Week_11_Model_Explaination

November 19, 2024

[15]:
```
# Standard Libraries
import io
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objects as go

# Deep Learning and PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import models
# Image Processing
from PIL import Image
from torchvision import transforms, models

# File Handling
import h5py

# Metrics and Evaluation
from sklearn.metrics import classification_report, roc_auc_score, roc_curve, auc
from sklearn.utils.class_weight import compute_class_weight
from captum.attr import FeatureAblation

# Progress Visualization
from tqdm import tqdm
```

0.1 Create Custom Dataset

[34]:
```
class MultiInputDataset(Dataset):
    def __init__(self, hdf5_file, df, transform=None):
        # Open the HDF5 file with error handling
        try:
            self.hdf5_file = h5py.File(hdf5_file, 'r')  # Read-only mode
        except Exception as e:
```

```
            raise IOError(f"Could not open HDF5 file: {hdf5_file}. Error: {e}")

        # Read the CSV file containing image labels and additional features
        try:
            self.labels_df = df
        except Exception as e:
            raise IOError(f"Could not read CSV file: {csv_file}. Error: {e}")

        # Ensure that all image IDs from the CSV are present in the HDF5 file
        self.image_ids = self.labels_df['isic_id'].values
        for image_id in self.image_ids:
            if str(image_id) not in self.hdf5_file.keys():
                raise ValueError(f"Image id {image_id} not found in HDF5 file.")

        # Store any transformations to be applied to the images
        self.transform = transform

    def __len__(self):
        # Return the total number of samples in the dataset
        return len(self.labels_df)

    def __getitem__(self, idx):
        # Get the image ID from the CSV file based on index
        image_id = str(self.labels_df.iloc[idx]['isic_id'])

        # Load the image data from the HDF5 file
        image_bytes = self.hdf5_file[image_id][()]

        # Convert the image bytes to a PIL Image
        image = Image.open(io.BytesIO(image_bytes))

        # Apply any specified transformations to the image
        if self.transform:
            image = self.transform(image)

        # Retrieve the label
        label = torch.tensor(self.labels_df.iloc[idx]['target'], dtype=torch.
↪long)  # Adjust dtype if needed

        # Retrieve other features, excluding 'isic_id' and 'target'
        other_variables = self.labels_df.iloc[idx].drop(['isic_id', 'target']).
↪values.astype(float)

        # Convert other variables (metadata) to a tensor
        metadata_tensor = torch.tensor(other_variables, dtype=torch.float32)

        # Return the image, metadata, and label
```

```
        return image, metadata_tensor, label
```

[17]:
```
def get_train_transform(resize_size=(224, 224), crop_size=128,␣
↪rotation_degree=10, normalize_means=(0.5, 0.5, 0.5), normalize_stds=(0.5, 0.
↪5, 0.5)):
    """
    Returns the transformations for the training dataset, including data␣
↪augmentation.

    Args:
        resize_size (tuple): The size to resize the image before cropping.
        crop_size (int): The size of the random crop.
        rotation_degree (int): Maximum degree for random rotation.
        normalize_means (tuple): Means for normalization.
        normalize_stds (tuple): Standard deviations for normalization.

    Returns:
        transforms.Compose: The composed transformations for the training set.
    """
    return transforms.Compose([
        transforms.Resize(resize_size),  # Resize to specified size
        transforms.RandomResizedCrop(crop_size, scale=(0.8, 1.0)),  # Random␣
↪crop with scale
        transforms.RandomRotation(rotation_degree),  # Randomly rotate images
        transforms.ToTensor(),  # Convert image to PyTorch tensor
        transforms.Normalize(normalize_means, normalize_stds)  # Normalize with␣
↪specified means and stds
    ])

def get_normal_transform(resize_size=(224, 224), normalize_means=(0.5, 0.5, 0.
↪5), normalize_stds=(0.5, 0.5, 0.5)):
    """
    Returns the transformations for the validation/test dataset (without data␣
↪augmentation).

    Args:
        resize_size (tuple): The size to resize the image.
        normalize_means (tuple): Means for normalization.
        normalize_stds (tuple): Standard deviations for normalization.

    Returns:
        transforms.Compose: The composed transformations for the validation/
↪test set.
    """
    return transforms.Compose([
        transforms.Resize(resize_size),  # Resize to specified size
```

```
        transforms.ToTensor(),  # Convert image to PyTorch tensor
        transforms.Normalize(normalize_means, normalize_stds)  # Normalize with␣
↪specified means and stds
    ])
```

[18]:
```
# Function to compute partial AUC-above-TPR
def score(solution: np.array, submission: np.array, min_tpr: float = 0.80) ->␣
↪float:
    """
    Compute the partial AUC by focusing on a specific range of true positive␣
↪rates (TPR).

    Args:
        solution (np.array): Ground truth binary labels.
        submission (np.array): Model predictions.
        min_tpr (float): Minimum true positive rate to calculate partial AUC.

    Returns:
        float: The calculated partial AUC.

    Raises:
        ValueError: If the min_tpr is not within a valid range.
    """
    # Rescale the target to handle sklearn limitations and flip the predictions
    v_gt = abs(solution - 1)
    v_pred = -1.0 * submission
    max_fpr = abs(1 - min_tpr)

    # Compute ROC curve using sklearn
    fpr, tpr, _ = roc_curve(v_gt, v_pred)
    if max_fpr is None or max_fpr == 1:
        return auc(fpr, tpr)
    if max_fpr <= 0 or max_fpr > 1:
        raise ValueError(f"Expected min_tpr in range [0, 1), got: {min_tpr}")

    # Interpolate for partial AUC
    stop = np.searchsorted(fpr, max_fpr, "right")
    x_interp = [fpr[stop - 1], fpr[stop]]
    y_interp = [tpr[stop - 1], tpr[stop]]
    tpr = np.append(tpr[:stop], np.interp(max_fpr, x_interp, y_interp))
    fpr = np.append(fpr[:stop], max_fpr)
    partial_auc = auc(fpr, tpr)

    return partial_auc
```

0.2 Train DataLoader

[19]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

0.3 Model Building

[20]:
```
# EfficientNet Model
class CustomImageFeatureEfficientNet(nn.Module):
    def __init__(self, feature_input_size, pretrained=True):
        super(CustomImageFeatureEfficientNet, self).__init__()

        # Load a pretrained EfficientNet model for image feature extraction␣
↪(EfficientNet-B0 in this case)
        efficientnet = models.efficientnet_b0(pretrained=pretrained)  # You can␣
↪change this to another EfficientNet version like B1 or B7
        self.efficientnet = nn.Sequential(*list(efficientnet.children())[:-1]) ␣
↪# Remove the final classification layer

        # The output of EfficientNet-B0's last conv layer is 1280-dimensional
        self.fc_image = nn.Linear(1280, 512)  # Reduce dimension to match your␣
↪custom architecture

        # Fully connected layer for metadata (feature data)
        self.fc_metadata = nn.Linear(feature_input_size, 128)

        # Dropout layer to prevent overfitting
        self.dropout = nn.Dropout(0.5)  # 50% dropout

        # Final fully connected layer for binary classification (combined image␣
↪+ feature input)
        self.fc_combined = nn.Linear(512 + 128, 1)  # For binary classification

    def forward(self, image, metadata):
        # Forward pass for the image through EfficientNet (without the final␣
↪classification layer)
        x = self.efficientnet(image)  # EfficientNet feature extraction
        x = x.view(x.size(0), -1)  # Flatten the EfficientNet output
        image_features = F.relu(self.fc_image(x))

        # Process metadata (feature data)
        metadata_features = F.relu(self.fc_metadata(metadata))

        # Ensure the batch sizes are consistent
        assert image_features.shape[0] == metadata_features.shape[0], \
            f"Batch sizes do not match! Image batch size: {image_features.
↪shape[0]}, Metadata batch size: {metadata_features.shape[0]}"
```

```
        # Concatenate image features and metadata features
        combined_features = torch.cat((image_features, metadata_features),␣
↪dim=1)

        # Dropout and final classification layer
        combined_features = self.dropout(combined_features)
        output = self.fc_combined(combined_features)

        # If you're using BCELoss, uncomment the next line to apply sigmoid
        output = torch.sigmoid(output)

        return output
```

0.4 Winning Model

[21]:
```
#create test dataset
effnet_test_dataset = MultiInputDataset(hdf5_file='../data/raw/test_image.
↪hdf5', df=pd.read_csv('../data/processed/processed-test-metadata1.csv'),␣
↪transform=get_normal_transform(resize_size=224,224)))
# Create test DataLoader
effnet_test_dataloader = DataLoader(effnet_test_dataset,  batch_size=16,␣
↪shuffle=True)
```

