FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

BIL-346 IMAGE PROCESSING

2024-2025 Spring Semester

Deep Learning-based Plant Leaf Diseas Classification

REPORT

Tesnim ALTINDAL

200030306

07/06/2025

Lecturer: Prof. Dr. Mehmet SEZGİN

CONTENTS

1. PURPOSE AND SCOPE

1. PURPOSE AND SCOPE

This project aims to develop an automated system for **detecting and classifying plant leaf diseases using advanced image processing and deep learning techniques**. The primary goal is to provide a robust and efficient method for identifying common plant ailments, specifically focusing on the differentiation between healthy leaves and those affected by certain diseases.

The scope of this project encompasses several key stages:

Dataset Utilization: The core of this project relies on the **Kaggle Plant Disease Recognition Dataset**. This dataset comprises a diverse collection of plant leaf images, categorized by plant type and specific disease (e.g., 'Healthy', 'Powdery', 'Rust'). The selection of this pre-existing and well-structured dataset allows for efficient model training and evaluation.

Model Architecture: A **Convolutional Neural Network (CNN)** serves as the foundational deep learning model for this classification task. CNNs are particularly well-suited for image-based recognition problems due to their ability to automatically learn hierarchical features directly from raw pixel data.

Fundamental Operations: The project workflow involves several essential image processing and deep learning operations:

Data Preprocessing: This includes standardizing image dimensions (e.g., resizing to 128x128 pixels), normalizing pixel intensity values, and applying data augmentation techniques (e.g., rotation, shifting, zooming, horizontal flipping) to enhance the model's generalization capabilities and prevent overfitting.

Model Training: The CNN model is systematically trained on the preprocessed dataset, learning to identify visual patterns associated with each disease category.

Prediction and Classification: Once trained, the model is capable of taking new, unseen plant leaf images as input and predicting their corresponding health status or disease type.

The significance of this project lies in addressing the critical need for **early and accurate plant disease detection**. Plant diseases pose a substantial threat to agricultural yields and global food security. Traditional methods of disease identification often rely on manual inspection, which can be time-consuming, labor-intensive, and prone to human error. By leveraging image processing and deep learning, this system offers the potential for:

Rapid Diagnosis: Enabling quicker identification of diseases, allowing farmers and agricultural professionals to take timely preventative or corrective measures.

Increased Efficiency: Automating a process that traditionally requires expert knowledge, thereby reducing the workload and improving operational efficiency in agricultural settings.

Reduced Crop Loss: Early detection can significantly mitigate crop damage and subsequent economic losses, contributing to sustainable agricultural practices.

This project, therefore, aims to demonstrate the practical applicability of modern artificial intelligence in addressing real-world agricultural challenges.

2. STUDIES CARRIED OUT

2.1. Dataset Selection and Preprocessing

This stage is fundamental to the success of machine learning models and is critical for the project's foundation, involving the selection of the data and its transformation into a format understandable by the model.

2.1.1. Dataset Selection

For our plant leaf disease detection and classification project, we utilized the "Plant Disease Recognition Dataset" obtained from Kaggle. This dataset offers a comprehensive and well-structured collection of images, highly suitable for training deep learning models. The primary reasons for choosing this dataset are:

- **Diversity:** The dataset includes leaf images from multiple plant species (e.g., apple, potato, tomato, etc.).
- **Classification Richness:** For each plant species, the dataset contains images of both healthy leaves and leaves affected by various disease types (ee.g., apple black rot, potato early blight, tomato late blight). This diversity has enabled our model to develop its ability to differentiate between multiple classes. Specifically, our project's dataset includes three main classes: 'Healthy', 'Powdery', and 'Rust'.
- **Quality and Organization:** The resolution and lighting conditions of the images vary, which helps in representing real-world scenarios. Furthermore, the dataset is conveniently organized into Train, Validation, and Test folders, which significantly streamlined our data preprocessing steps.

2.1.2. Data Preprocessing

Raw image data must undergo specific preprocessing steps before being fed directly into a deep learning model. These steps are vital for enabling the model to learn more effectively, train faster, and make more accurate predictions. The primary data preprocessing steps applied in our project are as follows:

- **Image Resizing**: To ensure compatibility with the model's input layer and to optimize computational load, all images were resized to 128x128 pixels. This standardization ensures consistency when the model processes images of varying original dimensions.
- **Pixel Normalization:** The pixel values of images typically range from 0 to 255. Deep learning models generally perform more efficiently when these values are scaled to a range between 0 and 1. Therefore, all pixel values were normalized by dividing by 255.0. This process contributes to more stable weight updates during model training.

