

**PROJECT TITLE:**

TumorTrace: MRI-Based AI for Breast Cancer Detection

**INTRODUCTION:**

Breast cancer is one of the most common cancers among women, making early detection essential for improving treatment outcomes. Magnetic Resonance Imaging (MRI) is a valuable diagnostic tool that offers detailed images of breast tissue, allowing for the detection of tumors that may not be visible through traditional methods. However, the interpretation of MRI scans can be subjective, leading to potential diagnostic inaccuracies.

The TumorTrace project aims to enhance breast cancer detection by leveraging artificial intelligence (AI) and machine learning (ML) techniques. By automating the identification of malignant and benign tumors in MRI scans, TumorTrace seeks to support healthcare professionals in making more accurate and timely diagnoses. Ultimately, this project aspires to improve early detection and patient outcomes in breast cancer care.

**DATASET:**

The dataset selected for the TumorTrace project consists of MRI images that have undergone image segmentation, ensuring precise delineation of regions of interest (ROIs) related to breast tumors. This segmentation process enhances the quality of the images by highlighting relevant features while reducing background noise, thereby facilitating more accurate analysis. The dataset is divided into three distinct subsets: training, testing, and validation.

**WHY HAS THE DATASET BEEN RESIZED AND NORMALIZED AS PER IMAGENET DATASET DIMENSIONS?**

Resizing and normalizing dataset as per ImageNet dimensions is a standard practice in deep learning that enhances model compatibility, improves performance through transfer learning, and ensures consistent training and evaluation conditions. It's a way to harness the power of models pretrained on large datasets to solve specific problems more effectively and efficiently.

**CONVOLUTIONAL NEURAL NETWORKS:**

**The CS231n course overview on convolutional networks (CNNs/ConvNets):**

- **Neural Networks and Image Data:** ConvNets are a specialized type of neural network that excel at handling grid-like data, such as images. Instead of fully connecting each neuron to every input, ConvNets use a different architecture that exploits spatial hierarchies in data.
- **Layers in ConvNets:**

- **Convolutional Layer:** This layer applies a set of filters to the input image, creating feature maps that detect various elements (e.g., edges, textures). Each filter slides across the image and computes dot products between the filter values and the input.
  - **Pooling Layer:** This layer down-samples the input (usually after the convolutional layer) to reduce its dimensions and computational complexity. Max pooling is a common technique that selects the maximum value from a region of the image.
  - **Fully Connected Layer:** After multiple convolution and pooling layers, the high-level reasoning in the network is typically done by fully connected layers that aggregate the features learned and classify the data.
- **Benefits:** Their architecture allows ConvNets to capture local dependencies, preserve spatial hierarchies, and reduce the number of parameters (compared to fully connected networks).
- **Architecture:**
  - CNNs are composed of layers: Convolutional (CONV), Pooling (POOL), and Fully Connected (FC)
  - Input layers are typically divisible by 2 many times (e.g., 32, 64, 224)
- **Convolutional Layer:**
  - Uses learnable filters to detect features
  - Hyperparameters: filter size ( $F$ ), stride ( $S$ ), padding ( $P$ )
  - Typically use small filters (3x3 or 5x5) with stride 1 and padding
- **Pooling Layer:**
  - Reduces spatial dimensions of input
  - Common: Max pooling with 2x2 receptive field and stride 2
  - No learnable parameters
- **Fully Connected Layer:**
  - Similar to traditional neural networks
  - Can be converted to CONV layers and vice versa
- **Design Principles:**
  - Prefer stacking small filters over using large receptive fields
  - Use padding to maintain spatial dimensions in CONV layers
  - Let POOL layers handle downsampling
- **Famous Architectures:**
  - LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, ResNet
- **Computational Considerations:**
  - Memory is often the main bottleneck
  - Sources of memory usage: activations, parameters, and miscellaneous data
  - Reducing batch size can help fit models in memory
- **Parameter Sharing:**
  - Reduces number of parameters
  - Allows detection of features regardless of position in image
- **Implementation:**
  - Can be implemented as matrix multiplication using im2col operation
  - Backpropagation for CONV is also a convolution operation

## **WHY CNNS ARE MORE EFFICIENT THAN FULLY CONNECTED NETWORKS FOR IMAGE CLASSIFICATION?**

Convolutional Neural Networks (CNNs) are more efficient than fully connected networks (FCNs) for image classification due to their ability to exploit the spatial structure of images. Here's why:

### **a. Spatial Hierarchies:**

- **Local connectivity:** CNNs use convolutional layers that apply filters to small regions of the image, capturing spatial features like edges, textures, and shapes. This localized approach helps capture the hierarchical structure of images (from low-level features to high-level features).
- **Shared weights:** CNNs use the same filter (kernel) across the image, which means fewer parameters compared to FCNs, where every pixel is connected to every node in the next layer. This reduces the complexity and the risk of overfitting, making CNNs more efficient for images.
- **Translation invariance:** CNNs can detect patterns anywhere in the image because filters slide over the image, allowing for translation invariance, meaning a feature can be recognized no matter where it appears.

### **b. Parameter Efficiency:**

- **Fewer parameters:** In FCNs, every input pixel is connected to every neuron in the next layer, leading to an enormous number of parameters, especially for large images. In CNNs, the use of convolutional filters significantly reduces the number of connections and parameters.
- **Pooling layers:** CNNs use pooling (usually max-pooling) to reduce the spatial size of the image representations, which further reduces the number of parameters and speeds up computation, all while retaining important features.

### **c. Feature Extraction:**

CNNs automatically learn relevant features from the data through multiple layers of convolutions, while FCNs do not have the spatial awareness to capture hierarchical and local image features efficiently.

## **DATA AUGMENTATION:**

Data augmentation is a technique used in machine learning to increase the size and diversity of a training dataset by creating modified versions of existing data points. This helps improve the model's robustness and generalization capabilities.

### **Key Points:**

#### **1. Purpose:**

- To artificially expand the dataset, reducing overfitting and enhancing model performance on unseen data.

#### **2. Common Techniques:**

- Geometric Transformations: Rotation, flipping, translation, scaling.
- Color Adjustments: Brightness, contrast, saturation changes.
- Distortions: Shearing, zooming, cropping.

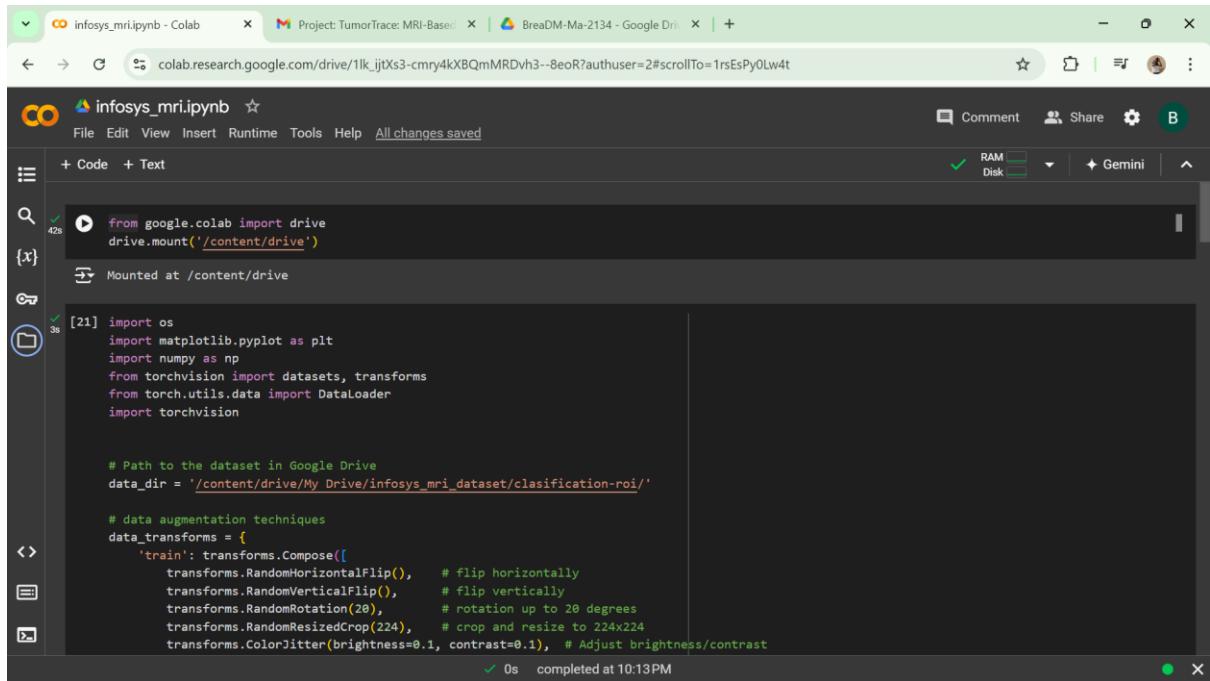
- Noise Addition: Introducing random noise to make models more robust.

### 3. Implementation:

- Can be done on-the-fly during training or as a pre-processing step.

### 4. Benefits:

- Improves accuracy, reduces overfitting, and is a cost-effective way to enhance datasets without needing more labeled data.



