INFOSYS SPRINGBOARD 5.0

Project TumorTrace:
MRI-Based AI for Breast Cancer Detection

[Infosys Springboard 5.0]

Mentor: - Anurag Sista

Name : M U DARSHAN

COMPANY : Infosys

Role : AI/ML Intern

LinkedIn : https://www.linkedin.com/in/mu-darshan

Github : https://github.com/DARSHANVIT

Duration : 7th October 2024 – December 2024

Ai/Ml Internship Assignments & Tasks :

Abstract

The purpose of this project is to develop and evaluate a machine learning model for classifying breast cancer using MRI images, aiming to provide a robust tool for early and accurate cancer detection. Breast cancer, a

leading cause of cancer-related deaths, requires advanced diagnostic techniques to ensure early identification and treatment. The project leverages deep learning, specifically Convolutional Neural Networks (CNNs), to distinguish between benign and malignant breast tumors, offering potential assistance for medical professionals in clinical decision-making.

For this purpose, a custom dataset consisting of labeled MRI images of breast cancer cases is used to train and evaluate three different deep learning models: VGG16, ResNet18, and ResNet50. These models, each with distinct architectural features, are trained to classify the images into benign and malignant categories. The project carefully monitors several performance metrics, including accuracy, loss, area under the curve (AUC), confusion matrix, specificity, and sensitivity, to assess the models' effectiveness in differentiating between tumor types.

The training process involves advanced optimization techniques to improve the model's performance and generalization. These techniques include learning rate scheduling, early stopping to prevent overfitting, and batch size adjustments to ensure efficient model training. The results of these experiments are expected to provide insights into the strengths and weaknesses of each architecture and help identify the most effective deep learning approach for breast cancer classification from MRI images. This project holds significant promise in advancing medical imaging for cancer detection, potentially aiding in more reliable and timely diagnoses.

Introduction

1. Background

Breast cancer remains one of the most prevalent cancers globally, contributing significantly to cancerrelated mortality rates, particularly among women. Early detection is critical for improving survival rates and ensuring effective treatment options. Traditional diagnostic methods, such as mammography and biopsy, rely heavily on manual interpretation by medical professionals. These approaches, while effective, are prone to human error, can be time-consuming, and heavily depend on the expertise of the radiologists. To address these limitations, automated systems leveraging deep learning techniques, such as Convolutional Neural Networks (CNNs), are being developed to improve the accuracy and efficiency of breast cancer diagnosis.

By applying CNNs to medical imaging data, we can significantly reduce diagnostic times and help in detecting benign and malignant tumors with higher precision. These automated systems can assist radiologists by offering a supplementary tool for quick, accurate diagnoses, especially when large volumes of data are involved.

2. Problem Statement

Despite advancements in medical imaging, manual detection of breast cancer faces several challenges:

- Complexity and Variability of Tumor Images: The appearance of tumors can vary significantly in terms of size, shape, and texture, making it difficult to apply uniform diagnostic standards.
- Misclassification Risks: Subtle or ambiguous images can be misclassified as benign or malignant, especially in cases with overlapping characteristics.
- Volume of Imaging Data: The increasing amount of data generated by imaging techniques places significant strain on medical professionals, necessitating automated solutions.

Given these challenges, automated, deep learningbased solutions are essential to reduce dependency on human interpretation, mitigate misclassification risks, and streamline the diagnostic process.

3. Objectives

This project aims to develop and evaluate a deep learning-based classification system for MRI breast cancer images, with the following key objectives:

1. **Develop a Deep Learning-based Classification Model**: A system that differentiates between benign and malignant breast tumors based on MRI scans.

2. **Optimize the Model's Performance:** Using techniques such as regularization, early stopping, and data preprocessing (normalization, resizing) to enhance accuracy and prevent overfitting.

3. **Evaluate Performance on Key Metrics:** Metrics such as training accuracy, validation accuracy, test accuracy, sensitivity, specificity, and AUC will be used to assess the model's effectiveness.

4. **Explore the Potential of Automated Classification Systems:** The project aims to investigate the usefulness of these automated systems in assisting radiologists with quicker, more reliable tumor classification.

4. Dataset Overview

1.**Source :**

https://www.sciencedirect.com/science/article/abs/pii/S0010482523007205

The dataset used for this project consists of MRI breast cancer images, labeled into Benign and Malignant categories. This dataset is used to train and evaluate the model. The images are part of a custom dataset divided into three sets: training, validation, and testing.

2. Structure

- Total Number of Images:

- Training Set: 20,434 images o Validation Set: 1,989 images o Testing Set: 6,851 images

- Labels:
  - Benign o Malignant
- Dataset Split:
  - The dataset is split into 70% training, 10% validation, and 20% testing. This distribution ensures that the training set is large enough for the model to learn effectively, while the validation and testing sets provide sufficient data for model evaluation.

3. Data Loading

The images are processed using PyTorch's DataLoader for efficient handling. The DataLoader takes care of batching, shuffling, and loading the images. This allows the images to be fed into the model in mini-batches, which not only speeds up training but also improves the generalization of the model by reducing variance during training.

4. Data Preprocessing

The preprocessing pipeline was adapted specifically for the dataset in this project and includes the following steps:

1. Resizing and Normalization:

- All images were resized to 224 x 224 pixels, matching the required input size for the ResNet18 model.
- The pixel values, originally ranging from 0 to 255, were scaled down to a range of 0 to 1 by dividing by 255.
- Further normalization was applied using the ImageNet dataset's mean and standard deviation:
  - Mean: [0.485, 0.456, 0.406]
  - Standard Deviation: [0.229, 0.224, 0.225]

2. Data Augmentation:

- To increase the diversity of the dataset and improve the model's generalization capabilities, the following augmentations were applied during training:
  - **Random Horizontal Flip:** Applied with a probability of 0.5.

- **Random Rotation:** Images were rotated randomly by up to ±15 degrees.
- **Random Resized Crop: A** random crop was taken and resized to 224 x 224 pixels.
- **Color Jitter:** Adjusted parameters for brightness, contrast, saturation, and hue:
- Brightness: 0.2, Contrast: 0.2, Saturation: 0.2, Hue: 0.1.

3. Conversion to Tensor:

- The images were converted to PyTorch tensors, which are needed for the deep learning pipeline.

4. Batching and Loading:

- Once preprocessed, the images were batched and shuffled using DataLoader for efficient model training.

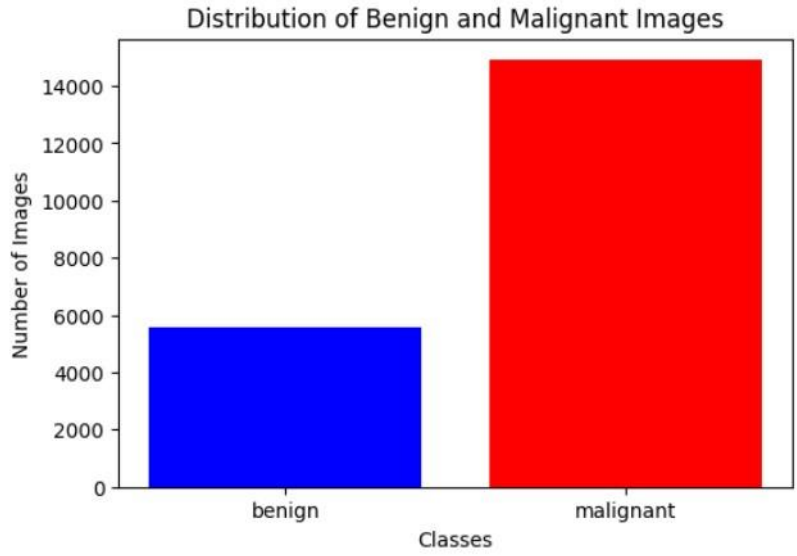
5. Model Architecture

The classification model used for the project is based on three CNN architectures: VGG16, ResNet18, and
ResNet50. These models were chosen for their ability to extract hierarchical features from images, enabling the effective classification of complex medical images.

