Infosys Springboard 5.0

Project:

TumorTrace: MRI-Based AI for Breast Cancer Detection

Subtitle: Using Deep Learning to Identify Benign and Malignant Tumors

Name: Siva Nandini Bommupalla

Role: AI&ML Intern

Abstract:

Breast cancer, a leading cause of cancer-related mortality globally, underscores the significance of early and precise diagnosis. This project endeavors to develop a robust image classification system capable of distinguishing between benign and malignant breast tumors using cutting-edge deep learning techniques.

The project employs a convolutional neural network (CNN) architecture, specifically ResNet18, to train the model on an MRI breast cancer image dataset with distinct labels for benign and malignant cases.

Key methodologies include preprocessing the dataset to ensure optimal input, implementing a ResNet18-based classification model, and applying techniques such as early stopping and regularization to prevent overfitting. While achieving a training accuracy of 80%, validation and test accuracy stabilized at 70%, indicating challenges stemming from the dataset's complexity and the model's generalizability.

This project contributes to the medical imaging field by showcasing the potential of CNNs in automating tumor classification, which can aid radiologists in clinical decision-making. Future endeavors include exploring data augmentation, fine-tuning hyperparameters, and leveraging transfer learning to further enhance the model's performance.

Introduction

1. Background

Breast cancer, one of the most common cancers affecting women globally, accounts for a substantial portion of cancer-related deaths annually. Early detection is crucial in enhancing survival rates and facilitating effective treatment. Traditional diagnostic methods, such as mammography and biopsy, heavily rely on manual interpretation by medical professionals, which can be time-consuming, susceptible to human error, and dependent on the expertise of the radiologist. Recent advancements in deep learning have revolutionized the field by developing automated systems that can assist in early detection and diagnosis with high accuracy and efficiency.

2. Problem Statement

Despite the progress in medical imaging, the manual diagnosis of breast cancer continues to pose significant challenges. The intricate and diverse nature of tumor images poses a considerable obstacle, as does the risk of misclassification, particularly in cases that are subtle or ambiguous. Moreover, the increasing volume of imaging data necessitates a robust and automated solution capable of accurately classifying breast tumor images into benign and malignant categories. This solution should reduce human dependency and streamline diagnostic workflows.

3. Objectives

This project aims to:

1. Develop a deep learning-based image classification system to differentiate between benign and malignant breast tumors.
2. Optimize the classification model using advanced techniques such as regularization, early stopping, and data preprocessing to enhance accuracy.
3. Evaluate the model's performance on metrics such as training accuracy, validation accuracy, and test accuracy, and identify areas for improvement.
4. Explore the potential of automated tumor classification systems to assist radiologists in clinical decision-making.

4. Dataset Overview

1. Source

The dataset used in this project consists of MRI breast cancer images, categorized into two labels: *Benign* and *Malignant*. The images are organized into three subsets: training, validation, and testing, to facilitate the development and evaluation of the classification model.

2. Structure

- **Total Number of Images**:
  - **Train Set**: 20,434 images
  - **Validation Set**: 1,989 images
  - **Test Set**: 6,851 images
- **Labels**:
  - *Benign*
  - *Malignant*
- **Dataset Split**:
  - The dataset is split into **training** (70%), **validation** (10%), and **testing** (20%), ensuring adequate representation of both labels across each subset.

3. Data Loading

The dataset is loaded and processed using PyTorch's DataLoader to handle image loading, batching, and shuffling efficiently. The DataLoader ensures that images are fed into the model in mini-batches, which improves training performance and generalization.

5. Methodology

Preprocessing

Efficient preprocessing is essential to ensure that the MRI breast cancer images are in an optimal format for deep learning. This step standardizes the input data and enhances the model's robustness by incorporating variability during training.

1. Resizing and Normalization

To ensure consistency in image dimensions and improve convergence during training, the following preprocessing steps were applied:

- **Resizing**:
    - All images were resized to a fixed dimension of (224 x 224) pixels to match the input size required by the ResNet18 model.
    - This ensures uniformity across the dataset, as MRI images can vary in resolution and aspect ratio.
- **Normalization:**

Pixel values, originally ranging from 0 to 255, were scaled to a range of 0 to 1 by dividing by 255.

Further normalization was applied using the mean and standard deviation of the ImageNet dataset:

Mean: [0.485, 0.456, 0.406]

Standard Deviation: [0.229, 0.224, 0.225]

This step adjusts the data distribution to match the pre-trained model's expected input format.

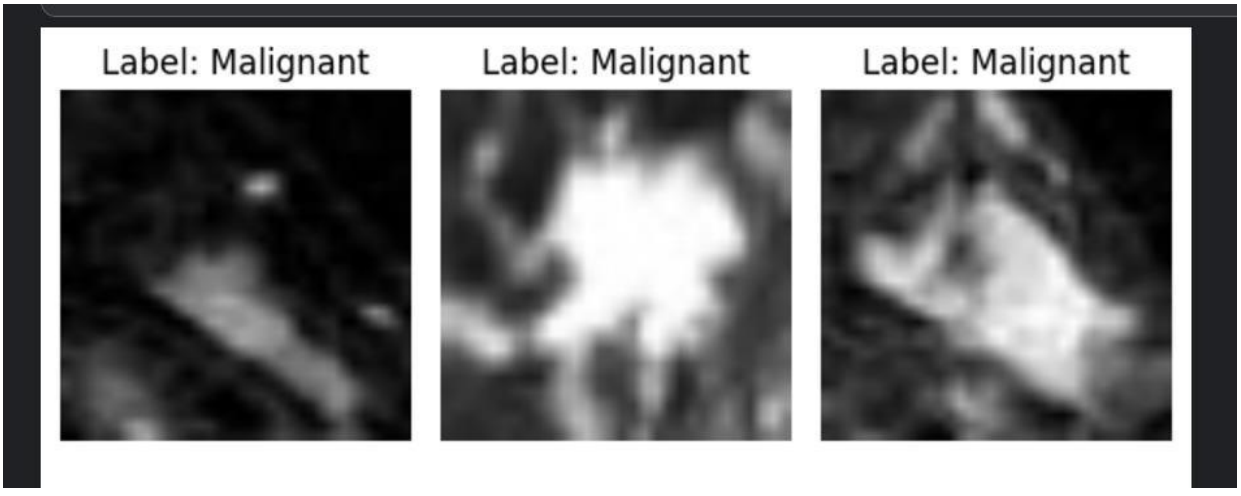
2. Data Augmentation

Data augmentation techniques were applied during training to improve the model’s ability to generalize and handle unseen data. The following augmentations were used:

- **Random Horizontal Flip**:
    - Randomly flipped images horizontally with a probability of 0.5 to introduce spatial variability.
- **Random Rotation**:
    - Images were randomly rotated by up to ±15 degrees to account for orientation differences.
- **Random Resized Crop**:
    - A random crop was applied, resizing the crop to 224 x 224 pixels while maintaining a scale between 80% and 100% of the original size.
- **Color Jitter**:

    - Adjustments to brightness, contrast, saturation, and hue were applied to simulate variations in imaging conditions.
    - The parameters for ColorJitter were set as:
        - Brightness: 0.2
        - Contrast: 0.2
        - Saturation: 0.2
        - Hue: 0.1
- **Conversion to Tensor**:
    - Images were converted to PyTorch tensors for efficient processing.
- **Normalization**:
    - Applied ImageNet mean and standard deviation for scaling pixel values.

