Project Overview

Tumour Trace

The 'Tumour Trace' project leverages advanced deep learning techniques to develop a robust pipeline for detecting and classifying tumour regions from medical images. Using PyTorch and torch vision libraries, the project incorporates data preprocessing, augmentation, and state-of-the-art models to ensure high performance. This report details the objectives, methodology, and results obtained during the analysis.

Project Details

Introduction

Detecting and classifying tumors in medical images is a critical task in healthcare, enabling early diagnosis and treatment planning. The 'Tumour Trace' project aims to address this challenge by employing deep learning techniques to process and analyze medical image datasets. This report provides insights into the project's objectives, approach, and outcomes.

Goals

The primary objectives of the 'Tumour Trace' project are:

- To create a reliable and efficient pipeline for tumor classification.
- To preprocess and augment datasets to enhance model generalization.
- To evaluate the performance of models using metrics such as accuracy, ROC-AUC, and confusion matrices.

Methodology

The project involves the following steps:

- **Data preprocessing: Resizing images to 224x224, applying normalization, and augmenting training data with random flips, rotations, and color adjustments.**
- **Data loading: Organizing datasets for training, validation, and testing using the PyTorch DataLoader module.**
- **Model training: Employing pre-trained models and fine-tuning them for tumor classification.**
- **Evaluation: Utilizing confusion matrices and ROC-AUC curves to measure model performance.**

Code

```
import os
import torch
import torchvision.models as models
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
from tqdm import tqdm
from sklearn.metrics import confusion_matrix, roc_curve, auc
```

This block initializes the necessary libraries and tools for the project, including PyTorch, torchvision, and supporting utilities for data handling, model creation, and performance evaluation.

```
# Define transformations for training data with augmentation
train_transform = transforms.Compose([
 transforms.Resize((224, 224)), # Resize to 224x224 pixels
 transforms.RandomHorizontalFlip(p=0.5), # Randomly flip images horizontally
 transforms.RandomRotation(degrees=15), # Randomly rotate within ±15 degrees
 transforms.ColorJitter(brightness=0.2, # Adjust brightness, contrast, saturation, and hue
 contrast=0.2,
 saturation=0.2,
 hue=0.1),
 transforms.ToTensor(), # Convert image to PyTorch tensor
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize to [-1, 1]
])

# Define transformations for validation and test data (no augmentation)
test_val_transform = transforms.Compose([
 transforms.Resize((224, 224)), # Resize to 224x224 pixels
 transforms.ToTensor(), # Convert image to PyTorch tensor
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize to [-1, 1]
])
```

This block defines transformations for data preprocessing and augmentation. The training transformations include resizing, random horizontal flipping, rotation, and color adjustments to increase data diversity and robustness.

```
# Load datasets with respective transformations
train_dataset = datasets.ImageFolder(
 root='/kaggle/input/tumortrace/clasification-roi/train', # Update the path to your dataset
 transform=train_transform
)
val_dataset = datasets.ImageFolder(
 root='/kaggle/input/tumortrace/clasification-roi/val', # Update the path to your dataset
 transform=test_val_transform
)
test_dataset = datasets.ImageFolder(
 root='/kaggle/input/tumortrace/clasification-roi/test', # Update the path to your dataset
 transform=test_val_transform
)
```

This block configures datasets for training, validation, and testing. It uses ImageFolder to organize datasets and applies the specified transformations for preprocessing.

```
# Create DataLoaders for each dataset
train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)

# Display class names in the dataset
print(f"Available class names: {train_dataset.classes}")
```

This code initializes DataLoaders for the training, validation, and test datasets, each with a batch size of 32. It shuffles the training data for randomness while keeping the validation and test data in order. Additionally, it prints the class names available in the training dataset.

```
# Function to count images in each class
def count_classes(dataset):
 class_counts = {class_name: 0 for class_name in dataset.classes}

 for _, labels in DataLoader(dataset, batch_size=1):
 for label in labels:
 class_name = dataset.classes[label.item()]
 class_counts[class_name] += 1

 return class_counts

# Count images in each dataset
train_counts = count_classes(train_dataset)
val_counts = count_classes(val_dataset)
test_counts = count_classes(test_dataset)

# Print the counts
print(f"Train dataset counts: {train_counts}")
print(f"Validation dataset counts: {val_counts}")
print(f"Test dataset counts: {test_counts}")
```

This code defines a function count_classes to count the number of images in each class within a dataset. It loops through the dataset using a DataLoader and updates a dictionary class_counts based on the labels of each image. The function is then called for the training, validation, and test datasets, and the image counts for each class are printed for each dataset.

Output :

Train dataset: 5559 benign, 14875 malignant

Validation dataset: 408 benign, 1581 malignant

Test dataset: 1938 benign, 4913 malignant

```
# Function to convert tensor back to PIL image
def tensor_to_pil(tensor):
 unnormalize = transforms.Normalize(
 mean=[-0.5 / 0.5, -0.5 / 0.5, -0.5 / 0.5],
 std=[1 / 0.5, 1 / 0.5, 1 / 0.5]
 )
 tensor = unnormalize(tensor).clamp(0, 1) # Undo normalization and clamp to valid range
 return transforms.ToPILImage()(tensor) # Convert tensor to PIL image
```

This function tensor_to_pil converts a normalized tensor back into a PIL image. It first reverses the normalization using the unnormalize transformation, then clamps the values to ensure

they fall within the valid range (0, 1). Finally, it converts the tensor to a PIL image using transforms.ToPILImage().

```
# Function to display augmented images
def show_augmented_images(dataset, class_name, num_images=5):
 if class_name not in dataset.classes:
 raise ValueError(f"Class '{class_name}' not found. Available classes: {dataset.classes}")

 plt.figure(figsize=(15, 5)) # Create figure

 # Find the class index for the requested class
 class_idx = dataset.class_to_idx[class_name]

 # Load a sample image of the requested class
 for img, label in DataLoader(dataset, batch_size=1, shuffle=True):
 if label.item() == class_idx: # Check if the image matches the desired class
 sample_img = img[0] # Extract the image tensor
 pil_img = tensor_to_pil(sample_img) # Convert tensor to PIL image
 break

 # Display multiple augmented versions of the same image
 for i in range(num_images):
 augmented_img = train_transform(pil_img) # Apply random transformations again

 plt.subplot(1, num_images, i + 1) # Create a subplot for each image
 plt.imshow(augmented_img.permute(1, 2, 0).numpy()) # Convert tensor to numpy
 plt.axis('off') # Hide axes

 plt.suptitle(f'Augmented versions of class: {class_name}', fontsize=16)
 plt.tight_layout()
 plt.show()
```