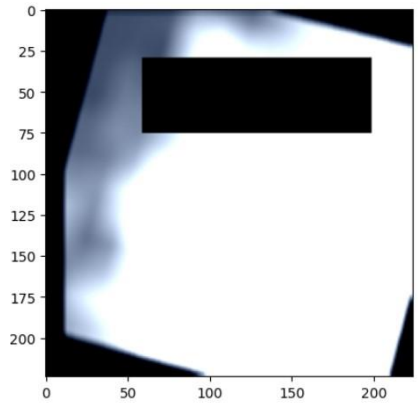
By leveraging these CNN architectures and the preprocessing steps outlined above, the project aims to develop a model capable of distinguishing between benign and malignant breast cancer tumors with high accuracy. The performance of the models is evaluated using key metrics such as accuracy, sensitivity, specificity, and AUC to assess their clinical viability.

OUTPUT: -

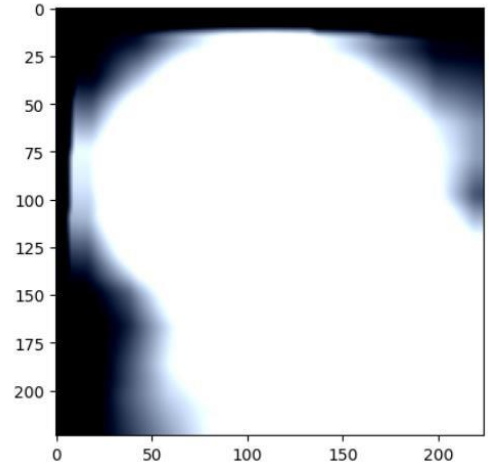
```
Total Benign Images: 5559
Total Malignant Images: 14895
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Augmented Images from Training Set:
```



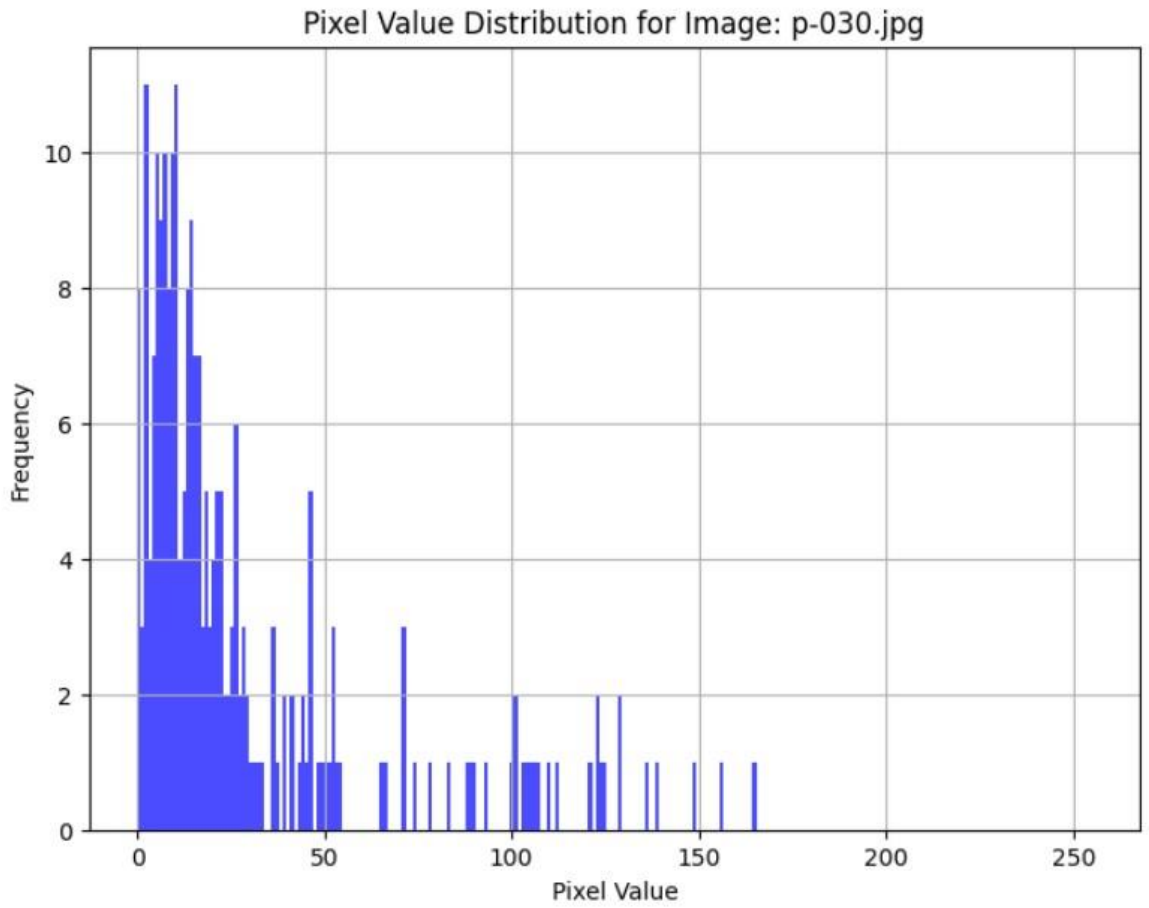
Feature Extraction Techniques

In addition to deep learning-based classification, this project also leverages several feature extraction techniques to preprocess and analyze MRI breast cancer images. These methods focus on extracting specific image features, such as texture, edges, and intensity distributions, which are essential for distinguishing between benign and malignant tumors. Feature extraction helps in enhancing the model's ability to recognize subtle tumor characteristics that may otherwise be overlooked.

1. Histogram of Pixel Values

A histogram of pixel values represents the distribution of pixel intensities across an image. This feature extraction method is crucial for understanding the overall intensity distribution, contrast, and brightness in an image. It allows for the enhancement of image quality by adjusting contrast or brightness levels, which prepares the data for further processing and classification.

- Libraries Used: OpenCV, NumPy
- Output: Histogram plot showing the frequency of pixel intensities across the image.



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

GLCM is a statistical technique used to analyze the spatial relationship between pixels in grayscale images. This method calculates texture features by examining how pixel intensities change in relation to their neighbors. Key properties derived from GLCM include:

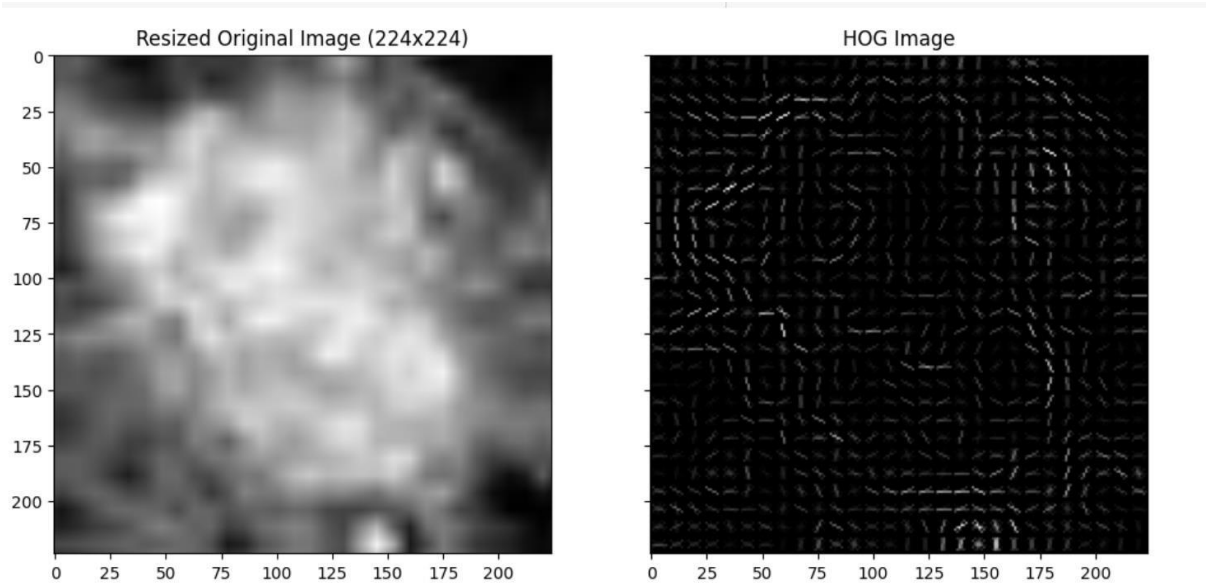
- Contrast: Measures the intensity contrast between a pixel and its neighbor.
- Correlation: Indicates the degree of linear dependency between pixel intensities.
- Energy: Represents the uniformity of texture (higher energy implies less texture variation).
- Homogeneity: Measures the closeness of pixel intensity distributions in the image.
- Libraries Used: skimage.feature

- Output: GLCM features such as contrast, correlation, energy, and homogeneity.

