Sri Lanka Institute of Information Technology

Year 4, Semester I, 2024
Assignment

Potato Disease Classification Using CNN Architectures on the Plant Village Dataset

Deep Learning (SE4050)

BSc (Hons) in Information Technology Specializing in Software Engineering

Group Details

Student Registration Number	Student Name	Group
IT21251900	Rajapaksha R.M.S.D.	Y4.S1.WE.SE.01
IT21302862	Sri Samadhi L.A.S.S.	Y4.S1.WE.SE.02
IT21178054	Kumari T.A.T.N.	Y4.S1.WD.IT.01
IT21360428	Monali G.M.N.	Y4.S1.WE.SE.02

GitHub Link

IT21251900/DL-Project (github.com)

YouTube Presentation Link

https://www.youtube.com/playlist?list=PLOC02hssbXJ99RsQVUi8lepMcHTRT8xwH

Table of Contents

Introduction

Problem Statement

Potato crops are a crucial agricultural commodity, feeding millions worldwide. However, they are susceptible to various diseases, which can drastically affect yield and quality. Among the most notorious are Early Blight and Late Blight, both of which can cause significant losses if left unchecked. Identifying these diseases early and accurately is essential for effective intervention and treatment.

In this project, we tackle the problem of classifying potato leaf diseases using image data and deep learning techniques. Specifically, we employ supervised learning methods to classify leaf images into categories such as "Healthy," "Early Blight," and "Late Blight." This project leverages the **Plant Village** dataset, which is widely recognized for plant disease classification research, offering a large and diverse set of annotated images. By building robust deep learning models, the system aims to automate the detection of these diseases from images of potato leaves.

Objective

The goal of this project is to develop a machine learning solution to classify potato leaf diseases. Specifically, we aim to:

- Develop and Compare CNN Models: Train and evaluate four distinct Convolutional Neural Networks (CNNs) for accuracy and generalization.
- Provide a Disease Detection API: Deploy a FastAPI backend to process image uploads and provide real-time disease predictions.
- Deploy a React Frontend: Build a user-friendly React interface for easy image uploads and viewing of classification results.

Supervised Learning Approach

Supervised learning was chosen for this project because the problem we are tackling involves **labeled data**, where the input images are associated with known categories (e.g., healthy or diseased classes). In supervised learning, the model is trained to learn the mapping between the input data and its corresponding labels, allowing it to predict the correct class when new, unseen data is presented. This fits well with our objective, which is to classify images into predefined categories based on their features.

The key reasons for choosing supervised learning are:

1. **Labeled Dataset**: Our dataset comes with ground truth labels, meaning each input image is associated with a known class. Supervised learning algorithms are specifically designed to work with labeled data, learning from examples and improving their performance over time.
2. **High Predictive Accuracy**: Supervised learning techniques, especially convolutional neural networks (CNNs), have been widely used in image classification tasks and have shown superior performance. CNNs are highly effective in automatically learning spatial hierarchies of features from images, which are crucial in our classification task.
3. **Clear Performance Metrics**: Since supervised learning involves labeled data, it is easier to evaluate the model's performance through established metrics like accuracy, precision, recall, and F1-score. These metrics provide a clear understanding of how well the model generalizes to unseen data.
4. **Control Over Training Process**: Supervised learning allows fine-tuning of the model by adjusting hyperparameters and providing more control over the training process. This flexibility ensures that we can iteratively improve the model's performance by adjusting the architecture, learning rate, and other factors.

Background Information on Supervised Learning and CNN

Supervised learning is a machine learning approach where models learn from labeled datasets. In the context of this project, the labels represent different categories of plant diseases, and the input data consists of images. CNNs, a type of deep learning model particularly well-suited for image classification tasks, are employed in this project. CNNs automatically extract features from images, learning patterns such as edges, textures, and colors, which are crucial for identifying plant diseases.

CNNs consist of several layers:

- **Convolutional layers**: Extract features using convolution operations.
- **Pooling layers**: Reduce the spatial dimensions of the data.
- **Fully connected layers**: Combine features to classify images into the correct category.

This architecture is ideal for the classification of plant disease images, where intricate patterns in leaf textures and colors need to be identified to distinguish between healthy plants and various diseases.

Scope

- **Data Preprocessing:** Image cleaning, resizing, normalizing, and augmentation using the Plant Village dataset.
- **Model Development:** Exploring four CNN architectures with techniques like batch normalization, dropout, and transfer learning.
- **API Development:** FastAPI will be used to deploy trained models for real-time image classification.
- **Frontend Development:** A React-based application will allow users to upload images and receive predictions.

Dataset Overview

Dataset Description

The dataset used for this project is the **Plant Village Dataset**, specifically focusing on potato diseases. This dataset is well-suited for tasks like image classification in plant pathology, as it contains high-quality images of plant leaves, labeled for various diseases.

- **Dataset Name:** Plant Village Dataset (Potato Disease Subset).

- **Size:** The subset contains 2152 **images** across **3 classes**. These classes include different plant diseases, but for the purpose of this project, we focus on images of **potato leaves** with three specific labels:
    1. **Healthy**
    2. **Early Blight**
    3. **Late Blight**

    Each image represents a close-up shot of a potato leaf, either healthy or affected by one of these two diseases. The large number of images allows for robust training, testing, and validation of the models, reducing the chances of overfitting.

- **Source:** The dataset is publicly available on **Kaggle**. It is one of the most widely used datasets in plant disease classification tasks due to its diversity and quality.

- **Image Format:** All images are colored and provided in JPEG format. They vary in size and quality, necessitating preprocessing steps like resizing and normalization.

- **Classes of Interest:**
    - **Healthy:** These images depict healthy potato leaves without any visible signs of disease.
    - **Early Blight:** Caused by the fungus *Alternaria solani*, Early Blight appears as small dark brown or black spots on the leaves, often surrounded by a yellow halo.

- **Late Blight:** Caused by the oomycete *Phytophthora infestans*, Late Blight manifests as large dark patches on leaves, with a water-soaked appearance. This disease is particularly devastating and can destroy crops rapidly if left unchecked.

- Attributes:
  - **Input**: Images of size 256x256, with 3 color channels (RGB).
  - **Output**: Labels corresponding to disease types.

Feature Selection and Preprocessing Techniques

Feature selection for image data involves the extraction of visual features such as textures, shapes, and colors. In CNNs, this is done automatically by the convolutional layers. The preprocessing steps we applied to the dataset include:

- **Resizing**: All images were resized to a uniform size of 256x256 pixels.
- **Normalization**: Pixel values were scaled to the range [0, 1] to improve model convergence during training.
- **Data Augmentation**: Techniques such as random rotation, flipping, and zooming were applied to artificially expand the dataset and improve model generalization.

Challenges

Despite the benefits of the Plant Village dataset, several challenges arose during model training:

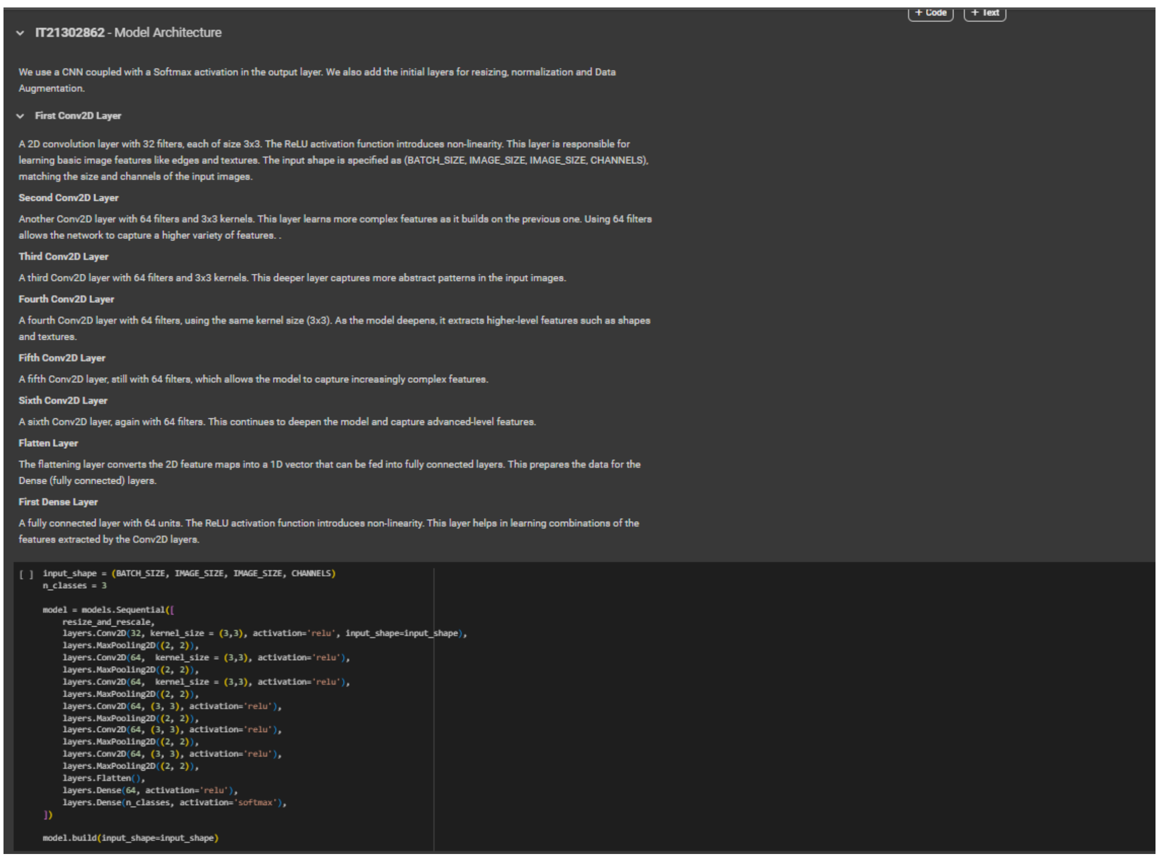
- **Class Imbalance:** The "Healthy" class is overrepresented, leading to biased models. This was addressed with data augmentation and a tuned loss function.
- **Image Quality:** Variations in lighting and resolution affected consistency. Normalization and preprocessing were applied to mitigate these issues.
- **Leaf Occlusion:** Some images had partially occluded leaves or overlapping patterns, complicating the identification of disease symptoms.

Model Architectures

Model 1

This model uses a **Sequential CNN** architecture with six Conv2D layers, which is effective for image classification tasks. CNNs are chosen for their ability to automatically detect important features such as edges, textures, and patterns in images.

- **Layers Chosen**: Multiple Conv2D layers followed by MaxPooling layers enable downsampling and feature extraction. The model flattens before the Dense layers to connect to the final classification task.
- **Activation Functions**: **ReLU** is used for all convolutional layers because of its ability to mitigate the vanishing gradient problem. The **Softmax** function is applied at the output layer for multi-class classification.



+ Code + Text

IT21302862 - Model Architecture

We use a CNN coupled with a Softmax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

First Conv2D Layer

A 2D convolution layer with 32 filters, each of size 3x3. The ReLU activation function introduces non-linearity. This layer is responsible for learning basic image features like edges and textures. The input shape is specified as (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS), matching the size and channels of the input images.

Second Conv2D Layer

Another Conv2D layer with 64 filters and 3x3 kernels. This layer learns more complex features as it builds on the previous one. Using 64 filters allows the network to capture a higher variety of features. .

Third Conv2D Layer

A third Conv2D layer with 64 filters and 3x3 kernels. This deeper layer captures more abstract patterns in the input images.

Fourth Conv2D Layer

A fourth Conv2D layer with 64 filters, using the same kernel size (3x3). As the model deepens, it extracts higher-level features such as shapes and textures.

Fifth Conv2D Layer

A fifth Conv2D layer, still with 64 filters, which allows the model to capture increasingly complex features.

