International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

DEEP LEARNING-BASED PLANT DISEASE DETECTION USING MOBILENET V3 AND IMAGE CLASSIFICATION

Vidhanshu Kachhwaha[1*], Dr. Chetna[2]
[1]*Student at Stonehill International School, Email: vidhanshu.k11@gmail.com
[2]Head of Department (Design and Computer Science), Stonehill International School
Email: chetna.kachhwaha@gmail.com

Abstract
The automation of plant disease identification can significantly enhance agricultural productivity by enabling early intervention. This research proposes a deep learning-based multi-class image classification system using the MobileNet V3 architecture to identify 38 distinct plant disease categories. Using a publicly available image dataset from Kaggle, this study conducted a full model development pipeline including data preprocessing, exploratory data analysis, transfer learning, training with validation, and deployment via a Gradio interface. The system was evaluated using accuracy and multi-class log loss metrics and demonstrates promising results for real-time agricultural applications.

Keywords—Plant Disease Detection, Deep Learning, MobileNet V3, Image Classification, TensorFlow, Transfer Learning, IoT, Smart Agriculture.

I. Introduction

Agriculture plays a vital role in global food security. However, crop diseases result in substantial economic losses annually. Traditional manual inspection methods are often slow, subjective, and labor-intensive. Machine learning, particularly deep learning, presents a scalable, efficient alternative for real-time disease detection.

The AI model powering Nature Surfer(a rover that can collect data from the fields) is a Random Forest Classifier, which I chose for its robustness and high accuracy in terms of handling environmental and agricultural datasets. This model works by creating multiple decision trees during training and outputs the seed type that receives the majority vote across all the trees. It is extremely effective in classifying complex data with many variables, such as soil nutrient levels and environmental conditions, and thus making it ideal for recommending suitable seeds for different farm conditions. Its ability to identify the most important features (like nitrogen, pH, or rainfall) helps in making sure that the final seed suggestion is both scientifically grounded and practical for real-world use.

This study presents a system using MobileNet V3**,** a lightweight convolutional neural network (CNN), optimized for mobile and edge devices. The aim is to create an accurate, efficient model to classify plant leaf images into one of 38 disease classes, with practical usability via an interactive web application.

II. Literature Review

The Plant Health Identification Project aims to develop a deep learning-based system using the MobileNet V3 model to accurately classify images of plants into 38 different classes based on the disease they have. The project involves exploratory data analysis to understand the dataset's distribution, diversity, and labelling. Data pre-processing techniques will be applied, followed by model selection and transfer learning using the MobileNet V3 architecture. The model will be trained on the validation and training data sets and evaluated on a test set using accuracy and multi-class log loss as metrics. Finally, the trained model will be deployed using the Nature Surfer robot. Through these steps, the project strives to create an efficient and reliable solution for identifying plant diseases.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The dataset used to train the AI model includes several critical agricultural parameters. Input features collected from the field include soil nitrogen (N), phosphorus (P), potassium (K) levels, moisture content, and pH value. In addition, parameters from the environment, such as temperature, humidity, light intensity, rainfall presence, and GPS coordinates, are included. These parameters are gathered through sensors present on the rover. The target variable in the dataset is the best seed type to plant out of a library of 22 different options, where each is associated with optimal conditions for growth.
Convolutional Neural Networks (CNNs) have shown superior performance in image classification tasks. Approaches like AlexNet, VGG, and ResNet are effective but computationally intensive. MobileNet V3 addresses these constraints by combining efficiency with accuracy, making it ideal for field deployment. Past research using models such as ResNet50 and DenseNet for plant disease classification showed high accuracy, but with large computational overhead.

III. Dataset And Analysis

A. Dataset

The dataset used originates from the New Plant Diseases Dataset on Kaggle [1], consisting of over 87,000 images labeled across 38 classes, including diseased and healthy samples. It includes plant varieties like Apple, Tomato, Corn, Grape, and Soybean. It is a labeled dataset (with labels being contained in training_labels_csv and validation_labels_csv), meaning each image is associated with a specific plant health issue. The aforementioned CSVs have been created by using file directories and were not included in the Kaggle dataset, New Plant Diseases Dataset. (2018, November 18). Kaggle.

- Training images: 70,295
- Validation images: 17,572

Each image is labelled in the format Plant_Disease.
The project involves several steps to build the plant disease identification system:

1. Exploratory Data Analysis: Before proceeding with the model training, it is crucial to conduct exploratory data analysis (EDA) to gain insights into the dataset. This analysis aims to understand various aspects of the data. EDA on the Labels CSV: We will explore the labels CSV file, examining the available columns, data types, and any missing or erroneous values. This step ensures that the dataset is well-prepared for subsequent analyses and modeling tasks. Spread of Data Between Classes: We will analyze the distribution of the number of images across different diseases. This examination will help us identify any potential class imbalances or variations in the dataset. Number of Unique diseases: We will reconfirm the total number of diseases present in the dataset. This information will give us an idea of the diversity and variety of diseases captured. Visualizing Disease classes: We will create visualizations to showcase some diseases present in the dataset. This will provide a glimpse into the different classes and their representation in the data.

2. Data Pre-processing: To prepare the data for training, it needs to be pre-processed. This includes converting the images into a suitable format for deep learning models, such as tensors or arrays. Additional pre-processing steps may involve converting read images to JPEGs, resizing the images, and normalizing pixel values.
The code below makes a copy of the original list to avoid modifying it. Randomly, it picks n elements using random.choice() and removes each chosen item from the copy to ensure uniqueness. Then it returns the list of randomly selected items. select_n_random_items() helps randomly sample from a list without replacement, and return_disease_plant_name() cleans up and splits a label string into a readable plant name and disease name for presentation or analysis.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
def select_n_random_items(lst, n):
  lst_copy = lst.copy()
  selected_items = []
  for i in range (1, n + 1):
    item = rn.choice(lst_copy)
    selected_items.append(item)
    lst_copy.remove(item)

  return selected_items

def return_disease_plant_name(string):
    """
    Extracts the name and disease information from a label string in the format '<name>__<disease>_<disease>'.

    Parameters:
    - string (str): A string containing information in the format '<name>__<disease>_<disease>'.

    Returns:
    - tuple: A tuple containing two elements:
        - name (str): The name extracted from the input string.
        - disease (str): The disease information extracted from the input string, with multiple underscores formatted.

    Example:
    If the input string is 'plant_name__leaf_spot_disease', this function will return ('plant_name', 'leaf spot disease').

    Note:
    The function assumes that the input string follows the specified format, with double underscores separating the name and disease in

    """
    first_underscore_index = string.find('_')
    name = format_underscores(string[0:first_underscore_index])
    disease = format_underscores(string[first_underscore_index + 2:])
    return (name, disease)
```

To clean and standardize text labels or feature names, like converting filenames or labels to a readable form, preparing features for reporting or visualization, the code below is used.
format_underscores()Clean and format strings with underscores as part of Data Pre-processing / Cleaning, and list_to_string() converts a list to a string for display or reporting. The code below is generating readable labels, summarizing predictions, outputs, or processed results, and displaying lists in model outputs (e.g., recommended items or detected classes)

```
def format_underscores(input_string):
    '''
    Formats an input string by capitalizing each word separated by underscores.

    Parameters:
        - input_string (str): The input string containing underscores.

    Returns:
        str: The formatted string with words capitalized and underscores replaced by spaces.

    Example usage:
        formatted_string = format_underscores("hello_world")
        print(formatted_string)  # Output: "Hello World"
    '''
    words = input_string.split('_')
    formatted_words = [word.capitalize() for word in words]
    formatted_string = ' '.join(formatted_words)
    return formatted_string

def list_to_string(input_list, commas=False):
    '''
    Converts a list of items into a formatted string.