3. Histogram of Oriented Gradients (HOG)

HOG focuses on the structure and shape of objects within an image. It calculates gradient orientations across localized regions of the image, which helps in visualizing edge directions and intensity changes. This feature extraction technique is useful for identifying tumor boundaries and outlines.

- Libraries Used: skimage.feature
- Output: A histogram showing the distribution of gradient orientations, highlighting structural details of objects (such as tumor boundaries).

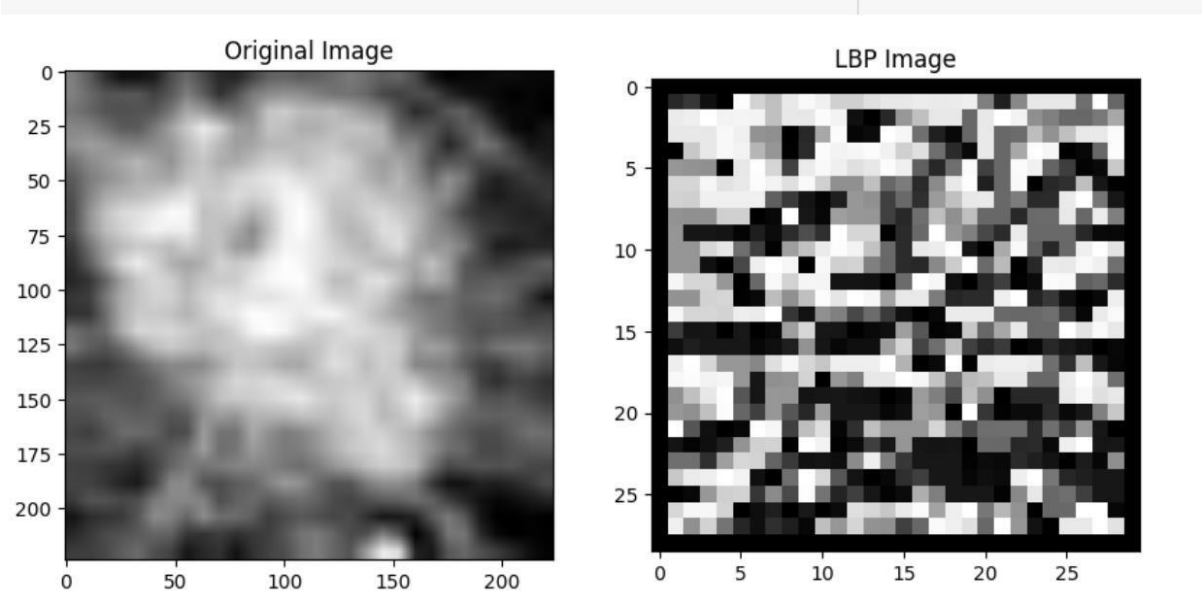


4. Local Binary Pattern (LBP)

LBP is a texture descriptor that captures local intensity patterns by comparing each pixel with its neighboring pixels. It is effective in highlighting micro-patterns that are characteristic of tumor textures. Several LBP variants are used:

- Basic LBP: Computes binary thresholds around each pixel to capture local texture.

- Mean-Based LBP: Incorporates the mean intensity of the surrounding pixels to enhance feature extraction.
- Variance and Median-Based LBP: Measures the statistical properties (variance, median) of LBP distributions for more robust texture analysis.
- Libraries Used: skimage.feature
- Output: LBP texture map showing the local patterns of pixel intensities in the image.



- ---

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

MVM-LBP extends the traditional LBP method by combining multiple statistical measures, such as mean, variance, and median, to capture more detailed texture patterns. This approach is particularly useful for distinguishing tumors with subtle differences in texture, as it enhances the representation of complex tumor features. Key advantages of MVM-LBP include:

- Richer Texture Representation: Captures finer texture details, which are crucial for distinguishing subtle tumor patterns.
- Robustness to Noise: The method is more resilient to noise and variations in intensity compared to basic LBP.
- Effective for Heterogeneous Tumors: Useful in cases where the tumor exhibits heterogeneous textures, allowing for better classification of tumors with varied regions.

- Implementation: Compute mean, variance, and median values in a local neighborhood. Then, generate LBP codes for each of these statistics and combine them to form a composite feature, which enhances discriminatory power.

Model Architecture

In this project, we utilized two powerful pre-trained models, **VGG16** and **ResNet18**, which have been adapted to perform binary classification (benign vs. malignant) for breast cancer detection in MRI images. Both models leverage transfer learning to enhance performance on a relatively smaller dataset by utilizing pre-trained weights from ImageNet.

1. Custom VGG16 Model

Overview: VGG16 is a widely used convolutional neural network architecture that is 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. For this project, the VGG16 model is fine-tuned for the task of classifying MRI images of breast cancer into benign and malignant categories.

Key Features and Modifications:

- **Pre-trained Base**: The VGG16 model is initialized with pre-trained weights from ImageNet to benefit from transfer learning. This helps the model learn from a large variety of images and improves performance on smaller datasets.
- **Custom Classifier**: The original VGG16 classifier is replaced with a custom fully connected layer tailored for binary classification. The modified classifier includes: o Two hidden layers, each of size 4096 neurons

  with ReLU activation functions. o Dropout layers for regularization to avoid overfitting. o A final output layer with two nodes representing the two classes (benign vs. malignant).

Advantages:

- VGG16 is easy to modify and adapt for different tasks.

- Its deep network architecture captures intricate hierarchical patterns in the image, making it suitable for complex image classification tasks.
- The simplicity of VGG16 ensures ease of debugging and training.

Why Use This Model:

- VGG16's architecture, though deep, is straightforward, making it easy to implement and adapt.
- It works well as a baseline model and can generalize effectively on moderately complex datasets like medical images.

```
import torch

import torch.nn as nn

from torchvision import models
 class CustomVGG16(nn.Module):


    def __init__(self, num_classes=2):


      # Initialize the parent class

      super(CustomVGG16, self).__init__()


      # Load the pre-trained VGG16 model

      vgg16 = models.vgg16(pretrained=True)


      # Extract the features and avgpool layers

      self.features = vgg16.features

      self.avgpool = vgg16.avgpool


```

```
        # Define a new classifier using nn.Sequential


        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),  # First linear layer
            nn.ReLU(),                  # ReLU activation
            nn.Dropout(),               # Dropout layer
            nn.Linear(4096, 4096),        # Second linear layer
            nn.ReLU(),                  # ReLU activation

            nn.Dropout(),               # Dropout layer
            nn.Linear(4096, num_classes)  # Final linear layer for binary classification


        )
    def forward(self, x):


        # Pass the input through the features layer

        x = self.features(x)


        # Use the avgpool layer and reshape the output to a 2D tensor

        x = self.avgpool(x)
        x = torch.flatten(x, 1)  # Flatten the tensor
(batch_size, num_features)


        # Pass the reshaped output to the custom classifier

        x = self.classifier(x)
```

```
        return x

# Example of creating an instance of CustomVGG16
model = CustomVGG16(num_classes=2)
```

2. Custom ResNet18 Model

Overview: ResNet18 is a variant of the ResNet family that introduces residual connections, which help solve the vanishing gradient problem and allow for training deeper networks without performance degradation. This makes ResNet18 suitable for tasks that require deep learning models but where computational efficiency is also important.

Key Features and Modifications:

- **Pre-trained Base**: The ResNet18 model is initialized with pre-trained weights from ImageNet. The pretrained model is used for feature extraction, and the fully connected layer is replaced with a custom classifier.
- **Custom Fully Connected Layer**: The final fully connected layer is modified to output two classes (benign vs. malignant), instead of the 1000 classes in ImageNet. This is done by changing the output layer's dimensions to match the number of classes for binary classification.