[22]:
```
# Initialize the final model with 9 output features
final_model = CustomImageFeatureEfficientNet(9)

# Define the path to the saved model weights
final_model_path = "best_model7.pth"
```

```
/home/jupyter-sohka/.local/lib/python3.10/site-
packages/torchvision/models/_utils.py:208: UserWarning:

The parameter 'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.

/home/jupyter-sohka/.local/lib/python3.10/site-
packages/torchvision/models/_utils.py:223: UserWarning:

Arguments other than a weight enum or `None` for 'weights' are deprecated since
0.13 and may be removed in the future. The current behavior is equivalent to
passing `weights=EfficientNet_B0_Weights.IMAGENET1K_V1`. You can also use
`weights=EfficientNet_B0_Weights.DEFAULT` to get the most up-to-date weights.
```

[23]:
```
# Initialize the final model with 9 output features
final_model = CustomImageFeatureEfficientNet(9)
```

```
# Define the path to the saved model weights
final_model_path = "../models/best_model7.pth"

# Load the model weights into final_model, mapping to CPU if necessary
final_model.load_state_dict(torch.load(final_model_path, map_location=torch.
↪device('cpu')))

# Set the model to evaluation mode, which disables dropout and batch␣
↪normalization updates
final_model.eval()

# Initialize lists to store labels and predicted probabilities for later␣
↪analysis
all_labels, all_probs = [], []

# Disable gradient computation for testing phase to save memory and improve␣
↪performance
with torch.no_grad():
    # Loop through batches in the test dataloader
    for images, metadata, labels in effnet_test_dataloader:
        # Move data to the specified device (e.g., CPU or GPU) and adjust label␣
↪shape
        images, metadata = images.to(device), metadata.to(device)
        labels = labels.float().to(device).unsqueeze(1)

        # Forward pass to get probabilities from the model
        probs = final_model(images, metadata)

        # Collect labels and predicted probabilities, converting to numpy arrays
        all_labels.extend(labels.cpu().numpy())
        all_probs.extend(probs.cpu().numpy())

        # Generate binary predictions based on a 0.5 threshold
        predicted = (probs > 0.5).float()

    # Calculate the partial AUROC score (adjusted for your specific function)
    partial_auroc = score(np.array(all_labels), np.array(all_probs))
    print(f'The partial AUROC of the final model on the test images is␣
↪{partial_auroc}')

    # Print the classification report, evaluating performance on Class 0 and␣
↪Class 1
    print(classification_report(all_labels, (np.array(all_probs) >= 0.5).
↪astype(int), target_names=['Class 0', 'Class 1']))
```

```
/tmp/ipykernel_2234129/4210569010.py:8: FutureWarning:

You are using `torch.load` with `weights_only=False` (the current default
value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.


The partial AUROC of the final model on the test images is 0.13625176183538829
              precision    recall  f1-score   support

     Class 0       0.98      0.92      0.95      1431
     Class 1       0.24      0.61      0.35        59

    accuracy                           0.91      1490
   macro avg       0.61      0.77      0.65      1490
weighted avg       0.95      0.91      0.93      1490
```

0.5 Feature Importance

[10]:
```
# Initialize Feature Ablation
feature_ablation = FeatureAblation(final_model)

# Split metadata into individual features (assuming metadata has shape␣
↪[batch_size, num_features])
individual_metadata_features = [metadata[:, i:i+1] for i in range(metadata.
↪shape[1])]

# Combine all inputs (image + individual metadata features) for ablation
inputs = (images, metadata)

# Perform feature ablation
attributions = feature_ablation.attribute(
    inputs=inputs,  # Include image and metadata
```

```
    target=labels.squeeze().long()  # Ensure the target is correctly shaped
)

# Summarize contributions
# Image importance
image_importance_score = attributions[0].sum().item()
# Load metadata columns
columns = pd.read_csv('../data/processed/processed-train-metadata1.csv').
 ↪columns.tolist()

# Exclude non-metadata columns (e.g., 'isic_id', 'target') if they exist
metadata_columns = [col for col in columns if col not in ['isic_id', 'target']]

# Image importance
image_importance_score = attributions[0].sum().item()

# Metadata importance (considering all features combined)
metadata_attributions = attributions[1]  # Metadata attributions
metadata_importance_scores = {
    metadata_columns[i]: metadata_attributions[:, i].sum().item()
    for i in range(metadata.shape[1])
}

# Print results
print("Image Importance Score:", image_importance_score)
print("Metadata Feature Importance Scores:", metadata_importance_scores)
```

```
Image Importance Score: 28.72800064086914
Metadata Feature Importance Scores: {'age_approx': 0.003150713862851262,
'clin_size_long_diam_mm': 0.0004841720510739833, 'sex_female':
7.600174285471439e-07, 'sex_male': 0.00032329559326171875,
'anatom_site_general_anterior torso': 6.821283022873104e-07,
'anatom_site_general_head/neck': 0.0, 'anatom_site_general_lower extremity':
-0.0012100040912628174, 'anatom_site_general_posterior torso': 0.0,
'anatom_site_general_upper extremity': 0.0}
```

0.6 Extracting and analyzing 5 individual predictions

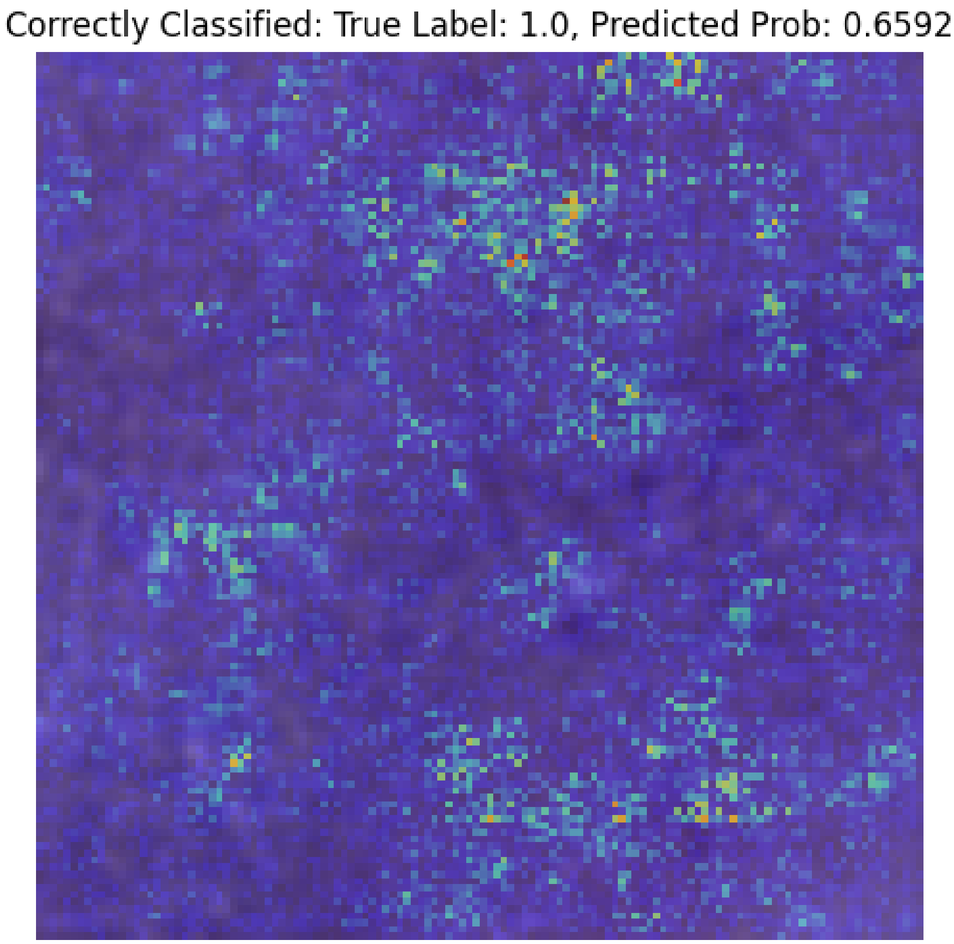
[12]:
```
# Initialize lists to store true labels, predicted probabilities, and both␣
 ↪correctly/misclassified examples
all_labels, all_probs = [], []
misclassified_images = []  # To store misclassified samples
correctly_classified_images = []  # To store correctly classified samples

# Iterate through the test dataloader
try:
    for i, (image, metadata, labels) in enumerate(effnet_test_dataloader):
```

```
        # Move data to the specified device
        image, metadata, labels = (
            image.to(device),
            metadata.to(device),
            labels.float().to(device).unsqueeze(1)  # Ensure labels have the␣
↪correct shape
        )

        # Clone tensors to make them leaf variables and set requires_grad=True
        image = image.clone().detach().requires_grad_(True)
        metadata = metadata.clone().detach().requires_grad_(True)

        # Forward pass with gradient tracking
        final_model.zero_grad()  # Clear previous gradients
        output = final_model(image, metadata)

        # Store true labels and predicted probabilities for analysis
        all_labels.extend(labels.cpu().numpy())
        all_probs.extend(output.detach().cpu().numpy())  # Detach probabilities␣
↪to avoid retaining graph

        # Calculate binary predictions (0 or 1)
        predicted = (output > 0.5).float()
        classification_indices = (predicted == labels).squeeze().cpu().numpy() ␣
↪# True if correct, False otherwise

        # Separate correctly classified and misclassified samples
        for j, is_correct in enumerate(classification_indices):
            if is_correct and labels[j].item() == 1:  # Filter for true label =␣
↪1
                correctly_classified_images.append((image[j], metadata[j],␣
↪labels[j].item(), output[j].item()))  # Store tensors and predictions
            elif not is_correct and labels[j].item() == 1:  # Filter for true␣
↪label = 1
                misclassified_images.append((image[j], metadata[j], labels[j].
↪item(), output[j].item()))  # Store tensors and predictions

    # Display heatmaps for correctly classified images with true label 1
    print("Displaying correctly classified samples with true label 1:")
    for i, (img_tensor, metadata_tensor, true_label, pred_prob) in␣
↪enumerate(correctly_classified_images[:3]):  # Show up to 3 images
        img_tensor = img_tensor.unsqueeze(0).clone().detach().
↪requires_grad_(True)  # Add batch dimension
        metadata_tensor = metadata_tensor.unsqueeze(0).clone().detach().
↪requires_grad_(True)  # Add batch dimension
```