Plotting sample images from dataset



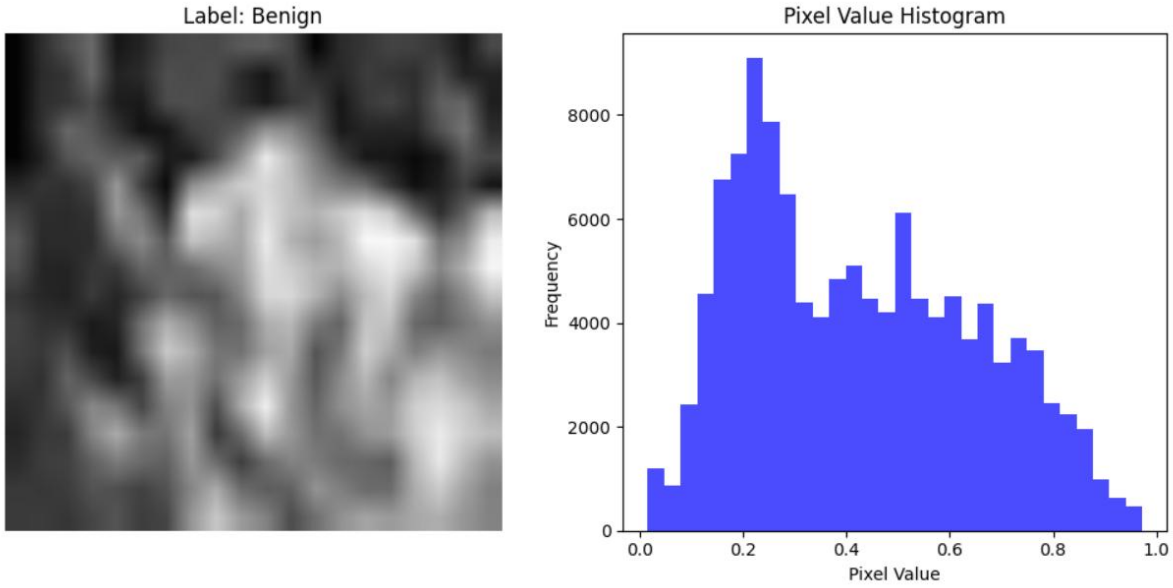
Feature Extraction Techniques

In addition to deep learning-based classification, several feature extraction techniques were applied to preprocess and analyze the MRI breast cancer images. These methods emphasize specific image features such as texture, edges, and intensity distributions, which are critical for distinguishing benign and malignant tumors.

1. Histogram of Pixel Values

- A histogram of pixel values represents the frequency distribution of pixel intensities in an image.
- This helps to understand the overall intensity distribution and contrast of the image, which is vital for enhancing image quality and preparing data for further processing.
- **Libraries Used**: OpenCV, NumPy.

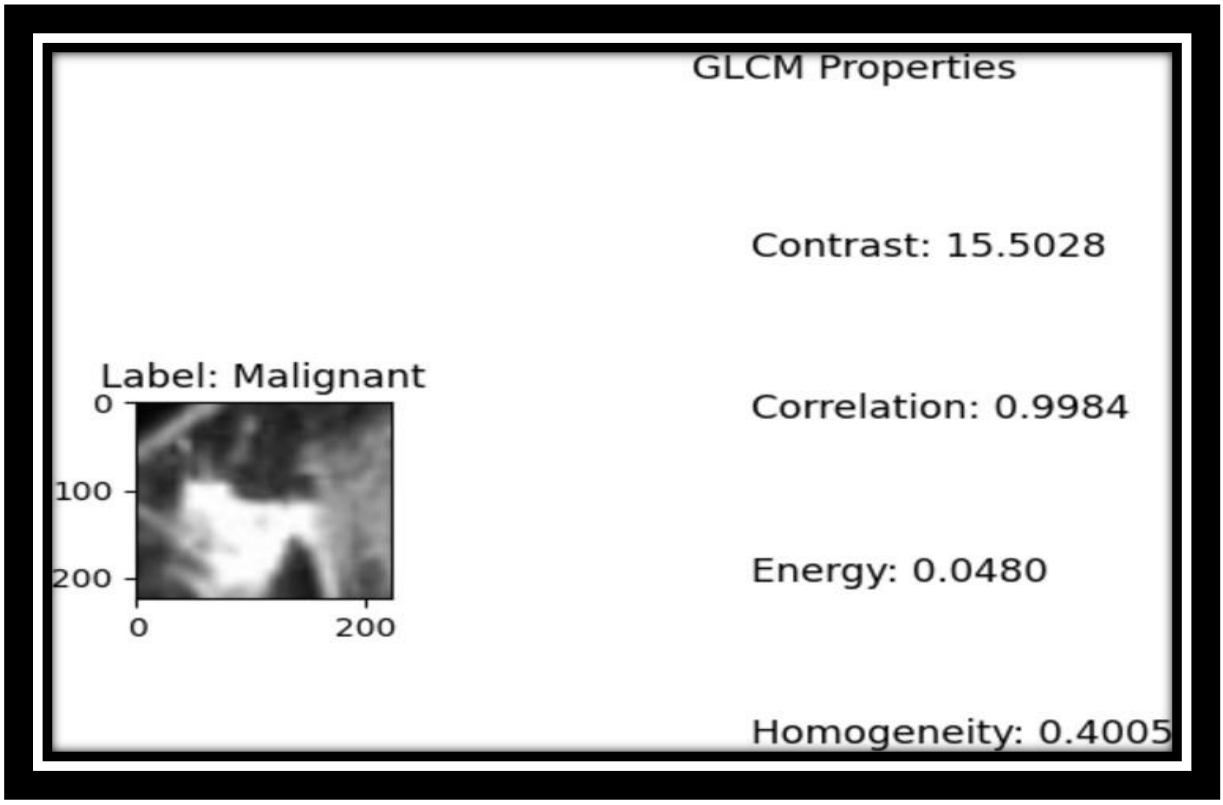
Output image



2. Gray-Level Co-Occurrence Matrix (GLCM) Properties

- GLCM is a statistical method of examining the spatial relationship of pixels in grayscale images.
- Key properties computed include:
  - **Contrast**: Measures intensity contrast between a pixel and its neighbor over the entire image.
  - **Correlation**: Indicates the degree of linear dependency of pixel intensities.
  - **Energy**: Represents textural uniformity (higher energy implies less texture variation).
  - **Homogeneity**: Measures the closeness of pixel distribution in the GLCM.
- **Libraries Used**: skimage.feature.

Output image



Histogram of Oriented Gradients (HOG)

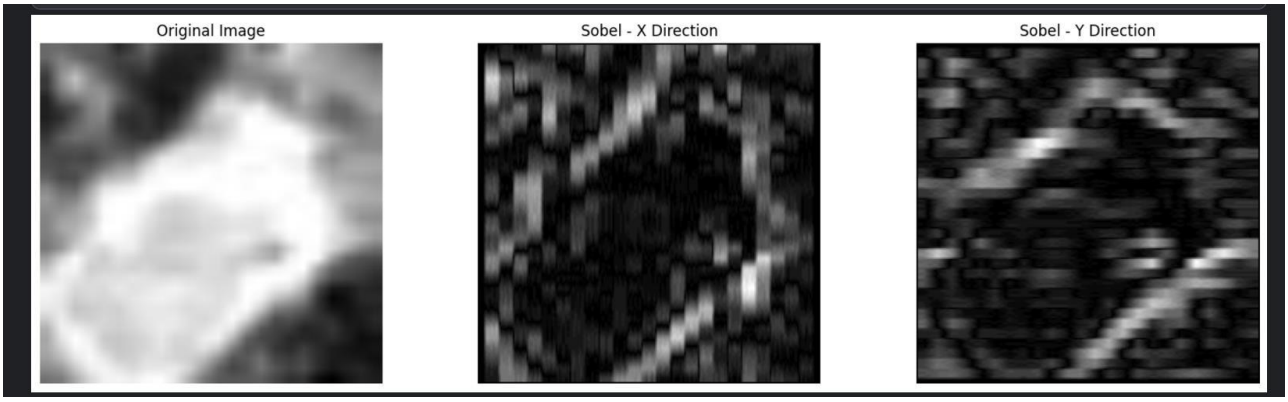
- HOG focuses on the structure and shape of objects in an image by counting the occurrences of gradient orientations in localized portions of the image.
- Used for:
  - Visualizing edge directions and intensity gradients.
  - Extracting features that highlight object outlines.
- **Libraries Used**: skimage.feature.