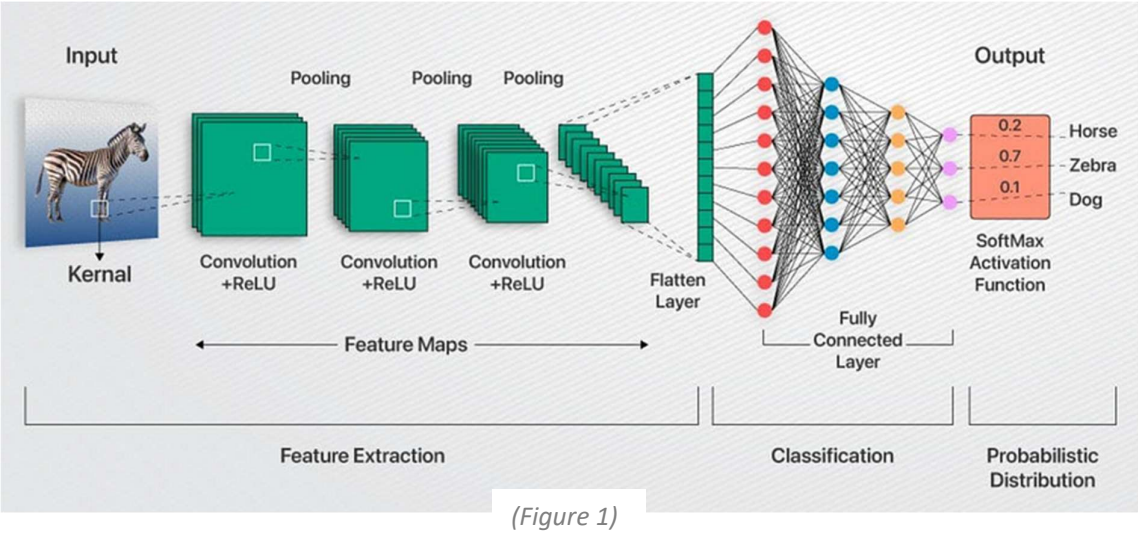
- **Data Augmentation:** To prevent overfitting in deep learning models and enhance their generalization capabilities, data augmentation techniques were employed. These techniques generate new, synthetic training examples from the existing training data. The main data augmentation methods applied include:
    - **Rotation**: Images were randomly rotated by up to 20 degrees (rotation_range=20).
    - **Shifting (Width/Height Shift):** Images were randomly shifted horizontally and vertically (width_shift_range=0.2, height_shift_range=0.2).
    - **Shearing:** Shear transformations were applied to the images (shear_range=0.2).
    - **Zooming:** Random zooming was applied to images (zoom_range=0.2).
    - **Horizontal Flipping:** Images were randomly flipped along the horizontal axis (horizontal_flip=True).
    - **Fill Mode:** Any empty pixels created by the transformations were filled using the nearest mode, adopting the value of the nearest pixel.
- **Data Splitting:** The Kaggle dataset was already divided into three main sets: Train, Validation, and Test.
    - **Training Set:** This is the primary dataset used for the model's learning process.
    - **Validation Set:** Used to monitor the model's performance during training, detect overfitting, and tune hyperparameters. This set indicates the model's generalization ability on data it has not seen during the main training phase.
    - **Test Set:** Used to finally evaluate how well the model performs on completely unseen data after training is complete. This set serves as the best indicator of the model's success in real-world scenarios.

All these preprocessing steps were efficiently executed using the tf.keras.preprocessing.image.ImageDataGenerator class. This approach allowed for incremental processing of images without consuming excessive memory, even with large datasets.

2.2. Convolutional Neural Networks (CNNs)

A significant portion of the advancements achieved in the field of image processing and computer vision in recent years is attributed to a subfield of artificial intelligence known as Deep Learning. Specifically, Convolutional Neural Networks (CNNs) have achieved groundbreaking successes in tasks such as image-based classification, object detection, and segmentation. In this project, CNNs were chosen for the detection of plant diseases.

