

## Documentation of the Three Architectures

### 1) ResNet Implementation from scratch:

#### What is ResNet?

**ResNet (Residual Network)** was introduced in the paper "*Deep Residual Learning for Image Recognition*" by Kaiming He et al., 2015 ([Link](#)).

- **Key Idea:** Uses residual connections (**shortcuts**) to allow information to skip layers, mitigating the **vanishing gradient problem** in deep networks.
- **Why Residuals Work:** Instead of learning a full mapping, residual blocks let the model learn the **difference (residual) between input and output**, simplifying optimization.

#### Key Features of ResNet:

- **Residual Connections:** ResNet introduces **shortcut connections to bypass one or more layers**, solving the **vanishing gradient problem**.
- **Bottleneck Blocks:** Efficiently reduce computation using **1x1** convolutions.
- **Scalability:** Enables the training of networks with **hundreds** or **thousands** of layers (e.g., ResNet-50, ResNet-101).

#### Advantages of ResNet:

1. **Deeper Networks:** Residual connections enable the **effective training** of deep networks without performance degradation.
2. **Gradient Flow:** Shortcut connections help gradients flow **backward without vanishing**.
3. **Feature Reuse:** Reuses features from **earlier layers**, improving efficiency and accuracy.

## Step-by-Step Code Explanation:

### 1)Dataset Preparation

```
import os
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from sklearn.preprocessing import label_binarize
import torchvision.transforms as transforms
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Dataset
from torch.utils.data.sampler import SubsetRandomSampler
import torch.optim as optim
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, roc_curve, auc
from torch.optim.lr_scheduler import StepLR
from tqdm import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
```

- **Purpose:** Import necessary libraries.
  - **torch:** PyTorch for defining and training the model.
  - **torchvision:** Utilities for image datasets and transformations.
  - **sklearn:** Used for evaluation metrics (e.g., accuracy, F1-score).
  - **matplotlib/seaborn:** For visualizations (confusion matrices, ROC).

## 2)Preprocessing and Data Augmentation

### Step 1: Compute Dataset Statistics

```
# Step 1: Define the transform for calculating mean and std
first_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Step 2: Load the training dataset for normalization
train_dataset = datasets.ImageFolder(root=train_dir, transform=first_transform)

# Step 3: Calculate mean and std
def calculate_mean_std(dataloader):
    mean = 0.0
    std = 0.0
    total_imgs = 0
    for images, _ in dataloader:
        batch_mean = torch.mean(images, dim=[0, 2, 3])
        batch_std = torch.std(images, dim=[0, 2, 3])
        batch_size = images.size(0)
        mean += batch_mean * batch_size
        std += batch_std * batch_size
        total_imgs += batch_size
    mean /= total_imgs
    std /= total_imgs
    return mean, std

dataloader = DataLoader(train_dataset, batch_size=64, shuffle=False)
mean, std = calculate_mean_std(dataloader)
print(f"Dataset Mean: {mean}, Dataset Std: {std}")
```

#### Explanation:

1. **Initial Transformation:**
  - **Resizes** all images to **224x224**.
  - Converts images into **tensors**.
2. **Calculate Mean/Std:** Used for **normalizing** pixel values during training.
  - **Mean:** Average pixel value across dataset.
  - **Std:** Measures pixel value variation.

3. **Why 224x224?** This size is widely used in pretrained models (e.g., ImageNet) and balances computational efficiency with feature extraction. Smaller images might **lose detail**, while larger images require **more computation**.

## Step 2: Final Transformations

```
train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std),
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std),
])
```

- **Augmentation (Training Only):**
  - **RandomResizedCrop**: Randomly crops images for added diversity.
  - **RandomHorizontalFlip**: Flips images horizontally to improve robustness.
- **Normalization:**
  - Centers pixel values around 0 for **faster convergence**.

## Step 3: Data Loading

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

**DataLoader**: Efficiently batches data for training and evaluation.

- **shuffle=True** ensures randomness for training.

## ResNet Architecture

### 3) Bottleneck Block

#### Why Use Bottleneck Blocks?

- **Efficiency:** A bottleneck block reduces computation by compressing the number of channels using 1x1 convolutions.
- **Structure:**
  - First, reduce dimensions using **1x1 convolutions**.
  - Then apply **3x3 convolutions** for **feature extraction**.
  - Finally, expand dimensions back using another **1x1 convolution**.

In [5]:

```
class BottleneckBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BottleneckBlock, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels * 4, kernel_size=1, stride=1, bias=False),
            nn.BatchNorm2d(out_channels * 4)
        )
        self.downsample = downsample
        self.relu = nn.ReLU()
```

#### Explanation:

1. **Conv1 (1x1 Convolution):**
  - **Reduces** the number of **channels**, making subsequent computations more efficient.
2. **Conv2 (3x3 Convolution):**

- Extracts spatial features. The stride may be **2 for downsampling**.
  - Padding ensures that **spatial dimensions** remain consistent.
3. **Conv3 (1x1 Convolution):**
- **Expands the number of channels** to match the input for residual addition.
4. **Downsample:**
- Ensures that the dimensions of the **input match the output** if they differ (e.g., due to stride).

## 4)Residual Connection

```
def forward(self, x):
    residual = x
    out = self.conv1(x)
    out = self.conv2(out)
    out = self.conv3(out)
    if self.downsample:
        residual = self.downsample(x)
    out += residual
    out = self.relu(out)
    return out
```

### Key Insight:

- **Residual connections** allow gradients to flow directly through the identity path, mitigating the **vanishing gradient problem**.
- The network learns residual **mappings (differences) instead of full transformations**, simplifying optimization.

## 5)ResNet Class

In [6]:

```
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=5749):
        super(ResNet, self).__init__()
        self.inplanes = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer0 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer1 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer2 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer3 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)
```

### Explanation:

#### 1. Conv1:

- Applies a large kernel (7x7) convolution to **extract initial features**.
- Stride (step) of 2 reduces **spatial dimensions by half**.

#### 2. MaxPool:

- Further reduces **spatial dimensions**, retaining key features.

#### 3. Residual Layers:

- **\_make\_layer** stacks multiple bottleneck blocks, each focusing on feature extraction at different scales.

#### 4. Global Pooling:

- **AdaptiveAvgPool2d((1, 1))** converts feature maps to a fixed size, regardless of input dimensions.

#### 5. Fully Connected Layer:

- Maps the extracted features to **class probabilities**.