The screenshot shows a Google Colab notebook titled "infosys\_mri.ipynb". The code cell [21] contains Python code for dataset augmentation and loading. It imports necessary libraries (google.colab, drive, os, matplotlib.pyplot, numpy, torchvision, torch.utils.data) and defines data augmentation techniques using transforms.Compose. The data augmentation includes horizontal flip, vertical flip, rotation up to 20 degrees, random resized crop (224x224), and color jitter (brightness=0.1, contrast=0.1). The data directory is set to '/content/drive/My Drive infosys\_mri\_dataset/classification-roi/'. The code then loads the dataset using datasets.ImageFolder and creates data loaders using DataLoader with batch\_size=32 and shuffle=True for 'train', 'val', and 'test' splits. The class names are retrieved from the 'train' dataset.

```

from google.colab import drive
drive.mount('/content/drive')

[21]
import os
import matplotlib.pyplot as plt
import numpy as np
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torchvision

# Path to the dataset in Google Drive
data_dir = '/content/drive/My Drive infosys_mri_dataset/classification-roi/'

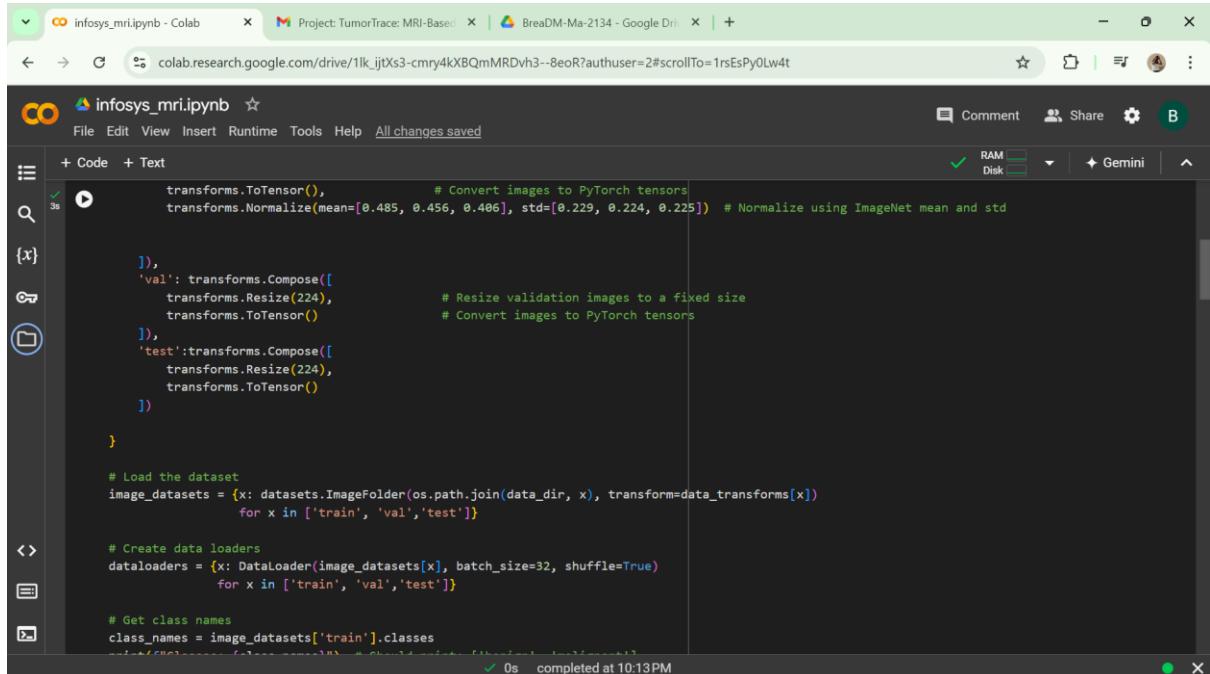
# data augmentation techniques
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomHorizontalFlip(), # flip horizontally
        transforms.RandomVerticalFlip(), # flip vertically
        transforms.RandomRotation(20), # rotation up to 20 degrees
        transforms.RandomResizedCrop(224), # crop and resize to 224x224
        transforms.ColorJitter(brightness=0.1, contrast=0.1), # Adjust brightness/contrast
        transforms.ToTensor(), # Convert images to PyTorch tensors
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Normalize using ImageNet mean and std
    ]),
    'val': transforms.Compose([
        transforms.Resize(224), # Resize validation images to a fixed size
        transforms.ToTensor() # Convert images to PyTorch tensors
    ]),
    'test': transforms.Compose([
        transforms.Resize(224),
        transforms.ToTensor()
    ])
}

# Load the dataset
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform=data_transforms[x])
                  for x in ['train', 'val', 'test']}

# Create data loaders
dataloaders = {x: DataLoader(image_datasets[x], batch_size=32, shuffle=True)
               for x in ['train', 'val', 'test']}

# Get class names
class_names = image_datasets['train'].classes

```



The screenshot shows the continuation of the Google Colab notebook "infosys\_mri.ipynb". The code cell [39] continues where the previous one left off, defining the data loaders for 'train', 'val', and 'test' splits. The data loaders use the DataLoader class with a batch size of 32 and shuffle set to True for each split.

```

# Create data loaders
dataloaders = {x: DataLoader(image_datasets[x], batch_size=32, shuffle=True)
               for x in ['train', 'val', 'test']}

```

The screenshot shows a Google Colab notebook titled "infosys\_mri.ipynb". The code cell contains Python code for counting the number of benign and malignant tumors in training, validation, and test datasets. The output shows the distribution of classes and counts for each set.

```

print(f"Classes: {class_names}") # Should print: ['benign', 'malignant']

# Count the number of images in each class for train and val datasets
def count_class_distribution(dataset):
    benign_count = 0
    malignant_count = 0
    for _, label in dataset.samples:
        if label == 0:
            benign_count += 1
        else:
            malignant_count += 1
    return benign_count, malignant_count

# Count distribution in training and validation sets
train_benign, train_malignant = count_class_distribution(image_datasets['train'])
val_benign, val_malignant = count_class_distribution(image_datasets['val'])
test_benign, test_malignant = count_class_distribution(image_datasets['test'])

print(f"Training set - Benign: {train_benign}, Malignant: {train_malignant}")
print(f"Validation set - Benign: {val_benign}, Malignant: {val_malignant}")
print(f"Test set - Benign: {test_benign}, Malignant: {test_malignant}")

```

Output:

```

Classes: ['Benign', 'Malignant']
Training set - Benign: 5599, Malignant: 14875
Validation set - Benign: 408, Malignant: 1581
Test set - Benign: 1938, Malignant: 4913

```

## WHAT IS FEATURE EXTRACTION?

**Feature extraction** refers to the process of transforming raw data into a set of features that better represent the underlying patterns in the data, making it easier for a model to classify or make predictions. In the context of CNNs and image classification:

- **Initial Layers:** CNNs extract low-level features like edges, corners, and textures.
- **Deeper Layers:** As you move deeper into the network, CNNs extract more complex features, such as shapes, objects, or even entire faces, in image classification tasks.
- **Importance:** Feature extraction is crucial because it allows the network to reduce the input data to more meaningful and manageable representations, leading to better learning and performance.

In traditional machine learning, feature extraction was done manually by selecting specific characteristics, but CNNs automate this process.

## **VGG-16 ARCHITECTURE:**

VGG-16 consists of:

- 13 Convolutional layers
- 3 Fully connected layers

The architecture can be summarized as:

1. **Convolutional Layers:**
  - o 2 Conv layers: 64 filters (3x3 kernels), input 224×224×3 → output 224×224×64 (conv1\_1 and conv1\_2)
  - o 2 Conv layers: 128 filters (3x3 kernels), input 112×112×64 → output 112×112×128 (conv2\_1 and conv2\_2)

- 3 Conv layers: 256 filters (3x3 kernels), input 56x56x128 → output 56x56x256 (conv3\_1, conv3\_2, conv3\_3)
- 3 Conv layers: 512 filters (3x3 kernels), input 28x28x256 → output 28x28x512 (conv4\_1, conv4\_2, conv4\_3)
- 3 Conv layers: 512 filters (3x3 kernels), input 14x14x512 → output 14x14x512 (conv5\_1, conv5\_2, conv5\_3)
- Followed by max pooling layers after each group.

## 2. Fully Connected Layers:

- 1 FC layer: 4096 neurons
- 1 FC layer: 4096 neurons
- 1 FC layer: 1000 neurons (for classification)

### Formula for Parameters:

For convolutional layers:

$$\begin{aligned} \text{Parameters} = & (n_{\text{filters}} \times n_{\text{input channels}} \times \text{kernel height} \times \text{kernel width}) \\ & + n_{\text{filters}} \text{ (for biases)} \end{aligned}$$

For fully connected layers:

$$\text{Parameters} = (\text{input units} \times \text{output units}) + \text{output units}$$

### Step-by-Step Calculation:

1. **Layer 1: conv1\_1:**  $(64 \times 3 \times 3 \times 3) + 64 = 1792$
2. **Layer 2: conv1\_2:**  $(64 \times 64 \times 3 \times 3) + 64 = 36,928$
3. **Layer 3: conv2\_1:**  $(128 \times 64 \times 3 \times 3) + 128 = 73,856$
4. **Layer 4: conv2\_2:**  $(128 \times 128 \times 3 \times 3) + 128 = 147,584$
5. **Layer 5: conv3\_1:**  $(256 \times 128 \times 3 \times 3) + 256 = 295,168$
6. **Layer 6: conv3\_2:**  $(256 \times 256 \times 3 \times 3) + 256 = 590,080$
7. **Layer 7: conv3\_3:**  $(256 \times 256 \times 3 \times 3) + 256 = 590,080$
8. **Layer 8: conv4\_1:**  $(512 \times 256 \times 3 \times 3) + 512 = 1,180,160$
9. **Layer 9: conv4\_2:**  $(512 \times 512 \times 3 \times 3) + 512 = 2,359,808$
10. **Layer 10: conv4\_3:**  $(512 \times 512 \times 3 \times 3) + 512 = 2,359,808$
11. **Layer 11: conv5\_1:**  $(512 \times 512 \times 3 \times 3) + 512 = 2,359,808$
12. **Layer 12: conv5\_2:**  $(512 \times 512 \times 3 \times 3) + 512 = 2,359,808$
13. **Layer 13: conv5\_3:**  $(512 \times 512 \times 3 \times 3) + 512 = 2,359,808$

### Fully Connected Layers:

1. **Layer 14: FC1:**  $(7 \times 7 \times 512) \times 4096 + 4096 = 102,764,544$
2. **Layer 15: FC2:**  $4096 \times 4096 + 4096 = 16,781,$
3. **Layer 16: FC3:**  $4096 \times 1000 + 1000 = 4,097,000$

**Total Number of Parameters:**

Summing all the parameters:

$$1792 + 36,928 + 73,856 + 147,584 + 295,168 + 590,080 + 590,080 + 1,180,160 + 2,359,808 + 2,359,808 + 2,359,808 + 2,359,808 + 102,764,544 + 16,781,312 + 4,097,000 = 138,357,544$$