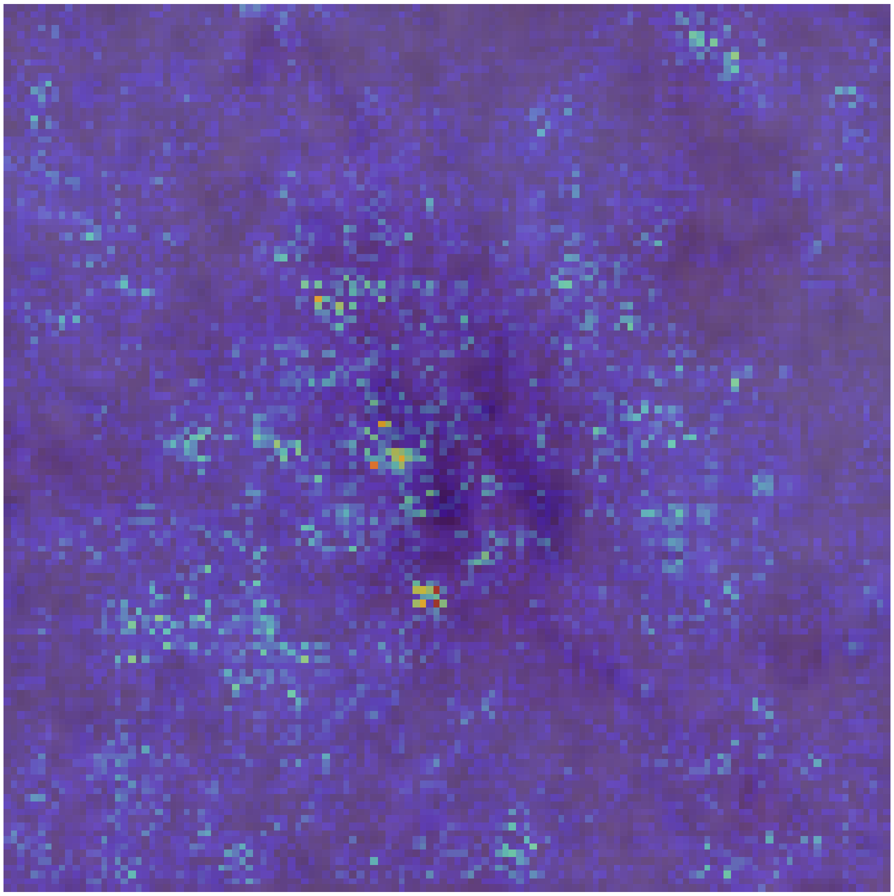
```
        # Perform forward pass with gradient tracking
        final_model.zero_grad()  # Clear gradients
        output = final_model(img_tensor, metadata_tensor)
        target_class = output.argmax(dim=1)  # Get the target class
        output[0, target_class].backward()  # Compute gradients

        # Compute gradient magnitude for visualization
        gradient_magnitude = img_tensor.grad.abs().squeeze().cpu().numpy()
        if gradient_magnitude.ndim == 3:
            gradient_magnitude = np.mean(gradient_magnitude, axis=0)  # Average␣
↪over color channels

        # Normalize gradient magnitude to [0, 1]
        gradient_magnitude = (gradient_magnitude - gradient_magnitude.min()) / (
            gradient_magnitude.max() - gradient_magnitude.min()
        )

        # Denormalize the image
        img_array = img_tensor.detach().cpu().numpy().squeeze().transpose(1, 2,␣
↪0)  # Convert to HWC format
        mean = np.array([0.5, 0.5, 0.5])  # Replace with your normalization mean
        std = np.array([0.5, 0.5, 0.5])   # Replace with your normalization std
        img_array = std * img_array + mean  # Denormalize
        img_array = np.clip(img_array, 0, 1)  # Clip to [0, 1]

        # Overlay heatmap on the original image
        fig, ax = plt.subplots(figsize=(6, 6))
        ax.imshow(img_array, cmap=None if img_array.shape[-1] == 3 else 'gray')␣
↪ # Show the original image
        ax.imshow(gradient_magnitude, cmap='jet', alpha=0.5)  # Overlay the␣
↪heatmap with transparency
        ax.set_title(f"Correctly Classified: True Label: {true_label},␣
↪Predicted Prob: {pred_prob:.4f}")
        ax.axis('off')
        plt.show()

    # Display heatmaps for misclassified images with true label 1
    print("Displaying misclassified samples with true label 1:")
    for i, (img_tensor, metadata_tensor, true_label, pred_prob) in␣
↪enumerate(misclassified_images[:3]):  # Show up to 3 images
        img_tensor = img_tensor.unsqueeze(0).clone().detach().
↪requires_grad_(True)  # Add batch dimension
        metadata_tensor = metadata_tensor.unsqueeze(0).clone().detach().
↪requires_grad_(True)  # Add batch dimension

        # Perform forward pass with gradient tracking
```

```
        final_model.zero_grad()  # Clear gradients
        output = final_model(img_tensor, metadata_tensor)
        target_class = output.argmax(dim=1)  # Get the target class
        output[0, target_class].backward()  # Compute gradients

        # Compute gradient magnitude for visualization
        gradient_magnitude = img_tensor.grad.abs().squeeze().cpu().numpy()
        if gradient_magnitude.ndim == 3:
            gradient_magnitude = np.mean(gradient_magnitude, axis=0)  # Average␣
↪over color channels

        # Normalize gradient magnitude to [0, 1]
        gradient_magnitude = (gradient_magnitude - gradient_magnitude.min()) / (
            gradient_magnitude.max() - gradient_magnitude.min()
        )

        # Denormalize the image
        img_array = img_tensor.detach().cpu().numpy().squeeze().transpose(1, 2,␣
↪0)  # Convert to HWC format
        mean = np.array([0.5, 0.5, 0.5])  # Replace with your normalization mean
        std = np.array([0.5, 0.5, 0.5])   # Replace with your normalization std
        img_array = std * img_array + mean  # Denormalize
        img_array = np.clip(img_array, 0, 1)  # Clip to [0, 1]

        # Overlay heatmap on the original image
        fig, ax = plt.subplots(figsize=(6, 6))
        ax.imshow(img_array, cmap=None if img_array.shape[-1] == 3 else 'gray')␣
↪ # Show the original image
        ax.imshow(gradient_magnitude, cmap='jet', alpha=0.5)  # Overlay the␣
↪heatmap with transparency
        ax.set_title(f"Misclassified: True Label: {true_label}, Predicted Prob:␣
↪{pred_prob:.4f}")
        ax.axis('off')
        plt.show()

except Exception as e:
    print(f"Error during model evaluation: {e}")
```