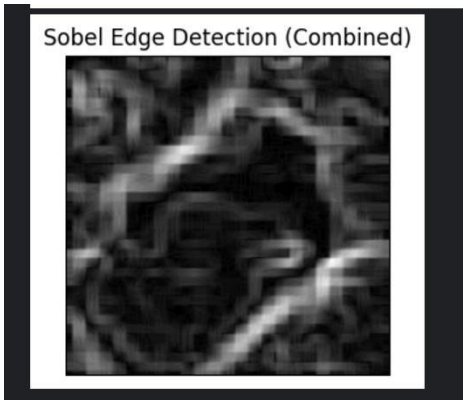
Output image

4. Sobel Operator

- The Sobel operator computes the gradient magnitude and direction for an image, emphasizing edges.
- It helps identify sharp intensity changes, which are critical for detecting tumor boundaries.
- The Sobel operator calculates derivatives in both horizontal and vertical directions (SobelX and SobelY).
- **Libraries Used**: OpenCV.

Output image

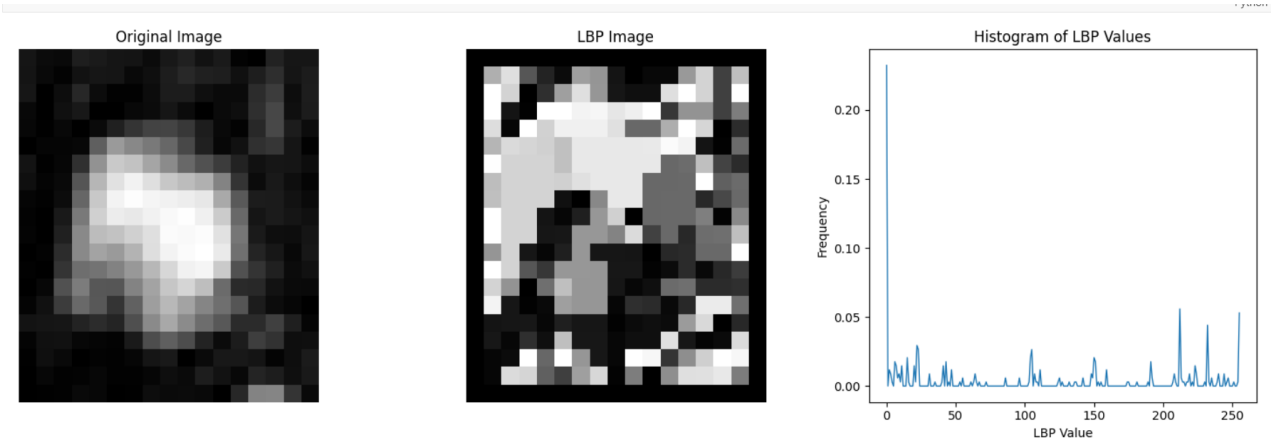




5. Local Binary Pattern (LBP)

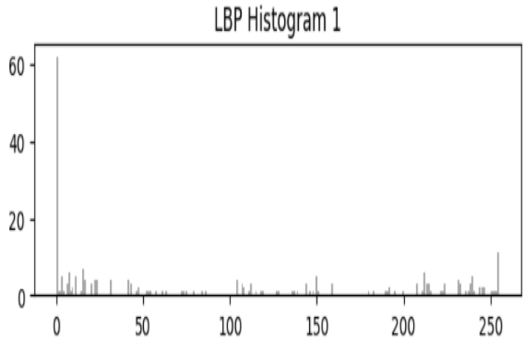
- LBP is a texture descriptor that captures local intensity patterns by comparing each pixel with its neighbors.
- Useful for capturing micro-patterns in tumor textures.
- Variants of LBP computed include:
  - **Basic LBP**: Based on binary thresholds around the center pixel.
  - **Mean-Based LBP**: Incorporates the mean intensity in the neighborhood for more robust feature extraction.
  - **Variance and Median-Based LBP**: Measures the statistical properties of LBP distributions for a deeper analysis of texture.
- **Libraries Used**: skimage.feature.

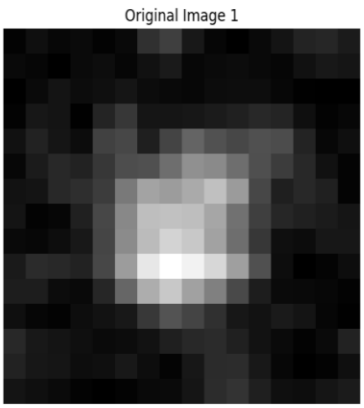
Output image



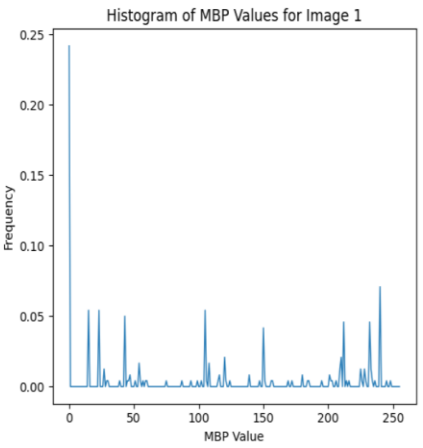

Original Image 1


LBP Image with Variance 50.89


LBP Histogram 1
60
40
20
0
0
50
100
150
200
250


Original Image 1

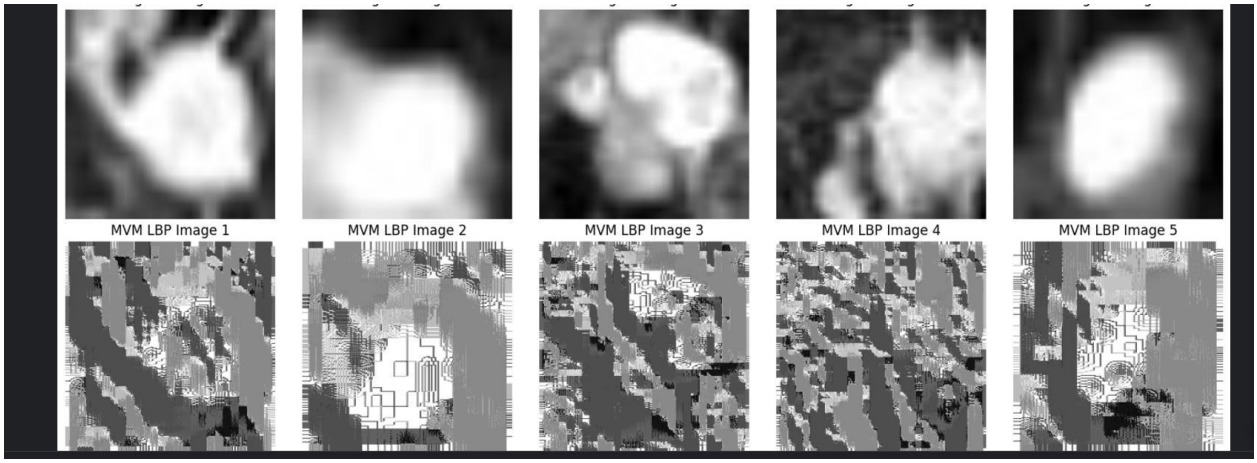

MBP Image 1


Histogram of MBP Values for Image 1
Frequency
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
MBP Value

6. Multi-Variance Median LBP (MVM-LBP)

- **Overview**:
  - Extends traditional LBP by combining multiple statistical properties such as mean, variance, and median.
  - Captures complex texture patterns that reflect fine-grained tumor features.
  - Useful for enhancing texture representation, particularly for distinguishing tumors with subtle differences.
- **Advantages**:
  - Provides richer texture details than basic LBP.
  - Robust to noise and variations in intensity.
  - Effective for distinguishing heterogeneous tumor regions.
- **Implementation**:
  - Compute mean, variance, and median values in a local neighborhood.
  - Generate LBP codes for each statistic and aggregate them into a composite feature.
  - This composite MVM-LBP feature provides enhanced discriminatory power.