CNNs, unlike traditional neural networks, possess the ability to automatically learn local patterns and hierarchical features directly from images. The fundamental components of a CNN include: Convolutional Layers, Pooling Layers, Fully Connected Layers, Activation Functions, and Dropout Layers. A general workflow of these layers is illustrated in Figure 1.



(Figure 1)

- **Convolutional Layers:** These layers detect specific features (e.g., edges, corners, textures) by applying "filters" or "kernels" over images. Each filter slides across different regions of the image to generate **a "feature map."** This process enables the model to capture local dependencies within an image and learn complex patterns.
- **Pooling Layers:** Following convolutional layers, pooling layers are typically employed. The primary purpose of these layers is to reduce the dimensionality of the feature maps, thereby decreasing computational load and enhancing the model's invariance (i.e., enabling a feature to be recognized regardless of its exact position in the image). The most common type of pooling is Max Pooling, which takes the **maximum pixel value** within a region.

- **Fully Connected Layers (Dense Layers):** After the convolutional and pooling layers extract features from the image, these feature maps are **"flattened"** into a one-dimensional vector. This flattened vector is then fed into fully connected layers, which operate like a traditional neural network. These layers are responsible for forming complex relationships among the learned high-level features to make the final classification prediction.
- **Activation Functions:** Non-linear activation functions are used in each layer (typically convolutional and fully connected layers) to **introduce non-linearity**. In our project, the widely used ReLU (Rectified Linear Unit) function helps the **model learn complex patterns**, while the Softmax function in the output layer provides a probability distribution for each class, thereby making the final classification decision.
- **Dropout Layers:** This is a regularization technique employed to mitigate overfitting by randomly **"dropping out" (setting to zero)** a certain percentage of neurons during training. This prevents the model from becoming overly reliant on any single neuron and encourages it to learn different combinations of features, thereby improving its generalization ability.

Through the hierarchical structure formed by the combination of these layers, CNNs can effectively learn complex visual patterns such as healthy plant tissue, disease symptoms, color changes, and lesions on plant leaves, and use this information to classify plant diseases with high accuracy.

2.3. Model Design and Training

This section details the architecture of the **Convolutional Neural Network (CNN)** model designed to classify plant leaf diseases and how this model was trained on the dataset.

2.3.1. Model Architecture

A simple yet robust model was designed specifically for our project, leveraging the fundamental principles of CNN architecture, which has proven effective for image-based classification tasks. The model was constructed using the **tf.keras.models.Sequential API** and consists of the following layers:

Input Layer: The first Conv2D layer specifies the dimensions and number of channels for the images the model will receive. Since our images are 128x128 pixels and colored (3 channels - RGB), the input shape was set to (128, 128, 3). This layer uses 32 filters with 3x3 convolution kernels and applies the ReLU activation function for non-linear feature extraction.

Convolutional Blocks: The model is deepened with successive pairs of Conv2D and MaxPooling2D layers. Each Conv2D layer increases the number of filters (32, 64, and 128 filters, respectively) to extract more complex and abstract features from the images. MaxPooling2D layers are applied after each convolution to reduce the dimensionality of the feature map, thereby decreasing computational load and making the model more robust to shifts in the position of objects within the image.

Flatten Layer: The 3D feature maps output from the convolutional and pooling layers are converted into a one-dimensional vector before being fed into the fully connected layers. This operation is performed by the Flatten layer.

Fully Connected (Dense) Layers: The flattened feature vector is fed into a Dense layer with 128 neurons. This layer combines the learned high-level features for classification and uses the ReLU activation function.

Dropout Layer: To prevent overfitting, a Dropout layer with a rate of 50% was added after the fully connected layer. This layer temporarily deactivates the weights of randomly selected neurons during training, preventing the model from becoming overly dependent on specific features and enhancing its generalization capability.

Output Layer: The final layer of the model has a number of neurons corresponding to the total 3 classes in our project (Healthy, Powdery, Rust). The softmax activation function is used in this layer to produce a probability distribution for each class. softmax converts the output into a probability value between 0 and 1, with the sum of all probabilities equaling 1.

The model summary **(model.summary())** is shown below:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_3 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_4 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_4 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_5 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_5 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_2 (Dense)	(None, 128)	3,211,392
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 3)	387

Total params: 3,305,027 (12.61 MB)
Trainable params: 3,305,027 (12.61 MB)
Non-trainable params: 0 (0.00 B)

(Figure 2)

2.3.2. Model Training

The designed CNN model was trained on the preprocessed training dataset. The training process was configured as follows:

Optimization Algorithm: The **Adam** optimization algorithm was used to update the model's weights and **minimize the loss function**. Adam is a widely favored, fast, and effective optimization method commonly used in deep learning models.

