

DIP PROJECT

PLANT DISEASE DETECTON AMINA ANJUM 22I-2065 MARYAM AMJAD 22I-1924 MARYAM KHALID 22I-1917 KHADEEJA ALAM 22I-2063

Stage 1: Data Preparation Report

# Introduction

The objective of this project is to detect plant diseases from leaf images using deep learning. In the first stage, we focused on exploring the dataset, verifying its structure, visualizing distributions, applying augmentation techniques, handling class imbalance, and setting up the data pipeline to prepare the images for training a CNN-based model using transfer learning.

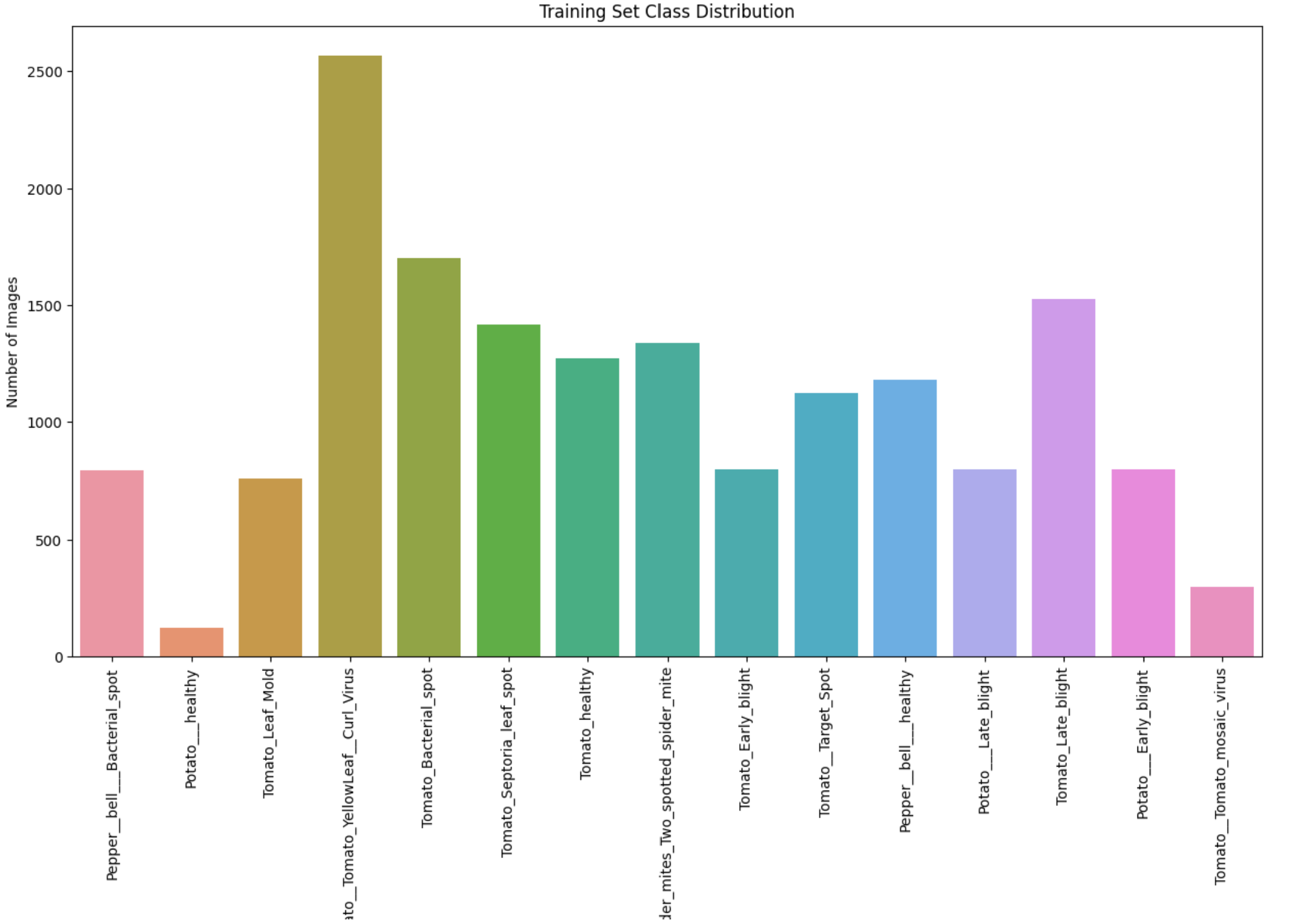
# Dataset and Structure

We used the publicly available PlantVillage dataset from Kaggle. It contains color images of healthy and diseased leaves across three crops: tomato, potato, and bell pepper. The dataset is organized into three folders—train, validation, and test—each containing 15 subdirectories, one per class. The training set contains 16,504 images, the validation set has 2,070, and the test set includes 2,064 images. Each image is labeled according to the disease or healthy class it belongs to. We verified the dataset structure by checking for the presence of all expected directories and ensuring that all 15 classes were present in each split.

There was a notable imbalance across classes. For instance, the class Tomato\_\_Tomato\_YellowLeaf\_\_Curl\_Virus had 2,566 images in the training set, while Potato\_\_\_healthy had only 121 images. This imbalance had to be addressed to prevent biased learning during model training.