**So, VGG-16 has approximately 138.3 million parameters.**

## CONTINUED PROJECT REPORT: SUBMITTED BY BHAVYA TIWARI

### 1. PLOT IMAGE AND PIXEL VALUE VS FREQUENCY GRAPH

CODE:

```
infosys_mri.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
3s
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage.feature import hog
from skimage import exposure

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/Breast-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L') # Convert to grayscale

# Resize the image to 224x224
image = image.resize((224, 224))

# Convert image to a NumPy array
image_np = np.array(image)

# Plot the original image, HOG image, and histogram of pixel values
plt.figure(figsize=(18, 6))

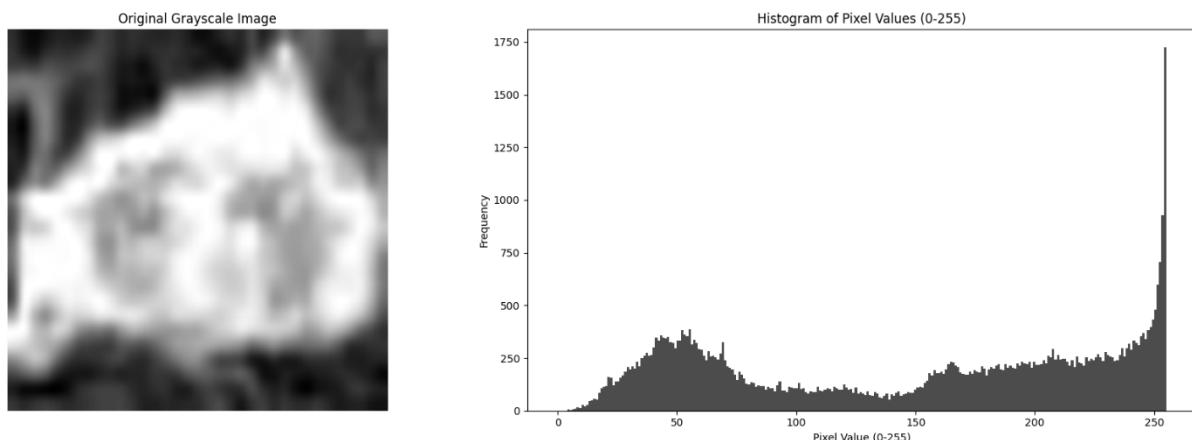
# Display the original grayscale image
plt.subplot(1, 2, 1)
plt.imshow(image_np, cmap='gray')
plt.title('Original Grayscale Image')
plt.axis('off')

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

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

✓ 2s completed at 11:09 PM

OUTPUT:



## 2. HOG FEATURE EXTRACTION

CODE:

The screenshot shows a Jupyter Notebook interface with the following code:

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage.feature import hog
from skimage import exposure

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L') # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

# Perform HOG feature extraction
hog_features, hog_image = hog(image_np, orientations=8, pixels_per_cell=(16, 16),
                               cells_per_block=(1, 1), visualize=True, channel_axis=None)

# Rescale HOG image for better visualization
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# Plot the original and HOG images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_np, cmap='gray')
plt.title('Original Grayscale Image')

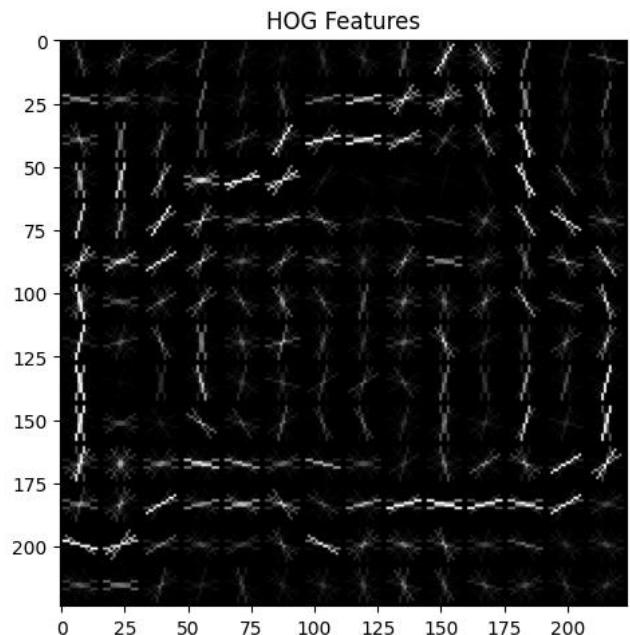
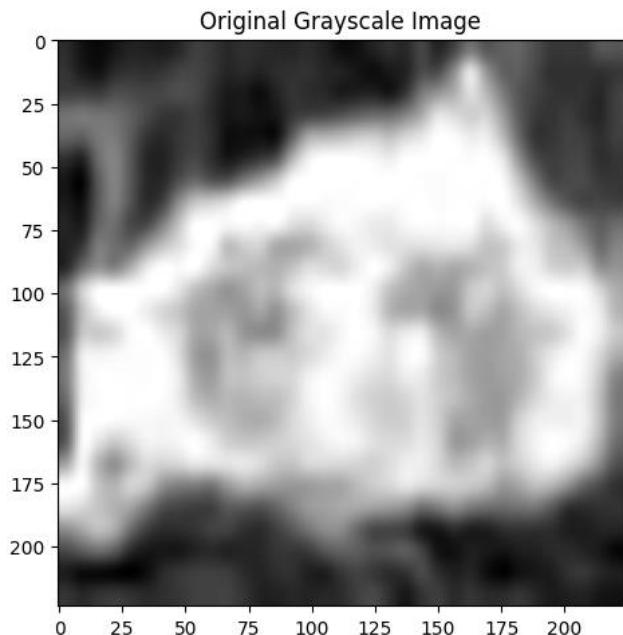
plt.subplot(1, 2, 2)
plt.imshow(hog_image_rescaled, cmap='gray')
plt.title('HOG Features')

plt.show()

print("Image Array:")
print(image_np)
print("HOG Features Shape:", hog_features.shape)
```

At the bottom right of the code cell, there is a status bar indicating "✓ 2s completed at 11:09 PM".

OUTPUT:



### 3. SOBEL OPERATOR

CODE:

infosys\_mri.ipynb ☆

File Edit View Insert Runtime Tools Help

+ Code + Text

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from numba import prange

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/Breast-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L') # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

# Sobel 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]])

# Function to perform convolution using the kernel
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
```

✓ 2s completed at 11:09PM

infosys\_mri.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Apply convolution using Sobel X and Y kernels
gradient_x = convolve(image_np, sobel_x)
gradient_y = convolve(image_np, sobel_y)

# Compute the gradient magnitude
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

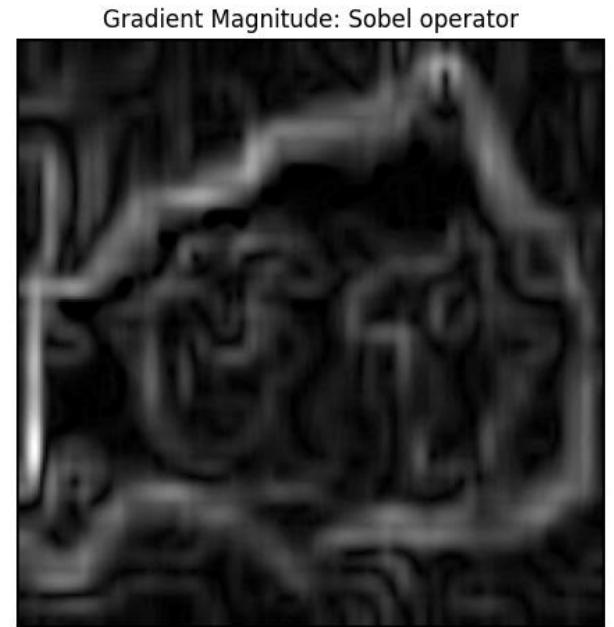
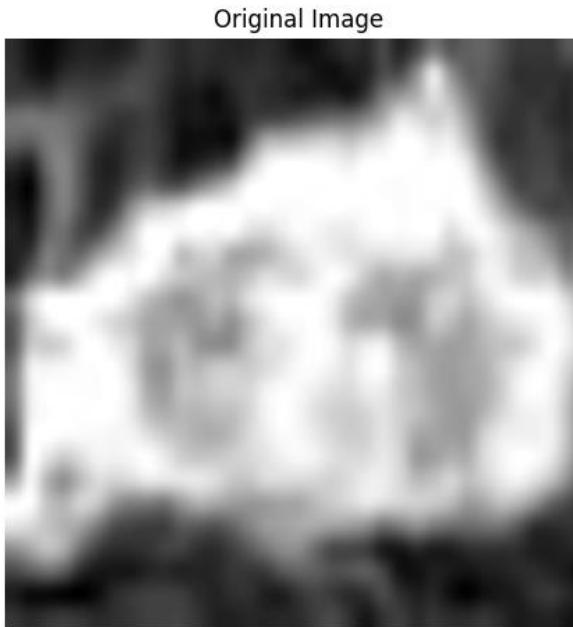
# Normalize the result for visualization (to range 0-1)
gradient_magnitude = (gradient_magnitude - gradient_magnitude.min()) / (gradient_magnitude.max() - gradient_magnitude.min())

# Display the original and gradient magnitude images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(image_np, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Gradient Magnitude: Sobel operator')
plt.imshow(gradient_magnitude, cmap='gray')
plt.axis('off')

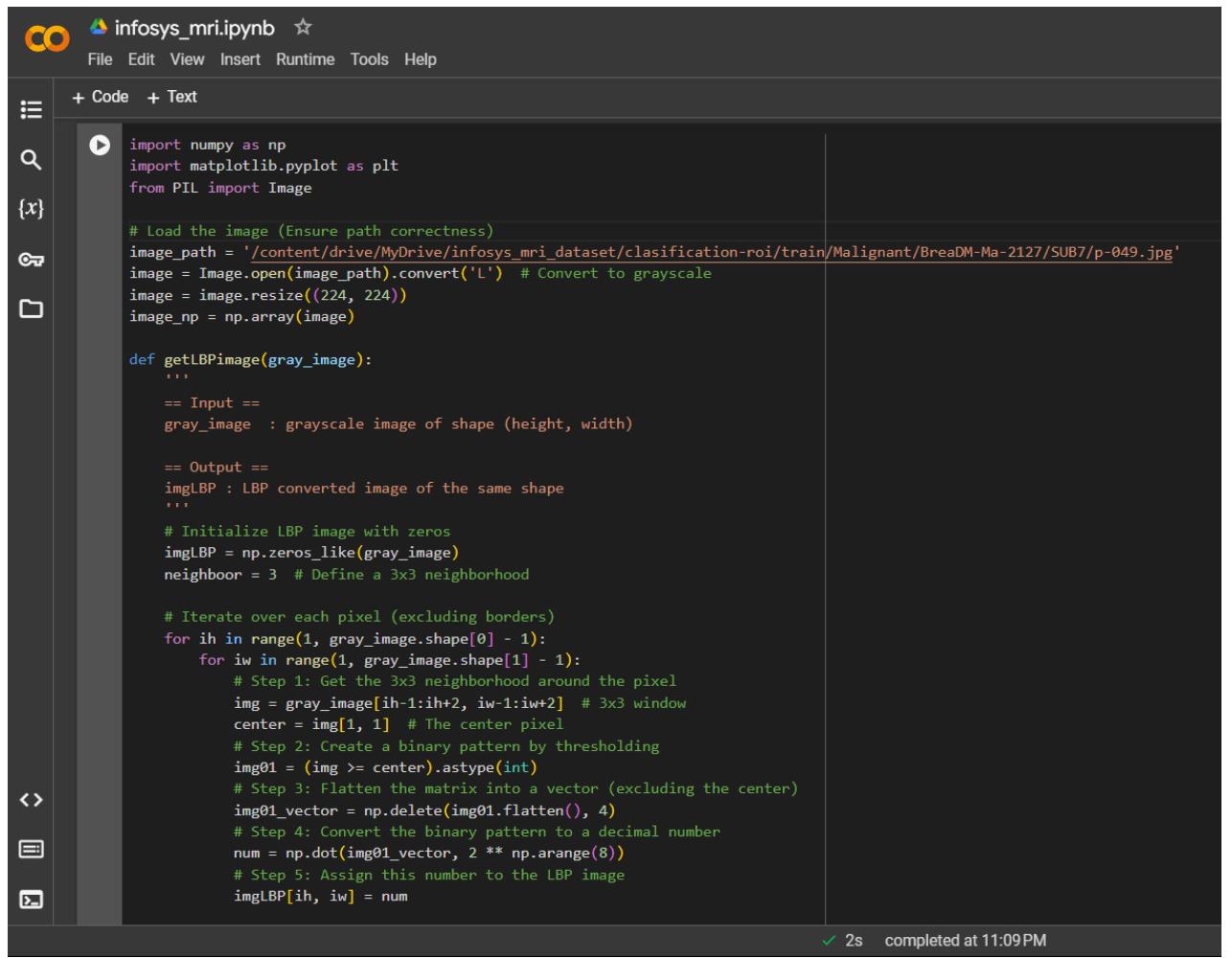
plt.show()
```