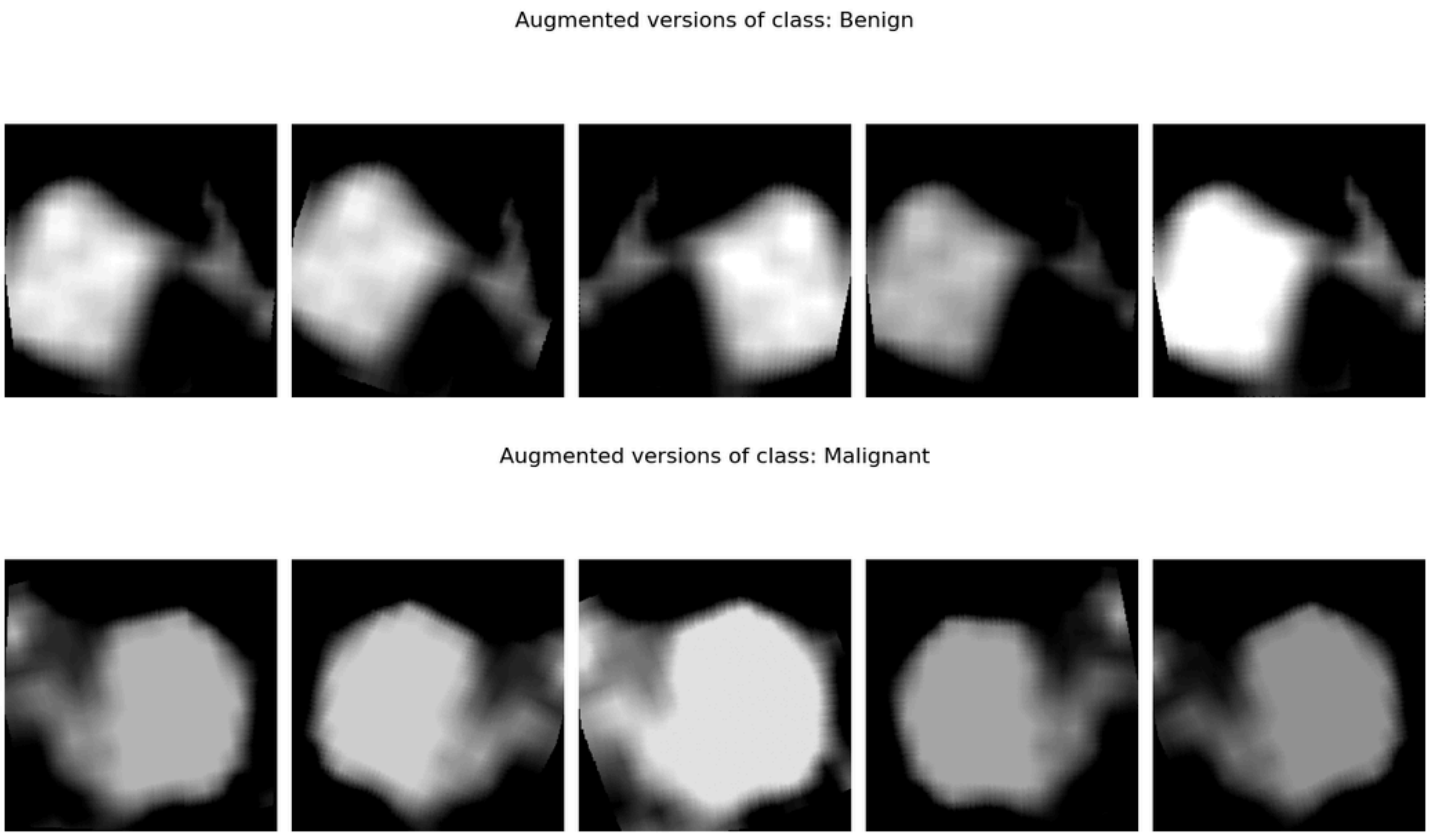
This function show_augmented_images displays multiple augmented versions of an image from a specified class in the dataset. It first checks if the class exists in the dataset. Then, it loads a sample image of the requested class and converts it to a PIL image. The function applies random transformations (using train_transform) multiple times to generate augmented versions of the image. Finally, it displays these augmented images in a row using matplotlib.

```
# Display augmented images for the first class
show_augmented_images(train_dataset, class_name=train_dataset.classes[0], num_images=5)

# If there are multiple classes, display augmented images for the second class
if len(train_dataset.classes) > 1:
 show_augmented_images(train_dataset, class_name=train_dataset.classes[1], num_images=5)
```

This code calls the show_augmented_images function to display augmented versions of images from the first and second classes of the train_dataset. It first displays 5 augmented images for the first class, and if there is more than one class in the dataset, it also displays 5

augmented images for the second class. The images are generated by applying random transformations defined in train_transform.

Output :




```
# Import libraries for image processing and visualization
import cv2
import matplotlib.pyplot as plt
from skimage.feature import hog
from skimage import exposure
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

These libraries are typically used together for image classification, feature extraction, and visualization tasks in computer vision.

```
# Load and resize an image
def load_and_resize_image(image_path, size=(224, 224)):
 image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE) # Load as grayscale
 return cv2.resize(image, size) # Resize to target dimensions
```

This function load_and_resize_image loads an image from a given file path and resizes it to a specified size. It reads the image in grayscale using OpenCV (cv2.IMREAD_GRAYSCALE) and

then resizes the image to the target dimensions, which default to (224, 224). This is useful for preparing images for further processing or model input.

```
# Plot resized image and pixel intensity histogram
def plot_image_and_histogram(image):
 pixel_values = image.flatten()

 plt.figure(figsize=(12, 6))

 # Plot the resized image
 plt.subplot(1, 2, 1)
 plt.imshow(image, cmap='gray')
 plt.title('Resized Image')
 plt.axis('off')

 # Plot the histogram
 plt.subplot(1, 2, 2)
 plt.hist(pixel_values, bins=256, range=(0, 256), color='gray', alpha=0.7)
 plt.title('Histogram of Pixel Values')
 plt.xlabel('Pixel Intensity')
 plt.ylabel('Frequency')

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

This function plot_image_and_histogram displays a grayscale resized image and its pixel intensity histogram side by side. It first shows the image, then plots the distribution of pixel values, giving insight into the image's brightness and contrast.

```
# Compute and plot HOG features
def plot_hog_features(image):
 hog_features, hog_image = hog(image, orientations=9, pixels_per_cell=(8, 8),
 cells_per_block=(2, 2), visualize=True, block_norm='L2-Hys')
 hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

 plt.figure(figsize=(12, 6))

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

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

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

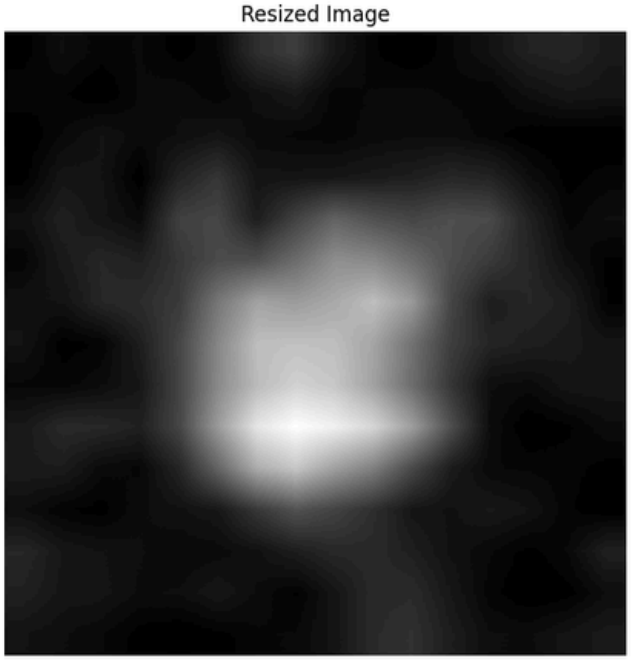
This function plot_hog_features computes and visualizes the Histogram of Oriented Gradients (HOG) features of a given image. It calculates the HOG features and the corresponding HOG image, then rescales the HOG image for better visualization. It displays the original image and the HOG feature map side by side using matplotlib.

```
image_path = "/kaggle/input/tumortrace/clasification-roi/test/Benign/BreaDM-Be-1810/SUB1/p-030.jpg" # Update with a valid path
image_resized = load_and_resize_image(image_path)