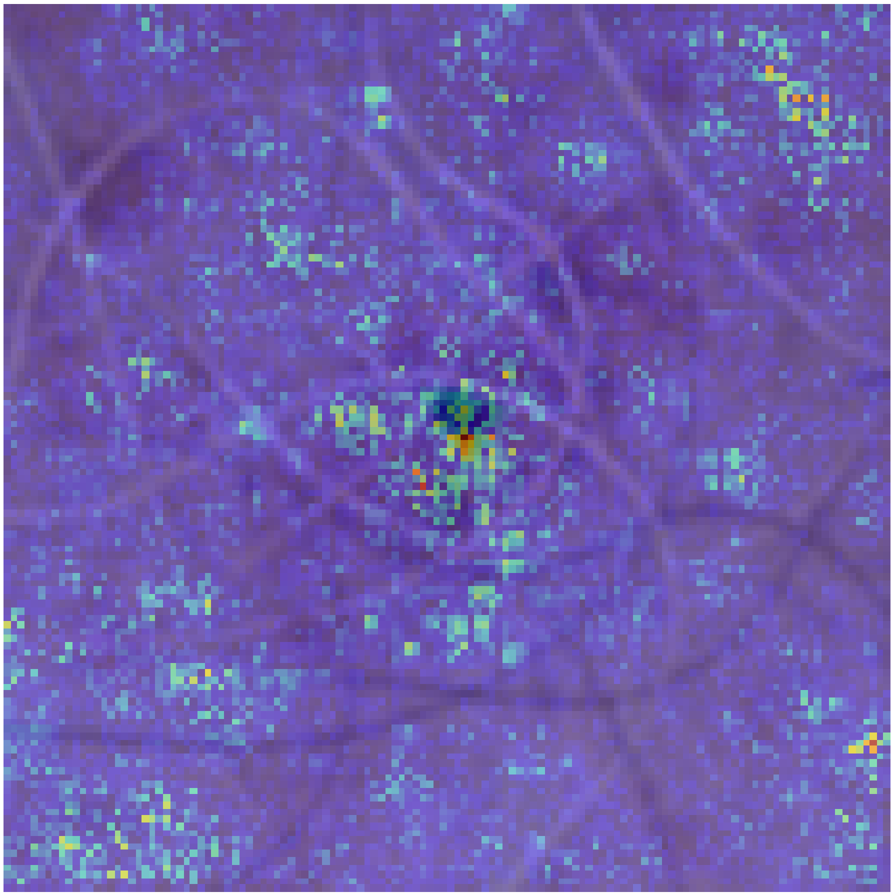
```
Displaying correctly classified samples with true label 1:
```



Correctly Classified: True Label: 1.0, Predicted Prob: 0.8670

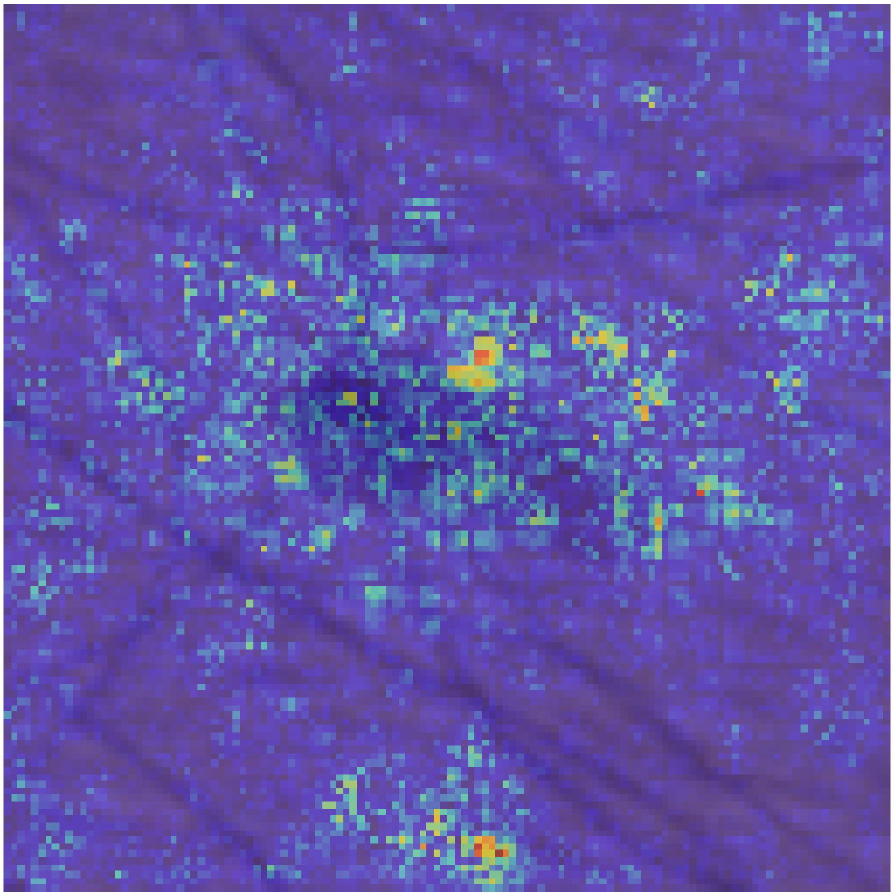


Correctly Classified: True Label: 1.0, Predicted Prob: 0.8317

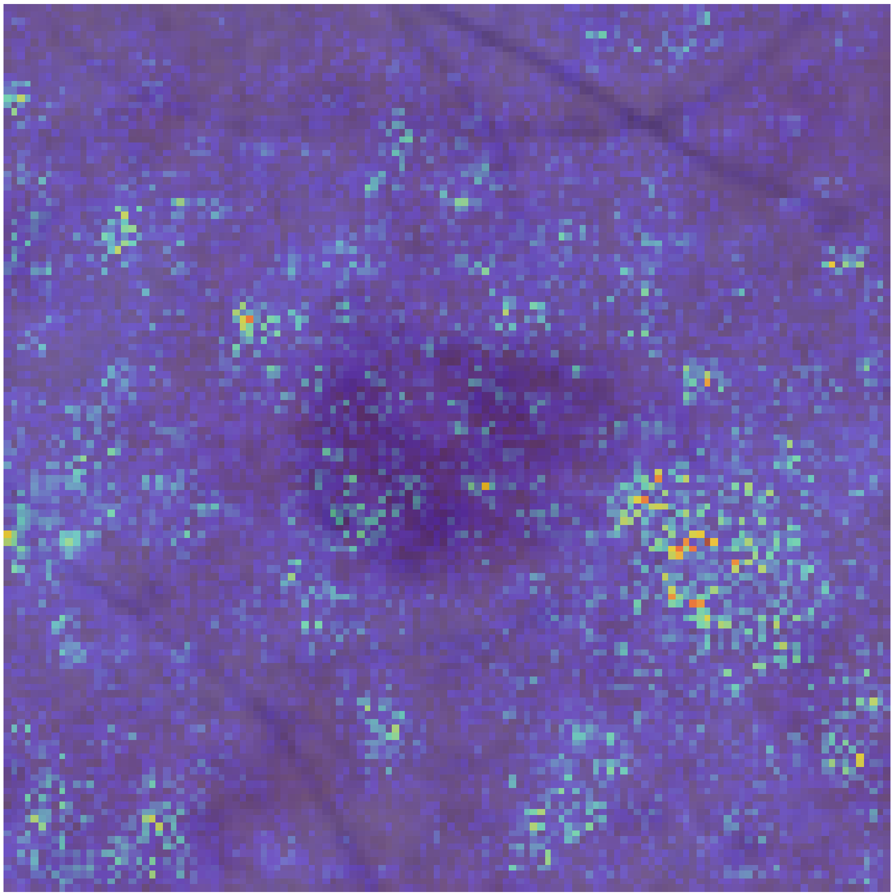


```
Displaying misclassified samples with true label 1:
```

