INFOSYS SPRINGBOARD

NAME: Katta Sai Rutvik

PROJECT TITLE: TumorTrace: MRI-Based AI for Breast Cancer Detection

MENTOR: Anurag Sista

Index

CHAPTER 1

INTRODUCTION

1.1 Introduction to TumorTrace: MRI-Based AI for Breast Cancer Detection

About Breast Cancer:

Breast cancer is a condition in which cells in the breast grow uncontrollably, forming tumors that can be benign (non-cancerous) or malignant (cancerous). It is the most common cancer in women but can also affect men, though less frequently. The disease primarily arises in the ducts (ductal carcinoma) or lobules (lobular carcinoma), which are the milk-producing glands.

The exact causes of breast cancer are not fully understood, but several risk factors have been identified. These include genetic mutations, such as BRCA1 and BRCA2, a family history of breast cancer, hormonal influences, and lifestyle factors like obesity, alcohol consumption, and lack of physical activity. Age and gender are significant risk factors, with the likelihood of developing breast cancer increasing as women grow older.

Treatment for breast cancer varies depending on the type, stage, and overall health of the patient. Options include surgery (lumpectomy or mastectomy), radiation therapy, chemotherapy, hormone therapy, and targeted therapies. Early detection through regular screening, such as mammograms, can significantly improve outcomes by identifying the disease in its early stages.

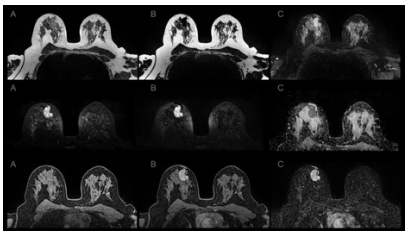


Fig 1.1.1

Effects of Cancer on Our Lives:

Physical Impact

Breast cancer and its treatments—such as surgery, chemotherapy, radiation, and hormone therapy—can lead to fatigue, pain, lymphedema (swelling in the arm or chest), hair loss, and changes in body image. Hormonal treatments may cause menopause-like symptoms, such as hot flashes, weight gain, and bone thinning. Long-term side effects, including the risk of recurrence, can affect daily functioning and overall health.

Emotional and Psychological Effects

A diagnosis of breast cancer often brings emotional distress, anxiety, depression, and fear of recurrence. Many individuals struggle with a sense of loss, changes in self-esteem, and the psychological adjustment to altered body image after surgery or other treatments. The uncertainty about the future can lead to prolonged mental health challenges.

Social and Relationship Changes

Breast cancer can impact relationships with family, friends, and partners. Physical changes and emotional stress may affect intimacy and communication. Support from loved ones is critical, but patients may also experience feelings of isolation or guilt for being a "burden."

1.2 Importance of MRI for cancer

MRI (Magnetic Resonance Imaging) is pivotal in breast tumor detection and classification due to its superior ability to visualize soft tissues. Unlike mammograms or ultrasounds, MRI produces detailed cross-sectional and 3D images, enabling a comprehensive examination of the breast structure. This capability is especially important for detecting abnormalities in dense breast tissues, where other imaging methods may fall short.

Another significant advantage of MRI is its high sensitivity to both benign and malignant tumors. It excels at identifying small or early-stage tumors, enhancing the chances of early diagnosis and successful treatment. This sensitivity makes MRI a preferred choice for high-risk patients or cases where other diagnostic tools yield inconclusive results

MRI is also highly effective in differentiating between tumor types. The contrast in MRI images highlights differences in tissue properties, aiding in distinguishing benign lesions from malignant ones. Techniques like contrast-enhanced MRI provide even greater clarity, offering valuable insights for accurate diagnosis and treatment planning.

1.3 Artificial Intelligence and its Role for cancer:

Artificial Intelligence (AI) has revolutionized medical imaging, including the detection and classification of breast tumors. AI algorithms, especially those based on deep learning, excel at analyzing vast amounts of imaging data with precision, speed, and consistency. These capabilities are particularly valuable in detecting abnormalities in MRI scans, where subtle differences in tissue structures can indicate the presence of tumors.

Deep learning models, such as convolutional neural networks (CNNs), have been widely adopted in breast tumor detection. These models learn from labeled datasets to identify patterns and features that may be too subtle for human eyes. For instance, they can differentiate between benign and malignant tumors with high accuracy, reducing the chances of misdiagnosis. By automating the detection process, AI also alleviates the workload of radiologists, enabling them to focus on complex cases.

Another significant role of AI in tumor detection is its ability to provide quantitative insights. AI algorithms can calculate metrics such as tumor size, shape, and volume, which are critical for staging and treatment planning. These quantitative assessments not only enhance diagnostic accuracy but also allow for personalized treatment approaches, improving patient outcomes.

AI's integration with MRI technology is also advancing the field of early detection. By analyzing historical and real-time imaging data, AI systems can identify tumors at their nascent stages, even before symptoms appear. This capability is crucial for improving survival rates, as early-stage cancers are often more treatable.

Moreover, AI enhances the efficiency of MRI analysis through automated workflows. Tasks such as image preprocessing, segmentation, and classification are streamlined, significantly reducing the time required for diagnosis. This efficiency is particularly valuable in resource-constrained settings, where radiologist expertise may be limited.

1.4 Objective of the Project

The increasing prevalence of breast cancer globally underscores the need for effective diagnostic tools that can complement traditional methods. While MRI imaging is one of the most reliable modalities for detecting breast abnormalities due to its high sensitivity, analyzing these images manually is time-intensive and prone to human error. This project addresses this challenge by integrating artificial intelligence, specifically deep learning, to automate and enhance the detection process. By doing so, the system can act as a second opinion for radiologists, offering reliable predictions and reducing the likelihood of misdiagnoses.

Deep learning algorithms excel at identifying subtle patterns in medical imaging that may not be immediately apparent to the human eye. This project uses state-of-the-art neural networks trained on MRI datasets to classify tumors as benign or malignant with high accuracy. The integration of techniques such as data augmentation, feature extraction, and model optimization ensures that the system is robust and generalizes well to unseen data. Such advancements signify a paradigm shift in healthcare, where AI not only supports but also augments human expertise.

CHAPTER 2

Data and Methodology

First used colab, later shifted to kaggle notebook for more usage of GPU

2.1 About our Dataset

The dataset used in this project is specifically curated for breast tumor detection and classification. It is organized into three distinct folders: **train**, **test**, and **validation**, ensuring a clear separation of data for model training, evaluation, and fine-tuning. The dataset consists of MRI images labeled into two classes: **benign** and **malignant**, representing non-cancerous and cancerous tumors, respectively. Below are the specifics of the dataset:

There are few subfolders inside

Structure:

1)Train Folder:

- Path: /content/clasification-roi/train
- Purpose: Used to train the deep learning model by providing sufficient examples for learning patterns and features of both benign and malignant tumors.
- Examples: /content/clasification-roi/train/Benign/BreaDM-Be-1801/SUB1/p-032.jpg

2)Test Folder:

- Path: /content/clasification-roi/test
- Purpose: Used to evaluate the model's performance after training. Ensures the model generalizes well to unseen data.
- Examples: /content/clasification-roi/test/Malignant/BreaDM-Ma-1801/SUB1/p- 041.jpg

3)Validation Folder:

- Path: /content/clasification-roi/val
- Purpose: Used for hyperparameter tuning and monitoring the model's performance during training to avoid overfitting.
- Examples: /content/clasification-roi/val/Malignant/BreaDM-Ma-2029/VIBRANT+C1/p-045.jpg

2.2 Data Preprocessing

Overview:

Data Processing Steps

Data processing is a critical phase in the project that ensures the model receives consistent and meaningful input, maximizing its learning potential. Here is an explanation of each step involved

Steps in Data Preprocessing:

[2]:
```
import os
from PIL import Image
import torch
```

[3]:
```
import os

# List the contents of the 'trace-tumor-notebook' directory
extracted_dir = '/kaggle/input/trace-tumor-notebook/'
print(os.listdir(extracted_dir))
```

```
['val', 'test', 'train']
```

[4]:
```
# Paths to the train, val, and test directories
train_dir = os.path.join(extracted_dir, 'train')
val_dir = os.path.join(extracted_dir, 'val')
test_dir = os.path.join(extracted_dir, 'test')

# List contents of the train, val, and test directories
print(f"Train directory contents: {os.listdir(train_dir)}")
print(f"Validation directory contents: {os.listdir(val_dir)}")
print(f"Test directory contents: {os.listdir(test_dir)}")
```

```
Train directory contents: ['Benign', 'Malignant']
Validation directory contents: ['Benign', 'Malignant']
Test directory contents: ['Benign', 'Malignant']
```

2.1. Data Loading

Purpose: To systematically load and organize MRI images for training, validation, and testing.

The dataset is divided into three main subsets:

•**Training set**: Used for learning patterns.

•**Validation set**: Used for model tuning and hyperparameter adjustment.

•**Test set**: Used for final performance evaluation.

To efficiently load and manage the dataset for the breast tumor detection project, a custom implementation of PyTorch's Dataset module was developed. This custom dataset class handles the hierarchical structure of the dataset, associates images with their respective labels, and ensures compatibility with PyTorch's DataLoader.

Custom Dataset Class

The CustomImageDataset class is designed to dynamically load image paths and their corresponding labels from the dataset's directory structure.

Key Features

•**Recursive Directory Traversal**: It traverses the dataset's root directory, identifying image files across subdirectories.

•**Label Assignment**: Labels are automatically assigned based on the subdirectory names.

•**Supported Formats**: Handles common image file extensions, including .jpg, .jpeg, .png, and .bmp.

Implementation Details

1.**Initialization (**__init__**)**:

•Takes the dataset's root directory (root_dir) as input.

•Iterates through the directory structure using a helper function, find_images, to populate:

2.**Helper Function (**find_images**)**:

•Recursively scans subdirectories.

•Filters files by valid image extensions.

3.**Length Method (**__len__**)**:

•Returns the total number of images in the dataset.

4.**Get Item Method (**__getitem__**)**:

•Retrieves an image and its label based on the index.

•Ensures all images are in RGB format.

```
print(f"Number of training images: {len(train_dataset)}")
print(f"Number of validation images: {len(val_dataset)}")
print(f"Number of test images: {len(test_dataset)}")
```

```
Number of training images: 20434
Number of validation images: 1989
Number of test images: 6851
```

```
from collections import Counter
image_sizes = [Image.open(img_path).size for img_path in train_dataset.image_paths]
print(Counter(image_sizes))
```

```
Counter({(27, 26): 527, (26, 21): 510, (20, 24): 442, (27, 24): 374, (34, 34): 357, (43, 43): 340, (46, 37): 323, (39, 66): 323, (27, 28): 323, (34, 32): 323, (16, 14): 272, (24, 32): 272, (15, 14): 272, (44, 75): 272, (33, 31): 272, (14, 12): 255, (29, 27): 255, (25, 29): 255, (25, 30): 255, (56, 52): 238, (21, 24): 238, (25, 18): 238, (28, 18): 238, (28, 30): 238, (33, 37): 238, (17, 19): 238, (17, 14): 221, (16, 13): 221, (44, 36): 221, (42, 39): 221, (41, 61): 221, (49, 39): 221, (49, 49): 221, (30, 25): 221, (24, 26): 204, (19, 21): 204, (25, 32): 204, (15, 15): 187, (32, 38): 187, (20, 18): 187, (26, 30): 187, (23, 21): 187, (17, 15): 170, (20, 26): 170, (28, 28): 170, (35, 26): 170, (18, 18): 170, (36, 34): 170, (37, 40): 170, (13, 12): 153, (34, 27): 153, (19, 19): 153, (23, 18): 153, (30, 27): 153, (17, 17): 153, (12, 14): 153, (23, 42): 153, (35, 32): 153, (31, 20): 136, (23, 28): 136, (28, 33): 136, (24, 30): 136, (25, 22): 136, (24, 34): 136, (32, 36): 136, (26, 20): 136, (13, 11): 119, (36, 26): 119, (21, 48): 119, (33, 17): 119, (24, 23): 119, (20, 27): 119, (34, 24): 119, (20, 22): 119, (35, 23): 119, (31, 23): 119, (32, 34): 102, (29, 29): 102, (13, 13): 102, (31, 29): 102, (33, 35): 102, (45, 42): 102, (22, 23): 102, (22, 22): 102, (25, 24): 102, (17, 18): 102, (23, 26): 102, (11, 10): 102, (24, 21): 102, (34, 22): 102, (28, 41): 102, (18, 13): 102, (22, 17): 85, (19, 22): 85, (10, 13): 85, (30, 28): 85, (27, 32): 85, (26, 25): 85, (21, 19): 85, (14, 10): 85, (26, 29): 85, (22, 19): 85, (17, 29): 85, (25, 21): 85, (23, 27): 85, (21, 30): 68, (16, 18): 68, (26, 37): 68, (20, 23): 68, (14, 15): 68, (15, 19): 68, (13, 16): 68, (13, 15): 68, (18, 17): 68, (18, 16): 68, (18, 12): 68, (17, 20): 68, (19, 25): 68, (21, 14): 68, (16, 11): 51, (20, 11): 51, (15, 20): 51, (12, 12): 51, (18, 19): 51, (11, 11): 51, (11, 13): 51, (9, 7): 51, (16, 19): 51, (15, 13): 51, (19, 16): 51, (14, 14): 34, (24, 41): 34, (19, 14): 34, (16, 16): 34})
```

[5]:

```
from torch.utils.data import Dataset

class CustomImageDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.image_paths = []
        self.labels = []
        valid_extensions = ('.jpg', '.jpeg', '.png', '.bmp')


        def find_images(path, label):
            for entry in os.listdir(path):
                entry_path = os.path.join(path, entry)
                if os.path.isdir(entry_path):
                    find_images(entry_path, label)
                elif os.path.isfile(entry_path) and entry_path.lower().endswith(valid_extensions):
                    self.image_paths.append(entry_path)
                    self.labels.append(label)


        for class_dir in os.listdir(root_dir):
            class_path = os.path.join(root_dir, class_dir)
            if os.path.isdir(class_path):
                find_images(class_path, class_dir)

        #  Print loaded image paths
        print("Loaded image paths:")
        for path in self.image_paths:
            print(path)

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB')  # Ensure the image is in RGB format
        return image, self.labels[idx]

#  datasets without transformations
train_dataset = CustomImageDataset(root_dir='/kaggle/input/trace-tumor-notebook/train')
val_dataset = CustomImageDataset(root_dir='/kaggle/input/trace-tumor-notebook/val')
test_dataset = CustomImageDataset(root_dir='/kaggle/input/trace-tumor-notebook/test')
```

```
Loaded image paths:
/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1820/SUB3/p-042.jpg
/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1820/SUB3/p-040.jpg
/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1820/SUB3/p-039.jpg
```

2.2 Visualizing Dataset by Class

This section focuses on visualizing images from specific classes in the dataset to better understand their visual characteristics. By isolating and displaying a set number of images for each class (e.g., *Benign* and *Malignant*), we can verify the dataset's correctness and observe any distinctive features.