Output image



Model Architecture

1. Custom VGG16 Model

Overview:

The VGG16 architecture is a popular convolutional neural network known for its simplicity and effectiveness in image classification tasks. It uses a stack of convolutional layers followed by fully connected layers, making it ideal for hierarchical feature extraction.

Key Features and Modifications:

- **Pre-trained Base:** The pre-trained VGG16 model on ImageNet is utilized to leverage transfer learning, improving performance with limited data.
- **Custom Classifier:**
  - The original classifier was replaced with a custom fully connected layer tailored to this binary classification task (benign vs. malignant).
  - Includes two hidden layers of size 4096, ReLU activation functions, dropout layers for regularization, and a final output layer with two nodes.

- **Advantages:**
    - VGG16 is straightforward to modify and adapt for different tasks.
    - The depth of the network allows capturing intricate patterns in the input images.

- **Why Use This Model:**
    - The simplicity of the architecture ensures ease of training and debugging.
    - Its ability to generalize well on moderately complex datasets makes it an excellent baseline model.

Model code

```
import torch
import torch.nn as nn
import torchvision.models as models
class CustomResNet18(nn.Module):
    def __init__(self, num_classes=2): # You can change num_classes if required
        super(CustomResNet18, self).___init__()

        # Load the pre-trained ResNet18 model
```

```
        model_resnet18 = models.resnet18(pretrained=True)

        # Keep the convolutional, batch normalization, and maxpool
layers as is
        self.conv1 = model_resnet18.conv1
        self.bn1 = model_resnet18.bn1
        self.relu = model_resnet18.relu
        self.maxpool = model_resnet18.maxpool

        # Retaining the deeper layers (layer1 to layer4)
        self.layer1 = model_resnet18.layer1
        self.layer2  =  model_resnet18.layer2
        self.layer3  =  model_resnet18.layer3
        self.layer4 = model_resnet18.layer4
        # Adaptive average pooling (for flattening the output from the
last layer)
        self.avgpool = model_resnet18.avgpool

        # The number of input features from the last ResNet block
        self.features = model_resnet18.fc.in_features

        # Change the fully connected layer (fc) to fit your custom
```

```
number of classes
        # If you're doing binary classification, you can leave it as is (num_classes=2)
        self.fc = nn.Linear(self.features, num_classes)


    def forward(self, x):
        # Forward pass through the layers
        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)

        # Apply average pooling to the output
        x = self.avgpool(x)

        # Flatten the output (reshape the tensor)
        x = x.view(x.size(0), -1)
```

```
# Pass through the custom fully connected layer (fc)
x = self.fc(x)


return x
```

2. Custom ResNet18 Model

Overview:

ResNet18, a lightweight version of the ResNet family, introduces residual connections, which address the vanishing gradient problem and allow for deeper networks without degrading performance.

Key Features and Modifications:

- **Pre-trained Base:** The backbone ResNet18 model, pre-trained on ImageNet, is used for feature extraction.
- **Custom Fully Connected Layer:**
  - The final fully connected layer was replaced with a custom layer to match the number of classes (2 for this project).
- **Advantages:**
  - Residual connections improve gradient flow and learning.
  - ResNet18 is computationally efficient, making it suitable for faster training on systems without GPUs.

- **Why Use This Model:**
    - Offers a balance between performance and computational cost.
    - Works effectively even with relatively smaller datasets, as it prevents overfitting by design.

Model code

```
import torch

import torch.nn as nn

import torchvision.models as models
class CustomResNet18(nn.Module):

    def __init__(self, num_classes=2): # You can change num_classes if required

        super(CustomResNet18, self).___init__()


        # Load the pre-trained ResNet18 model

        model_resnet18 = models.resnet18(pretrained=True)


        # Keep the convolutional, batch normalization, and maxpool layers as is

        self.conv1 = model_resnet18.conv1

        self.bn1 = model_resnet18.bn1
```

```
        self.relu = model_resnet18.relu
        self.maxpool = model_resnet18.maxpool

        # Retaining the deeper layers (layer1 to layer4)
        self.layer1 = model_resnet18.layer1
        self.layer2  =  model_resnet18.layer2
        self.layer3  =  model_resnet18.layer3
        self.layer4 = model_resnet18.layer4
        # Adaptive average pooling (for flattening the output from the last layer)
        self.avgpool = model_resnet18.avgpool

        # The number of input features from the last ResNet block
        self.features = model_resnet18.fc.in_features

        # Change the fully connected layer (fc) to fit your custom number of classes
        # If you're doing binary classification, you can leave it as is (num_classes=2)
        self.fc = nn.Linear(self.features, num_classes)

    def forward(self, x):
```

```
# Forward pass through the layers
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)

# Apply average pooling to the output
x = self.avgpool(x)

# Flatten the output (reshape the tensor)
x = x.view(x.size(0), -1)

# Pass through the custom fully connected layer (fc)
x = self.fc(x)
return x
```

3. Custom ResNet50 Model

Overview:

ResNet50, a deeper version of ResNet18, extends the capacity of the network to learn more complex patterns and features, thanks to its 50-layer architecture.

Pre-trained Base: Like ResNet18, the pre-trained ResNet50 model on ImageNet is used as the starting point.

- **Custom Fully Connected Layer:**
  - The original fully connected layer was replaced with one suitable for binary classification.
- **Advantages:**
  - Depth allows capturing more abstract features in the data.
  - Residual connections make it robust to the vanishing gradient issue, despite its depth.
- **Why Use This Model:**
  - Ideal for datasets with complex patterns or when the computational power is sufficient for training deeper models.

```
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)
        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
```

Comparative Analysis and Decision

- **VGG16 Custom Model:** Offers simplicity and reliability but may underperform on datasets with high complexity compared to ResNet architectures.
- **ResNet18:** Provides an efficient trade-off between performance and resource usage, ideal for systems without GPUs.
- **ResNet50:** Offers the best feature extraction capacity due to its depth but requires higher computational power.

The final model was selected based on **validation accuracy** and **generalization performance** to ensure the best classification outcomes.

Training Process Overview

The breast cancer image classification model's training optimizes performance and prevents overfitting, using a loss function, optimizer, learning rate schedule, and early stopping mechanism.

Training Loop

The training loop is implemented using the PyTorch framework. It performs several key tasks: iterating through the training data, computing the loss, performing backpropagation, and updating model weights.

1. **Data Loading**
   The training data is provided via a PyTorch DataLoader instance (train_loader). Each batch contains image data and corresponding labels, which are passed through the model for training.

2. **Loss Function**
   A **Cross-Entropy Loss** function is used to calculate the difference between the predicted class probabilities and the true labels. This loss is suitable for multi-class classification tasks, including binary classification in this case (benign vs. malignant).

3. **Optimizer**
   The **SGD (Stochastic Gradient Descent)** optimizer is used for weight updates. The optimizer adjusts the model's weights based on the gradients computed during backpropagation. It uses:

   - **Learning Rate**: The learning rate is dynamically adjusted every 10 epochs by reducing it by a factor of 0.1, ensuring that the model converges smoothly over time.

- **Momentum**: Momentum is set to 0.9, which helps accelerate gradient vectors in the right directions and dampens oscillations.
- **L2 Regularization (Weight Decay)**: To avoid overfitting, an L2 penalty is applied with a weight decay factor of 0.01.

4. **Training Process**
    - For each epoch, the model is set to training mode, and the loss and accuracy are calculated for each batch of images.
    - For each batch, the image data and labels are moved to the GPU for faster computation (using .cuda()), and the model performs a forward pass to produce predictions.
    - After calculating the loss, the optimizer's gradients are zeroed, and backpropagation is performed to update the model's weights.
    - The training accuracy is computed by comparing the predicted labels to the true labels, and this information is used for tracking the model's performance.

