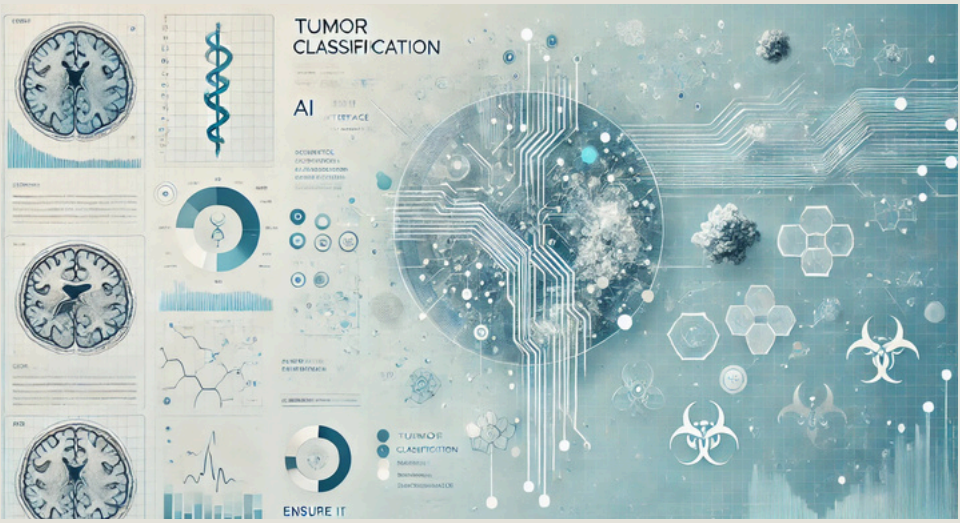
PROJECT TUMOR TRACE: MRI-BASED AI FOR BREAST CANCER DETECTION



Name: Shravani Sanjay More
Company Name : Infosys
Platform : Infosys Springboard
Role : AI/ML Intern
Mentor : Anurag Sista
Duration : 7th October 2024 – December 2024
Email : moreshravani93@gmail.com

LinkedIn : www.linkedin.com/in/shravani-more-511430247
Github:
:https://github.com/rustiiiiiii/TumorTrace_Infosys_Internship_Oct2024/tree/shravani-branch

Table of Contents

CHAPTER 1 : INTRODUCTION

1.1 INTRODUCTION

The "Tumor Trace: MRI-Based AI for Breast Cancer Detection" project represents a groundbreaking effort to harness the power of machine learning and artificial intelligence in the fight against breast cancer. By analyzing MRI images of the breast, this innovative initiative seeks to revolutionize the early detection and diagnosis of this devastating disease. MRI, or magnetic resonance imaging, is a powerful diagnostic tool that can capture detailed, high-resolution images of the body's internal structures, including potential tumors or abnormalities within the breast tissue.

The project's AI-driven approach aims to leverage advanced algorithms and deep learning models to scour these MRI scans, identifying even the most subtle indicators of Malignant or Benign growths with unparalleled accuracy. This could enable clinicians to catch breast cancer at its earliest, most treatable stages, dramatically improving patient outcomes and saving lives. Moreover, the AI-powered analysis can be performed with lightning speed, far outpacing the capabilities of human radiologists reviewing these complex images. As breast cancer remains one of the leading causes of cancer-related deaths among women worldwide, innovations like the "Tumor Trace" project hold immense promise to transform the landscape of early detection and transform the future of cancer care.

1.2 PROBLEM STATEMENT

Breast cancer remains one of the most critical health challenges for women, being one of the leading causes of death globally. The importance of early detection cannot be overstated, as it dramatically improves the chances of successful treatment and recovery. Traditionally, the analysis of MRI images to identify breast cancer has relied heavily on human expertise, which can be both time-consuming and susceptible to errors. Given the need for accuracy and efficiency in diagnosis, this project is focused on harnessing the power of machine learning to develop a sophisticated model capable of classifying MRI images as either benign or malignant. By training the machine learning model on a diverse dataset of MRI scans, we aim to enable faster and more reliable assessments, thereby assisting healthcare professionals in their efforts to detect cancer at an early stage. This technological advancement not only promises to reduce the workload on radiologists but also aims to minimize the stress and uncertainty for patients awaiting results. By providing quicker and more accurate diagnoses, we hope to enhance treatment outcomes and ultimately save lives, making a significant contribution to both individual health and public awareness around breast cancer prevention and care.

1.3 OBJECTIVE

1.3.1 General Objective

To develop an AI model capable of accurately detecting malignant and benign breast tumors from MRI images.

1.3.2 Specific Objectives

- To collect and preprocess a dataset of MRI images.
- To design and implement a machine learning model for tumor classification.
- To evaluate the model's performance and optimize it for better accuracy.
- To develop a user-friendly interface for radiologists and patients to interact with the model.

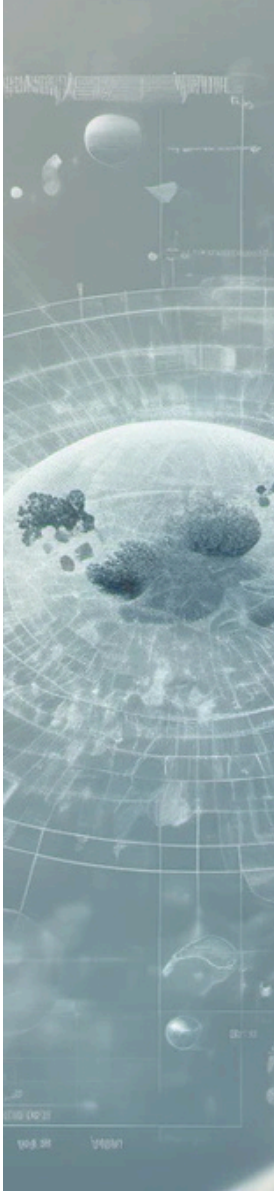
1.4 SCOPE AND LIMITATIONS

1.4.1 Scope

- Development of a machine learning model for breast cancer detection.
- Creation of a web interface for model interaction.
- Analysis and visualization of model performance metrics.

1.4.2 Limitations

- The model's accuracy depends on the quality and quantity of the training data.
- The system requires significant computational resources for training and deployment.
- Potential biases in the dataset could affect the model's performance.



1.5 GOAL

Aid in the early detection of breast cancer to improve patient outcomes.

1.6 APPROACH

Utilize deep learning models, data preprocessing, and evaluation metrics to predict tumor status.

CHAPTER 2 : SOFTWARE REQUIREMENT SPECIFICATION

2.1 REQUIREMENT ANALYSIS

2.1.1 Software Interface Requirements

- Operating System: Linux, Windows, or MacOS
- Web Browser: Chrome, Firefox, or Safari

2.1.2 Tools

- Python programming language
- Jupyter Notebook for development and testing
- PyTorch for model building
- Flask or Django for web development

2.1.3 Language

- Python

2.1.4 Non-functional Requirements

- Usability: The system should be user-friendly and easy to navigate.
- Performance: The model should provide real-time or near real-time predictions.
- Reliability: The system should be robust and handle edge cases effectively.

2.2 FEASIBILITY ANALYSIS

2.2.1 Technical Feasibility

The project leverages existing machine learning libraries and frameworks, making the technical implementation feasible.

2.2.2 Economic Feasibility

The project requires minimal financial investment, with most tools and frameworks being open-source.

CHAPTER 3 : METHODOLOGY

3.1 OVERVIEW

The Tumor Trace project employs a structured methodology to develop an AI-based model for detecting breast cancer through MRI images. This process encompasses several critical steps: data collection, data preprocessing, model development, model training, and evaluation. Each step is essential in ensuring the accuracy and reliability of the final model.

Workflow

- Data Collection – Obtain labeled MRI datasets.
- Data Preprocessing – Resize images, normalize pixel values, data augmentation.
- Model Development – Implement deep learning models (VGG16, ResNet18, Resnet50).
- Model Training - The preprocessed dataset is split into training, validation, and test sets. During training
- Evaluation – Assess performance using metrics like accuracy, precision, recall, F1 score, AUC curve.

3.2 PROGTAMING LANGUAGES

- Python

3.3 LIBRARIES AND FRAMEWORKS

- numpy==1.21.0
- pandas==1.3.0
- matplotlib==3.4.3
- tensorflow==2.6.0
- scikit-learn==0.24.2
- opencv-python==4.5.3.20210927
- scikit-image==0.18.3
- jupyter==1.0.0
- flask==3.1.0

3.4 WEB TECHNOLOGIES:

- HTML
- CSS
- JavaScript

CHAPTER 4 : DATA COLLECTION AND PREPROCESSING

4.1 DATA COLLECTION

A diverse dataset of MRI images was obtained from the Computers in Biology and Medicine source, specifically from "BreastDM: A DCE-MRI dataset for breast tumor image segmentation and classification." This dataset includes annotated images indicating whether the tumors are benign or malignant. The objective is to ensure a representative sample that covers various demographics and cancer stages, thereby enhancing the model's generalizability.

4.2 DATA PREPARATION

Data preparation is a critical phase in machine learning projects, ensuring that the dataset is ready for training and evaluation. This process involves several steps, including cleaning, transforming, and augmenting the data. In this project, the MRI images were prepared to ensure consistency, quality, and diversity, enhancing the model's ability to learn effectively.

