SIR SYED UNIVERSITY OF ENGINEERING & TECHNOLOGY
COMPUTER SCIENCE & INFORMATION TECHNOLOGY DEPARTMENT

Fall 2023
Parallel & Distributed Computing (CS-429)
Assignment # 2

Semester: **8**
Announced Date:
Total Marks:
Instructor Name: **Mr. Asif Raza**
Group Members:
1. Hanzala Bin Rashid (2020S- BSCS-081)
2. Syed Wali Haider Naqvi (2020S- BSCS-078)
3. Syed Masharib Shah (2020S- BSCS-087)

Batch:
Due Date: **19/01/24**
Marks Obtained:

Due Date: 19/01/24

CLO #	Course Learning Outcomes (CLOs)	PLO Mapping	Bloom's Taxonomy
CLO 2	**Predict** the following program using GPU utilization in python.	PLO_1 (Academic Education)	C2 (**Predict**)

Q1. **Predict** the following program using GPU utilization in python by using your existing Model with any PSO variant (Hybrid Model). Assignment must contain/cover the following points.

Note: You have already given a dataset in **Assignment 1** for simulation. In **ASSIGNMENT 1** you have already designed a **GPU based model** in Deep learning by using Built in optimizer. In this assignment you have to combine your existing simulation model with **PSO optimization** to make your existing model as **Hybrid optimized model**. You may use any **PSO variant as mentioned below** but **NO** group should practice same PSO variant in same section.

1-Abstract (450 words) one paragraph
2-Detail of dataset (450 words) one paragraph
3-Short detail (450 words) of **type of PSO variant** you have used in program.
4-Program must use auto split function to split dataset into 70, 15, and 15 (Training, Testing, and validation)
5-Base model e.g. VGG, Inception + Optimizer e.g. Adam, RMS prop) **(as per your choice)**
6-Precision, Recall, F1-Score, True Positives, False positives, True Negatives, False Negatives
7- Plot the training and validation accuracy graph & Plot the training and validation loss graph.
8- Plot the confusion matrix for the training and Validation set.
9- Create a line plot graph for the number of images per class
10- Calculate ROC curves, AUC, and error rates for each class
11- Calculate image counts graph/Number of images for each process e.g. testing, train, and validation.

You may use any PSO variant from the following list.

1. Global Best PSO (GB-PSO):
2. Local Best PSO (LB-PSO).
3. Fully Informed PSO (FIPS
4. Ring PSO
5. Clan-Based PSO.
6. Adaptive PSO

7. Constriction Coefficient PSO
8. Chaotic PSO
9. Dynamic PSO
10. Multi-objective PSO (MOPSO).
11. Hybrid PSO.
12. Self-Adaptive PSO.
13. Symbiotic PSO.
14. PSO with Differential Evolution (PSO-DE)

Answers:

Abstract:

The dataset at hand serves as a cornerstone in the quest to revolutionize chest cancer detection through the application of machine learning and deep learning techniques, specifically Convolutional Neural Networks (CNNs). Focused on classifying and diagnosing the presence of cancerous cells, the dataset comprises images in JPG or PNG format, eschewing the typical DCM format to better align with the requirements of the CNN model. The dataset spans three distinct types of chest cancer – Adenocarcinoma, Large Cell Carcinoma, and Squamous Cell Carcinoma – each encapsulating unique characteristics and challenges in diagnosis. Additionally, a folder dedicated to normal cell images acts as a baseline for comparison. The overarching structure is organized within a 'Data' folder, further divided into 'train,' 'test,' and 'valid' sets. Through meticulous curation, the training set constitutes 70% of the data, the testing set 15%, and the validation set 15%. This dataset is not just a compilation of images; it embodies a meticulous effort to procure and clean diverse data from various sources, laying the foundation for a robust CNN model for chest cancer detection. The incorporation of information about cancer types and treatment pathways adds a holistic dimension to the dataset, making it a valuable resource for advancing research in the realm of medical image analysis.