Sixth Conv2D Layer

A sixth Conv2D layer, again with 64 filters. This continues to deepen the model and capture advanced-level features.

Flatten Layer

The flattening layer converts the 2D feature maps into a 1D vector that can be fed into fully connected layers. This prepares the data for the Dense (fully connected) layers.

First Dense Layer

A fully connected layer with 64 units. The ReLU activation function introduces non-linearity. This layer helps in learning combinations of the features extracted by the Conv2D layers.

```
[ ] input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
    n_classes = 3

    model = models.Sequential([
        resize_and_rescale,
        layers.Conv2D(32, kernel_size = (3,3), activation='relu', input_shape=input_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(n_classes, activation='softmax'),
    ])

    model.build(input_shape=input_shape)
```

Model 2

This model is a more complex CNN with five Conv2D layers, using **Adam** as the optimizer for faster convergence.

- **Layers Chosen**: With increasing filter sizes, each convolutional layer captures progressively more complex patterns. MaxPooling downscales feature maps to reduce computational complexity.
- **Activation Functions**: Again, **ReLU** for hidden layers and **Softmax** for the final output layer due to the classification task.

IT21251900 - Model Architecture

This section defines a custom Convolutional Neural Network (CNN) model using the Sequential API. The architecture is built for image classification tasks and includes multiple convolutional layers for feature extraction, followed by dense layers for classification.

Convolutional and MaxPooling Layers

First Conv2D Layer

A 2D convolution layer with 32 filters, each of size 3x3. The ReLU activation function is used to introduce non-linearity. The input shape is set to (224, 224, 3), corresponding to RGB images of size 224x224. A MaxPooling layer with a 2x2 pool size is added to downsample the image.

Second Conv2D Layer

Another Conv2D layer with 64 filters and 3x3 kernels, followed by MaxPooling for further downsampling.

Third Conv2D Layer

A Conv2D layer with 128 filters, followed by MaxPooling to capture more complex features.

Fourth Conv2D Layer

A Conv2D layer with 256 filters, with a MaxPooling layer to continue reducing the spatial dimensions.

Fifth Conv2D Layer

A final Conv2D layer with 512 filters and MaxPooling. This deeper layer captures even more complex features of the input images. python

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model_2 = Sequential()

model_2.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3))) # Changed input_shape to (224, 224, 3)
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(64, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(128, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

# Adjusted the architecture to reduce output volume before flattening
model_2.add(Conv2D(256, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(512, (3, 3), activation='relu')) # Added another Conv2D layer
model_2.add(MaxPooling2D(pool_size=(2, 2))) # Added another MaxPooling2D layer


model_2.add(Flatten())
# This dense layer's input shape is adjusted automatically based on the preceding layer
model_2.add(Dense(256, activation='relu'))
model_2.add(Dropout(0.5))
num_classes = 10  # Replace 10 with the actual number of classes in your dataset
model_2.add(Dense(num_classes, activation='softmax'))
```

Model 3

This model includes Batch Normalization, which stabilizes learning and improves performance in deep architectures.

- **Layers Chosen**: Conv2D layers capture image features, and Batch Normalization ensures stable and faster training.
- **Activation Functions**: **ReLU** for hidden layers, **Softmax** for output.

IT21178054 - Model Architecture

Model Initialization

Initialize a Sequential model to stack layers for the CNN architecture.

First Convolutional Layer

Add a 2D convolution layer with 32 filters of size 3x3. The ReLU activation function introduces non-linearity. The input shape is set to (256, 256, 3), corresponding to RGB images of size 256x256.

Batch Normalization

Normalize the output of the previous layer to stabilize and accelerate training.

Second Convolutional Layer

Add another convolutional layer with 64 filters, followed by batch normalization and max pooling.

Third Convolutional Layer

Add a convolutional layer with 128 filters, followed by batch normalization and max pooling.

Fourth Convolutional Layer

Add a convolutional layer with 256 filters, followed by batch normalization and max pooling.

```
from tensorflow.keras import models, layers
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Dense, Dropout, BatchNormalization, Flatten
from tensorflow.keras.models import Sequential

model_3 = Sequential()

model_3.add(Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(64, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(128, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(256, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Flatten())
# Fully Connected Layer with Dropout for regularization
model_3.add(Dense(128, activation='relu'))
model_3.add(Dropout(0.5))

# Output Layer with Softmax Activation
model_3.add(Dense(len(class_names), activation='softmax'))
```

Model 4

This model focuses on generalization using four convolutional layers and dropout to prevent overfitting.

- **Layers Chosen**: Each Conv2D block is followed by MaxPooling, batch normalization, and Dropout for regularization.
- **Activation Functions**: **ReLU** and **Softmax** for classification.

IT21360428 - Model Architecture

This code defines a convolutional neural network (CNN) using Keras for image classification.

The model consists of four convolutional blocks, each with convolutional, batch normalization, and max pooling layers to extract features from images.

After flattening the feature maps, a fully connected layer with dropout is used for regularization, followed by an output layer with softmax activation for multi-class classification.

The model is designed to preprocess images by resizing and rescaling pixel values to the range [0, 1].

Finally, the model's architecture is summarized to display the layers and parameters.

```python
# Define the image size and number of channels
IMAGE_SIZE = 256
CHANNELS = 3

# You need to manually define class names based on your dataset directory
class_names = ['healthy', 'diseased_class1', 'diseased_class2', 'diseased_class3']  # Modify this according to your dataset

# Set the number of classes
n_classes = len(class_names)

# Define input shape
input_shape = (IMAGE_SIZE, IMAGE_SIZE, CHANNELS)

# Define the resize and rescale layers
resize_and_rescale = tf.keras.Sequential([
    tf.keras.layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),
    tf.keras.layers.Rescaling(1./255),
])

# Define the model
model_4 = models.Sequential([
    resize_and_rescale,  # Use the resize and rescale layer

    # First Convolutional Block
    layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=input_shape),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Second Convolutional Block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Third Convolutional Block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Fourth Convolutional Block
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Flatten the data
    layers.Flatten(),

    # Dense layers with Dropout for regularization
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(n_classes, activation='softmax'),  # Output layer with the correct number of classes
])

# Build and summarize the model
model_4.build(input_shape=(None, IMAGE_SIZE, IMAGE_SIZE, CHANNELS))  # Build the model with the correct input shape
model_4.summary()
```

Results

Model 1

- **Test Accuracy**: Achieved a test accuracy of **97.63%** after 50 epochs..
- **Performance**: The model showed strong performance due to multiple Conv2D layers that captured diverse image features effectively. Loss decreased steadily during training, indicating good model convergence.
- **Challenges**: There was a noticeable improvement in validation accuracy across epochs, with minimal overfitting thanks to the inclusion of data augmentation.

```
history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=EPOCHS,
)
```

```
Epoch 1/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 532s 526ms/step - accuracy: 0.4962 - loss: 0.9273 - val_accuracy: 0.4375 - val_loss: 0.9524
Epoch 2/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.5856 - loss: 0.8384 - val_accuracy: 0.7344 - val_loss: 0.5681
Epoch 3/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 355ms/step - accuracy: 0.7749 - loss: 0.4962 - val_accuracy: 0.8281 - val_loss: 0.4016
Epoch 4/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 349ms/step - accuracy: 0.8187 - loss: 0.4167 - val_accuracy: 0.8542 - val_loss: 0.3371
Epoch 5/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 348ms/step - accuracy: 0.8465 - loss: 0.3436 - val_accuracy: 0.8594 - val_loss: 0.3506
Epoch 6/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 341ms/step - accuracy: 0.8694 - loss: 0.3505 - val_accuracy: 0.8906 - val_loss: 0.3042
Epoch 7/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.8977 - loss: 0.2645 - val_accuracy: 0.9271 - val_loss: 0.2054
Epoch 8/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 364ms/step - accuracy: 0.9181 - loss: 0.2084 - val_accuracy: 0.8854 - val_loss: 0.2818
Epoch 9/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9146 - loss: 0.2282 - val_accuracy: 0.9219 - val_loss: 0.1931
Epoch 10/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9413 - loss: 0.1647 - val_accuracy: 0.9375 - val_loss: 0.1661
Epoch 11/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 371ms/step - accuracy: 0.9316 - loss: 0.1777 - val_accuracy: 0.9062 - val_loss: 0.2162
Epoch 12/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.9248 - loss: 0.1862 - val_accuracy: 0.9479 - val_loss: 0.1295
Epoch 13/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9455 - loss: 0.1391 - val_accuracy: 0.9427 - val_loss: 0.1141
Epoch 14/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 338ms/step - accuracy: 0.9470 - loss: 0.1309 - val_accuracy: 0.9115 - val_loss: 0.2137
Epoch 15/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 18s 339ms/step - accuracy: 0.9382 - loss: 0.1726 - val_accuracy: 0.9531 - val_loss: 0.1038
Epoch 16/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 22s 366ms/step - accuracy: 0.9620 - loss: 0.1110 - val_accuracy: 0.8958 - val_loss: 0.2763
Epoch 17/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 342ms/step - accuracy: 0.9638 - loss: 0.0961 - val_accuracy: 0.9219 - val_loss: 0.2160
Epoch 18/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9608 - loss: 0.1153 - val_accuracy: 0.9375 - val_loss: 0.1026
Epoch 19/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9686 - loss: 0.0827 - val_accuracy: 0.9427 - val_loss: 0.1148
Epoch 20/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9655 - loss: 0.0805 - val_accuracy: 0.9688 - val_loss: 0.0711
Epoch 21/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9736 - loss: 0.0712 - val_accuracy: 0.9583 - val_loss: 0.1084
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9808 - loss: 0.0458 - val_accuracy: 0.9375 - val_loss: 0.1208
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.9823 - loss: 0.0587 - val_accuracy: 0.9531 - val_loss: 0.1119
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 358ms/step - accuracy: 0.9826 - loss: 0.0600 - val_accuracy: 0.8854 - val_loss: 0.2718
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 343ms/step - accuracy: 0.9467 - loss: 0.1361 - val_accuracy: 0.9792 - val_loss: 0.0515
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9846 - loss: 0.0337 - val_accuracy: 0.9844 - val_loss: 0.0379
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 368ms/step - accuracy: 0.9807 - loss: 0.0551 - val_accuracy: 0.9375 - val_loss: 0.1078
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9865 - loss: 0.0360 - val_accuracy: 0.9531 - val_loss: 0.1691
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9779 - loss: 0.0628 - val_accuracy: 0.9479 - val_loss: 0.1723
```

```
scores = model.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - accuracy: 0.9763 - loss: 0.0931
```

Model 2

- **Test Accuracy**: Achieved a test accuracy of **99.36%** after 50 epochs..
- **Performance**: Model 2 performed the best among the four models, likely due to its deeper architecture with five Conv2D layers. The increasing filter sizes and more complex feature extraction likely contributed to this high accuracy.
- **Challenges**: The model required a longer training time due to the larger number of parameters (over 4 million), but this led to significant accuracy improvements.