4.2.1 Data Understanding

- **Loading the Dataset**

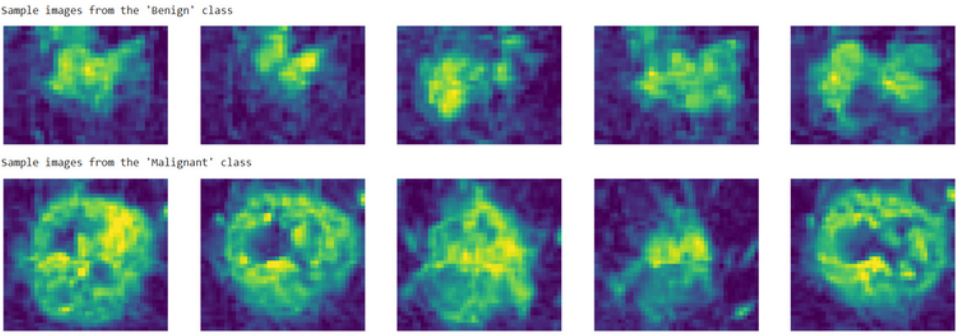
```
import os
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import cv2


# Define the path to your dataset
data_dir = r"/kaggle/input/tumor-trace-dataset"

# Check the directory structure
train_dir = os.path.join(data_dir, 'train')
val_dir = os.path.join(data_dir, 'val')
test_dir = os.path.join(data_dir, 'test')

# Print the classes available in the train directory
print("Classes in training set:")
print(os.listdir(train_dir))

print("\nClasses in val set:")
print(os.listdir(val_dir))

print("\nClasses in test set:")
print(os.listdir(test_dir))
```

```
Classes in training set:
['Benign', 'Malignant']

Classes in val set:
['Benign', 'Malignant']

Classes in test set:
['Benign', 'Malignant']
```

- **Visualization**

To visually inspect the quality and variability of the MRI images, several sample images were displayed. This step helps in understanding the nature of the data and identifying any potential issues early on.

```
import os
import matplotlib.pyplot as plt
from PIL import Image

# Displayingg sample images from a given directory
def show_sample_images(class_dir, num_images=5):

    class_images = os.listdir(class_dir)[:num_images]
    plt.figure(figsize=(15, 5))
    for i, img_name in enumerate(class_images):
        img_path = os.path.join(class_dir, img_name)
        img = Image.open(img_path)
        plt.subplot(1, num_images, i + 1)
        plt.imshow(img)
        plt.axis('off')
    plt.show()

# Defining the train\Benign\BreaDM-Be-1801 directory
benign_images = r"/kaggle/input/tumor-trace-dataset/train/Benign/BreaDM-Be-1801"
img_dir = os.path.join(benign_images, 'SUB1')

malignant_images = r"/kaggle/input/tumor-trace-dataset/train/Malignant/BreaDM-Ma-1802"
img_dir = os.path.join(benign_images, 'SUB2')

# Showing sample images from the 'Benign' class
print("Sample images from the 'Benign' class")
show_sample_images(os.path.join(benign_images, 'SUB1'))

# Showing sample images from the 'Maligant' class
print("Sample images from the 'Malignant' class")
show_sample_images(os.path.join(malignant_images, 'SUB2'))
```



4.2.2 Data Transformation

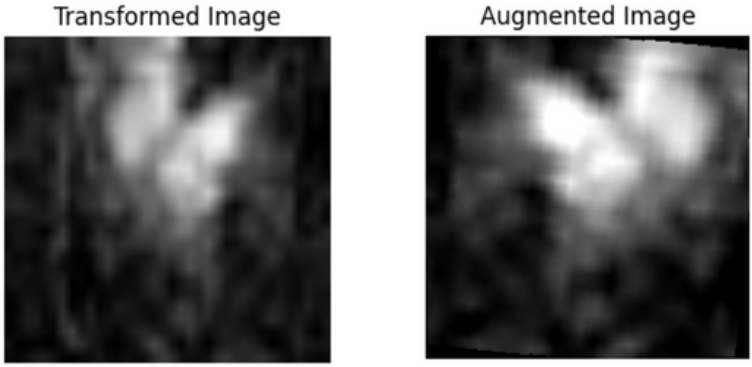
Data transformation is a critical step in preparing the MRI images for training machine learning models. This process involves several key tasks to ensure the data is in the right format and quality for effective learning:

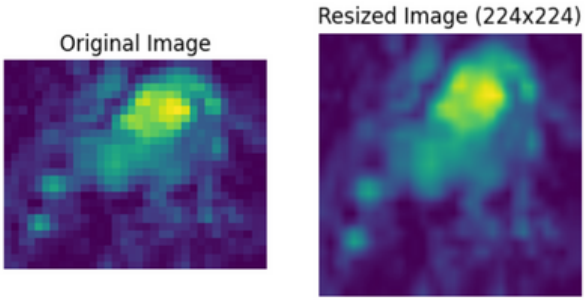
- **Grayscale Conversion:** All images are converted to grayscale to reduce complexity while retaining essential information, making the processing more efficient.

- **Image Resizing**: Each image is resized to a standardized dimension (e.g., 224x224 pixels) to ensure consistency across the dataset, which is crucial for the neural network models.
- **Normalization:** Pixel values are scaled to a range suitable for neural network input, typically [0, 1] or [-1, 1], to facilitate faster convergence during training.
- **Data Augmentation:** Techniques such as rotation, flipping, shifting, and scaling are applied to the images to artificially increase the diversity of the training dataset. This helps in improving the model's generalization capabilities and reduces the risk of overfitting.

```
# Define data augmentation with additional techniques
train_transform = transforms.Compose([
     transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(30),  # Increased rotation range
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.3),
    transforms.RandomGrayscale(p=0.3),  # Increased grayscale probability
    transforms.RandomAffine(degrees=15),  # Slight affine transformations for rotation/translation
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

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





4.2.3 Setting up Dataloaders

Creating Dataset with updated transformation

```
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder

data_dir = "/kaggle/input/tumor-trace-dataset"
train_dataset = datasets.ImageFolder(root=os.path.join(data_dir, 'train'), transform=train_transform)
val_dataset = datasets.ImageFolder(root=os.path.join(data_dir, 'val'), transform=test_transform)
test_dataset = datasets.ImageFolder(root=os.path.join(data_dir, 'test'), transform=test_transform)
```

Creating DataLoader

```
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=4, drop_last=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=4, drop_last=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=4, drop_last=True)

# Check the size of each dataset
print(f'Training set size: {len(train_loader.dataset)}')
print(f'Validation set size: {len(val_loader.dataset)}')
print(f'Test set size: {len(test_loader.dataset)}')
```

```
Training set size: 20434
Validation set size: 1989
Test set size: 6851
```

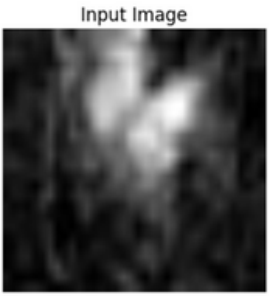
4.3 EXPLORATORY DATA ANALYSIS (EDA)

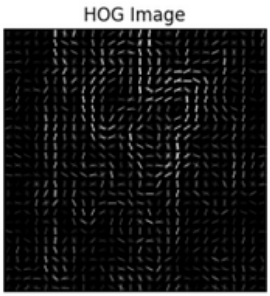
Exploratory Data Analysis (EDA) plays a crucial role in understanding the dataset's structure, identifying patterns, and ensuring data quality before training machine learning models. In this project, EDA focused on analyzing MRI images of breast tumors to uncover distinguishing features between benign and malignant cases. The main steps in EDA include inspecting the distribution of tumor types (benign vs. malignant), visualizing image characteristics, and extracting key features using image processing techniques.

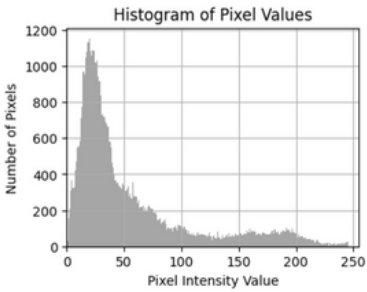
To explore the underlying patterns in the images, several feature extraction methods were applied:

- **Histogram of Oriented Gradients (HOG):** Used to capture the gradient information and edge structures, which help identify the shape and boundary of tumors.