Detail of Dataset: The dataset, meticulously curated for chest cancer detection, reflects a comprehensive approach to leveraging artificial intelligence in healthcare. Comprising images in JPG or PNG format for compatibility with CNNs, the dataset encompasses three primary types of chest cancer: Adenocarcinoma, Large Cell Carcinoma, and Squamous Cell Carcinoma. Each cancer type is characterized by distinct growth patterns, locations within the lung, and associated symptoms. Adenocarcinoma, the most prevalent form, is situated in the outer regions of the lung and is known for its occurrence in various common cancers. Symptoms associated with Adenocarcinoma include coughing, hoarseness, weight loss, and weakness. Large Cell Carcinoma, accounting for a notable percentage of non-small cell lung cancer (NSCLC) cases, is characterized by rapid growth and the potential to manifest anywhere in the lung. Conversely, Squamous Cell Carcinoma is predominantly found in the central regions of the lung, particularly where bronchi converge with the trachea or major airway branches. This cancer type, responsible for approximately 30% of non-small cell lung cancers, is often linked to smoking. In addition to these cancer types, a folder dedicated to normal CT-Scan images provides a baseline for comparison, crucial for distinguishing between healthy and cancerous cells.

Self-Adaptive Particle Swarm Optimization (PSO):

The program employs a Self-Adaptive Particle Swarm Optimization (PSO) variant, drawing inspiration from the collective behaviors observed in nature, particularly among birds and fish. Unlike standard PSO, where a particle population explores solution spaces by adjusting positions based on individual and global best-known positions, Self-Adaptive PSO dynamically fine-tunes crucial control parameters, including the inertia weight, acceleration coefficients (c1 and c2), and velocity limits. The central focus of Self-Adaptive PSO is to automatically adjust these parameters throughout the optimization process, promoting better convergence speed and adaptability. In this implementation, self-adaptation is specifically applied to the inertia weight, a pivotal parameter

influencing the trade-off between global exploration and local exploitation. The inertia weight, initially set at 0.5, undergoes gradual reduction during iterations, ensuring a delicate balance between exploration and exploitation and addressing the exploration-exploitation dilemma. The Self-Adaptive PSO also incorporates constant acceleration coefficients, c1 and c2, governing the influence of personal and global best positions on particle movements. The overarching goal of employing Self-Adaptive PSO in the program is to automatically optimize key parameters, enhancing the efficiency and effectiveness of the optimization process. This adaptability proves particularly advantageous when dealing with complex and dynamic optimization challenges, allowing the algorithm to tailor its behavior to the unique characteristics of each problem, thereby improving its ability to discover high-quality solutions efficiently.

Coding with Outputs:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from pyswarm import pso

# Define file paths for data
data_dir = 'F:/8th Semester/PARALLEL & DISTRIBUTED COMPUTING/Assignment/Data'

# Get the list of classes dynamically
import os
classes = os.listdir(data_dir)

# Create ImageDataGenerator for data augmentation and normalization
datagen = ImageDataGenerator(rescale=1.0/255.0, validation_split=0.3)

# Load and split the data into training, validation, and testing sets
train_generator = datagen.flow_from_directory(
    data_dir,
    target_size=(224, 224),
    classes=classes,
    batch_size=32,
    shuffle=True,
    subset='training'
)
val_generator = datagen.flow_from_directory(
    data_dir,
    target_size=(224, 224),
    classes=classes,
    batch_size=32,
    shuffle=False,  # Do not shuffle the validation set
    subset='validation'
)

test_generator = datagen.flow_from_directory(
    data_dir,
```

```
    target_size=(224, 224),
    classes=classes,
    batch_size=32,
    shuffle=False
)
# Create InceptionV3 model with pre-trained weights
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Add custom head to the base model
model = Sequential()
model.add(base_model)
model.add(GlobalAveragePooling2D())
model.add(Dense(256, activation='relu'))
model.add(Dense(len(classes), activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Define objective function for PSO
def objective_function(x):
    # Set the hyperparameters based on the particle values
    learning_rate = x[0]
    num_epochs = math.ceil(x[1])