    Parameters:
        - input_list (list): The list of items to be converted to a string.
        - commas (bool): If True, separate items with commas; otherwise, separate with spaces.

    Returns:
        str: The formatted string containing items from the list.

    Example usage:
        items = ['apple', 'banana', 'orange']
        result_string = list_to_string(items, commas=True)
        print(result_string)  # Output: "apple, banana, orange"
    '''
    # Convert each item in the list to a string
    str_list = [format_underscores(str(item)) for item in input_list]
    # Join the items with spaces between them
    if commas:
        result_string = ', '.join(str_list)
    else:
        result_string = " ".join(str_list)
    return result_string
```

The code below displays a specific image from the validation dataset and prints its label. It loads the image using matplotlib.image and then displays the image and hides the axis. Then it prints the corresponding class label of the displayed image.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
# random image from validation dataset

#image path
image_path = validation_file_paths[1000]

# Load the image
image = mpimg.imread(image_path)

# Display the image
plt.imshow(image)
plt.axis('off')  # Turn off axis labels and ticks
plt.show()
print(validation_class_labels[1000])
```

Data Loading and Labeling are a foundational part of data preprocessing**.** It loads paths to image files**,** associates each file with its corresponding label, and prepares data to be fed into a data generator or model pipeline. The code below is loading and organizing file paths and class labels for both training and validation datasets from a directory structure.

```
training_classes = os.listdir('/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New Pl
valid_classes = os.listdir('/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New Plant
```

```
training_file_paths = []
training_class_labels = []
for training_class in training_classes:
  class_file_paths = os.listdir(f'/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New
  for fl in class_file_paths:
    training_file_paths.append(f'/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New
    training_class_labels.append(training_class)

validation_file_paths = []
validation_class_labels = []
for validation_class in valid_classes:
  class_file_paths = os.listdir(f'/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New
  for fl in class_file_paths:
    validation_file_paths.append(f'/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/Ne
    validation_class_labels.append(validation_class)
```

This code is for analyzing and visualizing class distribution in your training dataset, along with summary statistics like mean, median, quartiles, standard deviation, and IQR. This will help to visualize class imbalance, understand whether the dataset is skewed (some diseases have many more images than others), and make decisions depending on whether to augment minority classes or whether to apply class weights in model training.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
# setting seaborn style
sns.set_style('whitegrid')

# Preparing unique_diseases and the number of times they occur in the dataset
unique_diseases, value_counts = np.unique(training_labels_csv['Label'], return_counts=True)

# Preparing the plot
fig, ax = plt.subplots()
plt.figure(facecolor='#000000')
ax.bar(unique_diseases, value_counts, label='Elements in each class', color='#50C878', align = 'edge')
ax.axhline(y=np.mean(value_counts), color='#461959', linestyle='--', label='Mean')
ax.axhline(y=np.percentile(value_counts, 50), color='#7A316F', linestyle='--', label='Median/Q2')
ax.axhline(y=np.percentile(value_counts, 75), color='#CD6688', linestyle='--', label='Q3')
ax.axhline(y=np.percentile(value_counts, 25), color='teal', linestyle='--', label='Q1')
ax.legend(bbox_to_anchor=(1.45, 0.9), loc='upper right')

# Annotation text for statistical summary
callout_text = f'''
Max Value: {max(value_counts)}
Min Value: {min(value_counts)}
Range: {max(value_counts) - min(value_counts)}
Variation: {round(np.var(value_counts))}
Standard Deviation: {round(np.std(value_counts))}
IQR: {np.percentile(value_counts, 75) - np.percentile(value_counts, 25)}
Q1: {np.percentile(value_counts, 25)}
Q2: {np.percentile(value_counts, 50)}
Q3: {np.percentile(value_counts, 75)}
Mean: {round(np.mean(value_counts))}
'''
callout_text_header = f'Statistical summary'
ax.annotate(callout_text, xy=(1.03, 0.01), xycoords='axes fraction',
            bbox=dict(boxstyle='round', facecolor='white', edgecolor='#50C878'))
ax.annotate(callout_text_header, xy=(1.03, 1.0), xycoords='axes fraction',
            bbox=dict(boxstyle='round', facecolor='white', edgecolor='#50C878'))


ax.set(title='Number of images for each label in the training dataset', xlabel='* Names of classes have been removed\n due to the prese
ax.set_xticks([])
```

3. Model Selection: For this project, the MobileNet V3 model will be utilized. MobileNet V3 is a lightweight neural network architecture designed for mobile and embedded devices. It strikes a balance between model size and accuracy, making it an excellent choice for this plant health identification task.

This block prepares the tools and libraries required for:

- Reading data
- Visualizing images
- Preprocessing inputs
- Building and eventually training a deep learning model

This code block imports libraries and modules required for various tasks in the workflow — including data loading, visualization, model building, and training.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
%matplotlib inline
# visualization
import matplotlib.pyplot as plt
from matplotlib.pyplot import imread
import matplotlib.image as mpimg
import seaborn as sns
#
from PIL import Image
from IPython.display import Image as im

# model building and handling data
from sklearn.model_selection import train_test_split
import tensorflow as tf
import tensorflow_hub as hub

from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
#  managing functions
import functools
# handling CSVs
import pandas as pd

# numerical computation
import numpy as np

# randomization
import random as rn

# handling directories
import os

# time
import time
import datetime
```

The image below is a real example output from the exploratory visualization step of our pipeline — specifically used to verify data correctness before training a model. This is a leaf with visible disease symptoms in combination with the visualization code. This is to check data quality, verify correct labeling, and understand class distribution or variability in input samples.

Fig: Apple___Black_rot

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The code below helps with inspecting training data visually**, e**nsuring images are loaded correctly, and **c**hecking class label integrity.

```
# random image from the training dataset

# image path
image_path = training_file_paths[23]

# Load the image
image = mpimg.imread(image_path)

# Display the image
plt.imshow(image)
plt.axis('off')  # Turn off axis labels and ticks
plt.show()
print(validation_class_labels[-1])
```

To visually inspect samples from the dataset and ensure data has been correctly loaded, labeled, and possibly augmented (this image is rotated, indicating augmentation). This image appears as the exact image that would be displayed when running the above code, so it's the visual output of that code block.

Figure: Tomato___healthy

The lines below validate that the number of file paths matches the number of labels for both training and validation datasets, and confirm the number of classes. This involves preparing, organizing, and cleaning data before feeding it into a machine learning model.

```
len(training_file_paths), len(training_class_labels), len(validation_file_paths), len(validation_class_labels)
```

```
(70295, 70295, 17572, 17572)
```

```
len(training_classes), len(valid_classes)
```

```
(38, 38)
```

```
training_labels_csv = pd.DataFrame({'FilePath':training_file_paths, 'Label':training_class_labels})
training_labels_csv.to_csv('/content/drive/MyDrive/Colab Notebooks/plant-health-project/training-labels.csv')
training_labels_csv = training_labels_csv.sample(frac = 1)
training_labels_csv.reset_index(drop=True, inplace=True)

valid_labels_csv = pd.DataFrame({'FilePath':validation_file_paths, 'Label':validation_class_labels})
valid_labels_csv.to_csv('/content/drive/MyDrive/Colab Notebooks/plant-health-project/valid-labels.csv')
valid_labels_csv = valid_labels_csv.sample(frac = 1)
valid_labels_csv.reset_index(drop=True, inplace=True)
#order of files randomised to prevent overfitting
```

```
valid_labels_csv
```