# Visualize
plot_image_and_histogram(image_resized)
plot_hog_features(image_resized)
```

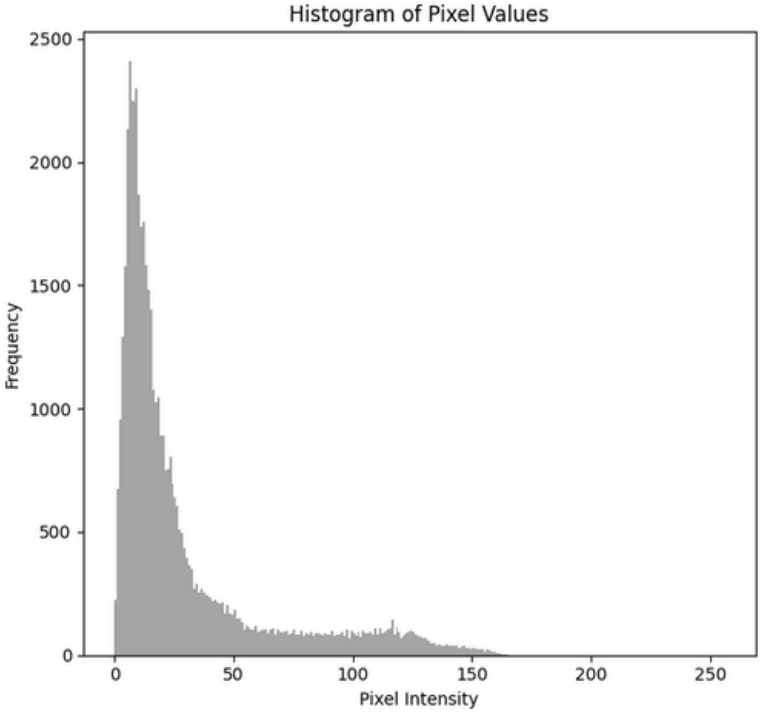
This code loads and resizes an image from the given path, then visualizes the resized image with its pixel intensity histogram and HOG features using the respective plotting functions.

Output :

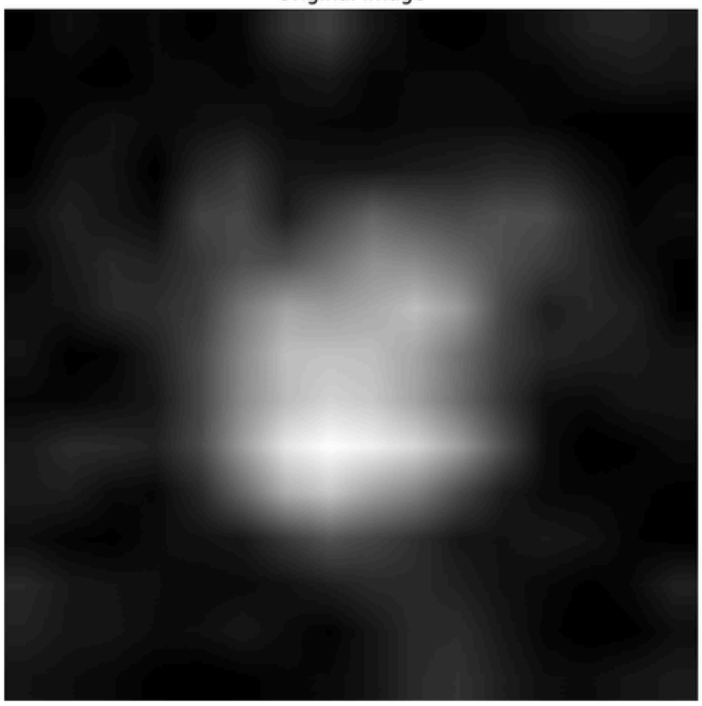
Resized Image



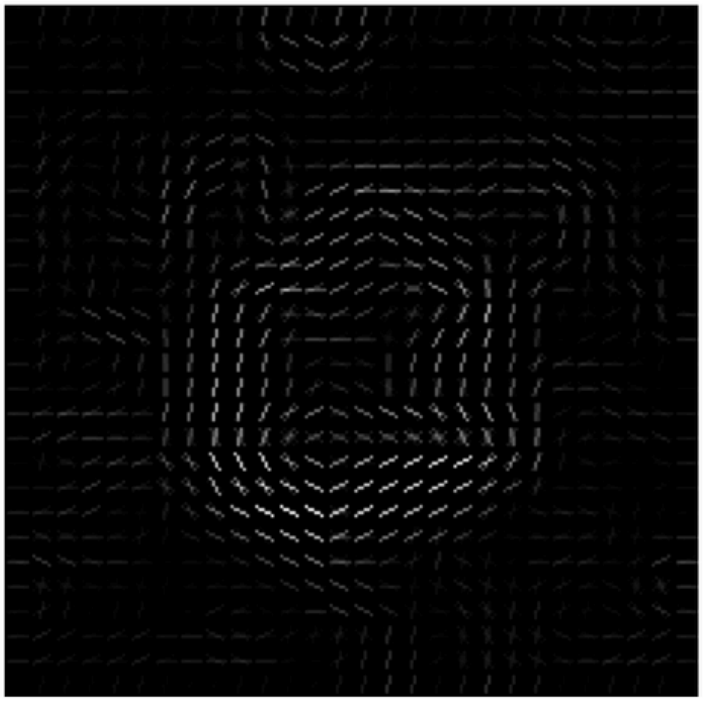
Histogram of Pixel Values



Original Image



HOG Visualization



```
def convolve(image, kernel):
 """Perform convolution on an image with a kernel."""
 image_x, image_y = image.shape
 kernel_x, kernel_y = kernel.shape

 height_radius, width_radius = np.array(kernel.shape) // 2
 output = np.zeros(image.shape)

 # Stride and padding
 stride = 1
 padding = 0
 output_x = int(((image_x - kernel_x + 2 * padding) // stride) + 1)
 output_y = int(((image_y - kernel_y + 2 * padding) // stride) + 1)

 print(f"Output Dimensions: {output_x}x{output_y}")
```

This convolve function calculates the output dimensions of an image after applying a convolution with a given kernel, considering stride and padding. It computes the height and width of the resulting image and prints these dimensions. The actual convolution operation is not yet performed in this function.

```
def compute_lbp(image):
 """Compute the Local Binary Pattern of an image."""
 height, width = image.shape
 lbp_image = np.zeros((height - 2, width - 2), dtype=np.uint8) # Initialize LBP image

 # Iterate over each pixel (excluding the border)
 for i in range(1, height - 1):
 for j in range(1, width - 1):
 center_pixel = image[i, j]
 binary_string = '' # Initialize binary string for LBP pattern

 # Check the 8 neighbors in clockwise order
 binary_string += '1' if image[i-1, j-1] >= center_pixel else '0' # Top-left
 binary_string += '1' if image[i-1, j] >= center_pixel else '0' # Top-center
 binary_string += '1' if image[i-1, j+1] >= center_pixel else '0' # Top-right
 binary_string += '1' if image[i, j+1] >= center_pixel else '0' # Middle-right
 binary_string += '1' if image[i+1, j+1] >= center_pixel else '0' # Bottom-right
 binary_string += '1' if image[i+1, j] >= center_pixel else '0' # Bottom-center
 binary_string += '1' if image[i+1, j-1] >= center_pixel else '0' # Bottom-left
 binary_string += '1' if image[i, j-1] >= center_pixel else '0' # Middle-left

 # Convert binary string to decimal and assign to the LBP image
 lbp_value = int(binary_string, 2)
 lbp_image[i-1, j-1] = lbp_value

 return lbp_image # Return the computed LBP image
```

This display_lbp function loads an image from the specified path, computes its Local Binary Pattern (LBP), and then displays both the original and LBP images side by side. It uses load_and_resize_image to load the image and compute_lbp to calculate the LBP. The images are shown using matplotlib, with titles and no axis for a cleaner view

```
def compute_mvm_lbp(image):
 """Compute Mean-Variance-Median Local Binary Pattern."""
 height, width = image.shape
 lbp_mean = np.zeros((height - 2, width - 2), dtype=np.uint8)
 lbp_variance = np.zeros((height - 2, width - 2), dtype=np.uint8)
 lbp_median = np.zeros((height - 2, width - 2), dtype=np.uint8)

 for i in range(1, height - 1):
 for j in range(1, width - 1):
 window = image[i - 1:i + 2, j - 1:j + 2].flatten()
 neighbors = np.delete(window, 4)

 mean, variance, median = np.mean(neighbors), np.var(neighbors), np.median(neighbors)
 lbp_mean[i - 1, j - 1] = int(''.join(['1' if p >= mean else '0' for p in neighbors]), 2)
 lbp_variance[i - 1, j - 1] = int(''.join(['1' if p >= variance else '0' for p in neighbors]), 2)
 lbp_median[i - 1, j - 1] = int(''.join(['1' if p >= median else '0' for p in neighbors]), 2)

 return lbp_mean, lbp_variance, lbp_median
```

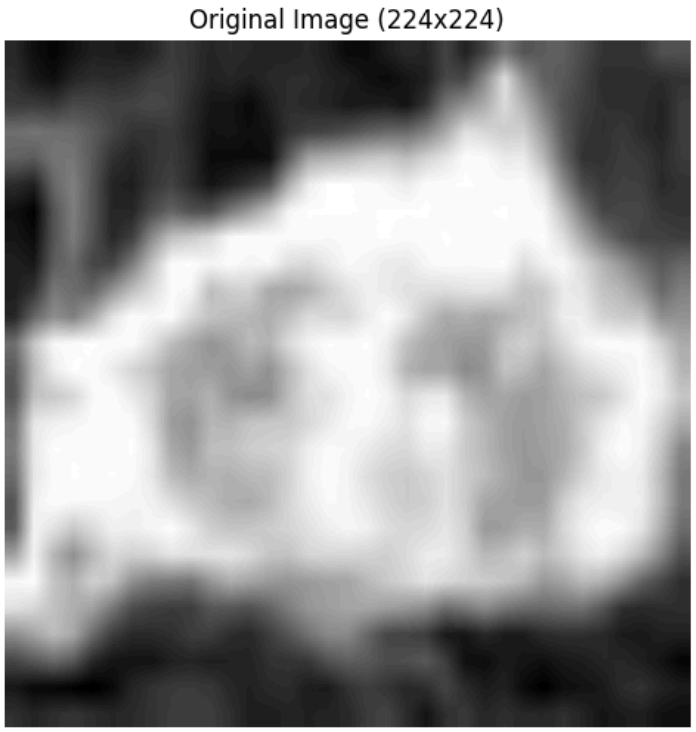
The compute_mvm_lbp function calculates three types of Local Binary Patterns (LBP) based on the mean, variance, and median of neighboring pixels in a 3x3 window. It compares each neighbor to these statistics and generates three LBP images: mean, variance, and median. These images are returned as separate outputs.

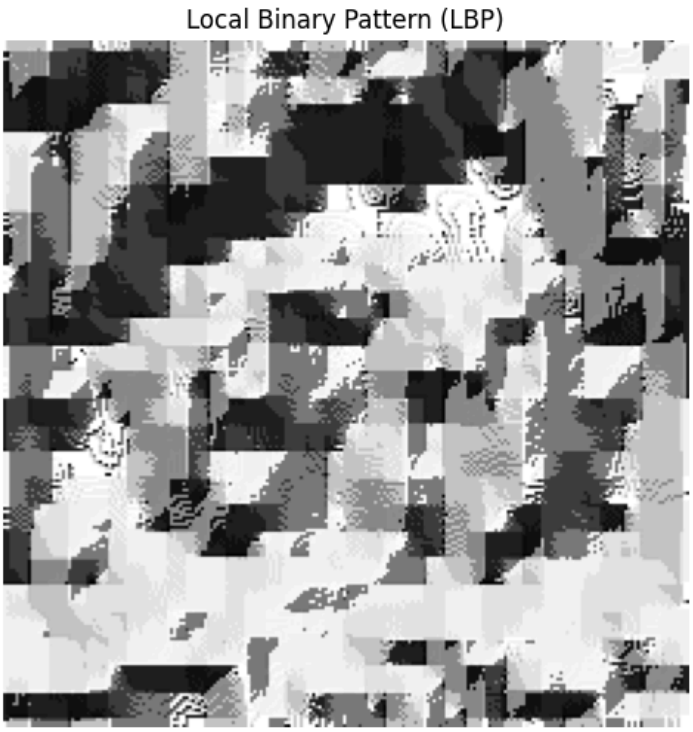
```
# Load and preprocess the image
image_path = '/kaggle/input/tumortrace/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'