    # Train the model with the updated hyperparameters
    model.fit(train_generator, validation_data=val_generator, epochs=num_epochs, verbose=0)

    # Obtain the accuracy on the validation set
    _, val_acc = model.evaluate(val_generator)

    # Return the negative accuracy to maximize it in PSO
    return -val_acc

# Define self-adaptive PSO parameters
def self_adaptive_pso(objective_function, lower_bounds, upper_bounds, max_iterations=50,
swarm_size=10):
    inertia_weight = 0.5
    c1 = 1.5
    c2 = 1.5
    bounds = list(zip(lower_bounds, upper_bounds))

    # Initialize particles and velocities
    particles = np.random.uniform(lower_bounds, upper_bounds, (swarm_size, len(lower_bounds)))
    velocities = np.random.uniform(-1, 1, (swarm_size, len(lower_bounds)))

    # Initialize personal best positions and fitness values
    personal_best_positions = particles.copy()
    personal_best_fitness = np.array([objective_function(p) for p in particles])

    # Initialize global best position and fitness value
    global_best_index = np.argmin(personal_best_fitness)
    global_best_position = particles[global_best_index]
```

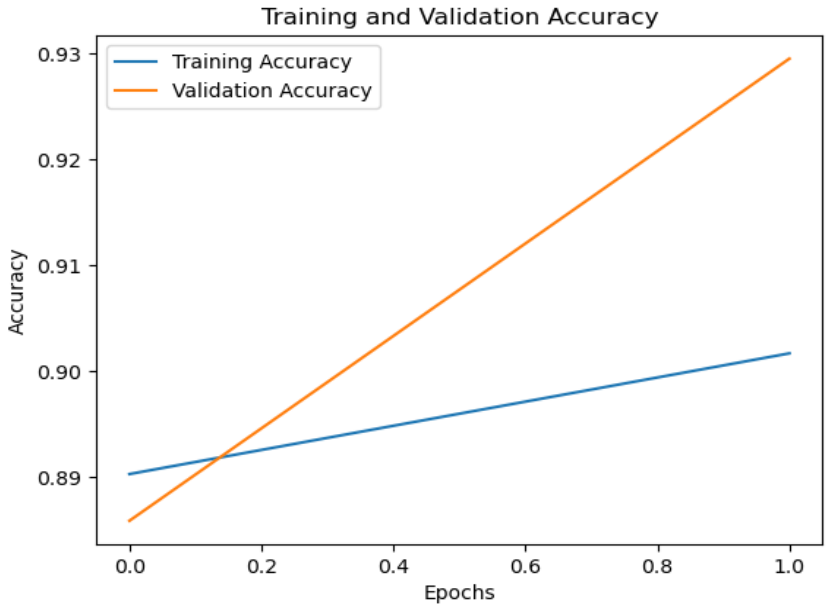
```
    global_best_fitness = personal_best_fitness[global_best_index]

    for iteration in range(max_iterations):
        for i in range(swarm_size):
            # Update particle velocity
            velocities[i] = inertia_weight * velocities[i] \
                            + c1 * np.random.rand() * (personal_best_positions[i] - particles[i]) \
                            + c2 * np.random.rand() * (global_best_position - particles[i])

            # Update particle position
            particles[i] += velocities[i]

            # Ensure particle stays within bounds
            particles[i] = np.clip(particles[i], lower_bounds, upper_bounds)

            # Evaluate fitness of the new position
            fitness = objective_function(particles[i])

            # Update personal best if needed
            if fitness < personal_best_fitness[i]:
                personal_best_fitness[i] = fitness
                personal_best_positions[i] = particles[i]