## OUTPUT:



## 4. LOCAL BINARY PATTERN

### CODE:



```
infosys_mri.ipynb ☆
File Edit View Insert Runtime Tools Help
+ Code + Text
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/Breast-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L') # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimage(gray_image):
    ...
    == Input ==
    gray_image : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape
    ...
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighbor = 3 # Define a 3x3 neighborhood

    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window
            center = img[1, 1] # The center pixel
            # Step 2: Create a binary pattern by thresholding
            img01 = (img >= center).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num
```

✓ 2s completed at 11:09 PM

infosys\_mri.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```

return imgLBP

# Apply LBP to the single image
imgLBP = getLBPIimage(image_np)

# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()

# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))

# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")

# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("LBP Converted Image")

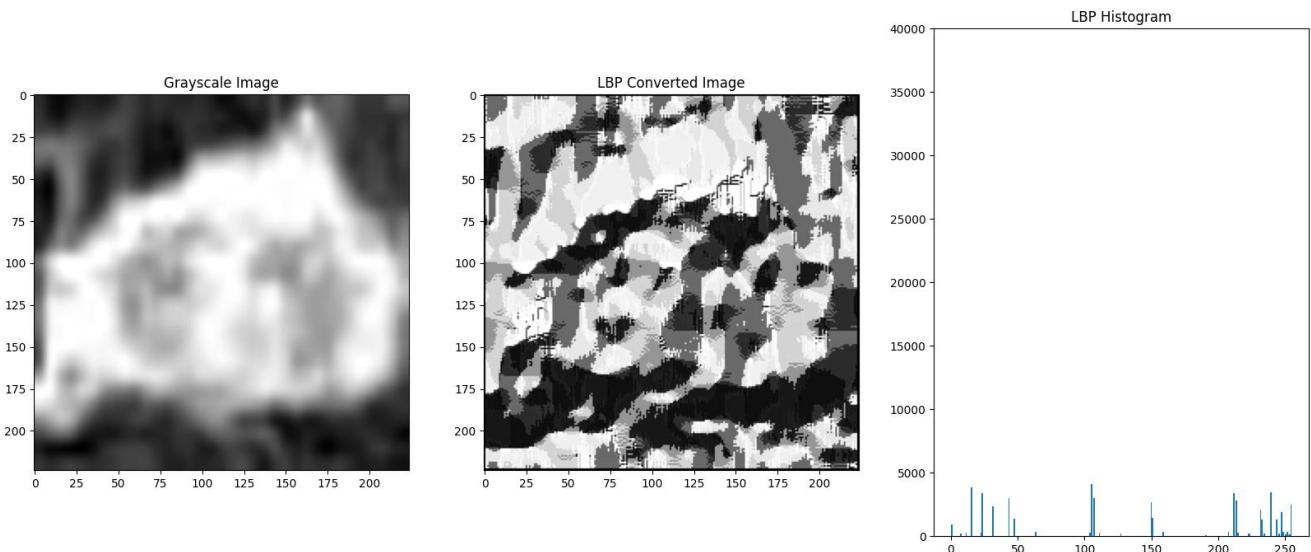
# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_xlim(0, 40000)
lbp = lbp[:-1]

# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")

plt.show()

```

## OUTPUT:



## 5. MEAN-BASED LBP:

CODE:

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/Breast-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L') # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingMean(gray_image):
    ...
    == Input ==
    gray_image : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using mean value
    ...
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighbor = 3 # Define a 3x3 neighborhood
    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window
            mean_value = np.mean(img) # Calculate mean of the neighborhood
            # Step 2: Create a binary pattern by thresholding with mean value
            img01 = (img >= mean_value).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num

    return imgLBP
```

The cell has been run successfully, indicated by a green checkmark and the message "2s completed at 11:09PM".

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
# Apply LBP to the single image using mean value
imgLBP = getLBPimageUsingMean(image_np)

# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()

# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))

# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")

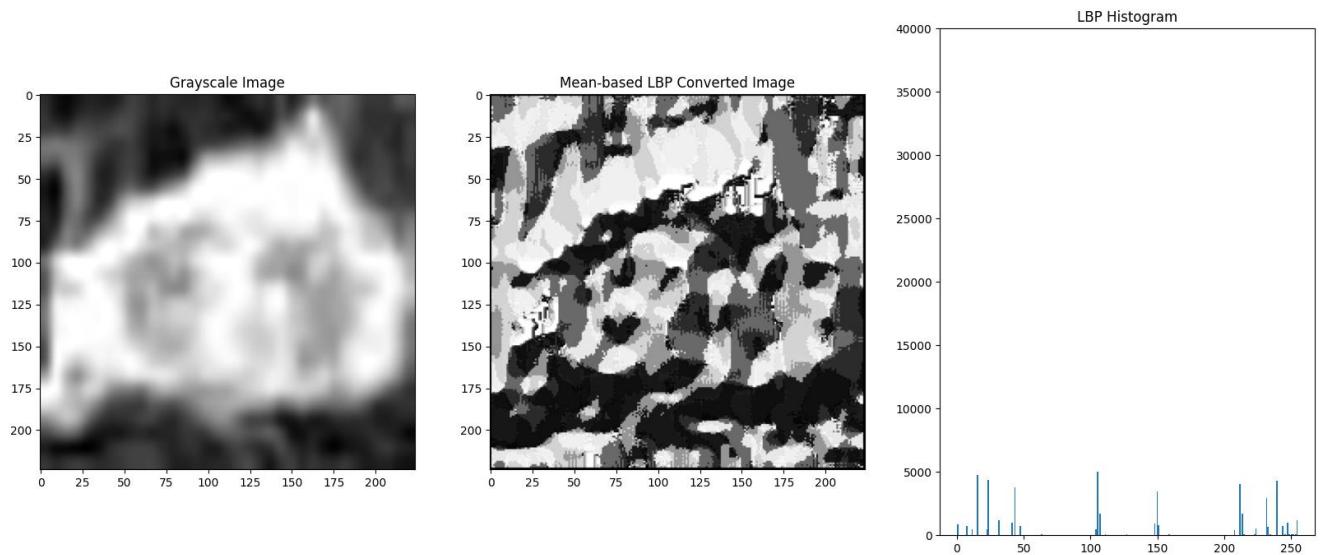
# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("Mean-based LBP Converted Image")

# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_xlim(0, 40000)
lbp = lbp[:-1]

# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")

plt.show()
```

## OUTPUT:



## 6. MEDIAN-BASED LBP CONVERTED IMAGE

### CODE:

```
File Edit View Insert Runtime Tools Help Saving...  
+ Code + Text  
import numpy as np  
import matplotlib.pyplot as plt  
from PIL import Image  
  
# Load the image (Ensure path correctness)  
image_path = '/content/drive/MyDrive infosys_mri_dataset/clasification-roi/train/Malignant/Breast-Ma-2127/SUB7/p-049.jpg'  
image = Image.open(image_path).convert('L')  
image = image.resize((224, 224))  
image_np = np.array(image)  
  
def getLBPimageUsingMedian(gray_image):  
    ...  
    == Input ==  
    gray_image : grayscale image of shape (height, width)  
  
    == Output ==  
    imgLBP : LBP converted image of the same shape using median value  
    ...  
    # Initialize LBP image with zeros  
    imgLBP = np.zeros_like(gray_image)  
    neighbor = 3 # Define a 3x3 neighborhood  
    # Iterate over each pixel (excluding borders)  
    for ih in range(1, gray_image.shape[0] - 1):  
        for iw in range(1, gray_image.shape[1] - 1):  
            # Step 1: Get the 3x3 neighborhood around the pixel  
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window  
            median_value = np.median(img) # Calculate median of the neighborhood  
            # Step 2: Create a binary pattern by thresholding with median value  
            img01 = (img >= median_value).astype(int)  
            # Step 3: Flatten the matrix into a vector (excluding the center)  
            img01_vector = np.delete(img01.flatten(), 4)  
            # Step 4: Convert the binary pattern to a decimal number  
            num = np.dot(img01_vector, 2 ** np.arange(8))  
            # Step 5: Assign this number to the LBP image  
            imgLBP[ih, iw] = num  
  
    return imgLBP
```

✓ 2s completed at 11:09PM



infosys\_mri.ipynb



File Edit View Insert Runtime Tools Help All changes saved



+ Code + Text



{x}



```
# Apply LBP to the single image using median value
imgLBP = getLBPimageUsingMedian(image_np)

# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()

# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))

# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")