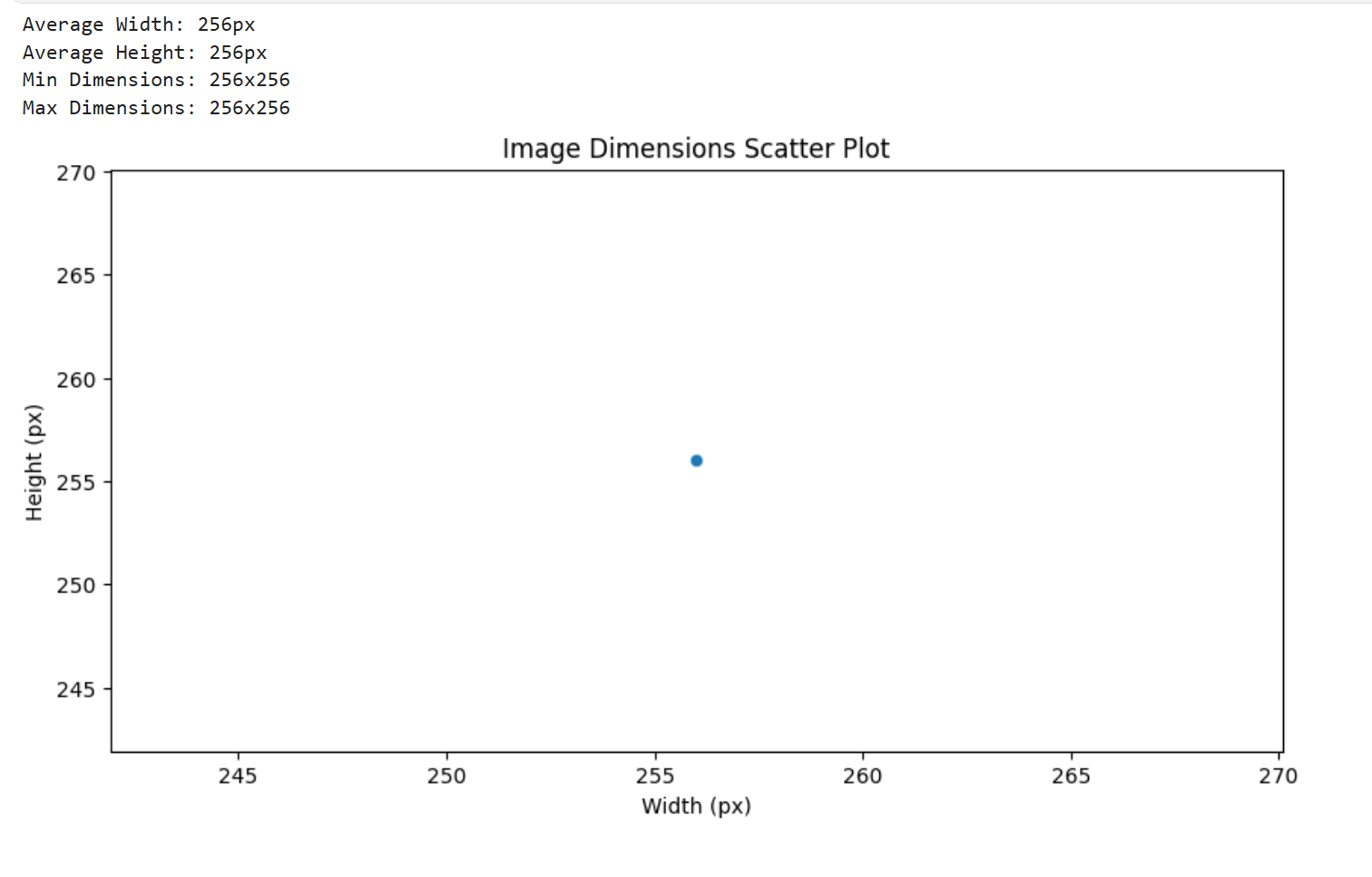
# Class Distribution Visualization

We visualized the training class distribution using a bar plot to understand the imbalance more clearly. The chart confirmed that several classes had disproportionately fewer images than others, which would affect model performance if not accounted for.



# Image Dimension Consistency

To ensure compatibility with convolutional layers and avoid resizing artifacts, we analyzed the dimensions of a random sample of images across all classes. All images were found to be uniformly sized at 256×256 pixels. This consistency eliminated the need for further resizing and allowed us to feed images directly into the model without distortion.



# Sample Image Display

We displayed sample images from three representative classes: Tomato\_\_Tomato\_YellowLeaf\_\_Curl\_Virus, Potato\_\_\_healthy, and Tomato\_Leaf\_Mold. The images were clear, focused, and visually captured relevant disease patterns, validating the dataset’s visual quality.



# Augmentation Strategy

To make the model robust against overfitting and improve its ability to generalize, we applied several augmentation techniques to the training set using Keras' ImageDataGenerator. These included random horizontal and vertical flips, rotations up to 20°, shear transformations, zooming up to 10%, brightness shifts, and reflective filling at the edges. Pixel values were also normalized to the range [0, 1]. For the validation and test sets, only normalization was applied to preserve the integrity of evaluation data.

# Class Weighting

Due to the class imbalance observed earlier, we calculated class weights using scikit-learn’s class\_weight utility. These weights were normalized to range between 1 and 2. Classes with fewer samples, such as Potato\_\_\_healthy, were assigned a higher weight (2.00), while frequent classes like Tomato\_\_Tomato\_YellowLeaf\_\_Curl\_Virus were assigned lower weights (1.00). This ensured that the model paid proportional attention to minority classes during training.