Learning Rate Scheduling

The learning rate is adjusted dynamically during training to improve convergence. Initially set to 0.01, the learning rate is reduced by a factor of 0.1 every 10 epochs. This step-down approach helps the model converge more smoothly and avoids

overshooting during the later stages of training. The learning rate is also capped at a minimum value of 1e-5 to ensure it doesn’t become too small and impede learning.

Early Stopping

To prevent overfitting and save computational resources, an **Early Stopping** mechanism is used. This technique monitors the model's validation loss and halts training if the loss stops improving after a certain number of epochs (patience).

1. **Patience**: The patience parameter defines how many epochs the training should continue without improvement in the validation loss before stopping. In this case, patience is set to 5 epochs, meaning if the validation loss doesn't improve for 5 consecutive epochs, training will be stopped early.
2. **Saving the Best Model**: If the validation loss improves, the model weights are saved. This ensures that the model with the best performance (lowest validation loss) is retained for future inference.
3. **Stopping Condition**: The training stops when the model's performance does not improve beyond a certain threshold (delta), which helps avoid overfitting by preventing the model from continuing to train unnecessarily when it is no longer learning useful patterns.

Batch Size

The batch size is set to **32** during training. This choice strikes a balance between training speed and model performance. A larger batch size may lead to more stable gradient estimates, while a smaller batch size may help the model generalize better by introducing more noise into the optimization process.

Performance Monitoring

Throughout the training process, the model's performance is closely monitored, primarily through training accuracy. By tracking how well the model performs on the training dataset, adjustments can be made to the learning rate, optimizer, or other hyperparameters to improve results. The accuracy is printed out at the end of each epoch for real-time performance tracking.

This structured training process, combined with a dynamic learning rate schedule and early stopping, ensures efficient training of the model while preventing overfitting and improving generalization for the breast cancer image classification task

Validation and Testing Evaluation

Validation Phase

The validation process is conducted after each epoch during the model training to monitor its generalization capability. The validation function evaluates the model on the validation dataset, which was not part of the training set. Here are the key components of the validation procedure:

1. Model Evaluation Mode
   The model is set to evaluation mode (model.eval()) to ensure that layers like dropout or batch normalization behave accordingly during inference.

2. Loss Calculation
   The negative log likelihood loss (nll_loss) is used to compute the loss on the validation set. This is a suitable loss function for multi-class classification problems and is used to measure how well the predicted probabilities align with the true class labels.

3. Predictions
   The predicted class labels are obtained by finding the maximum value in the output tensor using max(1)[1], which gives the class with the highest probability. These predictions are accumulated for later analysis.

4. Softmax Probabilities for AUC
   The softmax function is applied to the output to convert the model's logits into probabilities, which are used for computing the ROC curve and AUC (Area Under the Curve)

score. The probabilities are collected across all validation batches.

5. Metrics
    - Confusion Matrix: A confusion matrix is computed to understand the true positives, false positives, true negatives, and false negatives. This helps in understanding model performance in detail.
    - AUC: The ROC Curve and AUC score are computed to evaluate the trade-off between sensitivity and specificity across different classification thresholds. A higher AUC indicates better model performance.
    - Specificity and Sensitivity: These metrics are calculated to understand how well the model performs in detecting both positive and negative classes. Sensitivity (recall) measures the model's ability to identify positive cases, while specificity measures its ability to identify negative cases.

The overall test loss is averaged over all validation samples, and the accuracy is computed as the percentage of correct predictions.

Testing Phase

After the training process, the model is evaluated on the test dataset, which is a completely separate dataset that the model has never seen before.

The key components of the testing process are:

1. Model Evaluation Mode
   The model is set to evaluation mode (model.eval()) to disable training-specific behaviors like dropout.

2. Loss Calculation
   Similar to the validation phase, the negative log likelihood loss (nll_loss) is used to calculate the loss on the test set.

3. Predictions and Probabilities
   Predictions are made by selecting the class with the highest probability for each test sample. These predictions, along with the true labels, are stored for further analysis.

4. ROC Curve and AUC
   The ROC Curve and AUC score are computed based on the predicted probabilities and the true labels. These metrics provide insight into how well the model distinguishes between classes across various decision thresholds.

5. Classification Report
   The classification report provides a detailed breakdown of precision, recall, and F1 score for each class, helping to assess how well the model performs on each individual class. This is especially useful in imbalanced datasets where accuracy alone may not be sufficient.

6. Confusion Matrix
   The confusion matrix is computed to understand how many samples were misclassified as false positives or false negatives, providing valuable insights into the model's weaknesses.

7. Specificity, Sensitivity, and Accuracy
    - Specificity is calculated as 1 - False Positive Rate (FPR). It measures the ability of the model to correctly identify negative instances.
    - Sensitivity (Recall) is calculated as the True Positive Rate (TPR), indicating the model's ability to correctly identify positive instances.
    - The overall accuracy is reported, which is the percentage of correct predictions in the test set.

Summary of Evaluation Metrics

Accuracy: Measures the proportion of correct predictions.

Precision: Measures the proportion of true positives out of all predicted positives.

Recall (Sensitivity): Measures the proportion of true positives out of all actual positives.

F1 Score: Provides a balanced measure of precision and recall.

Specificity: Measures the proportion of true negatives out of all actual negatives.

ROC-AUC: Provides a measure of the model's ability to distinguish between positive and negative classes.

Confusion Matrix: Provides a detailed summary of the model's classification performance across all classes.

By analyzing these metrics, we can understand how well the model is performing on the test data and determine whether it is ready for deployment or needs further improvement. These evaluations help identify any biases or performance issues, such as misclassification of certain classes, which can guide future improvements in model design and training.

6. Results

In this section, we present the performance of the model evaluated using several key metrics, including accuracy, loss, confusion matrix, and AUC score. The following components highlight the evaluation process and provide insights into the model's behavior on both the validation and test datasets.

Performance Metrics

1. **Training and Validation Accuracy/Loss**
    - The **training accuracy** and **validation accuracy** give us a measure of how well the model fits the training data and generalizes to unseen validation data. It is important to track these metrics during training to detect overfitting or underfitting.
    - **Training Loss**: The training loss is tracked during the training process to monitor how well the model is learning from the training data.
    - **Validation Loss**: The validation loss is computed after each epoch to assess the model's performance on the validation dataset. This helps to detect potential

overfitting if the validation loss begins to increase while the training loss decreases.

2. **Confusion Matrix**
   The **confusion matrix** is calculated for both the validation and test datasets to evaluate the number of correct and incorrect classifications across all classes. It provides a detailed view of the true positives, false positives, true negatives, and false negatives. This is especially useful for multi-class classification tasks, where a more granular view of the model's performance is required.
   - For example, a confusion matrix for a binary classification model could look like this:

[[True Negatives, False Positives],

[False Negatives, True Positives]]

3. **ROC Curve and AUC**
   The **Receiver Operating Characteristic (ROC) curve** and **Area Under the Curve (AUC)** are used to evaluate the model's ability to distinguish between classes across different thresholds. A higher AUC indicates a better ability of the model to distinguish between positive and negative classes.
   - The **AUC score** is computed for both the validation and test sets, and it reflects the model's overall classification performance across all possible thresholds.

4. **Specificity and Sensitivity**

- **Sensitivity** (or recall) is the proportion of true positives correctly identified by the model. This metric is critical in situations where it is important to identify as many positive samples as possible.
- **Specificity** measures the proportion of true negatives correctly identified by the model, which is important in avoiding false positives.

Examples of Correct and Incorrect Predictions

- **Correct Predictions**: These are instances where the model correctly classifies the data into the correct class. You can include a few examples showing the image, predicted label, and true label.
- **Incorrect Predictions**: These are instances where the model misclassifies the data. Including examples of such misclassified instances can provide insights into potential areas of improvement, such as specific classes the model struggles to identify.