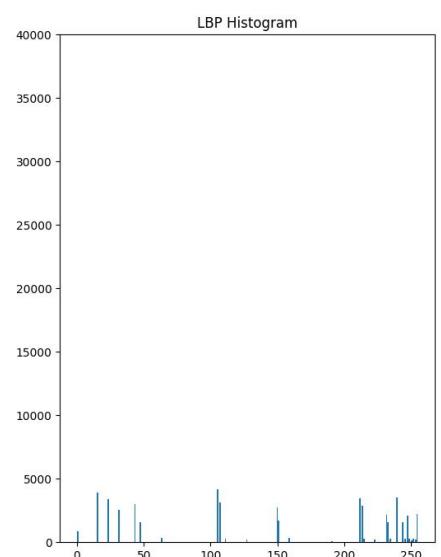
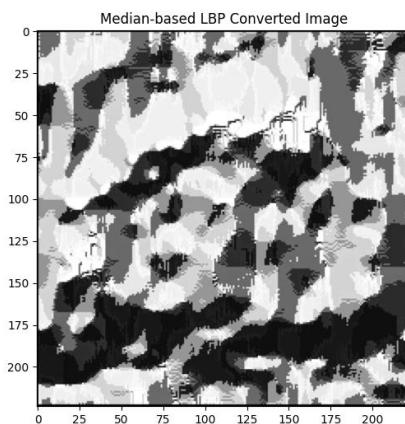
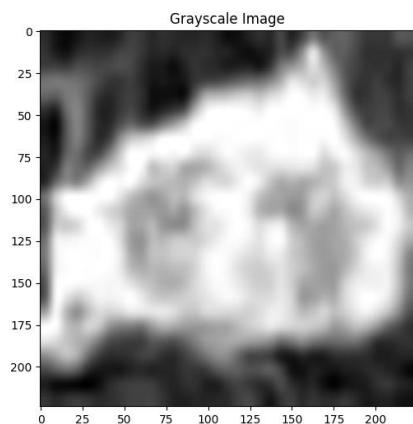
# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("Median-based LBP Converted Image")

# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_xlim(0, 40000)
lbp = lbp[:-1]

# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")

plt.show()
```

## OUTPUT:



## **7. VARIANCE-BASED LBP:**

**CODE:**

The screenshot shows a Jupyter Notebook interface with the file name "infosys\_mri.ipynb". The code cell contains Python code for generating a Variance-based LBP image from a grayscale input image. The code uses numpy and matplotlib.pyplot libraries. It loads an image, converts it to grayscale, and resizes it. Then, it defines a function `getLBPimageUsingVariance` which takes a grayscale image as input and returns a Variance-based LBP image. The function iterates over each pixel, excluding borders, and creates a 3x3 neighborhood around each pixel. It calculates the variance of the neighborhood and creates a binary pattern by thresholding. This pattern is then flattened into a vector, converted to a decimal number, and assigned to the LBP image at the corresponding position. The code cell has a play button icon and a status bar indicating "2s completed at 11:09 PM".

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingVariance(gray_image):
    ...
    == Input ==
    gray_image : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using variance
    ...

    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighbor = 3 # Define a 3x3 neighborhood
    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window
            variance_value = np.var(img) # Calculate variance of the neighborhood
            # Step 2: Create a binary pattern by thresholding with the variance
            img01 = (img >= variance_value).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num

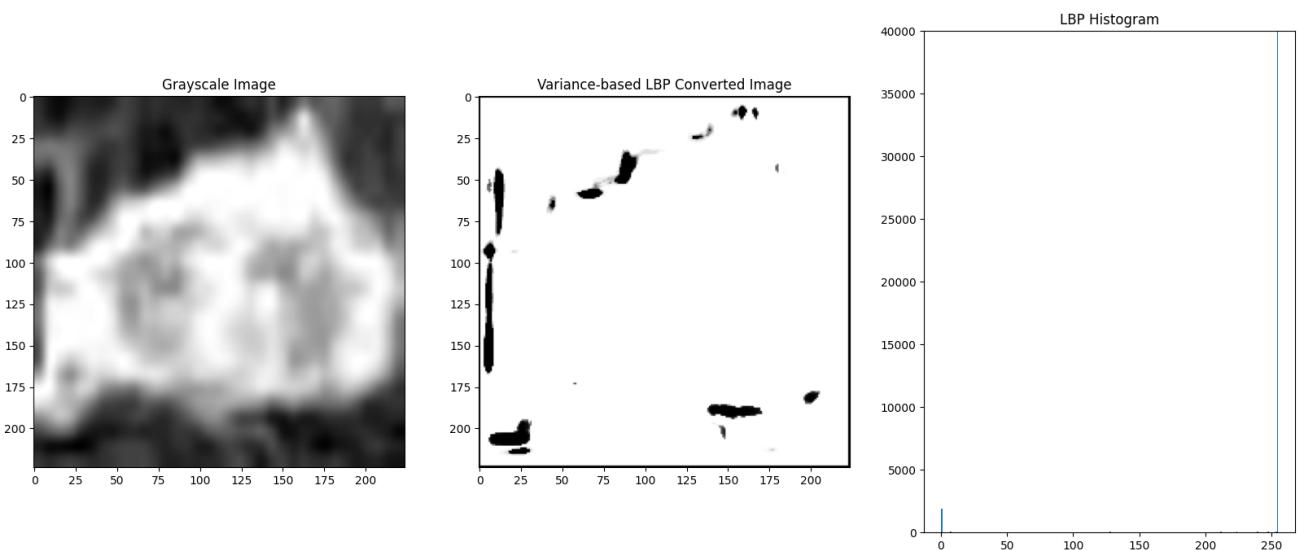
    return imgLBP
```

The screenshot shows a continuation of the Jupyter Notebook with the same file name "infosys\_mri.ipynb". The code cell contains Python code for visualizing and analyzing the LBP histogram. It applies the LBP conversion to the input image, flattens the LBP image for a histogram, and plots the original grayscale image, the LBP converted image, and the histogram. The histogram plot shows LBP values with high frequencies. The code cell has a play button icon.

```
# Apply LBP to the single image using variance
imgLBP = getLBPimageUsingVariance(image_np)
# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()
# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))
# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")
# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("Variance-based LBP Converted Image")
# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_xlim(0, 40000)
lbp = lbp[:-1]
# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")
plt.show()
```

## OUTPUT:

254



## 8. MVM-BASED LBP:

### CODE:

A screenshot of a Jupyter Notebook cell. The code implements the Mean-Variance-Median (MVM) metric for Local Binary Pattern (LBP) conversion. It starts by loading a grayscale image, then defines a function `getLBPimageUsingMVM` that iterates over each pixel (excluding borders). For each pixel, it gets a 3x3 neighborhood, calculates the mean, variance, and median of the neighborhood, and then uses these to calculate a MVM threshold. A binary pattern is created by thresholding the neighborhood with this MVM threshold. Finally, the binary pattern is converted to a decimal number and assigned to the corresponding pixel in the output LBP image. The code is annotated with comments explaining each step of the process.

```
# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingMVM(gray_image):
    """
    == Input ==
    gray_image : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using Mean-Variance-Median (MVM) metric
    """

    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)

    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window
            # Step 2: Calculate Mean, Variance, and Median of the neighborhood
            mean_value = np.mean(img)
            variance_value = np.var(img)
            median_value = np.median(img)
            # Step 3: Calculate MVM threshold
            mvm_threshold = (mean_value + np.sqrt(variance_value) + median_value) / 3
            # Step 4: Create a binary pattern by thresholding with MVM threshold
            img01 = (img >= mvm_threshold).astype(int)
            # Step 5: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 6: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 7: Assign this number to the LBP image
            imgLBP[ih, iw] = num

    return imgLBP
```

infosys\_mri.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

```

# Apply MVM-LBP to the single image
imgLBP = getLBPimageUsingMVM(image_np)
# Flatten the MVM-LBP image for histogram
vecimgLBP = imgLBP.flatten()
# Plot the original image, MVM-LBP image, and histogram
fig = plt.figure(figsize=(20, 8))
# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")
# Plot MVM-based LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("MVM-based LBP Converted Image")
# Plot MVM-LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_xlim(0, 40000)
lbp = lbp[:-1]

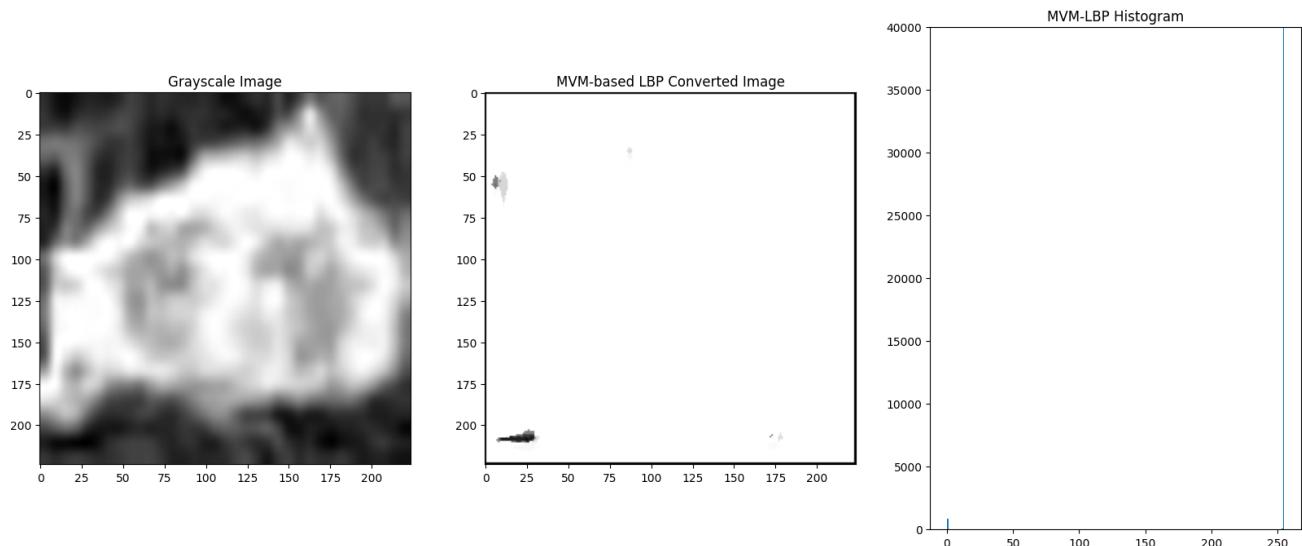
# Print MVM-LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("MVM-LBP Histogram")

plt.show()

```

## OUTPUT:

254



## **9. MVM BASED LBP FOR 5 IMAGES:**

**CODE:**

The screenshot shows a Jupyter Notebook interface with two code cells. The first cell contains the definition of the `getLBPimageUsingMVM` function, which takes a grayscale image as input and returns an LBP converted image using the Mean-Variance-Median (MVM) metric. The second cell contains the main script for processing five images, applying the MVM-LBP method, and plotting the results.

```
[14] def getLBPimageUsingMVM(gray_image):
    ...
    == Input ==
    gray_image : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using Mean-Variance-Median (MVM) metric
    ...
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)

    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2] # 3x3 window
            # Step 2: Calculate Mean, Variance, and Median of the neighborhood
            mean_value = np.mean(img)
            variance_value = np.var(img)
            median_value = np.median(img)
            # Step 3: Calculate MVM threshold
            mvm_threshold = (mean_value + np.sqrt(variance_value) + median_value) / 3
            # Step 4: Create a binary pattern by thresholding with MVM threshold
            img01 = (img >= mvm_threshold).astype(int)
            # Step 5: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 6: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 7: Assign this number to the LBP image
            imgLBP[ih, iw] = num
    return imgLBP
```