Key Features

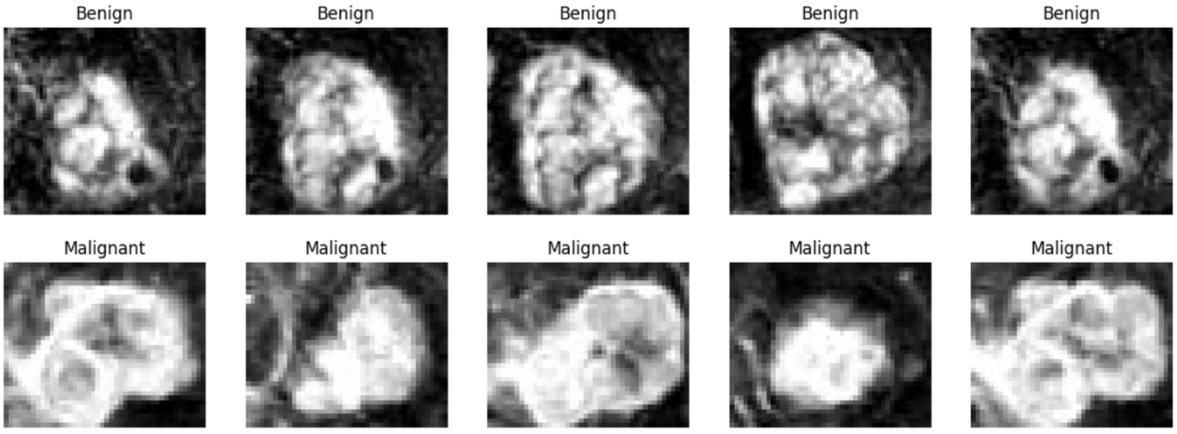
•**Class-Specific Visualization:** Displays images for a specific class selected by the user.

•**Flexible Selection:** Allows specifying the number of images to display.

•**Adaptability to Grayscale or RGB:** Handles both grayscale and color images seamlessly.

•**Interactive Understanding:** Helps in identifying visual patterns or anomalies within each class.

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

def show_images_by_class(dataset, class_name, num_images=5):
    plt.figure(figsize=(15, 5))
    count = 0

    for i in range(len(dataset)):
        img, label = dataset[i]

        # PIL Image to NumPy
        img = np.array(img) / 255.0
        if label == class_name:
            plt.subplot(1, num_images, count + 1)

            if img.ndim == 3:  # (H, W, C)
                plt.imshow(img)
            elif img.ndim == 2:  # (H, W)
                plt.imshow(img, cmap='gray')
            plt.title(label)
            plt.axis('off')
            count += 1

        if count == num_images:
            break

    plt.show()

# first 5 images from the Benign class
show_images_by_class(train_dataset, class_name='Benign', num_images=5)

#  first 5 images from the Malignant class
show_images_by_class(train_dataset, class_name='Malignant', num_images=5)
```



2.3 Counting Images by Class

In this section, we focus on counting the number of images belonging to each class (Benign or Malignant) in the dataset. This helps in understanding the class distribution, which is essential for tasks such as data balancing or identifying any potential bias in the dataset.

```python
from collections import Counter

def count_images_by_class(dataset):
    class_counter = Counter()

    for img_path in dataset.image_paths:
        if 'Malignant' in img_path:
            class_counter['Malignant'] += 1
        elif 'Benign' in img_path:
            class_counter['Benign'] += 1

    return class_counter
class_counts = count_images_by_class(test_dataset)
print(f"Total Benign Images: {class_counts['Benign']}")
print(f"Total Malignant Images: {class_counts['Malignant']}")
```

```
Total Benign Images: 1938
Total Malignant Images: 4913
```

+ Code + Markdown

11]:

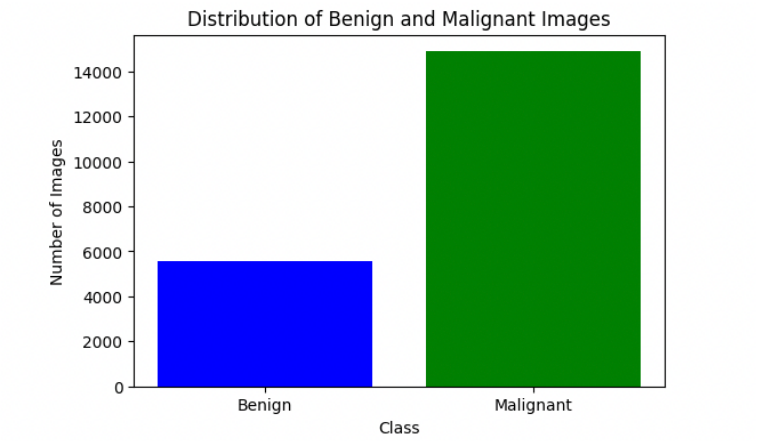
```python
from collections import Counter

def count_images_by_class(dataset):
    class_counter = Counter()

    for img_path in dataset.image_paths:
        if 'Malignant' in img_path:
            class_counter['Malignant'] += 1
        elif 'Benign' in img_path:
            class_counter['Benign'] += 1

    return class_counter
class_counts = count_images_by_class(val_dataset)
print(f"Total Benign Images: {class_counts['Benign']}")
print(f"Total Malignant Images: {class_counts['Malignant']}")
```

```
Total Benign Images: 408
Total Malignant Images: 1581
```

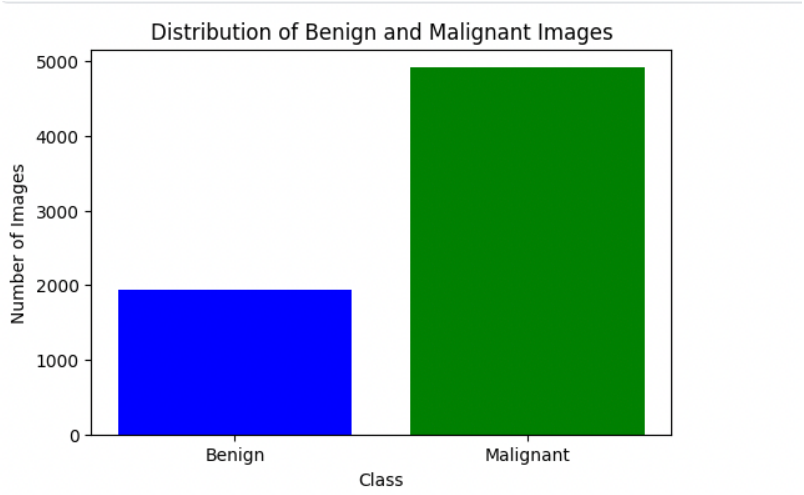
12]:

```python
class_counts = count_images_by_class(train_dataset)


def plot_class_distribution(class_counts):
    classes = list(class_counts.keys())
    counts = list(class_counts.values())

    plt.figure(figsize=(6, 4))
    plt.bar(classes, counts, color=['blue', 'green'])
    plt.xlabel('Class')
    plt.ylabel('Number of Images')
    plt.title('Distribution of Benign and Malignant Images')
    plt.show()


plot_class_distribution(class_counts)
```

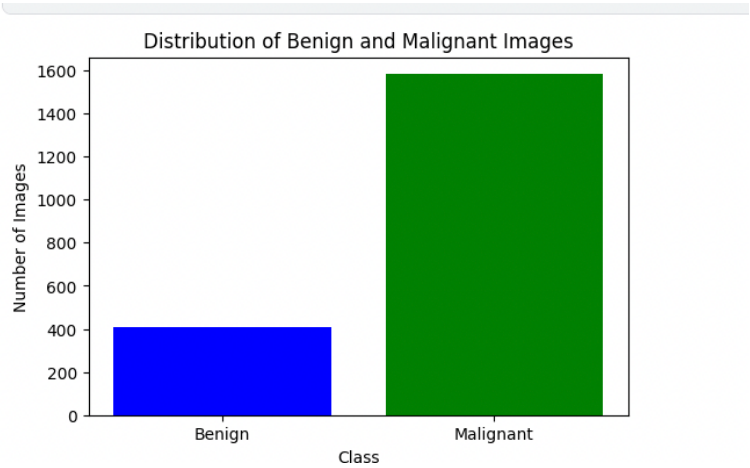
For train



For test



For validation



2.4 Data Loading and Transformation

To efficiently load, transform, and manage the dataset for training a breast tumor detection model, we have developed a custom dataset class using PyTorch's Dataset and DataLoader. This enables easy handling of images in the dataset, applies transformations, and organizes them into batches for training.

Implementation Details

1.**Custom Dataset Class:**

•We define a CustomImageDataset class that inherits from PyTorch's Dataset to allow flexible image loading.

•This class includes a recursive function find_images that traverses the directory structure, extracts image paths, and assigns labels based on the directory names (Benign or Malignant).

•Images are then loaded and returned as tensors after being transformed.

2.**Transformations:**

•The images are resized to 224x224, a common size for deep learning models, and are converted into PyTorch tensors.

•We apply normalization using the standard ImageNet mean and standard deviation to scale pixel values, which helps the model converge faster during training.

3.**DataLoader:**

•The DataLoader class from PyTorch is used to create an iterable over the dataset.

•We set the batch_size to 32 for each batch of images during training, with the training set being shuffled while the validation and test sets remain in order for evaluation.

```
        def find_images(path, label):
            for entry in os.listdir(path):
                entry_path = os.path.join(path, entry)
                if os.path.isdir(entry_path):
                    find_images(entry_path, label)  # Recursive call for subdirectories
                elif os.path.isfile(entry_path) and entry_path.lower().endswith(valid_extensions):
                    self.image_paths.append(entry_path)
                    self.labels.append(self.label_map[label])  # Convert to numerical label

        # Loop over each class directory in the dataset
        for class_dir in os.listdir(root_dir):
            class_path = os.path.join(root_dir, class_dir)
            if os.path.isdir(class_path):
                find_images(class_path, class_dir)

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)

        label = self.labels[idx]  # Now this will be an integer label
        return image, label

# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize image to 224x224
    transforms.ToTensor(),          # Convert image to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize using ImageNet mean and std
])

# Paths to your dataset directories (updated based on your given paths)
train_dir = '/kaggle/input/trace-tumor-notebook/train'
val_dir = '/kaggle/input/trace-tumor-notebook/val'
test_dir = '/kaggle/input/trace-tumor-notebook/test'

# Initialize datasets with the transformations
train_dataset = CustomImageDataset(root_dir=train_dir, transform=transform)
val_dataset = CustomImageDataset(root_dir=val_dir, transform=transform)
test_dataset = CustomImageDataset(root_dir=test_dir, transform=transform)

# Create DataLoader
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)

for batch_idx, (images, labels) in enumerate(train_loader):
    print(f"Batch {batch_idx + 1}")
    print(f"Image batch shape: {images.size()}")
    print(f"Label batch: {labels}")
    break  # To prevent looping through all batches in this example
```

```
Batch 1
Image batch shape: torch.Size([32, 3, 224, 224])
Label batch: tensor([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1,
        1, 1, 1, 1, 0, 0, 1, 1])
```

```
# Get the number of batches in your DataLoader
num_batches = len(train_loader)