## 6) Training Loop

```
# Configuration
batch_size = 64
epochs = 20
lr = 0.001
weight_decay = 1e-4
step_size = 10
gamma = 0.1

# Model setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

```
# Loss, optimizer, scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_decay)
scheduler = StepLR(optimizer, step_size=step_size, gamma=gamma)

# Training and Validation
for epoch in range(epochs):
    # Training Phase
    model.train()
    train_loss = 0.0
    train_preds, train_labels = [], []

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Record loss and predictions
    train_loss += loss.item()
    _, preds = torch.max(outputs, 1)
    train_preds.extend(preds.cpu().numpy())
    train_labels.extend(labels.cpu().numpy())
```



```

# Scheduler step
scheduler.step()

train_accuracy = accuracy_score(train_labels, train_preds)

# Validation Phase
model.eval()
val_loss = 0.0
val_preds, val_labels = [], []

with torch.no_grad():
    for images, labels in val_loader:
        images, labels = images.to(device), labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        val_loss += loss.item()
        _, preds = torch.max(outputs, 1)
        val_preds.extend(preds.cpu().numpy())
        val_labels.extend(labels.cpu().numpy())

val_accuracy = accuracy_score(val_labels, val_preds)

print(f"Epoch [{epoch+1}/{epochs}], "
      f"Train Loss: {train_loss/len(train_loader):.4f}, Train Accuracy: {train_accuracy*100:.2f}%, "
      f"Val Loss: {val_loss/len(val_loader):.4f}, Val Accuracy: {val_accuracy*100:.2f}%")

```

## Explanation:

1. Loss Calculation:
  - `CrossEntropyLoss` computes the difference between predicted and true labels.
2. Backpropagation:
  - `optimizer.zero_grad()` clears gradients to avoid accumulation.
  - `loss.backward()` computes gradients.
  - `optimizer.step()` updates weights.
3. Learning Rate Scheduler:
  - Reduces the learning rate after every 10 epochs to improve convergence (`gamma=0.1`)..
4. Evaluation:
  - `model.eval()` disables dropout and batch normalization.
  - Validation metrics ensure the model generalizes to unseen data.

## Results for ResNet

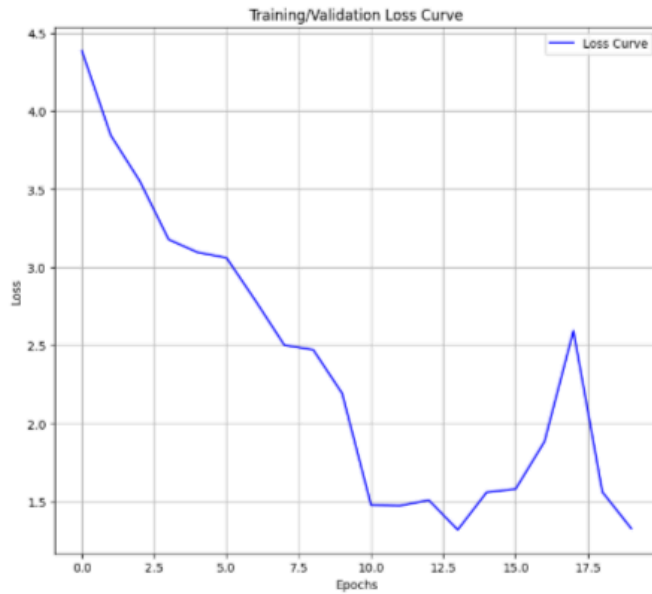
Accuracy: 73.63%

F1 Score: 0.7319

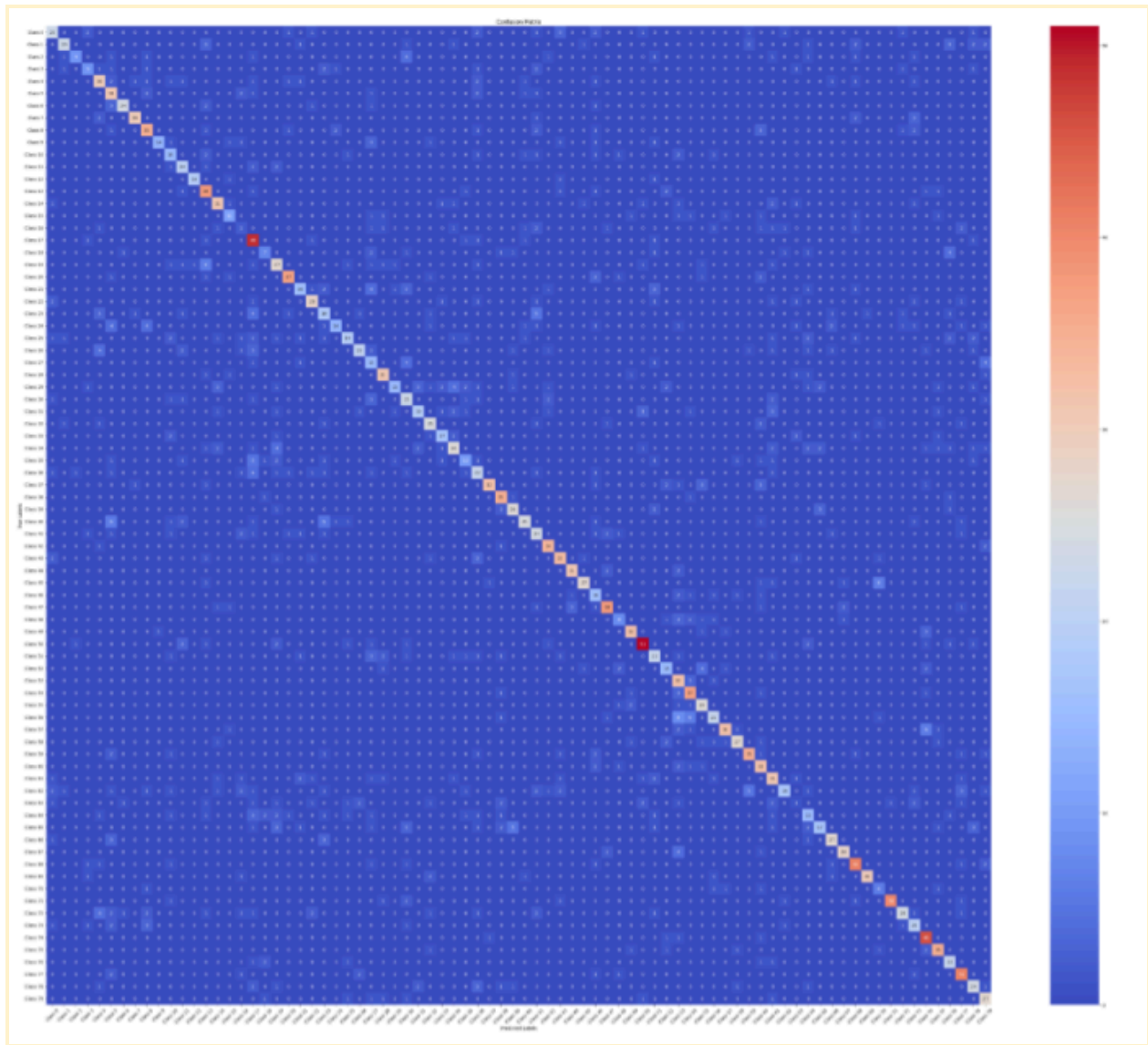
Precision: 0.7456

Recall: 0.7363

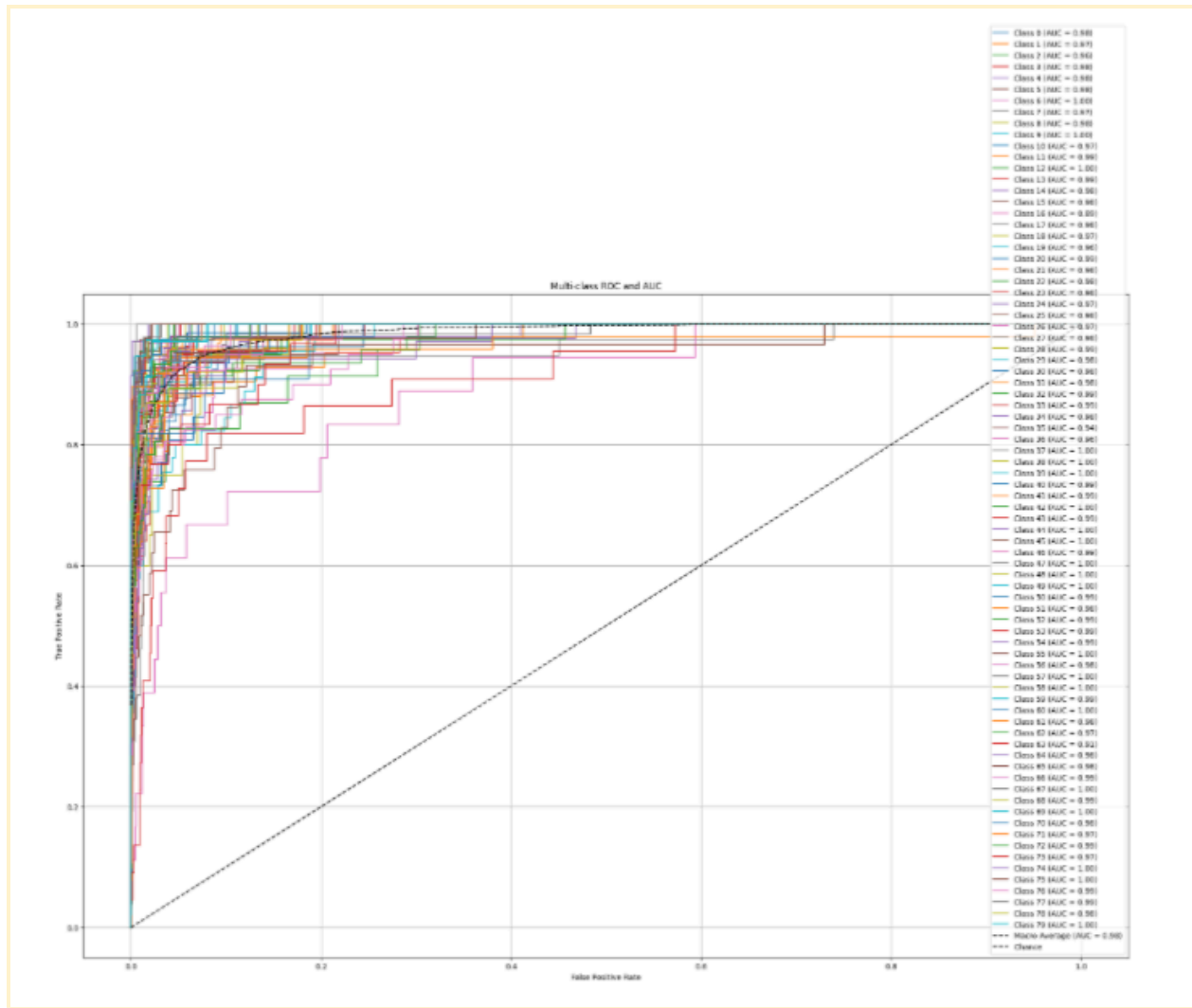
### loss curve with visualization



### confusion matrix with visualization



## ROC & AUC



## References and Papers

**Residual Networks (ResNet):** Kaiming He, et al., *Deep Residual Learning for Image Recognition*, CVPR 2016. [Paper Link](#)

**Bottleneck Architecture in ResNet:** Ioffe and Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. [Paper Link](#)

**Adam Optimizer:** Kingma and Ba, *Adam: A Method for Stochastic Optimization*. [Paper Link](#)

PyTorch Documentation ([Link](#)).

## 2) Xception Finetuning:

### What is Xception?

Xception (Extreme Inception) is a deep convolutional neural network architecture introduced by François Chollet in the paper:

- "Xception: Deep Learning with Depthwise Separable Convolutions" ([Paper Link](#)).

### Key Features of Xception:

- **Depthwise Separable Convolutions:** A **more efficient** way to learn spatial and **channel-wise features**.
- **Fully Convolutional Network:** Contains **no dense layers** in its base architecture.
- **Efficiency:** Demonstrates **better accuracy** and **fewer parameters compared to Inception models**.

## 1)Importing Libraries