```
history = model_2.fit(
    train_ds,
    validation_data=val_ds,
    batch_size=BATCH_SIZE,
    verbose=1,
    epochs=EPOCHS
)
```

```
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 286ms/step - accuracy: 0.9799 - loss: 0.0597 - val_accuracy: 0.9740 - val_loss: 0.0461
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 279ms/step - accuracy: 0.9808 - loss: 0.0470 - val_accuracy: 0.9792 - val_loss: 0.0567
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9682 - loss: 0.0766 - val_accuracy: 0.9740 - val_loss: 0.0756
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 292ms/step - accuracy: 0.9828 - loss: 0.0490 - val_accuracy: 0.9635 - val_loss: 0.1160
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 288ms/step - accuracy: 0.9860 - loss: 0.0492 - val_accuracy: 0.9583 - val_loss: 0.1124
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9864 - loss: 0.0384 - val_accuracy: 0.9896 - val_loss: 0.0387
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9848 - loss: 0.0433 - val_accuracy: 0.9792 - val_loss: 0.0473
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9709 - loss: 0.0726 - val_accuracy: 0.9896 - val_loss: 0.0306
Epoch 30/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 283ms/step - accuracy: 0.9872 - loss: 0.0248 - val_accuracy: 0.9792 - val_loss: 0.0636
Epoch 31/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 301ms/step - accuracy: 0.9878 - loss: 0.0374 - val_accuracy: 0.9948 - val_loss: 0.0140
Epoch 32/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 287ms/step - accuracy: 0.9642 - loss: 0.1259 - val_accuracy: 0.9740 - val_loss: 0.0867
Epoch 33/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 283ms/step - accuracy: 0.9767 - loss: 0.0699 - val_accuracy: 0.9844 - val_loss: 0.0419
Epoch 34/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 299ms/step - accuracy: 0.9899 - loss: 0.0292 - val_accuracy: 0.9948 - val_loss: 0.0174
Epoch 35/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 280ms/step - accuracy: 0.9904 - loss: 0.0343 - val_accuracy: 0.9740 - val_loss: 0.1202
Epoch 36/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 281ms/step - accuracy: 0.9886 - loss: 0.0437 - val_accuracy: 0.9792 - val_loss: 0.0355
Epoch 37/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9937 - loss: 0.0252 - val_accuracy: 0.9740 - val_loss: 0.1114
Epoch 38/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9897 - loss: 0.0325 - val_accuracy: 0.9844 - val_loss: 0.0619
Epoch 39/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 15s 280ms/step - accuracy: 0.9915 - loss: 0.0206 - val_accuracy: 0.9844 - val_loss: 0.0249
Epoch 40/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 292ms/step - accuracy: 0.9909 - loss: 0.0225 - val_accuracy: 1.0000 - val_loss: 0.0101
Epoch 41/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 287ms/step - accuracy: 0.9943 - loss: 0.0219 - val_accuracy: 1.0000 - val_loss: 0.0032
Epoch 42/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 289ms/step - accuracy: 0.9934 - loss: 0.0248 - val_accuracy: 0.9948 - val_loss: 0.0105
Epoch 43/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 306ms/step - accuracy: 0.9859 - loss: 0.0413 - val_accuracy: 0.9896 - val_loss: 0.0235
Epoch 44/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 307ms/step - accuracy: 0.9812 - loss: 0.0545 - val_accuracy: 0.9948 - val_loss: 0.0124
Epoch 45/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 304ms/step - accuracy: 0.9904 - loss: 0.0263 - val_accuracy: 0.9896 - val_loss: 0.0250
Epoch 46/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 296ms/step - accuracy: 0.9870 - loss: 0.0259 - val_accuracy: 0.9896 - val_loss: 0.0175
Epoch 47/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 302ms/step - accuracy: 0.9970 - loss: 0.0095 - val_accuracy: 0.9844 - val_loss: 0.0425
Epoch 48/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 306ms/step - accuracy: 0.9923 - loss: 0.0229 - val_accuracy: 0.9948 - val_loss: 0.0212
Epoch 49/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 298ms/step - accuracy: 0.9925 - loss: 0.0142 - val_accuracy: 1.0000 - val_loss: 0.0043
Epoch 50/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 298ms/step - accuracy: 0.9991 - loss: 0.0042 - val_accuracy: 0.9948 - val_loss: 0.0070
```

```
scores_2 = model_2.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 7s 28ms/step - accuracy: 0.9936 - loss: 0.0515
```

Model 3

- **Test Accuracy**: Achieved a test accuracy of **76.42%** after 50 epochs.
- **Performance**: Although the architecture was simpler than Model 2, the inclusion of Batch Normalization improved training stability. However, the performance was lower compared to Model 2, likely due to underfitting or insufficient feature extraction in the deeper layers.
- **Challenges**: The model struggled to generalize on the validation set, suggesting that additional tuning or a more complex architecture might be required for better performance.

```
history = model_3.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=EPOCHS,
)
```

```
Epoch 1/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 50s 549ms/step - accuracy: 0.8021 - loss: 4.8974 - val_accuracy: 0.4479 - val_loss: 40.0144
Epoch 2/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 349ms/step - accuracy: 0.9085 - loss: 0.5371 - val_accuracy: 0.4479 - val_loss: 68.2467
Epoch 3/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 355ms/step - accuracy: 0.9119 - loss: 0.3546 - val_accuracy: 0.4479 - val_loss: 80.0459
Epoch 4/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 368ms/step - accuracy: 0.9231 - loss: 0.3743 - val_accuracy: 0.4479 - val_loss: 69.6785
Epoch 5/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 349ms/step - accuracy: 0.9275 - loss: 0.2420 - val_accuracy: 0.4479 - val_loss: 54.5098
Epoch 6/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 383ms/step - accuracy: 0.9277 - loss: 0.2703 - val_accuracy: 0.4479 - val_loss: 42.5919
Epoch 7/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9211 - loss: 0.2676 - val_accuracy: 0.4479 - val_loss: 27.7518
Epoch 8/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9479 - loss: 0.1801 - val_accuracy: 0.6615 - val_loss: 7.9593
Epoch 9/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 358ms/step - accuracy: 0.9516 - loss: 0.1606 - val_accuracy: 0.6875 - val_loss: 5.5214
Epoch 10/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 349ms/step - accuracy: 0.9385 - loss: 0.1836 - val_accuracy: 0.8906 - val_loss: 1.6778
Epoch 11/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 361ms/step - accuracy: 0.9437 - loss: 0.2077 - val_accuracy: 0.9323 - val_loss: 0.2869
Epoch 12/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9604 - loss: 0.1363 - val_accuracy: 0.6406 - val_loss: 3.1695
Epoch 13/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 381ms/step - accuracy: 0.9595 - loss: 0.1359 - val_accuracy: 0.8385 - val_loss: 1.7539
Epoch 14/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 345ms/step - accuracy: 0.9399 - loss: 0.2416 - val_accuracy: 0.8125 - val_loss: 0.8085
Epoch 15/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9271 - loss: 0.2667 - val_accuracy: 0.7083 - val_loss: 2.9910
Epoch 16/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 365ms/step - accuracy: 0.9397 - loss: 0.2280 - val_accuracy: 0.9115 - val_loss: 0.4364
Epoch 17/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 342ms/step - accuracy: 0.9693 - loss: 0.0969 - val_accuracy: 0.8542 - val_loss: 0.8443
Epoch 18/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 342ms/step - accuracy: 0.9244 - loss: 0.2487 - val_accuracy: 0.5729 - val_loss: 5.1617
Epoch 19/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 374ms/step - accuracy: 0.9578 - loss: 0.1347 - val_accuracy: 0.6510 - val_loss: 20.7774
Epoch 20/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 343ms/step - accuracy: 0.9413 - loss: 0.1946 - val_accuracy: 0.6875 - val_loss: 1.9151
Epoch 21/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 360ms/step - accuracy: 0.9326 - loss: 0.1797 - val_accuracy: 0.5990 - val_loss: 2.8666
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 352ms/step - accuracy: 0.9514 - loss: 0.1782 - val_accuracy: 0.6510 - val_loss: 3.9134
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 359ms/step - accuracy: 0.9634 - loss: 0.1230 - val_accuracy: 0.6146 - val_loss: 5.8954
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 361ms/step - accuracy: 0.9430 - loss: 0.2192 - val_accuracy: 0.8906 - val_loss: 0.6860
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.9420 - loss: 0.1821 - val_accuracy: 0.4792 - val_loss: 30.7310
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 358ms/step - accuracy: 0.9606 - loss: 0.1245 - val_accuracy: 0.4896 - val_loss: 27.4846
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 347ms/step - accuracy: 0.9555 - loss: 0.1299 - val_accuracy: 0.9271 - val_loss: 0.2916
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 348ms/step - accuracy: 0.9656 - loss: 0.1744 - val_accuracy: 0.8385 - val_loss: 2.2625
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 367ms/step - accuracy: 0.9691 - loss: 0.1232 - val_accuracy: 0.4844 - val_loss: 52.6334
```

```
test_loss, test_acc = model_3.evaluate(test_ds)
print(f"Test accuracy: {test_acc}")
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.7642 - loss: 9.8739
Test accuracy: 0.73828125
```

Model 4

- **Training Accuracy**: Achieved a test accuracy of **97.33%** after 50 epochs.
- **Performance**: This model showed potential but demonstrated underfitting, with the validation accuracy lagging behind the training accuracy by a significant margin. This suggests that either the model requires more epochs to train or more advanced data augmentation techniques to improve generalization.
- **Challenges**: The model's simplicity compared to Models 1 and 2 may have contributed to its lower accuracy. Additionally, it might have benefitted from more complex feature extraction or regularization methods to reduce overfitting.

```python
# Train the model
history_4 = model_4.fit(
    train_ds,
    epochs=EPOCHS,
    validation_data=val_ds,
    verbose=1
)
```

```
Epoch 2/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 79ms/step - accuracy: 0.9332 - loss: 2.3844 - val_accuracy: 0.4479 - val_loss: 229.3153
Epoch 3/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 80ms/step - accuracy: 0.9442 - loss: 0.9710 - val_accuracy: 0.4479 - val_loss: 249.7777
Epoch 4/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 79ms/step - accuracy: 0.9602 - loss: 0.5993 - val_accuracy: 0.4479 - val_loss: 234.0843
Epoch 5/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 82ms/step - accuracy: 0.9463 - loss: 1.1435 - val_accuracy: 0.4479 - val_loss: 235.9537
Epoch 6/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 81ms/step - accuracy: 0.9737 - loss: 0.3235 - val_accuracy: 0.4479 - val_loss: 185.5254
Epoch 7/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 81ms/step - accuracy: 0.9805 - loss: 0.2078 - val_accuracy: 0.4479 - val_loss: 211.8311
Epoch 8/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9734 - loss: 0.5782 - val_accuracy: 0.7917 - val_loss: 32.7511
Epoch 9/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9677 - loss: 0.3998 - val_accuracy: 0.4948 - val_loss: 55.7125
Epoch 10/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9743 - loss: 0.2798 - val_accuracy: 0.9375 - val_loss: 4.2887
Epoch 11/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9597 - loss: 0.8631 - val_accuracy: 0.9635 - val_loss: 1.3291
Epoch 12/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 86ms/step - accuracy: 0.9697 - loss: 0.8062 - val_accuracy: 0.9635 - val_loss: 1.8586
Epoch 13/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9635 - loss: 0.6652 - val_accuracy: 0.5312 - val_loss: 65.6163
Epoch 14/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9697 - loss: 0.7214 - val_accuracy: 0.8021 - val_loss: 6.6685
Epoch 15/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9797 - loss: 0.4578 - val_accuracy: 0.9375 - val_loss: 3.1483
Epoch 16/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9826 - loss: 0.3913 - val_accuracy: 0.7292 - val_loss: 33.0342
Epoch 17/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9783 - loss: 0.3822 - val_accuracy: 0.8958 - val_loss: 3.0845
Epoch 18/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9803 - loss: 0.3253 - val_accuracy: 0.5312 - val_loss: 76.3252
Epoch 19/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9844 - loss: 0.3548 - val_accuracy: 0.9583 - val_loss: 0.7972
Epoch 20/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9774 - loss: 0.2371 - val_accuracy: 0.4792 - val_loss: 743.0941
Epoch 21/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9895 - loss: 0.2723 - val_accuracy: 0.5365 - val_loss: 86.7308
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9899 - loss: 0.2314 - val_accuracy: 0.5573 - val_loss: 48.1991
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9954 - loss: 0.0331 - val_accuracy: 0.9167 - val_loss: 2.4063
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9867 - loss: 0.2074 - val_accuracy: 0.9792 - val_loss: 0.7489
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9805 - loss: 0.3205 - val_accuracy: 0.9844 - val_loss: 0.0877
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9942 - loss: 0.0594 - val_accuracy: 0.8802 - val_loss: 3.7508
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9823 - loss: 0.2125 - val_accuracy: 0.8594 - val_loss: 17.5817
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9878 - loss: 0.2666 - val_accuracy: 0.9115 - val_loss: 2.6445
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9722 - loss: 0.5200 - val_accuracy: 0.5365 - val_loss: 38.9405
Epoch 30/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9820 - loss: 0.3599 - val_accuracy: 0.9792 - val_loss: 0.1090
Epoch 31/50
```