```
import cv2
from skimage.feature import hog
from skimage import exposure
import matplotlib.pyplot as plt

# Load the image
image_path = r"/kaggle/input/tumor-trace-dataset/train/Benign/BreaDM-Be-1801/SUB1/p-032.jpg"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Resizing the image to 224x224
resized_image = cv2.resize(image, (224, 224))

hog_features, hog_image = hog(
    resized_image,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True,
    feature_vector=True,
    block_norm='L2-Hys'
)
pixel_values = resized_image.flatten()

# Rescaling histogram
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))
```



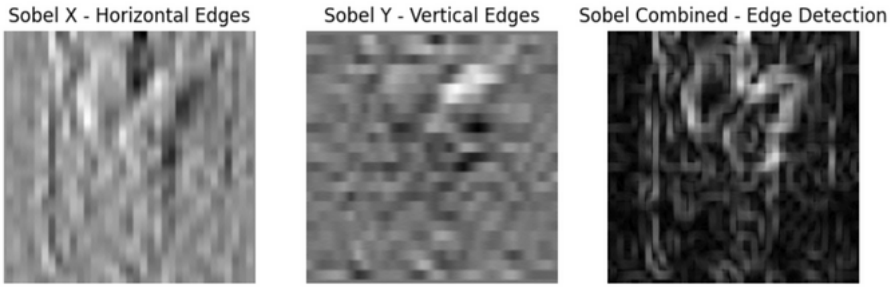




- **Sobel Operator:** Applied to detect edges and highlight areas of high intensity changes, assisting in identifying tumor boundaries.

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

# Function to perform convolution
def convolve(image, kernel):
    image_x, image_y = image.shape   # dimensions (height and width) of the image.
    kernel_x, kernel_y = kernel.shape # dimensions of the kernel
    print(kernel)

    # Determining the output dimensions
    stride = 1 # step size by which we move the kernel across the image (set to 1 here).
    padding = 0
    output_x = int(((image_x - kernel_x + 2 * padding) // stride) + 1) # dimensions of the output image after applying the kernel. #1
    output_y = int(((image_y - kernel_y + 2 * padding) // stride) + 1) # dimensions of the output image after applying the kernel.#1

    # Initialize the output feature map
    output = np.zeros((output_x, output_y)) # matrix initialized to zeros to store the convolution result.
    #initialized to zeros to ensure it starts with a clean slate.

    # Perform convolution in that we  move the kernel over the image.
    for i in range(output_x):
        for j in range(output_y):
            region = image[i:i + kernel_x, j:j + kernel_y]
            output[i, j] = np.sum(region * kernel)

    return output
```

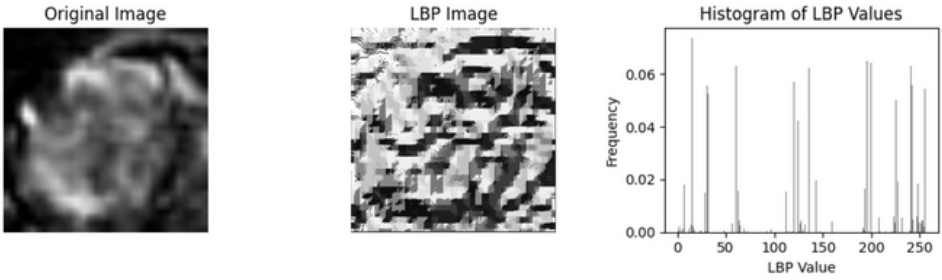
```
# Sobel X operator (horizontal edges)
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
])

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

# Load the image and convert to grayscale
image = resized_image

image = np.array(image)
# Converting the Image to a Numpy Array:

# Apply Sobel X operator
sobel_x_image = convolve(image, sobel_x)

# Apply Sobel Y operator
sobel_y_image = convolve(image, sobel_y)

# Calculate the magnitude of gradients
sobel_combined = np.sqrt(sobel_x_image**2 + sobel_y_image**2)
```



- **Local Binary Pattern (LBP):** Utilized to capture textural features by analyzing pixel intensity variations within a local neighborhood.

  a. Divide image into cells: For each pixel in a cell, compare it with its neighbors.
  b. Threshold the neighborss: If the neighbor's value is greater than or equal to center pixel's value, set it to 1; otherwise, set it to 0.
  c. Generate a binary number: Concatenate all the binary values (0s and 1s) to form a binary number.
  d. Convert to decimal: Convert the binary number to a decimal value.
  e. Replace the pixel value: The center pixel is replaced with the decimal value.

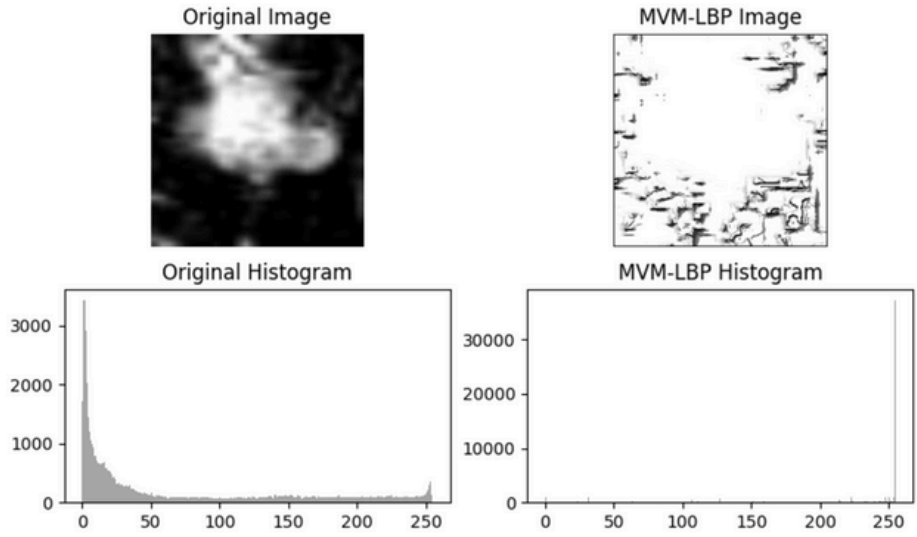
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import local_binary_pattern


def edge_detection(img_path):

    # Load the image in the greyscale modeee
    image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Resizing the image to 224x224
    image = cv2.resize(image, (224, 224))

    # Apply LBP
    radius = 1  # radius of 1 means that the neighbors are directly adjacent to the center pixel.

    n_points = (8 * radius)  # 8 * radius because in a unit circle (radius 1), there are 8 discrete sampling points.

    lbp_image = local_binary_pattern(image, n_points, radius, method="default")

    """Example: 'default'

'default': Regular LBP, where each bit of the binary pattern is a direct thresholding result.
'uniform': LBP where patterns with at most two 0-1 or 1-0 transitions are considered uniform. This results in a smaller feature set.
'ror': Rotation-invariant patterns achieved by bitwise rotation.
'var': Rotation-invariant variance measures of the LBP codes.
Purpose: Specifies how to handle the binary patterns, allowing for different levels of robustness and feature reduction."""
```



Each element in this array represents the brightness of a pixel, with values typically ranging from 0 (black) to 255 (white).

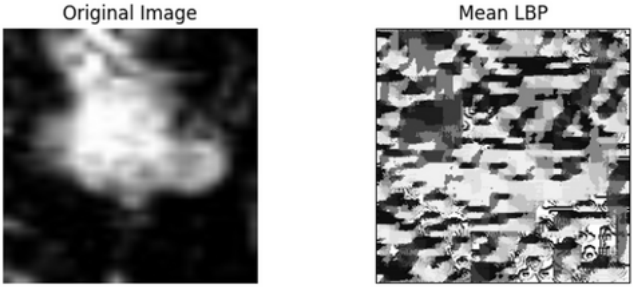
- **Mean-Variance-Median Local Binary Pattern (MVM-LBP):** An advanced LBP technique that combines mean, variance, and median to extract richer textural information from the images. Instead of comparing each neighbor to just the center pixel, we compare each neighbor to a threshold that combines the mean, variance, and median of the pixels in the 3x3 window.

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

# Function to calculate Mean-Variance-Median Local Binary Pattern (MVM-LBP)
def calculate_mvm_lbp(image):
    rows, cols = image.shape  # Get dimensions of the image
    mvm_lbp_image = np.zeros((rows, cols), dtype=np.uint8)  # Initializingg an empty image for MVM-LBP

    for i in range(1, rows - 1):  # Loop over each pixel, excluding the border
        for j in range(1, cols - 1):
            # 3x3 window around the center pixel
            window = image[i - 1:i + 2, j - 1:j + 2]

            #  mean, variance, and median of the 3x3 window
            mean = np.mean(window)
            variance = np.var(window)
            median = np.median(window)

            # threshold =  average of mean, square root of variance, and median
            threshold = (mean + np.sqrt(variance) + median) / 3

            # 8-bit binary pattern based on the threshold
            binary_str = ''.join(['1' if window[x, y] >= threshold else '0'
                                  for x in range(3) for y in range(3) if (x, y) != (1, 1)])  # no center value is counting
            # Convert binary string to integer and set it as the new pixel value
            mvm_lbp_image[i, j] = int(binary_str, 2)

    return mvm_lbp_image
```



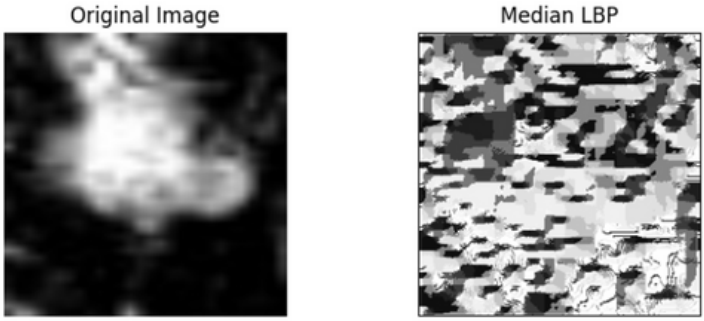
- **LBP - Mean:** Sum of all values / Number of values

```
def mean_lbp(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rows, cols = gray.shape
    mean_lbp_image = np.zeros((rows, cols), dtype=np.uint8)


    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            center = gray[i, j]
            neighborhood = [
                gray[i-1, j-1], gray[i-1, j], gray[i-1, j+1],
                gray[i, j+1], gray[i+1, j+1], gray[i+1, j],
                gray[i+1, j-1], gray[i, j-1]
            ]
            local_mean = np.mean(neighborhood)
            binary_str = ''.join(['1' if pixel >= local_mean else '0' for pixel in neighborhood])
            mean_lbp_image[i, j] = int(binary_str, 2)

    return mean_lbp_image
```



- **LBP - Median :** It refers to a variant of Local Binary Pattern (LBP) that uses the median value of the pixel intensities in the local neighborhood around each central pixel, instead of comparing them to the center pixel's intensity. The median value is more robust to noise, as it represents the middle value in a sorted list of neighboring intensities. This approach helps in capturing texture patterns while minimizing the impact of outliers or noise, which is useful for tumor classification in medical images like MRI scans.