```
In [1]: import tensorflow as tf
        from tensorflow.keras.applications import Xception
        from tensorflow.keras import layers, Model
        import os
        import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns
        from sklearn.metrics import roc_curve, auc, confusion_matrix, classification_report
        from itertools import cycle
        import math
        import pandas as pd
```

### Explanation:

- **TensorFlow and Keras** are used for **deep learning model development**.
- **Xception**: **Pretrained Xception** model for **transfer learning**.
- **layers and Model**: Utilities for building and customizing models.
- Libraries like **seaborn** and **matplotlib** are used for **visualization**, while **sklearn** handles **performance metrics**.

## 2) Parameters and Preprocessing

```
i]:  
# Set parameters  
IMG_HEIGHT = 299  
IMG_WIDTH = 299  
BATCH_SIZE = 32  
AUTO = tf.data.AUTOTUNE
```

### Explanation:

- **299x299**: Default input size for **Xception**.
- **BATCH\_SIZE**: Number of samples processed in each batch.
- **AUTO**: Optimizes data loading using **TensorFlow's pipeline**.

## 3) Image Preprocessing and Augmentation

```

# Read and preprocess image
def process_path(file_path, label):
    img = tf.io.read_file(file_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
    img = tf.cast(img, tf.float32)
    img = tf.keras.applications.xception.preprocess_input(img)
    return img, label

# Data augmentation
def augment(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)

    # Random brightness, saturation, and contrast
    image = tf.image.random_brightness(image, 0.2)
    image = tf.image.random_saturation(image, 0.5, 2.0)
    image = tf.image.random_contrast(image, 0.5, 2.0)

    # Random crop and resize
    image = tf.image.random_crop(image, [IMG_HEIGHT-30, IMG_WIDTH-30, 3])
    image = tf.image.resize(image, [IMG_HEIGHT, IMG_WIDTH])

    return image, label

```

### Explanation:

- **process\_path**: Standardizes images to **299x299** and applies **Xception**-specific preprocessing.
- **augment**: **Randomly alters** images to improve model generalization.

## 4)Create Dataset

```
# Create lists of paths and labels
image_paths = []
labels = []

for class_name in class_names:
    class_path = os.path.join(directory, class_name)
    for img_name in os.listdir(class_path):
        if img_name.lower().endswith(('.png', '.jpg', '.jpeg')):
            img_path = os.path.join(class_path, img_name)
            image_paths.append(img_path)
            labels.append(class_dict[class_name])