print(f"Total number of batches: {num_batches}")
```

```
Total number of batches: 639
```

2.5 Data Augmentation for the Training Dataset

Data augmentation is an essential technique in deep learning, particularly for training image classifiers. It artificially increases the size of the training dataset by generating new images from the original ones through random transformations. This helps prevent overfitting by exposing the model to more varied data, leading to better generalization to unseen data.

In this project, various augmentation techniques are applied to the training dataset, enhancing the model's robustness to variations in the input images. The following augmentations are used:

1.**Random Horizontal Flip:** This operation flips the image horizontally with a 50% probability. This helps the model learn to identify tumors regardless of their orientation in the image.

2.**Random Vertical Flip:** Similar to horizontal flip, this operation flips the image vertically with a 50% probability. This transformation ensures that the model is invariant to vertical flips, which can occur naturally in medical imaging.

3.**Random Rotation:** Images are randomly rotated within a range of -20 to 20 degrees. This augmentation makes the model invariant to slight rotations, which may be common due to patient positioning during scanning.

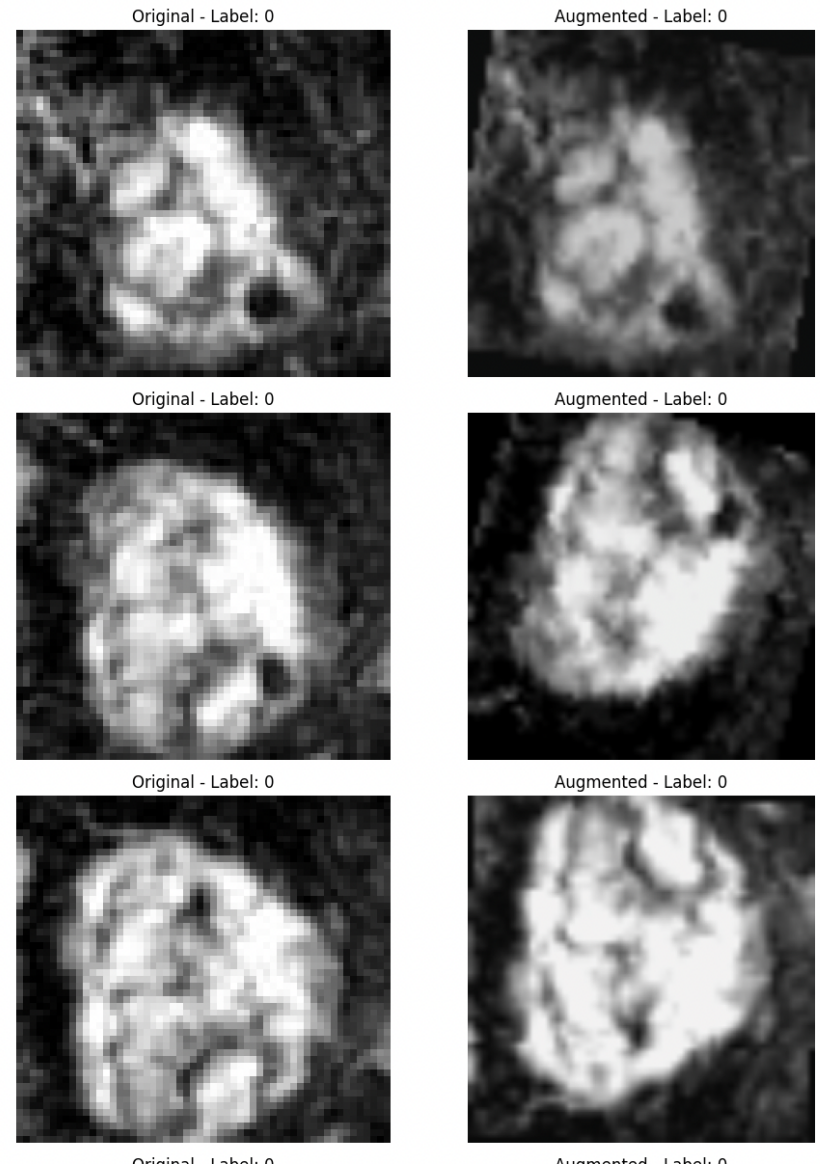
4.**Color Jitter:** This adjustment alters the brightness, contrast, saturation, and hue of the image. It mimics natural variations in image acquisition and lighting conditions, improving the model's ability to generalize across different imaging environments.

5.**Resize:** All images are resized to a fixed dimension (224x224 pixels). This step standardizes the input size for the neural network, ensuring uniformity in the data fed into the model.

6.**ToTensor:** Converts the image from a PIL format to a PyTorch tensor. Tensors are required for processing by neural networks in PyTorch.

7.**Normalization:** The images are normalized using ImageNet's mean and standard deviation values. This ensures that pixel values are scaled in a way that speeds up the model's convergence during training.By applying these augmentations, the model becomes more resilient to variations in the input data, improving its ability to detect tumors under different conditions.

```python
# Define the augmentations and normalization for the training dataset
augmentation_and_normalization_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),       # Randomly flip the image horizontally with 50% probability
    transforms.RandomVerticalFlip(p=0.5),         # Randomly flip the image vertically with 50% probability
    transforms.RandomRotation(degrees=20),        # Randomly rotate the image by up to 20 degrees
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # Random color jitter
    transforms.Resize((224, 224)),                # Resize the image to 224x224
    transforms.ToTensor(),                        # Convert image to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])    # Normalize using ImageNet mean and std
])

# Create dataset and apply augmentations to all images
train_dataset_aug = CustomImageDataset(
    root_dir='/kaggle/input/trace-tumor-notebook/train',  # Change this to your path
    transform=augmentation_and_normalization_transforms
)

# Function to display images before and after augmentation
def show_augmentations(dataset, num_images=5):
    fig, axs = plt.subplots(num_images, 2, figsize=(10, num_images * 4))

    for i in range(num_images):
        # Load original image (before augmentations)
        img_path = dataset.image_paths[i]
        original_img = Image.open(img_path).convert('RGB')

        # Load augmented image
        augmented_img, label = dataset[i]

        # Plot original image
        axs[i, 0].imshow(original_img)
        axs[i, 0].set_title(f"Original - Label: {dataset.labels[i]}")
        axs[i, 0].axis('off')

        # Plot augmented image (convert back to NumPy for visualization)
        augmented_img_np = augmented_img.permute(1, 2, 0).numpy()  # Convert to (H, W, C)
        augmented_img_np = augmented_img_np * [0.229, 0.224, 0.225] + [0.485, 0.456, 0.406]  # Denormalize
        augmented_img_np = augmented_img_np.clip(0, 1)  # Ensure values are between 0 and 1

        axs[i, 1].imshow(augmented_img_np)
        axs[i, 1].set_title(f"Augmented - Label: {dataset.labels[i]}")
        axs[i, 1].axis('off')

    plt.tight_layout()
    plt.show()

# Show a few images before and after augmentation
show_augmentations(train_dataset_aug, num_images=5)
```



2.6. Plotting Colorful Pixel Histograms for Image Channels

In this section, we explore how to visualize the distribution of pixel intensities in the red, green, and blue (RGB) channels of images. This is done by plotting a colorful histogram for each channel to help analyze the pixel intensity distribution, which can be useful for preprocessing tasks like normalization or adjusting image contrast.

Key Features

•**RGB Channel-Specific Histograms**: The histograms are plotted separately for each RGB channel, providing insight into how each color component is distributed across the dataset. This can help identify any potential color bias in the dataset.

•**Handling DataLoader**: The function integrates with PyTorch's DataLoader, allowing it to process images in batches, which is particularly useful for handling large datasets efficiently.

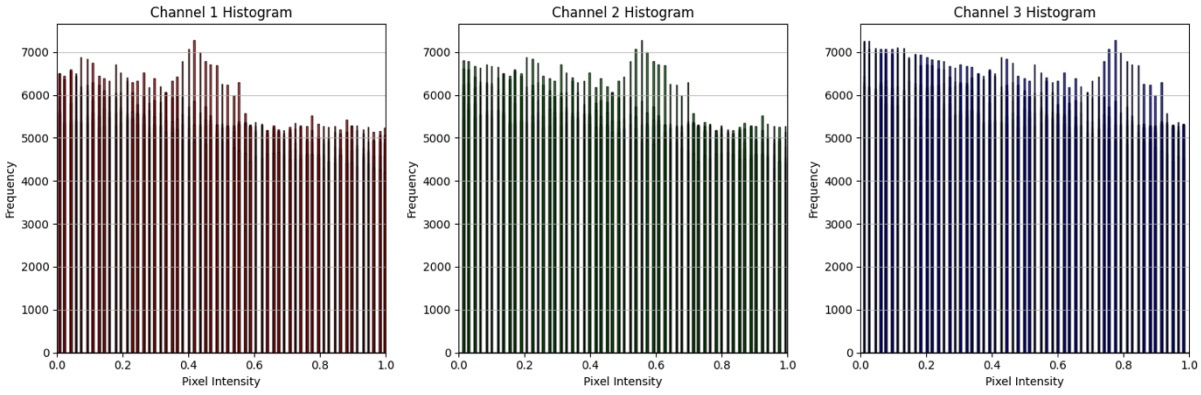
•**Normalized Pixel Intensity**: Pixel values are assumed to be normalized between 0 and 1, which is common in many deep learning pipelines. This allows the histograms to be plotted on a consistent scale, making it easier to analyze the overall distribution of pixel values.

Implementation Steps

1.**Plotting the Histograms**:For each image batch, histograms are plotted for each color channel (Red, Green, and Blue) individually. The histograms show the frequency of each pixel intensity value within the range of 0 to 1.

2.**Customizing the Plot**:The histograms are customized to display with color-specific labels (Red, Green, and Blue) and gridlines for better readability. The pixel intensity values are assumed to be normalized in the range [0, 1], which is why the histograms are plotted within this range.This approach provides a clear visualization of the pixel distribution in each color channel, aiding in data analysis and preparation steps like ensuring proper color balance or correcting potential dataset issues.

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

# Function to plot colorful histogram of pixel values
def plot_colorful_pixel_histogram(data_loader, num_images=5):
    # Set up the colors for each channel (R, G, B)
    colors = ['red', 'green', 'blue']
    plt.figure(figsize=(15, 5))

    # Loop through a few batches of the data loader
    for i, (images, _) in enumerate(data_loader):
        if i >= num_images:  # Limit to a certain number of images
            break

        # Convert images to numpy and reshape
        images_np = images.permute(0, 2, 3, 1).cpu().numpy()  # Change to (batch, height, width, channels)
        images_np = images_np.reshape(-1, images_np.shape[-1])  # Flatten to (num_pixels, channels)

        # Calculate and plot the histogram for each channel (R, G, B)
        for channel in range(images_np.shape[1]):
            plt.subplot(1, 3, channel + 1)
            plt.hist(images_np[:, channel], bins=256, range=(0, 1), color=colors[channel], alpha=0.7, edgecolor='black')
            plt.title(f'Channel {channel + 1} Histogram')
            plt.xlim(0, 1)  # Assuming pixel values are normalized between 0 and 1
            plt.xlabel('Pixel Intensity')
            plt.ylabel('Frequency')
            plt.grid(axis='y', alpha=0.75)  # Add gridlines for better readability

    plt.tight_layout()
    plt.show()

# Call the function to plot colorful histograms
plot_colorful_pixel_histogram(train_loader)
```



CHAPTER 3

Preprocessing

3.0 Extracting and Visualizing HOG Features for Image Analysis

Histogram of Oriented Gradients (HOG) is a powerful feature extraction technique commonly used in computer vision tasks such as object detection. HOG captures the distribution of gradients or edges within localized cells in an image, making it particularly useful for tasks requiring structural and shape information. This section demonstrates how to preprocess images by converting them to grayscale and extracting HOG features, followed by visualizing both the original and HOG features.

3.1 Preprocessing Steps and HOG Feature Extraction

The goal of this preprocessing step is to extract meaningful features from the images, which will be used for further tasks such as classification. Below is the detailed explanation of each part of the process:

1.**HOG Feature Extraction**:

•The skimage.feature.hog() function is used to extract the HOG features. It divides the image into small cells, computes gradient histograms for each cell, and aggregates these histograms across blocks of cells. The HOG method also generates a visualization that highlights the most prominent gradients in the image.

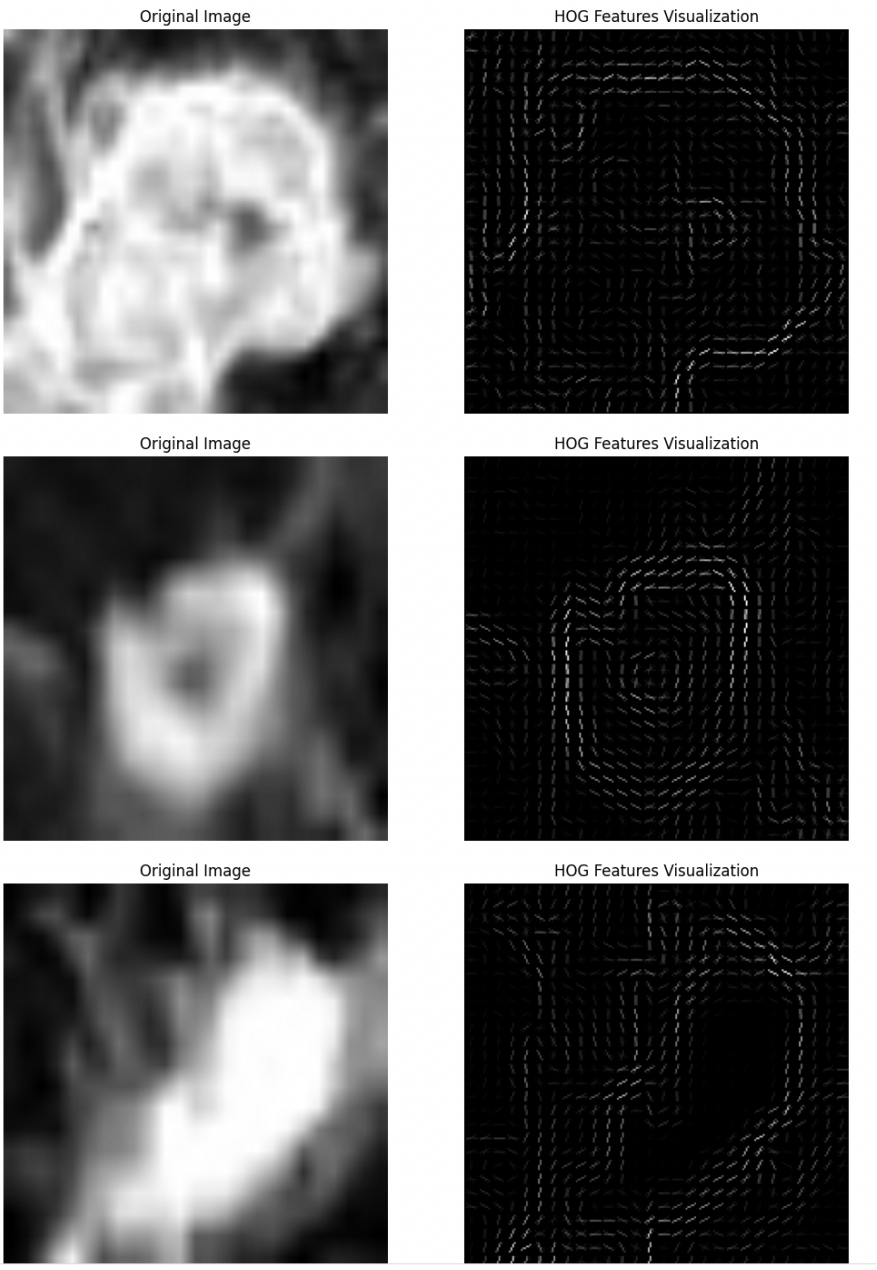
2.**HOG Image Rescaling**:

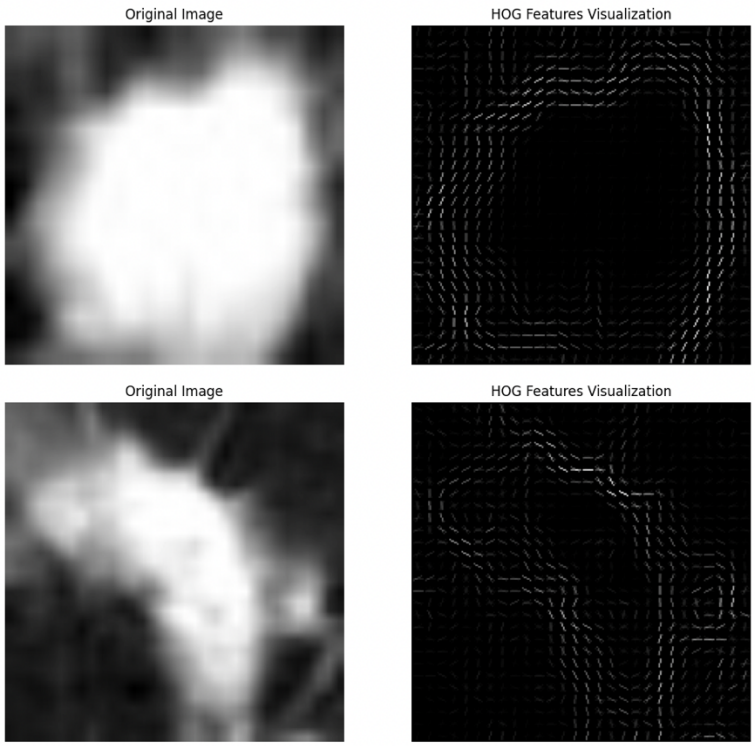
•The computed HOG image is rescaled to improve contrast and make the features more visually interpretable. The rescaling ensures that the intensity of the gradients is adjusted to fall within a specified range for better visualization.