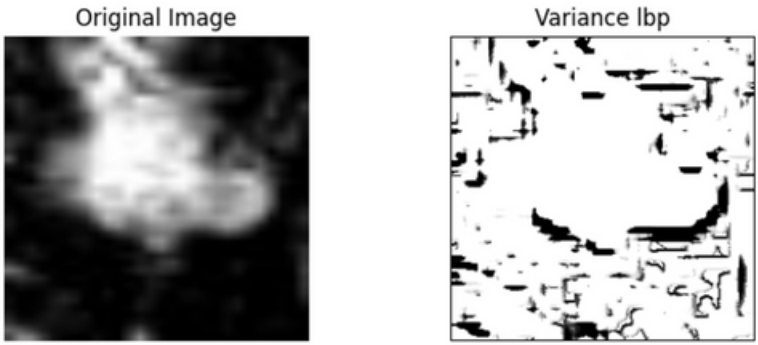
```
# Function for Median LBP
def median_lbp(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rows, cols = gray.shape
    median_lbp_image = np.zeros((rows, cols), dtype=np.uint8)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            center = gray[i, j]
            neighborhood = [
                gray[i-1, j-1], gray[i-1, j], gray[i-1, j+1],
                gray[i, j+1], gray[i+1, j+1], gray[i+1, j],
                gray[i+1, j-1], gray[i, j-1]
            ]
            local_median = np.median(neighborhood)
            binary_str = ''.join(['1' if pixel >= local_median else '0' for pixel in neighborhood])
            median_lbp_image[i, j] = int(binary_str, 2)

    return median_lbp_image
```



- **LBP - Variance:** Variance = Average of the squared differences from the Mean.

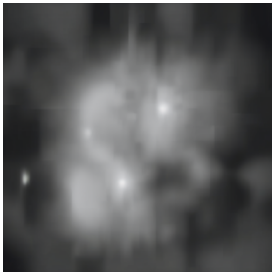
It refers to a variant of Local Binary Pattern (LBP) that focuses on the variability or spread of pixel intensities in a local neighborhood around each pixel. It measures the difference between the central pixel and its surrounding neighbors by calculating the variance of their intensity values. Higher variance indicates more variation in the pixel intensities, which can help capture texture details in the image, especially useful for distinguishing between different tumor types (benign vs. malignant) in medical imaging.

```python
# FUNCTION for the variance
def variance_lbp(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rows, cols = gray.shape
    variance_lbp_image = np.zeros((rows, cols), dtype=np.uint8)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            center = gray[i, j]
            neighborhood = [
                gray[i-1, j-1], gray[i-1, j], gray[i-1, j+1],
                gray[i, j+1], gray[i+1, j+1], gray[i+1, j],
                gray[i+1, j-1], gray[i, j-1]
            ]
            local_variance = np.var(neighborhood)
            binary_str = ''.join(['1' if pixel >= local_variance else '0' for pixel in neighborhood])
            variance_lbp_image[i, j] = int(binary_str, 2)

    return variance_lbp_image
```



GLCM (Gray-Level Co-occurrence Matrix): To analyze and quantify texture patterns in images, which can aid in differentiating malignant and benign tumors based on structural differences.

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import greycomatrix, greycoprops
from skimage import io, color

# Load image (ensure the image is grayscale)
image = io.imread('r"/kaggle/input/tumor-trace-dataset/train/Malignant/BreaDM-Ma-1926/SUB1/p-037.jpg"')

# Convert the image to grayscale if it's not already
gray_image = color.rgb2gray(image) * 255  # Convert to grayscale and scale to [0, 255]

# Convert to integer type (necessary for GLCM computation)
gray_image = gray_image.astype(np.uint8)

# Compute the GLCM matrix
glcm = greycomatrix(gray_image, distances=[1], angles=[0], symmetric=True, normed=True)

# Extract properties from the GLCM matrix
contrast = greycoprops(glcm, prop='contrast')
correlation = greycoprops(glcm, prop='correlation')
energy = greycoprops(glcm, prop='energy')
homogeneity = greycoprops(glcm, prop='homogeneity')

# Print the computed texture features
print(f'Contrast: {contrast}')
print(f'Correlation: {correlation}')
print(f'Energy: {energy}')
print(f'Homogeneity: {homogeneity}')
```



GLCM Properties

Contrast: 5.0629

Correlation: 0.9989

Energy: 0.0456

Homogeneity: 0.5139

These transformations are applied using libraries like OpenCV, PIL, and torchvision, ensuring that the dataset is robust and ready for the training phase. By transforming the data appropriately, we enhance the model's ability to learn and make accurate predictions.

CHAPTER 5 : MODEL ARCHITECTURE

In the Tumor Trace project, we used several convolutional neural network (CNN) architectures to develop the breast cancer detection model using MRI images. The architecture of the model is critical in extracting meaningful features from the images and learning the complex patterns that distinguish between benign and malignant tumors. Below, we describe the architecture of the models used, including VGG16, ResNet18, and ResNet50, which were pre-trained and fine-tuned for this task.

5.1 VGG16 ARCHITECTURE

VGG16 is a deep convolutional neural network that was developed by the Visual Geometry Group (VGG) at the University of Oxford. It is known for its simplicity and depth, making it effective for image classification tasks

Key Characteristics:

- **16 Layers:** The "16" in VGG16 refers to the number of layers with weights (13 convolutional layers and 3 fully connected layers).
- **Convolutional Layers:** These layers extract features from the images using filters, followed by ReLU (Rectified Linear Unit) activations.
- **MaxPooling Layers:** These layers are used to reduce spatial dimensions, preventing overfitting and reducing the computational load.
- **Fully Connected Layers:** After the convolutional layers, the image features are flattened and passed through fully connected layers to make the final prediction.

Architecture Overview:

- **Input Layer:** Images of size 224x224 pixels.
- **Convolutional Layers:** 13 layers with filters of size 3x3 and stride 1, with ReLU activations.
- **Max-Pooling Layers:** After certain convolutional layers to downsample the feature maps.
- **Fully Connected Layers:** Three fully connected layers, the last one having a softmax activation to output class probabilities (benign or malignant).

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

model_name = 'vgg16'
#Customvgg16 class inherit from nn.Module
class CustomVGG16(nn.Module):
    def __init__(self, num_classes=2):
        super(CustomVGG16, self).__init__()

        # Loading pre-trained VGG16 model from tourchvision.models
        vgg16 = models.vgg16(pretrained=True)

        # Extracting the features and avgpool layers frm pretrainedmodel
        self.features = vgg16.features
        self.avgpool = vgg16.avgpool

        # Define a new classifier nm.Sequential
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096), #input and output
            nn.ReLU(inplace=True), #for activation
            nn.Dropout(), #for regularization
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):

        x = self.features(x) # Passing the input through the features layer
        x = self.avgpool(x) # Using the avgpool layer
        x = torch.flatten(x, 1) # Reshaping the output to a 2D tensor
        x = self.classifier(x) # Passing the reshaped output to the custom classifier
        return x

# printing model
if __name__ == "__main__":
    model = CustomVGG16(num_classes=2)
    print(model)
```

5.2 RESNET18 ARCHITECTURE

ResNet18 is a relatively shallow version of the ResNet architecture, consisting of 18 layers. It is designed to have fewer parameters while still maintaining the effectiveness of deep learning models through the use of residual connections.

Key Characteristics:

- **Number of Layers:** ResNet18 has 18 layers, which consist of convolutional layers, residual blocks, and fully connected layers.
- **Residual Blocks:** It uses basic residual blocks, where the input is directly added to the output of the layer, which helps combat the vanishing gradient problem and enables easier training of deeper networks.
- **Block Structure:** The basic block consists of two 3x3 convolutional layers, followed by batch normalization and ReLU activations.

- **Depth:** Being a relatively shallow network compared to other ResNet variants (like ResNet50), ResNet18 is computationally less expensive and faster for training. It is effective for simpler datasets or tasks with fewer features.
- **Bottleneck:** ResNet18 does not use bottleneck blocks (which are used in deeper networks like ResNet50) to reduce the number of parameters. Instead, it relies on the simple two-layer structure of residual blocks.
- **Final Layer:** A fully connected layer with a softmax activation function is used to classify the output into the required categories (benign or malignant in this case).