# Create tf.data.Dataset
ds = tf.data.Dataset.from_tensor_slices((image_paths, labels))

# Shuffle if training
if is_training:
    ds = ds.shuffle(len(image_paths), reshuffle_each_iteration=True)

# Map preprocessing function
ds = ds.map(process_path, num_parallel_calls=AUTO)

# Apply augmentation if training
if is_training:
    ds = ds.map(augment, num_parallel_calls=AUTO)

# Batch and prefetch
ds = ds.batch(BATCH_SIZE)
ds = ds.prefetch(AUTO)

return ds, len(image_paths)
```

## Why?

- Efficiently loads, preprocesses, and augments images, reducing **bottlenecks** during training.

## 5)Model Definition



## Load Pretrained Xception

```
def create_model():  
    # Base model  
    base_model = Xception(weights=xception_weights, include_top=False, input_shape=(IMG_HEIGHT, IMG_WIDTH, 3))  
    base_model.trainable = False
```

Why?

- **include\_top=False**: Removes the classification head for fine-tuning.
- **trainable=False**: **Freezes** weights to preserve pretrained knowledge.

## Add Custom Layers

```
# Add custom layers  
x = layers.GlobalAveragePooling2D()(x)  
x = layers.BatchNormalization()(x)  
  
# Dense layers with regularization  
x = layers.Dense(1024, kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)  
x = layers.BatchNormalization()(x)  
x = layers.Activation('relu')(x)  
x = layers.Dropout(0.3)(x)  
  
x = layers.Dense(512, kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)  
x = layers.BatchNormalization()(x)  
x = layers.Activation('relu')(x)  
x = layers.Dropout(0.3)(x)  
  
outputs = layers.Dense(NUM_CLASSES, activation='softmax')(x)  
  
return Model(inputs, outputs)
```

Explanation:

- **GlobalAveragePooling2D**: Reduces spatial dimensions to a **vector**.
- **Dense layers**: Learn **task-specific features**.
- **Dropout**: Mitigates **overfitting**.

## 6) Training the Model

In [10]:

```
# Create model
model = create_model()

# Compile model
initial_learning_rate = 1e-3
optimizer = tf.keras.optimizers.Adam(learning_rate=initial_learning_rate)

model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Callbacks
callbacks = [
    tf.keras.callbacks.ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_accuracy'),
    tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5, restore_best_weights=True),
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', factor=0.2, patience=3, min_lr=1e-6)
]
```

### Train the model with 10 epochs

In [11]:

```
# First training phase
print("Phase 1: Training top layers...")
history1 = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10,
    callbacks=callbacks
)
```

### Finetune it with 15 epochs

In [12]:

```
# Fine-tuning phase
print("Phase 2: Fine-tuning Xception layers...")
base_model = model.layers[1]
base_model.trainable = True