3.**Visualization**:

•For each image, the original grayscale image and the HOG visualization are displayed side by side. This allows for a direct comparison between the raw grayscale image and the extracted HOG features, highlighting how the HOG method captures important edges and gradients.

```python
import torch
from skimage.feature import hog
from skimage import exposure
import numpy as np
import matplotlib.pyplot as plt

# Assuming 'train_loader' is your DataLoader with images loaded
def extract_hog_features(data_loader):
    hog_features = []
    hog_labels = []
    count = 0  # Initialize a counter to keep track of the number of images displayed

    # Iterate through the DataLoader
    for images, labels in data_loader:
        # Convert tensor to numpy array and compute HOG features
        for img, label in zip(images, labels):
            if count >= 5:  # Stop after displaying 5 images
                break

            # Convert tensor to numpy array
            img_np = img.permute(1, 2, 0).numpy()  # Change from (C, H, W) to (H, W, C)
            img_np = (img_np * [0.229, 0.224, 0.225] + [0.485, 0.456, 0.406])  # Denormalize
            img_np = img_np.clip(0, 1)  # Ensure values are between 0 and 1

            # Convert to grayscale for HOG
            img_gray = 0.2989 * img_np[:, :, 0] + 0.5870 * img_np[:, :, 1] + 0.1140 * img_np[:, :, 2]  # Correct grayscale conversion

            # Compute HOG features
            features, hog_image = hog(img_gray,
                                      orientations=9,
                                      pixels_per_cell=(8, 8),
                                      cells_per_block=(2, 2),
                                      visualize=True)

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

            # Store features and labels
            hog_features.append(features)
            hog_labels.append(label.item())  # Convert tensor to Python integer

            # Display original grayscale image and HOG image side by side
            plt.figure(figsize=(12, 6))
            plt.subplot(1, 2, 1)
            plt.imshow(img_gray, cmap='gray')  # Display the grayscale image
            plt.title("Original Image")
            plt.axis('off')

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

            plt.show()  # Show the images

            count += 1  # Increment the counter

        if count >= 5:  # Break outer loop if 5 images have been processed
            break

    return hog_features, hog_labels

# Extract HOG features and display images
hog_features, hog_labels = extract_hog_features(train_loader)
```


Original Image
HOG Features Visualization
Original Image
HOG Features Visualization
Original Image
HOG Features Visualization



3.2 Preprocessing: Sobel Edge Detection on Image Batch

In this step, Sobel edge detection is used to enhance the edges of the images in the dataset. This can help highlight features or boundaries in the images, which can be useful for various computer vision tasks like object detection or segmentation. Below is a detailed breakdown of the process:

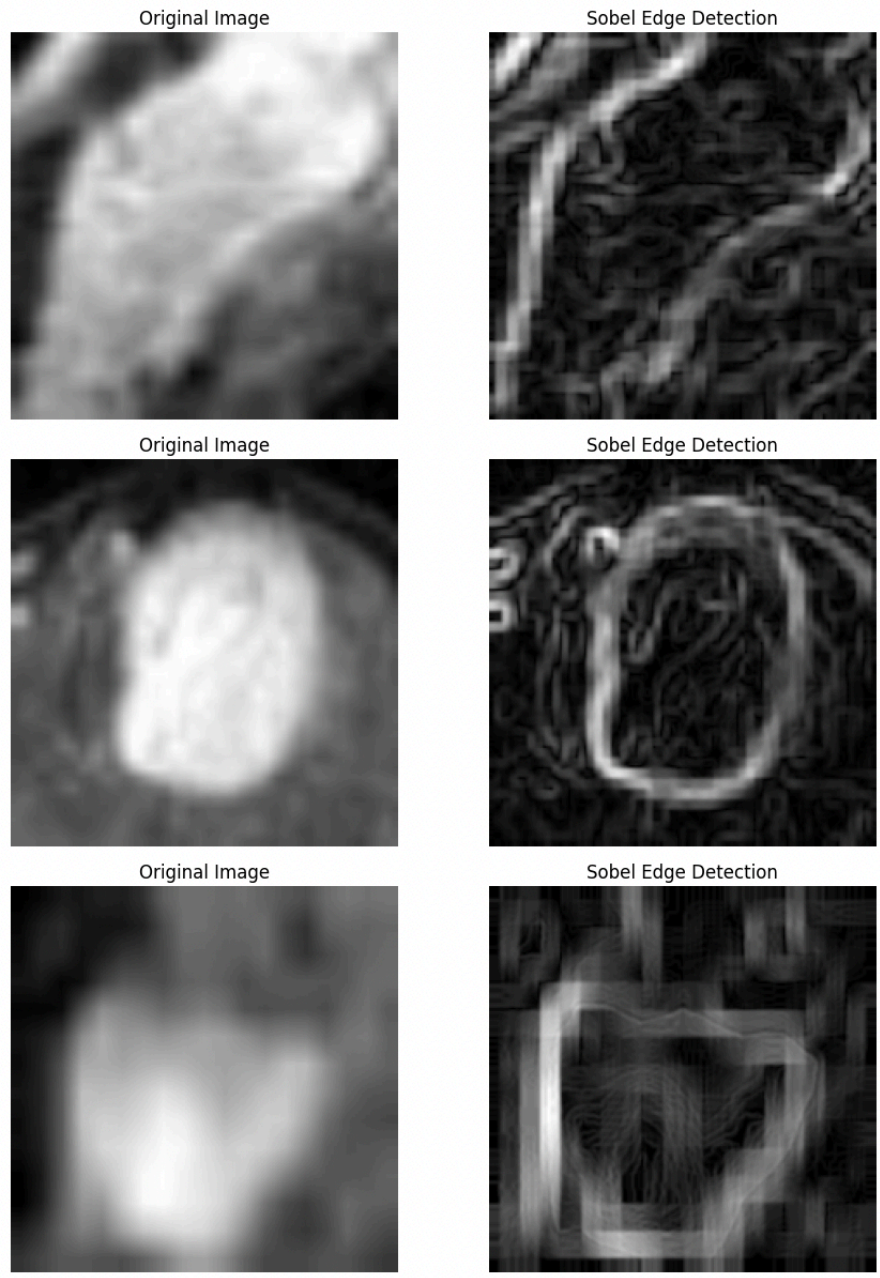
3.2.1.**Sobel Edge Detection**:

•The Sobel operator is applied to detect edges in both the horizontal (X direction) and vertical (Y direction) gradients of the image. This allows the detection of edges along both axes.

•The gradient magnitude is calculated by combining the results of the X and Y Sobel filters using the Euclidean distance formula

•This results in an edge map that highlights the regions with significant intensity changes (edges).

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

def apply_sobel_edge_detection_on_batch(data_loader, batch_size=5):
    images, _ = next(iter(data_loader))

    fig, axs = plt.subplots(batch_size, 2, figsize=(10, batch_size * 4))

    for i in range(batch_size):
        # Convert the tensor image to NumPy array for OpenCV processing
        img_tensor = images[i]
        img_np = img_tensor.permute(1, 2, 0).numpy()  # Convert (C, H, W) to (H, W, C)
        img_np = img_np * [0.229, 0.224, 0.225] + [0.485, 0.456, 0.406]  # Denormalize
        img_np = (img_np * 255).astype(np.uint8)  # Convert to uint8 for OpenCV

        # Check if the image is already grayscale
        if img_np.shape[2] == 1:  # If it has only one channel
            gray_image = img_np
        else:
            # Convert the image to grayscale if it's not already
            gray_image = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)

        # Apply Sobel Edge Detection
        sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=5)  # Sobel in X direction
        sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=5)  # Sobel in Y direction
        edges_sobel = np.sqrt(sobel_x**2 + sobel_y**2)  # Magnitude of Sobel edges

        # Plot the original and Sobel edge detection result
        axs[i, 0].imshow(gray_image, cmap='gray')  # Original grayscale image
        axs[i, 0].set_title("Original Image")
        axs[i, 0].axis('off')

        axs[i, 1].imshow(edges_sobel, cmap='gray')  # Sobel edge detection image
        axs[i, 1].set_title("Sobel Edge Detection")
        axs[i, 1].axis('off')

    plt.tight_layout()
    plt.show()

# Now call the function and pass your DataLoader object
apply_sobel_edge_detection_on_batch(train_loader, batch_size=12)
```



3.3 Convolution Operation

This function performs a 2D convolution operation on the input matrix (A) using a filter (Filt)

I**nputs**:

•A: A 3x3 matrix (image).

•Filt: A 2x2 filter (kernel).

Process:

•The function calculates the half-size of the filter (hrad and wrad), and applies the filter to every region of the matrix A where the filter fits.

•For each region, it computes the sum of element-wise multiplication between the filter and the matrix region.

Output:

•The resulting matrix after applying the convolution operation.

```python
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

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

def conv(A, Filt):
    A_x, A_y = A.shape
    Filt_x, Filt_y = Filt.shape

    # Calculate the padding or radius for height and width
    hrad, wrad = Filt_x // 2, Filt_y // 2
    print(f"Filter half sizes - hrad: {hrad}, wrad: {wrad}")

    # Initialize the output matrix with zeros
    output_matr = np.zeros((A_x - Filt_x + 1, A_y - Filt_y + 1))

    # Loop over every element in the input array where the filter can be applied
    for i in range(hrad, A_x - hrad):
        for j in range(wrad, A_y - wrad):

            region = A[i-hrad:i-hrad+Filt_x, j-wrad:j-wrad+Filt_y]

            output_matr[i-hrad, j-wrad] = np.sum(region * Filt)

    return output_matr

# Call the function
result = conv(A, Filt)
print(result)
```

```
Filter half sizes - hrad: 1, wrad: 1
[[-4.  0.]
 [ 0.  0.]]
```

This code applies a 2D convolution operation on a grayscale image using a specified filter.

Steps:

1.Load the image from the given path and convert it to grayscale.

2.Convert the grayscale image into a NumPy array.

3.Define a filter (Filt), in this case, a 2x2 filter.

4.Apply the convolution operation using the previously defined conv() function.

5.Display the original and convolved images side by side.

Inputs:

•img_path: Path to the image.

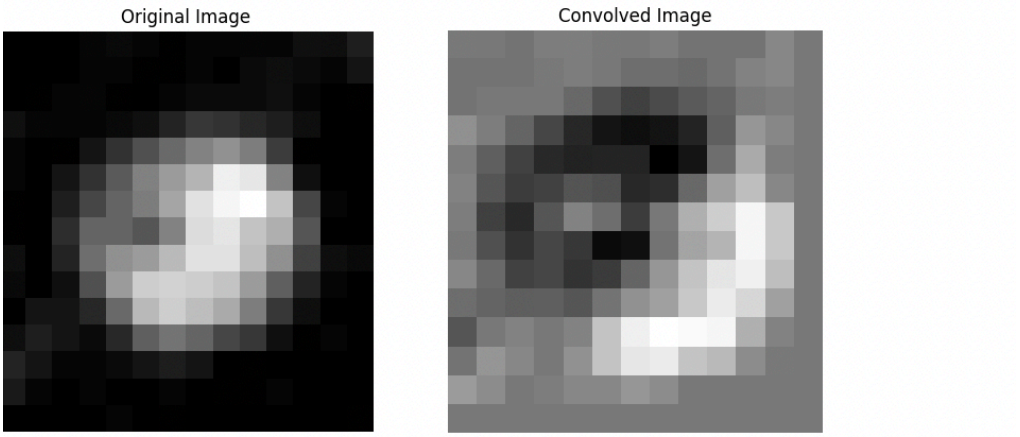
•Filt: The convolution filter (2x2 matrix).

Output:

•Displays the original and convolved images for visual comparison.