image = load_and_resize_image(image_path)

# Compute and Display LBP
lbp_image = compute_lbp(image)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title('Original Image (224x224)')
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(lbp_image, cmap='gray')
plt.title('Local Binary Pattern')
plt.axis('off')
plt.show()

# Compute and Display MVM-LBP
lbp_mean, lbp_variance, lbp_median = compute_mvm_lbp(image)

plt.figure(figsize=(15, 5))
plt.subplot(1, 4, 1)
plt.title('Original Image (224x224)')
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 4, 2)
plt.title('Mean-LBP')
plt.imshow(lbp_mean, cmap='gray')
plt.axis('off')

plt.subplot(1, 4, 3)
plt.title('Variance-LBP')
plt.imshow(lbp_variance, cmap='gray')
plt.axis('off')

plt.subplot(1, 4, 4)
plt.title('Median-LBP')
plt.imshow(lbp_median, cmap='gray')
plt.axis('off')

plt.show()
```

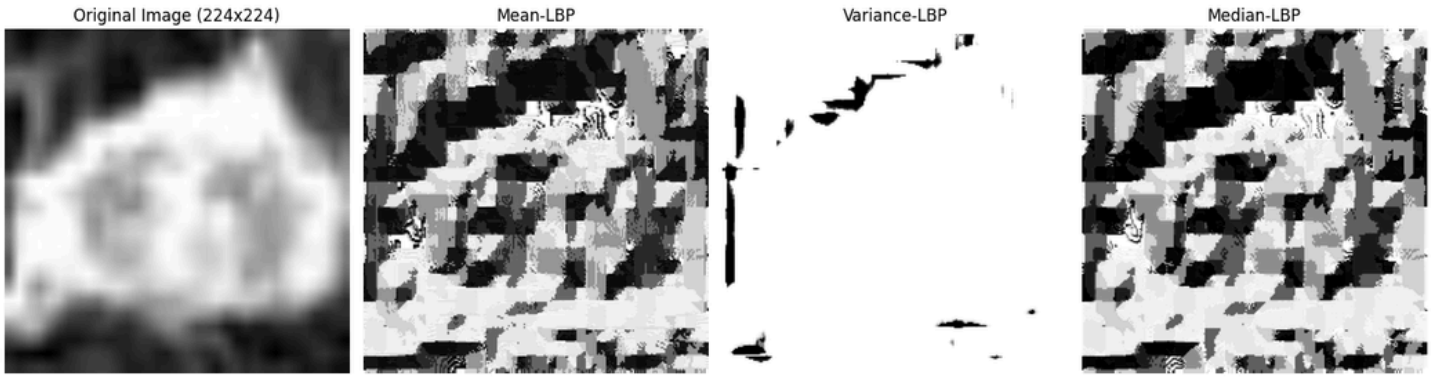
This code loads and resizes an image, then computes and displays its Local Binary Pattern (LBP). It also computes the Mean-Variance-Median Local Binary Pattern (MVM-LBP) and

visualizes the original image alongside the LBP and the three MVM-LBP variants (mean, variance, and median). Both visualizations are shown side by side using matplotlib.

Output:







```
import numpy as np
from skimage import io, color, transform
import matplotlib.pyplot as plt

def resize_and_normalize(image, size=(224, 224)):
 """
 Resizes the image to the specified size and normalizes pixel values to 0-255.
 """
 # Resize the image
 image_resized = transform.resize(image, size, anti_aliasing=True)

 # Normalize to range [0, 255] and convert to uint8
 image_normalized = (image_resized * 255).astype(np.uint8)

 return image_normalized

def compute_glcm(image, distance=1, angle=0, levels=256):
 """
 Computes a Gray-Level Co-occurrence Matrix (GLCM) for a given grayscale image.

 Parameters:
 - image: The input grayscale image as a 2D NumPy array.
 - distance: Pixel distance for GLCM calculation (1 for adjacent pixels).
 - angle: Angle in degrees (0 for horizontal, 90 for vertical, 45 for diagonal).
 - levels: Number of gray levels (typically 256 for 8-bit images).

 Returns:
 - glcm: A 2D NumPy array representing the GLCM.
 """
 glcm = np.zeros((levels, levels), dtype=np.int32)
 rows, cols = image.shape

 # Define direction based on angle
 if angle == 0: # Horizontal
 row_offset, col_offset = 0, distance
 elif angle == 90: # Vertical
 row_offset, col_offset = distance, 0
 elif angle == 45: # Diagonal (bottom-left to top-right)
 row_offset, col_offset = -distance, distance
 elif angle == 135: # Diagonal (top-left to bottom-right)
 row_offset, col_offset = distance, distance
 else:
 raise ValueError("Angle must be 0, 45, 90, or 135 degrees.")

 # Calculate GLCM
 for i in range(rows):
 for j in range(cols):
 # Determine neighboring pixel based on offsets
 row_neighbor = i + row_offset
 col_neighbor = j + col_offset
```

```
 # Check if the neighboring pixel is within image boundaries
 if 0 <= row_neighbor < rows and 0 <= col_neighbor < cols:
 current_pixel = image[i, j]
 neighbor_pixel = image[row_neighbor, col_neighbor]

 # Accumulate GLCM counts
 glcm[current_pixel, neighbor_pixel] += 1

 return glcm

# Load the image
image = io.imread('/kaggle/input/tumortrace/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg') # Replace 'path_to_image' with the actual path

# Convert to grayscale if it's an RGB image
if image.ndim == 3:
 image = color.rgb2gray(image)

# Resize and normalize the image
image_processed = resize_and_normalize(image)

# Compute the horizontal GLCM with 1 pixel offset and 0 degrees (horizontal)
glcm = compute_glcm(image_processed, distance=1, angle=0)

# Print the GLCM matrix to inspect its raw values
print("Horizontal GLCM (0 degrees):")
print(glcm)
```

This code resizes and normalizes an image, then computes its Gray-Level Co-occurrence Matrix (GLCM) for horizontal pixel pairs (0 degrees). The image is first converted to grayscale if needed. The GLCM is calculated using a 1-pixel offset and printed for inspection. This is useful for texture analysis in image processing.

Output :

```
Horizontal GLCM (0 degrees):

[[ 0  0  0 ...  0  0  0]

 [ 0  0  0 ...  0  0  0]

 [ 0  0  0 ...  0  0  0]

 ...

 [ 0  0  0 ... 646 116  0]

 [ 0  0  0 ... 104 659 10]

 [ 0  0  0 ...  0 10 42]]
```

```
# Import necessary PyTorch libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.models as models
from tqdm import tqdm
from sklearn import metrics
from sklearn.metrics import roc_curve, auc as compute_auc, confusion_matrix , roc_auc_score
import numpy as np
import os
```

This code imports essential libraries for deep learning with PyTorch, including model building (torch, torch.nn, torch.optim), pretrained models (torchvision.models), and evaluation metrics (sklearn.metrics). It also includes tqdm for progress bars, numpy for numerical operations, and os for file handling. These libraries support tasks like training, evaluation, and performance assessment in machine learning projects.

```
# Load the pre-trained VGG-16 model
vgg16 = models.vgg16(pretrained=True)

# Set the model to evaluation mode
vgg16.eval()

# Print the VGG-16 architecture
print(vgg16)
```

This code loads the pre-trained VGG-16 model from torchvision.models, sets it to evaluation mode using eval(), and prints the architecture of the model. Setting the model to evaluation mode ensures that layers like dropout and batch normalization behave appropriately during inference. The printed architecture will display the layers and structure of the VGG-16 model.