Architecture Overview:

- **Input Layer**: 224x224 image.
- **Convolutional Layers:** 1 initial convolution layer followed by 4 blocks of residual layers.
- **Residual Blocks:** Each block consists of two 3x3 convolutional layers.
- **Fully Connected Layer:** The output is flattened and passed through a fully connected layer for classification.

```
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 use for deepening the layers in the architecture which will increase

        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
```

```
model_name = 'resnet50'
model = Resnet50(num_classes=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

5.3 RESNET50 ARCHITECTURE

ResNet50 is a deeper version of the ResNet architecture with 50 layers. It is more complex and capable of handling more intricate features due to its increased depth, making it suitable for large and complex datasets.

Key Characteristics:

- **Number of Layers:** ResNet50 consists of 50 layers, which include convolutional layers, residual blocks, and fully connected layers.
- **Residual Blocks (Bottleneck):** Unlike ResNet18, ResNet50 uses bottleneck residual blocks, where each block has three convolutional layers instead of two. The first two convolutions in the bottleneck block reduce the number of channels, and the last convolution restores the original number of channels. This allows ResNet50 to have more layers while keeping the parameter count manageable.
- **Bottleneck Structure:** Each bottleneck block uses 1x1 convolutions to reduce and then restore the number of channels, improving computational efficiency and memory usage.
- **Final Layers:** Similar to ResNet18, ResNet50 also ends with a fully connected layer to classify the images.

Architecture Overview:

- **Input Layer:** 224x224 image.
- **Convolutional Layers:** 1 initial convolution layer followed by 16 bottleneck residual blocks (comprising 3 layers each).
- **Bottleneck Residual Blocks:** Three convolutional layers (1x1, 3x3, 1x1) per block to efficiently handle large numbers of channels.
- **Fully Connected Layer:** The output from the final residual block is passed through a fully connected layer for classification.

```
class Resnet50(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet50, self).__init__()

        model_resnet50 = models.resnet50(pretrained=True)

        self.conv1 = model_resnet50.conv1
        self.bn1 = model_resnet50.bn1
        self.relu = model_resnet50.relu
        self.maxpool = model_resnet50.maxpool

        self.layer1 = model_resnet50.layer1
        self.layer2 = model_resnet50.layer2
        self.layer3 = model_resnet50.layer3
        self.layer4 = model_resnet50.layer4

        self.avgpool = model_resnet50.avgpool
        self._features = model_resnet50.fc.in_features

        self.fc = nn.Linear(self._features, num_classes)
```

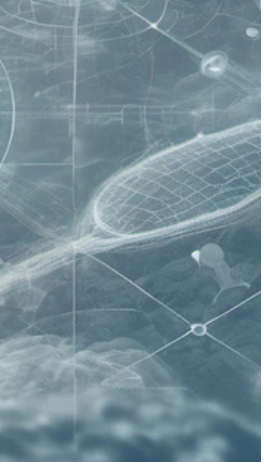
```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)

        x = self.fc(x)

        return x
```

```
model_name = 'resnet50'
model = Resnet50(num_classes=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

5.4 FINE-TUNING THE PRE-TRAINED MODELS

All the models (VGG16, ResNet18, and ResNet50) were pre-trained on the ImageNet dataset, which means they had already learned useful features like edges, textures, and patterns. We fine-tuned these models on the breast cancer MRI dataset to specialize them for our classification task.

- **Freeze the Initial Layers:** The early layers (feature extraction layers) were frozen to retain the knowledge learned from ImageNet. Only the fully connected layers (the classifier part) were trained.
- **Modify the Final Layer:** The final fully connected layer was modified to output two classes (benign and malignant), matching the number of categories in our dataset.
- **Fine-Tuning Strategy:** We used a lower learning rate for the pre-trained layers and a higher learning rate for the newly added layers to ensure that the pre-trained features are not disrupted while training the new classifier.



CHAPTER 6: TRAINING, VALIDATION, AND TESTING

This chapter covers the key components of training, validating, and testing the model. These steps are crucial for ensuring that the model learns correctly from the data and performs well on unseen data. The following sections detail each aspect of this process.

6.1 TRAINING FUNCTION

The Training Function is responsible for teaching the model by adjusting its weights based on the training data. The main steps include:

- **Forward Pass:** The input data is passed through the model to generate predictions.
- **Loss Calculation:** The difference between predicted and actual labels is calculated using a loss function, such as Cross-Entropy Loss.
- **Backward Pass**: The gradients are computed using backpropagation, and the model's weights are updated to minimize the loss.
- **Optimizer:** An optimization algorithm (e.g., Adam, SGD) is used to adjust the weights based on the gradients.
- **Epochs:** The process is repeated for several epochs, with the model gradually improving.

```python
import torch
import torch.optim as optim
from torch.amp import GradScaler, autocast
from tqdm import tqdm

def train(epoch, model, num_epochs, loader, criterion, optimizer, device):
    model.train()
    correct = 0
    total_loss = 0
    scaler = GradScaler()  # Correct use of GradScaler
    optimizer = optim.AdamW(model.parameters(), lr=0.00005, weight_decay=1e-4)

    for data, label in tqdm(loader, desc=f'Epoch {epoch}/{num_epochs}', unit='batch'):
        data, label = data.to(device), label.to(device)

        optimizer.zero_grad()
        with autocast(device_type='cuda'):
            output = model(data)
            loss = criterion(output, label)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(label.view_as(pred)).sum().item()

    accuracy = 100. * correct / len(loader.dataset)
    avg_loss = total_loss / len(loader)
    print(f'Train Epoch: {epoch} \tLoss: {avg_loss:.6f} \tAccuracy: {accuracy:.2f}%')

    return avg_loss, accuracy
```

6.2 VALIDATION FUNCTION

The Validation Function evaluates the model's performance on a validation set during training. This is done after each epoch to check how well the model generalizes to new, unseen data:

- **Forward Pass:** The model generates predictions using the validation data.
- Loss Calculation: The loss is computed for the validation set.
- **Metrics Calculation:** Common metrics like accuracy, AUC, sensitivity, and specificity are used to assess the model's performance.
- **Early Stopping:** If the validation performance does not improve after a certain number of epochs (defined by patience), training can be stopped early to prevent overfitting.

```
import torch
import torch.nn.functional as F
from sklearn import metrics
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve, auc
from torch.cuda.amp import autocast

def validation(model, val_dataloader, criterion, device, plot_metrics=False):
    name = 'val'
    model.eval()  # Set the model to evaluation mode
    val_loss = 0
    correct = 0
    all_predictions = []
    all_targets = []
    probabilities = []

    with torch.no_grad():  # Ensuring no gradients are computed, saving memory and computation
        for data, target in val_dataloader:
            data, target = data.to(device), target.to(device)
            val_output = model(data)  # Getting model's predictions
            loss = criterion(val_output, target)
            val_loss += loss.item() * data.size(0)  # Calculate the loss
            pred = val_output.argmax(dim=1, keepdim=True)  # Get the index of the max log-probability
            all_predictions.extend(pred.cpu().numpy())  # Gets the predicted class for each input
            probs = F.softmax(val_output, dim=1).cpu().data.numpy()  # Softmax is used to convert the range of values in an array between 0-1

            probabilities.append(probs)
            correct += pred.eq(target.view_as(pred)).sum().item()
            all_targets.extend(target.cpu().numpy())  # Collect all target labels

    all_predictions = np.array(all_predictions)
    all_targets = np.array(all_targets)
    probabilities = np.vstack(probabilities)
    cm = metrics.confusion_matrix(all_targets, all_predictions)
    print("Confusion Matrix:\n", cm)


    classification_rep = classification_report(all_targets, all_predictions, target_names=["benign", "malignant"])
    print(f'Classification Report of Validation:\n{classification_rep}\n')

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

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

    val_loss /= len(val_dataloader.dataset)
    accuracy = 100. * correct / len(val_dataloader.dataset)

    print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(1 - fpr[0], tpr[0], auc_score))
    print('\nValidation set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
            val_loss, correct, len(val_dataloader.dataset), accuracy))

    if plot_metrics:
        classification_rep = classification_report(all_targets, all_predictions, target_names=["benign", "malignant"])
        print(f'Classification Report of Validation:\n{classification_rep}\n')

        print("Confusion Matrix of Validation:\n")
        plot_confusion_matrix(cm, classes=['benign', 'malignant'])
        plot_auc(fpr, tpr, auc_score)

    return accuracy, val_loss, auc_score
```

6.3 EARLY STOPPING

Early Stopping is used to prevent overfitting and unnecessary training. This technique monitors the validation loss during training. If the validation loss does not improve after a set number of consecutive epochs (defined by the patience parameter), training is halted early. This helps to avoid overfitting by stopping training when the model starts to memorize the training data, ensuring better generalization to unseen data.