```
img_path = '/kaggle/input/trace-tumor-notebook/test/Benign/BreaDM-Be-1813/SUB2/p-017.jpg'  # Change to your path
img = Image.open(img_path).convert('L')  # Convert to grayscale

# Convert image to NumPy array
img_np = np.array(img)

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

convolved_image = conv(img_np, Filt)

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


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

# Convolved image
plt.subplot(1, 2, 2)
plt.imshow(convolved_image, cmap='gray')
plt.title('Convolved Image')
plt.axis('off')

plt.show()
```

```
ilter half sizes - hrad: 1, wrad: 1
```



3.5 Sobel Edge Detection on Image Batch without using inbuilt function

This code applies Sobel edge detection on a batch of images, highlighting edges in the X and Y directions.

Steps:

1.Define Sobel operators for detecting edges in both X and Y directions.

2.Define a convolution function to apply the Sobel operators on each image.

3.Load a batch of images from the DataLoader.

4.Convert each image to grayscale and apply Sobel edge detection using the convolution() function.

5.Calculate the gradient magnitude from the X and Y gradients.

6.Display the original grayscale image alongside the result of Sobel edge detection.

Inputs:

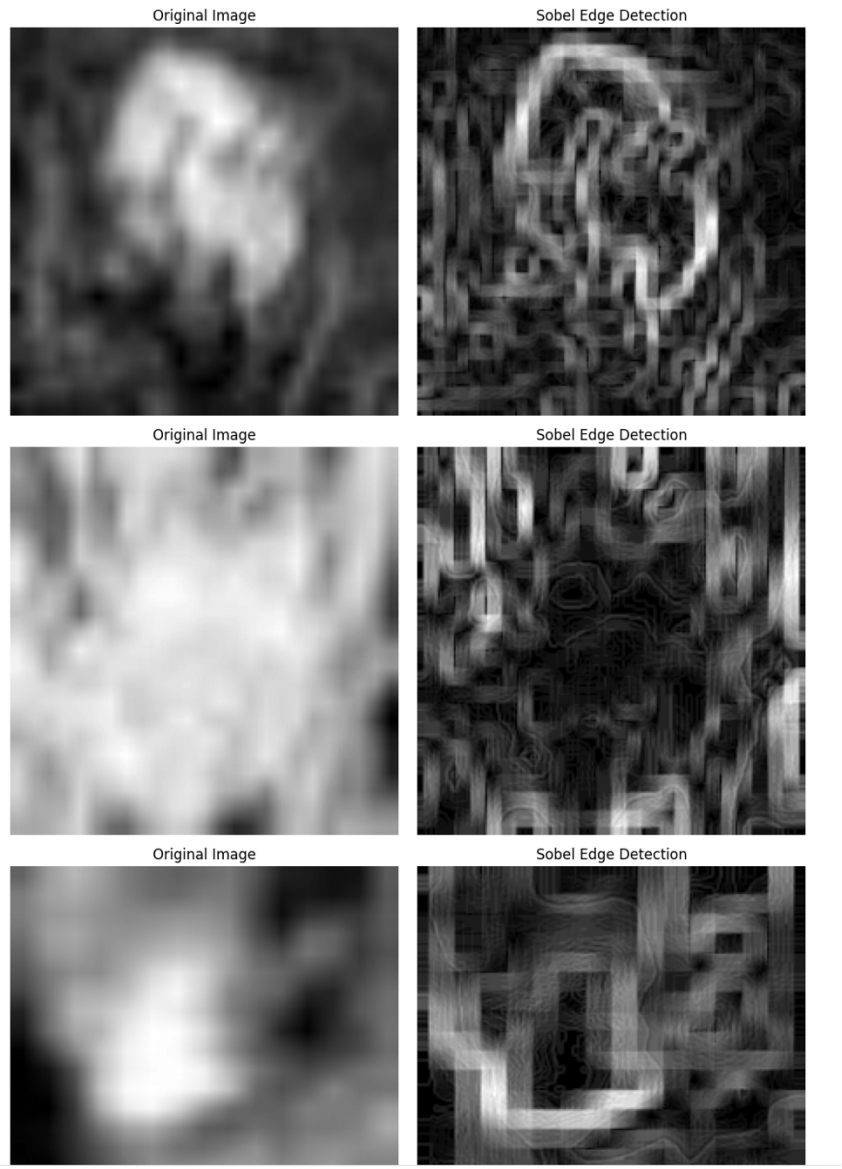
•train_loader: DataLoader containing the images.

•sobel_x_operator, sobel_y_operator: Sobel filters for edge detection.

Output:

•Displays the original grayscale image and the Sobel edge-detected image for each image in the batch.

```python
# Define Sobel operators for X and Y directions
sobel_x_operator = np.array([[1, 0, -1],
                             [2, 0, -2],
                             [1, 0, -1]])  # Sobel X operator

sobel_y_operator = np.array([[1, 2, 1],
                             [0, 0, 0],
                             [-1, -2, -1]])  # Sobel Y operator

def convolution(image, kernel):
    # Get dimensions of the image and the kernel
    image_x, image_y = image.shape
    kernel_x, kernel_y = kernel.shape

    # Calculate output dimensions
    output_x = image_x - kernel_x + 1
    output_y = image_y - kernel_y + 1

    # Initialize output array
    output = np.zeros((output_x, output_y))

    # Perform convolution
    for i in range(output_x):
        for j in range(output_y):
            sub_matrix = image[i:i + kernel_x, j:j + kernel_y]
            output[i, j] = np.sum(sub_matrix * kernel)

    return output

def apply_sobel_on_batch(data_loader, batch_size=5):
    images, _ = next(iter(data_loader))  # Get one batch of images

    fig, axs = plt.subplots(batch_size, 2, figsize=(10, batch_size * 5))

    for i in range(batch_size):
        # Convert the tensor image to NumPy array for OpenCV processing
        img_tensor = images[i]
        img_np = img_tensor.permute(1, 2, 0).numpy()  # Convert (C, H, W) to (H, W, C)
        img_np = img_np * [0.229, 0.224, 0.225] + [0.485, 0.456, 0.406]  # Denormalize
        img_np = (img_np * 255).astype(np.uint8)  # Convert to uint8 for OpenCV

        # Convert the image to grayscale
        gray_image = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)

        # Apply Sobel operators
        gradient_x = convolution(gray_image, sobel_x_operator)
        gradient_y = convolution(gray_image, sobel_y_operator)

        # Calculate the magnitude of the gradient
        magnitude = np.sqrt(gradient_x ** 2 + gradient_y ** 2)
        magnitude = np.clip(magnitude, 0, 255).astype(np.uint8)

        # Plot the original grayscale image and the Sobel image
        axs[i, 0].imshow(gray_image, cmap='gray')
        axs[i, 0].set_title("Original Image")
        axs[i, 0].axis('off')

        axs[i, 1].imshow(magnitude, cmap='gray')
        axs[i, 1].set_title("Sobel Edge Detection")
        axs[i, 1].axis('off')

    plt.tight_layout()
    plt.show()

# Call the function to apply Sobel edge detection on a batch from your train loader
apply_sobel_on_batch(train_loader, batch_size=5)
```



3.6 Local Binary Pattern (LBP) on Images

This code applies Local Binary Pattern (LBP) to grayscale images.

LBP Calculation:

•A 3x3 neighborhood is used to compute the LBP for each pixel.

•Binary values are generated based on whether neighboring pixels are greater than or equal to the center pixel.

•The binary vector is converted to a decimal value and stored in the LBP image.

Visualization:

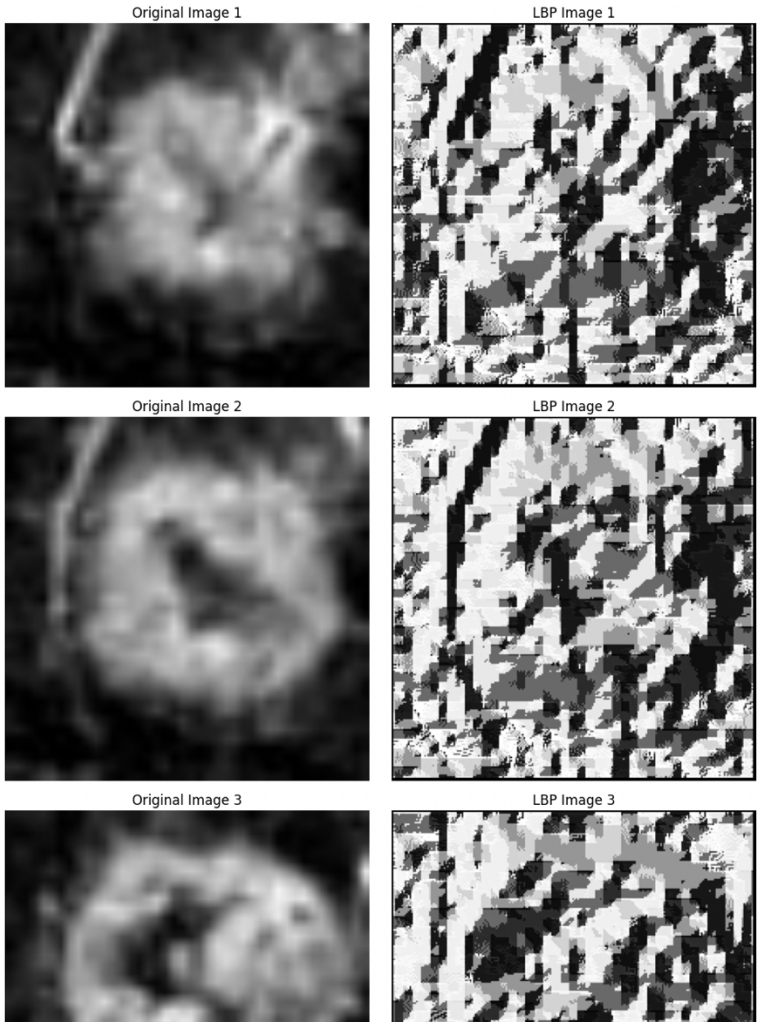
•Displays the original and corresponding LBP images side by side for each input image.

Key Functions:

•getLBPimage: Computes LBP for each image.

•process_images: Loads images, applies LBP, and displays results.

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

# Custom LBP function
def getLBPimage(gray_image):
    imgLBP = np.zeros_like(gray_image)
    neighboor = 3
    for ih in range(0, gray_image.shape[0] - neighboor):
        for iw in range(0, gray_image.shape[1] - neighboor):


            # Step 1: 3x3 neighborhood
            img = gray_image[ih:ih + neighboor, iw:iw + neighboor]
            center = img[1, 1]
            img01 = (img >= center) * 1.0
            img01_vector = img01.T.flatten()

            # Step 2: Binary operation
            img01_vector = np.delete(img01_vector, 4)

            # Step 3: Convert to decimal
            where_img01_vector = np.where(img01_vector)[0]
            if len(where_img01_vector) >= 1:
                num = np.sum(2 ** where_img01_vector)
            else:
                num = 0
            imgLBP[ih + 1, iw + 1] = num

    return imgLBP

# Function to load images
def process_images(image_paths):
    fig, axs = plt.subplots(3, 2, figsize=(10, 15))

    for i, img_path in enumerate(image_paths):

        image = cv2.imread(img_path)
        image_resized = cv2.resize(image, (224, 224))
        gray_image = cv2.cvtColor(image_resized, cv2.COLOR_BGR2GRAY)

        # Apply LBP
        lbp_image = getLBPimage(gray_image)

        # Display original and LBP image
        axs[i, 0].imshow(cv2.cvtColor(image_resized, cv2.COLOR_BGR2RGB))
        axs[i, 0].set_title(f"Original Image {i+1}")
        axs[i, 0].axis('off')

        axs[i, 1].imshow(lbp_image, cmap='gray')
        axs[i, 1].set_title(f"LBP Image {i+1}")
        axs[i, 1].axis('off')

    plt.tight_layout()
    plt.show()

image_paths = [
    '/kaggle/input/trace-tumor-notebook/train/Malignant/BreaDM-Ma-1802/SUB1/p-035.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Malignant/BreaDM-Ma-1802/SUB1/p-036.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Malignant/BreaDM-Ma-1802/SUB1/p-040.jpg']

process_images(image_paths)
```



3.7 Mean Binary Point (MBP) for Image Processing

This code applies the **Mean Binary Point (MBP)** technique to images.

Mean Binary Point Calculation:

•For each pixel in the image (excluding borders), a 3x3 window is selected.

•The mean of the window is computed.

•A binary pattern is created based on whether the surrounding pixel values are greater than or equal to the mean.

•This binary pattern is converted into an integer value and stored in the MBP image.

Normalization:

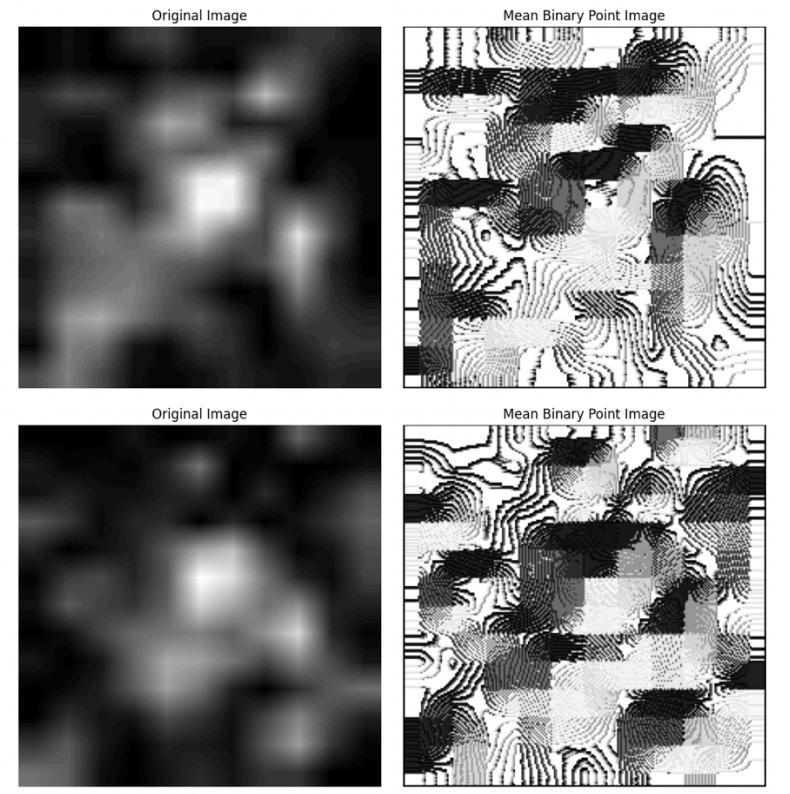
•The resulting MBP image is normalized for better visibility using cv2.normalize().