Advantages:

- The use of residual connections improves gradient flow and helps in the training of deeper networks, preventing overfitting while maintaining high performance.
- ResNet18 is computationally efficient, making it suitable for training on systems with limited hardware resources (such as those without GPUs).

Why Use This Model:

- ResNet18 offers a good balance between performance and computational cost.

- The residual connections prevent overfitting and improve model performance, especially on small datasets like medical images.

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

# ResNet18 class that inherits from nn.Module
class Resnet18(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet18, self).__init__()

        # Load the pretrained ResNet18 model
        model_resnet18 = models.resnet18(pretrained=True)

        # Extract layers from pretrained model
        self.conv1 = model_resnet18.conv1       # initial convolutional layer
        self.bn1 = model_resnet18.bn1           # batch normalization layer
        self.relu = model_resnet18.relu         # ReLU activation function
        self.maxpool = model_resnet18.maxpool   # max pooling layer

        # ResNet blocks for feature extraction
        self.layer1 = model_resnet18.layer1
        self.layer2 = model_resnet18.layer2
        self.layer3 = model_resnet18.layer3
        self.layer4 = model_resnet18.layer4     # deeper layers for increasing depth of the network

        # Average pooling layer
        self.avgpool = model_resnet18.avgpool

        # Replace the fully connected layer for custom number of classes
        self._features = model_resnet18.fc.in_features
        self.fc = nn.Linear(self._features, num_classes)
     def forward(self, x):
```

```
        x = self.conv1(x)                # apply convolutional layer
        x = self.bn1(x)                  # apply batch normalization
        x = self.relu(x)                 # apply ReLU activation
        x = self.maxpool(x)              # apply max pooling
        x = self.layer1(x)               # pass through ResNet layer 1
        x = self.layer2(x)               # pass through ResNet layer 2
        x = self.layer3(x)               # pass through ResNet layer 3
        x = self.layer4(x)               # pass through ResNet layer 4
        x = self.avgpool(x)              # apply average pooling
        x = x.view(x.size(0), -1)        # flatten for the fully connected layer
        x = self.fc(x)                   # apply fully connected layer for output
        return x

# Print the model architecture
if __name__ == "__main__":
    model1 = Resnet18(num_classes=2)
```

3. Custom ResNet50 Model

Overview: ResNet50 is a deeper version of ResNet18, utilizing a 50-layer architecture that allows the model to learn more complex patterns and hierarchical features. It extends the capacity of the network, making it suitable for tasks involving more intricate datasets. With residual connections, ResNet50 avoids the vanishing gradient problem and efficiently handles deeper networks.

Key Features and Modifications:

- **Pre-trained Base**: Similar to ResNet18, ResNet50 starts with pre-trained weights from ImageNet, enabling the model to learn from a diverse set of images and transfer those learned features to the task of breast cancer classification.

- **Custom Fully Connected Layer**: The final fully connected layer is replaced with a custom layer suitable for binary classification (benign vs. malignant), ensuring that the model outputs the correct number of classes.

Advantages:

- **Depth**: The 50 layers in ResNet50 provide the ability to capture more abstract and complex features in the input images, which is useful for distinguishing between subtle differences in image data.
- **Residual Connections**: These connections improve the gradient flow during training, mitigating the issues that typically arise when training very deep networks (e.g., vanishing gradients).

Why Use This Model:

- **Suitable for Complex Patterns**: ResNet50 is ideal for datasets with more complex and subtle patterns, such as medical images, where deeper feature extraction is required.
- **Computational Power**: While ResNet50 requires higher computational resources, it provides superior performance when sufficient computational power is available.

```
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__()

# Load the pretrained ResNet50 model
model_resnet50 = models.resnet50(pretrained=True)

# Extract layers from the pretrained model
self.conv1 = model_resnet50.conv1       # initial convolutional layer
self.bn1 = model_resnet50.bn1           # batch normalization layer
self.relu = model_resnet50.relu         # ReLU activation function
self.maxpool = model_resnet50.maxpool   # max pooling layer
```

```
        # ResNet blocks for feature extraction
        self.layer1 = model_resnet50.layer1
        self.layer2 = model_resnet50.layer2
        self.layer3 = model_resnet50.layer3
        self.layer4 = model_resnet50.layer4      # deeper layers for increasing depth
    of the network

        # Average pooling layer
        self.avgpool = model_resnet50.avgpool

        # Replace the fully connected layer for custom number of classes

        self._features = model_resnet50.fc.in_features
        self.fc = nn.Linear(self._features, num_classes)
    def forward(self, x):

        x = self.conv1(x)                 # apply convolutional layer

        x = self.bn1(x)                  # apply batch normalization

        x = self.relu(x)                  # apply ReLU activation

        x = self.maxpool(x)                # apply max pooling

        x = self.layer1(x)                # pass through
    ResNet layer 1
        x = self.layer2(x)                # pass through
    ResNet layer 2
        x = self.layer3(x)                # pass through
    ResNet layer 3
        x = self.layer4(x)                # pass through
    ResNet layer 4
        x = self.avgpool(x)                # apply average pooling