- **Patience:** The number of epochs without improvement in the validation loss before stopping.
- **Best Model Selection:** The model's weights are saved during training when the validation performance is the best, and the model state is restored after early stopping is triggered.
- **Trigger Condition:** Early stopping is triggered if the validation loss does not improve over a specified number of epochs, preventing unnecessary computations and overfitting.

```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 # How long to wait after last time validation loss improved.
        self.verbose = verbose #If true, prints a message for each validation loss improvement.
        self.delta = delta # Minimum change in monotered quality to quantofy improvement
        self.path = path # file path for saving the model checkpoint
        self.trace_func = trace_func # fun to output message
        self.counter = 0 # track of epochs with no improvement.
        self.best_score = None #store the best score
        self.early_stop = False #indicating whether to stop training.
        self.val_loss_min = np.Inf # store the minimum validation loss encountered during training
        #self.best_epoch = 0 #The epoch number of the best model.
        #Epoch is used to describe one complete pass through the entire training dataset.


    def __call__(self, val_loss, model):
        # method is called at end of each epoch to check if the validation loss has improved.
        score = -val_loss

        if self.best_score is None:
            self.best_score = score #the current score is set as the best score
            self.save_checkpoint(val_loss, model)

        elif score < self.best_score + self.delta:
            self.counter += 1
            self.trace_func(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0
```

```python
# checkpoint methon
    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}). save model')
        torch.save(model.state_dict(), self.path)
        self.val_loss_min = val_loss
```

6.4 TEST FUNCTION

The Test Function is used to evaluate the model's performance after training is complete, using a separate test dataset. The test set is unseen during training, which helps to estimate how the model will perform on real-world data:

- **Final Model Evaluation:** The model is tested on the test set to generate final predictions.
- **Metrics Calculation:** Similar to the validation function, metrics like accuracy, AUC, sensitivity, and specificity are computed.
- **Confusion Matrix:** A confusion matrix is generated to visualize the true positives, false positives, true negatives, and false negatives, which helps in understanding model performance.

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

def test(model, test_dataloader, criterion, device):
    name = "test"
    model.eval()  # Set the model to evaluation mode
    test_loss = 0
    correct = 0
    all_predictions = []
    all_targets = []
    all_probabilities = []

    with torch.no_grad():  # Ensuring no gradients are computed, saving memory and computation.
        for data, target in test_dataloader:
            data, target = data.to(device), target.to(device)

            test_output = model(data)  # Getting model's predictions
            loss = criterion(test_output, target)
            test_loss += loss.item() * data.size(0)  # Calculate the loss
            pred = test_output.argmax(dim=1, keepdim=True)  # Get the index of the max log-probability

            all_predictions.extend(pred.cpu().numpy())  # Gets the predicted class for each input.
            all_targets.extend(target.cpu().numpy())  # Get the true labels
            probabilities = F.softmax(test_output, dim=1).cpu().numpy()  # Softmax is used to convert the range of values in an array between 0-1
            all_probabilities.extend(probabilities)

            correct += pred.eq(target.view_as(pred)).sum().item()

    all_predictions = np.array(all_predictions)
    all_targets = np.array(all_targets)
    all_probabilities = np.array(all_probabilities)
```

```
    # Classification metrics - accuracy, f1 score
    classification_rep = metrics.classification_report(all_targets, all_predictions, target_names=['benign', 'malignant'], digits=4)
    print('Classification Report of Test:')
    print(classification_rep)

    # Compute confusion matrix
    cm = metrics.confusion_matrix(all_targets, all_predictions)
    print('Confusion Matrix:')
    print(cm)
    plot_confusion_matrix(cm, ['benign', 'malignant'])

    # ROC AUC
    num_classes = all_probabilities.shape[1]
    label_onehot = np.eye(num_classes)[all_targets.astype(int)]

    fpr, tpr, thresholds = roc_curve(label_onehot.ravel(), all_probabilities.ravel())
    auc_value = metrics.auc(fpr, tpr)
    plot_auc(fpr, tpr, auc_value)

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

    return accuracy, test_loss, auc_value
```

6.5 HYPERPARAMETER TUNING

Hyperparameter Tuning involves optimizing the parameters that control the learning process. These parameters are not learned by the model during training but are manually set before training:

- **Learning Rate:** Determines the step size during gradient descent. A smaller learning rate might take longer to converge, while a larger rate might overshoot the minimum.
- **Batch Size:** Specifies the number of samples in each batch used to compute the gradients.
- **Epochs:** The number of times the entire dataset is passed through the model during training.
- **Regularization:** Techniques like dropout or weight decay can be used to prevent overfitting.
- Hyperparameter tuning can be done manually or using automated methods like Grid Search or Random Search.

```
model.to(device)  # here cudu device
model = CustomVGG16(num_classes=2).to(device)
best_accuracy = 0
total_epochs = 50
momentum = 0.9
no_cuda = False
log_interval = 10
model_name = "vgg16"
# Define optimizer and learning rate scheduler
optimizer = optim.AdamW(model.parameters(), lr=0.00005, weight_decay=1e-4)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=5, verbose=True)
train_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler, num_workers=4)
```

6.6 OPTIMIZER AND LOSS FUNCTIONS

Optimizer: An optimizer is an algorithm used to update the model's weights based on the gradients computed during the backward pass. The most commonly used optimizers are:

- **Stochastic Gradient Descent (SGD):** A simple and efficient optimizer.
- **Adam (Adaptive Moment Estimation):** An adaptive optimizer that adjusts the learning rate for each parameter.
- **RMSprop:** An optimizer that divides the learning rate by a moving average of the recent gradient magnitudes.
- **Loss Function:** A loss function quantifies how well the model's predictions match the actual labels. For classification tasks like tumor detection, the most common loss function is Cross-Entropy Loss, which computes the difference between predicted class probabilities and the true labels.

CHAPTER 7: RESULTS AND IMPACT

7.1 TRAINING RESULTS (LOSS, ACCURACY, AUC):

- This section focuses on how well the model performed during training.
- After each epoch of training, the model's performance is measured on the training dataset. Metrics like training loss, accuracy, and AUC (Area Under Curve) are calculated. These help in monitoring how well the model is learning over time.

Key Metrics:

- **Training Loss:** How far off the model's predictions are from the actual values.
- **Training Accuracy**: The percentage of correct predictions made by the model on the training dataset.
- **AUC (Area Under Curve)**: A performance metric for classification problems, especially for binary classification.

7.2 CONFUSION MATRIX AND EVALUATION METRICS:

After training, the model is evaluated on the validation dataset and test dataset.
Confusion Matrix and evaluation metrics like Sensitivity, Specificity, Precision, Recall, F1-Score, Accuracy, and AUC are computed for the validation and test datasets.

- **Validation Set Evaluation**: This helps to monitor how the model is generalizing and whether it's overfitting. The Validation Function is called during each epoch in the training process.
- **Test Set Evaluation:** After training, the model is evaluated on the test set (which was not seen during training or validation). This is where you get the final performance metrics.

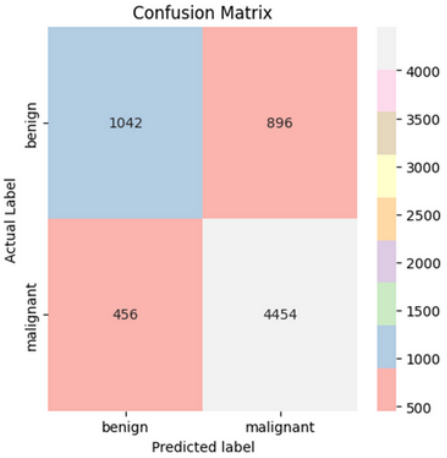
Key Metrics to include:

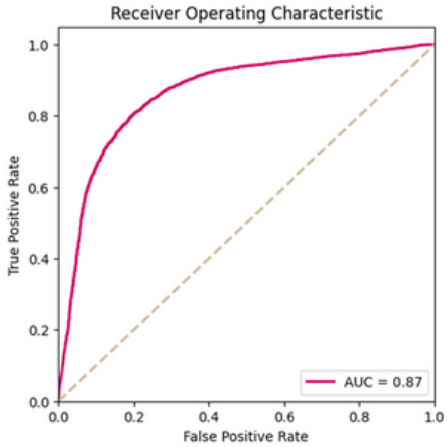
- **Confusion Matrix:** Shows the number of true positives, true negatives, false positives, and false negatives.
- **Accuracy:** Percentage of correctly classified instances.
- AUC: Indicates how well the model distinguishes between classes (benign vs. malignant).
- **Sensitivity and Specificity:** Measures the model's ability to detect positive cases (benign/malignant).
- **F1-Score:** Combines precision and recall into one metric.
- **Precision:** It is about how accurate the positive predictions are.
- **Recall :** It is about how well the model captures all the positive cases.

7.3 COMPARISON OF MODELS (VGG16 VS RESNET18 VS RESNET50)

The performance of VGG16, ResNet18, and ResNet50 on the same dataset.

7.3.1 VGG16 MODEL RESULTS

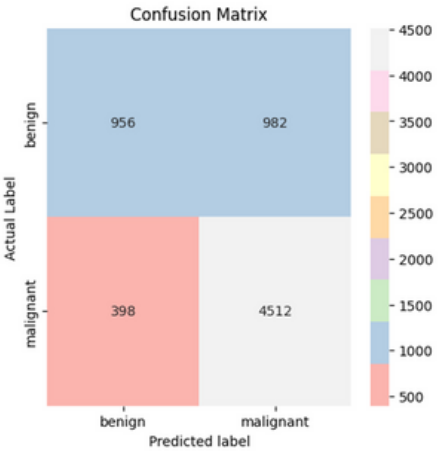


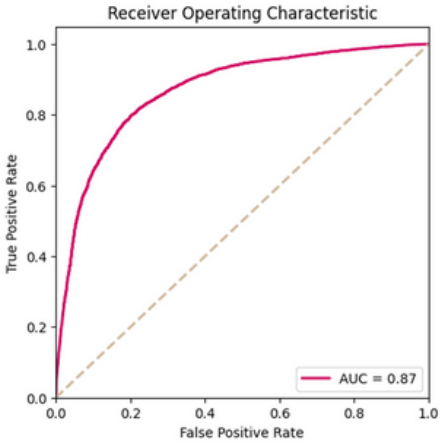


ACCURACY: 80.40% **AUC:** 0.8714