```python
scores = model_4.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 6s 28ms/step - accuracy: 0.9733 - loss: 0.9552
```

Comparison of Results

Model 2 achieved the highest test accuracy of **99.36%**, making it the best performer. Its deeper architecture with five Conv2D layers and more complex feature extraction likely contributed to this superior performance. Despite the larger number of parameters and longer training time, it delivered the most accurate results.

Model 1 followed closely with a test accuracy of **97.63%**. It showed strong performance with minimal overfitting, due to effective data augmentation techniques. Its six Conv2D layers allowed it to capture diverse image features efficiently.

Model 4 achieved a test accuracy of **97.33%**, slightly lower than Model 1. This model demonstrated potential but exhibited signs of underfitting, where the validation accuracy lagged behind the training accuracy. It may have benefitted from more training epochs or advanced data augmentation to improve generalization.

Model 3 had the test accuracy of **76.42% which is lower than comparing with other 3 models**. While Batch Normalization improved training stability, the simpler architecture resulted in insufficient feature extraction, leading to underfitting. This model struggled to generalize well compared to the others, indicating the need for architectural improvements or additional tuning.

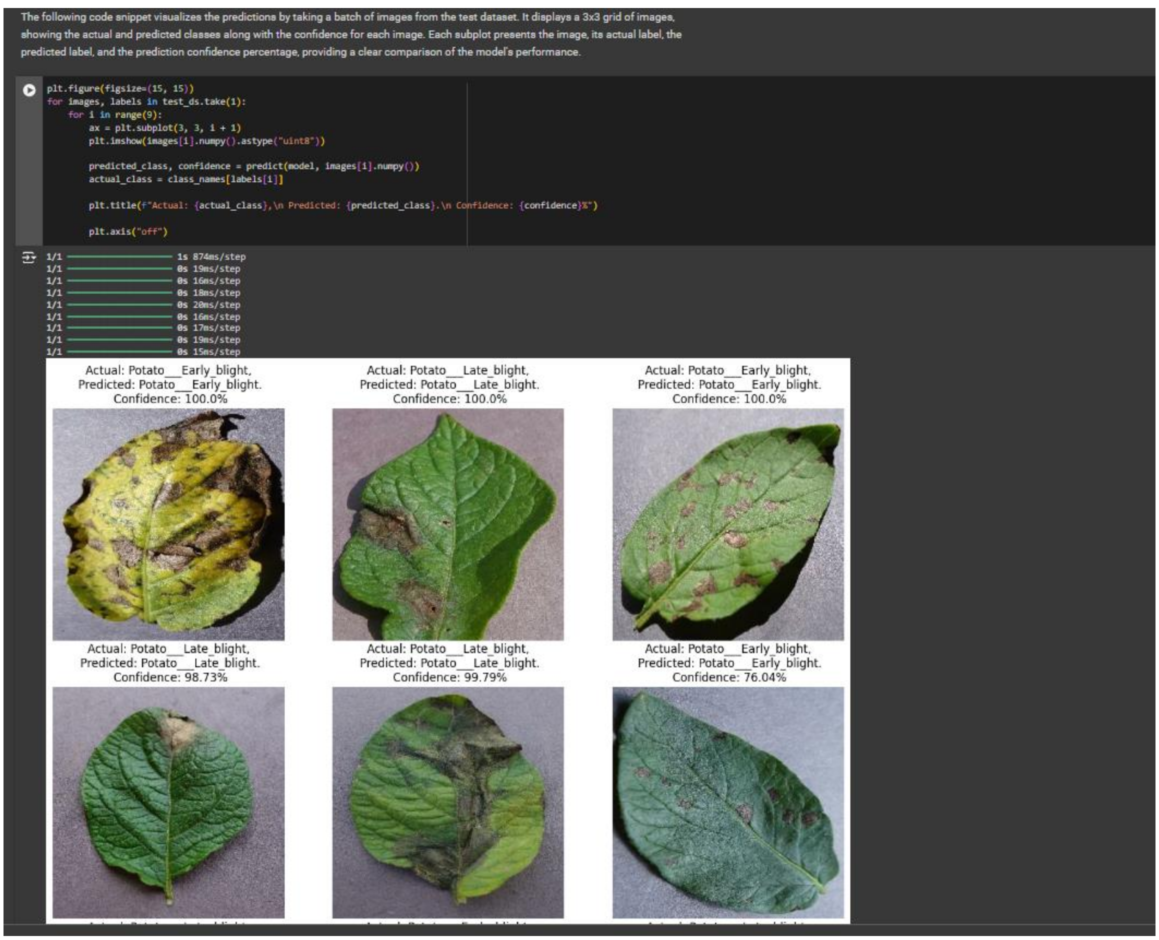
Training & Validation Accuracy and Loss

- **Model Performance**: It helps assess how well the model generalizes to unseen data.

- **Overfitting Detection**: A large gap between training and validation metrics signals overfitting, where the model performs well on training data but poorly on new data.

- **Hyperparameter Tuning**: Tracking these metrics informs adjustments to improve model performance, such as tuning learning rate or batch size.

- **Early Stopping**: Monitoring allows for early stopping to prevent overfitting and save computational resources.

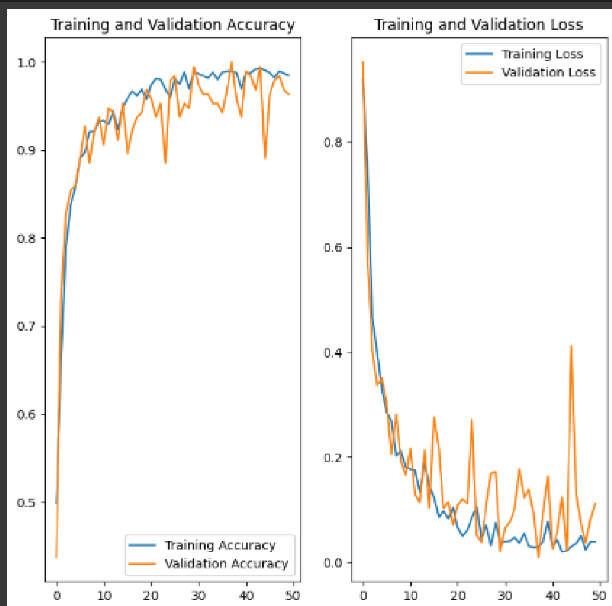


Model 1

Plotting Training and Validation Accuracy

The first subplot displays the training accuracy and validation accuracy over epochs. The x-axis represents the number of epochs, while the y-axis represents accuracy values. `acc` refers to the list of training accuracy values, and `val_acc` refers to the validation accuracy values.

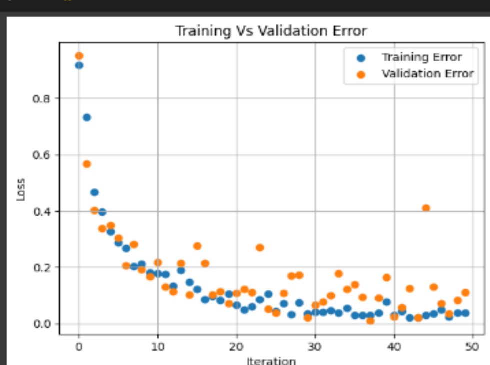
```
[ ] plt.figure(figsize=(8, 8))
    plt.subplot(1, 2, 1)
    plt.plot(range(len(acc)), acc, label='Training Accuracy')
    plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')

    plt.subplot(1, 2, 2)
    plt.plot(range(len(loss)), loss, label='Training Loss')
    plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.show()
```



Scatter Plot: Training vs Validation Error

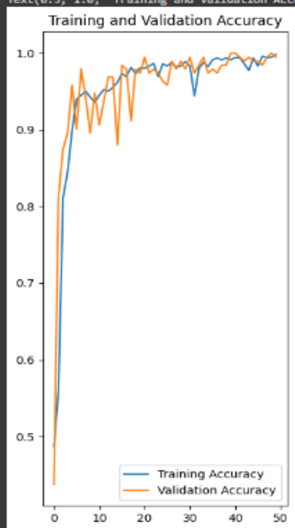
This plot visualizes the loss (error) during training and validation across different iterations (epochs). The x-axis represents the number of epochs, while the y-axis shows the loss values. `history.history['loss']` contains the training loss values, and `history.history['val_loss']` contains the validation loss. The scatter plot allows easy comparison between training and validation errors to assess how well the model is generalizing.

```
[ ] plt.scatter(x=history.epoch,y=history.history['loss'],label='Training Error')
    plt.scatter(x=history.epoch,y=history.history['val_loss'],label='Validation Error')
    plt.grid(True)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Training Vs Validation Error')
    plt.legend()
    plt.show()
```

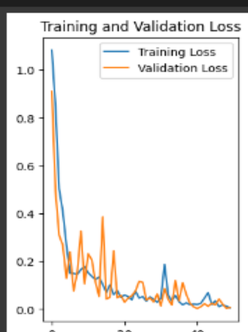


Model 2

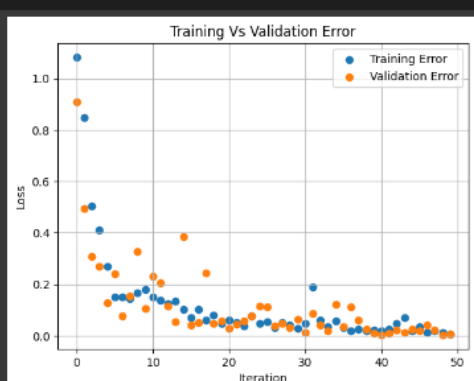
```
Text(0.5, 1.0, 'Training and Validation Accuracy')
```




```
plt.subplot(1, 2, 2)
plt.plot(range(len(loss)), loss, label='Training Loss')
plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

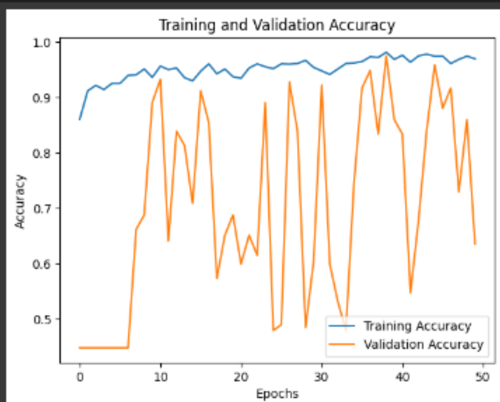


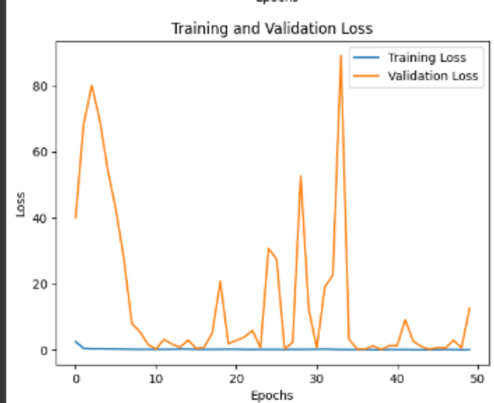

```
plt.scatter(x=history.epoch,y=history.history['loss'],label='Training Error')
plt.scatter(x=history.epoch,y=history.history['val_loss'],label='Validation Error')
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Vs Validation Error')
plt.legend()
plt.show()
```