        x = x.view(x.size(0), -1)          # flatten for the fully connected layer

        x = self.fc(x)                   # apply fully connected layer for output

        return x

# Print the model architecture
if __name__ == "__main__":
```

```
model2 = resnet50(num_classes=2)
```

Comparative Analysis and Decision

- **VGG16 Custom Model**: While VGG16 is known for its simplicity and reliability, it may not perform as well as ResNet architectures on more complex datasets. Its shallow depth limits its ability to capture intricate patterns compared to deeper models like ResNet.
- **ResNet18**: Offers a balanced trade-off between performance and computational efficiency. It is particularly well-suited for systems with limited hardware (e.g., without GPUs), providing good results with lower computational cost.
- **ResNet50**: Outperforms VGG16 and ResNet18 in terms of feature extraction due to its deeper architecture. It is ideal for datasets with complex patterns but demands higher computational resources. If computational power is available, ResNet50 provides the best results for the task of breast cancer classification.

Training Loop

The training loop in PyTorch is a crucial part of the model training process. It iterates through the training dataset, calculates the loss, performs backpropagation, and updates the model's weights. Below are the steps involved in the training loop:

---

1. Data Loading

The training data is provided through a PyTorch DataLoader instance, referred to as train_loader. Each batch of data includes:

- **Image Data**: The input images to be processed by the model.
- **Labels**: The true class labels for the images, indicating whether the tumor is benign or malignant.

These batches are passed to the model for training, with each pass corresponding to one iteration in the training loop.

---

2. Loss Function

The **Cross-Entropy Loss** function is used to calculate the difference between the predicted class probabilities and the true labels. It is suitable for classification tasks, including binary classification (benign vs. malignant) in this case. The formula for Cross-Entropy Loss is:

L(y,y^)=−∑ylog⁡(y^)L(y, \hat{y}) = -\sum y \log(\hat{y})L(y,y^)=−∑ylog(y^) Where:

- yyy represents the true label (one-hot encoded).
- y^\hat{y}y^ represents the predicted probabilities from the model.

The loss guides the optimization process, helping the model adjust its weights to reduce errors in predictions.

---

3. Optimizer

The **Stochastic Gradient Descent (SGD)** optimizer is used to update the model's weights. It adjusts the weights based on the gradients computed during backpropagation. Several components are configured for the optimizer:

- **Learning Rate**: Initially set to 0.01, the learning rate is reduced dynamically every 10 epochs by a factor of 0.1. This gradual reduction ensures smooth convergence and prevents the model from overshooting the optimal solution during the later stages of training. The learning rate is capped at a minimum value of 1×10−51 \times 10^{-5}1×10−5 to prevent it from becoming too small and halting learning.
- **Momentum**: Momentum is set to 0.9, which accelerates the gradients in the correct direction and helps avoid oscillations, particularly in regions where the gradient may be noisy.
- **L2 Regularization (Weight Decay)**: To avoid overfitting, an L2 penalty is applied using a weight decay factor of 0.01. This regularization term discourages large weight values, promoting simpler models that generalize better to unseen data.

---

4. Training Process

The training process consists of the following steps:

- **Epoch Setup**: For each epoch, the model is set to **training mode** (model.train()), which enables certain features like dropout (if applicable). This ensures the model behaves appropriately during training.
- **Batch Processing**: The model processes each batch of images and their corresponding labels: o The image data and labels are moved to the GPU for faster computation using .cuda().
    - o The model performs a **forward pass**, generating predictions for the batch.
- **Loss Calculation**: The loss function calculates the loss between the predicted labels and the true labels for the current batch.
- **Backpropagation**:
    - **Zero Gradients**: The optimizer's gradients are reset to zero (optimizer.zero_grad()) to prevent accumulation from previous iterations.
    - **Gradient Calculation**: Backpropagation is performed to compute the gradients, which tell the optimizer how to adjust the model's weights.
- **Optimizer Step**: The optimizer updates the model's weights based on the computed gradients (optimizer.step()).
- **Training Accuracy**: After each batch, the accuracy is calculated by comparing the predicted labels to the true labels. This helps monitor the model's performance throughout training.

Learning Rate Scheduling

To improve convergence, the learning rate is adjusted dynamically throughout training. The learning rate is:

- **Initial Value**: Set to 0.01 at the start.
- **Reduction Schedule**: Reduced by a factor of 0.1 every 10 epochs.

- **Minimum Cap**: Capped at 1×10−51 \times 10^{5}1×10−5 to prevent the learning rate from becoming too small and effectively stopping the training process.

This step-down approach ensures that the model converges smoothly and avoids overshooting the optimal solution in the later stages of training.

Early Stopping

Early stopping is a crucial technique used to prevent overfitting and save computational resources. It works by monitoring the model's validation loss and halting training when the loss stops improving. This mechanism ensures that the model does not continue training unnecessarily after it has reached its optimal performance.

---

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.
- If the validation loss does not improve for 5 consecutive epochs, training will be halted early.

This parameter helps avoid overfitting by preventing the model from training further once it has stopped learning useful patterns.

---

2. Saving the Best Model

During training, if the validation loss improves (i.e., it decreases), the model's weights are saved. This ensures that the **best-performing model**—with the lowest validation loss—is retained for future inference. The saved model can then be used for testing and deployment.

---

3. Stopping Condition

The training stops when the model's performance does not improve beyond a certain threshold, known as **delta**. This threshold helps to prevent unnecessary training when the model is no longer learning useful patterns, thus saving computational resources and avoiding overfitting.

Batch Size

The batch size for training is set to **32**. This value strikes a balance between:

- **Training Speed**: A larger batch size can lead to more stable gradient estimates, speeding up training.
- **Generalization**: A smaller batch size may introduce more noise into the optimization process, which
  can help the model generalize better by avoiding overfitting.

Thus, a batch size of 32 is optimal for ensuring efficient and effective model training.

Performance Monitoring

Throughout the training process, **performance monitoring** is crucial. The primary metric used to evaluate the model's progress is **training accuracy**. This helps to track how well the model is performing on the training data.

- **Accuracy** is printed at the end of each epoch to provide real-time feedback on the model's performance.
- Adjustments can be made to hyperparameters such as the learning rate or optimizer based on the observed accuracy to improve model performance.

This structured process of monitoring and adjustments ensures that the model is trained efficiently while preventing overfitting and improving generalization.

Validation and Testing Evaluation

Validation Phase

The **validation** phase occurs after each epoch during training and helps to evaluate the model's generalization ability on data it has not seen during training. The key steps in the validation phase include:

1. **Model Evaluation Mode**:

- The model is switched to **evaluation mode** (model.eval()), disabling layers like dropout or batch normalization that behave differently during training.

2. **Loss Calculation**:

   - The **Negative Log Likelihood Loss (nll_loss)** is used to calculate the loss on the validation set. This loss function is appropriate for multi-class classification and measures how well the predicted probabilities align with the true 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 index (class) with the highest probability.

4. **Softmax Probabilities for AUC**:

   - The **softmax function** is applied to the model's outputs to convert logits into probabilities.

     These probabilities are used for computing the **ROC curve** and **AUC** (Area Under the Curve).

5. **Metrics**:

   - **Confusion Matrix**: Helps in understanding the number of true positives, false positives, true negatives, and false negatives.
   - **AUC**: The ROC curve and AUC score evaluate the trade-off between sensitivity and specificity across various thresholds.
   - **Sensitivity and Specificity**: These metrics provide insights into the model's ability to detect both positive and negative classes:
     - **Sensitivity** (Recall) measures the model's ability to identify positive cases.
     - **Specificity** measures the model's ability to identify negative cases.

The **validation loss** and **accuracy** are computed, helping to track the model's generalization performance.

Testing Phase

After training, the model is evaluated on the **test dataset**, which the model has never seen before. This provides an unbiased evaluation of the model's generalization. The key components of the testing process are:

1. **Model Evaluation Mode**:
    - Similar to the validation phase, the model is set to **evaluation mode** (model.eval()) to disable training-specific behaviors like dropout.

2. **Loss Calculation**:
    - The **Negative Log Likelihood 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 are stored for further analysis.

4. **ROC Curve and AUC**:
    - The **ROC Curve** and **AUC** score are computed to assess how well the model distinguishes between the classes.

5. **Classification Report**:
    - The **classification report** provides a detailed breakdown of precision, recall, and F1 score for each class. This is particularly useful for evaluating models trained on imbalanced datasets.

6. **Confusion Matrix**:
    - The **confusion matrix** provides valuable insight into the model's performance, helping to identify misclassifications like false positives and false negatives.

7. **Specificity, Sensitivity, and Accuracy**:
    - **Specificity**: The true negative rate, calculated as 1–False Positive Rate (FPR)1 - \text{False Positive Rate (FPR)}1–False Positive Rate (FPR).

- **Sensitivity** (Recall): The true positive rate, indicating how well the model identifies positive instances.
- **Accuracy**: The percentage of correct predictions on the test dataset.

Summary of Evaluation Metrics

- **Accuracy**: Proportion of correct predictions.
- **Precision**: Proportion of true positives out of all predicted positives.
- **Recall (Sensitivity)**: Proportion of true positives out of all actual positives.
- **F1 Score**: Balanced measure of precision and recall.
- **Specificity**: Proportion of true negatives out of all actual negatives.
- **ROC-AUC**: Measure of the model's ability to distinguish between positive and negative classes.
- **Confusion Matrix**: Summary of the model's classification performance.

By analyzing these evaluation metrics, we gain insights into the model's performance on the test data. This helps determine whether the model is ready for deployment or if further improvements are needed.

6. Results

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

Performance Metrics

1. Training and Validation Accuracy/Loss

The following metrics are crucial for understanding the model's learning process:

- **Training Accuracy**: This metric indicates how well the model fits the training data. It is tracked during the training process to ensure that the model is learning effectively from the training set.

- **Validation Accuracy**: This measures how well the model generalizes to unseen data from the validation set. Tracking this helps us detect **overfitting** or **underfitting** during training.
- **Training Loss**: This is tracked throughout training to measure how well the model is minimizing the loss function on the training set.
- **Validation Loss**: Computed after each epoch, the validation loss assesses the model's performance on the validation set. A rising validation loss, while training loss continues to decrease, may indicate overfitting.

In the training process, both the **training accuracy** and **validation accuracy** help identify if the model is learning properly or if it needs further adjustments (e.g., reducing complexity or adjusting hyperparameters). Similarly, the **training loss** and **validation loss** help ensure that the model is not overfitting to the training data.

---

2. Confusion Matrix

The **confusion matrix** is calculated for both the validation and test datasets. It provides a detailed view of the model's classification performance across all classes by showing:

- **True Positives (TP)**: Correctly classified positive instances.
- **False Positives (FP)**: Negative instances incorrectly classified as positive.
- **True Negatives (TN)**: Correctly classified negative instances.
- **False Negatives (FN)**: Positive instances incorrectly classified as negative.

For a binary classification model, a confusion matrix may look like:

Model Evaluation Results

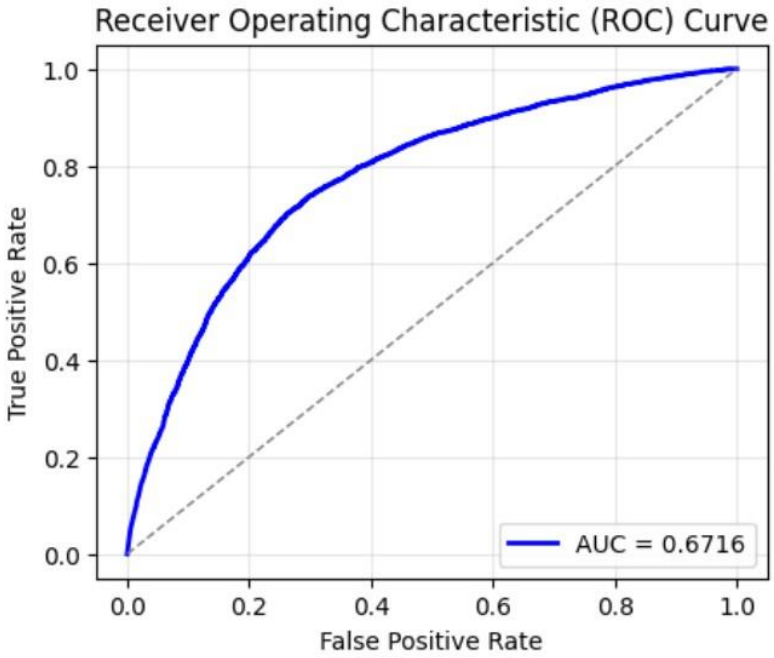
In this section, we provide the evaluation results for three different models: **VGG16**, **ResNet18**, and **ResNet50**. The evaluation includes training and testing accuracy, loss, AUC, and other performance metrics such as precision, recall, F1-score, confusion matrix, specificity, and sensitivity. These metrics help assess the model's performance and ability to generalize to unseen data.

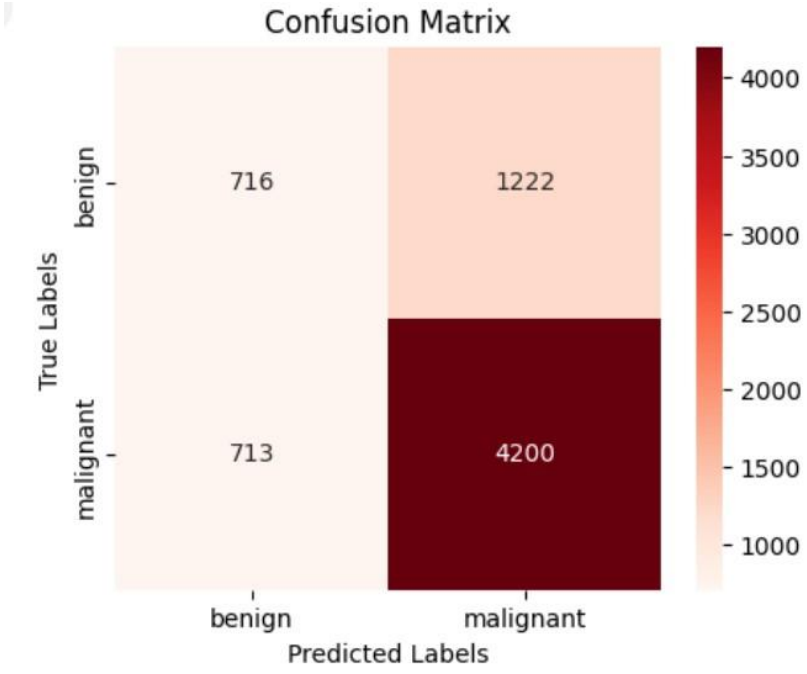
Model 1: VGG16