# Recompile with lower learning rate
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history2 = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=15,
    callbacks=callbacks
)
```

**Adam Optimizer:** Efficient gradient descent with **adaptive learning rates**.  
**Loss Function:** Suitable for multi-class classification.

## Why?

- **Initial training** focuses on **top layers**.
- **Fine-tuning** adjusts **pretrained layers** for the dataset.

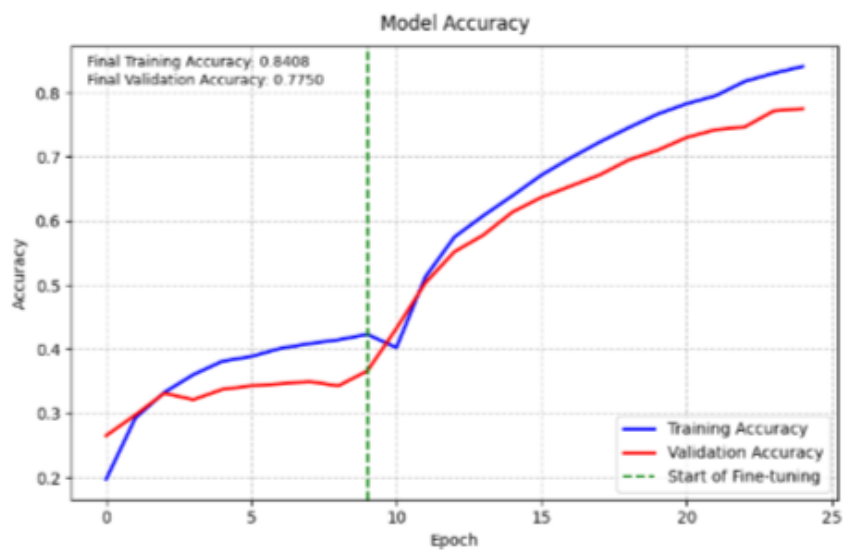
## Results for Xception

### Accuracy

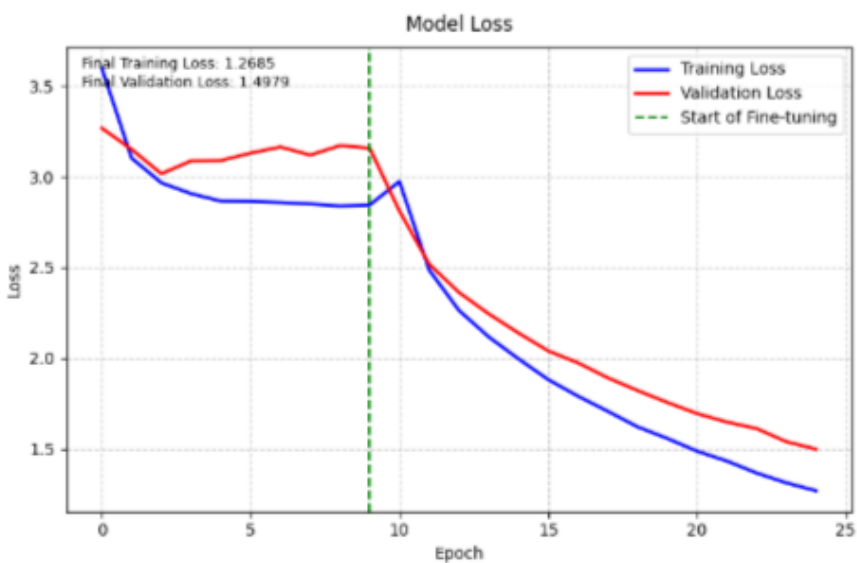
Training Accuracy: 0.8408

Validation Accuracy: 0.7750

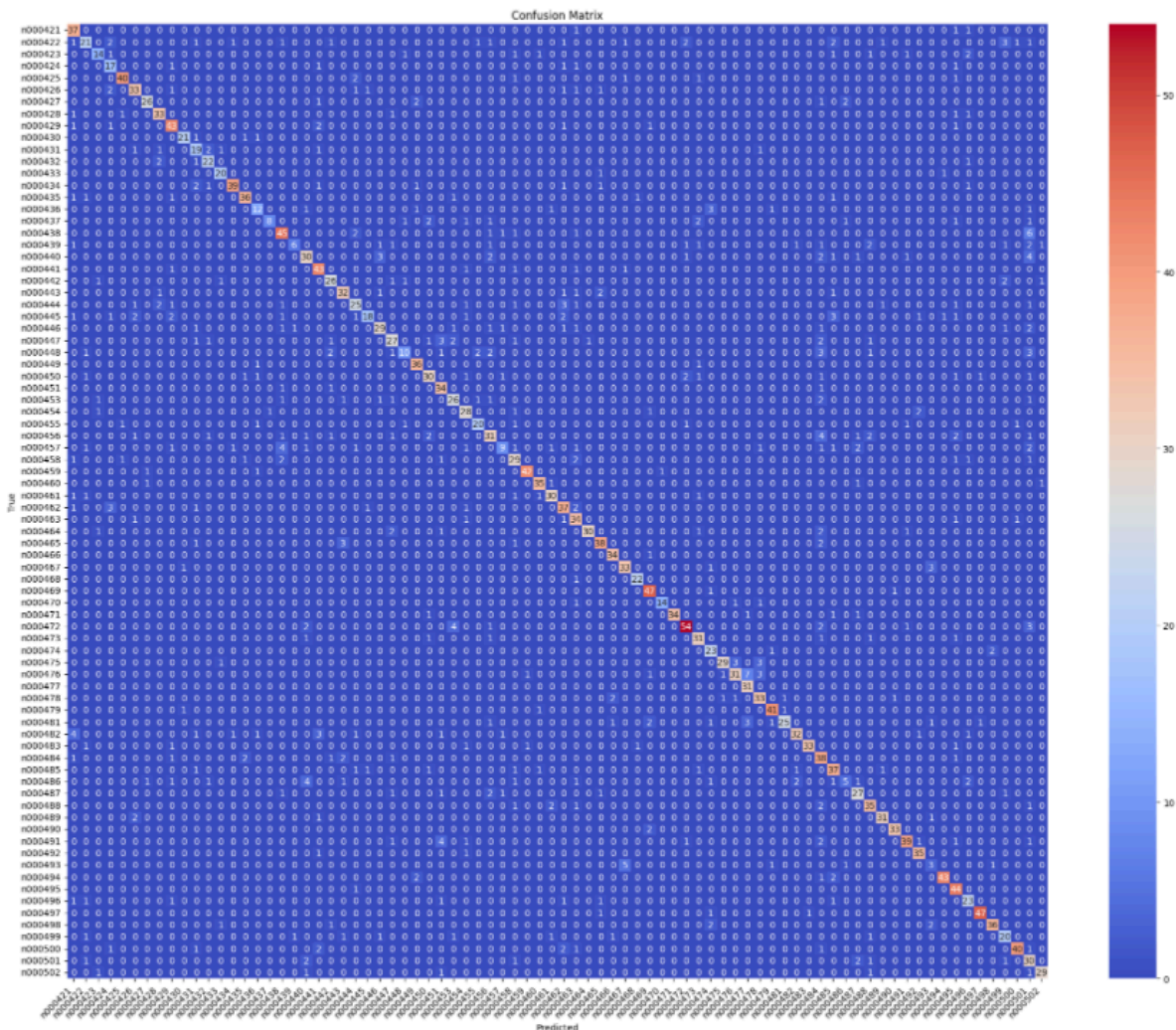
### Accuracy with visualization



### Loss curve with visualization



confusion matrix with visualization



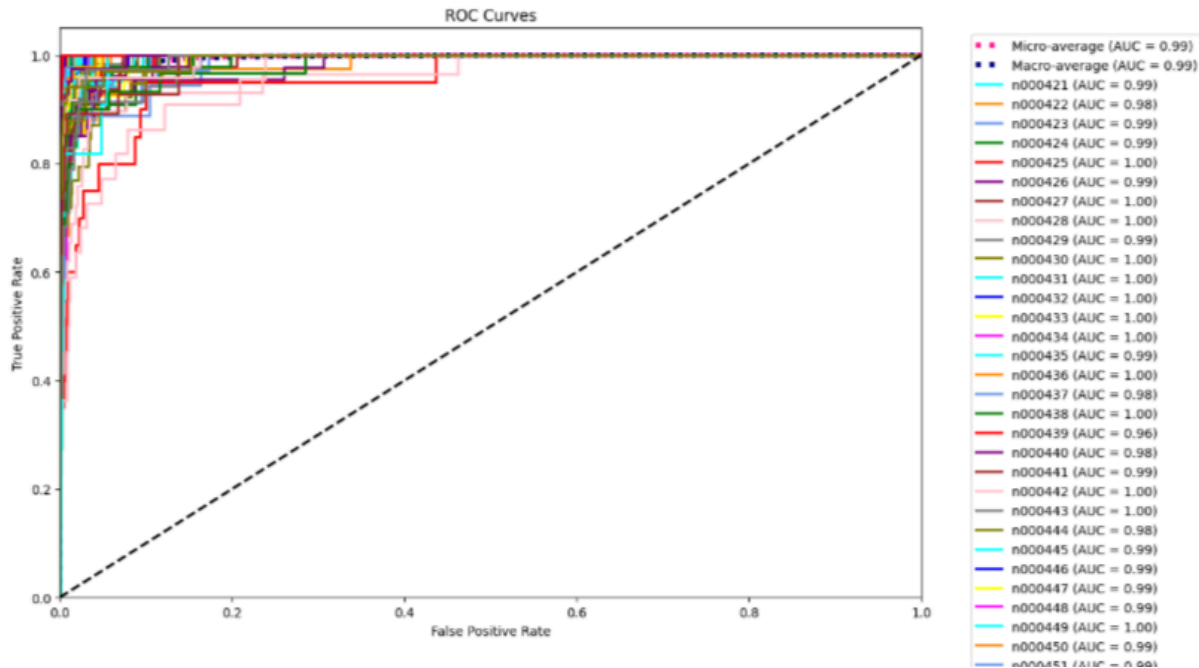
Recall, Precision and f-score

Average Metrics:

=====

	Precision	Recall	F1-Score
Macro Average	0.794687	0.777405	0.777017
Weighted Average	0.804026	0.794820	0.792012

## ROC, AUC graph



## References and Papers:-

Original Paper: [Xception: Deep Learning with Depthwise Separable Convolutions](#) (François Chollet, 2017).

### 3) DenseNet Finetuning:

#### What is DenseNet?

**DenseNet**, short for **Dense Convolutional Network**, was introduced in the paper: "Densely Connected Convolutional Networks" by Gao Huang et al. ([Paper Link](#)).

#### Key Features:

##### 1. Dense Connections:

- Every layer is connected to all **subsequent layers**, ensuring maximum feature reuse.
- This **reduces redundancy and improves gradient flow**.

##### 2. Parameter Efficiency:

- Since layers reuse features, DenseNet requires **fewer parameters** compared to traditional CNNs.

##### 3. Benefits:

- Reduces the **vanishing gradient problem**.
- Reduces **overfitting by promoting feature reuse**.
- Requires **fewer parameters and computations compared to ResNet**.

#### 1)Importing Libraries