The screenshot shows the execution of the main script. It lists the paths of five images, processes them using the `getLBPimageUsingMVM` function, and then plots the original grayscale images, the MVM-based LBP converted images, and their histograms.

```
# List of image paths
image_paths = [
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Benign/BreaDM-Be-1805/SUB1/p-025.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Benign/BreaDM-Be-1805/VIBRANT+C1/p-029.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2117/SUB5/p-085.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2128/SUB8/p-052.jpg'
]

# Create a figure for plotting
fig, axs = plt.subplots(len(image_paths), 3, figsize=(20, 4 * len(image_paths)))

# Process each image
for idx, image_path in enumerate(image_paths):
    # Load and preprocess the image
    image = Image.open(image_path).convert('L')
    image = image.resize((224, 224))
    image_np = np.array(image)

    # Apply MVM-LBP to the single image
    imgLBP = getLBPimageUsingMVM(image_np)
    # Flatten the MVM-LBP image for histogram
    vecimgLBP = imgLBP.flatten()

    # Plot original grayscale image
    axs[idx, 0].imshow(image_np, cmap="gray")
    axs[idx, 0].set_title(f"Grayscale Image {idx + 1}")
    axs[idx, 0].axis('off')

    # Plot MVM-based LBP converted image
    axs[idx, 1].imshow(imgLBP, cmap="gray")
    axs[idx, 1].set_title(f"MVM-based LBP Image {idx + 1}")
    axs[idx, 1].axis('off')

    # Plot MVM-LBP histogram
    freq, lbp, _ = axs[idx, 2].hist(vecimgLBP, bins=256, color='black', alpha=0.7)
    axs[idx, 2].set_xlim(0, 40000)
    lbp = lbp[:-1]
```

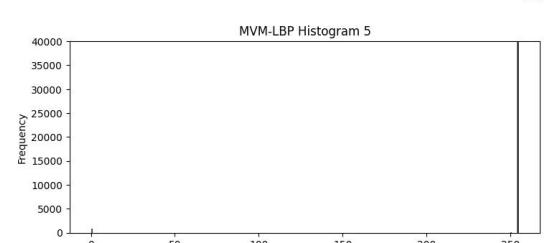
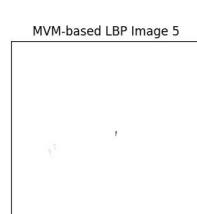
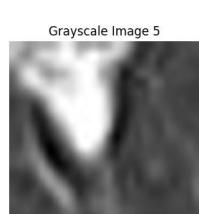
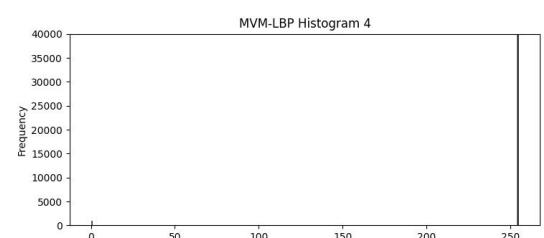
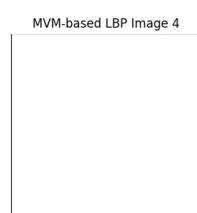
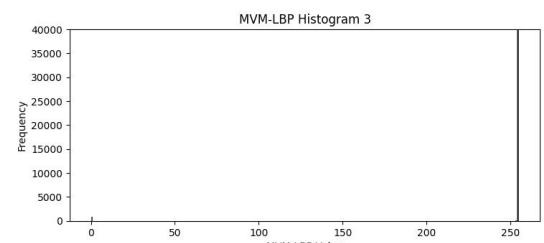
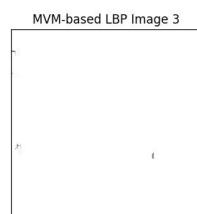
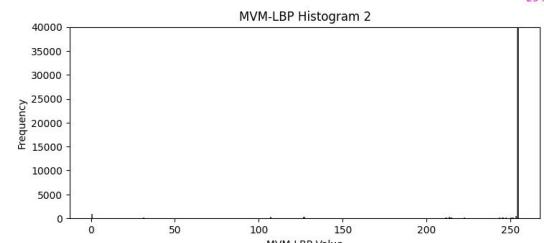
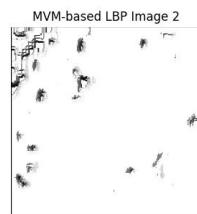
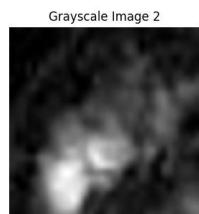
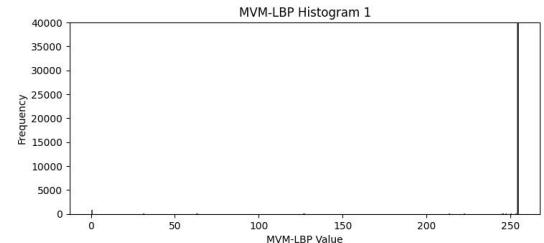
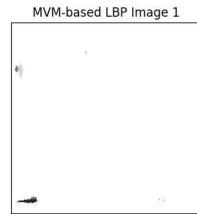
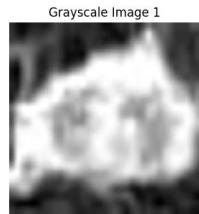


```
# Print MVM-LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    axs[idx, 2].text(x, fr, "{:6.0f}".format(x), color="magenta")

axs[idx, 2].set_title(f"MVM-LBP Histogram {idx + 1}")
axs[idx, 2].set_xlabel("MVM-LBP Value")
axs[idx, 2].set_ylabel("Frequency")

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

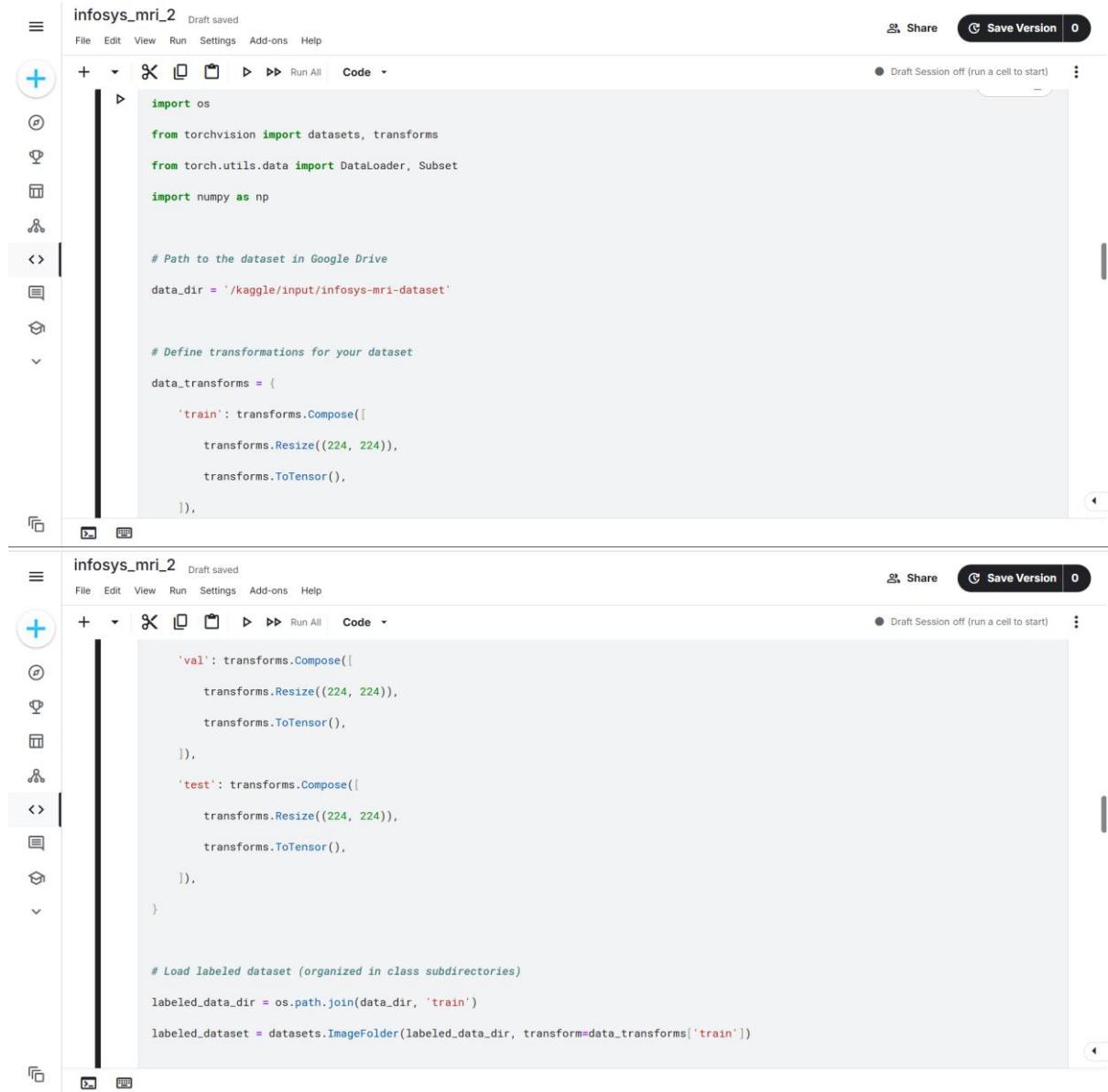
## OUTPUT:



## **POST MVM-LBP CONTINUED PROJECT REPORT: SUBMITTED BY BHAVYA TIWARI**

### **1. Explored GCLM however outputs were not favourable**

### **2. Check for unlabelled data, create data loaders**



The image shows two vertically stacked Jupyter Notebook cells. Both cells have the title "infosys\_mri\_2" and are marked as "Draft saved". The top cell contains the following code:

```
import os
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import numpy as np

# Path to the dataset in Google Drive
data_dir = '/kaggle/input infosys-mri-dataset'

# Define transformations for your dataset
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
    ]),
    'val': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
    ]),
    'test': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
    ]),
}

# Load labeled dataset (organized in class subdirectories)
labeled_data_dir = os.path.join(data_dir, 'train')
labeled_dataset = datasets.ImageFolder(labeled_data_dir, transform=data_transforms['train'])
```

The bottom cell contains the continuation of the code from the top cell:

```
'train': transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
]),

'labeled_dataset = datasets.ImageFolder(labeled_data_dir, transform=data_transforms['train'])
```

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

```
# Extract indices for labeled data

labeled_indices = np.arange(len(labeled_dataset))