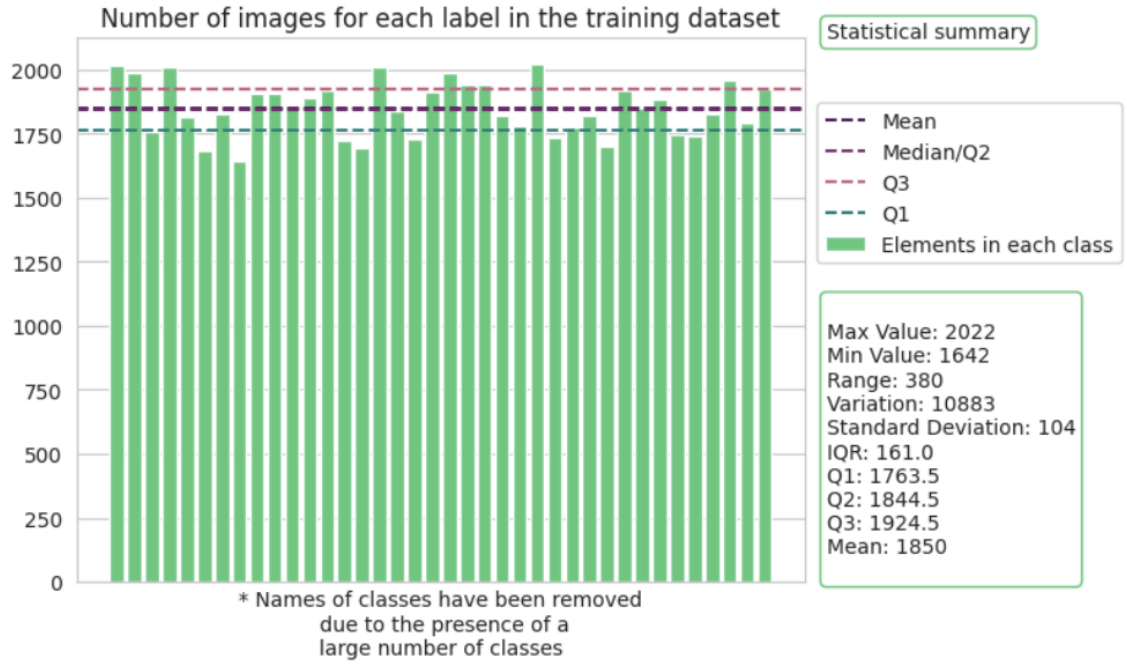
Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The code below creates a bar plot to visualize the number of images per class. Adds statistical lines showing Mean, Median (Q2), Q1, and Q3 (25th and 75th percentiles). It also displays a detailed statistical summary (range, std deviation, IQR, etc.) as an annotation.

```
(array(['Apple___Apple_scab', 'Apple___Black_rot',
        'Apple___Cedar_apple_rust', 'Apple___healthy',
        'Blueberry___healthy', 'Cherry_(including_sour)___Powdery_mildew',
        'Cherry_(including_sour)___healthy',
        'Corn_(maize)___Cercospora_leaf_spot Gray_leaf_spot',
        'Corn_(maize)___Common_rust_',
        'Corn_(maize)___Northern_Leaf_Blight', 'Corn_(maize)___healthy',
        'Grape___Black_rot', 'Grape___Esca_(Black_Measles)',
        'Grape___Leaf_blight_(Isariopsis_Leaf_Spot)', 'Grape___healthy',
        'Orange___Haunglongbing_(Citrus_greening)',
        'Peach___Bacterial_spot', 'Peach___healthy',
        'Pepper,_bell___Bacterial_spot', 'Pepper,_bell___healthy',
        'Potato___Early_blight', 'Potato___Late_blight',
        'Potato___healthy', 'Raspberry___healthy', 'Soybean___healthy',
        'Squash___Powdery_mildew', 'Strawberry___Leaf_scorch',
        'Strawberry___healthy', 'Tomato___Bacterial_spot',
        'Tomato___Early_blight', 'Tomato___Late_blight',
        'Tomato___Leaf_Mold', 'Tomato___Septoria_leaf_spot',
        'Tomato___Spider_mites Two-spotted_spider_mite',
        'Tomato___Target_Spot', 'Tomato___Tomato_Yellow_Leaf_Curl_Virus',
        'Tomato___Tomato_mosaic_virus', 'Tomato___healthy'], dtype=object),
 38)
```

The bar plot visualizes the number of images per class. It adds statistical lines showing Mean, Median (Q2), Q1, and Q3 (25th and 75th percentiles). The plot also includes a detailed statistical summary (range, std deviation, IQR, etc.) as an annotation.




```
<Figure size 640x480 with 0 Axes>
```

The code below is used to analyze and visualize the distribution of labels (diseases**)** in the validation dataset, along with displaying summary statistics such as mean, quartiles, standard deviation, and

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

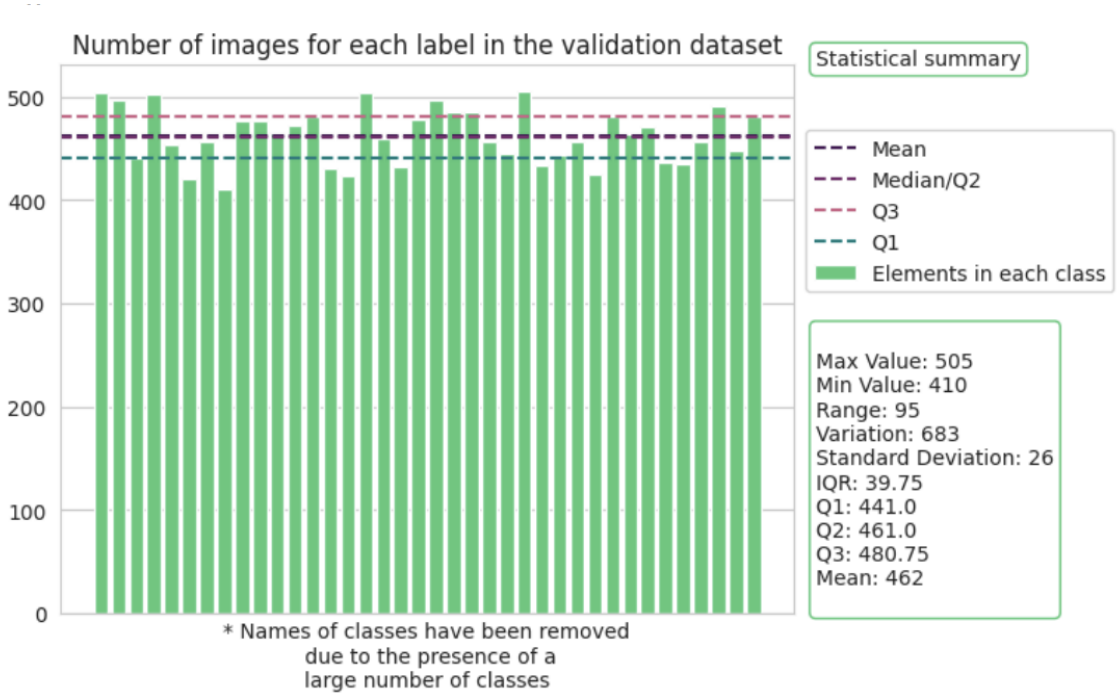
range. This data preprocessing phase, specifically Exploratory Data Analysis (EDA), ensures that the data is balanced before proceeding to model training or evaluation.

```
# setting seaborn style
sns.set_style('whitegrid')

# Preparing unique_diseases and the number of times they occur in the dataset
unique_diseases, value_counts = np.unique(valid_labels_csv['Label'], return_counts=True)

# Preparing the plot
fig, ax = plt.subplots()
plt.figure(facecolor='#000000')
ax.bar(unique_diseases, value_counts, label='Elements in each class', color='#50C878', align = 'edge')
ax.axhline(y=np.mean(value_counts), color='#461959', linestyle='--', label='Mean')
ax.axhline(y=np.percentile(value_counts, 50), color='#7A316F', linestyle='--', label='Median/Q2')
ax.axhline(y=np.percentile(value_counts, 75), color='#CD6688', linestyle='--', label='Q3')
ax.axhline(y=np.percentile(value_counts, 25), color='teal', linestyle='--', label='Q1')
ax.legend(bbox_to_anchor=(1.45, 0.9), loc='upper right')

# Annotation text for statistical summary
callout_text = f'''
Max Value: {max(value_counts)}
Min Value: {min(value_counts)}
Range: {max(value_counts) - min(value_counts)}
Variation: {round(np.var(value_counts))}
Standard Deviation: {round(np.std(value_counts))}
IQR: {np.percentile(value_counts, 75) - np.percentile(value_counts, 25)}
Q1: {np.percentile(value_counts, 25)}
Q2: {np.percentile(value_counts, 50)}
Q3: {np.percentile(value_counts, 75)}
Mean: {round(np.mean(value_counts))}
'''
callout_text_header = f'Statistical summary'
ax.annotate(callout_text, xy=(1.03, 0.01), xycoords='axes fraction',
            bbox=dict(boxstyle='round', facecolor='white', edgecolor='#50C878'))
ax.annotate(callout_text_header, xy=(1.03, 1.0), xycoords='axes fraction',
            bbox=dict(boxstyle='round', facecolor='white', edgecolor='#50C878'))


ax.set(title='Number of images for each label in the validation dataset', xlabel='* Names of classes have been removed\n due to the pre
ax.set_xticks([])
```