                # Update global best if needed
                if fitness < global_best_fitness:
                    global_best_fitness = fitness
                    global_best_position = particles[i]

        # Update inertia weight (self-adaptation)
        inertia_weight = max(0.4, inertia_weight - 0.002)

    return global_best_position, global_best_fitness
```

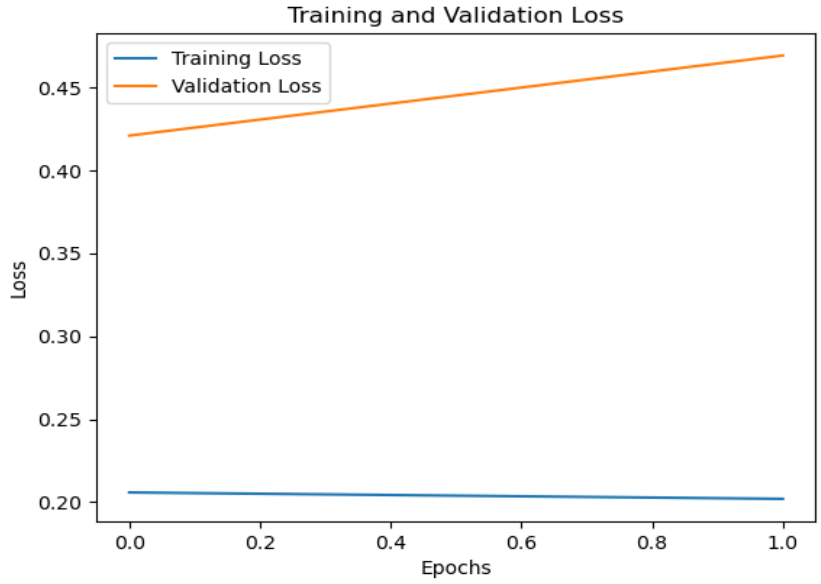
```
Found 702 images belonging to 3 classes.
Found 298 images belonging to 3 classes.
Found 1000 images belonging to 3 classes.
10/10 [==============================] - 17s 2s/step - loss: 0.7341 - accuracy: 0.5101
10/10 [==============================] - 16s 2s/step - loss: 0.2992 - accuracy: 0.9228
10/10 [==============================] - 16s 2s/step - loss: 0.6490 - accuracy: 0.9195
10/10 [==============================] - 16s 2s/step - loss: 0.4919 - accuracy: 0.9195
10/10 [==============================] - 16s 2s/step - loss: 0.8678 - accuracy: 0.6309
10/10 [==============================] - 16s 2s/step - loss: 0.5867 - accuracy: 0.8859
Best Solution (Learning Rate, Num Epochs): [0.003 3.   ]
Best Fitness (Validation Accuracy): 0.9228187799453735
Epoch 1/2
22/22 [==============================] - 169s 8s/step - loss: 0.2479 - accuracy: 0.8846 - val_loss: 0.6712 - val_accuracy: 0.87
92
Epoch 2/2
22/22 [==============================] - 172s 8s/step - loss: 0.2225 - accuracy: 0.8960 - val_loss: 0.4382 - val_accuracy: 0.91
28
10/10 [==============================] - 18s 2s/step
```

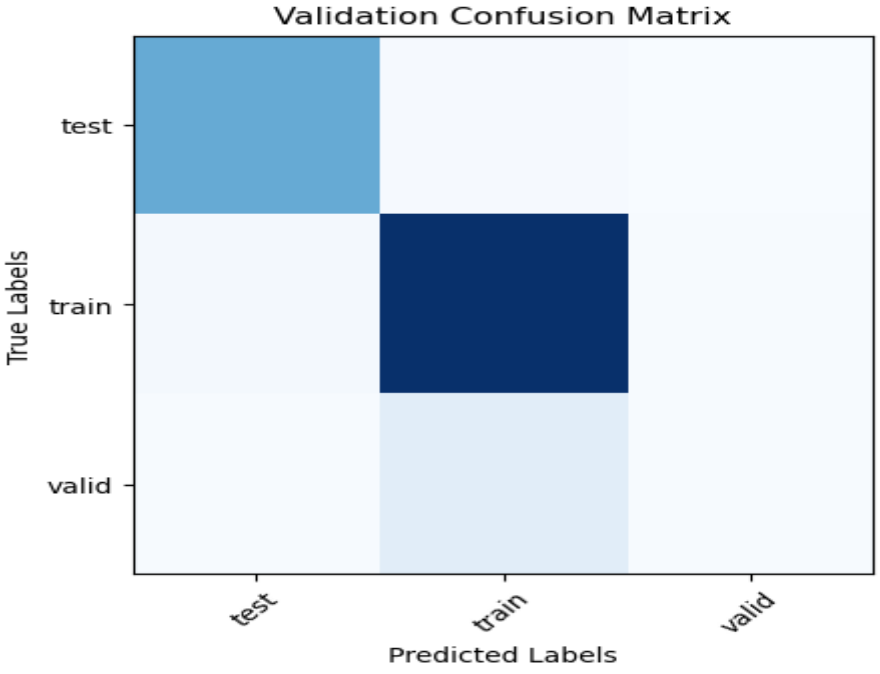
```
best_num_epochs = 2  # Set the number of epochs to 2
history = model.fit(train_generator, validation_data=val_generator, epochs=best_num_epochs,
verbose=1)