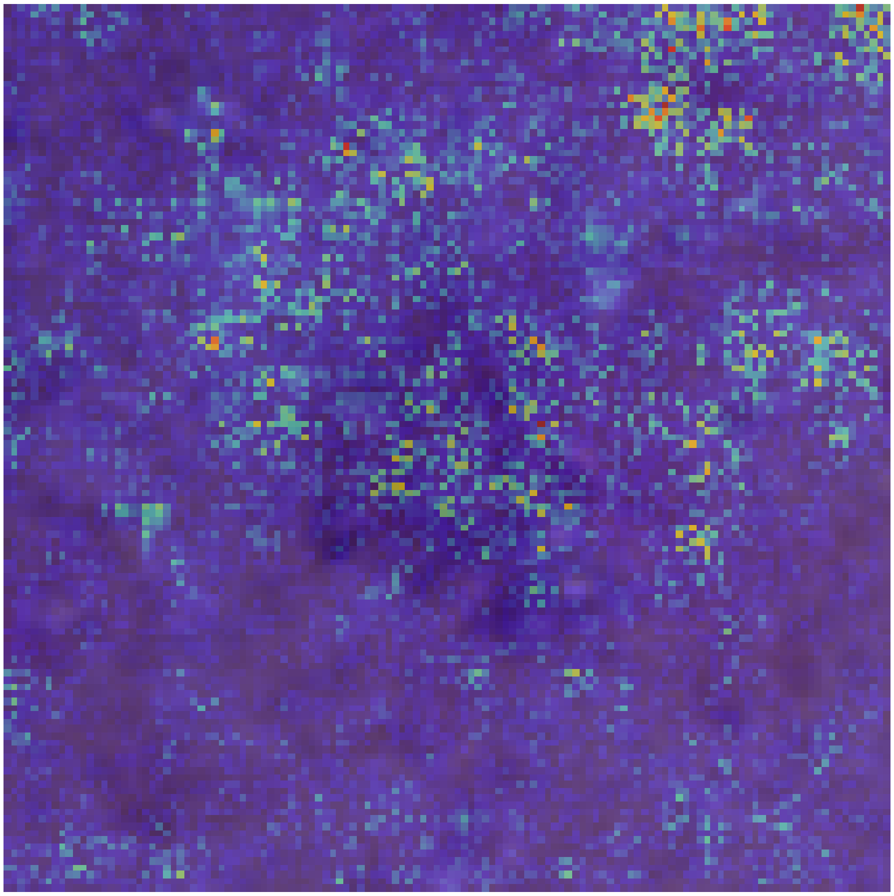
Misclassified: True Label: 1.0, Predicted Prob: 0.4495



Misclassified: True Label: 1.0, Predicted Prob: 0.0037



Misclassified: True Label: 1.0, Predicted Prob: 0.0920

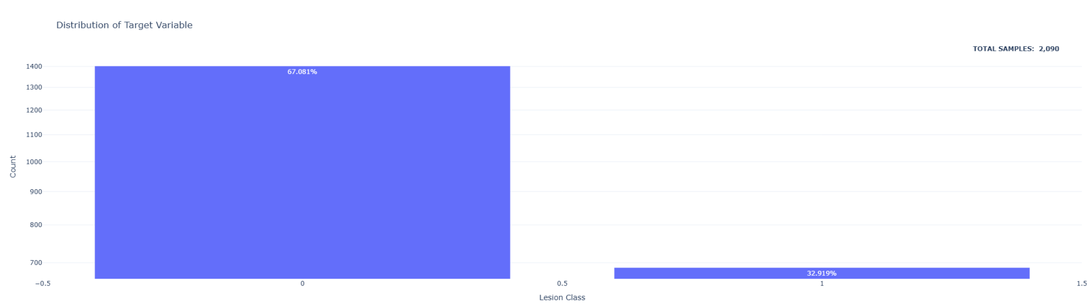


0.7 Analysis and quantification of bias

[6]:

```
train_data = pd.read_csv('../data/processed/processed-train-metadata1.csv')
# Target Distribution

# Count the occurrences of each target value and sort by index
target_counts = train_data['target'].value_counts().sort_index()

# Calculate the total number of samples in the training DataFrame
total = len(train_data)

# Create a list of percentages for each target class, formatted as a string
percentage = [f'{count/total:0.3%}' for count in target_counts]

# Create a bar plot to visualize the distribution of the target variable
fig = go.Figure(data=[
    go.Bar(
```

```
        x=target_counts.index,  # X-axis represents the unique target classes
        y=target_counts.values,  # Y-axis represents the counts of each class
        text=percentage,  # Display percentages on top of the bars
        textposition='auto'  # Automatically position text on bars
    )
])

# Update layout of the plot with titles and formatting
fig.update_layout(
    title='Distribution of Target Variable',  # Main title of the plot
    xaxis_title='Lesion Class',  # Title for the X-axis
    yaxis_title='Count',  # Title for the Y-axis
    template='plotly_white',  # Use a white background for the plot
    height=600, width=1200  # Set the dimensions of the plot
)

# Set the y-axis to a logarithmic scale to better visualize class distributions
fig.update_layout(yaxis=dict(type='log'))

# Add an annotation to show the total number of samples in the dataset
fig.add_annotation(
    text=f"<b>TOTAL SAMPLES:  {total:,}</b>",  # Format total count with commas
    xref="paper", yref="paper",  # Reference the entire paper for positioning
    x=0.98, y=1.05,  # Position the annotation near the top-right corner
    showarrow=False,  # Do not show an arrow pointing to the text
    font=dict(size=12)  # Set the font size for the annotation
)

# Display the plot
fig.show()
```



I have resampled the data during the preprocessing phrase. Resampling the data multiple times is generally discouraged because it can lead to overfitting, especially when the model repeatedly sees artificially replicated or manipulated samples. Overfitting occurs when the model memorizes the patterns of the resampled data rather than learning generalizable features, leading to poor performance on unseen data. Instead, strategies like using robust loss functions, regularization techniques, or leveraging augmentation methods for the minority class can help address imbalances without introducing redundancy or bias from excessive resampling.

0.8 Retrain the model with cleaned-up input

I chose to remove features with minimal or negative importance to the model, specifically 'anatom_site_general_lower extremity', 'anatom_site_general_posterior torso', and 'anatom_site_general_upper extremity'. This decision simplifies the model by reducing irrelevant inputs, enhancing its interpretability, and minimizing the risk of overfitting. Additionally, I incorporated pos_weight into the BCEWithLogitsLoss function to address the class imbalance in the dataset, ensuring that the model places greater emphasis on correctly classifying malignant skin lesions, which are underrepresented. This adjustment is intended to improve metrics such as recall and F1-score for the minority class without significantly compromising overall model performance.

[30]:
```
# drop features that  provide negatives importance score
test_metadata = pd.read_csv('../data/processed/processed-test-metadata1.csv')
val_metadata = pd.read_csv('../data/processed/processed-validation-metadata1.
 ↪csv')

features_to_drop = ['anatom_site_general_lower extremity',␣
 ↪'anatom_site_general_posterior torso', 'anatom_site_general_upper extremity']

train_metadata = train_data.drop(columns=features_to_drop, axis=1)
val_metadata = val_metadata.drop(columns=features_to_drop, axis=1)
test_metadata = test_metadata.drop(columns=features_to_drop, axis=1)
```

[31]:
```
# Import necessary libraries

import numpy as np
import torch

# Example target variable
# Ensure the target column exists in the dataset
try:
    target = train_metadata["target"]
except KeyError as e:
    raise KeyError("The 'target' column is missing from the provided dataset.␣
 ↪Please check your data.") from e
except Exception as e:
    raise Exception(f"An unexpected error occurred while accessing the target␣
 ↪variable: {e}")

# Compute class weights with error handling
try:
    # Compute balanced class weights
    class_weights = compute_class_weight(
```

```
        class_weight='balanced',  # Use balanced weighting to account for class␣
↪imbalance
        classes=np.unique(target),  # Provide unique class labels
        y=target  # Target variable
    )
except ValueError as e:
    raise ValueError("Error in computing class weights. Ensure the target␣
↪variable has valid class labels.") from e
except Exception as e:
    raise Exception(f"An unexpected error occurred while computing class␣
↪weights: {e}")

# Create a dictionary to map class labels to their weights
try:
    class_weights_dict = {cls: weight for cls, weight in zip(np.unique(target),␣
↪class_weights)}
except Exception as e:
    raise Exception(f"Error in creating the class weights dictionary: {e}")

# Extract the positive class weight for use in BCEWithLogitsLoss
try:
    pos_weight = torch.tensor(class_weights[1], dtype=torch.float32)
except IndexError as e:
    raise IndexError("Error accessing the positive class weight. Ensure your␣
↪target variable has at least two classes.") from e
except Exception as e:
    raise Exception(f"An unexpected error occurred while setting the pos_weight␣
↪tensor: {e}")

# Print the results
try:
    print("Class Weights:", class_weights_dict)
    print("Positive Class Weight (pos_weight):", pos_weight.item())
except Exception as e:
    raise Exception(f"An unexpected error occurred while printing the results:␣
↪{e}")
```

```
Class Weights: {0: 0.7453637660485022, 1: 1.5188953488372092}
```