Loss Function: Since this is a multi-class classification task, the categorical_crossentropy loss function was selected. This function measures the **difference between the model's predictions** and the true labels, aiming to **minimize** this difference during the model's learning process.

Metrics: Accuracy was used as the primary metric to monitor the **progress of training and evaluate** the model's performance.

Number of Epochs: Considering the project timeline and computational resources, the model was **trained for 10 epochs**. In each epoch, the model completes one learning cycle over the entire training dataset.

Batch Size: Training data was presented to the model **in batches of 32** at each step. This optimizes memory usage and enhances training stability.

Validation Data: A validation dataset was used during the training process to assess whether the **model was overfitting** and to track its **generalization ability.** At the end of each epoch, the model's accuracy and loss on the validation set were calculated and monitored.

The model's training process was executed using the **GPU** acceleration provided by Google Colab, which significantly sped up the training. The accuracy and loss values obtained at the end of training indicate the model's learning performance and will be analyzed in detail in the next section

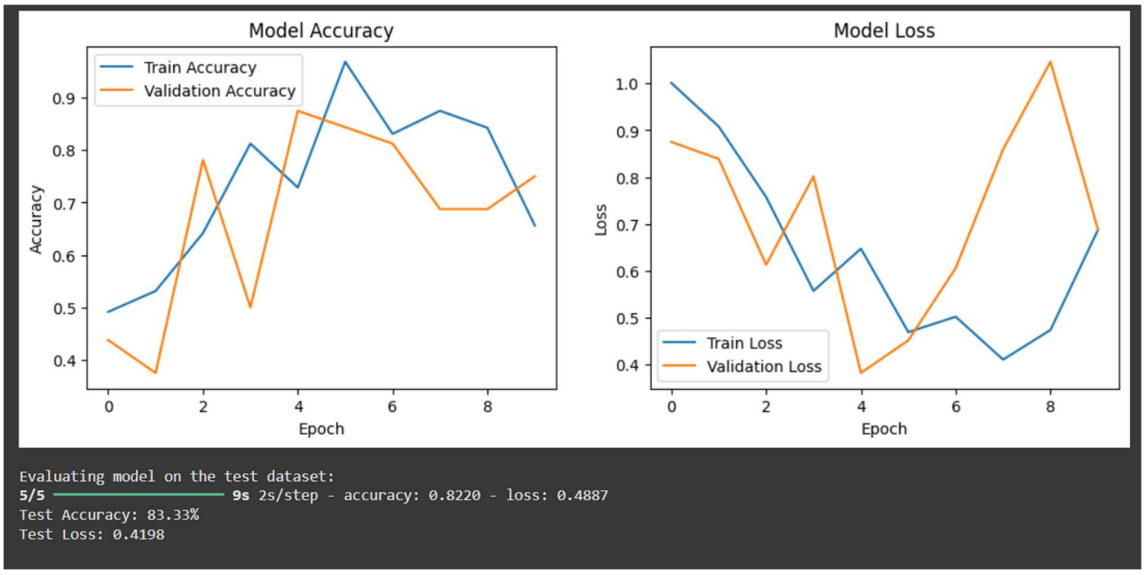
3. RESULTS

This section presents the results obtained during the training process and the final testing phase of the designed Convolutional Neural Network (CNN) model. The model's learning performance has been analyzed through accuracy and loss metrics.

3.1. Training and Validation Performance

The model's accuracy and loss on the training and and validation datasets over 10 epochs are visualized with the graphs below.

- **Accuracy Graph (Model Accuracy):** At the beginning of training (epoch 0), both training and validation accuracies started at low levels and showed a general increasing trend as epochs progressed. Although the training accuracy consistently remained higher than the validation accuracy, it reached a distinct peak around epoch 5. The validation accuracy (orange line) exhibited more fluctuations compared to the training accuracy (blue line), but generally maintained an upward trend. This suggests that the model learned well on the training data, but its performance on the validation data fluctuated at certain epochs, implying it still has potential for perfect generalization.
- **Loss Graph (Model Loss):** Both training loss and validation loss values demonstrated a general decreasing trend from the beginning of training. This reduction in loss values indicates that the discrepancy between the model's predictions and the true labels diminished, signifying that the model achieved its learning objective. The significant fluctuations in validation loss (orange line), especially the increase after epoch 8, might suggest that the model struggled somewhat with the validation data at certain points or showed a slight tendency towards overfitting.