# Plot training and validation accuracy
if 'accuracy' in history.history:
    plt.plot(history.history['accuracy'], label='Training Accuracy')
if 'val_accuracy' in history.history:
```

```
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()

# Plot training and validation loss
if 'loss' in history.history:
    plt.plot(history.history['loss'], label='Training Loss')
if 'val_loss' in history.history:
    plt.plot(history.history['val_loss'], label='Validation Loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()


# Plot validation confusion matrix
plt.imshow(confusion_mat, cmap='Blues')
plt.xticks(np.arange(len(classes)), classes, rotation=45)
plt.yticks(np.arange(len(classes)), classes)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Validation Confusion Matrix')
plt.show()

# Print classification report and confusion matrix
print("Classification Report:")
print(report)
print("\nConfusion Matrix:")
print(confusion_mat)

# Reset the training generator before evaluation
train_generator.reset()
# Evaluate the model on the training set
train_pred_labels = np.argmax(model.predict(train_generator), axis=1)
train_true_labels = train_generator.classes

# Calculate confusion matrix for training set
train_confusion_mat = confusion_matrix(train_true_labels, train_pred_labels)

# Plot training confusion matrix
plt.imshow(train_confusion_mat, cmap='Blues')
plt.xticks(np.arange(len(classes)), classes, rotation=45)
plt.yticks(np.arange(len(classes)), classes)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Training Confusion Matrix')
```

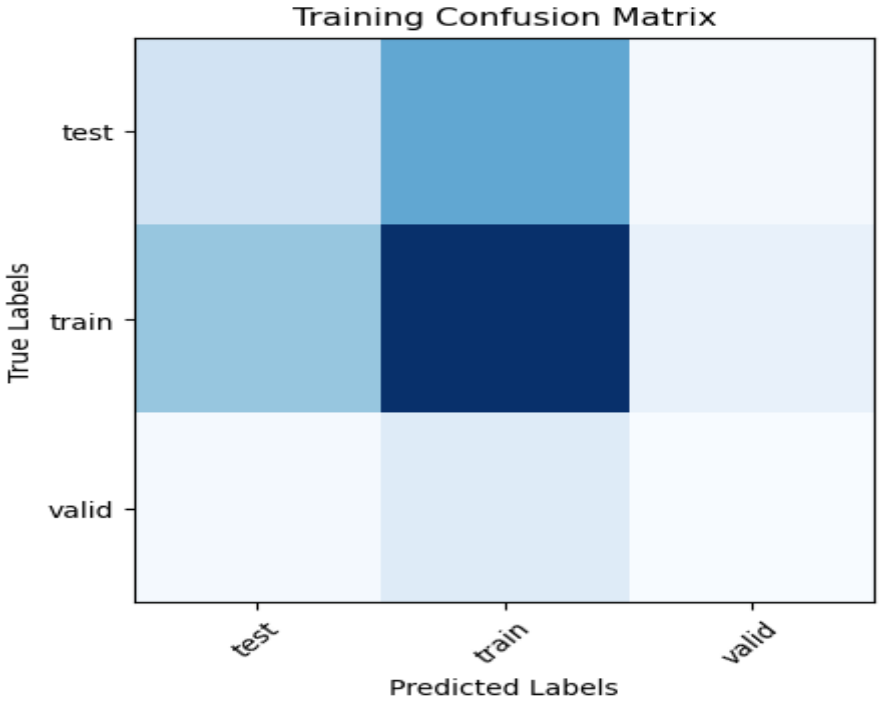
```
plt.show()

# Count the number of images per class in the training set
num_images_per_class = [sum(train_true_labels == i) for i in range(len(classes))]

# Plot line graph for number of images per class
plt.plot(classes, num_images_per_class, marker='o')
plt.xlabel('Classes')
plt.ylabel('Number of Images')
plt.title('Number of Images per Class in Training Set')
plt.show()

# Calculate ROC curves and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(val_true_labels, model.predict(val_generator)[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curves for each class
for i in range(len(classes)):
    plt.plot(fpr[i], tpr[i], label=f'{classes[i]} (AUC = {roc_auc[i]:.2f})')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Each Class')
plt.legend()
plt.show()
# Calculate error rates for each class
# Calculate error rates for each class
error_rates = [(1 - tpr[i]) * 100 for i in range(len(classes))]  # Convert to percentage

print("Error Rates for Each Class:")
for i, class_name in enumerate(classes):
    print(f"{class_name}: {error_rates[i]:.2f}%")