[32]:

```
# Training and validation loop function
def train_and_validate(
    model: nn.Module,
    train_dataloader: torch.utils.data.DataLoader,
    val_dataloader: torch.utils.data.DataLoader,
    criterion: nn.Module,
    optimizer: torch.optim.Optimizer,
    epochs: int,
```

```
    device: torch.device,
    best_model_path: str,
    early_stopping_patience: int = 5,
    min_tpr: float = 0.80

) -> nn.Module:
    """
    Train and validate a PyTorch model with early stopping, AUROC, partial AUC,␣
↪and error handling.

    Args:
        model (nn.Module): The model to be trained and validated.
        train_dataloader (torch.utils.data.DataLoader): Dataloader for training␣
↪data.
        val_dataloader (torch.utils.data.DataLoader): Dataloader for validation␣
↪data.
        criterion (nn.Module): Loss function.
        optimizer (torch.optim.Optimizer): Optimizer to update the model.
        epochs (int): Number of training epochs.
        device (torch.device): The device (CPU or GPU) to use.
        early_stopping_patience (int): Early stopping patience.
        min_tpr (float): The minimum true positive rate for calculating partial␣
↪AUC.

    Returns:
        nn.Module: The trained model.
    """
    # Initialize tracking variables
    best_val_loss = float('inf')
    best_epoch = 0
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []
    early_stopping_counter = 0

    # Start the training and validation loop
    for epoch in range(epochs):
        print(f'Epoch {epoch + 1}/{epochs}')

        # Training phase
        model.train()
        running_train_loss = 0.0
        correct_train = 0
        total_train = 0
        all_train_labels = []
        all_train_probs = []
```

```python
        progress_bar = tqdm(train_dataloader, desc=f'Training Epoch {epoch + 1}')

        try:
            # Loop through the training batches
            for i, (image, metadata, labels) in enumerate(progress_bar):
                image, metadata, labels = image.to(device), metadata.to(device), labels.float().to(device)
                labels = labels.unsqueeze(1)  # Adjust labels to have the right shape for binary classification

                optimizer.zero_grad()

                # Forward pass
                probs = model(image, metadata)

                if probs.shape != labels.shape:
                    raise ValueError(f"Shape mismatch: Predictions shape {probs.shape} does not match labels shape {labels.shape}")

                # Calculate loss and backpropagate
                loss = criterion(probs, labels)
                loss.backward()
                optimizer.step()

                # Update running loss
                running_train_loss += loss.item()

                # Store labels and predictions for accuracy calculations
                all_train_labels.extend(labels.cpu().detach().numpy())
                all_train_probs.extend(probs.cpu().detach().numpy())

                # Calculate binary predictions for training accuracy
                predicted_train = (probs >= 0.5).float()
                total_train += labels.size(0)
                correct_train += (predicted_train == labels).sum().item()

                # Update progress bar
                progress_bar.set_postfix(train_loss=running_train_loss / (i + 1))

            # Calculate training accuracy and loss
            train_accuracy = 100 * correct_train / total_train
            train_losses.append(running_train_loss / len(train_dataloader))
            train_accuracies.append(train_accuracy)
```

```
        except ValueError as ve:
            print(f"Error during training loop: {ve}")
            break

    # Validation phase
    model.eval()
    running_val_loss = 0.0
    correct = 0
    total = 0
    all_labels = []
    all_probs = []

    progress_bar = tqdm(val_dataloader, desc=f'Validating Epoch {epoch +␣
↪1}')

    with torch.no_grad():
        try:
            # Loop through the validation batches
            for i, (images, metadata, labels) in enumerate(progress_bar):
                images, metadata, labels = images.to(device), metadata.
↪to(device), labels.float().to(device)
                labels = labels.unsqueeze(1)

                probs = model(images, metadata)

                loss = criterion(probs, labels)
                running_val_loss += loss.item()

                all_labels.extend(labels.cpu().detach().numpy())
                all_probs.extend(probs.cpu().detach().numpy())

                # Calculate binary predictions for validation accuracy
                predicted = (probs >= 0.5).float()
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

                progress_bar.set_postfix(val_loss=running_val_loss / (i +␣
↪1))

            val_accuracy = 100 * correct / total
            val_loss = running_val_loss / len(val_dataloader)
            val_accuracies.append(val_accuracy)
            val_losses.append(val_loss)

            # Calculate AUROC
            try:
                valid_auroc = roc_auc_score(all_labels, all_probs)
```

```
                except ValueError as ve:
                    print(f"AUROC Calculation Error: {ve}")
                    valid_auroc = 0.0

                # Calculate partial AUC-above-TPR
                try:
                    partial_auroc = score(np.array(all_labels), np.
↪array(all_probs), min_tpr=min_tpr)
                except ValueError as ve:
                    print(f"Partial AUC Calculation Error: {ve}")
                    partial_auroc = 0.0

                print(f'Epoch [{epoch}/{epochs}], Train Loss: {train_losses[-1]:
↪.4f}, Val Loss: {val_loss:.4f}, '
                      f'Val Accuracy: {val_accuracy:.2f}%, Val AUROC:␣
↪{valid_auroc:.4f}, Partial AUROC: {partial_auroc:.4f}')

                # Early stopping based on validation loss
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    best_epoch = epoch + 1
                    early_stopping_counter = 0
                    torch.save(model.state_dict(), best_model_path)
                else:
                    early_stopping_counter += 1

                if early_stopping_counter >= early_stopping_patience:
                    print(f"Early stopping triggered at epoch {epoch}")
                    break

            except Exception as e:
                print(f"Error during validation loop: {e}")
                break

    print(f"Best Epoch: {best_epoch}, Best Validation Loss: {best_val_loss:.
↪4f}")
    print('Training Complete')

    # Plot training and validation loss
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()
    plt.show()
```

```
    # Plot training and validation accuracy
    plt.figure(figsize=(10, 5))
    plt.plot(train_accuracies, label='Train Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Training and Validation Accuracy')
    plt.legend()
    plt.show()

    # Generate classification report
    try:
        print("Classification Report:")
        print(classification_report(all_labels, (np.array(all_probs) >= 0.5).
 ↪astype(int), target_names=['Class 0', 'Class 1']))
    except Exception as e:
        print(f"Error generating classification report: {e}")

    return model
```

[35]:
```
# Initialize the dataset
effnet_train_dataset = MultiInputDataset(hdf5_file='../data/raw/train_images.
 ↪hdf5', df=train_metadata,␣
 ↪transform=get_train_transform(resize_size=(224,224)))
effnet_val_dataset = MultiInputDataset(hdf5_file='../data/raw/validation_image.
 ↪hdf5', df=val_metadata,␣
 ↪transform=get_normal_transform(resize_size=(224,224)))
# Create a DataLoader
effnet_train_dataloader = DataLoader(effnet_train_dataset,  batch_size=16,␣
 ↪shuffle=True)
effnet_val_dataloader = DataLoader(effnet_val_dataset,  batch_size=16,␣
 ↪shuffle=True)
```

[39]:
```
model =  CustomImageFeatureEfficientNet(feature_input_size=6)  # Assuming 9␣
 ↪features for metadata
model.to(device)
# Initialize optimizer
optimizer = optim.Adam(model.parameters(), lr= 1.1621608010269284e-05)
# Define the loss function with the class weights
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight.to(device))
# Set the number of epochs
epochs = 20
batch_size = 16
best_model_path = "best_model11.pth"
```

/home/jupyter-sohka/.local/lib/python3.10/site-

```
packages/torchvision/models/_utils.py:208: UserWarning:

The parameter 'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.

/home/jupyter-sohka/.local/lib/python3.10/site-
packages/torchvision/models/_utils.py:223: UserWarning:

Arguments other than a weight enum or `None` for 'weights' are deprecated since
0.13 and may be removed in the future. The current behavior is equivalent to
passing `weights=EfficientNet_B0_Weights.IMAGENET1K_V1`. You can also use
`weights=EfficientNet_B0_Weights.DEFAULT` to get the most up-to-date weights.
```

0.9 Train and Validation

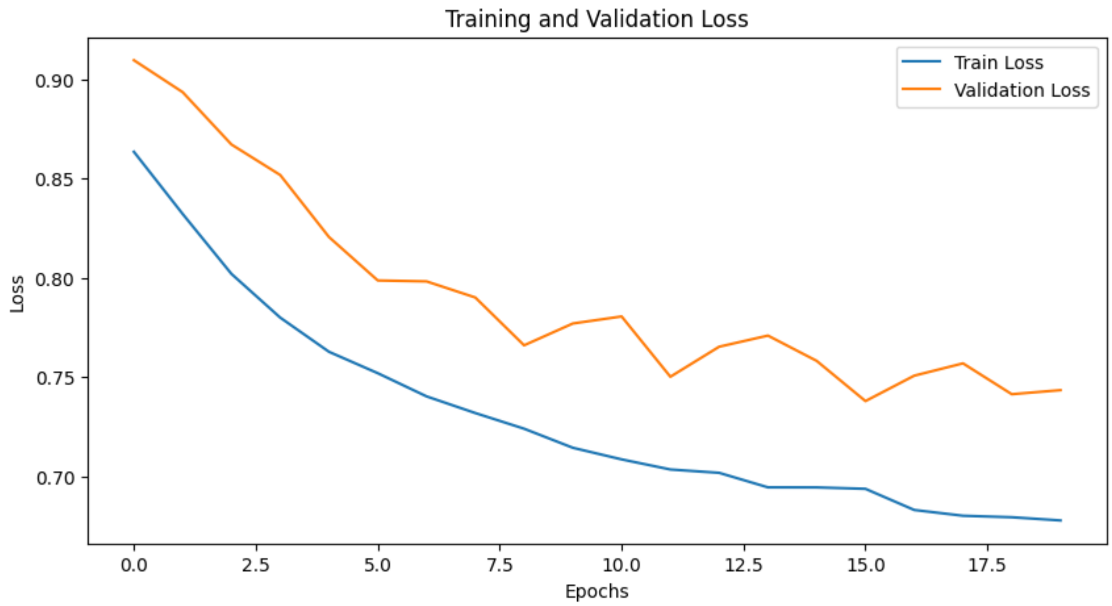
[40]:
```
train_and_validate(model,effnet_train_dataloader, effnet_val_dataloader,
 ↪criterion, optimizer, epochs, device, best_model_path )
```