The graph below is the output of the above code that analyzes and visualizes the class distribution of the validation dataset. With the help of the above code, we counted how many images exist per class label in the validation dataset, and then it was visualized using a bar chart below. The chart adds statistical metrics (mean, quartiles, standard deviation, etc.) to summarize the dataset.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)



Below is the Exploratory Data Analysis (EDA) stage that helps to verify the dataset's quality and integrity before it's used to train a model. The function show_plant_diseases() is used to visualize sample images from the training dataset, either randomly or filtered by a specific disease name. It displays the images in a grid layout using matplotlib.

```
# function to show n plant diseases
def show_plant_diseases(n = 5,
                        path_list = training_file_paths,
                        random_state = 404, n_cols = 5,
                        labels = training_labels_csv['Label'],
                        disease_name = None):
  """
    Display n random images of plant diseases from the dataset.

    Parameters:
        n (int, optional): Number of random images to display. Default is 5.
        path_list (list, optional): List of file paths to images. Default is `filepaths`.
        random_state (int, optional): Random seed for shuffling images. Default is 42.
        n_cols (int, optional): Number of columns in the displayed image grid. Default is 5.
        labels (numpy array, optional): Array of labels corresponding to the file paths. Default is `np.array(labels_csv['Label'])`.
        disease_name (str, optional): If specified, only display images of the specified plant disease. Default is None.

    Returns:
        None: This function displays the images using Matplotlib.

    Note:
        - If disease_name is None, the function randomly shuffles the images and displays n of them.
        - If disease_name is specified, the function displays n images of the given disease (if available).
        - The images are plotted in a grid with n_cols columns.
        - The function assumes that images are stored in the provided file paths.
  """
  np.random.seed(random_state)
  if disease_name == None:
    # shuffling file paths and labels
    combined = list(zip(training_class_labels, training_file_paths))
    np.random.shuffle(combined)
    output_labels, output_path_list = zip(*combined)
  else:
    filtered_labels_csv = training_labels_csv[training_labels_csv['Label'] == disease_name]
    output_labels = np.array(filtered_labels_csv['Label'])
    output_path_list = filtered_labels_csv['FilePath']

  # calculating number of rows
  n_rows = (n // n_cols) + 1

  # plotting image
  plt.figure(figsize = (15, 15))
  plt.suptitle(f'{n} random photos from the dataset!', fontsize = 42)

  for i in range(n):
    image = Image.open(output_path_list[i])
    ax = plt.subplot(n_rows, n_cols, i + 1)
    ax.imshow(image)
    ax.set_xlabel(f'''Name: {return_disease_plant_name(output_labels[i])[0]}
Disease: {return_disease_plant_name(output_labels[i])[1]}
    ''', fontsize = 12)
    ax.set_xticks([])
    ax.set_yticks([])
    plt.grid(False)
  plt.tight_layout()
  plt.show()
```

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The function show_plant_diseases() is designed to randomly select images from the dataset, optionally filter by a specific plant disease label. And **d**isplay these images in a grid with appropriate annotations for the plant name and disease.
The resulting output (as shown in your second image) visually confirms that the images are correctly loaded, the labels (e.g., Tomato - Target Spot, Potato - Healthy) are correctly mapped, and there is a variety of classes in the dataset.

4. Model Creation: To create the neural network model for plant health identification, we will employ the MobileNet V3 architecture, initialized with pre-trained parameters from a large-scale image classification dataset - namely the ImageNet dataset. This approach, known as transfer learning, allows us to leverage knowledge gained from a diverse range of images and apply it to our specific task. We will add a Dense layer with an appropriate activation function to the model to adapt it for disease classification. After compiling the model with the Categorical Cross entropy loss (which measures how close the predictions of the model are to the actual data), the Adam optimizer (changes weights and other parameters to ensure the minimum loss when the model is learning), and the accuracy metric, we will build the model using the specified input shape.
This code sets up the Colab environment, prepares the data, and installs tools for monitoring and UI deployment. Actual model creation (e.g., building a neural network, training it) happens after this setup.

```
#CODE NOT TO BE RUN UNLESS FILES ARE ABSENT
!unzip "/content/drive/MyDrive/Colab Notebooks/plant-health-project/archive (1).zip" -d "/content/drive/MyDrive/Colab Notebooks/plant-h
```

```
# mounting drive and libraries
from google.colab import drive
drive.mount('/content/drive')
```

```
#code already ran; removed output to make it look cleaner
!pip install gradio
```

```
#code already ran; removed output to make it look cleaner
!pip install mlnotify
```

The code below performs two major tasks: firstly, visualizing a batch of preprocessed training images with their labels. Secondly, initializing model input settings for transfer learning using a pretrained model from TensorFlow Hub. It helps **to** inspect if the images and labels are aligned correctly before actual training and ensures there are no preprocessing or label-mismatch issues**.** Here we are defining the architecture input size and selecting the pretrained base model (MobileNetV3). This is the step just before building your model layers.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
# visualizing batches
def visualize_batches(images, labels):
  """
  Visualizes a batch of images and their corresponding labels.

  Parameters:
      images (numpy array): A batch of preprocessed images to be visualized.
      labels (numpy array): The corresponding labels for the images, typically in one-hot encoded format - boolean labels.

  Visualization:
      The function creates a 5x5 grid of subplots to display 25 images from the batch.
      It sets the title of the entire visualization using `plt.suptitle()`,
      indicating that it shows the first batch of photos.
      For each image, it creates a subplot using `plt.subplot()` and displays the image using `plt.imshow()`.
      The title of each subplot shows the predicted disease label associated with the image.
      The axes of the subplots are turned off to remove any unnecessary clutter.

  Returns:
      This function does not return any value.
      Instead, it generates a visual representation of the batch of images and their labels on the Matplotlib plot.
  """
  plt.figure(figsize = (20, 20))
  plt.suptitle('25 Photos from a training batch!', fontsize = 32, fontweight = 'bold')
  for i in range(25):
    ax = plt.subplot(5, 5, i + 1)
    plt.imshow(images[i])
    plt.title(f'''
Name: {return_disease_plant_name(unique_diseases[labels[i].argmax()])[0]}
Disease: {return_disease_plant_name(unique_diseases[labels[i].argmax()])[1]}
          ''')
    plt.axis('off')
  plt.tight_layout()
```

The code below performs model selection and model creation. It defines and compiles a model using transfer learning, but does not train or evaluate it yet.
The function create_model() creates a neural network model using TensorFlow and Keras. Then we load a pre-trained model using the hub.KerasLayer(MODEL_URL), where MODEL_URL is the URL to the pre-trained model. Post this, add a Dense layer to the model with units = OUTPUT_SHAPE and activation = 'softmax'. Now compile the model using model.compile() with the CategoricalCrossentropy loss function, Adam optimizer, and accuracy metric. This builds the model with a specified input shape using model.build(INPUT_SHAPE). Finally, it returns the compiled and built model.