X_labeled = Subset(labeled_dataset, labeled_indices) # Features (X) for labeled

y_labeled = [labeled_dataset.targets[i] for i in labeled_indices] # Labels (y) for labeled

# Load unlabeled dataset if it's in a separate folder (no class subfolders)

unlabeled_data_dir = os.path.join(data_dir, 'unlabeled')

if os.path.exists(unlabeled_data_dir):

    unlabeled_dataset = datasets.ImageFolder(unlabeled_data_dir, transform=data_transforms['train'])

    # All images in this dataset are treated as unlabeled

    X_unlabeled = DataLoader(unlabeled_dataset, batch_size=32, shuffle=True)

else:

    print("Unlabeled dataset directory not found.")
```

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

```
# Create data loaders for labeled and unlabeled

train_loader = DataLoader(X_labeled, batch_size=32, shuffle=True)

val_loader = DataLoader(datasets.ImageFolder(os.path.join(data_dir, 'val'), transform=data_transforms['val']), batch_size=32, shuffle=True)

test_loader = DataLoader(datasets.ImageFolder(os.path.join(data_dir, 'test'), transform=data_transforms['test']), batch_size=32, shuffle=False)

# Store all loaders in a dictionary for easier access

dataloaders = {

    'train': train_loader,

    'val': val_loader,

    'test': test_loader,

    'unlabeled': X_unlabeled if 'X_unlabeled' in locals() else None # Check if unlabeled data exists
}
```

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

```
print("Labeled and unlabeled data loaders created successfully.")
```

Unlabeled dataset directory not found.  
Labeled and unlabeled data loaders created successfully.

+ Code + Markdown

### 3. Load VGG-16, RESNET-18,RESNET-50

The screenshot shows a Jupyter Notebook interface with two code cells. The top cell, labeled [13]:, contains Python code to load a pre-trained VGG-16 model from PyTorch's torchvision library. The bottom cell, labeled [15]:, contains Python code to load a pre-trained ResNet-18 model from PyTorch's torchvision library. Both cells show the code being run and the resulting output, which includes a warning about the 'pretrained' parameter being deprecated and the download of the model weights.

```
[13]:  
import torch  
  
import torchvision.models as models  
# Load the VGG-16 model pre-trained on ImageNet  
vgg16 = models.vgg16(pretrained=True)  
# Set the model to evaluation mode  
  
vgg16.eval()  
  
/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.  
    warnings.warn(  
/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or 'None' for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing 'weights=VGG16_Weights.IMAGENET1K_V1'. You can also use 'weights=VGG16_Weights.DEFAULT' to get the most up-to-date weights.  
    warnings.warn(msg)  
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pth  
100%|██████████| 528M/528M [00:02<00:00, 235MB/s]  
  
[13]: VGG(  
    (features): Sequential(  
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU(inplace=True)  
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU(inplace=True)  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (6): ReLU(inplace=True)  
    )  
    (avgpool): AdaptiveAvgPool2d(output_size=1)  
    (classifier): Sequential(  
        (0): Linear(in_features=128, out_features=4096, bias=True)  
        (1): ReLU(inplace=True)  
        (2): Linear(in_features=4096, out_features=4096, bias=True)  
        (3): ReLU(inplace=True)  
        (4): Linear(in_features=4096, out_features=1000, bias=True)  
    )  
)  
  
[15]:  
import torch  
import torchvision.models as models  
  
# For ResNet18  
resnet18 = models.resnet18(pretrained=True)  
resnet18.eval() # Set the model to evaluation mode  
print("ResNet18 Model:")  
print(resnet18)  
  
/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or 'None' for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing 'weights=ResNet18_Weights.IMAGENET1K_V1'. You can also use 'weights=ResNet18_Weights.DEFAULT' to get the most up-to-date weights.  
    warnings.warn(  
Download: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth  
100%|██████████| 44.7M/44.7M [00:00<00:00, 81.7MB/s]  
ResNet18 Model:  
ResNet(  
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)  
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (relu): ReLU(inplace=True)  
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)  
    (layer1): Sequential(  
        (0): BasicBlock(  
            (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
            (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        )  
    )  
)
```

The image shows two separate Jupyter Notebook interfaces side-by-side, both titled "infosys\_mri\_2".

**Top Notebook (Resnet18):**

```
class Resnet18(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet18, self).__init__()
        model_resnet18 = models.resnet18(pretrained=True)
        self.conv1 = model_resnet18.conv1 # convolutional function
        self.bn1 = model_resnet18.bn1 # batch normalization
        self.relu = model_resnet18.relu # relu is your activation function.
        self.maxpool = model_resnet18.maxpool # maxpool is basically taking the biggest value per
                                             # sub_matrix
        self.layer1 = model_resnet18.layer1
        self.layer2 = model_resnet18.layer2
        self.layer3 = model_resnet18.layer3
        self.layer4 = model_resnet18.layer4 # these layers are used for deepening the layers in the architecture which will increase
                                         # the accuracy of the model.

        self.avgpool = model_resnet18.avgpool
        self.features = model_resnet18.fc.in_features
        self.fc = nn.Linear(self.features, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

**Bottom Notebook (Resnet50):**

```
class Resnet50(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet50, self).__init__()
        model_resnet50 = models.resnet50(pretrained=True)
        self.conv1 = model_resnet50.conv1
        self.bn1 = model_resnet50.bn1
        self.relu = model_resnet50.relu
        self.maxpool = model_resnet50.maxpool
        self.layer1 = model_resnet50.layer1
        self.layer2 = model_resnet50.layer2
        self.layer3 = model_resnet50.layer3
        self.layer4 = model_resnet50.layer4
        self.avgpool = model_resnet50.avgpool
        self.features = model_resnet50.fc.in_features
        self.fc = nn.Linear(self.features, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

#### 4. Define early stopping

The image displays three vertically stacked Jupyter Notebook code cells, each titled "infosys\_mri\_2 Draft saved". The interface includes a toolbar with icons for file operations, a code editor, and a status bar indicating "Draft Session off (run a cell to start)".

**Cell 1:**

```
class EarlyStopping:
    def __init__(self, patience=5, verbose=False, delta=0, path='checkpoint.pt', trace_func=print):
        self.patience = patience
        self.verbose = verbose
        self.delta = delta
        self.path = path
        self.trace_func = trace_func

        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
```

**Cell 2:**

```
def __call__(self, val_loss, model):
    score = -val_loss

    # Initialize best_score if not set, and check for improvement
    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
        if self.verbose:
            self.trace_func(f"EarlyStopping counter: {self.counter} out of {self.patience}")
        if self.counter >= self.patience:
            self.early_stop = True
    else:
```

**Cell 3:**

```
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}). Saving model...")
    torch.save(model.state_dict(), self.path)
    self.val_loss_min = val_loss
```

## 5. Define Train, Validation, Test Functions

The screenshot shows a Kaggle Notebook Editor interface with two code cells. The top cell contains the training function, and the bottom cell contains the validation function.

```
epoch = 0
total_epochs = 50
loader = train_loader # Ensure this is a DataLoader instance for training data
criterion = nn.CrossEntropyLoss()
l2_decay = 0.01
lr = 0.01 # Learning rate

def train(epoch, model, num_epochs, loader, criterion, l2_decay):
    learning_rate = max(lr*(0.1**((epoch//10)), 1e-5)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, weight_decay=l2_decay)
    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()

[27]: from sklearn.metrics import roc_curve, auc as compute_auc # Rename the imported 'auc' function
import sklearn.metrics as metrics

def validation(model, val_loader):
    model.eval() # Set model to evaluation mode
    test_loss = 0
    correct = 0
    all_predictions = [] # Store all predictions
    all_targets = [] # Store all targets
    possibilities = None # Store probabilities for AUC

    for data, target in val_loader:
```

Kaggle Notebook Editor | New Tab | kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit

**infosys\_mri\_2** Draft saved

File Edit View Run Settings Add-ons Help Share Save Version 0 Draft Session off (run a cell to start)

+ - Run All Code

```

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 and accumulate them
    pred = val_output.data.max(1)[1]
    all_predictions.extend(pred.cpu().numpy()) # Collect all predictions
    all_targets.extend(target.cpu().numpy()) # Collect all target labels

    # 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)

    # Calculate the number of correct predictions
    correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    # Compute confusion matrix
    cm = metrics.confusion_matrix(all_targets, all_predictions)

```

**infosys\_mri\_2** Draft saved

File Edit View Run Settings Add-ons Help Share Save Version 0 Draft Session off (run a cell to start)

+ - Run All Code

```

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

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(label_onehot.ravel(), possibilities.ravel())
auc_score = compute_auc(fpr, tpr) # Use 'compute_auc' to avoid conflicts

# Average test loss per sample
test_loss /= len(val_loader.dataset)

# Calculate specificity and sensitivity
specificity = 1 - fpr[1] if len(fpr) > 1 else 0
sensitivity = tpr[1] if len(tpr) > 1 else 0

print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(specificity, sensitivity, auc_score))
print('\nTest set: Average loss: {:.2f}, Accuracy: {:.2f}\n'.format(test_loss, 100. * correct / len(val_loader.dataset)))

return test_loss, 100. * correct / len(val_loader.dataset), cm, auc_score

```

[28]: total\_epochs = 50  
lr = 0.01

## 6. VGG-16 model training

The image shows two screenshots of a Kaggle Notebook Editor interface. Both screenshots have a green header bar with the title 'Kaggle Notebook Editor' and a tab labeled 'kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit'. The top screenshot shows a code cell numbered [29]:

```
import os
from IPython.display import FileLink

# Model training
model.to(device)

best_accuracy = 0
early_stop = EarlyStopping(patience=20, verbose=True)

project_name = 'tumor_classification'
model_name = 'vgg16'

# Set Kaggle working directory
os.chdir(r'/kaggle/working')

for epoch in range(1, total_epochs + 1):
    # Training step
    train(epoch, model, total_epochs, train_loader, criterion, lr_decay)

    # Validation step
    with torch.no_grad():
        test_loss, accuracy, cm, auc = validation(model, val_loader)

    # Handle model state for single/multiple GPUs
    model.state_dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel) else model.state_dict()

    # Save directory for models
```

The bottom screenshot shows the continuation of the code in the same cell:

```
# Save directory for models
model_save_dir = os.path.join('model', project_name, model_name)
if not os.path.exists(model_save_dir):
    os.makedirs(model_save_dir)

# Early stopping check
early_stop(test_loss, model)

# Save the best model
if auc > best_accuracy:
    best_accuracy = auc
    model_save_path = os.path.join(model_save_dir, f'{model_name}_epoch_{epoch}.pth')
    torch.save(model.state_dict, model_save_path, _use_new_zipfile_serialization=False)
    print(f"Model saved at: {os.path.abspath(model_save_path)}")