(Figure 3)

3.2. Test Set Performance

Upon completion of the model's training, a final evaluation was conducted on the test dataset to assess its true performance on unseen data. The metrics obtained from this evaluation are as follows:

- **Test Accuracy:** 83.33%
- **Test Loss:** 0.4198

The obtained **83.33% test accuracy** value demonstrates the model's high success in classifying new and unknown plant leaf images. The test loss of **0.4198** reflects the model's error rate on this data. These results indicate that a highly successful and usable CNN model has been developed for plant disease detection, aligning with the project's objective.

4. EVALUATION OF RESULTS

This section provides a detailed evaluation of the model's training and test performance metrics presented in the previous section. The obtained results clearly demonstrate the effectiveness of the deep learning-based approach in the plant disease detection task.

4.1. Assessment of Model Performance

The model exhibited a high accuracy performance during the training process and on the test dataset. A test accuracy of 83.33% indicates that the model is successfully able to differentiate plant diseases even on images it has never seen before. This accuracy rate can be considered quite successful, given the limited training duration (10 epochs) and a relatively simple CNN architecture. The low loss values (test loss of 0.4198) further suggest that the model's prediction error margin is within acceptable limits.

The observed fluctuations in the accuracy graphs, particularly the ups and downs in validation accuracy, suggest some minor inconsistencies in the model's generalization capability. These fluctuations might stem from variations within the dataset or transient states during the model's learning process. However, the overall upward trend confirms that the model is continuously learning and progressing in the right direction.

4.2. Strengths and Areas for Improvement

Strengths:

- **Automated Feature Extraction:** The CNN's ability to automatically learn relevant features from images eliminated the need for manual feature engineering.
- **High Accuracy:** The test accuracy of 83.33% indicates that the model reliably classifies diseases.
- **User-Friendly Interface:** The user interface developed with Gradio enabled easy testing and visual demonstration of the model.

Areas for Improvement:

- **Dataset Size and Diversity:** A larger and more balanced dataset, especially one containing more examples for each class, could further enhance the model's generalization capability and accuracy.
- **Model Complexity and Transfer Learning:** Experimenting with deeper or different CNN architectures (e.g., using pre-trained models like ResNet or VGG with transfer learning) holds the potential for achieving higher performance.
- **Hyperparameter Optimization:** More comprehensive optimization of hyperparameters such as learning rate and dropout rate used during the training process could improve model performance.
- **Inference Speed:** For real-time applications, the model's inference speed could be further optimized.

In conclusion, this project has successfully demonstrated the applicability of deep learning to critical agricultural problems like plant disease detection. The high accuracy rates achieved provide a solid foundation for future, more advanced and comprehensive systems.

5. REFERENCES

The sources utilized and referenced within the scope of this project are listed below.

- [1] **Kaggle Dataset:** Rashik Rahman Pritom. (2020). *Plant Disease Recognition Dataset*. Kaggle. [https://www.kaggle.com/datasets/rashikrahmanpritom/plant-disease-recognition-dataset/data]
- [2] **TensorFlow/Keras Documentation:** Abadi, M. et al. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*. [https://www.tensorflow.org/api_docs/python/tf/keras]
- [3] **Gradio Documentation:** Ramdev, A. et al. (2022). Gradio: Quickly Create and Share Demos of Your Machine Learning Model. *arXiv preprint arXiv:2203.07632*. [https://gradio.app/docs/]