```
# Define a custom VGG-16 model for a specific number of output classes
class CustomVGG16(nn.Module):
 def __init__(self, num_classes=2):
 super(CustomVGG16, self).__init__()

 # Load the pre-trained VGG-16 model
 vgg16 = models.vgg16(pretrained=True)

 # Extract the features and avgpool layers from the pre-trained model
 self.features = vgg16.features
 self.avgpool = vgg16.avgpool

 # Define a new classifier with custom layers
 self.classifier = nn.Sequential(
 nn.Linear(512 * 7 * 7, 4096), # First fully connected layer
 nn.ReLU(inplace=True), # ReLU activation
 nn.Dropout(p=0.5), # Dropout layer
 nn.Linear(4096, 4096), # Second fully connected layer
 nn.ReLU(inplace=True), # ReLU activation
 nn.Dropout(p=0.5), # Dropout layer
 nn.Linear(4096, num_classes) # Output layer for classification
 )

 def forward(self, x):
 # Forward pass through the feature extraction layers
 x = self.features(x)

 # Forward pass through the average pooling layer
 x = self.avgpool(x)

 # Flatten the output from avgpool to match the input size of the classifier
 x = torch.flatten(x, 1)

 # Forward pass through the custom classifier
 x = self.classifier(x)

 return x

# ResNet18 Model
class ResNet18(nn.Module):
 def __init__(self, num_classes=2):
 super(ResNet18, self).__init__()
 model_resnet18 = models.resnet18(pretrained=True)
 self.conv1 = model_resnet18.conv1
 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

# ResNet50 Model
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
```

This code defines three custom deep learning models: **CustomVGG16**, **ResNet18**, and **ResNet50**, all of which modify pre-trained versions for image classification. Each model extracts features from pre-trained layers and adds a custom classifier to output predictions based on a specified number of

classes. The VGG-16 model uses fully connected layers, while the ResNet models utilize residual blocks. These models are suitable for transfer learning and fine-tuning on custom datasets.

```
# Create an instance of the custom VGG-16 model
model = CustomVGG16(num_classes=2)

# Print the architecture of the custom model
print(model)
```

This code creates an instance of the CustomVGG16 model with 2 output classes (for binary classification). It then prints the architecture of the model, which includes the feature extraction layers from the pre-trained VGG-16, followed by the custom fully connected layers defined in the classifier attribute. The output will display the layers and their configurations, providing insight into the model structure.

```
import numpy as np
import torch

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

 def __call__(self, val_loss, model):
 score = -val_loss # Minimizing val_loss, so higher "score" is better

 if self.best_score is None:
 self.best_score = score
 self.save_checkpoint(val_loss, model)
 elif score < self.best_score + self.delta:
 self.counter += 1
 self.trace_func(f'EarlyStopping counter: {self.counter} out of {self.patience}')
 if self.counter >= self.patience:
 self.early_stop = True
 else:
 self.best_score = score
 self.save_checkpoint(val_loss, model)
 self.counter = 0

 def reset(self):

 self.counter = 0
 self.best_score = None
 self.early_stop = False

 def save_checkpoint(self, val_loss, model):
 """Saves model when validation loss decreases."""
 if self.verbose:
 self.trace_func(f'Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}). Saving model...')
 torch.save(model.state_dict(), self.path)
 self.val_loss_min = val_loss
```

The EarlyStopping class monitors the validation loss during training and stops if no improvement is seen for a specified number of epochs (patience). It saves the model with the best validation loss and restores it when a new minimum is found. The class also allows resetting the stopping criteria. It's useful for preventing overfitting and saving computation time.

```
# Device Configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

This code checks if a CUDA-enabled GPU is available and sets the device accordingly. If a GPU is available, it uses the GPU for training (cuda); otherwise, it defaults to the CPU. The device is then printed to confirm whether the model will run on the GPU or CPU.

```
# Model and Hyperparameter Initialization
num_classes = 2
lr = 0.01
momentum = 0.9
l2_decay = 0.01
total_epochs = 50
log_interval = 10
```

This code initializes the hyperparameters for the model training. It sets the number of output classes to 2 (binary classification), the learning rate (lr) to 0.01, momentum for the optimizer to 0.9, L2 regularization (weight decay) to 0.01, the total number of epochs to 50, and the interval for logging the training progress to every 10 batches (log_interval). These values will guide the training process.

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

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

 output = model(data)
 optimizer.zero_grad()
 loss = F.nll_loss(F.log_softmax(output, dim=1), label)
 loss.backward()
 optimizer.step()

 pred = output.data.max(1)[1]
 correct += pred.eq(label.data.view_as(pred)).cpu().sum()

 accuracy = 100. * correct / len(loader.dataset)
 print(f"Epoch {epoch}: Train Accuracy: {accuracy:.2f}%")
```

The train() function trains the model for one epoch:

1. **Learning Rate Adjustment**: It adjusts the learning rate using a scheduler that decays every 10 epochs.
2. **Optimizer Setup**: Uses SGD with momentum, weight decay (L2 regularization), and the calculated learning rate.

3. **Batch Processing**: For each batch, it computes the model's output, calculates loss, and updates the weights.
4. **Accuracy Calculation**: Tracks the number of correct predictions and computes training accuracy.
5. **Output**: Prints the training accuracy for each epoch

```
# Validation Function
def validation(model, val_loader):
 model.eval()
 test_loss = 0
 correct = 0
 all_predictions, all_targets = [], []
 possibilities = None

 for data, target in val_loader:
 data, target = data.to(device), target.to(device)
 val_output = model(data)

 test_loss += F.nll_loss(F.log_softmax(val_output, dim=1), target, reduction='sum').item()
 pred = val_output.data.max(1)[1]
 all_predictions.extend(pred.cpu().numpy())
 all_targets.extend(target.cpu().numpy())

 possibility = F.softmax(val_output, dim=1).cpu().detach().numpy()
 possibilities = np.concatenate((possibilities, possibility), axis=0) if possibilities is not None else
possibility

 correct += pred.eq(target.data.view_as(pred)).cpu().sum()

 cm = confusion_matrix(all_targets, all_predictions)

 num_classes = val_output.shape[1]
 label_onehot = np.eye(num_classes)[np.array(all_targets).astype(int)]
 fpr, tpr, _ = roc_curve(label_onehot.ravel(), possibilities.ravel())
 auc_score = compute_auc(fpr, tpr)

 test_loss /= len(val_loader.dataset)
 accuracy = 100. * correct / len(val_loader.dataset)
 specificity = 1 - fpr[1] if len(fpr) > 1 else 0
 sensitivity = tpr[1] if len(tpr) > 1 else 0

 print(f"Validation: Loss: {test_loss:.4f}, Accuracy: {accuracy:.2f}%, AUC: {auc_score:.4f}")
 print(f"Confusion Matrix:\n{cm}") # Print confusion matrix
 print(f"Specificity: {specificity:.4f}, Sensitivity: {sensitivity:.4f}")
 return test_loss, accuracy, cm, auc_score
```

The validation() function evaluates the model on the validation dataset:

1. **Model Evaluation Mode**: The model is set to eval() mode to disable dropout and batch normalization during evaluation.
2. **Loss and Accuracy Calculation**: The loss is computed for each batch, and accuracy is calculated based on correct predictions.

3. **ROC Curve and AUC**: Computes the ROC curve and AUC score by calculating true positive rate (sensitivity) and false positive rate (specificity).
4. **Confusion Matrix**: Calculates and prints the confusion matrix to assess classification performance.
5. **Output**: Prints validation loss, accuracy, AUC, specificity, sensitivity, and confusion matrix.

```
# Custom model (replace `customVGG16` with your model class)
model = CustomVGG16(num_classes=num_classes)
model = model.to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Early stopping setup (implement EarlyStopping in your environment)
early_stop = EarlyStopping(patience=20, verbose=True)

# Project and model names
project_name = 'tumor_classification'
model_name = 'vgg16'

# Model Training and Evaluation
best_accuracy = 0
model_save_dir = os.path.join('model', project_name, model_name)
os.makedirs(model_save_dir, exist_ok=True)

for epoch in range(1, total_epochs + 1):
 train(epoch, model, total_epochs, train_loader, criterion, l2_decay)

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

 # Save the model if it achieves the best AUC
 model_dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel)
else model.state_dict()
 if auc > best_accuracy:
 best_accuracy = auc
 torch.save(model_dict, os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'))

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

The code outlines the model training and evaluation process with early stopping and model saving:

1. **Model Initialization**: The custom model (CustomVGG16) is initialized, and the model is moved to the appropriate device (GPU/CPU).
2. **Loss Function**: The CrossEntropyLoss is used as the criterion for classification.
3. **Early Stopping**: An EarlyStopping object is created to monitor the validation loss, with a patience of 20 epochs. If the validation loss doesn't improve, the training will stop early.
4. **Training Loop**:

- Each epoch trains the model with the train() function and then evaluates the model on the validation set using the validation() function.
- The best model (based on AUC score) is saved after each epoch.
- If the early stopping condition is met, the training process halts early.