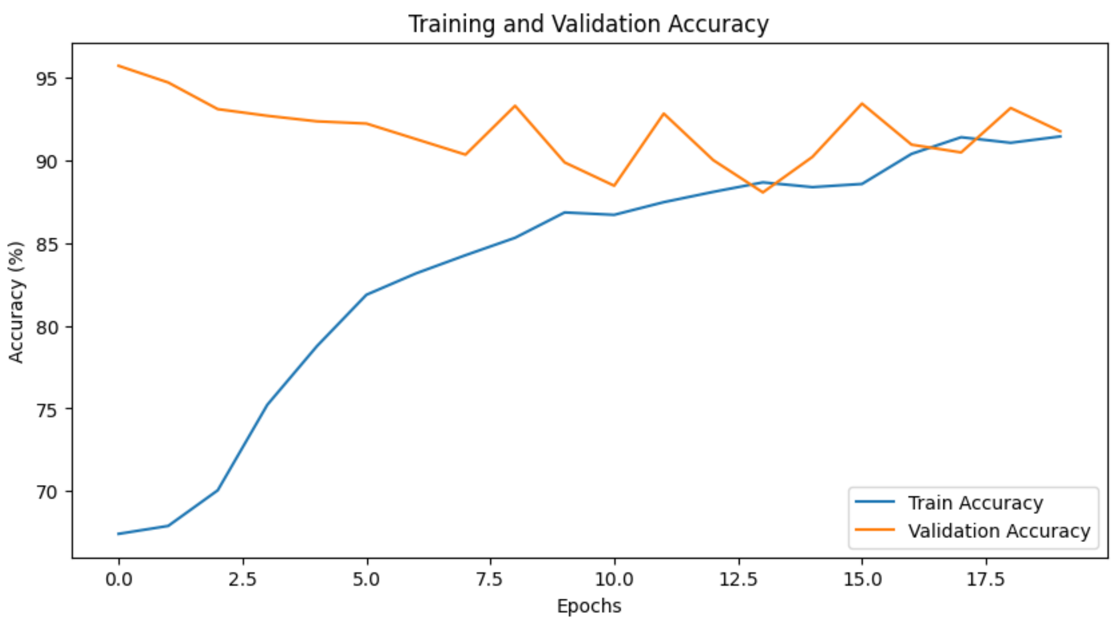
```
Epoch 1/20

Training Epoch 1: 100%|      | 131/131 [01:38<00:00,  1.34it/s,
train_loss=0.863]
Validating Epoch 1: 100%|      | 94/94 [00:44<00:00,  2.10it/s,
val_loss=0.909]

Epoch [0/20], Train Loss: 0.8634, Val Loss: 0.9094, Val Accuracy: 95.70%, Val
AUROC: 0.5913, Partial AUROC: 0.0361
Epoch 2/20

Training Epoch 2: 100%|      | 131/131 [01:45<00:00,  1.25it/s,
train_loss=0.832]
Validating Epoch 2: 100%|      | 94/94 [00:44<00:00,  2.10it/s,
val_loss=0.893]

Epoch [1/20], Train Loss: 0.8321, Val Loss: 0.8934, Val Accuracy: 94.70%, Val
AUROC: 0.6835, Partial AUROC: 0.0641
Epoch 3/20

Training Epoch 3: 100%|      | 131/131 [01:42<00:00,  1.28it/s,
train_loss=0.802]
Validating Epoch 3: 100%|      | 94/94 [00:43<00:00,  2.14it/s,
val_loss=0.867]

Epoch [2/20], Train Loss: 0.8019, Val Loss: 0.8670, Val Accuracy: 93.09%, Val
AUROC: 0.7489, Partial AUROC: 0.0797
Epoch 4/20

Training Epoch 4: 100%|      | 131/131 [01:40<00:00,  1.30it/s,
train_loss=0.78]
```

```
Validating Epoch 4: 100%|     | 94/94 [00:58<00:00,  1.60it/s,
val_loss=0.852]

Epoch [3/20], Train Loss: 0.7799, Val Loss: 0.8517, Val Accuracy: 92.68%, Val
AUROC: 0.7899, Partial AUROC: 0.0886
Epoch 5/20

Training Epoch 5: 100%|     | 131/131 [01:38<00:00,  1.33it/s,
train_loss=0.763]
Validating Epoch 5: 100%|     | 94/94 [00:45<00:00,  2.09it/s,
val_loss=0.82]

Epoch [4/20], Train Loss: 0.7627, Val Loss: 0.8205, Val Accuracy: 92.35%, Val
AUROC: 0.8081, Partial AUROC: 0.0879
Epoch 6/20

Training Epoch 6: 100%|     | 131/131 [01:47<00:00,  1.22it/s,
train_loss=0.752]
Validating Epoch 6: 100%|     | 94/94 [00:44<00:00,  2.13it/s,
val_loss=0.799]

Epoch [5/20], Train Loss: 0.7519, Val Loss: 0.7986, Val Accuracy: 92.21%, Val
AUROC: 0.8272, Partial AUROC: 0.0936
Epoch 7/20

Training Epoch 7: 100%|     | 131/131 [01:37<00:00,  1.35it/s,
train_loss=0.74]
Validating Epoch 7: 100%|     | 94/94 [00:42<00:00,  2.19it/s,
val_loss=0.798]

Epoch [6/20], Train Loss: 0.7403, Val Loss: 0.7981, Val Accuracy: 91.28%, Val
AUROC: 0.8272, Partial AUROC: 0.0910
Epoch 8/20

Training Epoch 8: 100%|     | 131/131 [01:38<00:00,  1.33it/s,
train_loss=0.732]
Validating Epoch 8: 100%|     | 94/94 [00:43<00:00,  2.17it/s,
val_loss=0.79]

Epoch [7/20], Train Loss: 0.7319, Val Loss: 0.7900, Val Accuracy: 90.34%, Val
AUROC: 0.8504, Partial AUROC: 0.1000
Epoch 9/20

Training Epoch 9: 100%|     | 131/131 [01:41<00:00,  1.29it/s,
train_loss=0.724]
Validating Epoch 9: 100%|     | 94/94 [00:46<00:00,  2.04it/s,
val_loss=0.766]

Epoch [8/20], Train Loss: 0.7240, Val Loss: 0.7660, Val Accuracy: 93.29%, Val
AUROC: 0.8475, Partial AUROC: 0.0909
Epoch 10/20

Training Epoch 10: 100%|     | 131/131 [01:45<00:00,  1.24it/s,
train_loss=0.714]
```

```
Validating Epoch 10: 100%|     | 94/94 [00:45<00:00,  2.07it/s,
val_loss=0.777]

Epoch [9/20], Train Loss: 0.7144, Val Loss: 0.7770, Val Accuracy: 89.87%, Val
AUROC: 0.8553, Partial AUROC: 0.0938
Epoch 11/20

Training Epoch 11: 100%|     | 131/131 [01:40<00:00,  1.30it/s,
train_loss=0.709]
Validating Epoch 11: 100%|     | 94/94 [00:44<00:00,  2.11it/s,
val_loss=0.78]

Epoch [10/20], Train Loss: 0.7086, Val Loss: 0.7805, Val Accuracy: 88.46%, Val
AUROC: 0.8634, Partial AUROC: 0.1049
Epoch 12/20

Training Epoch 12: 100%|     | 131/131 [01:39<00:00,  1.32it/s,
train_loss=0.703]
Validating Epoch 12: 100%|     | 94/94 [00:43<00:00,  2.16it/s,
val_loss=0.75]

Epoch [11/20], Train Loss: 0.7035, Val Loss: 0.7501, Val Accuracy: 92.82%, Val
AUROC: 0.8563, Partial AUROC: 0.0993
Epoch 13/20

Training Epoch 13: 100%|     | 131/131 [01:38<00:00,  1.32it/s,
train_loss=0.702]
Validating Epoch 13: 100%|     | 94/94 [00:58<00:00,  1.62it/s,
val_loss=0.765]

Epoch [12/20], Train Loss: 0.7018, Val Loss: 0.7653, Val Accuracy: 90.00%, Val
AUROC: 0.8712, Partial AUROC: 0.1071
Epoch 14/20

Training Epoch 14: 100%|     | 131/131 [01:45<00:00,  1.25it/s,
train_loss=0.694]
Validating Epoch 14: 100%|     | 94/94 [00:46<00:00,  2.02it/s,
val_loss=0.771]

Epoch [13/20], Train Loss: 0.6945, Val Loss: 0.7709, Val Accuracy: 88.05%, Val
AUROC: 0.8819, Partial AUROC: 0.1194
Epoch 15/20

Training Epoch 15: 100%|     | 131/131 [01:40<00:00,  1.31it/s,
train_loss=0.694]
Validating Epoch 15: 100%|     | 94/94 [00:45<00:00,  2.05it/s,
val_loss=0.758]

Epoch [14/20], Train Loss: 0.6945, Val Loss: 0.7581, Val Accuracy: 90.20%, Val
AUROC: 0.8765, Partial AUROC: 0.1126
Epoch 16/20

Training Epoch 16: 100%|     | 131/131 [01:41<00:00,  1.30it/s,
train_loss=0.694]
```

```
Validating Epoch 16: 100%|     | 94/94 [00:44<00:00,  2.13it/s,
val_loss=0.738]

Epoch [15/20], Train Loss: 0.6937, Val Loss: 0.7379, Val Accuracy: 93.42%, Val
AUROC: 0.8522, Partial AUROC: 0.0953
Epoch 17/20

Training Epoch 17: 100%|     | 131/131 [01:40<00:00,  1.30it/s,
train_loss=0.683]
Validating Epoch 17: 100%|     | 94/94 [00:44<00:00,  2.13it/s,
val_loss=0.751]

Epoch [16/20], Train Loss: 0.6831, Val Loss: 0.7507, Val Accuracy: 90.94%, Val
AUROC: 0.8746, Partial AUROC: 0.1103
Epoch 18/20

Training Epoch 18: 100%|     | 131/131 [01:40<00:00,  1.30it/s,
train_loss=0.68]
Validating Epoch 18: 100%|     | 94/94 [00:53<00:00,  1.75it/s,
val_loss=0.757]

Epoch [17/20], Train Loss: 0.6802, Val Loss: 0.7569, Val Accuracy: 90.47%, Val
AUROC: 0.8807, Partial AUROC: 0.1155
Epoch 19/20

Training Epoch 19: 100%|     | 131/131 [01:41<00:00,  1.29it/s,
train_loss=0.679]
Validating Epoch 19: 100%|     | 94/94 [00:45<00:00,  2.08it/s,
val_loss=0.741]

Epoch [18/20], Train Loss: 0.6794, Val Loss: 0.7414, Val Accuracy: 93.15%, Val
AUROC: 0.8604, Partial AUROC: 0.1017
Epoch 20/20

Training Epoch 20: 100%|     | 131/131 [01:40<00:00,  1.30it/s,
train_loss=0.678]
Validating Epoch 20: 100%|     | 94/94 [00:43<00:00,  2.15it/s,
val_loss=0.743]

Epoch [19/20], Train Loss: 0.6778, Val Loss: 0.7434, Val Accuracy: 91.74%, Val
AUROC: 0.8552, Partial AUROC: 0.0993
Best Epoch: 16, Best Validation Loss: 0.7379
Training Complete
```