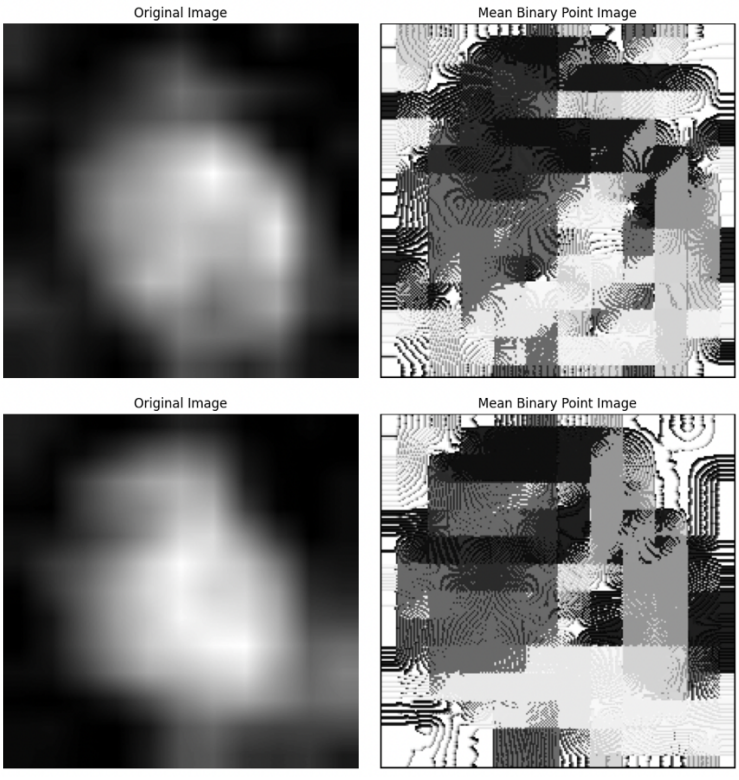
Key Functions:

•calculate_mean_binary_point: Computes MBP for each pixel in the image.

•display_results: Displays the original and MBP images in a side-by-side layout.

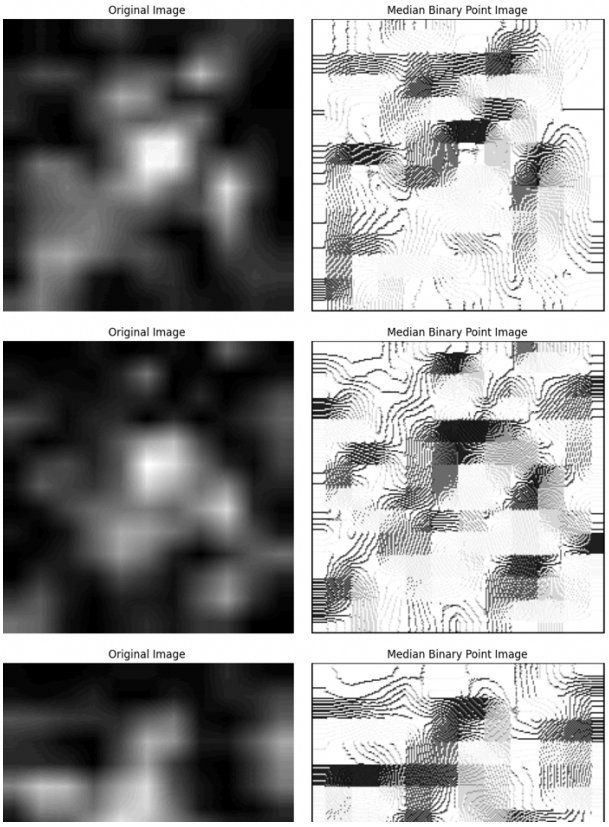
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def calculate_mean_binary_point(image, window_size=3):
    rows, cols = image.shape
    mbp_image = np.zeros((rows, cols), dtype=np.uint8)

    # Iterate through the image, skipping the border pixels
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            window = image[i - 1:i + 2, j - 1:j + 2]
            mean = np.mean(window)

            # Creating binary pattern based on the mean threshold
            binary_str = ''
            for x in range(3):
                for y in range(3):
                    if (x, y) != (1, 1):  # Exclude center pixel
                        binary_str += '1' if window[x, y] >= mean else '0'

            mbp_value = int(binary_str, 2)  # Convert binary string to integer
            mbp_image[i, j] = mbp_value

    # Normalize the MBP image for better visibility
    mbp_image = cv2.normalize(mbp_image, None, 0, 255, cv2.NORM_MINMAX)

    return mbp_image

def display_results(original_image, mbp_image):
    plt.figure(figsize=(10, 5))

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

    # MBP Image
    plt.subplot(1, 2, 2)
    plt.imshow(mbp_image, cmap='gray')
    plt.title('Mean Binary Point Image')
    plt.axis('off')

    plt.tight_layout()
    plt.show()

# List of image paths
image_paths = [
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-028.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-029.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-030.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB2/p-029.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB2/p-030.jpg'
]

# Process and display results for each image
for image_path in image_paths:
    # Load the grayscale image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    # Resize the image to 224x224
    image_resized = cv2.resize(image, (224, 224))

    # Calculate Mean Binary Point
    mbp_image = calculate_mean_binary_point(image_resized)

    # Display results
    display_results(image_resized, mbp_image)
```

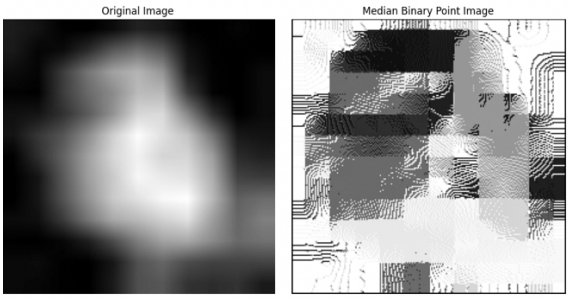

Original Image
Mean Binary Point Image
Original Image
Mean Binary Point Image


Original Image
Mean Binary Point Image
Original Image
Mean Binary Point Image

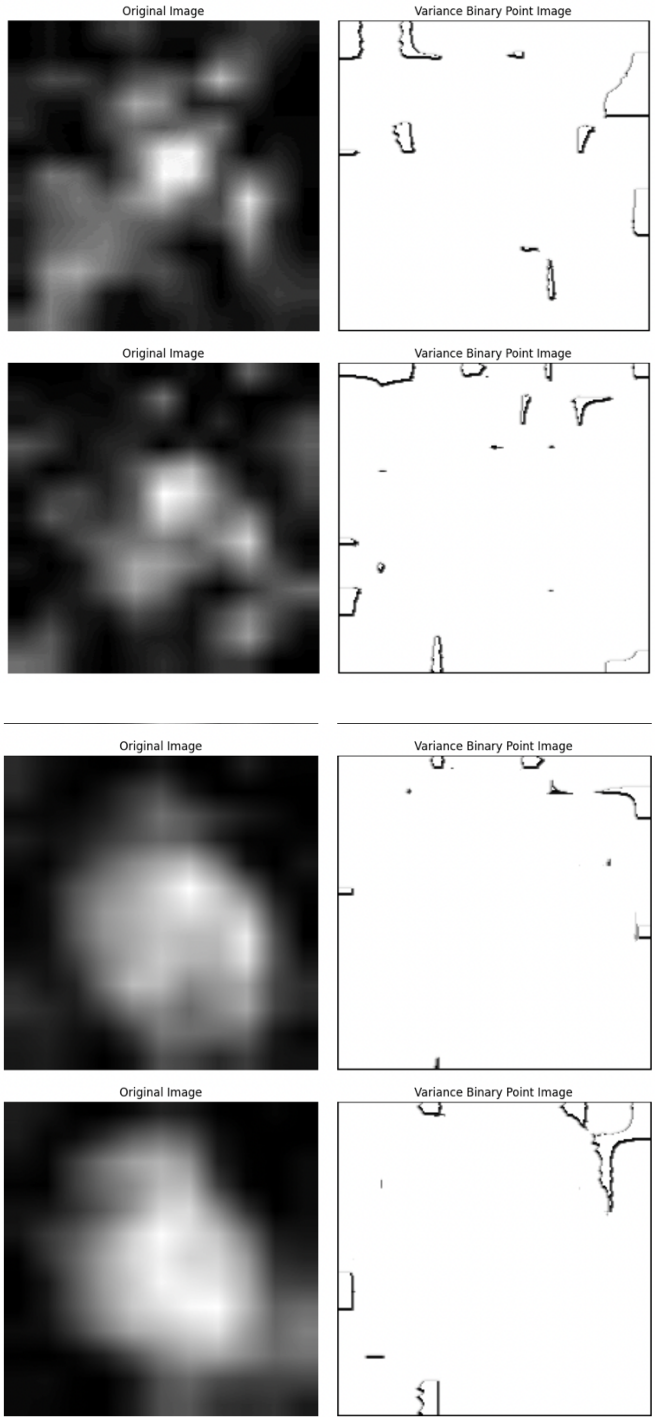
Similar code for median and variance just replace the formula

Median





Variance



3.8 Mean-Median-Variance LBP for Image Processing

This code implements a **Mean-Median-Variance LBP** (MMV-LBP) method for extracting patterns from images.

MMV-LBP Calculation:

•For each pixel (excluding borders), a 3x3 window is taken from the image.

•The **mean**, **variance**, and **median** of the window are computed.

•A threshold is calculated as the average of these three statistics:

•A binary pattern is created by comparing each pixel in the window (excluding the center pixel) with this threshold.

•The resulting binary pattern is converted to an integer and stored in the output image.

Visualization:

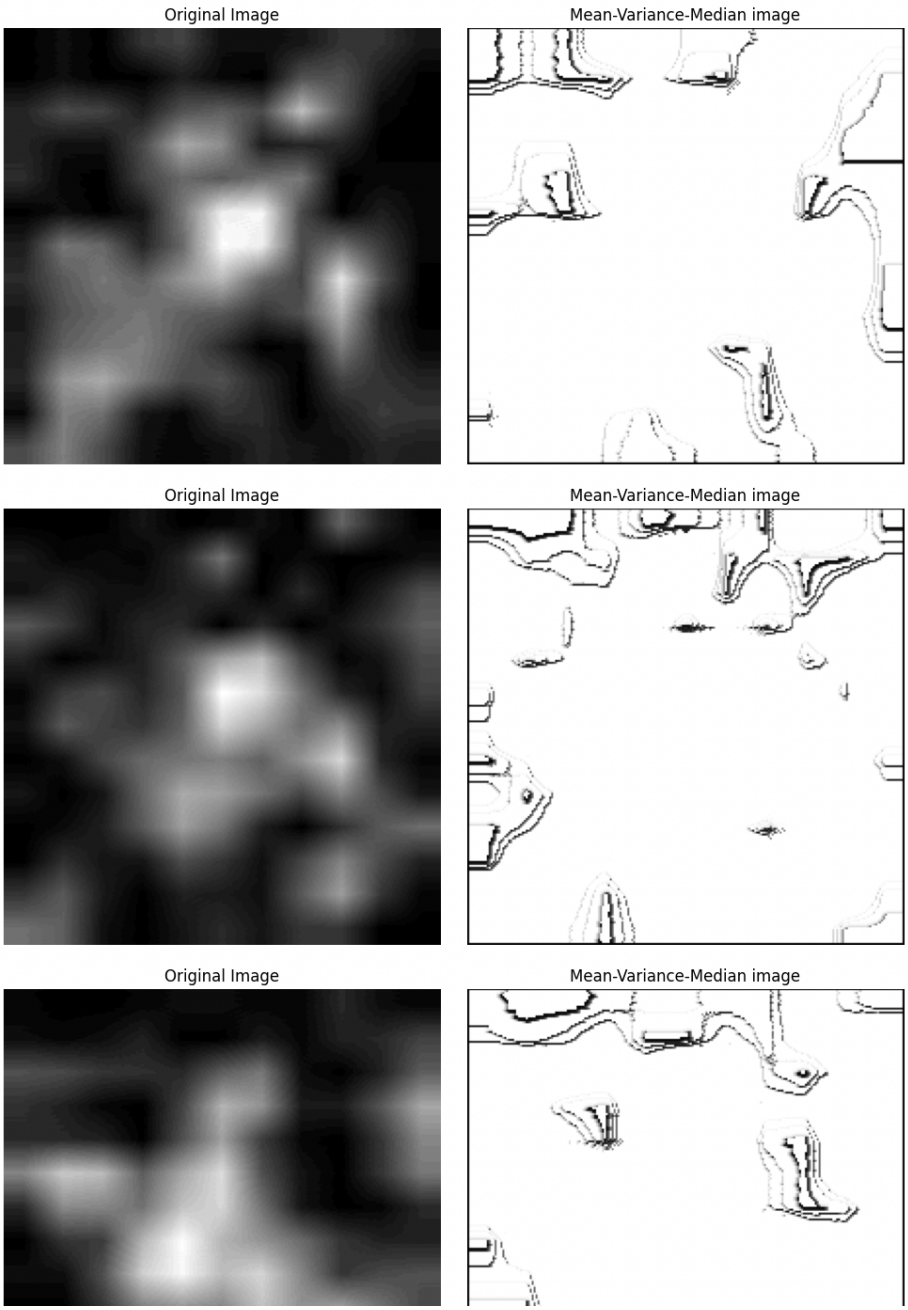
•The original image and the resulting MMV-LBP image are displayed side by side for comparison.

Key Functions:

•calculate_mmv_lbp: Computes the MMV-LBP for each pixel in the image.

•display_results: Displays the original and MMV-LBP images side by side.

```python
def calculate_mmv_lbp(image, window_size=3):
    rows, cols = image.shape
    mmv_lbp_image = np.zeros((rows, cols), dtype=np.uint8)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            window = image[i - 1:i + 2, j - 1:j + 2]
            mean = np.mean(window)
            variance = np.var(window)
            median = np.median(window)

            # Calculate threshold
            threshold = (mean + np.sqrt(variance) + median) / 3


            binary_str = ''
            for x in range(3):
                for y in range(3):
                    if (x, y) != (1, 1):  # Exclude center pixel
                        binary_str += '1' if window[x, y] >= threshold else '0'


            lbp_value = int(binary_str, 2)
            mmv_lbp_image[i, j] = lbp_value

    return mmv_lbp_image

def display_results(original_image, pattern_image, title):
    plt.figure(figsize=(10, 5))

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

    # Pattern Image
    plt.subplot(1, 2, 2)
    plt.imshow(pattern_image, cmap='gray')
    plt.title('Mean-Variance-Median image')
    plt.axis('off')

    plt.tight_layout()
    plt.show()


image_paths = [
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-028.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-029.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-030.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB2/p-029.jpg',
    '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB2/p-030.jpg'
]

for image_path in image_paths:

    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    image_resized = cv2.resize(image, (224, 224))


    mmv_lbp_image = calculate_mmv_lbp(image_resized)


    display_results(image_resized, mmv_lbp_image, 'Mean-Median-Variance LBP Image')
```



3.9 GLCM Calculation for Image

This code computes the **Gray Level Co-occurrence Matrix** (GLCM) for an image and displays the number of unique gray levels, the image matrix, and the GLCM matrix.

GLCM Computation:

•The function compute_horizontal_glcm() computes the **Horizontal GLCM** with a distance of 1 pixel, which captures how often pairs of pixel with specific values occur in a specified spatial relationship.

Key Points:

•**Gray Levels**: The total unique gray levels in the image are calculated and printed.

•**GLCM**: The GLCM is computed for the horizontal direction with a distance of 1 pixel and printed.