```
EarlyStopping counter: 19 out of 20
Epoch 22/50: 100%|██████████| 639/639 [03:17<00:00,  3.24batch/s]
train accuracy: 99.97063446044922%
Specificity: 1.0000, Sensitivity: 0.1131, AUC: 0.9181

Test set: Average loss: 0.6001, Accuracy: 82.20%

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

```
test(model, test_loader)
```

```
              precision    recall  f1-score   support

      benign     0.5010    0.3695    0.4253      1938
   malignant     0.7746    0.8549    0.8128      4913

    accuracy                         0.7176      6851
   macro avg     0.6378    0.6122    0.6190      6851
weighted avg     0.6972    0.7176    0.7032      6851

Confusion Matrix:
 [[ 716 1222]
 [ 713 4200]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.6716

Test set: Average loss: 1.7644, Accuracy: 4916/6851 (71.76%)
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
AUC = 0.6716
0.0
0.2
0.4
0.6
0.8
1.0

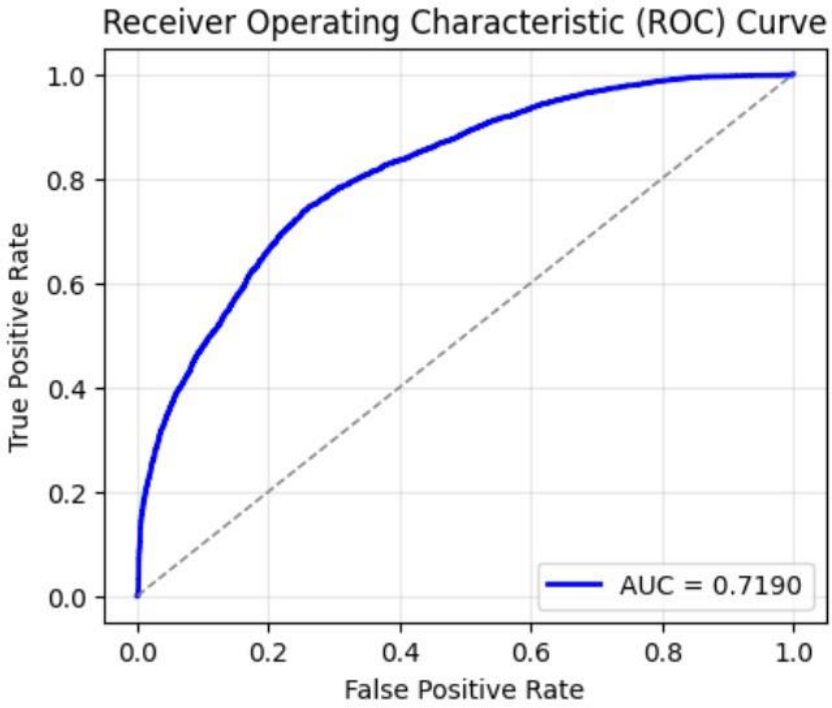

Confusion Matrix
True Labels
Predicted Labels
benign
malignant
716
1222
713
4200
4000
3500
3000
2500
2000
1500
1000

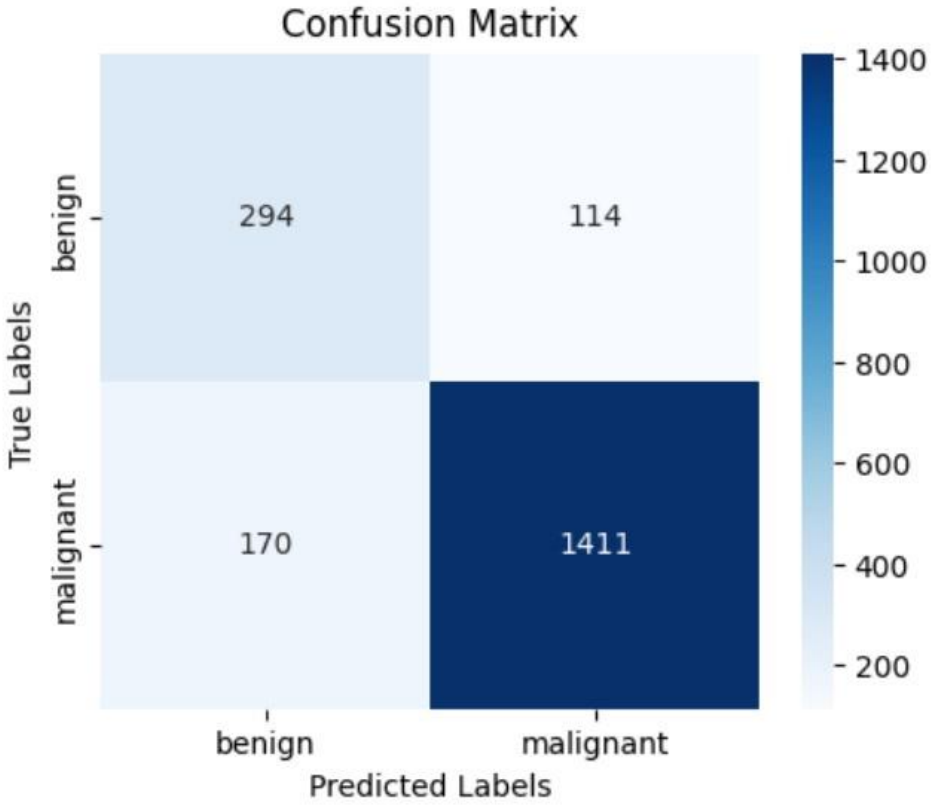
Model 2: ResNet18