Model 3

```python
# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.show()

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



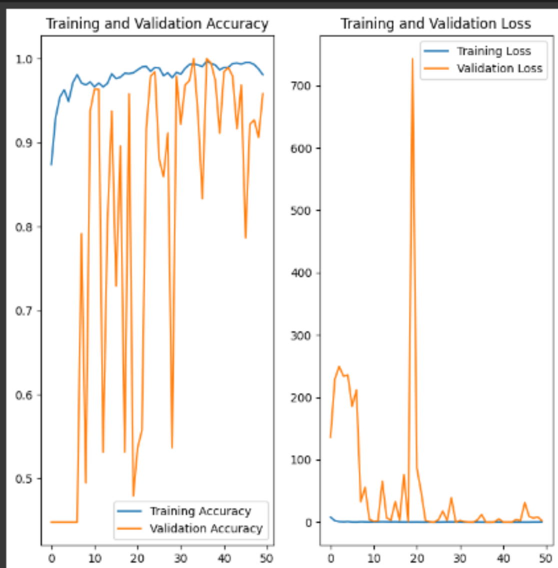


Model 4

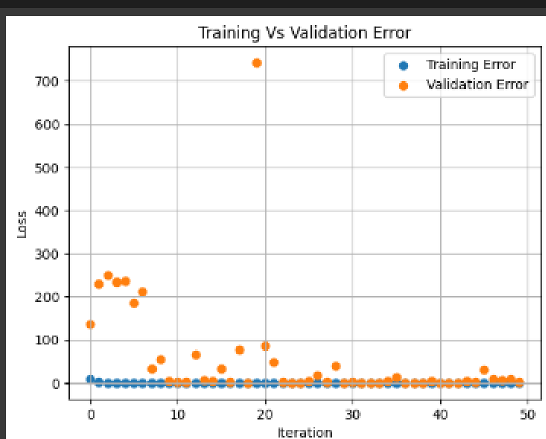
Plotting Training and Validation Accuracy

The first subplot displays the training accuracy and validation accuracy over epochs. The x-axis represents the number of epochs, while the y-axis represents accuracy values. `acc` refers to the list of training accuracy values, and `val_acc` refers to the validation accuracy values.

```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(len(loss)), loss, label='Training Loss')
plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```




```
plt.scatter(x=history_4.epoch,y=history_4.history['loss'],label='Training Error')
plt.scatter(x=history_4.epoch,y=history_4.history['val_loss'],label='Validation Error')
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Vs Validation Error')
plt.legend()
plt.show()
```



Confusion Matrix

Visualizing training and validation metrics plays a crucial role in understanding how a machine learning model evolves during training. To ensure effective learning and avoid overfitting, it's essential to monitor these metrics closely through graphical representations.

In this project, Matplotlib was used to create detailed visualizations that illustrate training accuracy, validation accuracy, training loss, and validation loss over time. These plots provide a clear visual summary of the model's performance, making it easier to spot trends, identify overfitting or underfitting, and understand the impact of hyperparameter adjustments.

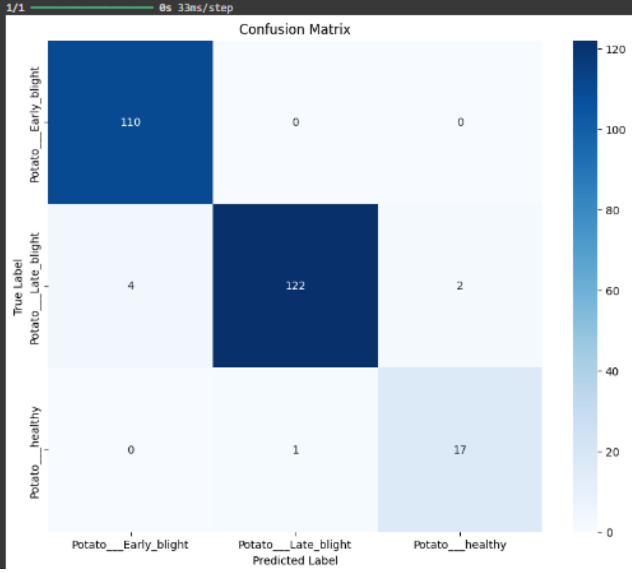
All team members relied on these visualizations to assess and improve the performance of their individual models designed to classify potato leaf diseases. The graphical representations served as an indispensable tool for evaluating model convergence, guiding model adjustments, and ensuring that each model maintained a healthy balance between accuracy and generalization.

Model 1

- **Potato Early Blight**: Predicted 110 correctly, 4 misclassified as Potato Late Blight, 0 misclassified as Healthy.
- **Potato Late Blight**: Predicted 122 correctly, 2 misclassified as Healthy, 0 misclassified as Early Blight.
- **Potato Healthy**: Predicted 17 correctly, 1 misclassified as Late Blight.

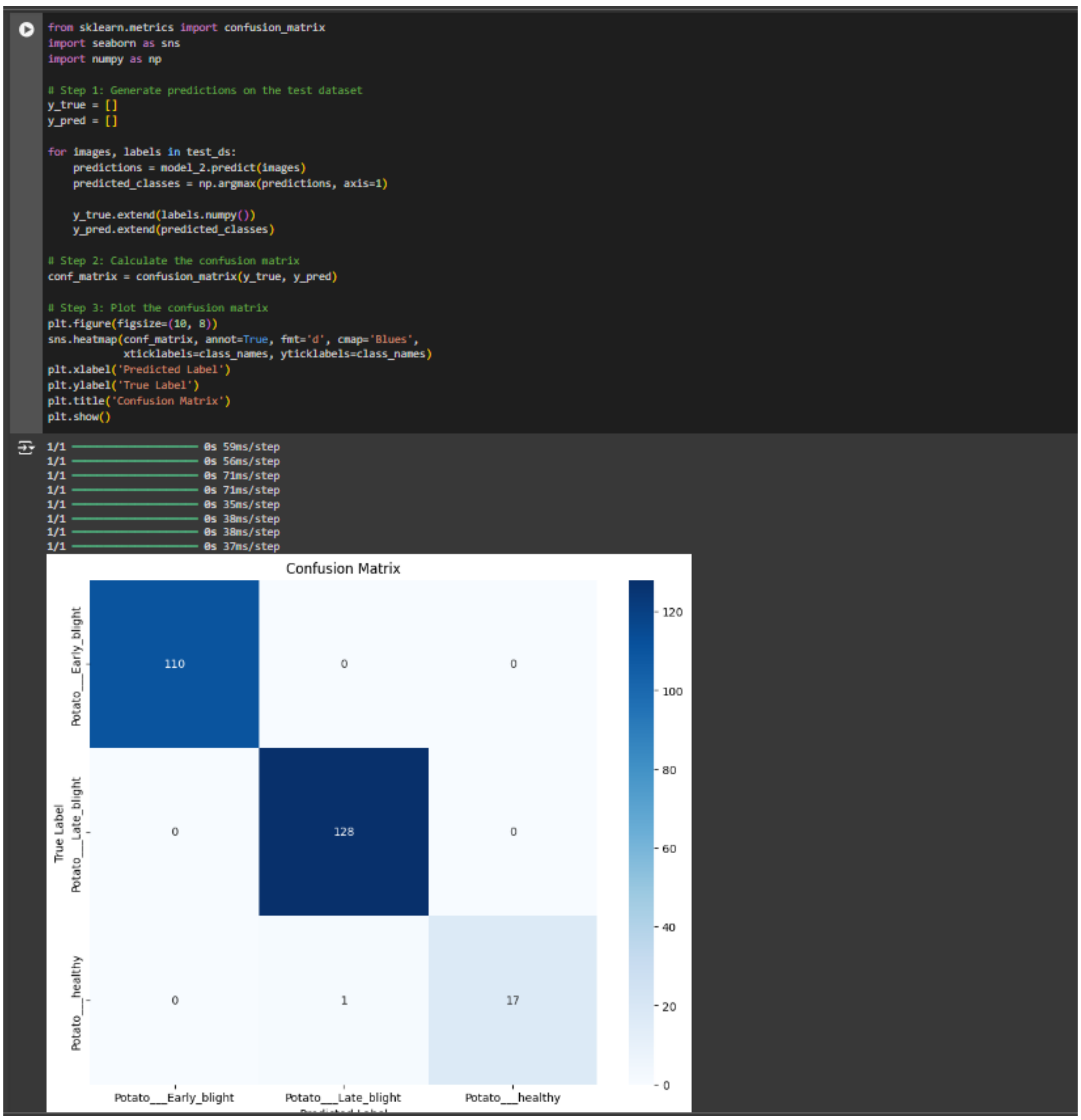
```python
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Step 1: Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)

    y_true.extend(labels.numpy())
    y_pred.extend(predicted_classes)

# Step 2: Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Step 3: Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 427ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
```



Model 2

- **Potato Early Blight**: Predicted 110 correctly, 0 misclassified as Potato Late Blight, 0 misclassified as Healthy.
- **Potato Late Blight**: Predicted 128 correctly, 1 misclassified as Healthy, 0 misclassified as Early Blight.
- **Potato Healthy**: Predicted 17 correctly, 1 misclassified as Late Blight.

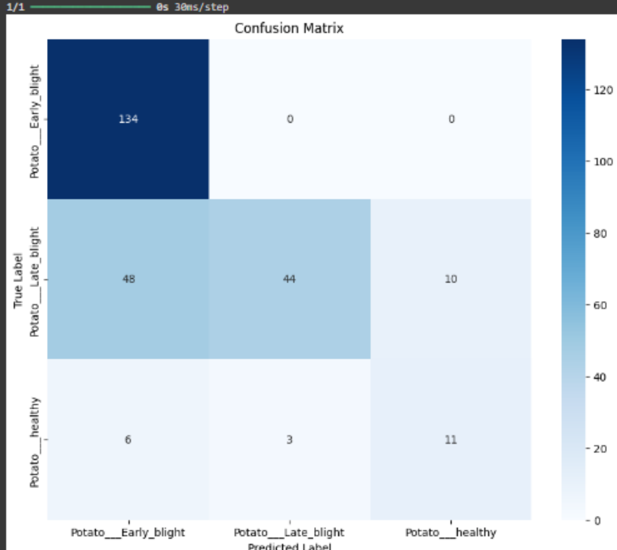


Model 3

- **Potato Early Blight**: Predicted 134 correctly, 48 misclassified as Potato Late Blight, 6 misclassified as Healthy.
- **Potato Late Blight**: Predicted 44 correctly, 10 misclassified as Healthy, 48 misclassified as Early Blight.
- **Potato Healthy**: Predicted 11 correctly, 3 misclassified as Late Blight, 6 misclassified as Early Blight.