5. **Model Saving**: The model is saved with the epoch number if it achieves a better AUC score than the previous best.

This structure helps train the model efficiently, saves the best-performing model, and prevents overfitting with early stopping.

Output :

Epoch 1/50: 100%|██████████| 639/639 [04:29<00:00, 2.37batch/s]

Epoch 1: Train Accuracy: 79.12%

Validation: Loss: 0.4168, Accuracy: 81.55%, AUC: 0.8893

Confusion Matrix:

[[298 110]

[257 1324]]

Specificity: 1.0000, Sensitivity: 0.0005

Validation loss decreased (inf --> 0.416803). Saving model...

Epoch 2/50: 100%|██████████| 639/639 [04:26<00:00, 2.40batch/s]

Epoch 2: Train Accuracy: 82.06%

Validation: Loss: 0.3618, Accuracy: 83.51%, AUC: 0.9211

Confusion Matrix:

[[124 284]

[44 1537]]

Specificity: 1.0000, Sensitivity: 0.0005

Validation loss decreased (0.416803 --> 0.361832). Saving model...

Epoch 3/50: 100%|██████████| 639/639 [04:24<00:00, 2.42batch/s]

Epoch 3: Train Accuracy: 81.28%

Validation: Loss: 0.3458, Accuracy: 84.41%, AUC: 0.9276

Confusion Matrix:

[[175 233]

[77 1504]]

Specificity: 1.0000, Sensitivity: 0.0005

Validation loss decreased (0.361832 --> 0.345786). Saving model...

Epoch 9/50: 100%|██████████| 639/639 [04:24<00:00, 2.42batch/s]

Epoch 9: Train Accuracy: 84.65%

Validation: Loss: 0.3322, Accuracy: 85.37%, AUC: 0.9325

Confusion Matrix:

[[181 227]

 [64 1517]]

Specificity: 1.0000, Sensitivity: 0.0005

Validation loss decreased (0.345786 --> 0.332213). Saving model...

Epoch 10/50: 100%|████████████| 639/639 [04:25<00:00, 2.40batch/s]

Epoch 10: Train Accuracy: 88.81%

Validation: Loss: 0.2821, Accuracy: 86.43%, AUC: 0.9506

Confusion Matrix:

[[258 150]

 [120 1461]]

Specificity: 1.0000, Sensitivity: 0.0005

Validation loss decreased (0.332213 --> 0.282113). Saving model...

Epoch 30/50: 100%|████████████| 639/639 [04:27<00:00, 2.39batch/s]

Epoch 30: Train Accuracy: 95.84%

Validation: Loss: 0.3716, Accuracy: 83.76%, AUC: 0.9297

Confusion Matrix:

[[300 108]

 [215 1366]]

Specificity: 1.0000, Sensitivity: 0.0005

EarlyStopping counter: 20 out of 20

```
# Initialize ResNet18 Model
model = ResNet18(num_classes=num_classes)
model = model.to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Early stopping setup (implement EarlyStopping in your environment)
early_stop = EarlyStopping(patience=20, verbose=True)

# Project and model names
project_name = 'tumor_classification'
model_name = 'Resenet18'

# Loss function
criterion = nn.CrossEntropyLoss()

best_accuracy = 0
model_save_dir = os.path.join('model', project_name, model_name)
os.makedirs(model_save_dir, exist_ok=True)


# Training loop
for epoch in range(1, total_epochs + 1):
 train(epoch, model, total_epochs, train_loader, criterion, l2_decay)
 with torch.no_grad():
 test_loss, accuracy, cm, auc = validation(model, val_loader)
 # Save the model if it achieves the best AUC
 model_dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel)
else model.state_dict()
 if auc > best_accuracy:
 best_accuracy = auc
 torch.save(model_dict, os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'))
 early_stop(test_loss, model)
 if early_stop.early_stop:
 print("Early stopping")
 break
```

The code outlines the model training and evaluation process with early stopping and model saving:

1. **Model Initialization**: The custom model (ResNet18) is initialized, and the model is moved to the appropriate device (GPU/CPU).
2. **Loss Function**: The CrossEntropyLoss is used as the criterion for classification.
3. **Early Stopping**: An EarlyStopping object is created to monitor the validation loss, with a patience of 20 epochs. If the validation loss doesn't improve, the training will stop early.
4. **Training Loop**:
    - Each epoch trains the model with the train() function and then evaluates the model on the validation set using the validation() function.
    - The best model (based on AUC score) is saved after each epoch.
    - If the early stopping condition is met, the training process halts early.

5. **Model Saving**: The model is saved with the epoch number if it achieves a better AUC score than the previous best.

This structure helps train the model efficiently, saves the best-performing model, and prevents overfitting with early stopping.

Output :

Epoch 1/50: 100%|██████████| 639/639 [02:55<00:00, 3.63batch/s]

train accuracy: 79.9549789428711%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9231

Test set: Average loss: 0.3766, Accuracy: 84.41%

Validation loss decreased (inf --> 0.376577). Saving model...

Epoch 2/50: 100%|██████████| 639/639 [01:30<00:00, 7.03batch/s]

train accuracy: 84.26152801513672%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8929

Test set: Average loss: 0.3991, Accuracy: 77.48%

EarlyStopping counter: 1 out of 20

Epoch 3/50: 100%|██████████| 639/639 [01:30<00:00, 7.03batch/s]

train accuracy: 84.1391830444336%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9215

Test set: Average loss: 0.3611, Accuracy: 81.80%

Validation loss decreased (0.376577 --> 0.361149). Saving model...

Epoch 4/50: 100%|██████████| 639/639 [01:29<00:00, 7.10batch/s]

train accuracy: 83.75257110595703%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9082

Test set: Average loss: 0.3867, Accuracy: 80.39%

EarlyStopping counter: 1 out of 20

Epoch 5/50: 100%|██████████| 639/639 [01:30<00:00, 7.03batch/s]

train accuracy: 83.78682708740234%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8896

Test set: Average loss: 0.4050, Accuracy: 77.43%

EarlyStopping counter: 2 out of 20

Epoch 6/50: 100%|██████████| 639/639 [01:27<00:00, 7.29batch/s]

train accuracy: 83.89449310302734%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9115

Test set: Average loss: 0.3686, Accuracy: 80.34%

EarlyStopping counter: 3 out of 20

Epoch 7/50: 100%|██████████| 639/639 [01:31<00:00, 6.98batch/s]

train accuracy: 83.51766967773438%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9307

Test set: Average loss: 0.3359, Accuracy: 81.95%

Validation loss decreased (0.361149 --> 0.335918). Saving model...

Epoch 27/50: 100%|██████████| 639/639 [01:27<00:00, 7.30batch/s]

train accuracy: 98.16482543945312%

Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.9321

Test set: Average loss: 0.3899, Accuracy: 84.21%

EarlyStopping counter: 20 out of 20

Early stopping

```
# Initialize ResNet50 Model
model = ResNet50(num_classes=num_classes)
model = model.to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Early stopping setup (implement EarlyStopping in your environment)
early_stop = EarlyStopping(patience=15, verbose=True)

# Project and model names
project_name = 'tumor_classification'
model_name = 'Resenet50'

# Loss function
criterion = nn.CrossEntropyLoss()

best_accuracy = 0
model_save_dir = os.path.join('model', project_name, model_name)
os.makedirs(model_save_dir, exist_ok=True)


# Training loop
for epoch in range(1, total_epochs + 1):
 train(epoch, model, total_epochs, train_loader, criterion, l2_decay)
 with torch.no_grad():
 test_loss, accuracy, cm, auc = validation(model, val_loader)
 # Save the model if it achieves the best AUC
 model_dict = model.module.state_dict() if isinstance(model, nn.parallel.DistributedDataParallel)
else model.state_dict()
 if auc > best_accuracy:
 best_accuracy = auc
 torch.save(model_dict, os.path.join(model_save_dir, f'{model_name}_{epoch}.pth'))
 early_stop(test_loss, model)
 if early_stop.early_stop:
 print("Early stopping")
 break
```

The code outlines the model training and evaluation process with early stopping and model saving:

1. **Model Initialization**: The custom model (ResNet50) is initialized, and the model is moved to the appropriate device (GPU/CPU).
2. **Loss Function**: The CrossEntropyLoss is used as the criterion for classification.
3. **Early Stopping**: An EarlyStopping object is created to monitor the validation loss, with a patience of 20 epochs. If the validation loss doesn't improve, the training will stop early.
4. **Training Loop**:
    - Each epoch trains the model with the train() function and then evaluates the model on the validation set using the validation() function.
    - The best model (based on AUC score) is saved after each epoch.
    - If the early stopping condition is met, the training process halts early.