```
In [3]: import tensorflow as tf
        from tensorflow.keras.applications import DenseNet121
        from tensorflow.keras import layers, Model
        import os
        import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns
        from sklearn.metrics import roc_curve, auc, confusion_matrix, classification_report
        from itertools import cycle
        import math
        import pandas as pd
```

## Explanation:

- `DenseNet121`: Loads a pretrained `DenseNet-121` model from `keras.applications`.
- `seaborn/matplotlib`: Used for `visualizing metrics` such as `confusion matrix` and `ROC curve`.
- `sklearn.metrics`: For evaluating classification performance.

## 2) Parameters and Preprocessing

```
In [4]: # Set parameters
        IMG_HEIGHT = 224 # DenseNet121 default input size
        IMG_WIDTH = 224
        BATCH_SIZE = 32
        AUTO = tf.data.AUTOTUNE

        # Get number of classes
        NUM_CLASSES = len(os.listdir(train_dir))
        print(f"Number of classes: {NUM_CLASSES}")
```

## Explanation:

- **IMG\_HEIGHT, IMG\_WIDTH:** Set to **224x224**, DenseNet-121's expected input size.
- **BATCH\_SIZE:** Determines the number of images per training step.
- **NUM\_CLASSES:** Calculates the number of classes based on subdirectories in the dataset.

## 3. Preprocessing Functions

### a. Image Loading

```
In [5]: def process_path(file_path, label):  
        # Read and preprocess image  
        img = tf.io.read_file(file_path)  
        img = tf.image.decode_jpeg(img, channels=3)  
        img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])  
        img = tf.cast(img, tf.float32)  
        # Preprocess for DenseNet  
        img = tf.keras.applications.densenet.preprocess_input(img)  
        return img, label
```

## Explanation:

- Loads and decodes the image.
- Resizes it to the required dimensions (**224x224**).
- Applies DenseNet's preprocessing (**subtracts mean pixel value, divides by standard deviation**).



## b. Data Augmentation

```
def augment(image, label):  
    # Data augmentation  
    image = tf.image.random_flip_left_right(image)  
    image = tf.image.random_flip_up_down(image)  
  
    # Random brightness, contrast, and saturation  
    image = tf.image.random_brightness(image, 0.2)  
    image = tf.image.random_saturation(image, 0.5, 2.0)  
    image = tf.image.random_contrast(image, 0.5, 2.0)  
  
    return image, label
```

### Explanation:

- **Random transformations** improve model **robustness** by simulating variations in data (e.g., flipped or brighter images).

## 4. Dataset Creation

```
def create_dataset(directory, is_training=False):  
    class_names = sorted(os.listdir(directory))  
    class_dict = {name: idx for idx, name in enumerate(class_names)}  
  
    image_paths = []  
    labels = []  
  
    for class_name in class_names:  
        class_path = os.path.join(directory, class_name)  
        for img_name in os.listdir(class_path):  
            if img_name.lower().endswith(('.png', '.jpg', '.jpeg')):  
                img_path = os.path.join(class_path, img_name)  
                image_paths.append(img_path)  
                labels.append(class_dict[class_name])  
  
    ds = tf.data.Dataset.from_tensor_slices((image_paths, labels))  
  
    if is_training:  
        ds = ds.shuffle(len(image_paths), reshuffle_each_iteration=True)  
  
    ds = ds.map(process_path, num_parallel_calls=AUTO)  
  
    if is_training:  
        ds = ds.map(augment, num_parallel_calls=AUTO)  
  
    ds = ds.batch(BATCH_SIZE)  
    ds = ds.prefetch(AUTO)  
  
    return ds, len(image_paths)
```

### Explanation:

- Loads image paths and labels.
- Applies preprocessing and augmentation (only for training data).
- **Prefetching**: Ensures data loading does not become a **bottleneck**.

### Benefits of Prefetching:

1. **Overlapping Data Preparation and Computation:**
  - Prefetching allows the pipeline to prepare the next batch of data **while** the current batch is being used for training.
  - This overlap minimizes idle time for the training hardware.
2. **Efficiency:**
  - Data is prepared ahead of time and readily available when required.
  - For large datasets, this can significantly reduce training time.
3. **Seamless Pipeline:**
  - By ensuring a constant flow of data to the GPU/CPU, prefetching maintains smooth training execution.

## 5. Model Creation

```
In [7]: def create_model():
# Load pre-trained DenseNet121
base_model = DenseNet121(
    weights='imagenet',
    include_top=False,
    input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)
)

# Freeze the base model
base_model.trainable = False

# Create model
inputs = tf.keras.Input(shape=(IMG_HEIGHT, IMG_WIDTH, 3))
x = base_model(inputs, training=False)

# Add custom layers
x = layers.GlobalAveragePooling2D()(x)
x = layers.BatchNormalization()(x)

# Dense layers
x = layers.Dense(1024, kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Dropout(0.5)(x)

x = layers.Dense(512, kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Dropout(0.5)(x)

# Output layer
outputs = layers.Dense(NUM_CLASSES, activation='softmax')(x)

return Model(inputs, outputs)
```

## Explanation:

### 1. DenseNet121 Base:

- `weights='imagenet'`: Loads pretrained weights from ImageNet.
- `include_top=False`: Removes the final classification layers, allowing custom layers.
- **Why DenseNet121?** Efficient parameter usage, strong feature reuse, and high accuracy.

### 2. Custom Head:

- `GlobalAveragePooling2D`: Reduces spatial dimensions to **1x1** while retaining features
- Dense layers with Batch Normalization, ReLU activation, Dropout, and L2 Regularization to **prevent overfitting**.
- Output layer uses **softmax for multiclass classification**.

## 6. Training

### Phase 1: Training Only Custom Layers

```
print("Phase 1: Training top layers...")
history1 = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10,
    callbacks=callbacks
)
```

**Initial Phase:** Only the custom layers are trained while the DenseNet base remains **frozen**.

### Phase 2: Fine-Tuning

```

base_model.trainable = True

# Freeze first 100 layers of DenseNet
for layer in base_model.layers[:100]:
    layer.trainable = False

# Recompile with lower learning rate
model.compile(
    optimizer=tf.keras.optimizers.Adam(
        learning_rate=1e-5,
        weight_decay=1e-6
    ),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