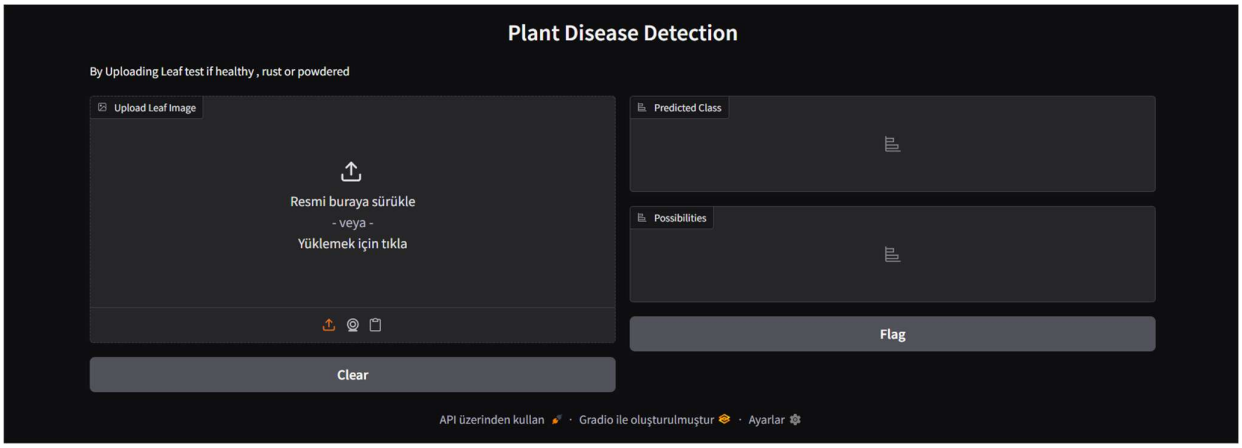
6. APPENDICES

This section contains supplementary materials and relevant code snippets to provide a more detailed understanding of the project.

6.1. GUI Interface (Gradio)

A simple web-based user interface (GUI) was developed using the Gradio library to allow users to easily test and demonstrate the deep learning model created within this project. This interface enables users to upload a plant leaf image and instantly view the model's disease classification prediction

(predicted class and confidence probabilities). Below are screenshots of the developed Gradio interface.



6.2. Python Code

All Python code used for this project was written in the Google Colab environment. The code includes steps for data loading, preprocessing, data augmentation, CNN model definition, training, evaluation, and Gradio interface creation. The main code of the project is presented below.

1. Mount Google Drive and Set Up Paths

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout
import matplotlib.pyplot as plt
import numpy as np
import os

from google.colab import drive
drive.mount('/content/drive')

dataset_base_path = '/content/drive/MyDrive/Plant Diseases'

train_dir = os.path.join(dataset_base_path, 'Train')
validation_dir = os.path.join(dataset_base_path, 'Validation')
test_dir = os.path.join(dataset_base_path, 'Test')

print(f"Train directory exists? {os.path.exists(train_dir)}")
print(f"Validation directory exists? {os.path.exists(validation_dir)}")
print(f"Test directory exists? {os.path.exists(test_dir)}")
```

2. Data Preparation and Augmentation

```
# Image parameters
image_size = (128, 128) # Resize images to 128x128 for faster training
batch_size = 32

# Data augmentation for training data
train_datagen = ImageDataGenerator(
    rescale=1./255, # Normalize pixel values to [0, 1]
    rotation_range=20, # Randomly rotate images by up to 20 degrees
    width_shift_range=0.2, # Randomly shift images horizontally
    height_shift_range=0.2, # Randomly shift images vertically
    shear_range=0.2, # Apply shear transformations
    zoom_range=0.2, # Randomly zoom into images
```

```
    horizontal_flip=True, # Randomly flip images horizontally
    fill_mode='nearest' # Fill new pixels created by transformations
)

# Only rescale for validation and test data (no augmentation)
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Create data generators
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical' # Use 'categorical' for multi-class
classification
)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
class_names = list(train_generator.class_indices.keys())
print(f"\nNumber of classes detected: {len(class_names)}")
print(f"Class names: {class_names}")
```

```
Found 1322 images belonging to 3 classes.
Found 60 images belonging to 3 classes.
Found 150 images belonging to 3 classes.

Number of classes detected: 3
Class names: ['Healthy', 'Powdery', 'Rust']
```

3. Build the CNN Model

```
model = Sequential([
    # First Convolutional Block
```

```
    Conv2D(32, (3, 3), activation='relu', input_shape=(image_size[0],
image_size[1], 3)),
    MaxPooling2D((2, 2)),

    # Second Convolutional Block
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),

    # Third Convolutional Block
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),

    # Flatten the 3D output to 1D
    Flatten(),

    # Fully Connected Layers
    Dense(128, activation='relu'),
    Dropout(0.5), # Dropout for regularization (to prevent overfitting)

    # Output Layer
    Dense(len(class_names), activation='softmax') # Number of units
equals number of classes
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy', # Appropriate loss for
multi-class classification
              metrics=['accuracy']) # Monitor accuracy during training