Model Evaluation Results

In this section, we provide the results for three different models: VGG16, ResNet18, and ResNet50. The evaluation includes the training and testing accuracy, loss, AUC, and other performance metrics such as precision, recall, F1-score, confusion matrix, specificity, and sensitivity.

Model 1: VGG16

Training Accuracy:

82.9402080566406%

Validation Results:

Average Loss: 0.4494

Accuracy: 82.55%

AUC: 0.8669

Test Results:

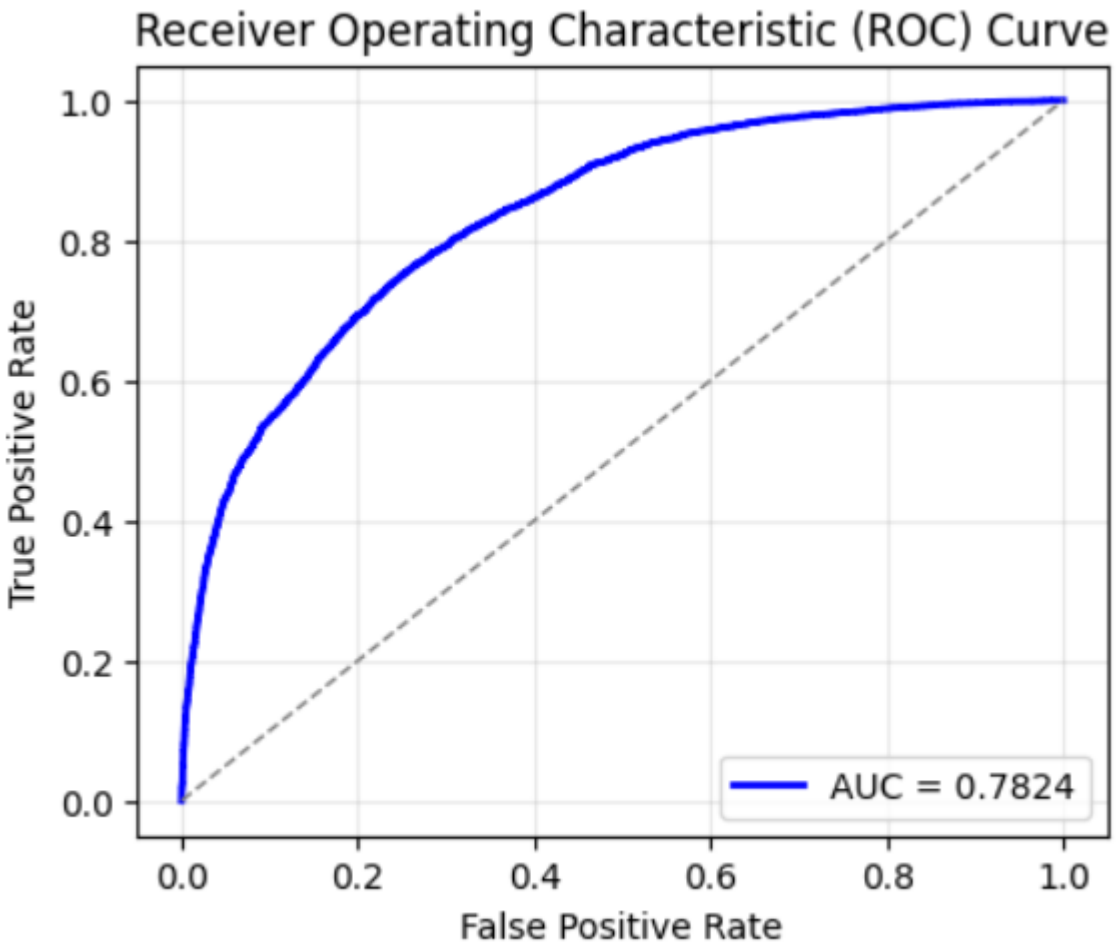
Average Loss: 0.5015

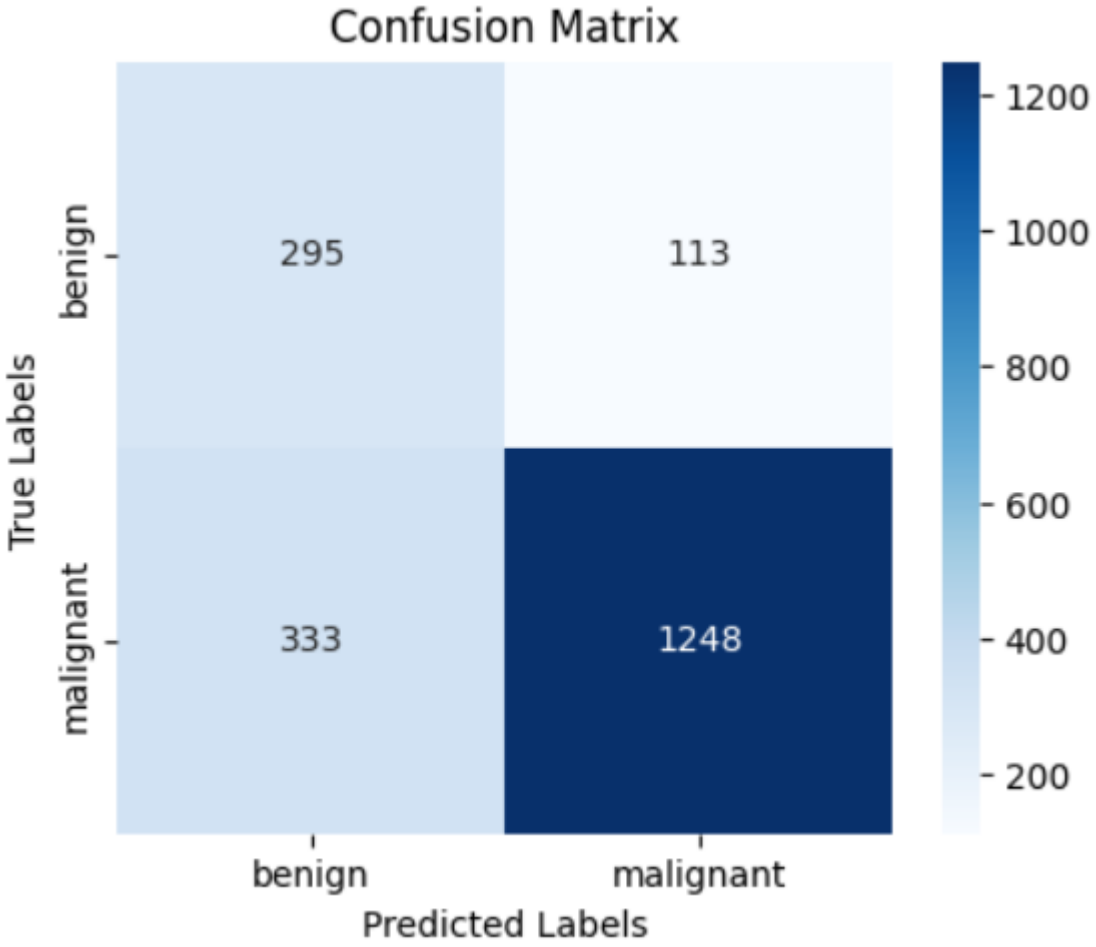
Accuracy: 77.58% (5301/6851)