```
EarlyStopping counter: 19 out of 20
Epoch 23/50: 100%|██████████| 639/639 [01:18<00:00,  8.16batch/s]
train accuracy: 99.99510955810547%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.9183

Test set: Average loss: 0.4066, Accuracy: 85.72%

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

	precision	recall	f1-score	support
benign	0.6200	0.2147	0.3189	1938
malignant	0.7537	0.9481	0.8398	4913
accuracy			0.7406	6851
macro avg	0.6868	0.5814	0.5794	6851
weighted avg	0.7159	0.7406	0.6925	6851

```
Confusion Matrix:
 [[ 416 1522]
 [ 255 4658]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7190

Test set: Average loss: 0.8269, Accuracy: 5074/6851 (74.06%)
```



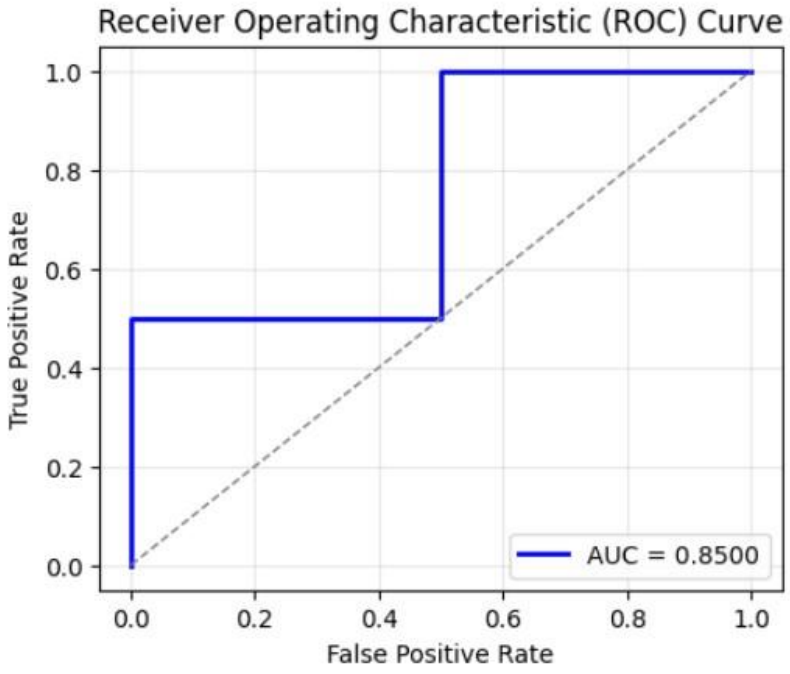


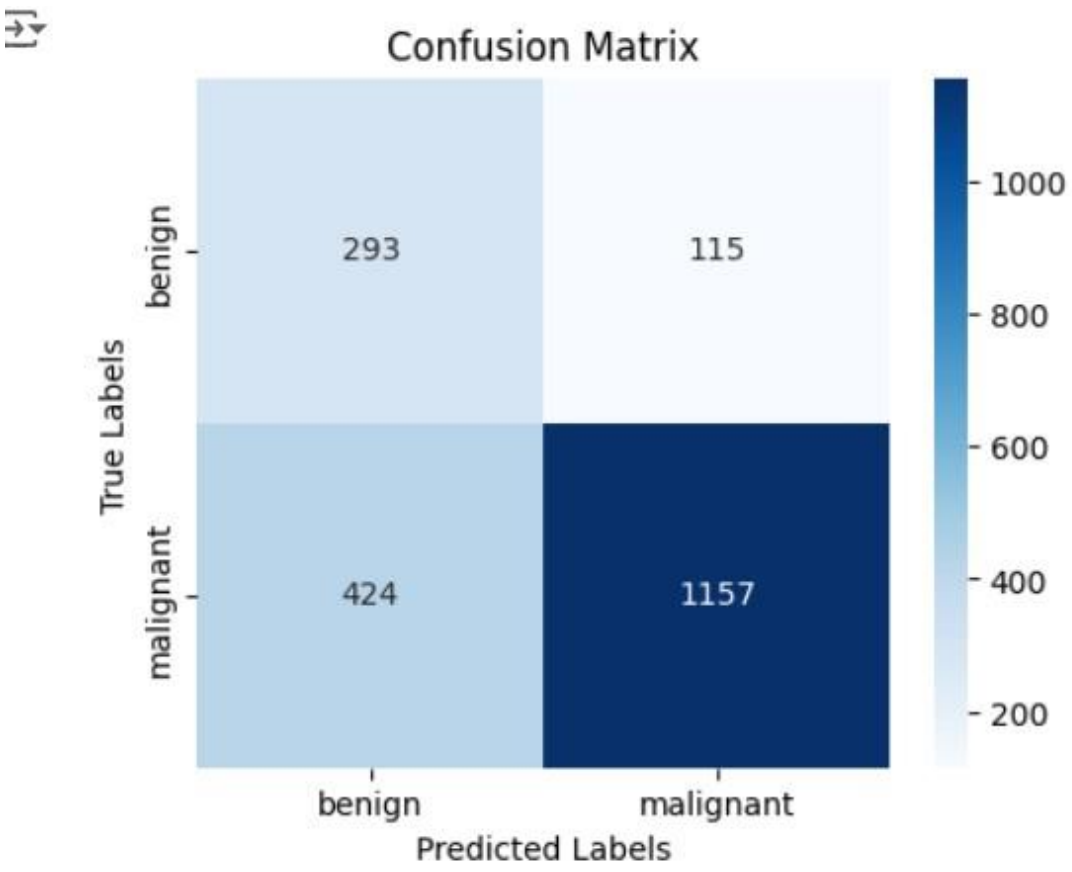
Model 3: ResNet50

```
EarlyStopping counter: 19 out of 20
Epoch 25/50: 100%|██████████| 639/639 [02:29<00:00,  4.28batch/s]
train accuracy: 99.96574401855469%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8217

Test set: Average loss: 0.7503, Accuracy: 72.90%

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

	precision	recall	f1-score	support
benign	0.5520	0.3617	0.4370	1938
malignant	0.7784	0.8842	0.8279	4913
accuracy			0.7364	6851
macro avg	0.6652	0.6229	0.6325	6851
weighted avg	0.7143	0.7364	0.7173	6851

```
Confusion Matrix:
 [[ 701 1237]
 [ 569 4344]]
Specificity: 1.0000, Sensitivity: 0.0000, AUC: 0.7018