```
import cv2


image_path = '/kaggle/input/trace-tumor-notebook/train/Benign/BreaDM-Be-1811/SUB1/p-029.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

image_resized = cv2.resize(image, (224, 224))


unique_gray_levels = np.unique(image_resized)
gray_levels = len(unique_gray_levels)
print("Number of gray levels:", gray_levels)


print("Image Matrix:")
print(image_resized)


glcm = compute_horizontal_glcm(image_resized, gray_levels)
print("GLCM Matrix (Horizontal Distance 1):")
print(glcm)
```

```
Number of gray levels: 49
Image Matrix:
[[ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 ...
 [21 21 21 ...  0  0  0]
 [21 21 21 ...  0  0  0]
 [21 21 21 ...  0  0  0]]
GLCM Matrix (Horizontal Distance 1):
[[2043  154    0 ...    0    0    0]
 [ 147 2824  273 ...    0    0    0]
 [   0  261 3146 ...    0    0    0]
 ...
 [   0    0    0 ...   16    2    2]
 [   0    0    0 ...    6   14    1]
 [   0    0    0 ...    0    3    6]]
```

CHAPTER 4

TRAINING

4.0 TRAINING :

In the training phase, the process involves loading the training dataset in batches and preprocessing the images. The model, either VGG16, ResNet-50, or ResNet-18, is initialized with a modified architecture for binary classification. The final fully connected layer is adjusted to match the number of output classes (binary), and the feature extraction layers are frozen to ensure that only the classifier layers are trained. A loss function like CrossEntropyLoss and an optimizer (Adam or SGD) are employed to minimize the loss during training. Each epoch involves a forward pass where the model makes predictions, followed by backpropagation to update the classifier layers while keeping the feature extractor layers fixed. Model performance is assessed using metrics such as accuracy, precision, recall, and F1-score.

4.1 Custom VGG16 Model for Binary Classification

This code implements a custom model based on the VGG16 architecture, modified for a binary classification task.

Model Architecture:

•**Feature Extractor:** The convolutional part of the pre-trained VGG16 model is used for feature extraction. It includes several convolutional layers followed by ReLU activation and pooling layers.

Classifier: A series of fully connected layers follows the feature extractor. The classifier consists of:

•A flattening layer to convert the feature map into a 1D vector.

•Two fully connected layers with 4096 units each and ReLU activation.

•Dropout layers for regularization.

•The final output layer with num_classes (2 in this case) for binary classification.

Forward Pass:

•The input image is passed through the feature extractor (self.features).

•The output is then processed through average pooling (self.avgpool) to reduce spatial dimensions.

•The feature map is flattened and passed through the fully connected layers to produce the classification output

Key Functions:

•**my_model:** Defines the custom model architecture based on the pre-trained VGG16, which includes the feature extractor and classifier parts.

•**forward:** Defines the flow of data through the network during the forward pass, from feature extraction to classification.

This model is initialized with num_classes=2 for binary classification tasks.

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

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

        vgg16 = models.vgg16(pretrained=True)


        self.features = vgg16.features
        self.avgpool = vgg16.avgpool


        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):

        x = self.features(x)


        x = self.avgpool(x)

        x = torch.flatten(x, 1)

        x = self.classifier(x)

        return x

model = my_model(num_classes=2)


print(model)
```

```
/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=
GG16_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, 212MB/s]
my_model(
  (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)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
)
```

4.2 Resnet50 Model

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

class Resnet50(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet50, self).__init__()
        model_resnet50 = models.resnet50(pretrained=True)

        # Copy layers from pretrained ResNet50
        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

        # Replace the fully connected layer
        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
```

4.3 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 architecture
        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
```

CHAPTER 5

VALIDATION

5.0 Training and Validation of Models on train and validation dataset

5.1 Early Stopping Implementation

This code defines an **EarlyStopping** mechanism to halt training when the validation loss stops improving for a specified number of epochs. Early stopping helps prevent overfitting by monitoring validation loss and saving the best model during training.

Key Components:

1.**Initialization (**__init__**):**

•**patience**: Number of epochs to wait after the last improvement before stopping.

•**verbose**: If True, logs messages when the model is saved or when the counter increases.

•**delta**: Minimum change in validation loss to qualify as an improvement.

•**path**: Filepath to save the best model checkpoint.

•**trace_func**: Function used for logging (default is print).

2.**Functionality (**__call__**):**

•**Input:**

•val_loss: Validation loss for the current epoch.

•model: The model being trained.

Behavior:

•Computes a score as the negative of the validation loss (minimization problem).

•Compares score to the best_score:

Improved Validation Loss: Updates best_score, resets the counter, and saves the model checkpoint.

No Improvement: Increments the counter. If the counter exceeds patience, sets the early_stop flag to True.

3.**Model Checkpoint Saving (**save_checkpoint**):**

Input:

•val_loss: Validation loss of the current epoch.

•model: The model to be saved.

Behavior:

Saves the model's state dictionary to the specified path.

Updates val_loss_min to the current validation loss.

```python
import numpy as np
import torch

class EarlyStopping:

    def __init__(self, patience=7, verbose=False, delta=0, path='checkpoint.pt', trace_func=print):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.path = path
        self.trace_func = trace_func

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

        if self.best_score is None:
            # First epoch case - save initial model
            self.best_score = score
            self.save_checkpoint(val_loss, model)

        elif score < self.best_score + self.delta:
            # No improvement in validation loss
            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:
            # Validation loss improved, save model and reset patience counter
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        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
```

5.2 Validation Function for Model Evaluation

This code defines a **validation function** that evaluates a trained model's performance on a validation dataset. It computes various metrics such as loss, accuracy, AUC, precision, recall, sensitivity, and specificity, along with the confusion matrix.

Key Components:

Model Evaluation Mode:

•The model.eval() method sets the model to evaluation mode, ensuring layers like dropout and batch normalization behave appropriately during testing.

Inputs:

•model: The trained model to be evaluated.

•val_loader: DataLoader for the validation dataset, providing batches of data and labels.

Outputs:

•**Metrics Calculated:**

•Average test loss.

•Accuracy (% of correctly classified samples).

•Confusion matrix (cm).

•AUC (Area Under the ROC Curve).

•Precision, recall, F1-score, sensitivity, and specificity.

```python
import torch
import numpy as np
from sklearn import metrics
from sklearn.metrics import roc_curve, auc, precision_score, recall_score, f1_score
import torch.nn.functional as F

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:
        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)

    # 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_value = auc(fpr, tpr)

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

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

    # Calculate additional metrics: Precision, Recall, and F1-Score
    precision = precision_score(all_targets, all_predictions, average='binary')  # For binary classification
    recall = recall_score(all_targets, all_predictions, average='binary')        # For binary classification
    f1 = f1_score(all_targets, all_predictions, average='binary')                # For binary classification

    print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(specificity, sensitivity, auc_value))
    print('Precision: {:.4f}, Recall: {:.4f}, F1-Score: {:.4f}'.format(precision, recall, f1))
    print('\nTest set: Average loss: {:.4f}, Accuracy: {:.2f}%\n'.format(test_loss, 100. * correct / len(val_loader.dataset)))

    return test_loss, 100. * correct / len(val_loader.dataset), cm, auc_value, precision, recall, f1
```

5.3 Training Function

This code defines a **training function** that handles model optimization for a single epoch during supervised learning. It applies learning rate decay and calculates accuracy for the epoch.

Key Components:

1.**Inputs:**

•epoch: Current epoch number.

•model: The model to be trained.

•num_epochs: Total number of epochs.

•l2_decay: Weight decay (L2 regularization) parameter.

• lr: Initial learning rate.

2.**Learning Rate Decay**

•Reduces the learning rate by a factor of 10 every 10 epochs.

•Prevents learning rate from going below 1 \times 10^{-5}.

3.**Optimizer:**

• Uses **Stochastic Gradient Descent (SGD)** with momentum and weight decay for optimization.

4.**Outputs:**

•Prints **training accuracy** for the epoch.

```python
import torch
import torch.nn.functional as F
from tqdm import tqdm

def train(epoch, model, num_epochs, loader, criterion, l2_decay, lr):
    # Calculate learning rate 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()  # Set model to training mode

    correct = 0
    for data, label in tqdm(loader, desc=f'Epoch {epoch + 1}/{num_epochs}', unit='batch'):
        data, label = data.float().to(device), label.long().to(device)  # Move data and labels to device

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(F.log_softmax(output, dim=1), label)  # Use log softmax for output
        loss.backward()
        optimizer.step()

        # Calculate number of correct predictions
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max log-probability
        correct += pred.eq(label.view_as(pred)).sum().item()  # Compare with true labels

    # Calculate accuracy
    accuracy = 100. * correct / len(loader.dataset)
    print(f'Train Accuracy: {accuracy:.2f}%')
```

5.3 Training Function for Epoch-wise Optimization

This code defines a **training function** that handles model optimization for a single epoch during supervised learning. It applies learning rate decay and calculates accuracy for the epoch.

Key Components:

1.**Optimizer:**

•Uses **Stochastic Gradient Descent (SGD)** with momentum and weight decay for optimization.

2.**Outputs:**

•Prints **training accuracy** for the epoch.

```python
total_epochs = 50
lr = 0.01
momentum = 0.9
no_cuda = False
num_classes = 2
log_interval = 16
l2_decay = 0.01
criterian=nn.CrossEntropyLoss()

# Initialize model
model = my_model(num_classes=num_classes)
model = model.to(device)  # Move model to the appropriate device (CPU or CUDA)

print(model)  # Optional: To print the model architecture
```

```
/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
  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
GG16_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
my_model(
  (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)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
```

Vgg16

```
Train Accuracy: 82.83%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9092
Precision: 0.9219, Recall: 0.8514, F1-Score: 0.8852

Test set: Average loss: 0.3888, Accuracy: 82.45%

EarlyStopping counter: 17 out of 20
Epoch 24/50: 100%|██████████| 639/639 [03:53<00:00,  2.74batch/s]
Train Accuracy: 83.18%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9088
Precision: 0.9342, Recall: 0.8349, F1-Score: 0.8818

Test set: Average loss: 0.3877, Accuracy: 82.20%

EarlyStopping counter: 18 out of 20
Epoch 25/50: 100%|██████████| 639/639 [03:54<00:00,  2.72batch/s]
Train Accuracy: 82.87%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9223
Precision: 0.9309, Recall: 0.8608, F1-Score: 0.8945

Test set: Average loss: 0.3625, Accuracy: 83.86%

EarlyStopping counter: 19 out of 20
Epoch 26/50: 100%|██████████| 639/639 [03:52<00:00,  2.75batch/s]
Train Accuracy: 83.13%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9015
Precision: 0.9353, Recall: 0.8223, F1-Score: 0.8751

Test set: Average loss: 0.3992, Accuracy: 81.35%

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

Notebook

+ Code + Markdown

Resnet50

```
Epoch 24/50
Epoch 25/50: 100%|██████████| 639/639 [02:37<00:00,  4.06batch/s]
Train Accuracy: 99.99%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8731
Precision: 0.9147, Recall: 0.7868, F1-Score: 0.8460

Test set: Average loss: 0.5846, Accuracy: 77.22%

Confusion Matrix:
[[ 292  116]
 [ 337 1244]]
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet50/resnet50_epoch_24.pth
EarlyStopping counter: 20 out of 20
Early stopping
```

```
Model save directory created: /kaggle/working/tumor_classification/resnet50
Epoch 1/50
Epoch 2/50: 100%|██████████| 639/639 [02:38<00:00,  4.03batch/s]
Train Accuracy: 86.78%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8031
Precision: 0.9116, Recall: 0.7046, F1-Score: 0.7949

Test set: Average loss: 0.5876, Accuracy: 71.09%

Confusion Matrix:
[[ 300  108]
 [ 467 1114]]
Best model saved to: /kaggle/working/tumor_classification/resnet50/resnet50_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet50/resnet50_epoch_1.pth
Validation loss decreased (inf --> 0.587645). Saving model...
Epoch 2/50
Epoch 3/50: 100%|██████████| 639/639 [02:37<00:00,  4.07batch/s]
Train Accuracy: 91.97%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.5741
Precision: 0.8349, Recall: 0.5180, F1-Score: 0.6393

Test set: Average loss: 0.9398, Accuracy: 53.54%

Confusion Matrix:
[[246 162]
 [762 819]]
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet50/resnet50_epoch_2.pth
EarlyStopping counter: 1 out of 20
Epoch 3/50
Epoch 4/50: 100%|██████████| 639/639 [02:37<00:00,  4.06batch/s]
Train Accuracy: 92.16%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9181
Precision: 0.8857, Recall: 0.8773, F1-Score: 0.8815

Test set: Average loss: 0.3789, Accuracy: 81.25%

Confusion Matrix:
[[ 229  179]
 [ 194 1387]]
Best model saved to: /kaggle/working/tumor_classification/resnet50/resnet50_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet50/resnet50_epoch_3.pth
Validation loss decreased (0.587645 --> 0.378924). Saving model...
Epoch 4/50
Epoch 5/50: 100%|██████████| 639/639 [02:38<00:00,  4.03batch/s]
Train Accuracy: 92.66%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9368
Precision: 0.9044, Recall: 0.9273, F1-Score: 0.9157

Test set: Average loss: 0.3670, Accuracy: 86.43%

Confusion Matrix:
[[ 253  155]
 [ 115 1466]]
Best model saved to: /kaggle/working/tumor_classification/resnet50/resnet50_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet50/resnet50_epoch_4.pth
Validation loss decreased (0.378924 --> 0.366960). Saving model...
Epoch 5/50
```

Resnet 18