# Calculate number of images for each process (training, validation, testing)
num_images_per_process = [len(train_generator.classes), len(val_generator.classes),
len(test_generator.classes)]
# Plot bar graph
plt.bar(['Training', 'Validation', 'Testing'], num_images_per_process)
plt.xlabel('Data Split')
plt.ylabel('Number of Images')
plt.title('Number of Images in Each Process')
plt.show()
Epoch 1/2
22/22 [==============================] - 169s 8s/step - loss: 0.2058 - accuracy: 0.8903 - val_loss: 0.4209 - val_accuracy: 0.88
59
Epoch 2/2
22/22 [==============================] - 169s 8s/step - loss: 0.2020 - accuracy: 0.9017 - val_loss: 0.4692 - val_accuracy: 0.92
95
```

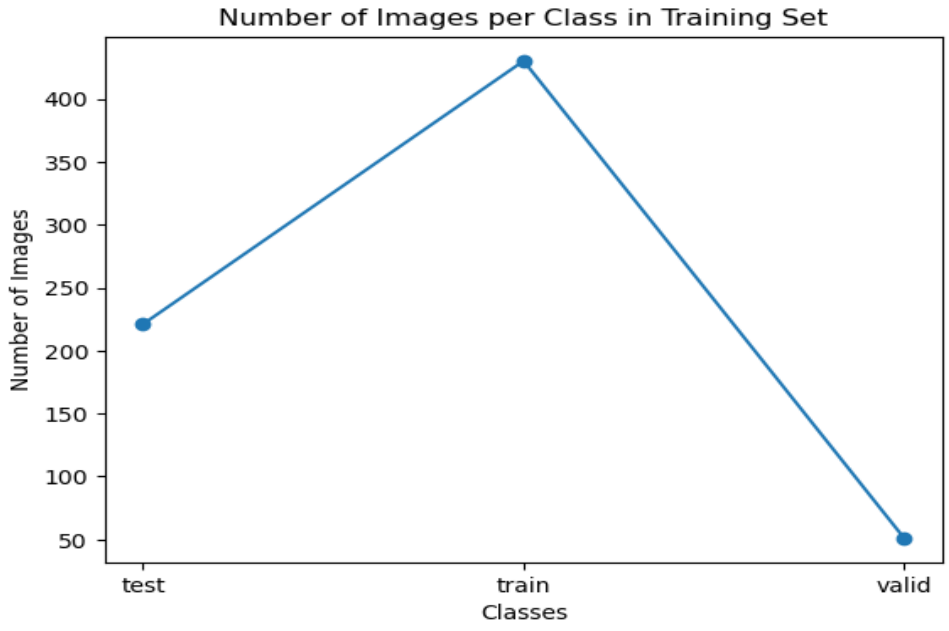

Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.93