```
Testing the model on the test dataset
Classification Report of Test:
              precision    recall  f1-score   support

      benign     0.6562    0.6481    0.6521      1938
   malignant     0.8618    0.8660    0.8639      4910

    accuracy                         0.8043      6848
   macro avg     0.7590    0.7570    0.7580      6848
weighted avg     0.8036    0.8043    0.8040      6848


Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.8714

test set: Average loss: 0.6977, Accuracy: 5508/6851 (80.40%)
```

7.3.2 RESNET18 MODEL RESULTS

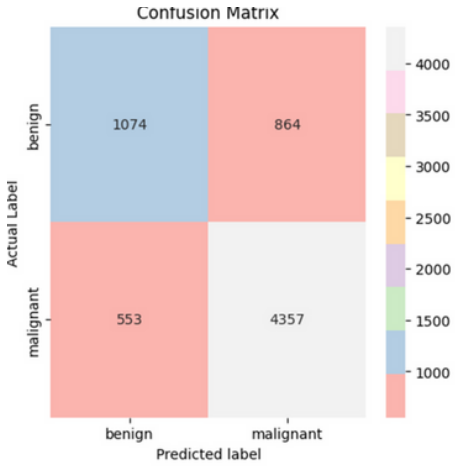


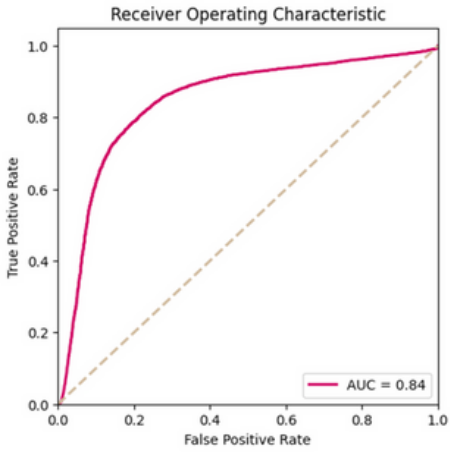


ACCURACY: 77.94% **AUC:** 0.8355

```
Testing the model on the test dataset
Classification Report of Test:
              precision    recall  f1-score   support

      benign     0.5913    0.7188    0.6488      1938
   malignant     0.8787    0.8039    0.8396      4910

    accuracy                         0.7798      6848
   macro avg     0.7350    0.7613    0.7442      6848
weighted avg     0.7973    0.7798    0.7856      6848


Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.8355

test set: Average loss: 0.7059, Accuracy: 5340/6851 (77.94%)
```

7.3.3 RESNET50 MODEL RESULTS





ACCURACY: 78.85% **AUC:** 0.8690

```
Testing the model on the test dataset
Classification Report of Test:
              precision    recall  f1-score   support

      benign     0.6725    0.4948    0.5702      1938
   malignant     0.8194    0.9049    0.8600      4910

    accuracy                         0.7888      6848
   macro avg     0.7460    0.6999    0.7151      6848
weighted avg     0.7779    0.7888    0.7780      6848


Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.8690

test set: Average loss: 1.1090, Accuracy: 5402/6851 (78.85%)
```

7.4 IMPACT OF RESULTS:

The impact of the model's results on early breast cancer detection is significant. High accuracy and AUC scores indicate that the model can reliably distinguish between benign and malignant cases. This reliability can greatly assist healthcare professionals, particularly radiologists, by providing a second opinion that is both consistent and unbiased, reducing diagnostic errors.

Precision ensures that the model minimizes false positives, which is crucial to avoid unnecessary stress and procedures for patients. High recall (sensitivity) means that the model successfully identifies most actual cancer cases, ensuring that fewer cases are missed, which is critical for early detection and treatment. By integrating this model into clinical workflows, the efficiency and effectiveness of breast cancer diagnosis can be enhanced, potentially saving lives and optimizing resource allocation in healthcare facilities.

7.5 LIMITATIONS

- **Data Imbalance:** The dataset may have more malignant cases than benign or vice versa, which can bias the model's performance and affect its ability to generalize.
- **Model Overfitting:** The model may achieve high accuracy on the training set but perform poorly on the test set, indicating that it has memorized the training data rather than learning general patterns.
- **Generalization:** The model's ability to generalize to unseen data can be limited, meaning it might not perform well on different datasets or under varying imaging conditions.
- **Scalability**: Handling large-scale data efficiently can be challenging, especially for complex models like ResNet50, which require significant computational resources.
- **Hardware and Computational Limitations:** Training deep learning models like ResNet50 demands powerful GPUs, which may not be available in all settings, limiting the model's deployment and use.

CHAPTER 8: USER INTERFACE

This chapter discusses the development and implementation of user interfaces for the tumor classification project. It includes interfaces built with Gradio and a custom web application using Flask.

8.1 INTRODUCTION TO GRADIO

Gradio is a user-friendly Python library that allows you to quickly create customizable UI components for machine learning models. It enables easy interaction with models, making them accessible to users who may not have a technical background.

8.2 DESIGNING THE USER INTERFACE WITH GRADIO

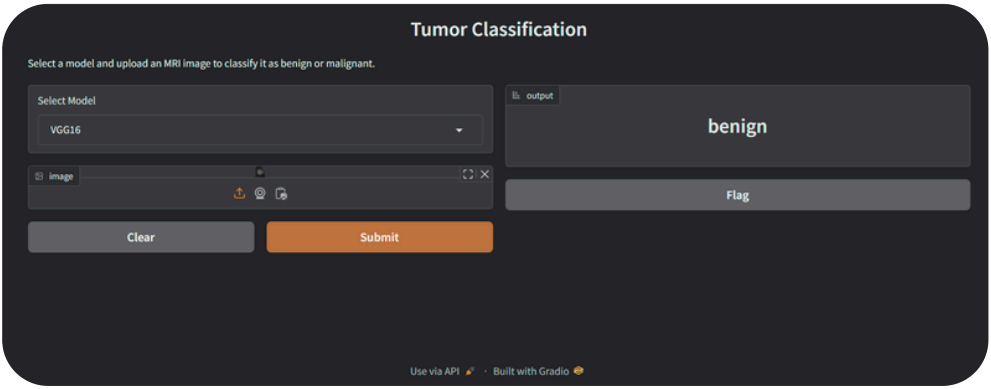
The Gradio interface was designed to allow users to select a model (VGG16, ResNet18, or ResNet50) and upload an MRI image for classification. The interface provides an intuitive dropdown menu for model selection and an image upload button. After uploading an image, the model processes it and returns a prediction indicating whether the tumor is benign or malignant.

```
#The pre-trained weights from vgg16.pth, restnet18.pth, resnet50 are loaded into the model.

vgg16_path = '/kaggle/input/tumor-trace-models/kaggle/VGG16_Model/vgg16_best.pth'
resnet18_path = '/kaggle/input/tumor-trace-models/kaggle/Resnet18_Model/resnet18_best.pth'
resnet50_path = '/kaggle/input/tumor-trace-models/kaggle/Resnet50_Model/resnet50_best.pth'

vgg16_model = CustomVGG16(num_classes=2)
vgg16_model.load_state_dict(torch.load(vgg16_path, map_location = torch.device('cpu')))
vgg16_model.eval() #model is set to evaluation


resnet18_model = Resnet18(num_classes=2)
resnet18_model.load_state_dict(torch.load(resnet18_path, map_location=torch.device('cpu')))
resnet18_model.eval()

resnet50_model = Resnet50(num_classes=2)
resnet50_model.load_state_dict(torch.load(resnet50_path, map_location=torch.device('cpu')))
resnet50_model.eval()
```

```
# Preprocessing the input image to match your modelss' requirements.

