epochs}")

        # 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)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
```

```
            optimizer.step()

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

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects.double() / len(train_loader.dataset)

        print(f"Training Loss: {epoch_loss:.4f}, Training Accuracy: {epoch_acc:.4f}")

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_running_corrects = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Track statistics
                val_running_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_running_corrects += torch.sum(preds == labels.data)

        val_loss = val_running_loss / len(val_loader.dataset)
        val_acc = val_running_corrects.double() / len(val_loader.dataset)

        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")

        # Save the model if it has the best accuracy so far
        if val_acc > best_accuracy:
            best_accuracy = val_acc
            best_model_wts = model.state_dict()
            early_stop_counter = 0  # Reset counter when validation improves
        else:
            early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break

    # Load the best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'best_resnet18_model.pth')
    print(f"Best Validation Accuracy: {best_accuracy:.4f}")

# Testing function
def test_resnet18(model, test_loader, device, class_names):
    """Test the model and plot metrics."""
    model.eval()
    test_corrects = 0
    all_labels = []
    all_preds = []
    all_probs = []

    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)
            probs = torch.softmax(outputs, dim=1)[:, 1]  # Assuming binary classification

            test_corrects += torch.sum(preds == labels.data)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
            all_probs.extend(probs.cpu().numpy())
```

```
    test_acc = test_corrects.double() / len(test_loader.dataset)
    print(f"Test Accuracy: {test_acc:.4f}")

    # Plot the confusion matrix
    plot_confusion_matrix(all_labels, all_preds, class_names)

    # Plot the ROC AUC curve
    plot_auc(all_labels, all_probs)

# Main workflow
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
img_size = 224

train_dir = '/kaggle/input/bcd-data/clasification-roi/train'
val_dir = '/kaggle/input/bcd-data/clasification-roi/val'
test_dir = '/kaggle/input/bcd-data/clasification-roi/test'

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(img_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

train_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
val_dataset = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

num_classes = len(train_dataset.classes)
class_names = train_dataset.classes
model = create_resnet18_model(num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)

train_resnet18(model, train_loader, val_loader, criterion, optimizer, device, num_epochs=50, patie
model.load_state_dict(torch.load('best_resnet18_model.pth'))
test_resnet18(model, test_loader, device, class_names)
```





Using gradio library on two models(resnet50,resnet18) to run and upload image and get result

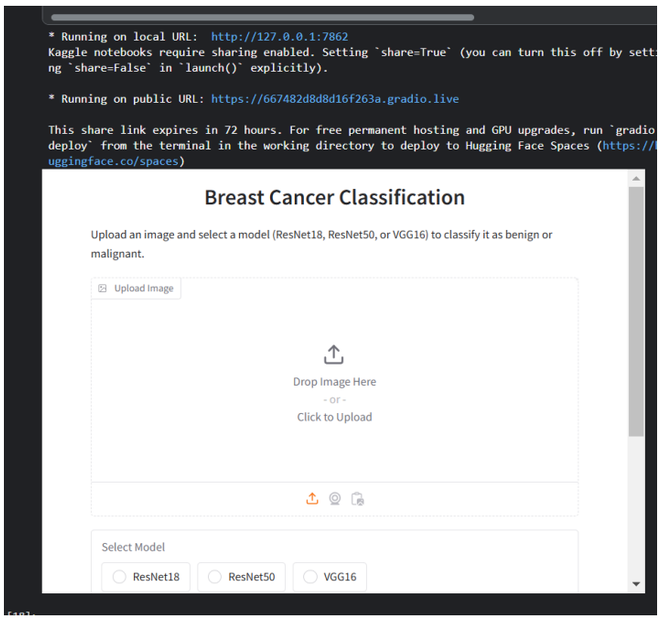
```
import gradio as gr
from PIL import Image
import torch
import torch.nn as nn
from torchvision import models, transforms

# Load your trained model dynamically
def load_model(model_name, model_path):
    if model_name == "ResNet18":
        model = models.resnet18(pretrained=False)  # Use pretrained=False for your custom-trained r
        model.fc = nn.Linear(model.fc.in_features, 2)  # Binary classification
    elif model_name == "ResNet50":
        model = models.resnet50(pretrained=False)
        model.fc = nn.Linear(model.fc.in_features, 2)  # Binary classification
    else:
        raise ValueError("Invalid model name.")

    # Load your trained model weights
    model.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))
    model.eval()
    return model

# Preprocessing function for input images
def preprocess_image(image):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

```
    ])
    return transform(image).unsqueeze(0)  # Add batch dimension

# Prediction function
def predict(image, model_name):
    image = Image.fromarray(image)  # Convert numpy array to PIL image
    input_tensor = preprocess_image(image)

    # Map model names to weight file paths
    model_paths = {
    "ResNet18": "/kaggle/working/best_resnet18_model.pth",
    "ResNet50": "/kaggle/working/best_resnet50_model.pth"
    }

    model_path = model_paths[model_name]
    model = load_model(model_name, model_path)

    with torch.no_grad():
        outputs = model(input_tensor)
        probabilities = torch.softmax(outputs, dim=1)[0]

        # Extract class names and their respective probabilities
        benign_prob = probabilities[0].item()
        malignant_prob = probabilities[1].item()
        predicted_idx = probabilities.argmax().item()
        predicted_class = class_names[predicted_idx]

    # Create a response string
    response = (
        f"Model: {model_name}\n"
        f"Predicted Class: {predicted_class}\n"
        f"Benign Probability: {benign_prob:.2f}\n"
        f"Malignant Probability: {malignant_prob:.2f}"
    )
    return response

# Define global variables
class_names = ["benign", "malignant"]  # Binary classes

# Define the Gradio interface
interface = gr.Interface(
    fn=predict,
    inputs=[
        gr.Image(type="numpy", label="Upload Image"),
        gr.Radio(["ResNet18", "ResNet50"], label="Select Model")
    ],
    outputs=gr.Textbox(label="Prediction"),
    title="Breast Cancer Classification",
    description="Upload an image and select a model (ResNet18 or ResNet50) to classify it as benign
)

# Launch the Gradio app
interface.launch()
```