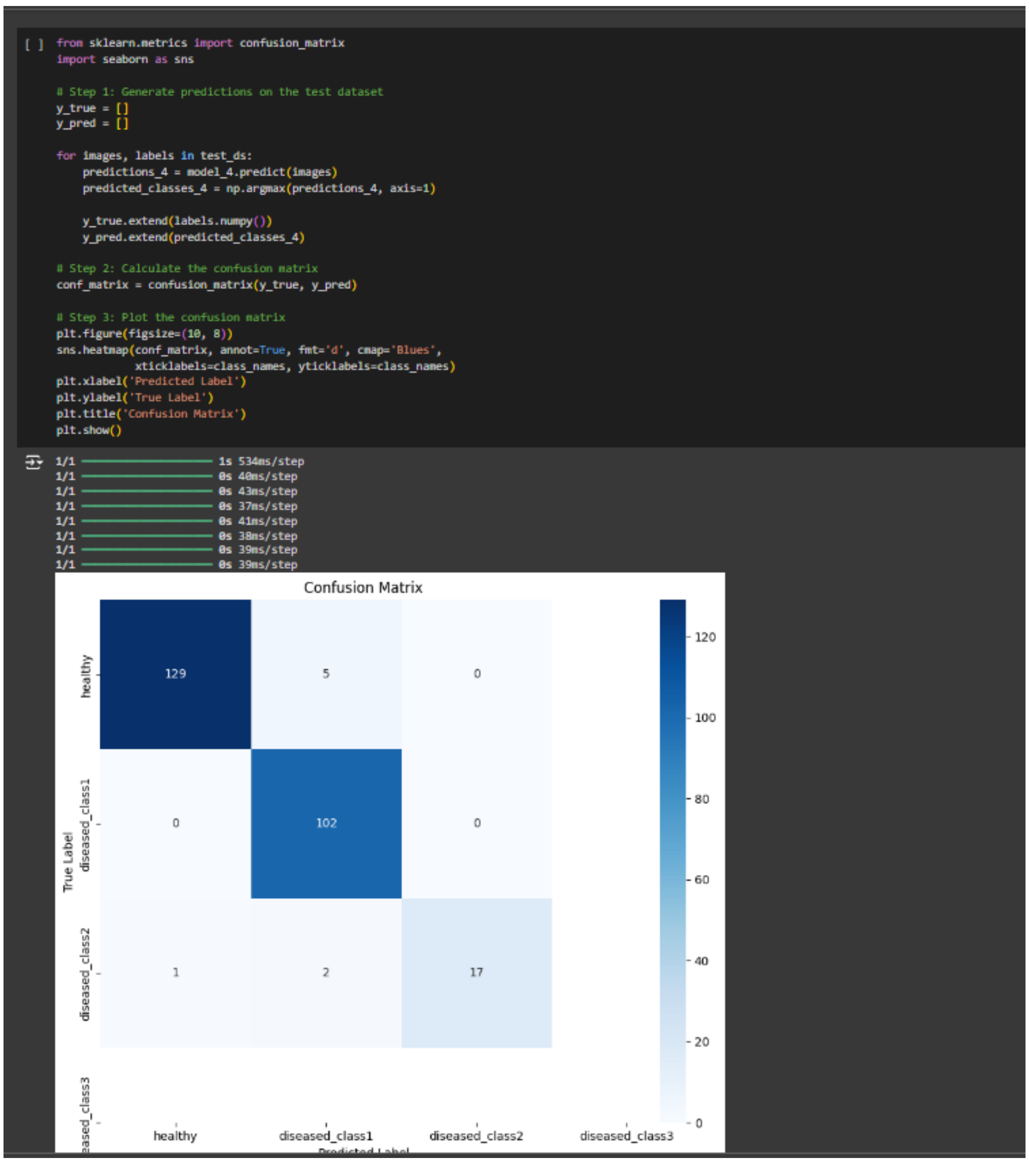
```
# Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model_3.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)
    # Convert labels to multiclass format if they are in multilabel-indicator format
    true_classes = np.argmax(labels.numpy(), axis=1) if labels.ndim > 1 else labels.numpy()
    y_true.extend(true_classes)
    y_pred.extend(predicted_classes)

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 605ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
```



Model 4

- **Potato Early Blight**: Predicted 129 correctly, 5 misclassified as Potato Late Blight, 0 misclassified as Healthy.
- **Potato Late Blight**: Predicted 102 correctly, 0 misclassified as Healthy, 0 misclassified as Early Blight.
- **Potato Healthy**: Predicted 17 correctly, 2 misclassified as Late Blight, 1 misclassified as Early Blight.



Overall Observation

- **Model 1** and **Model 2** performed consistently well, with minimal misclassifications across all classes.
- **Model 3** struggled significantly with Potato Late Blight predictions, misclassifying many cases as Early Blight.
- **Model 4** had generally good performance, though it had some misclassifications for Early Blight and Late Blight.

In conclusion, **Model 2** seems to offer the most balanced performance across all categories, while **Model 3** may need further refinement to handle the Late Blight class more effectively.

Critical Analysis and Discussion

How Accuracy Could Be Improved

- **Hyperparameter Tuning**: Adjusting learning rates, batch sizes, and the number of epochs could further improve accuracy. For instance, increasing the number of epochs for Model 4 could help prevent underfitting.

- **More Data Augmentation**: Increasing the variety of augmentations such as zoom, brightness, and contrast adjustments could help improve generalization, especially for underperforming models like Model 4.

- **Regularization**: Implementing techniques like **L2 regularization** could help prevent overfitting in high-capacity models like Model 2.

- **Pre-trained Models**: Using a pre-trained architecture such as **VGG16** or **ResNet** could improve performance, especially in complex classification tasks with limited training data.

Possible Future Work

- **Transfer Learning**: Implementing transfer learning with pre-trained models could yield higher accuracy with fewer training epochs.
- **Ensemble Methods**: Combining predictions from multiple models in an ensemble could improve overall performance by leveraging the strengths of each individual model.
- **Optimization Algorithms**: Experimenting with different optimizers such as **RMSprop** or **SGD** could fine-tune model performance and convergence.
- **Further Hyperparameter Optimization**: Automated techniques like **Grid Search** or **Random Search** could be used to systematically explore a wider range of hyperparameters.

Potato Disease Identification API

Overview

The **Potato Disease Identification API** is a web-based API developed using **FastAPI**. This API utilizes a machine learning model to identify diseases in potato leaves by analyzing uploaded images. The API can classify potato leaf images into three categories: **Early Blight**, **Late Blight**, or **Healthy**. The application is designed to provide quick and accurate predictions, assisting farmers and agricultural professionals in managing crop health.

How It Works

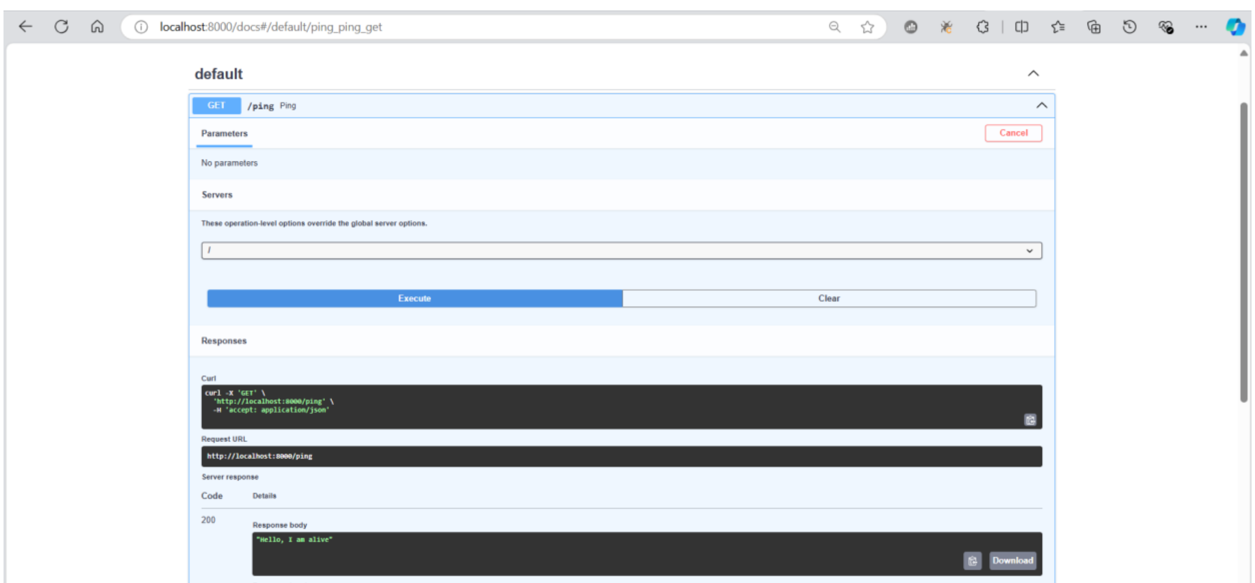
1. **Image Upload**: Users can upload an image of a potato leaf to the `/predict` endpoint.
2. **Preprocessing**: The uploaded image is resized and normalized to meet the input requirements of the machine learning model.
3. **Model Prediction**: A pre-trained TensorFlow model processes the image and predicts the class of disease or indicates that the plant is healthy.
4. **Response**: The API returns the predicted disease class along with a confidence score.

API Endpoints

1. Health Check

- **Endpoint**: `/ping`
- **Method**: `GET`
- **Description**: A simple health check endpoint to verify if the API is running.

Response:



2. Predict Disease

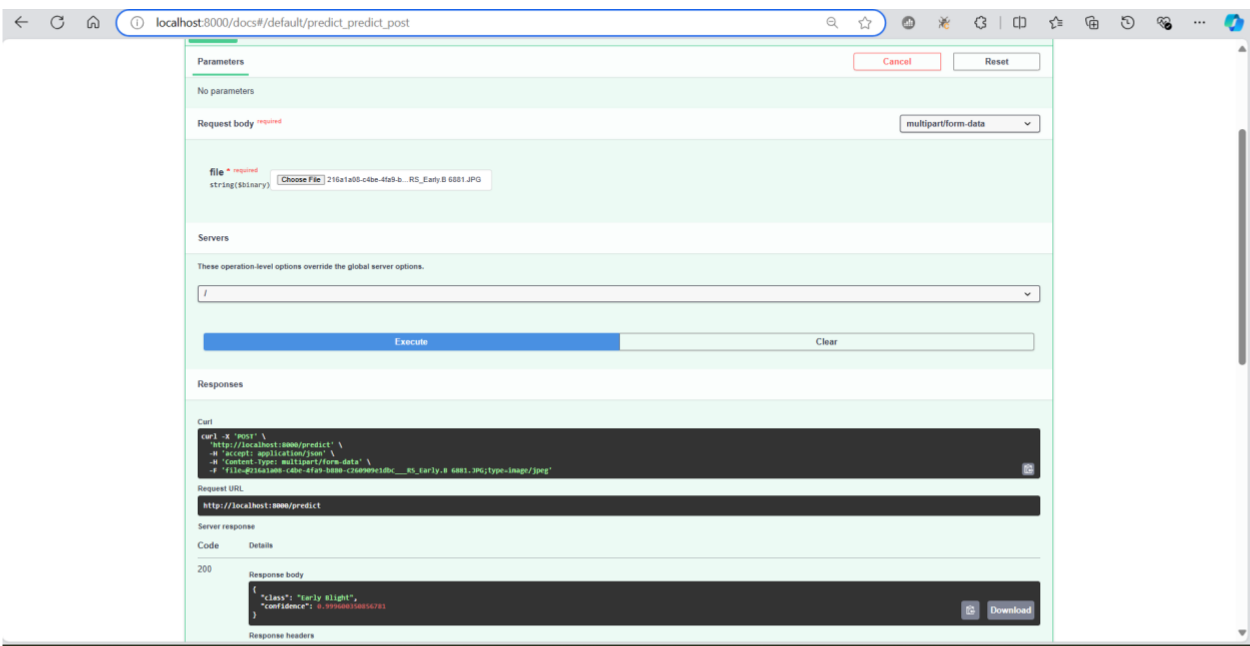
- **Endpoint**: `/predict`
- **Method**: `POST`
- **Description**: Upload an image of a potato leaf to receive a prediction about its health status.