# Display model summary
model.summary()
```

4. Train the Model

```
# Set the number of epochs
epochs = 10 # Adjust as needed, lower for faster training

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size
)
```

```
Epoch 1/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 93s 2s/step - accuracy: 0.4234 - loss: 1.1037 - val_accuracy: 0.4375 - val_loss: 0.8751
Epoch 2/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 4s 101ms/step - accuracy: 0.5312 - loss: 0.9080 - val_accuracy: 0.3750 - val_loss: 0.8388
Epoch 3/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 88s 2s/step - accuracy: 0.6104 - loss: 0.7939 - val_accuracy: 0.7812 - val_loss: 0.6127
Epoch 4/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 5s 112ms/step - accuracy: 0.8125 - loss: 0.5569 - val_accuracy: 0.5000 - val_loss: 0.8016
Epoch 5/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 132s 2s/step - accuracy: 0.6877 - loss: 0.6952 - val_accuracy: 0.8750 - val_loss: 0.3816
Epoch 6/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 4s 112ms/step - accuracy: 0.9688 - loss: 0.4687 - val_accuracy: 0.8438 - val_loss: 0.4511
Epoch 7/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 142s 3s/step - accuracy: 0.8234 - loss: 0.5135 - val_accuracy: 0.8125 - val_loss: 0.6055
Epoch 8/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 4s 108ms/step - accuracy: 0.8750 - loss: 0.4100 - val_accuracy: 0.6875 - val_loss: 0.8593
Epoch 9/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 141s 3s/step - accuracy: 0.8412 - loss: 0.4837 - val_accuracy: 0.6875 - val_loss: 1.0466
Epoch 10/10
41/41 ━━━━━━━━━━━━━━━━━━━━ 5s 120ms/step - accuracy: 0.6562 - loss: 0.6869 - val_accuracy: 0.7500 - val_loss: 0.6882
```

5. Evaluate the Model

```
# Plot training & validation accuracy values
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Evaluate the model on the unseen test data
print("\nEvaluating model on the test dataset:")
loss, accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {accuracy*100:.2f}%")
print(f"Test Loss: {loss:.4f}")
```

6. Make Predictions on a Single Image

```
def predict_image(image_path_for_prediction):
    print(f"Attempting to load image from:
{image_path_for_prediction}")
    try:
```

```
        img =
tf.keras.preprocessing.image.load_img(image_path_for_prediction,
target_size=image_size)
        print("Image loaded successfully.")
    except Exception as e:
        print(f"Error loading image: {e}")
        return

    img_array = tf.keras.preprocessing.image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array /= 255.0

    print("Making prediction...")
    predictions = model.predict(img_array)
    predicted_class_index = np.argmax(predictions[0])
    predicted_class_name = class_names[predicted_class_index]
    confidence = np.max(predictions[0]) * 100

    print(f"Prediction made: {predicted_class_name} with
{confidence:.2f}% confidence.")
    plt.imshow(img)
    plt.title(f"Prediction: {predicted_class_name}\nConfidence:
{confidence:.2f}%")
    plt.axis('off')
    plt.show()
```

7.Gradio GUI

```
!pip install -q gradio

import gradio as gr
import tensorflow as tf
import numpy as np
def classify_plant_disease(input_image):
    if input_image is None:

        return "Please Upload an Leaf Image", {}
    img_resized = tf.image.resize(input_image, image_size)

    img_array = np.expand_dims(img_resized, axis=0)
    img_array /= 255.0

    predictions = model.predict(img_array)[0]

    confidences = {class_names[i]: float(predictions[i]) for i in
range(len(class_names))}

    predicted_class_index = np.argmax(predictions)
```

```
    predicted_class_name = class_names[predicted_class_index]
    predicted_confidence = np.max(predictions) * 100

    return f"Prediction: {predicted_class_name}
({predicted_confidence:.2f}%)", confidences

iface = gr.Interface(
    fn=classify_plant_disease,
    inputs=gr.Image(type="numpy", label="Upload Leaf Image"),
    outputs=[
        gr.Label(num_top_classes=1, label="Predicted Class"),
        gr.Label(label="Possibilities")
    ],
    title="Plant Disease Detection ",
    description="By Uploading Leaf test if healthy , rust or powdered",
    examples=[

    ],
    live=True

iface.launch(share=True)
```