Using gradio library on all three models(resnet50,resnet18,vgg16)

```
import gradio as gr
from PIL import Image
import torch
import torch.nn as nn
from torchvision import models, transforms

# Load your trained model dynamically
def load_model(model_name, model_path):
    if model_name == "ResNet18":
        model = models.resnet18(pretrained=False)  # Use pretrained=False for your custom-trained m
        model.fc = nn.Linear(model.fc.in_features, 2)  # Binary classification
    elif model_name == "ResNet50":
        model = models.resnet50(pretrained=False)
        model.fc = nn.Linear(model.fc.in_features, 2)  # Binary classification
```

```
    elif model_name == "VGG16":
        model = models.vgg16(pretrained=False)
        model.classifier[6] = nn.Linear(model.classifier[6].in_features, 2)  # Binary classificatio
    else:
        raise ValueError("Invalid model name.")

    # Load your trained model weights
    model.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))
    model.eval()
    return model

# Preprocessing function for input images
def preprocess_image(image):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
    return transform(image).unsqueeze(0)  # Add batch dimension

# Prediction function
def predict(image, model_name):
    image = Image.fromarray(image)  # Convert numpy array to PIL image
    input_tensor = preprocess_image(image)

    # Map model names to weight file paths
    model_paths = {
    "ResNet18": "/kaggle/working/best_resnet18_model.pth",
    "ResNet50": "/kaggle/working/best_resnet50_model.pth",
    "VGG16": "/kaggle/working/best_vgg16_model.pth"
    }

    model_path = model_paths[model_name]
    model = load_model(model_name, model_path)

    with torch.no_grad():
        outputs = model(input_tensor)
        probabilities = torch.softmax(outputs, dim=1)[0]

        # Extract class names and their respective probabilities
        benign_prob = probabilities[0].item()
        malignant_prob = probabilities[1].item()
        predicted_idx = probabilities.argmax().item()
        predicted_class = class_names[predicted_idx]

    # Create a response string
    response = (
        f"Model: {model_name}\n"
        f"Predicted Class: {predicted_class}\n"
        f"Benign Probability: {benign_prob:.2f}\n"
        f"Malignant Probability: {malignant_prob:.2f}"
    )
    return response

# Define global variables
class_names = ["benign", "malignant"]  # Binary classes

# Define the Gradio interface
interface = gr.Interface(
    fn=predict,
    inputs=[
        gr.Image(type="numpy", label="Upload Image"),
        gr.Radio(["ResNet18", "ResNet50", "VGG16"], label="Select Model")
    ],
    outputs=gr.Textbox(label="Prediction"),
    title="Breast Cancer Classification",
    description="Upload an image and select a model (ResNet18, ResNet50, or VGG16) to classify it a
)

# Launch the Gradio app
interface.launch()
```



code implements a **Gradio-based web interface** that allows users to upload images and classify them as **benign or malignant** using different pre-trained deep learning models. The primary use cases of this code are as follows:

1. **Model Selection for Classification**: The user can choose between three popular pre-trained models—**ResNet18**, **ResNet50**, and **VGG16**—to classify breast cancer images. These models are tailored for binary classification (benign vs. malignant) and can be loaded dynamically from the specified file paths.
2. **Image Preprocessing and Prediction**: The code includes a preprocessing function to resize, crop, and normalize the input images. Once processed, the image is fed into the chosen model, and the model's output is passed through a softmax function to compute the probabilities of the image belonging to either of the two classes (benign or malignant).
3. **Real-Time Predictions**: The Gradio interface allows users to interact with the model in real-time. Users can upload an image and select the model they wish to use. The app will then return a prediction indicating whether the uploaded image is benign or malignant, along with the probability for each class.
4. **Medical Use Case**: The tool is intended for use in medical or diagnostic settings, where it can assist in classifying breast cancer images (e.g., mammograms or MRI scans) based on texture and visual features, aiding medical professionals in their diagnostic processes.
5. **Easy Deployment and Accessibility**: The Gradio interface simplifies the process of deploying machine learning models for non-technical users, making it possible for healthcare professionals or researchers to utilize deep learning models without requiring programming knowledge. It also makes the model accessible via a browser for easy testing and interaction.

Key Features:

- **Model Customization**: Users can select from multiple model architectures (ResNet18, ResNet50, and VGG16).
- **Interactive Prediction**: Users can upload an image and receive real-time predictions with probabilities for each class.
- **Web-Based Interface**: The Gradio interface provides a simple, user-friendly platform for interacting with the models.