Request:

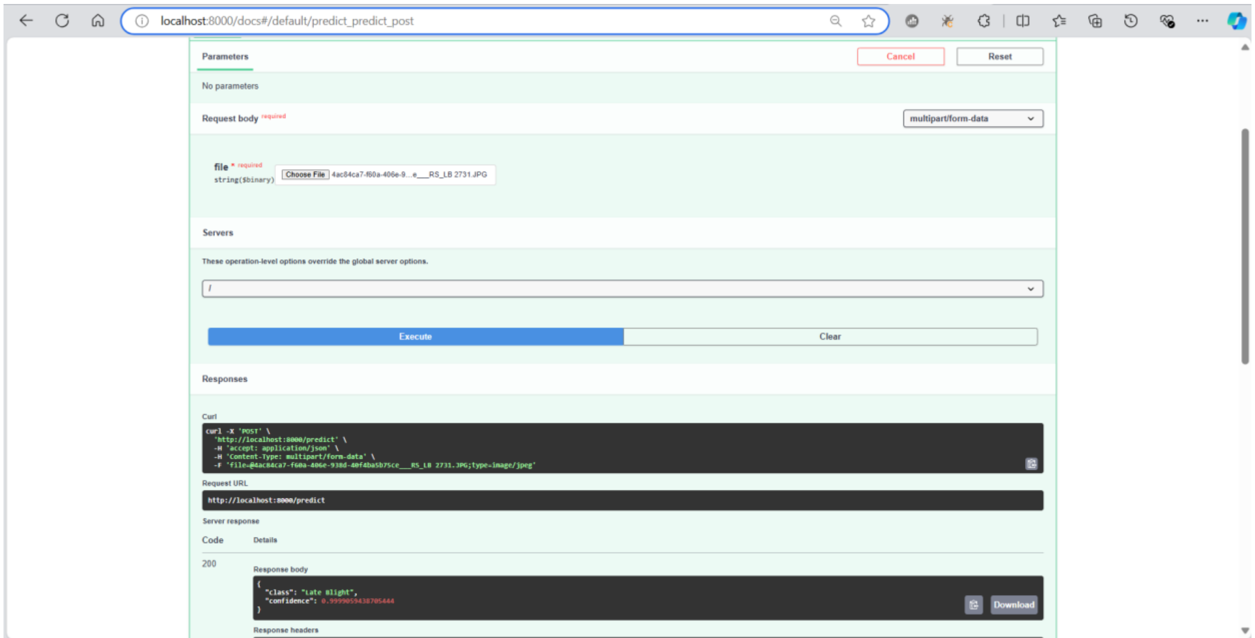
- **Form Data**:
  - `file`: The image file of the potato leaf (must be in a supported image format).

Response:

Early Blight Disease Identification.



Late Blight Disease Identification



React Application

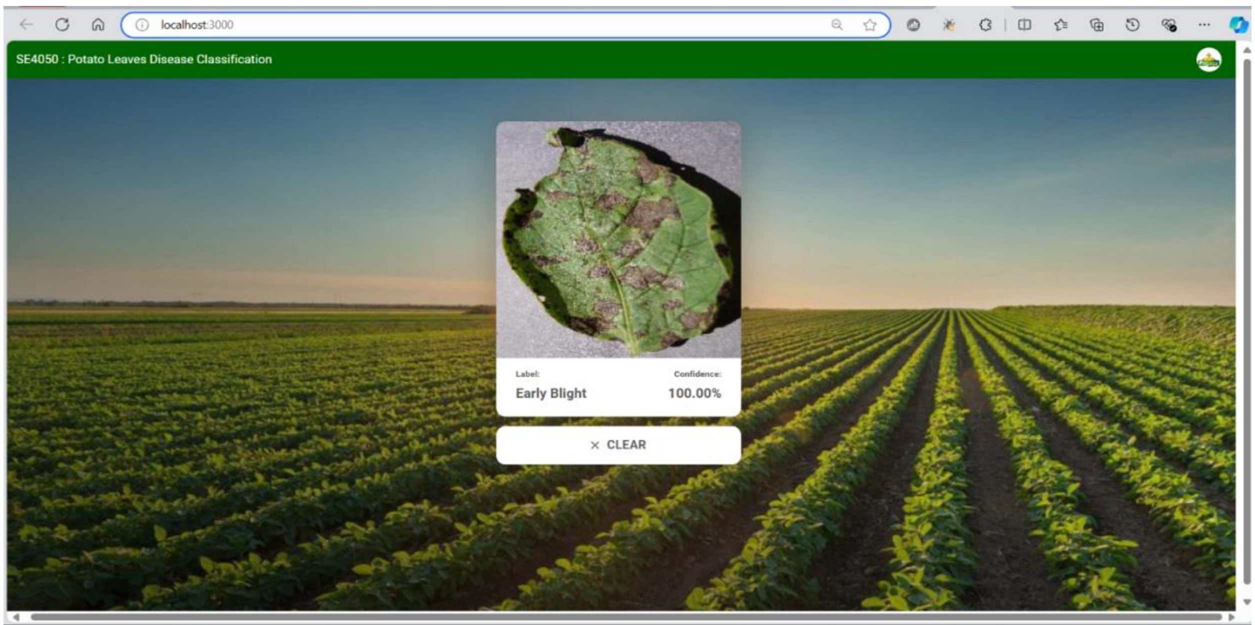
The React application interacts with the FastAPI to facilitate the image upload process and display the results. Below are the key functionalities implemented in the React app:

Features

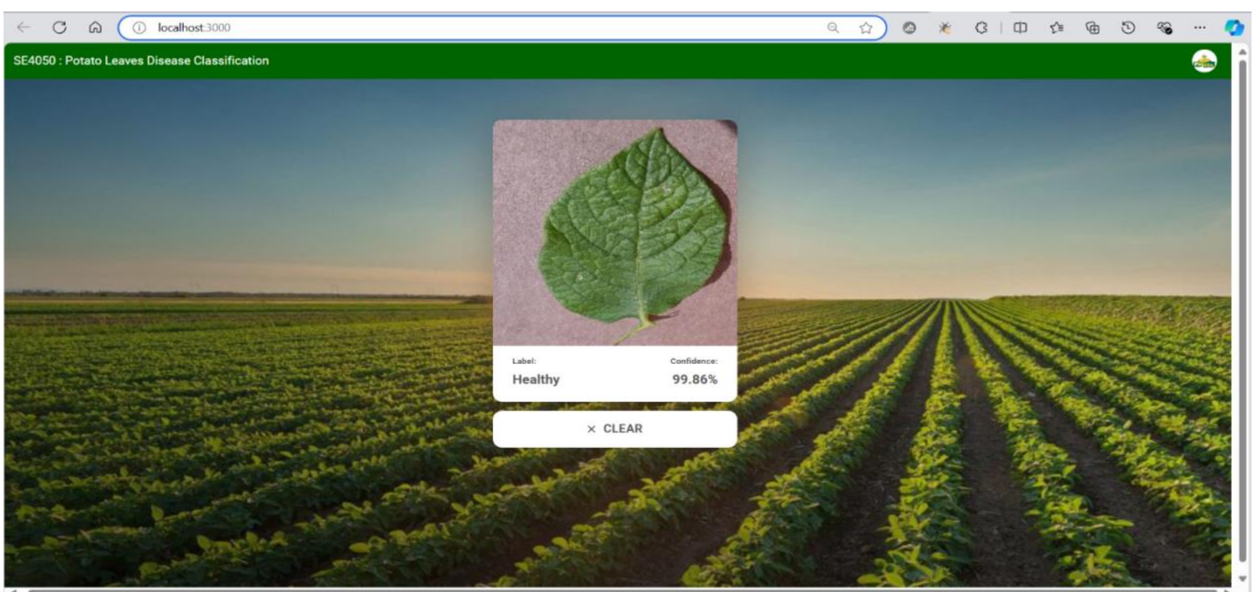
1. **User Interface**:
   - The app includes a user-friendly interface for uploading images of potato leaves.
2. **API Calls**:
   - The React application makes asynchronous calls to the FastAPI endpoints using **Axios** or **Fetch API**.
   - When the user uploads an image, the app sends a POST request to the /predict endpoint.

3. **Displaying Results**:
   - Upon receiving the response from the API, the React application displays the predicted class and confidence score to the user.
   - The results are shown in a structured format, allowing users to understand the health status of their potato plants at a glance.

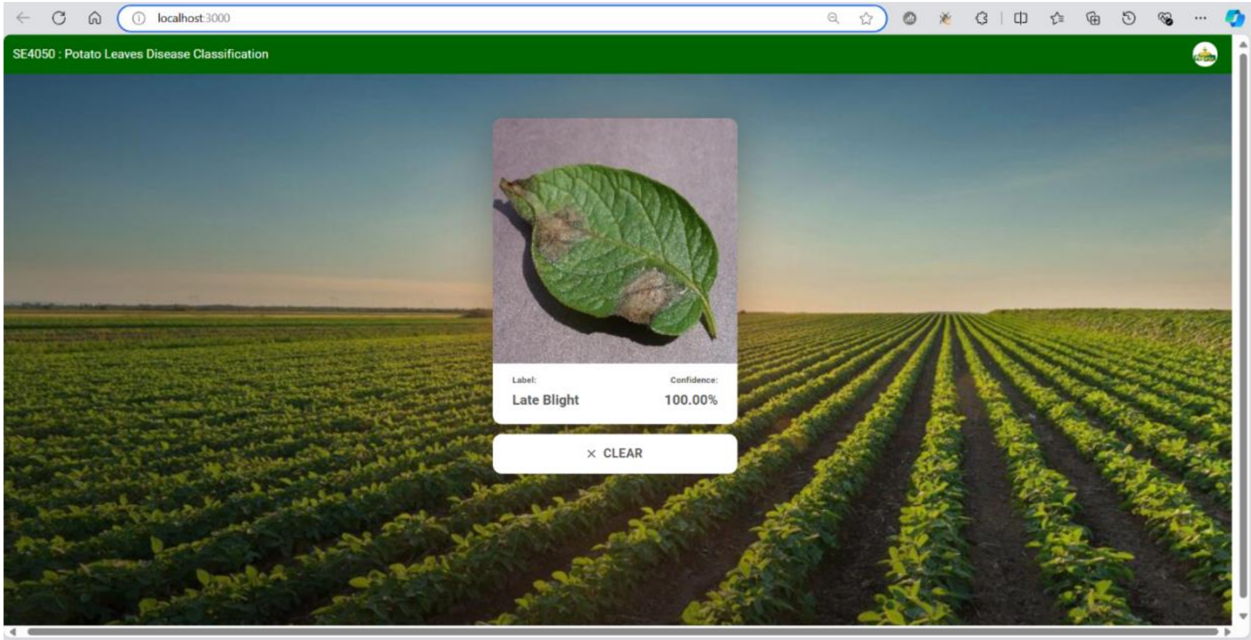
Early Blight Disease Identification.



Healthy Plants Identification



Late Blight Disease Identififcation



Workflow

1. **Upload Image**: The user selects a potato leaf image and clicks the "Upload" button.
2. **Send Request**: The React app sends the image to the /predict endpoint.
3. **Receive Response**: The API processes the image and returns the prediction.
4. **Display Prediction**: The React application displays the predicted disease class and confidence score.

To run the application

1. run the backend server(FasterAPI - potato-disease-application-python-api)
2. set the environment

```
$env:NODE_OPTIONS="--openssl-legacy-provider"
```

3. Install the dependencies

```
npm install

npm audit fix
```

Conclusion

In this study of potato disease identification using various CNN architectures, **Model 2** demonstrated the highest accuracy at **99.36%**, showcasing the effectiveness of deeper architectures and complex feature extraction. **Model 1** and **Model 4** also performed well with accuracies of **97.63%** and **97.33%**, respectively, although Model 4 exhibited signs of underfitting. In contrast, **Model 3** underperformed with a test accuracy of **76.42%**, indicating a need for further architectural enhancement. Overall, these results highlight the importance of model complexity and the balance between training efficiency and generalization in achieving high performance in image classification tasks.