Classification Report:

```
test_accuracy, test_loss, test_auc, fpr, tpr = test(model, test_loader)
```

```
              precision    recall  f1-score   support

      benign     0.5640    0.5186    0.5403      1938
   malignant     0.8159    0.8418    0.8287      4913

    accuracy                         0.7504      6851
   macro avg     0.6900    0.6802    0.6845      6851
weighted avg     0.7447    0.7504    0.7471      6851


Confusion Matrix:
 [[1005  933]
 [ 777 4136]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7824

test set: Average loss: 0.5217, Accuracy: 5141/6851 (75.04%)
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.7824


Confusion Matrix
True Labels
benign
malignant
295
113
333
1248
benign
malignant
Predicted Labels
1200
1000
800
600
400
200

Model 2: ResNet18

Training Accuracy: 97.86%

Validation Results:

Average Loss: 0.3862

Accuracy: 84.22

AUC: 0.8084

Test Results:

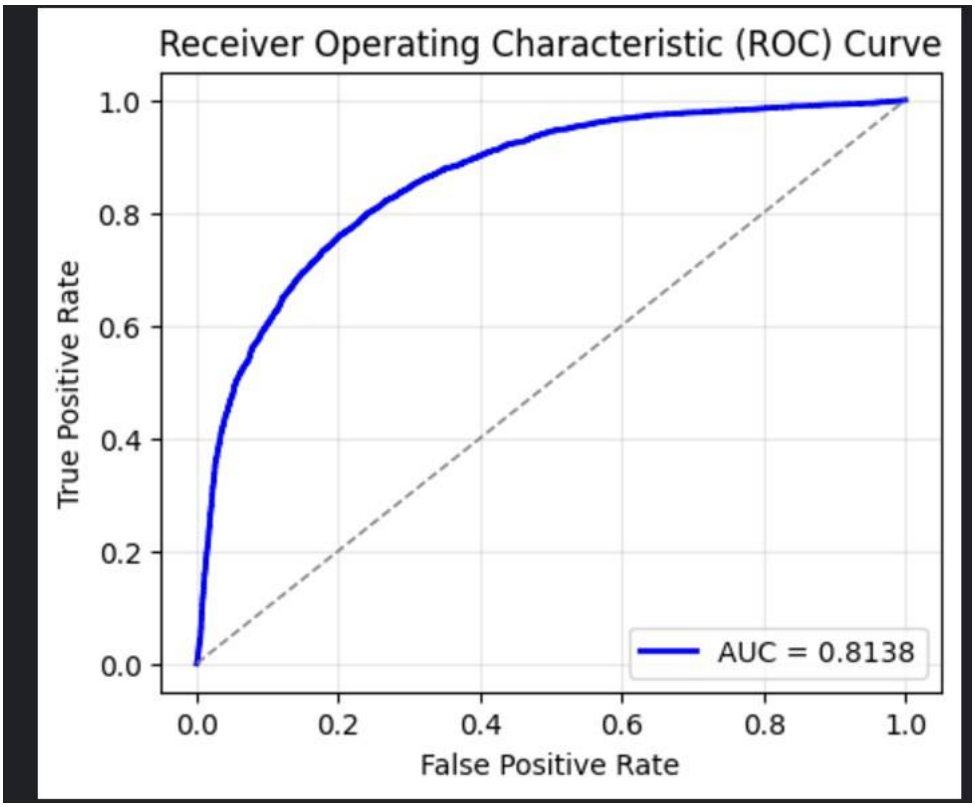
Average Loss: 0.6204

Accuracy: 77.14% (5285/6851)

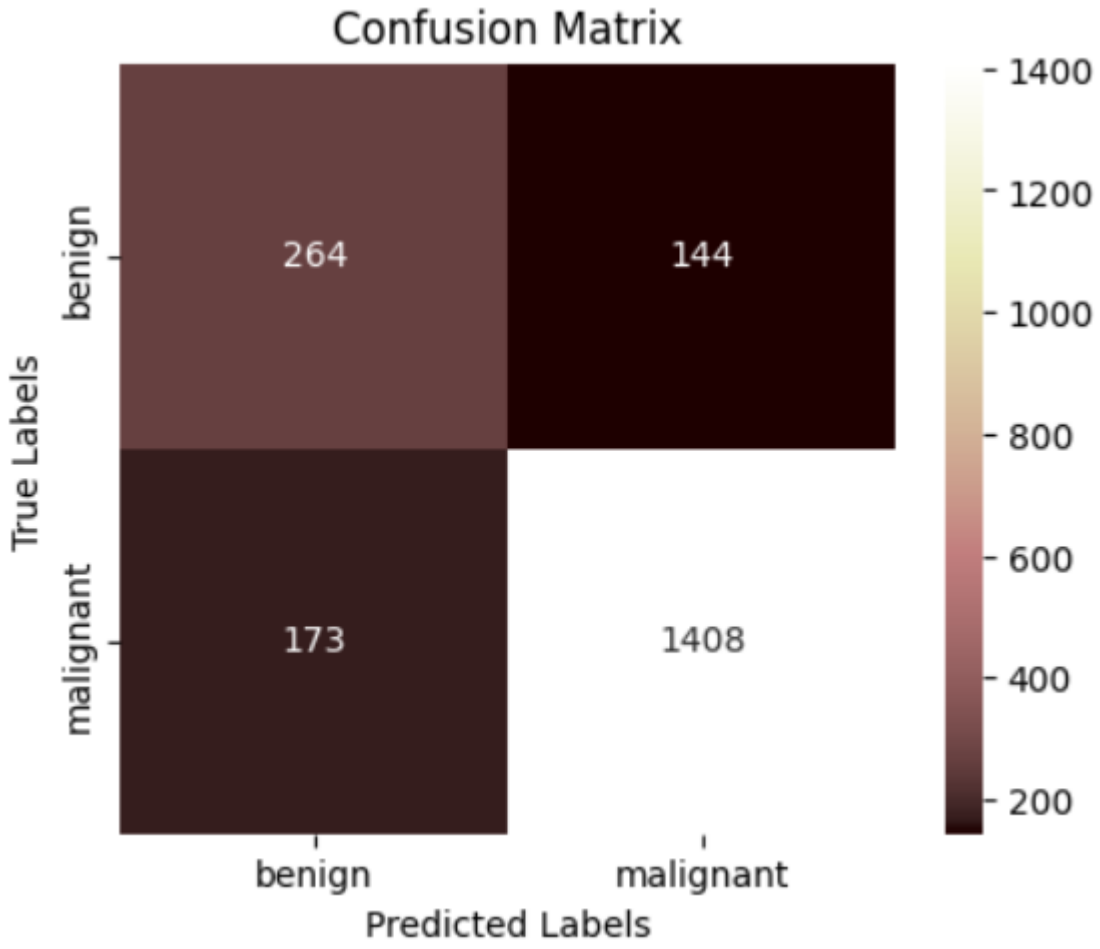
Classification Report:

```
test_accuracy, test_loss, test_auc, fpr, tpr = test(model, test_loader)
```

```
              precision    recall  f1-score   support

      benign     0.6338    0.4546    0.5294      1938
   malignant     0.8064    0.8964    0.8490      4913

    accuracy                         0.7714      6851
   macro avg     0.7201    0.6755    0.6892      6851
weighted avg     0.7576    0.7714    0.7586      6851


Confusion Matrix:
 [[ 881 1057]
 [ 509 4404]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.8084

test set: Average loss: 0.6559, Accuracy: 5285/6851 (77.14%)
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.8138

...