# Generate download link for Kaggle
print("Generating download link for the saved model...")
display(FileLink(model_save_path))

# Stop training if early stopping is triggered
if early_stop.early_stop:
    print("Early stopping")
    break
```

At the bottom of the code cell, there is a red status bar displaying the output of the last command: 'Epoch 2/50: 100%|██████████| 639/639 [04:15<00:00, 2.51batch/s] train accuracy: 74.03836822509766%

Kaggle Notebook Editor    New Tab    kaggle.com/code/bhavyatiwari2409infosys-mri-2/edit

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

Code

```
train accuracy: 99.9559555053711%
Specificity: 1.0000, Sensitivity: 0.1589, AUC: 0.9301

Test set: Average loss: 0.5684, Accuracy: 83.91%

EarlyStopping counter: 17 out of 20
Epoch 23/50: 100% | 639/639 [03:10<0:00, 3.35batch/s]
train accuracy: 99.9559555053711%
Specificity: 1.0000, Sensitivity: 0.1614, AUC: 0.9297

Test set: Average loss: 0.5778, Accuracy: 84.26%

EarlyStopping counter: 18 out of 20
Epoch 24/50: 100% | 639/639 [03:10<0:00, 3.35batch/s]
train accuracy: 99.9753253173828%
Specificity: 1.0000, Sensitivity: 0.1649, AUC: 0.9286

Test set: Average loss: 0.5934, Accuracy: 84.06%

EarlyStopping counter: 19 out of 20
Epoch 25/50: 100% | 639/639 [03:11<0:00, 3.33batch/s]
train accuracy: 99.9559555053711%
Specificity: 1.0000, Sensitivity: 0.1669, AUC: 0.9311

Test set: Average loss: 0.5668, Accuracy: 84.36%

EarlyStopping counter: 20 out of 20
Early stopping
```

## IMPROVEMENTS TO BE DONE: Handle overfitting

Kaggle Notebook Editor    New Tab    kaggle.com/code/bhavyatiwari2409infosys-mri-2/edit

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

Code

```
label_onehot = np.eye(num_classes)[np.array(true_labels).astype(int).tolist()]
fpr, tpr, thresholds = roc_curve(label_onehot.ravel(), possibilities.ravel())
auc_value = roc_auc_score(label_onehot, possibilities, average="macro")

# Print Specificity, Sensitivity, and AUC
print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(1 - fpr[0], tpr[0], auc_value))

# Print Final Results
print('\nTest Set: Average loss: {:.4f}, Accuracy: {} / {} ({:.2f}%)'.format(
    test_loss, correct, len_test_loader, accuracy))

precision recall f1-score support
benign      0.4451   0.2972   0.3564    1938
malignant    0.7549   0.8539   0.8013    4913

accuracy     0.6000   0.5755   0.5789    6851
macro avg    0.6000   0.5755   0.5789    6851
weighted avg 0.6673   0.6964   0.6755    6851

Confusion Matrix:
[[ 576 1362]
 [ 718 4195]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.6567

Test Set: Average loss: 2.0222, Accuracy: 4771/6851 (69.64%)
```

## 7. RESNET-18 training

The image shows two side-by-side screenshots of the Kaggle Notebook Editor interface. Both windows have the title "infosys\_mri\_2" and show a "Draft saved" message. The top window displays code for model training, including setting up the device, defining early stopping, and validating the model. The bottom window displays code for saving the trained model to a directory, ensuring it can run on multiple GPUs, and performing early stopping based on AUC.

```
[35]: # model training
model.to(device) # here device is cuda
best_accuracy = 0
early_stop = EarlyStopping(patience=20, verbose=True)
project_name = 'tumor_classification'
model_name = 'resnet18'

# we will be using epochs. epochs will be defined in another code block.

for epoch in range(1, total_epochs + 1):

    #train(epoch, model)#train(epoch, total_epochs, train_loader, criterion, l2_decay, lr)

    train(epoch, model, total_epochs, train_loader, criterion, l2_decay)

    with torch.no_grad():

        #test_loss, auc = validation(model , val_loader)
        test_loss, accuracy, cm, auc = validation(model, val_loader)
```

```
# making sure that the model can run on multiple GPUs

dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel) else model.state_dict()

model_save_dir = os.path.join('model', project_name, model_name)

if not os.path.exists(model_save_dir):
    os.makedirs(model_save_dir)

early_stop(test_loss, model)

if auc > best_accuracy:
    best_accuracy = auc

#torch.save(os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'), use_new_zिंfile_serialization=False)
```

The screenshot displays two separate runs of a Python script in the Kaggle Notebook Editor. Both runs are titled "infosys\_mri\_2" and show the same code structure:

```
#torch.save(os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'), _use_new_zipfile_serialization=False)
torch.save(dict, os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'), _use_new_zipfile_serialization=False)

if early_stop.early_stop:
    print("Early stopping")
    break
```

The output logs from both runs are identical, showing training progress and performance metrics:

Epoch 2/50: 100% [██████████| 639/639 [01:27<00:00, 7.33batch/s]  
train accuracy: 87.36908721923828%  
Specificity: 1.0000, Sensitivity: 0.0211, AUC: 0.8950  
Test set: Average loss: 0.3955, Accuracy: 78.13%  
Validation loss decreased (inf --> 0.395481). Saving model...  
Epoch 3/50: 100% [██████████| 639/639 [01:08<00:00, 9.31batch/s]  
train accuracy: 94.3378677368164%  
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8824  
Test set: Average loss: 0.4128, Accuracy: 76.27%  
EarlyStopping counter: 1 out of 20  
Epoch 4/50: 100% [██████████| 639/639 [01:09<00:00, 9.16batch/s]

In the first run (top), the script exits early at epoch 4 because the validation loss did not decrease. In the second run (bottom), the EarlyStopping counter reaches 20, causing the script to exit at epoch 20.

## IMPROVEMENTS TO BE DONE: Handle overfitting

## 8. RESNET-50 Training

The image shows two screenshots of the Kaggle Notebook Editor interface, each displaying a code cell for training a ResNet-50 model.

**Code Cell 1:**

```
[41]: # model training
model.to(device) # here device is cuda
best_accuracy = 0
early_stop = EarlyStopping(patience=20, verbose=True)
project_name = 'tumor_classification'
model_name = 'resnet50'
# we will be using epochs. epochs will be defined in another code block.
for epoch in range(1, total_epochs + 1):

    #train(epoch, model)#train(epoch, total_epochs, train_loader, criterion, l2_decay, lr)
    train(epoch, model, total_epochs, train_loader, criterion, l2_decay)

    with torch.no_grad():

        #test_loss, auc = validation(model , val_loader)
        test_loss, accuracy, cm, auc = validation(model, val_loader)

# making sure that the model can run on multiple GPUs
```

**Code Cell 2:**

```
dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel) else model.state_dict()

model_save_dir = os.path.join('model', project_name, model_name)

if not os.path.exists(model_save_dir):
    os.makedirs(model_save_dir)

early_stop(test_loss, model)

if auc > best_accuracy:
    best_accuracy = auc

#torch.save(os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'), _use_new_zipfile_serialization=False)
torch.save(dict, os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'), _use_new_zipfile_serialization=False)
print(f'{model_name}_{epoch}')
```

Kaggle Notebook Editor    New Tab    kaggle.com/code/bhavyatiwari2409infosys-mri-2/edit

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

```
if early_stop.early_stop:  
    print("Early stopping")  
    break
```

Epoch 2/50: 100%|██████████| 639/639 [02:33<00:00, 4.16batch/s]  
train accuracy: 88.04933166503906%  
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8566  
Test set: Average loss: 0.4940, Accuracy: 76.82%  
Validation loss decreased (inf --> 0.494037). Saving model...  
resnet50\_1  
Epoch 3/50: 100%|██████████| 639/639 [02:23<00:00, 4.45batch/s]  
train accuracy: 92.71312713623047%  
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8977  
Test set: Average loss: 0.4179, Accuracy: 80.19%  
Validation loss decreased (0.494037 --> 0.417853). Saving model...  
resnet50\_2  
Epoch 4/50: 100%|██████████| 639/639 [02:25<00:00, 4.40batch/s]  
train accuracy: 92.99207305908205%  
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.7250  
Test set: Average loss: 0.7502, Accuracy: 67.27%

Kaggle Notebook Editor    New Tab    kaggle.com/code/bhavyatiwari2409infosys-mri-2/edit

infosys\_mri\_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Draft Session off (run a cell to start)

```
train accuracy: 99.9902114868164%  
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8600  
Test set: Average loss: 0.5754, Accuracy: 75.57%  
EarlyStopping counter: 17 out of 20  
Epoch 23/50: 100%|██████████| 639/639 [02:21<00:00, 4.53batch/s]  
train accuracy: 99.9902114868164%  
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8749  
Test set: Average loss: 0.5284, Accuracy: 77.58%  
EarlyStopping counter: 18 out of 20  
Epoch 24/50: 100%|██████████| 639/639 [02:20<00:00, 4.54batch/s]  
train accuracy: 99.99510955810547%  
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8667  
Test set: Average loss: 0.5519, Accuracy: 77.07%  
EarlyStopping counter: 19 out of 20  
Epoch 25/50: 100%|██████████| 639/639 [02:21<00:00, 4.51batch/s]  
train accuracy: 99.99510955810547%  
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8735  
Test set: Average loss: 0.5309, Accuracy: 77.53%  
EarlyStopping counter: 20 out of 20  
Early stopping
```

The screenshot shows a Kaggle Notebook Editor window. The title bar says "Kaggle Notebook Editor" and "New Tab". The URL is "kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit". The notebook is titled "infosys\_mri\_2" and is in "Draft saved" mode. The toolbar includes icons for file operations like New, Open, Save, Run, Settings, Add-ons, Help, and a "Code" dropdown. A status message at the top right says "Draft Session off (run a cell to start)". The main area contains a code cell [45]:

```
accuracy, test_loss, auc_value = test(model, test_loader)
```

Below the code, the output shows:

```
Classification Report:
precision    recall   f1-score   support
benign       0.5742   0.2853   0.3812    1938
malignant    0.7648   0.9165   0.8338    4913

accuracy      0.7380   6851
macro avg     0.6695   0.6009   0.6075    6851
weighted avg  0.7109   0.7380   0.7058    6851

Confusion Matrix:
[[ 553 1385]
 [ 410 4503]]

Specificity: 1.0000, Sensitivity: 0.0002, AUC: 0.7085
Test set: Average loss: 0.8375, Accuracy: 5056/6851 (73.80%)
```

## 9. Updates in code, changing hyperparameters and adding dropout layers

## 10. Running live demo on gradio