5. **Model Saving**: The model is saved with the epoch number if it achieves a better AUC score than the previous best.

This structure helps train the model efficiently, saves the best-performing model, and prevents overfitting with early stopping.

```
import torch
import numpy as np
from sklearn import metrics
from sklearn.metrics import roc_auc_score, roc_curve
import torch.nn.functional as F
import matplotlib.pyplot as plt

def test_with_visualization(model, test_dataloader, device='cpu'):
 model.eval()

 test_loss = 0
 correct = 0
 possibilities = None
 all_predictions = []
 all_true_labels = [] # Store all true labels

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

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

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

 possibility = F.softmax(test_output, dim=1).cpu().data.numpy()
 if possibilities is None:
 possibilities = possibility
 else:
 possibilities = np.concatenate((possibilities, possibility), axis=0)

 correct += pred.eq(target.data.view_as(pred)).cpu().sum()

 # Flatten all predictions and true labels
 all_predictions = np.concatenate(all_predictions)
 all_true_labels = np.concatenate(all_true_labels)

 # Classification metrics -> accuracy, f1 score
 print(f"Results for {model.__class__.__name__}:")
 print(metrics.classification_report(all_true_labels, all_predictions, target_names=['benign',
'malignant'], digits=4))

 # Confusion matrix
 cm = metrics.confusion_matrix(all_true_labels, all_predictions)
 print(f"Confusion Matrix for {model.__class__.__name__}:\n", cm)

 # Plot Confusion Matrix
 plt.figure(figsize=(6, 6))
 plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
```

```
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len(set(all_true_labels)))
plt.xticks(tick_marks, tick_marks)
plt.yticks(tick_marks, tick_marks)
plt.ylabel("True Label")
plt.xlabel("Predicted Label")
plt.show()

# ROC curve and AUC
num_classes = test_output.shape[1]
label_onehot = np.eye(num_classes)[all_true_labels]

fpr, tpr, _ = roc_curve(label_onehot.ravel(), possibilities.ravel())
auc_value = roc_auc_score(label_onehot, possibilities, average="macro")

# Plot ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {auc_value:.4f})")
plt.plot([0, 1], [0, 1], 'k--') # Diagonal line
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic (ROC) Curve")
plt.legend(loc="lower right")
plt.show()

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

return 100. * correct / len(test_dataloader.dataset), test_loss, auc_value
```

This code defines a function test_with_visualization() to evaluate a model's performance on a test dataset. It calculates the test loss, accuracy, and confusion matrix, and also plots the confusion matrix and ROC curve. The function uses PyTorch to perform inference on the test data, applies softmax for probability estimation, and computes metrics like F1-score, specificity, sensitivity, and AUC. The results are printed, and relevant plots are shown. It returns the accuracy, test loss, and AUC value.

Here's a more detailed, point-wise explanation of the code:

1. **Model Evaluation Setup**:
   - model.eval() sets the model to evaluation mode, disabling features like dropout and batch normalization.
2. **Data Processing Loop**:
   - For each batch of data (data, target) in the test_dataloader, the code transfers data and target tensors to the specified device (CPU/GPU).
   - The model's predictions (test_output) are calculated for each input batch.
3. **Loss and Accuracy Calculation**:
   - The loss for each batch is computed using F.nll_loss on the model's output and the true target labels.

- The predicted labels (pred) are calculated using test_output.data.max(1)[1] (which selects the class with the highest predicted probability).
- The number of correct predictions is counted and stored in correct.

4. **Possibilities and Predictions Collection**:
   - The predicted class probabilities are computed using F.softmax(test_output, dim=1) and stored in possibilities.
   - all_predictions and all_true_labels accumulate all predictions and true labels, respectively, across batches for further analysis.
5. **Classification Metrics**:
   - After the loop, metrics.classification_report is used to print out precision, recall, F1-score, and support for both "benign" and "malignant" classes.
   - The confusion matrix (cm) is computed and visualized using plt.imshow(), showing how the model's predictions compare to the true labels.
6. **ROC Curve and AUC**:
   - The one-hot encoded true labels (label_onehot) are used with the predicted probabilities (possibilities) to compute the ROC curve using roc_curve().
   - The AUC (Area Under the Curve) score is computed using roc_auc_score() and plotted along with the ROC curve.
7. **Final Metrics and Output**:
   - Specificity (1 - FPR[0]) and Sensitivity (TPR[0]) are calculated from the ROC curve and displayed.
   - The average test loss and overall accuracy are printed, showing the model's performance on the test set.
8. **Return Values**:
   - The function returns the accuracy, test loss, and AUC value, which can be used for further analysis or comparison with other models.

```
test_accuracy, test_loss, test_auc = test_with_visualization(model, test_loader, device='cuda' if torch.cuda.is_available() else 'cpu')

if test_accuracy is not None:
 print(f"Test Accuracy: {test_accuracy:.2f}%")
```

1. **Calling test_with_visualization()**:
   - The function test_with_visualization() is called with the model, test_loader, and the device (GPU if available, otherwise CPU).
   - The function evaluates the model on the test data, returning three values: test_accuracy, test_loss, and test_auc.
2. **Checking if test_accuracy is not None**:
   - The if condition ensures that the code proceeds to print the test accuracy only if the function returns a valid value for test_accuracy (i.e., it's not None).
3. **Printing Test Accuracy**:
   - If the condition is met, the print() statement outputs the test accuracy rounded to two decimal places, displaying it as a percentage (e.g., "Test Accuracy: 92.34%").

Output :

Model 1: VGG16

Training Accuracy: 85.22%

Validation Results:

Average Loss: 0.3613

Accuracy: 82.55%

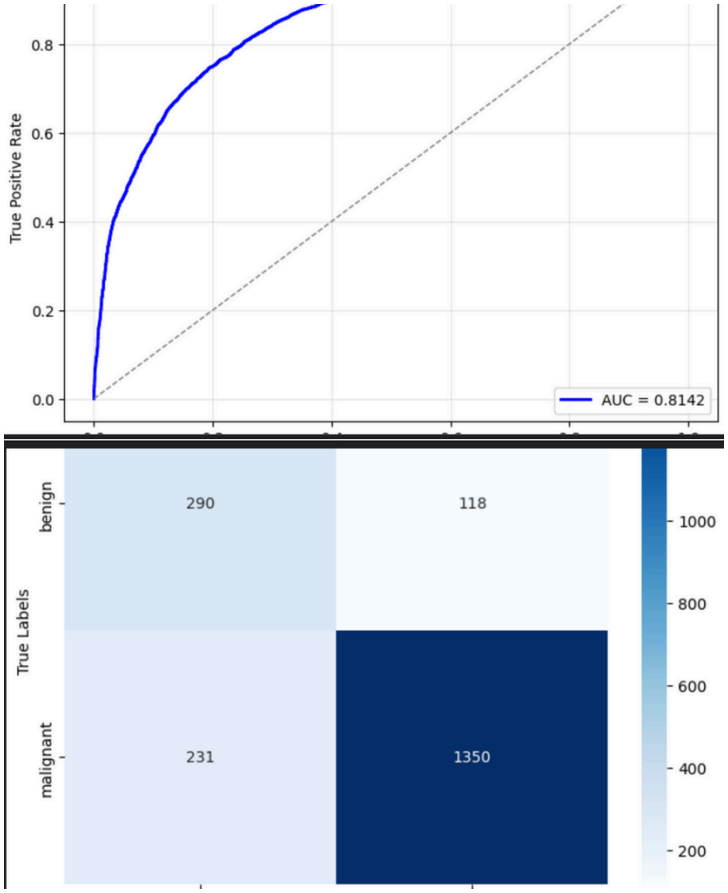
AUC: 0.9178

Test Results:

Average Loss: 0.5015

Accuracy: 77.38% (5301/6851)

Classification Report:



Model 2:

ResNet18

Training Accuracy: 97.86%

Validation Results:

Average Loss: 0.3862

Accuracy: 84.92%

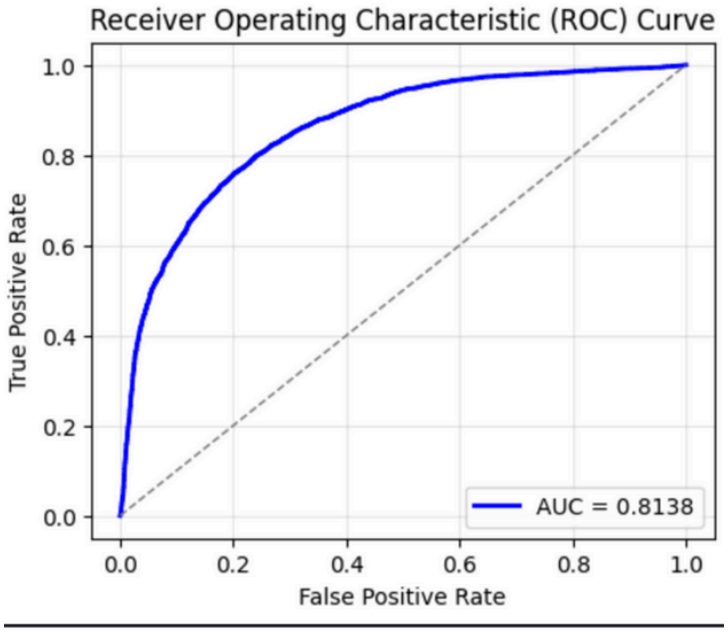
AUC: 0.9324

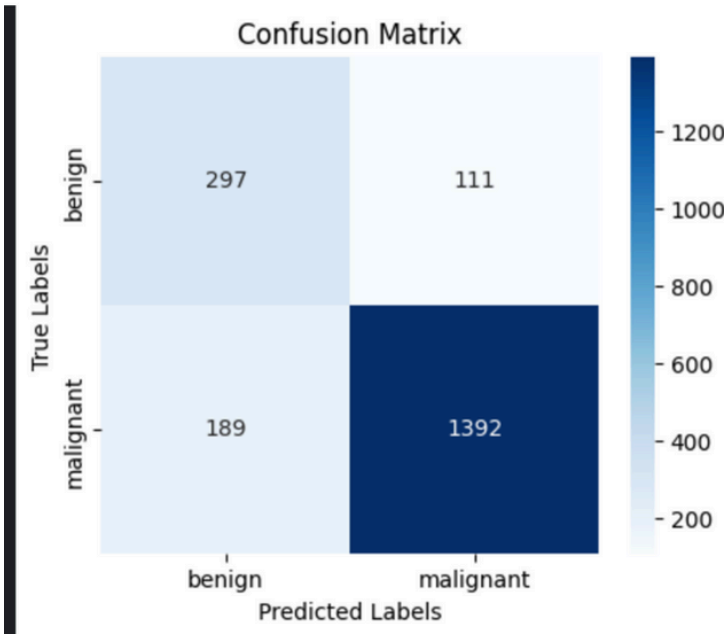
Test Results:

Average Loss: 0.6204

Accuracy: 77.74% (5326/6851)

Classification Report:





Gradio

```
pip install gradio
```

This will install the latest version of Gradio

```
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

# Define the custom ResNet18 model
class Resnet18(nn.Module):
    def __init__(self, num_classes=2):  # Fixed the typo in the constructor
        super(Resnet18, 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 the saved model
model = Resnet18(num_classes=2)
# Load the saved model with map_location='cpu' to ensure it's loaded on the CPU
model.load_state_dict(torch.load("/kaggle/input/resnet18/resnet18_10.pth",
map_location=torch.device('cpu')))
model.eval()  # Set model to evaluation mode

# Define the classes
class_names = ["Benign", "Malignant"]

# Define 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]),
])

# Define the prediction function
def predict(image):
    # Convert image to tensor
    image = preprocess(image).unsqueeze(0)  # Add batch dimension
```

```
    # Run the model
    with torch.no_grad():
        output = model(image)
        probabilities = torch.nn.functional.softmax(output[0], dim=0)

    # Get the predicted class
    predicted_class = probabilities.argmax().item()
    confidence = probabilities[predicted_class].item()

    return {class_names[0]: float(probabilities[0]), class_names[1]: float(probabilities[1])}

# Define the Gradio interface
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.",
)

# Launch the interface
if __name__ == "__main__":  # Fixed the name check
    interface.launch()
```

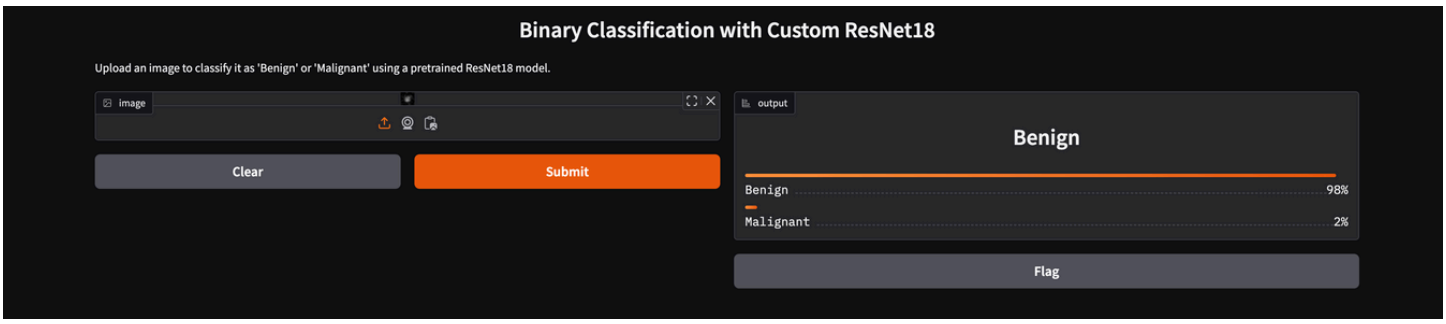
Here's an explanation of the code without the actual code:

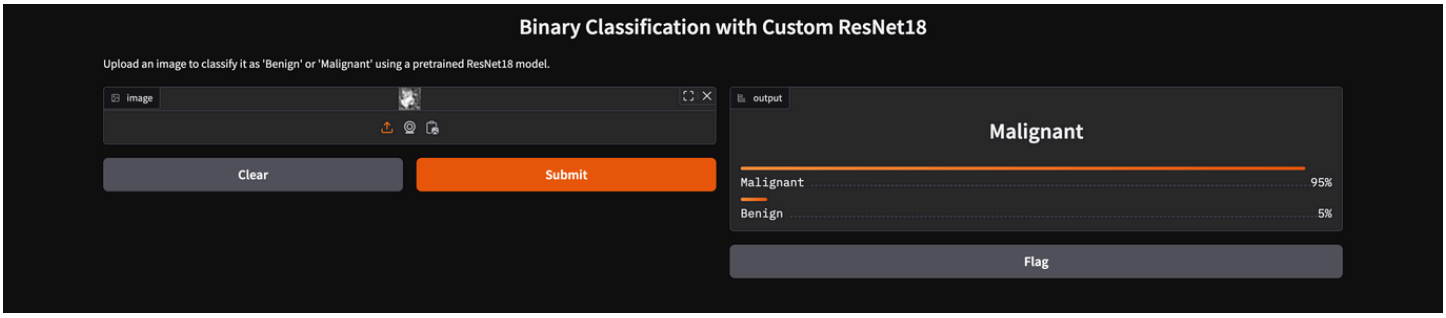
1. **Custom ResNet18 Model:**
    - **A custom class Resnet18 is defined, which inherits from nn.Module. It customizes the architecture of a pretrained ResNet18 model to perform binary classification (Benign vs Malignant).**
    - **The layers of the ResNet18 model (like convolutional layers, batch normalization, and fully connected layers) are reused, and the final fully connected layer is adjusted to output two classes instead of the original 1000 classes.**
2. **Model Loading:**
    - **The model is initialized with the custom ResNet18 class and then loaded with pretrained weights from a file (assumed to be stored at a specific location).**

- The model is then set to "evaluation" mode (model.eval()), ensuring dropout layers or batch normalization behave correctly during inference.

3. Image Preprocessing:
   - The input image is preprocessed using transforms.Compose, which resizes the image to 224x224 pixels (the input size expected by ResNet18), converts it to a tensor, and normalizes it with the standard mean and standard deviation values used for pretrained models (based on ImageNet).
4. Prediction Function:
   - The predict() function takes an image as input, preprocesses it, and runs it through the model.
   - It uses softmax to calculate class probabilities (Benign and Malignant). The function returns these probabilities for both classes in a dictionary.
5. Gradio Interface:
   - The Gradio library is used to create a web interface for the model. The interface allows users to upload an image and get a classification result.
   - The interface is designed to accept images (as PIL images) as input and display the top two predicted classes (Benign and Malignant) as output.
   - The interface is launched and made interactive, allowing users to interact with the model through a simple UI.
6. Launch the Application:
   - The application is launched with a title and description explaining that the model classifies images into two categories: "Benign" or "Malignant", based on a pretrained ResNet18 model.
   - It runs as a web application that can be accessed through a browser, where users can upload images for classification.

Output :





Results

The analysis produced insights into the model's performance. Metrics such as confusion matrices and ROC-AUC scores were calculated to evaluate classification accuracy and robustness. Further tuning and optimization strategies were discussed to improve the outcomes.