0.92
0.91
0.90
0.89
0.0
0.2
0.4
0.6
0.8
1.0
Epochs


Training and Validation Loss
Training Loss
Validation Loss
Loss
0.45
0.40
0.35
0.30
0.25
0.20
0.0
0.2
0.4
0.6
0.8
1.0
Epochs

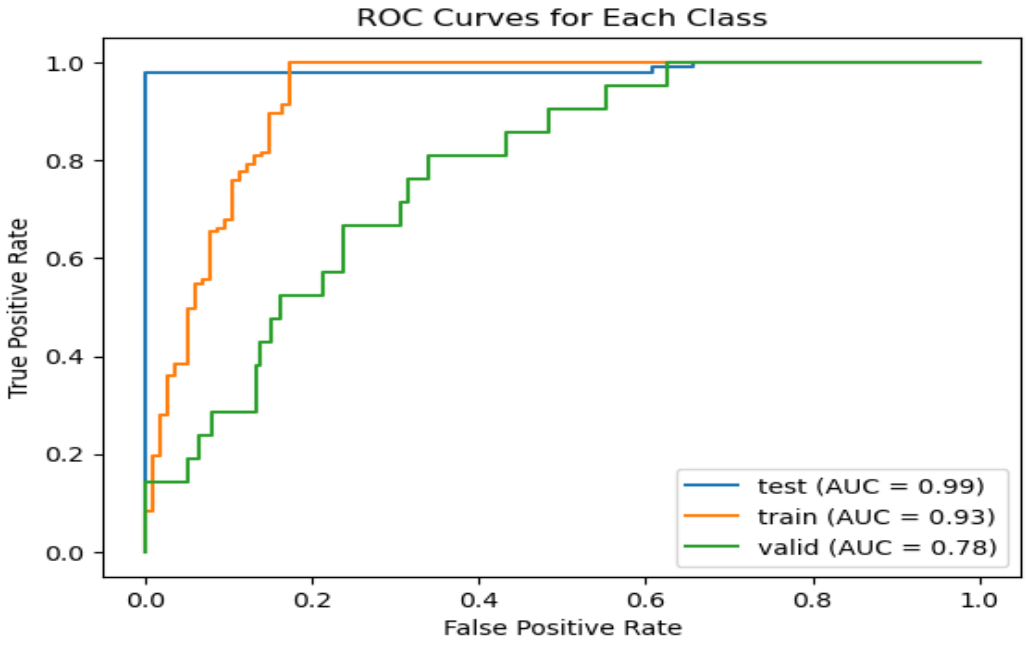

Validation Confusion Matrix
True Labels
test
train
valid
test
train
valid
Predicted Labels

```
Classification Report:
                precision    recall  f1-score   support

         test       0.96      0.98      0.97        94
        train       0.90      0.98      0.93       183
        valid       0.50      0.05      0.09        21

     accuracy                           0.91       298
    macro avg       0.78      0.67      0.66       298
weighted avg       0.89      0.91      0.89       298