References

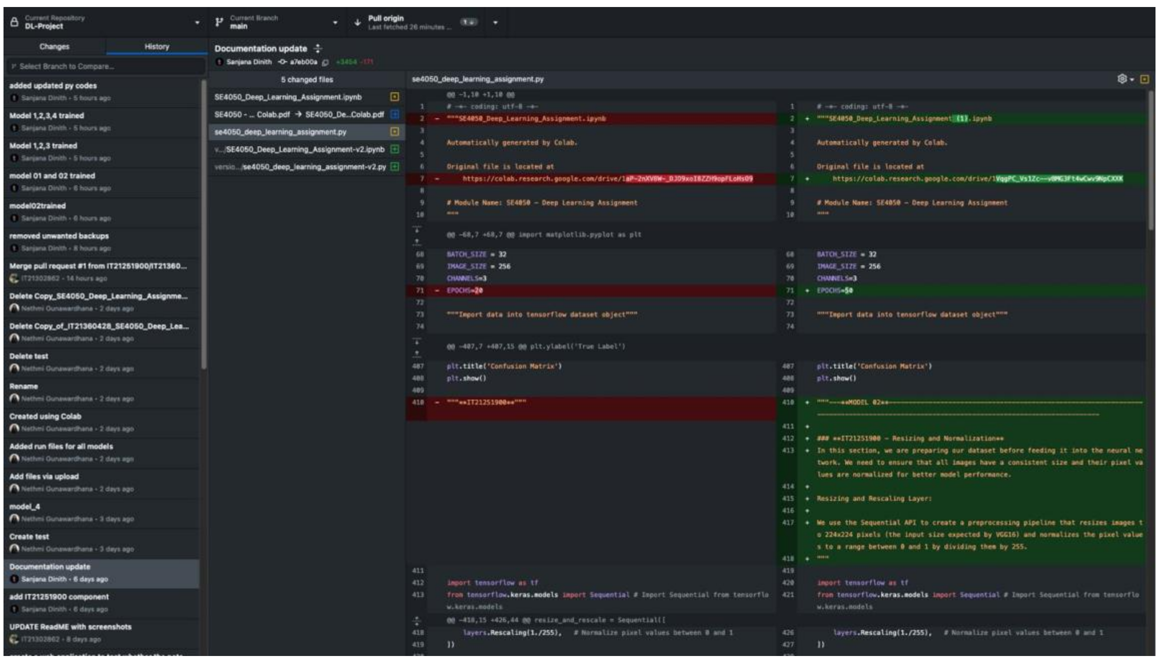
Plant Village Dataset :

https://www.kaggle.com/datasets/arjuntejaswi/plant-village

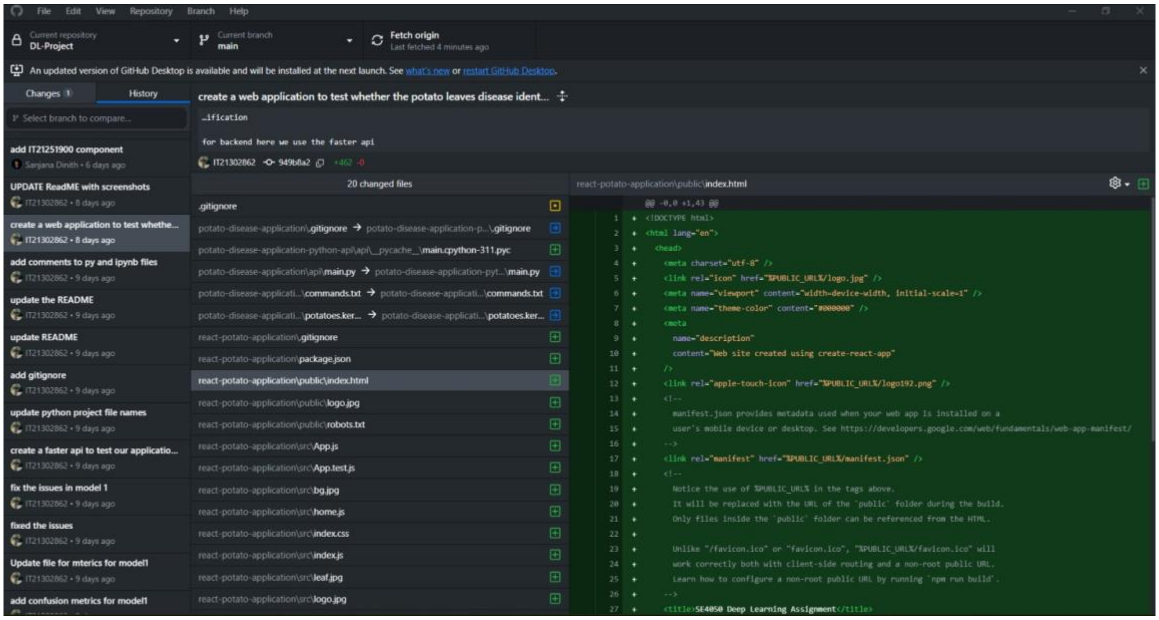
Appendix

Contribution

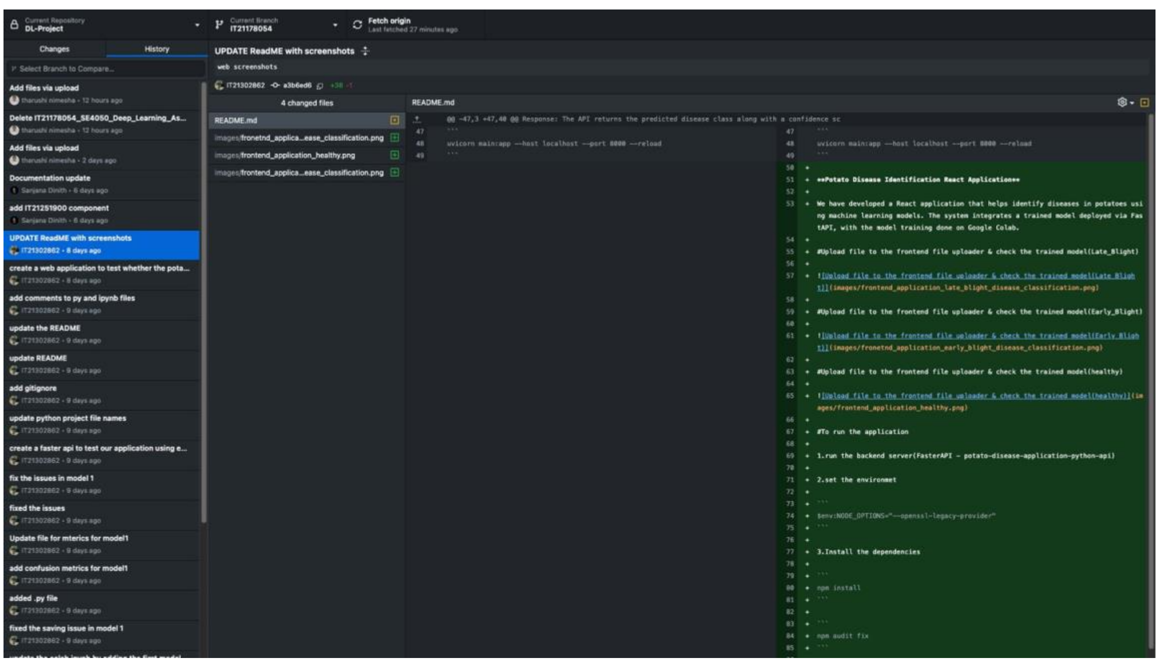
IT21251900



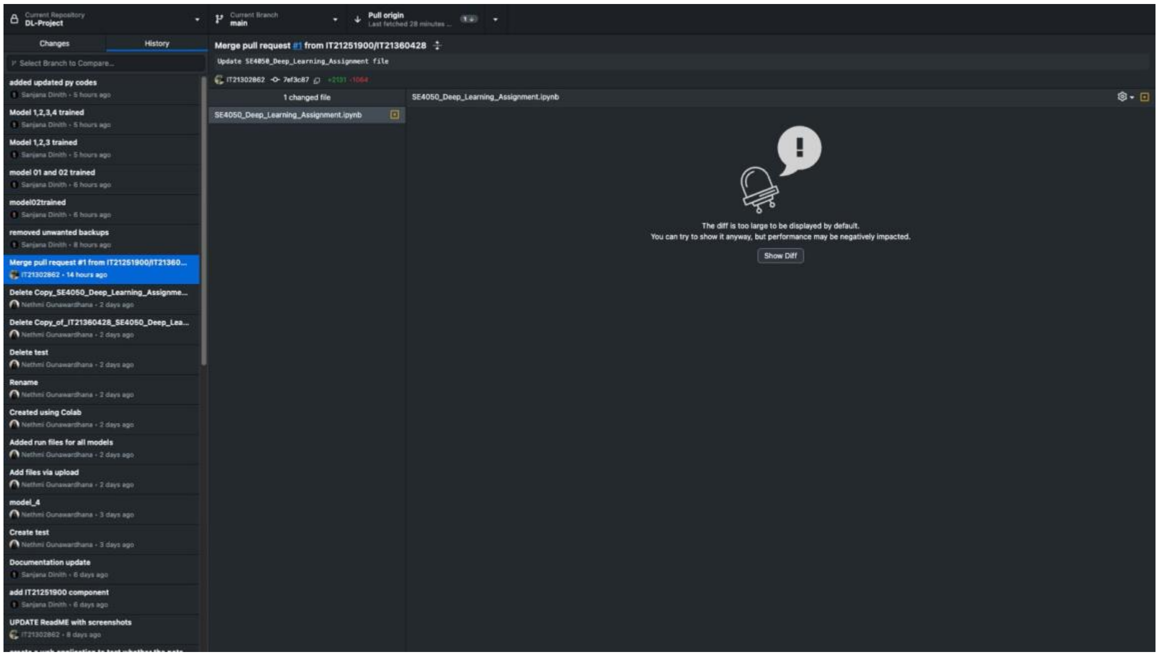
IT21302862



IT21178054



IT21360428


Current Repository
DL-Project
Current Branch
main
Pull origin
Last fetched 28 minutes ...
Changes
History
Select Branch to Compare...
added updated py codes
Sanjana Dinith • 5 hours ago
Model 1,2,3,4 trained
Sanjana Dinith • 5 hours ago
Model 1,2,3 trained
Sanjana Dinith • 5 hours ago
model 01 and 02 trained
Sanjana Dinith • 6 hours ago
model02trained
Sanjana Dinith • 6 hours ago
removed unwanted backups
Sanjana Dinith • 8 hours ago
Merge pull request #1 from IT21251900/IT21360...
IT21302862 • 14 hours ago
Delete Copy_SE4050_Deep_Learning_Assignme...
Nethmi Gunawardhana • 2 days ago
Delete Copy_of_IT21360428_SE4050_Deep_Lea...
Nethmi Gunawardhana • 2 days ago
Delete test
Nethmi Gunawardhana • 2 days ago
Rename
Nethmi Gunawardhana • 2 days ago
Created using Colab
Nethmi Gunawardhana • 2 days ago
Added run files for all models
Nethmi Gunawardhana • 2 days ago
Add files via upload
Nethmi Gunawardhana • 2 days ago
model_4
Nethmi Gunawardhana • 3 days ago
Create test
Nethmi Gunawardhana • 3 days ago
Documentation update
Sanjana Dinith • 6 days ago
add IT21251900 component
Sanjana Dinith • 6 days ago
UPDATE ReadME with screenshots
IT21302862 • 8 days ago
Merge pull request #1 from IT21251900/IT21360428
Update SE4050_Deep_Learning_Assignment file
IT21302862 7ef3c87
1 changed file
SE4050_Deep_Learning_Assignment.ipynb
SE4050_Deep_Learning_Assignment.ipynb
The diff is too large to be displayed by default.
You can try to show it anyway, but performance may be negatively impacted.
Show Diff