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

```python
#Prediction Function
def predict(model_name, image):
    #preprocessing the image
    # adds a new dimension at the 0th position (i.e., the first position).
    img = preprocess(image).unsqueeze(0) # This sets the batch size to 1-> [1, channels, height, width]

    # Based on the user's selection (model_name), the corresponding model is chosen.
    if model_name == "VGG16":
        model = vgg16_model
    elif model_name == "ResNet18":
        model = resnet18_model
    elif model_name == "ResNet50":
        model = resnet50_model
    else:
        return "Invalid model name"


    # Get the model's prediction
    with torch.no_grad(): #disable gradient calculation
        outputs = model(img)
        _, predicted = torch.max(outputs, 1)  #selects the class with the highest probability.

    # Define class labels
    classes = ["benign", "malignant"]
    return classes[predicted.item()]  # Return the predicted label
```

```python
import gradio as gr

# Interface
interface = gr.Interface(
    fn=predict,  # predict function to be called
    inputs=[
        gr.Dropdown(choices=["VGG16", "ResNet18", "ResNet50"], label="Select Model"),
        gr.Image(type="pil")
    ],
    outputs="label",
    title="Tumor Classification",
    description="Select a model and upload an MRI image to classify it as benign or malignant.",
    css=css  # Apply CSS to the interface
)

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



8.3 CUSTOM WEB APPLICATION USING FLASK

In addition to, a custom Web Application was built using Flask to provide a more personalized and flexible user interface. Flask is a lightweight web framework for Python that enables the creation of robust web applications.

Features of the Flask App:

- **Model Selection:** Dropdown menu for selecting the model.
- **Image Upload:** Option to upload an MRI image.
- **Prediction Display:** Displays the classification result (benign or malignant) to the user.

- **DIRECTORY**

```
project/
├── static/
│   └── css/
│       └── styles.css
├── templates/
│   └── index.html
│   └── result.html
├── models/
│   └── vgg16.pth
│   └── resnet18.pth
│   └── resnet50.pth
├── tumortraceinterface.py
```

- **RUN THE FLASK APP**

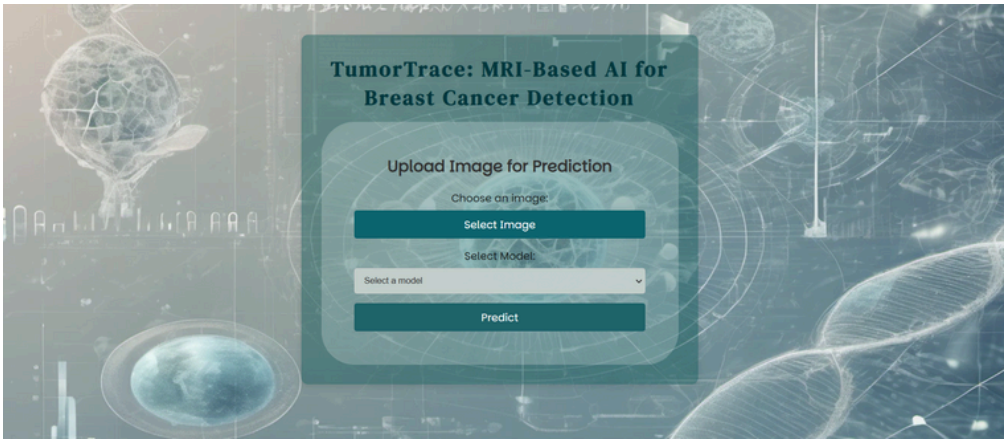
1. Navigate to the project directory: Open a terminal or command prompt and navigate to the project directory.
2. Install the required packages: Ensure you have Flask and PyTorch installed.
3. Run the Flask app:

```
python tumortraceinterface.py
```

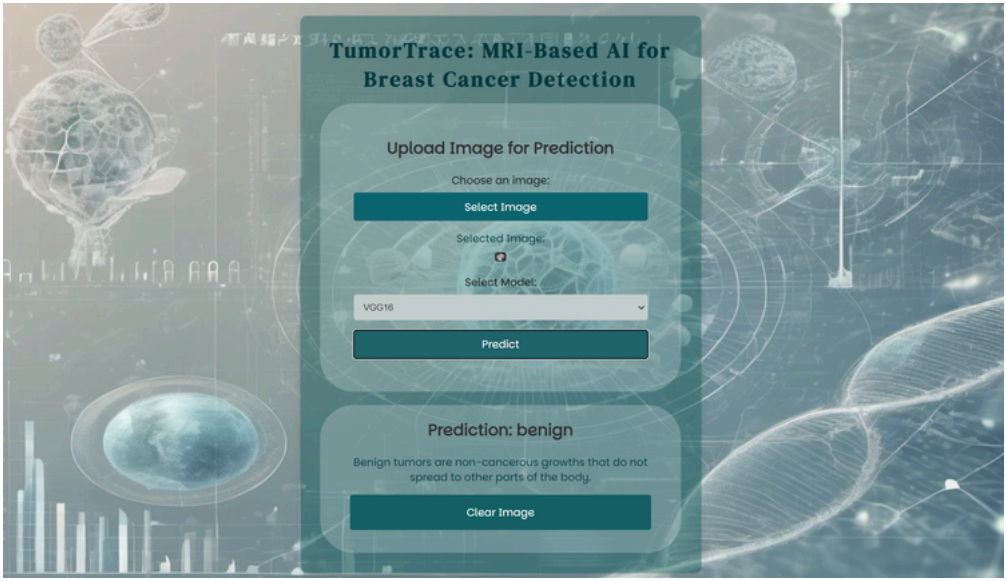
4. Open your web browser and go to http://127.0.0.1:5000/ to use the application.

This setup provides a complete web application for classifying MRI images of tumors using your pre-trained models, with a user-friendly interface and custom styling.

- **WEB APPLICATION**

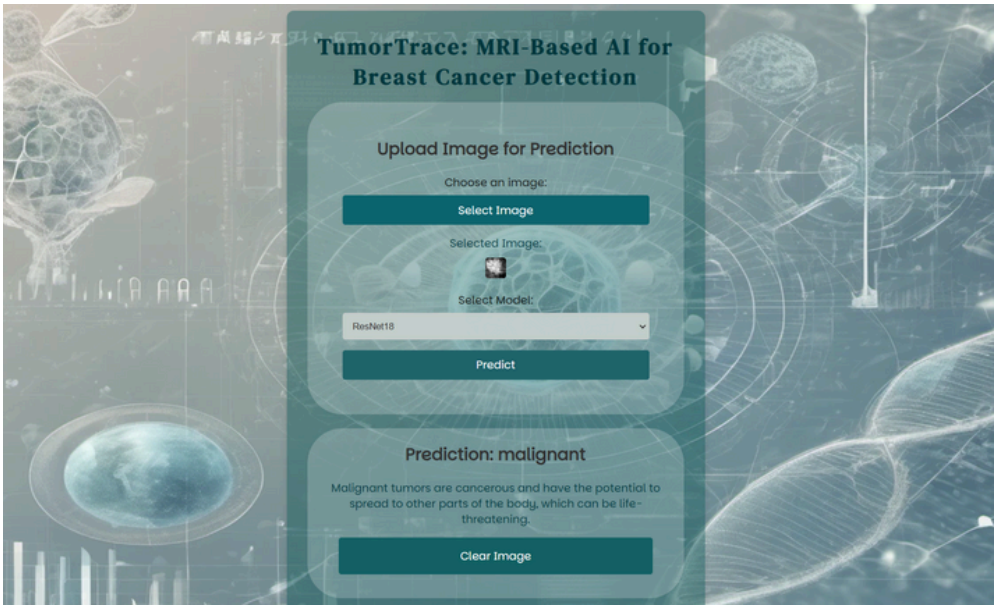


The image was taken from the testing folder - Benign MRI.



The image was uploaded to the application, and the VGG16 model was selected. The output returned was 'Benign,' along with its description.

The image was taken from the testing folder - Malignant MRI.



The image was uploaded to the application, and the Resnet18 model was selected. The output returned was 'Malignant' along with its description.

CHAPTER 9: CONCLUSION AND FUTURE WORK

9.1 SUMMARY OF FINDINGS

In this project, we developed a machine learning model for accurately classifying breast cancer status using MRI images. By leveraging deep learning techniques and using models such as VGG16, ResNet18, and ResNet50, we aimed to aid radiologists in early detection and diagnosis of breast cancer. The models were trained and evaluated on a diverse dataset, demonstrating varying levels of accuracy and performance. The best-performing model was identified based on evaluation metrics including accuracy, AUC, sensitivity, and specificity.

9.2 CONCLUSION

The Tumor Trace project successfully implemented and evaluated multiple deep learning models for classifying MRI images as benign or malignant. The models showed promising results, with the VGG16, ResNet18, and ResNet50 architectures providing significant insights into the potential of AI-assisted diagnostics. The final application, developed using Gradio and Flask, allows users to upload MRI images, select a model, and receive classification results, thereby offering a practical tool for aiding in the early detection of breast cancer. This project demonstrates the feasibility and effectiveness of integrating AI models into medical imaging workflows, potentially improving diagnostic accuracy and patient outcomes.

9.3 FUTURE WORK AND ENHANCEMENTS

Future work on the Tumor Trace project could involve several enhancements to improve model performance and usability:

- **Data Augmentation and Enrichment:** Incorporating more diverse and larger datasets to improve model generalization and robustness. Utilizing advanced data augmentation techniques to create more varied training samples.
- **Model Optimization:** Experimenting with more advanced neural network architectures and fine-tuning hyperparameters to further enhance model accuracy and reduce overfitting.

- **Integration with Medical Systems:** Developing APIs to integrate the AI model with existing medical imaging systems and electronic health records for seamless integration into clinical workflows.
- **User Interface Improvements:** Enhancing the user interface for better usability and accessibility. Adding features such as image preprocessing options, detailed result explanations, and user feedback mechanisms.
- **Real-time Inference and Deployment:** Optimizing the models for real-time inference and deploying them on cloud platforms to ensure scalability and availability for healthcare providers globally.
- **Explainability and Transparency:** Implementing techniques for model explainability to provide insights into how the models make decisions, increasing trust and adoption among healthcare professionals.