```
# creating model
def create_model():
  ''' Create a deep learning model for image classification using MobileNetV3 as a base.

    This function constructs a Sequential model using the TensorFlow Keras API. The base of the model is MobileNetV3,
    a pre-trained model available through TensorFlow Hub. The model is compiled with the CategoricalCrossentropy loss
    function, Adam optimizer, and accuracy as the evaluation metric.

    Returns:
        tf.keras.Model: The created deep learning model.

    Example:
        # Create the model
        model = create_model()
  '''
  show_download_symbol(duration = 2, interval = 0.2, waiting_message = 'Building model using Mobile Net V3')

  # setting  up model layers
  model = tf.keras.Sequential([
        hub.KerasLayer(MODEL_URL, input_shape=INPUT_SHAPE),
        tf.keras.layers.Dense(units=OUTPUT_SHAPE, activation='softmax')
    ])
  # compiling model
  model.compile(loss = tf.keras.losses.CategoricalCrossentropy(), optimizer = tf.keras.optimizers.Adam(), metrics = ['accuracy'])

  # building model
  model.build(INPUT_SHAPE)

  return model
```

It prints a summary of the deep learning model architecture created using MobileNetV3 and a dense output layer. The function create_model() builds a deep learning model using a pre-trained

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

MobileNetV3 backbone and adds a new classification layer. This displays a summary of the model's architecture using model.summary(). The model.summary() displays the architecture and parameter count of the model. So we can see that the total Parameters were 5,546,789, out of which Trainable Parameters were 38,076. Only the new Dense layer (used for fine-tuning your specific task). The non-trainable Parameters are 5,508,713. These come from MobileNetV3 and are frozen (not updated during training)

```
model = create_model()
model.summary()
```

```
Building model using Mobile Net V3 \

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1001)              5508713

 dense (Dense)               (None, 38)                38076

=================================================================
Total params: 5546789 (21.16 MB)
Trainable params: 38076 (148.73 KB)
Non-trainable params: 5508713 (21.01 MB)
```

5 . Model Deployment :

To ensure compatibility and reproducibility in the project, we verify the environment versions, debug the version mismatches, and log the dependencies for deployment or collaboration.

The code below defines two utility functions that help format strings and lists by cleaning and beautifying them—especially useful in data labelling or user interface presentation.

```
# checking tensorflow and tf hub version
print(f'''
1. TF hub version {hub.__version__}
2. TF version {tf.__version__}
'''
)

# latest version of both TF Hub and TF as of today
```

To do the system check to determine hardware capabilities, specifically before training large models (which may require a GPU), to optimize performance, and to configure runtime for TensorFlow, the following code is used. It checks for hardware available to TensorFlow.To check specifically for GPU, use tf.config.list_physical_devices('GPU').

tf.config.list_physical_devices() is a TensorFlow function that lists all physical devices available (CPU, GPU, etc.). If any devices are found (including CPU or GPU), it prints "GPU available". If no devices are found at all, it prints "No GPU available".

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
# GPU availability
if tf.config.list_physical_devices():
  print('GPU available')
else:
  print('No GPU available')
```

6. Model Training:

The pre-processed dataset will be split into training and validation sets. The training set will be used to train the MobileNet V3 model, while the validation set will be used to monitor the model's performance and tune hyperparameters. During training, the model will learn to classify images into the 38 unique classes on the provided labels. TensorBoard will be utilized to create logs and track training metrics for better analysis. Additionally, Early Stopping with a patience of 3 epochs will be employed during training. We will also be using a library called mlnotify. mlnotify is a powerful library designed for machine learning practitioners to monitor and track their model training progress directly on their smartphones. Finally, the preliminary model would only be trained on 10000 images to ensure that everything is working properly. After a preliminary design is created, the entire dataset will be utilized. This is done in a bid to save time.

The code below is preparing and aligning data (features and labels) for training and validation in an image classification AI model, likely for plant disease detection. This is data preparation before training**.** It ensures the input features (X_train) and output labels (y_train) are aligned and correctly formatted. This is a critical part of model training, where data is encoded (labels to one-hot), **f**eatures and labels are aligned, and Inputs to the model are structured properly.

```
# preparing labels, boolean_labels
training_labels = np.array(training_labels_csv['Label'])
training_boolean_labels = [label == unique_diseases for label in training_labels]


validation_labels = np.array(valid_labels_csv['Label'])
validation_boolean_labels = [label == unique_diseases for label in validation_labels]
```

```
len(training_labels), len(training_boolean_labels), len(validation_labels), len(validation_boolean_labels)
```

```
(70295, 70295, 17572, 17572)
```

```
training_labels[:10]
```

```
array(['Apple___Black_rot', 'Apple___Apple_scab', 'Grape___healthy',
       'Tomato___Leaf_Mold', 'Apple___Black_rot', 'Tomato___Leaf_Mold',
       'Blueberry___healthy', 'Corn_(maize)___Common_rust_',
       'Cherry_(including_sour)___Powdery_mildew', 'Peach___healthy'],
      dtype=object)
```

```
np.argmax(training_boolean_labels[20]), (unique_diseases.tolist()).index(training_labels[20])
```

```
(30, 30)
```

```
# setting up features and labels for validation and training respectively
X_train, X_val, y_train, y_val = training_labels_csv['FilePath'], valid_labels_csv['FilePath'], training_boolean_labels, validation_boo
```

```
unique_diseases[np.argmax(training_boolean_labels[20])], X_train[20]
```

```
('Tomato___Late_blight',
 '/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New Plant Diseases Dataset(Augment
ed)/train/Tomato___Late_blight/c95eaeb8-7673-4411-bbbe-b2464526f303___RS_Late.B 6233.JPG')
```

```
len(X_train), len(X_val), len(y_train), len(y_val)
```

```
(70295, 17572, 70295, 17572)
```

Data preparation is a core part of model training, making sure inputs and outputs are correctly formatted. The code below focuses on preparing and organizing labeled training and validation data

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

for an image classification task for a plant disease detection model using images. training_labels[:10] displays the first 10 labels from the training_labels array.
This line sets up X_train and X_val: Lists of file paths to training and validation images, and y_train and y_val: Corresponding one-hot encoded labels. These will be used as input and output for the model during training and validation.

```
training_labels[:10]
```

```
array(['Apple___Black_rot', 'Apple___Apple_scab', 'Grape___healthy',
       'Tomato___Leaf_Mold', 'Apple___Black_rot', 'Tomato___Leaf_Mold',
       'Blueberry___healthy', 'Corn_(maize)___Common_rust_',
       'Cherry_(including_sour)___Powdery_mildew', 'Peach___healthy'],
      dtype=object)
```

```
np.argmax(training_boolean_labels[20]), (unique_diseases.tolist()).index(training_labels[20])
```

```
(30, 30)
```

```
# setting up features and labels for validation and training respectively
X_train, X_val, y_train, y_val = training_labels_csv['FilePath'], valid_labels_csv['FilePath'], training_boolean_labels, validation_boo
```

```
unique_diseases[np.argmax(training_boolean_labels[20])], X_train[20]
```

```
('Tomato___Late_blight',
 '/content/drive/MyDrive/Colab Notebooks/plant-health-project/New Plant Diseases Dataset(Augmented)/New Plant Diseases Dataset(Augment
ed)/train/Tomato___Late_blight/c95eaeb8-7673-4411-bbbe-b2464526f303___RS_Late.B 6233.JPG')
```

```
len(X_train), len(X_val), len(y_train), len(y_val)
```

```
(70295, 17572, 70295, 17572)
```