Test set: Average loss: 0.8197, Accuracy: 5045/6851 (73.64%)
```





Gradio Interface for Breast Cancer Classification

To provide an interactive and user-friendly interface for the breast cancer classification model, we use Gradio. This allows users to upload images, and the model will predict whether the image corresponds to a benign or malignant tumor.

* Running on local URL: http://127.0.0.1:7860

Work done :

Day 1
Date: 14/10/2024
Agenda:

1. Correcting mistakes in augmentation
2. Understanding CNN architecture
   Points Discussed:

- Correcting augmentation code
  Action Items:

- Complete augmentation and set up dataloader
  Next Meeting:

- Complete data augmentation techniques
  Follow-up Task:

- Understand the architecture of deep learning models

Day 2
Date: 15/10/2024
Agenda:

1. Correcting mistakes in EDA
2. Understanding CNN architecture
   Points Discussed:

- Implementing different augmentation techniques

- Implementation of data loading using TensorFlow/PyTorch
  Action Items:

- Properly set up the data loaders in TensorFlow/PyTorch
  Next Meeting:

- Complete data augmentation techniques
  Follow-up Task:

- Start with various deep learning models

Day 3
Date: 16/10/2024
Agenda:

1. Understanding Feature Extraction from images
2. Understanding pixel values and what they represent
   Points Discussed:

- Feature extraction in images
  Action Items:

- Try out different feature extraction techniques for images and plot the images
  Next Meeting:

- Complete feature extraction techniques
  Follow-up Task:

- Start with various deep learning models

Day 4
Date: 17/10/2024
Agenda:

1. Understanding Feature Extraction from images
2. Implement HOG feature extraction
   Points Discussed:

- Feature extraction in images
  Action Items:

- Try out HOG feature extraction and plot the images
  Next Meeting:

- Try out different feature extraction techniques for images and plot the images
  Follow-up Task:

- Start with various deep learning models

Day 5
Date: 21/10/2024
Agenda:

1. Understanding Feature Extraction from images
2. Implement Edge detection
   Points Discussed:

- Scratch implementation of Sobel operator
  Action Items:
- Understand and implement convolution function in Python and implement Sobel operator
  Next Meeting:
- Discuss Local Binary points
  Follow-up Task:
- Implement Local Binary points

Day 6
Date: 22/10/2024
Agenda:

1. Understanding Local Binary points
   Points Discussed:

- Implement Local Binary points
  Action Items:
- Understand and implement Local Binary points in Python
  Next Meeting:
- Complete the implementation of the LBP
  Follow-up Task:
- Complete scratch implementation of LBP

Day 7
Date: 23/10/2024
Agenda:

1. Read about various ways to calculate binary points
   Points Discussed:

- Different types of binary points in research paper
  Action Items:
- Read research paper about different types of binary points
  Next Meeting:
- Start the implementation of Mean binary points, Median Binary points, and Variance Binary points
  Follow-up Task:
- Finish implementation of various binary points

Day 8
Date: 25/10/2024
Agenda:

1. Checking outputs of various binary points
   Points Discussed:

- Implement Mean, Median, and Variance Binary points
  Action Items:
- Understand and implement Local Binary points in Python
  Next Meeting:
- Analyze outputs of various binary points
  Follow-up Task:
- Complete implementation of MVM binary point

Day 9
Date: 28/10/2024
Agenda:

1. Checking outputs of various binary points
   Points Discussed:

- Implement Mean, Median, and Variance Binary points
  Action Items:

- Understand and implement Local Binary points in Python
  Next Meeting:
- Analyze outputs of various binary points
  Follow-up Task:
- Complete implementation of MVM binary point

Day 10
Date: 29/10/2024
Agenda:

1. Checking outputs of various binary points
   Points Discussed:
- Implement Mean, Median, and Variance Binary points
  Action Items:
- Understand and implement Local Binary points in Python
  Next Meeting:
- Analyze outputs of various binary points
  Follow-up Task:
- Complete implementation of MVM binary point and start GLCM analysis

---

Day 11
Date: 30/10/2024
Agenda:

1. Start with GLCM
   Points Discussed:
- Discussed GLCM concepts
  Action Items:
- Develop GLCM matrix from sample input matrix
  Next Meeting:
- Try out different types of spatial relationships
  Follow-up Task:
- Check the outputs of GLCM analysis

Day 12
Date: 01/11/2024
Agenda:

1. Check GLCM outputs
   Points Discussed:

- GLCM output and code implementation
  Action Items:

- Develop GLCM image from input image
  Next Meeting:

- Load VGG16 model
  Follow-up Task:

- Start working with the VGG16 model

Day 13
Date: 04/11/2024
Agenda:

1. Check GLCM outputs
   Points Discussed:

- GLCM output and code implementation
  Action Items:

- Develop GLCM image from input image
  Next Meeting:

- Load VGG16 model
  Follow-up Task:

- Start working with the VGG16 model

Day 14
Date: 05/11/2024
Agenda:

1. Load VGG16 model
   Points Discussed:

- Complete custom VGG16 model implementation
  Action Items:
- Start with early stopping
  Next Meeting:
- Complete Early Stopping
  Follow-up Task:
- Implement VGG16 model

Day 15
Date: 06/11/2024
Agenda:

1. Complete Early Stopping
   Points Discussed:
- Early Stopping functions
  Action Items:
- Complete early stopping and start with training
  Next Meeting:
- Start with training
  Follow-up Task:
- Understand training metrics

Day 16
Date: 07/11/2024
Agenda:

1. Started with training function
   Points Discussed:
- Optimizers to be implemented
  Action Items:
- Complete for loop in training function
  Next Meeting:
- Complete training function
  Follow-up Task:

- Understand training metrics

Day 17
Date: 08/11/2024
Agenda:

1. Started with training function
   Points Discussed:
- Optimizers to be implemented
  Action Items:
- Complete for loop in training function
  Next Meeting:
- Complete training function
  Follow-up Task:
- Start with validation function

Day 18
Date: 11/11/2024
Agenda:

1. Complete validation function and start with training
   Points Discussed:
- Implementation of validation function and training
  Action Items:
- Run the VGG16 model
  Next Meeting:
- Complete training
  Follow-up Task:
- Start with testing function

Day 19
Date: 12/11/2024
Agenda:

1. Complete validation function and start with training
   Points Discussed:

- Implementation of validation function and training
  Action Items:
- Run the VGG16 model
  Next Meeting:
- Complete training
  Follow-up Task:
- Start with testing function

Day 20
Date: 13/11/2024
Agenda:

1. Fixing training issues
   Points Discussed:

- How to upload dataset and notebook to Kaggle
  Action Items:
- Run Kaggle notebook
  Next Meeting:
- Check Kaggle outputs
  Follow-up Task:
- Start with testing function

Day 21
Date: 14/11/2024
Agenda:

1. Fixing training issues
   Points Discussed:

- Checking model training errors
  Action Items:
- Implement resnet18 and resnet50
  Next Meeting:

- Check Kaggle outputs
  Follow-up Task:
- Plot the AUC and confusion matrix

Day 22

Date: 15/11/2024
Agenda: Fixing training issues
Points Discussed:

- Checking model training errors
  Action Items:
- Implement ResNet18 and ResNet50
  Next Meeting:
- Check Kaggle outputs
  Follow-up Task:
- Plot the AUC and confusion matrix

Day 23

Date: 19/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
  Action Items:
- Correct errors
  Next Meeting:
- Check Kaggle outputs
  Follow-up Task:
- Plot the AUC and confusion matrix

Day 24

Date: 20/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
  Action Items:
- Correct errors
  Next Meeting:
- Check Kaggle outputs
  Follow-up Task:
- Plot the AUC and confusion matrix

Day 25

Date: 21/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
  Action Items:
- Correct errors
  Next Meeting:
- Implement a live demo of the model
  Follow-up Task:
- Complete documentation

Day 26

Date: 22/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
- Live demo using Gradio
  Action Items:
- Correct errors
  Next Meeting:

- Complete implementation of a live demo of the model
  Follow-up Task:
- Complete documentation and start with the PPT

Day 27

Date: 25/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
- Live demo using Gradio
  Action Items:
- Correct errors
  Next Meeting:
- Complete implementation of a live demo of the model
  Follow-up Task:
- Complete documentation and start with the PPT

Day 28

Date: 26/11/2024
Agenda: Check model outputs
Points Discussed:

- Model outputs and performance metrics
- Live demo using Gradio
  Action Items:
- Correct errors
  Next Meeting:
- Complete implementation of a live demo of the model
  Follow-up Task:
- Complete documentation and PPT

Day 29

Date: 27/11/2024
Agenda: Check GitHub repository, documentation, and PPT
Points Discussed:

- Check repository, documentation, and repository status
  Action Items:
- Correct any mistakes
  Next Meeting:
- Complete demo recording and PPT
  Follow-up Task:
- Complete documentation, PPT, and repository

Day 30

Date: 28/11/2024
Agenda: Check GitHub repository, documentation, and PPT
Points Discussed:

- Check repository, documentation, and repository status
  Action Items:
- Correct any mistakes
  Next Meeting:
- Complete demo recording and PPT
  Follow-up Task:
- Complete documentation, PPT, and repository