Confusion Matrix
264
144
173
1408
benign
malignant
True Labels
benign
malignant
Predicted Labels
1400
1200
1000
800
600
400
200

Model 3: ResNet50

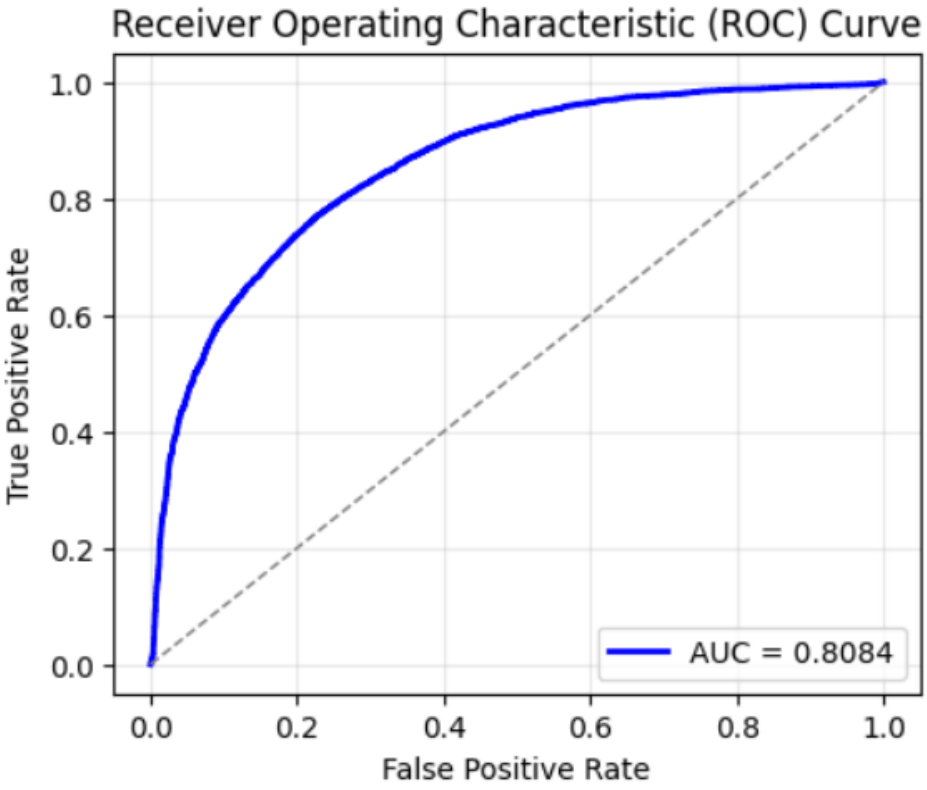
- **Training Accuracy**: 93.69%
- **Validation Results**:
  - **Average Loss**: 0.3713
  - **Accuracy**: 84.44%
  - **AUC**: 0.9235%
  -

Test Results:

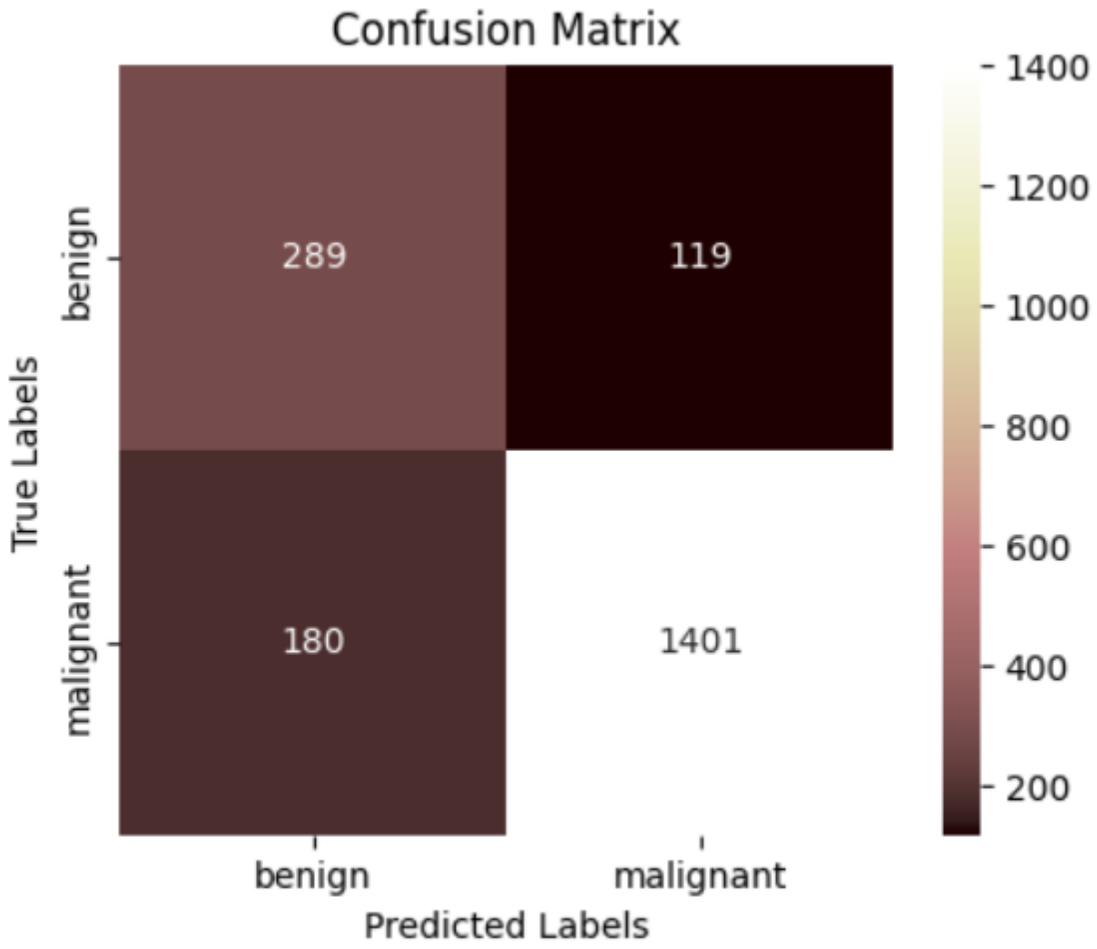
- Average Loss: 0.3862
- Accuracy: 84.92%
- AUC: 0.8084

```
test_accuracy, test_loss, test_auc, fpr, tpr = test(model, test_loader)
```

```
              precision    recall  f1-score   support

      benign     0.5913    0.4912    0.5366      1938
   malignant     0.8119    0.8661    0.8381      4913

    accuracy                         0.7600      6851
   macro avg     0.7016    0.6786    0.6874      6851
weighted avg     0.7495    0.7600    0.7528      6851


Confusion Matrix:
 [[ 952  986]
 [ 658 4255]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7912

test set: Average loss: 0.6254, Accuracy: 5207/6851 (76.00%)
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.8084

...



Gradio part implementation

```
import gradio as gr
import torch
import torchvision.transforms as transforms
from PIL import Image
from torchvision.models import resnet18
import torch.nn as nn

class CustomResNet18(nn.Module):
    def __init__(self, num_classes=2):
        super(CustomResNet18, self).__init__()
```

```
        model_resnet18 = resnet18(pretrained=True)
        self.conv1 = model_resnet18.conv1
        self.bn1 = model_resnet18.bn1
        self.relu = model_resnet18.relu
        self.maxpool = model_resnet18.maxpool
        self.layer1 = model_resnet18.layer1
        self.layer2 = model_resnet18.layer2
        self.layer3 = model_resnet18.layer3
        self.layer4 = model_resnet18.layer4
        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

# Load model
model = CustomResNet18(num_classes=2)
model.load_state_dict(torch.load('/content/resnet18_final_epoch
```

```
(2).pth', map_location=torch.device('cpu')))

#model.load_state_dict(torch.load("/content/resnet18_final_epoch (2).pth", map_location=torch.device('cpu')))
#model.load_state_dict(torch.load('/content/resnet18_final_epoch (2).pth'))
model.eval()

class_names = ["Benign", "Malignant"]

# Preprocessing for input images
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]),
])

def predict(image):
    image = preprocess(image).unsqueeze(0)  # Add batch dimension
    with torch.no_grad():
        output = model(image)
        probabilities = torch.nn.functional.softmax(output[0], dim=0)
    predicted_class = probabilities.argmax().item()
    return {class_names[0]: float(probabilities[0]), class_names[1]: float(probabilities[1])}

interface = gr.Interface(
    fn=predict,
    inputs=gr.Image(type="pil"),
    outputs=gr.Label(num_top_classes=2),
```

```
    title="Binary Classification with Custom ResNet18",
    description="Upload an image to classify it as 'Benign' or 'Malignant' using a pretrained ResNet18 model.",
)

if __name__ == "__main__":
    interface.launch()
```

Output