For the process of turning the image into tensors, the following steps are followed: the file path of the image is given as the input. Using TensorFlow to read the image file and store it in a variable called image. The image is resized to a desired size. In our case, this would be 224, 224 (height, width) since the model was trained on images of the said size. After this, the images are normalized by scaling their pixel values. The normalised images will be converted into a tensor using tf.constant() . This returns the processed image tensor as the output.

```
# turning images into tensors

# an example image from the dataset -->

image = tf.constant(imread(training_file_paths[42]))
image.shape
```

```
TensorShape([256, 256, 3])
```

```
def process_image(image_path):
    """
    Preprocesses an image file.

    Args:
        image_path (str): The file path of the image.

    Returns:
        tf.Tensor: Preprocessed image tensor.

    Preprocesses an image file by reading the image from the specified file path, decoding it as a JPEG image,
    normalizing the pixel values, and resizing the image to a target size of (224, 224) pixels.

    Returns the preprocessed image as a tensor.

    """
    # read in the image file from the filepath
    image = tf.io.read_file(image_path)
    # decode the image to a jpeg
    image = tf.image.decode_jpeg(image, channels=3)
    # image normalization
    image = tf.image.convert_image_dtype(image, tf.float32)
    # resize the image
    image = tf.image.resize(image, size=(224, 224))

    return image

def get_image_label(image_path, label):
    """
    Loads and preprocesses an image from the given file path and associates it with the corresponding label.

    Args:
        image_path (str): The file path of the image.
        label (as a boolean label): The label for the image.

    Returns:
        Tuple[tf.Tensor, boolean label]: A tuple containing the preprocessed image tensor and the associated label.
    """
    image = process_image(image_path)
    return (image, label)
```

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The code below converts the raw file paths and labels into preprocessed batched tf.data.Dataset objects. It supports training**,** validation**,** and testing modes. The function converts image file paths and labels into batches of image tensors using:

- process_image() → for test data (no labels)
- get_image_label() → for training/validation data (images + labels)

```
def get_data_batches(X, y = None, batch_size = 32, valid_data = False, test_data = False, deployment = False):
  """
    Create batches of data for training, validation, or testing.

    Parameters:
        X (numpy array): Input data (image file paths) as a NumPy array.
        y (numpy array, optional (not given when creating batched for test data)): Labels for the input data. Default is None.
        batch_size (int, optional): Size of each batch. Default is 32.
        valid_data (bool, optional): If True, create batches for validation data. Default is False.
        test_data (bool, optional): If True, create batches for test data. Default is False.
        deployment (bool, optional): If True, don't show download symbol. Default is False.

    Returns:
        tf.data.Dataset: A TensorFlow Dataset containing the data in batches.

    Note:
        - If `test_data` is True, the function creates batches for test data without labels.
        - If `valid_data` is True, the function creates batches for validation data with labels.
        - If both `valid_data` and `test_data` are False, the function creates batches for training data with labels.
        - The input data (X) and labels (y) are converted to TensorFlow constant tensors.
        - The `process_image` and `get_image_label` functions are used to process the images and labels.
        - For training data, the dataset is shuffled before creating batches. This is in a bid to remove bias
    """
  if test_data:
    if deployment:
      data = tf.data.Dataset.from_tensor_slices(tf.constant(X))
      data_batch = data.map(process_image).batch(batch_size)
    else:
      show_download_symbol(waiting_message = "Creating test data batches", duration = 2, interval = 0.01)
      data = tf.data.Dataset.from_tensor_slices(tf.constant(X))
      data_batch = data.map(process_image).batch(batch_size)
  elif valid_data:
    show_download_symbol(waiting_message='Creating valid data batches', duration = 2, interval = 0.02)
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), tf.constant(y)))
    data_batch = data.map(get_image_label).batch(batch_size)
  else:
    show_download_symbol(waiting_message = 'Creating a training data batch', duration = 2, interval = 0.4)
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), tf.constant(y)))
    data = data.shuffle(buffer_size = len(X))
    data_batch = data.map(get_image_label).batch(batch_size)

  return data_batch
```

The code below prepares and structures your training and validation datasets into batches, which is a key prerequisite before feeding them into your neural network. It ensures the data is clean, normalized, batched, and labeled properly for efficient training and validation.

get_data_batches(X_train, y_train): This creates training data batches from the training features (X_train) and labels (y_train). Internally, it processes images, applies transformations, and batches them for training.

get_data_batches(X_val, y_val, valid_data=True):This generates validation data batches.The flag valid_data=True tells the function that these are not training batches, but validation ones.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
train_data = get_data_batches(X_train, y_train)
val_data = get_data_batches(X_val, y_val, valid_data = True)
```

```
Creating a training data batch \

Creating valid data batches \
```

```
val_data.element_spec
```

```
(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
 TensorSpec(shape=(None, 38), dtype=tf.bool, name=None))
```

```
train_data.element_spec
```

```
(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
 TensorSpec(shape=(None, 38), dtype=tf.bool, name=None))
```

To monitor and control the system, the code below helps. Function create_tensorboard_callback () defines the function to create a TensorBoard callback for logging training metrics. It generates a log directory path based on the current timestamp using datetime.datetime.now().strftime("%Y%m%d-%h%m%s"). Finally, it returns a TensorBoard callback object pointing to the log directory.
Call backs are helper functions that a model can use during training to do such things as save its progress, check its progress, and terminate training if the model
stops improving. We'll create two call-backs

- Tensorboard- checking progress and logs
- Early stopping call back

```
%load_ext tensorboard
```

```
def create_tensorboard_callback():
  # create a log directory for storing Tensorboard logs
  current_time = datetime.datetime.now()
  log_dir = os.path.join("/content/drive/MyDrive/Colab Notebooks/plant-health-project/Logs", f"{current_time.day}-{current_time.month}-
  return tf.keras.callbacks.TensorBoard(log_dir, histogram_freq = 1, write_graph = True)
```

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor = 'val_accuracy', patience = 3)
```

```
# training the model
NUM_EPOCHS = 1#@param {type:"slider", min:1, max:100, step:2}
```

This code defines and executes the training phase of a deep learning model, integrating monitoring and early stopping mechanisms to improve efficiency and avoid overfitting. It is early stopping to prevent overfitting.TensorBoard is used to track performance and validation data to monitor accuracy during training. It wraps the full model training logic into a clean and reusable function.
epochs = NUM_EPOCHS sets how many times the dataset is passed through, and validation_data = val_data validates the model's performance during training. Callbacks add TensorBoard logging and early stopping mechanisms.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
def train_model():
  """Trains a given deep learning model on the provided training data.

    This function takes a deep learning model, provided by the `create_model` function, and trains it on the given
    training data. The training process includes the use of validation data for monitoring model performance during
    training. The training can be stopped early based on a specified `early_stopping` callback.

    Args:
        train_data (numpy.ndarray or tf.data.Dataset): The training data.
        val_data (numpy.ndarray or tf.data.Dataset): The validation data used for monitoring performance during training.
        NUM_EPOCHS (int): The number of training epochs.
        early_stopping (tf.keras.callbacks.EarlyStopping): A callback that monitors a certain metric to stop training early
                                                           if no improvement is observed.

    Returns:
        tf.keras.Model: The trained deep learning model.
  """
  model = create_model()
  tensorboard = create_tensorboard_callback()
  model.fit(x = train_data, epochs = NUM_EPOCHS,
            validation_data = val_data, validation_freq = 1,
            callbacks = [tensorboard, early_stopping])
  return model
```

The code snippet below handles training, saving, and loading of a deep learning model in TensorFlow using Kerasso. This can be reused, shared, or deployed without retraining. It wraps the full model training logic into a clean and reusable function. The train_model() function trains the model using the defined architecture and returns the trained model. The save_model() function saves the trained model in .h5 format to a specified directory on Google Drive. A timestamp and optional suffix are added to make the filename unique and descriptive. The load_model() function retrieves a previously saved model for future use or deployment. It ensures any custom layers (like KerasLayer from TensorFlow Hub) are properly loaded. %%notify from mlnotify is used to notify when long-running processes (like training) are complete.