```
Model save directory created: /kaggle/working/tumor_classification/resnet18
Epoch 1/50
Epoch 2/50: 100%|██████████| 639/639 [01:45<00:00,  6.03batch/s]
Train Accuracy: 87.46%
Specificity: 1.0000, Sensitivity: 0.0010, AUC: 0.7347
Precision: 0.9399, Recall: 0.6331, F1-Score: 0.7566

Test set: Average loss: 0.8531, Accuracy: 67.62%

Confusion Matrix:
[[ 344   64]
 [ 580 1001]]
Best model saved to: /kaggle/working/tumor_classification/resnet18/resnet18_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet18/resnet18_epoch_1.pth
Validation loss decreased (inf --> 0.853135). Saving model...
Epoch 2/50
Epoch 3/50: 100%|██████████| 639/639 [01:25<00:00,  7.47batch/s]
Train Accuracy: 93.85%
Specificity: 1.0000, Sensitivity: 0.0015, AUC: 0.9029
Precision: 0.9388, Recall: 0.8058, F1-Score: 0.8673

Test set: Average loss: 0.3921, Accuracy: 80.39%

Confusion Matrix:
[[ 325   83]
 [ 307 1274]]
Best model saved to: /kaggle/working/tumor_classification/resnet18/resnet18_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet18/resnet18_epoch_2.pth
Validation loss decreased (0.853135 --> 0.392066). Saving model...
Epoch 3/50
Epoch 4/50: 100%|██████████| 639/639 [01:22<00:00,  7.78batch/s]
Train Accuracy: 94.51%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9281
Precision: 0.9360, Recall: 0.8596, F1-Score: 0.8961

Test set: Average loss: 0.3873, Accuracy: 84.16%

Confusion Matrix:
[[ 315   93]
 [ 222 1359]]
Best model saved to: /kaggle/working/tumor_classification/resnet18/resnet18_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet18/resnet18_epoch_3.pth
Validation loss decreased (0.392066 --> 0.387278). Saving model...
Epoch 4/50
Epoch 5/50: 100%|██████████| 639/639 [01:22<00:00,  7.76batch/s]
Train Accuracy: 95.94%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9366
Precision: 0.9504, Recall: 0.8602, F1-Score: 0.9031

Test set: Average loss: 0.3402, Accuracy: 85.32%

Confusion Matrix:
[[ 337   71]
 [ 221 1360]]
Best model saved to: /kaggle/working/tumor_classification/resnet18/resnet18_best.pth
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet18/resnet18_epoch_4.pth
Validation loss decreased (0.387278 --> 0.340173). Saving model...
Epoch 5/50
```

```
Train Accuracy: 100.00%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.9161
Precision: 0.9298, Recall: 0.9045, F1-Score: 0.9170

Test set: Average loss: 0.4056, Accuracy: 86.98%

Confusion Matrix:
[[ 300  108]
 [ 151 1430]]
Model checkpoint saved to: /kaggle/working/tumor_classification/resnet18/resnet18_epoch_24.pth
EarlyStopping counter: 18 out of 20
Epoch 25/50
Epoch 26/50: 100%|██████████| 639/639 [01:22<00:00,  7.73batch/s]
```

CHAPTER 6

TESTING

6.0 Testing function

6.1 Visualization Functions for Model Performance

These functions provide graphical representations of key evaluation metrics to interpret the performance of a classification model.

Function 1: plot_auc

Purpose:

To plot the Receiver Operating Characteristic (ROC) curve, which illustrates the model's performance in distinguishing between classes across various thresholds.

Parameters:

- fpr (array): False Positive Rate for different thresholds.
- tpr (array): True Positive Rate for different thresholds.
- auc_value (float): The calculated Area Under the Curve (AUC) value.

Implementation Details:

1. **Figure Creation:** Initializes a plot for the ROC curve.
2. **ROC Curve:** Plots the fpr and tpr values.
3. **Reference Line:** Adds a diagonal line ([0, 1]) to represent random guessing.
4. **Axis and Labels:** Sets x-axis (False Positive Rate) and y-axis (True Positive Rate) limits and labels.
5. **Legend and Title:** Displays the AUC value and adds a title to the graph.
6. **Show Plot:** Renders the plot for visualization.

```python
def plot_auc(fpr, tpr, auc_value):
    plt.figure()
    plt.plot(fpr, tpr, color="darkorange", lw=2, label="ROC curve (area = {:.2f})".format(auc_value))
    plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Receiver Operating Characteristic")
    plt.legend(loc="lower right")
    plt.show()


def plot_confusion_matrix(cm, class_names):
    plt.figure(figsize=(6, 6))
    plt.imshow(cm, interpolation="nearest", cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.colorbar()
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)

    thresh = cm.max() / 2
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], "d"), horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel("True label")
    plt.xlabel("Predicted label")
    plt.tight_layout()
    plt.show()
```

6.2 Testing the Model

The test function evaluates the trained model on the test dataset and computes various performance metrics, including accuracy, F1-score, AUC, sensitivity, and specificity. It also visualizes the confusion matrix and ROC curve.

Function: test

Parameters:

- model: The trained PyTorch model to be evaluated.
- test_dataloader: DataLoader object for the test dataset.

Returns:

- accuracy (float): Percentage of correctly classified samples.
- test_loss (float): Average test loss over the dataset.
- auc_value (float): Area Under the Curve (AUC) for the ROC curve.

```python
def test(model, test_dataloader):
    model.eval()
    len_test_dataloader = len(test_dataloader.dataset)

    test_loss = 0
    correct = 0
    possibilities = None
    all_predictions = []
    all_targets = []
    labels = ['benign', 'malignant']

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

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

        pred = test_output.data.max(1)[1]
        all_predictions.append(pred.cpu().numpy())
        all_targets.append(target.cpu().numpy())

        possibility = F.softmax(test_output, dim=1).cpu().data.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()

    all_predictions = np.concatenate(all_predictions)
    all_targets = np.concatenate(all_targets)

    # Classification metrics -> accuracy, F1-score
    print(metrics.classification_report(all_targets, all_predictions, target_names=labels, digits=4))

    # Confusion matrix
    cm = metrics.confusion_matrix(all_targets, all_predictions)
    plot_confusion_matrix(cm, class_names=labels)

    # ROC and AUC
    num_classes = test_output.shape[1]
    label_onehot = np.eye(num_classes)[all_targets]
    fpr, tpr, _ = roc_curve(label_onehot.ravel(), possibilities.ravel())
    auc_value = roc_auc_score(label_onehot, possibilities, average="macro")
    plot_auc(fpr, tpr, auc_value)

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

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

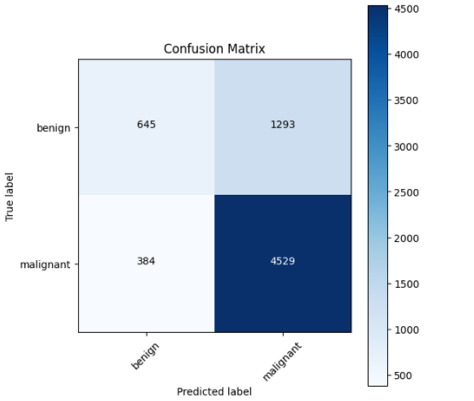
CHAPTER 7

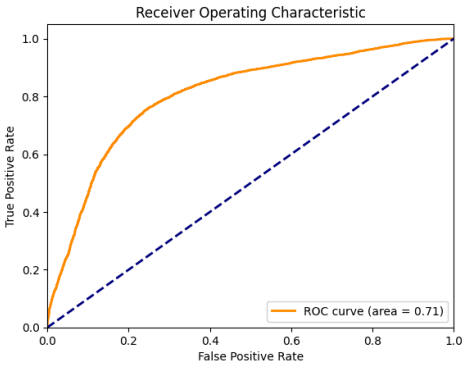
RESULTS AND ACCURACY

7.0 Results and accuracy for all three models

Vgg16

	precision	recall	f1-score	support
benign	0.6268	0.3328	0.4348	1938
malignant	0.7779	0.9218	0.8438	4913
accuracy			0.7552	6851
macro avg	0.7024	0.6273	0.6393	6851
weighted avg	0.7352	0.7552	0.7281	6851



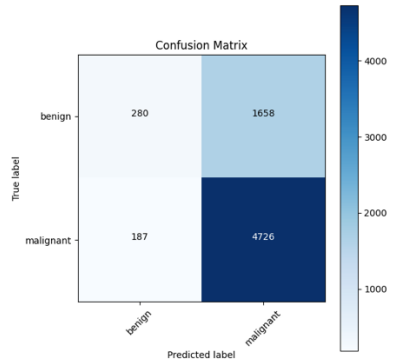


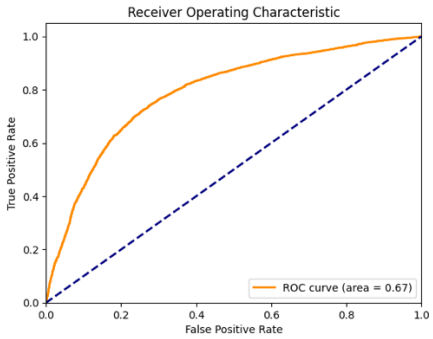
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7140

Test set: Average loss: 0.7387, Accuracy: 5174/6851 (75.52%)

Resnet 50

	precision	recall	f1-score	support
benign	0.5996	0.1445	0.2328	1938
malignant	0.7403	0.9619	0.8367	4913
accuracy			0.7307	6851
macro avg	0.6699	0.5532	0.5348	6851
weighted avg	0.7005	0.7307	0.6659	6851



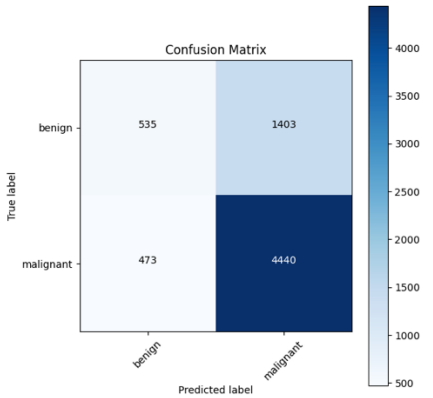


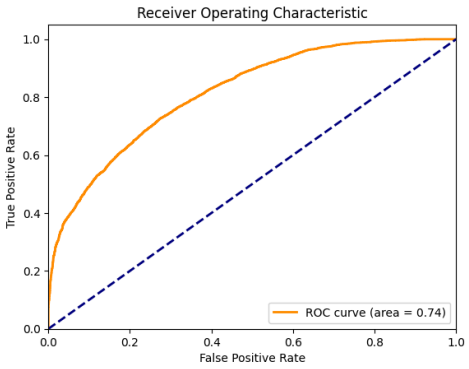
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.6748

Test set: Average loss: 1.1253, Accuracy: 5006/6851 (73.07%)

Resnet 18:

	precision	recall	f1-score	support
benign	0.5308	0.2761	0.3632	1938
malignant	0.7599	0.9037	0.8256	4913
accuracy			0.7262	6851
macro avg	0.6453	0.5899	0.5944	6851
weighted avg	0.6951	0.7262	0.6948	6851






```
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7369

Test set: Average loss: 0.7441, Accuracy: 4975/6851 (72.62%)
```

Model	**Accuracy**
Vgg16	75.52
Resnet50	73.07
Resnet18	72.62

As seen above , currently vgg16 has the best accuracy of 75.52%

7.1 Real Time application-Gradio

Interactive Interface with Gradio:

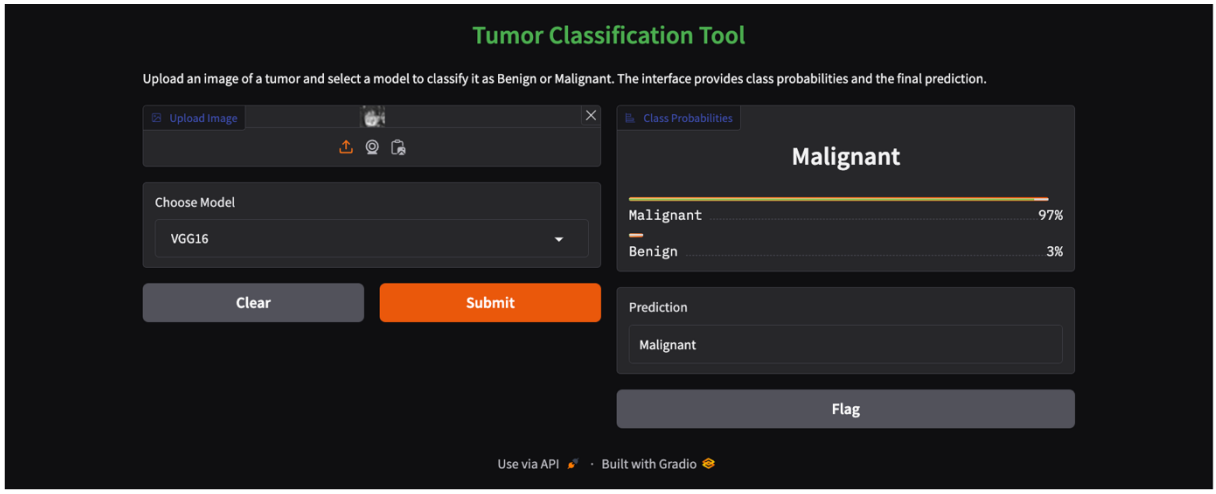
- Users upload an image and select a model (VGG16, ResNet-18, or ResNet-50).

•The app processes the image, makes a prediction (Benign or Malignant), and displays class probabilities.

•A visually styled interface is created using custom CSS.

Deployment:

•The Gradio interface is launched as a web-based application.



REFERENCE

8.0 References

1) https://www.ibm.com/topics/convolutional-neural-networks
2) https://www.mathworks.com/help/deeplearning/ref/vgg16.html
3) https://viso.ai/deep-learning/resnet-residual-neural-network/
4) https://pytorch.org/get-started/locally/
5) https://fairyonice.github.io/implement-lbp-from%20scratch.html
6) https://assets-eu.researchsquare.com/files/rs-2022969/v1_covered.pdf?c=1662821973
7) https://search.app?link=https%3A%2F%2Fwww.kaggle.com%2Fdiscussions%2Fgeneral%2F168323&utm_campaign=aga&utm_source=agsadl2%2Csh%2Fx%2Fgs%2Fm2%2F4