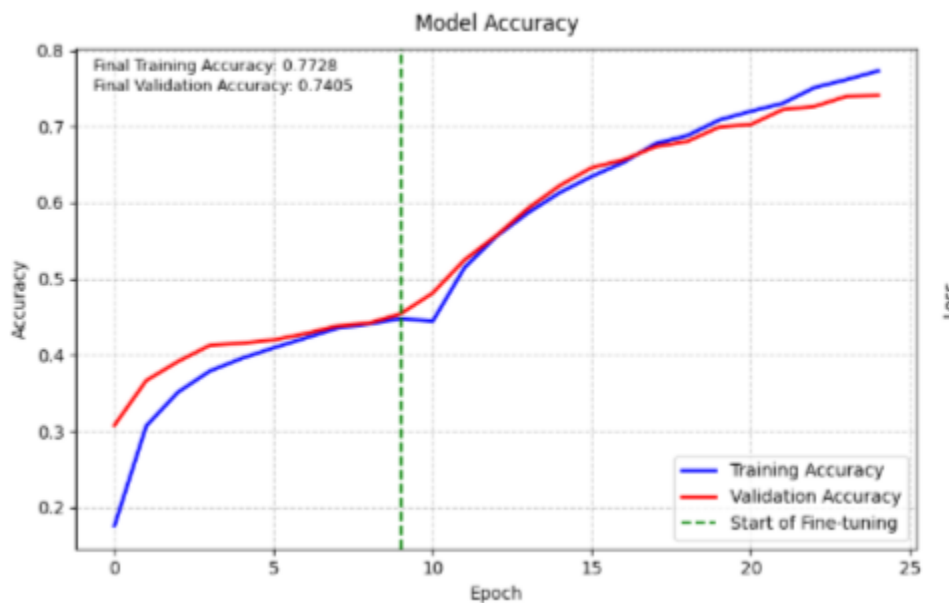
# Second training phase
history2 = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=15,
    callbacks=callbacks
)

```

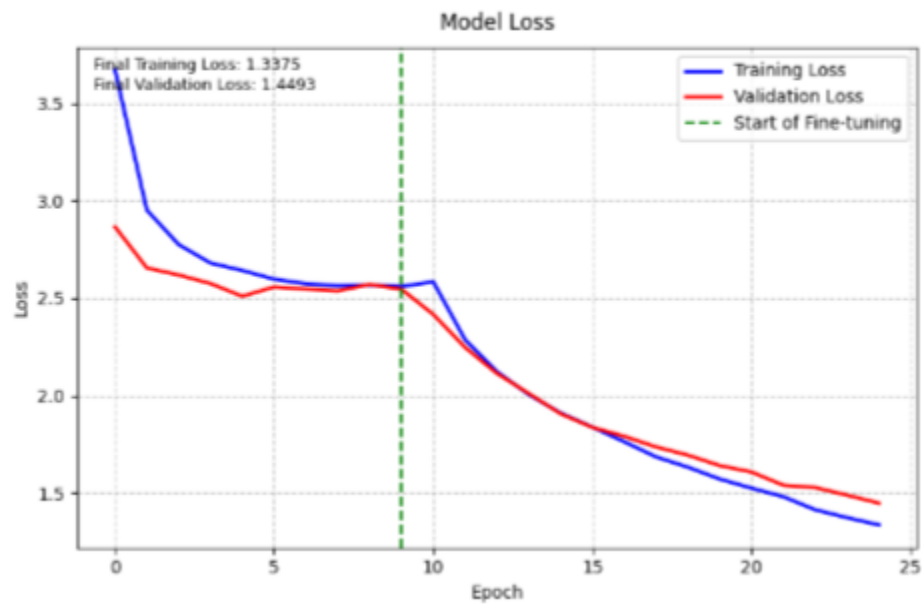
**Fine-Tuning:** Allows selected DenseNet layers to be **trainable** while **freezing** earlier layers to **retain pretrained features**.

## Results for DenseNet

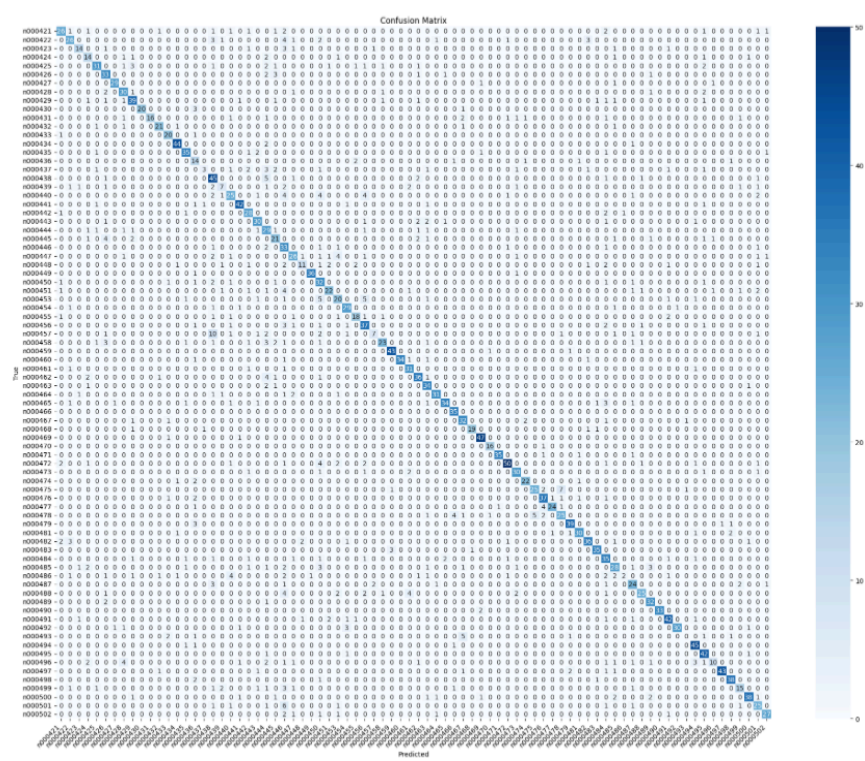
### accuracy with visualization



loss curve with visualization



confusion matrix with visualization



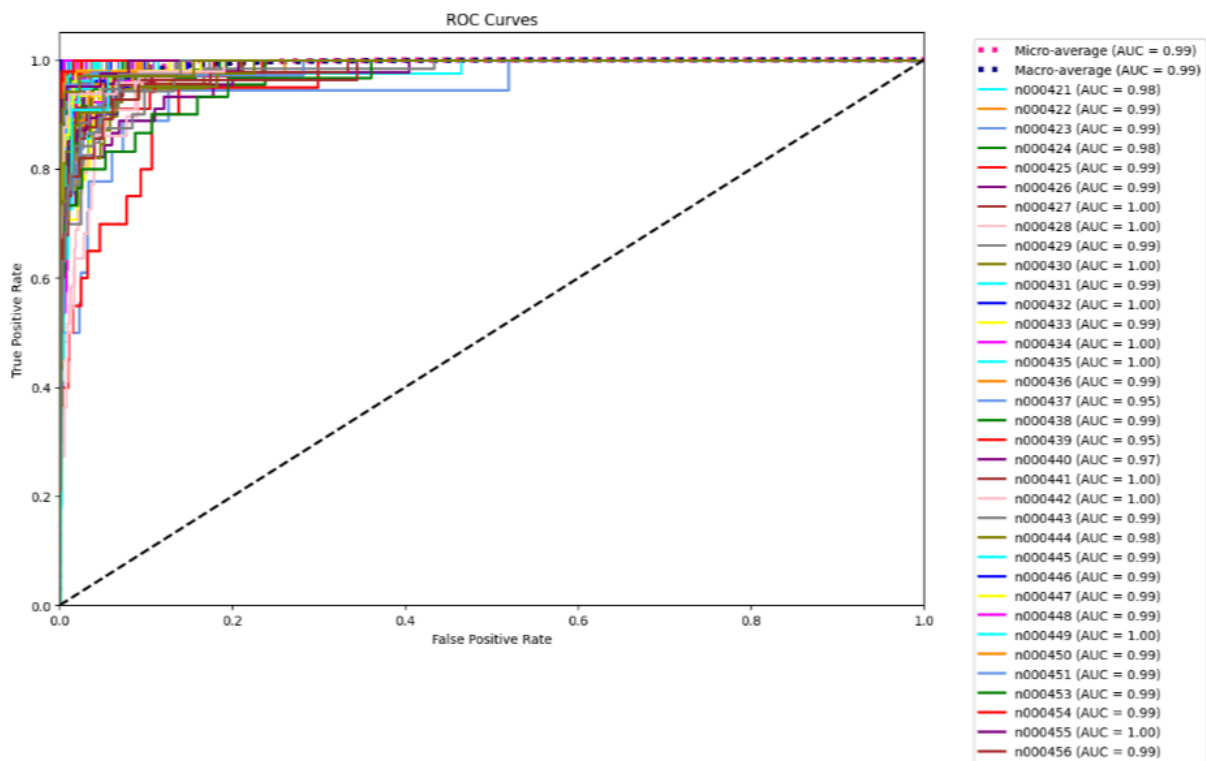
## recall, precision, f-score

Average Metrics:

=====

	Precision	Recall	F1-Score
Macro Average	0.762601	0.734106	0.734552
Weighted Average	0.772903	0.756811	0.753750

## ROC & AUC



## Accuracy

Training Accuracy: 0.7728

Validation Accuracy: 0.7405

**References and Papers:-** DenseNet Paper: Gao Huang, et al.,  
*Densely Connected Convolutional Networks.* [Paper Link](#)

# Comparison of ResNet, DenseNet, and Xception for Celebrity Face Classification

## Results and Performance Summary

Model	Accuracy	Key Features	Inference speed	Parameter Efficiency	Ease of Training
ResNet	73.63%	Residual connections mitigate vanishing gradients and allow deep architectures.	Moderate	Moderate	Easy
DenseNet	74.05%	Dense connections maximize feature reuse and improve gradient flow.	Slow	High (Low Redundancy)	Moderate
Xception	77.50%	Depthwise separable convolutions for computational efficiency and strong feature extraction.	Fast	Moderate	Moderate

## Pros and Cons of Each Architecture

### 1)ResNet

- **Pros:**
  1. **Residual Connections:** Enables effective training of **very deep networks** by bypassing layers through identity mappings.
  2. **Simplifies Optimization:** The network learns **residual mappings (differences)** instead of full transformations, reducing the risk of overfitting.
  3. **Well-suited for General Tasks:** A robust architecture that performs well on **a variety of image classification tasks**.

4. **Scalability:** Can be scaled to very deep architectures like ResNet-101 or ResNet-152.

- **Cons:**

1. **Parameter Redundancy:** Requires more parameters compared to DenseNet due to lack of feature reuse across layers.
  2. **Moderate Computational Demand:** Although residual connections improve training, the architecture is not as parameter-efficient as DenseNet.
- 

## 2)DenseNet

- **Pros:**

1. **Feature Reuse:** Dense connections ensure all layers can access features from preceding layers, improving efficiency.
2. **Parameter Efficiency:** Requires fewer parameters by reusing features and avoiding redundant calculations.
3. **Gradient Flow:** Alleviates vanishing gradients due to dense connections.

- **Cons:**

1. **Inference Speed:** Dense connections increase memory and computational overhead during inference.
  2. **Training Complexity:** Can be harder to train due to potential overfitting from over-reliance on feature reuse.
-



### 3)Xception

- **Pros:**

1. **Depthwise Separable Convolutions:** Reduces **computational complexity** by decoupling spatial and depthwise convolutions.