```
%%notify
mlnotify.start()
model = train_model()
save_model(model, suffix = 'complete-model-2-epochs')
```

```
model
```

```
def save_model(model, suffix=None):
  """
  Saves a given model in a models directory and appends a suffix (str)
  for clarity and reuse.
  """
  # Create model directory with current time
  current_time = datetime.datetime.now()
  modeldir = os.path.join("/content/drive/MyDrive/Colab Notebooks/plant-health-project/",
                          )
  model_path = modeldir + "-" + suffix + ".h5" # save format of model
  show_download_symbol(duration  = 2, interval = 0.4, waiting_message = f'saving model...')
  model.save(model_path)
  print('Successful')
  return model_path

def load_model(model_path):
  """
  Loads a saved model from a specified path.
  """
  show_download_symbol(duration  = 2, interval = 0.4, waiting_message = f'loading model...')
  model = tf.keras.models.load_model(model_path,
                                     custom_objects={"KerasLayer":hub.KerasLayer})
  print('Successful')
  return model
```

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The code snippet performs key tasks from the model management and evaluation phase. The code below sets up training and validation data using the get_data_batches() function. It prepares the validation dataset to evaluate how well the model generalizes to unseen data. It saves a trained model using HDF5 format (.h5), though a warning suggests using the newer .keras format. Then load the trained model from storage for further use or evaluation. Once it is loaded, it initializes TensorBoard to visualize training logs like accuracy and loss. Finally, it prepares validation data batches using the get_data_batches() function for model evaluation.

```
save_model(model, suffix = 'initial-model')
```

```
saving model... \
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning: You are saving your model as an HDF5 file via `m
odel.save()`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.kera
s')`.
  saving_api.save_model(
Successful
```

```
'/content/drive/MyDrive/Colab Notebooks/plant-health-project/-initial-model.h5'
```

```
model = load_model("/content/drive/MyDrive/Colab Notebooks/plant-health-project/-complete-model-2-epochs.h5")
```

```
model
```

```
%tensorboard --logdir "/content/drive/MyDrive/Colab Notebooks/plant-health-project/Logs"
```

```
Output hidden; open in https://colab.research.google.com to view.
```

```
X_train, X_val, y_train, y_val = training_labels_csv['FilePath'], valid_labels_csv['FilePath'][:80], training_boolean_labels, validatio
val_data = get_data_batches(X_val, y_val, valid_data = True)
```

```
Creating valid data batches \
```

7. Model Evaluation:

After training, the performance of the model will be evaluated using the validation set itself. Further evaluation would be done by creating preds_df. The accuracy of the model in classifying the plant diseases will be assessed by comparing its predictions to the ground truth label.

In addition to accuracy, we will employ the multi-class log loss as an evaluation metric. The multi-class log loss, also known as cross-entropy loss, measures the performance of the model by considering the probabilities assigned to each class. It penalizes the model for assigning low probabilities to the correct classes and rewards confident and accurate predictions. The formula for multi-class log loss is expressed as:

$$\log \text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

Where

- N is the number of samples in the test set.
- M is the number of unique plant diseases in the dataset.
- (y_{ij}) is the binary indicator (0 or 1) if the sample belongs to the class.
- (p_{ij})is the predicted probability by the model that sample i belongs to class (j).

The code below helps to evaluate the model's performance using the validation dataset and prepare data for further analysis or visualization. The data set is broken into individual image-label pairs. Each image is converted and labelled (tensors for TensorFlow) into a NumPy array for easy handling. Then the labels are converted to show the disease name.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

zipped_lists = list(zip(true_images, true_labels, predictions)) combines true images, labels, and predictions into a single list using zip(). Then it shuffles the list (with fixed seed for reproducibility), unzips them back into separate shuffled lists—ready for visualization or error analysis.

```
# making predictions on the validation data
predictions = model.predict(val_data)

# creating true labels and true images
unbatched_val_data = val_data.unbatch()
true_images = []
true_labels = []

for image, label in unbatched_val_data:
    true_images.append(image.numpy())  # Convert the TensorFlow tensor to a numpy array
    true_labels.append(unique_diseases[np.argmax(label.numpy())])
```

```
# shuffling images
# Zip the three lists together
np.random.seed(43)
zipped_lists = list(zip(true_images, true_labels, predictions))

# Shuffle the zipped list
np.random.shuffle(zipped_lists)

# Unzip the shuffled list to get the shuffled lists back
true_images_shuffled, true_labels_shuffled, predictions_shuffled = zip(*zipped_lists)
```

By utilizing the multi-class log loss, we can assess the model's performance in a more detailed manner, considering the confidence and accuracy of its predictions across all disease classes.

7. Deployment: Once the model has been trained and evaluated, it can be deployed to classify plant disease in our custom dataset. The system can take an input image of a plant's leaf, apply the trained Mobile Net V3 model, and predict the most likely class from the 38 unique classes in the dataset. For the proof-of-concept stage, we will be utilizing Gradio. The code below enhances user experience during deployment or data loading stages. The image defines a utility function named show_download_symbol, which creates a visual spinner animation in the terminal to simulate a “downloading” or “loading” effect. It provides feedback to the user while a background process (e.g., model download, dataset loading, file transfer) is occurring.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
def show_download_symbol(duration, interval, waiting_message):
    '''
    Displays a rotating downloading symbol for a specified duration.

    Parameters:
        - duration (float): The total duration (in seconds) to display the rotating symbol.
        - interval (float): The time interval (in seconds) between each rotation.
        - waiting_message (str): The message to display while waiting.

    Returns:
        None

    Displays a rotating downloading symbol, consisting of '|', '/', '-', and '\',
    to indicate an ongoing download process. The symbol rotates
    for the specified duration with the given time interval
    between each rotation. The waiting message is displayed
    alongside the symbol during the waiting period.

    Example usage:
        show_download_symbol(10, 0.5, "Downloading file...")
    '''
    iterations = int(duration / interval)

    for i in range(iterations):
        print(f"\r{waiting_message} |", end="")
        time.sleep(interval / 4)
        print(f"\r{waiting_message} /", end="")
        time.sleep(interval / 4)
        print(f"\r{waiting_message} -", end="")
        time.sleep(interval / 4)
        print(f"\r{waiting_message} \\ ", end="")
        time.sleep(interval / 4)
    print("\n")
```

During this part of the evaluation, it helps visually assess model performance and **identify** misclassifications and confidence levels. This is useful for debugging and model improvement. It is used for visualizing the model's predictions on a random set of validation images, comparing them with their actual labels, and showing the prediction confidence. It serves as a powerful qualitative evaluation tool. The loop iterates over 12 shuffled validation images and displays each image in a subplot. If the predicted label matches the actual label, set the title color to green (correct prediction); otherwise, set it to red (incorrect prediction).