Classification Report:

	precision	recall	f1-score	support
Class 0	0.98	0.93	0.96	1431
Class 1	0.25	0.56	0.35	59
accuracy			0.92	1490

```
   macro avg       0.62      0.75      0.65      1490
weighted avg       0.95      0.92      0.93      1490
```

[40]:
```
CustomImageFeatureEfficientNet(
  (efficientnet): Sequential(
    (0): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): SiLU(inplace=True)
      )
      (1): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=32, bias=False)
              (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(32, 8, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(8, 32, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (2): Conv2dNormActivation(
              (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0, mode=row)
        )
      )
      (2): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), groups=96, bias=False)
              (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(96, 4, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(4, 96, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.0125, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=144, bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
```

```
            (3): Conv2dNormActivation(
              (0): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.025, mode=row)
        )
      )
      (3): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(144, 144, kernel_size=(5, 5), stride=(2, 2),
padding=(2, 2), groups=144, bias=False)
              (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(144, 40, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.037500000000000006, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1),
```

```
bias=False)
              (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(240, 240, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=240, bias=False)
              (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(240, 40, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.05, mode=row)
        )
      )
      (4): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(240, 240, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), groups=240, bias=False)
              (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
```

```
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(240, 80, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.0625, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=480, bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.07500000000000001, mode=row)
      )
```

```
        (2): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=480, bias=False)
              (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.08750000000000001, mode=row)
        )
      )
      (5): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(480, 480, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=480, bias=False)
              (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(480, 112, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=672, bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1125, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=672, bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.125, mode=row)
      )
    )
    (6): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
```

```
            (1): Conv2dNormActivation(
              (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(2, 2),
padding=(2, 2), groups=672, bias=False)
              (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1375, mode=row)
        )
        (1): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
```

```
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.15000000000000002, mode=row)
        )
        (2): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1625, mode=row)
        )
        (3): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.17500000000000002, mode=row)
        )
      )
      (7): Sequential(
        (0): MBConv(
          (block): Sequential(
            (0): Conv2dNormActivation(
              (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (1): Conv2dNormActivation(
              (0): Conv2d(1152, 1152, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1152, bias=False)
              (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): SiLU(inplace=True)
            )
            (2): SqueezeExcitation(
              (avgpool): AdaptiveAvgPool2d(output_size=1)
              (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
              (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
```

```
              (activation): SiLU(inplace=True)
              (scale_activation): Sigmoid()
            )
            (3): Conv2dNormActivation(
              (0): Conv2d(1152, 320, kernel_size=(1, 1), stride=(1, 1),
bias=False)
              (1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            )
          )
          (stochastic_depth): StochasticDepth(p=0.1875, mode=row)
        )
      )
      (8): Conv2dNormActivation(
        (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): SiLU(inplace=True)
      )
    )
    (1): AdaptiveAvgPool2d(output_size=1)
  )
  (fc_image): Linear(in_features=1280, out_features=512, bias=True)
  (fc_metadata): Linear(in_features=6, out_features=128, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc_combined): Linear(in_features=640, out_features=1, bias=True)
)
```

0.10 Test Data Performance

[41]:
```
#create test dataset
effnet_test_dataset = MultiInputDataset(hdf5_file='../data/raw/test_image.
 ↪hdf5', df=test_metadata,␣
 ↪transform=get_normal_transform(resize_size=(128,128)))
# Create test DataLoader
effnet_test_dataloader = DataLoader(effnet_test_dataset,  batch_size=64,␣
 ↪shuffle=True)
```

[43]:
```
# Set the model to evaluation mode, which disables dropout and batch␣
 ↪normalization updates
model.eval()

# Initialize lists to store labels and predicted probabilities for later␣
 ↪analysis
all_labels, all_probs = [], []
```

```python
# Disable gradient computation for testing phase to save memory and improve␣
↪performance
with torch.no_grad():
    # Loop through batches in the test dataloader
    for images, metadata, labels in effnet_test_dataloader:
        # Move data to the specified device (e.g., CPU or GPU) and adjust label␣
↪shape
        images, metadata = images.to(device), metadata.to(device)
        labels = labels.float().to(device).unsqueeze(1)

        # Forward pass to get probabilities from the model
        probs = model(images, metadata)

        # Collect labels and predicted probabilities, converting to numpy arrays
        all_labels.extend(labels.cpu().numpy())
        all_probs.extend(probs.cpu().numpy())

        # Generate binary predictions based on a 0.5 threshold
        predicted = (probs > 0.5).float()

    # Calculate the partial AUROC score (adjusted for your specific function)
    partial_auroc = score(np.array(all_labels), np.array(all_probs))
    print(f'The partial AUROC of the final model on the test images is␣
↪{partial_auroc}')

    # Print the classification report, evaluating performance on Class 0 and␣
↪Class 1
    print(classification_report(all_labels, (np.array(all_probs) >= 0.5).
↪astype(int), target_names=['Class 0', 'Class 1']))
```

```
The partial AUROC of the final model on the test images is 0.1272880171505051
              precision    recall  f1-score   support

     Class 0       0.98      0.95      0.96      1431
     Class 1       0.30      0.58      0.40        59

    accuracy                           0.93      1490
   macro avg       0.64      0.76      0.68      1490
weighted avg       0.95      0.93      0.94      1490
```