2. **Strong Feature Extraction:** Excels in learning **hierarchical features**, making it suitable for complex tasks like facial recognition.
3. **Parameter Efficiency:** Balances between **DenseNet** and **ResNet** in terms of parameter requirements.
4. **Faster Inference:** Depthwise separable convolutions improve speed without **compromising accuracy**.

- **Cons:**

1. **Moderate Training Complexity:** Requires careful adjustment of hyperparameters to avoid **overfitting**.
2. **Not Fully Generalized:** While it excels in specific tasks, it may not always outperform traditional architectures on **simpler datasets**.

---

## Analysis and Recommendation for Celebrity Face Classification

### Dataset Characteristics:

1. **High Variability:** Celebrity datasets typically have **diverse lighting**, expressions, and occlusions.
2. **Relatively Small Number of Samples per Class:** Demands **strong feature extraction** to handle **overfitting**.
3. **Task Complexity:** Requires learning **subtle** differences between faces.

## Best Model for the Task: Xception

- **Reasons:**

1. **Higher Accuracy:** Xception achieves 77.50% accuracy, outperforming ResNet and DenseNet on this dataset.
  2. **Feature Extraction:** Depthwise separable convolutions allow efficient extraction of spatial hierarchies, crucial for distinguishing fine details in celebrity faces.
  3. **Inference Speed:** Faster inference with a comparable number of parameters makes it suitable for real-world applications like facial recognition systems.
  4. **Generalization:** Excels in handling datasets with variability in facial attributes, as observed from its superior validation performance.
- 

## Conclusion

For celebrity face classification:

1. **Xception** is the most suitable due to its accuracy and computational efficiency.
2. **DenseNet** is a close contender but struggles with inference speed and overfitting on small datasets.
3. **ResNet** provides robust performance but lacks the parameter efficiency and advanced feature extraction capabilities of the other models.

## References and papers for all documentation:

DenseNet (Densely Connected Convolutional Networks)

1. **Original Paper:**

- Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). *Densely Connected Convolutional Networks*. [ArXiv Link](#)

## 2. Github Implementations:

- DenseNet PyTorch Implementation [Link](#)
- 

## Xception (Extreme Inception)

### 1. Original Paper:

- Chollet, F. (2016). *Xception: Deep Learning with Depthwise Separable Convolutions*. [ArXiv Link](#)

### 2. Github Implementations:

- Xception TensorFlow Implementation [Link](#)
- 

## General Resources

### 1. Transfer Learning in Keras:

- Official Documentation: [Link](#)

### 2. ImageNet Dataset for Pretraining:

- ImageNet Official Website: [Link](#)

### 3. Visualization Tools:

- TensorFlow documentation: [Link](#)
- Grad-CAM for Model Explainability: [Link](#)

## This project applied by :

Farida Khaled

Madiha Saeed

Mohammed Tarek

Hanya Ruby

Esraa Ammar

Mohammed Yasser

## Our repo on Github

<https://github.com/Farida-EL-Shenawy/Neural-networks-and-Deep-learning-models>