IV. Results And Evaluation

To evaluate the performance of a trained AI model, the code below displays 12 random images from the validation set, along with their predicted label (what the model thinks the image shows), Actual label (the correct class), and Prediction probability (confidence of the model). This code helps to interpret model predictions, identify correct vs incorrect predictions, and visually inspect performance and possible misclassifications.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
# plotting 12 random photos with their prediction probabilities, predicted labels and actual labels

plt.figure(figsize=(10, 10))
plt.suptitle('12 random photos from the validation data!', fontsize=16, fontweight='bold')

for i in range(12):  # To display only 10 images
    ax = plt.subplot(4, 3, i + 1)
    ax.imshow(true_images_shuffled[i])

    if true_labels_shuffled[i] == unique_diseases[np.argmax(predictions_shuffled[i])]:
        title_color = 'green'
    else:
        title_color = 'red'

    ax.set_title(f'''
Predicted label:
Name: {return_disease_plant_name(unique_diseases[np.argmax(predictions_shuffled[i])])[0]}
Disease: {return_disease_plant_name(unique_diseases[np.argmax(predictions_shuffled[i])])[1]}
Actual label:
Name: {return_disease_plant_name((true_labels_shuffled[i]))[0]}
Disease: {return_disease_plant_name((true_labels_shuffled[i]))[1]}
Prediction probability: {(max(predictions_shuffled[i]) * 100):.2f}%
        ''',
           color=title_color
           )

    plt.axis('off')

plt.tight_layout()
plt.show()
```

Below are nine random photos from the validation data: The visual output is generated by the above code, which was used for evaluating a machine learning image classification model. **12** images from the validation dataset are arranged in a 4x3 grid. Each image displays:

- **Predicted label** (Plant type and disease condition).
- **Actual label** (Ground truth from the validation data).
- **Prediction probability** (Confidence level of the model).

All predictions here are correct (title color is green), indicating good model performance.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

This code processes the predictions from a trained machine learning model to extract the top 5 predicted classes (diseases) for each sample. To populate the data for x and y, where x is the 2D list containing the top 5 plants and y is the 2D list showing the corresponding probability for each prediction:

```
x = []
y = []

for prediction in predictions_shuffled:
  top_5_diseases = []
  top_5_predictions = []
  argsorted_prediction_reversed = (np.argsort(prediction)[::-1])[:5]
  for index in argsorted_prediction_reversed:
    top_5_diseases.append(unique_diseases[index])
    top_5_predictions.append(prediction[index])
  x.append(top_5_diseases)
  y.append(top_5_predictions)
```

```
x[0]
```

```
['Corn_(maize)___Cercospora_leaf_spot Gray_leaf_spot',
 'Corn_(maize)___Northern_Leaf_Blight',
 'Strawberry___healthy',
 'Corn_(maize)___Common_rust_',
 'Apple___Black_rot']
```

Below is the code to create the prediction graph of size 15815 inches. This code helps to analyze and interpret the model's predictions, assessing how well the model performs, both quantitatively (accuracy) and visually (confidence, top predictions). We assume you want to plot results for 12 data points**.** ax = plt.subplot(4, 3, index + 1) creates a 4×3 grid of subplots. It fills plots one by one (12 total). ax.plot(x[index], y[index], label='probability of prediction plots the top 5 predicted class labels (x[index]) vs their probabilities (y[index]).
ax.set_title(f"True label: {format_underscores(true_labels_shuffled[index])}", color=color)
Displays the true label as the subplot title, and finally, ax.set_xticklabels(x[index], rotation=45)makes x-axis labels (class names) easier to read.

```
# Create a new figure with a size of 15x15 inches
plt.figure(figsize=(15, 15))

# Set the main title for the entire visualization
plt.suptitle('''Top 5 predictions along with corresponding probabilities for the previous plot''')

# Loop through 12 data points (assumed) to create subplots
for index in range(12):
    # Create a new subplot at the index position
    ax = plt.subplot(4, 3, index + 1)

    # Plot the data for the current index
    ax.plot(x[index], y[index], label='probability of prediction')

    # Determine the color of the title based on true and predicted labels
    if true_labels_shuffled[index] == unique_diseases[np.argmax(predictions_shuffled[index])]:
        color = 'green'
    else:
        color = 'red'

    # Set the title for the current subplot
    ax.set_title(
        f''' True label: {format_underscores(true_labels_shuffled[index])}''',
        color=color
    )

    # Rotate the x-axis tick labels by 45 degrees for better readability
    ax.set_xticklabels(x[index], rotation=45)

# Adjust the spacing between subplots for better layout
plt.subplots_adjust(wspace=0.4, hspace=1.2)

# Show the final plot with all subplots
plt.show()
```

```
<ipython-input-86-a546e031a9a3>:28: UserWarning: FixedFormatter should only be used together with Fix
  ax.set_xticklabels(x[index], rotation=45)
```

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Below is a **visual output** of the code block. Each of the **12 subplots** corresponds to one of the **12 data samples** being evaluated, showing x and y values. Using ax.plot(x[index], y[index]), the graphs are plotted as shown below. All titles are **green**, indicating that the model predicted the correct disease (true label matches top-1 prediction).

Below is the function that is a wrapper for deployment that accepts an image input (as a NumPy array), saves it temporarily, performs prediction using a trained model, cleans up the temp file, and returns a user-friendly result. It's designed for use with **Gradio**, a tool for creating interactive web UIs. This function makes the model accessible to users by wrapping it in a callable pipeline. A Gradio interface was created to allow users to upload leaf images and receive instant predictions. The model returns plant Name, predicted Disease, and confidence Score. Gradio supports integration into mobile or web apps, making this model deployable in real farming contexts.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

```
def complete_model_pipeline(image):
    """
    Complete model pipeline for deploying the model using Gradio.

    This function takes an image as input, saves it temporarily to a file,
    loads the image as a test sample, performs prediction using the model,
    and then removes the temporarily saved image. The function returns the
    predicted label

    Args:
        image (np.ndarray): The NumPy array representing the input image.

    Returns:
        str: The predicted label.

    Parameters:
        - image: A NumPy array representing the input image.

    Note: This function is designed to be used with Gradio to deploy the model
    and obtain predictions interactively.

    """

    # Define the file path to temporarily save the image
    image_file_path = '/content/drive/MyDrive/Colab Notebooks/plant-health-project/myImage.jpg'

    # Convert the input NumPy array to a Pillow image object and save it as a file
    image = Image.fromarray(image)
    image.save(image_file_path)

    # Prepare test data for the model using the saved image file
    test_data = get_data_batches(X=[image_file_path], test_data=True, batch_size=1, deployment = True)

    # Perform prediction using the model
    predictions = model.predict(test_data, verbose = 0)

    # Get the index of the highest predicted value and obtain the corresponding breed label
    predicted_label_index = np.argmax(predictions[0])
    predicted_breed_label = unique_diseases[predicted_label_index]

    # Remove the temporarily saved image file
    os.remove(image_file_path)
    result = return_disease_plant_name(predicted_breed_label)
    border = '-' * 40  # Adjust the number of hyphens as needed
    formatted_string = f'''
+----------------------------------------+
|  Name of your plant: {result[0]:<20}
|  Disease your plant is facing: {result[1]:<20}
+----------------------------------------+
'''
    return formatted_string
```

```
iface = gr.Interface(
    fn=complete_model_pipeline,
    inputs="image",
    outputs="text",
    title = "Plant Disease Classification Application",
    description = 'Upload the image of a plant with a disease and get to know which disease it is facing (or not facing :))'
)

iface.launch(share = True)
```

V. Conclusion And Future Work

This research demonstrates the feasibility of using deep learning for plant disease identification. The MobileNet V3-based system:

- Achieves reliable multi-class classification
- It is suitable for real-time deployment
- Can be scaled to include more classes and sensors (e.g., soil health)
- Battery + Solar hybrid power system for continuous operation.
- Weather forecasting API integration to complement local sensor data.

Received: August 06, 2025

International Journal of Applied Mathematics
Volume 38 No. 6s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

- Seed inventory sensors for refill alerts.

References

[1] New Plant Diseases Dataset. (2018). Kaggle. https://www.kaggle.com /datasets/vipoooool/new-plant-diseases-dataset
[2] Howard, A., et al. (2019). *Searching for MobileNetV3*. arXiv preprint arXiv:1905.02244.
[3] TensorFlow Hub.https://tfhub.dev
[4] Gradio: https://www.gradio.app/

Received: August 06, 2025