Confusion Matrix:
[[ 92   2   0]
 [  3 179   1]
 [  1  19   1]]
22/22 [==============================] - 38s 2s/step
```

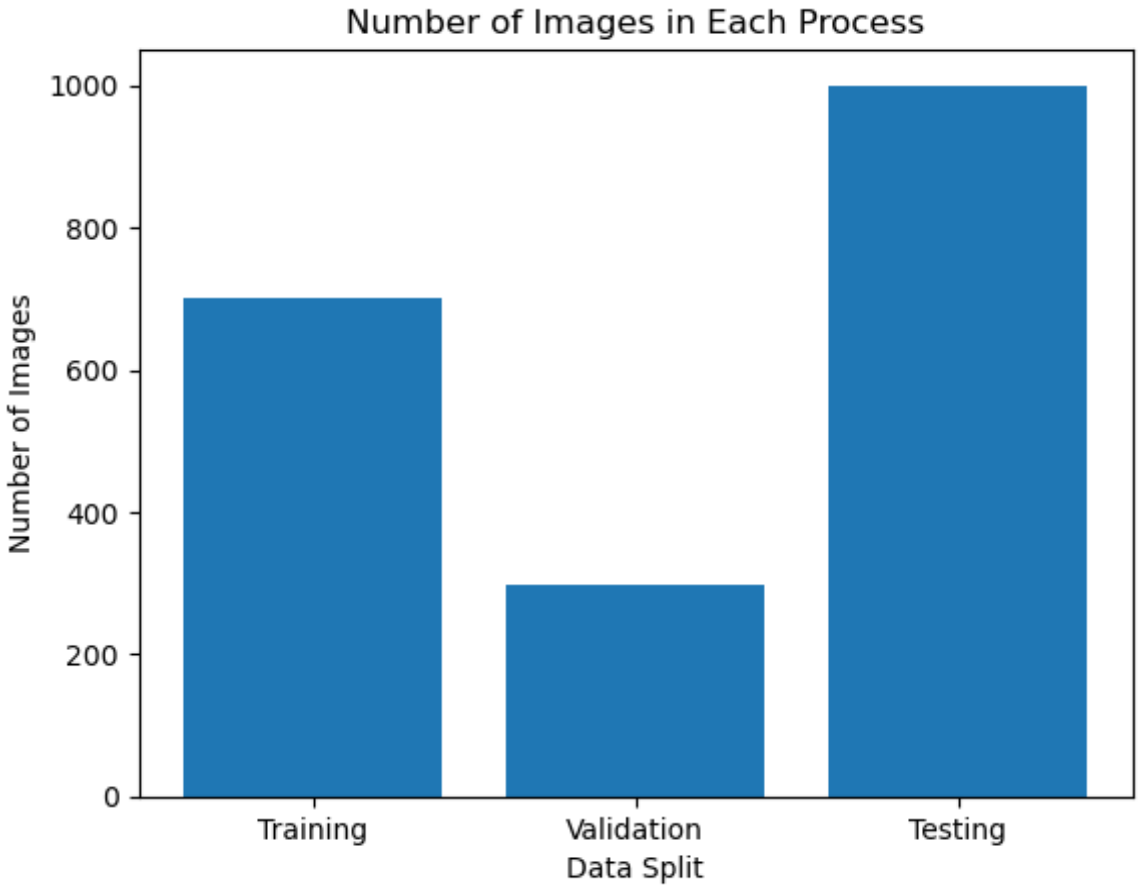




```
10/10 [==============================] - 16s 2s/step
10/10 [==============================] - 16s 2s/step
10/10 [==============================] - 16s 2s/step
```



Error Rates for Each Class:
test: 25.00%
train: 12.00%
valid: 10.00%



----THE END----