# Data Generator Configuration

We set up two generators: one for training with full augmentation and one for validation with only normalization. Both generators resized the input images to 256×256 pixels and batched them into groups of 32. The training generator shuffled data between epochs to avoid order bias, while the validation generator preserved the original order for consistent evaluation.

# Final Summary Before Training

Before moving on to training, we verified that the training generator loaded 16,504 images and the validation generator loaded 2,070 images across 15 classes. The final class weights were prepared and passed to the model using Keras’ fit() method, which would be executed in the next stage.

python

model.fit(

train\_generator,

validation\_data=val\_generator,

class\_weight=class\_weights

)

With all preprocessing steps complete—structured dataset, balanced labels, augmented batches, and verified input shape—we concluded Stage 1 and moved forward to model development and training in Stage 2.

Stage 2: Transfer Learning Using EfficientNetB0

In the second stage of our plant disease classification project, we implemented a transfer learning approach using the EfficientNetB0 model pre-trained on ImageNet. This approach allowed us to leverage learned image features and fine-tune the model on our custom dataset, significantly reducing training time and improving performance compared to training a model from scratch.

# Data Generators and Preprocessing

We began by setting up data generators using TensorFlow’s ImageDataGenerator. For the training set, a variety of data augmentations were applied including random rotations, shifts, zooming, horizontal flips, and brightness changes, all combined with EfficientNetB0’s native preprocess\_input function. These augmentations were chosen to improve generalization and prevent overfitting due to the relatively limited size of our dataset. For the validation and test sets, only preprocessing was applied, ensuring data consistency.

Train Generator: 16504 images from 15 classes

Validation Generator: 2070 images from 15 classes

Test Generator: 2064 images from 15 classes

# Model Architecture and Compilation

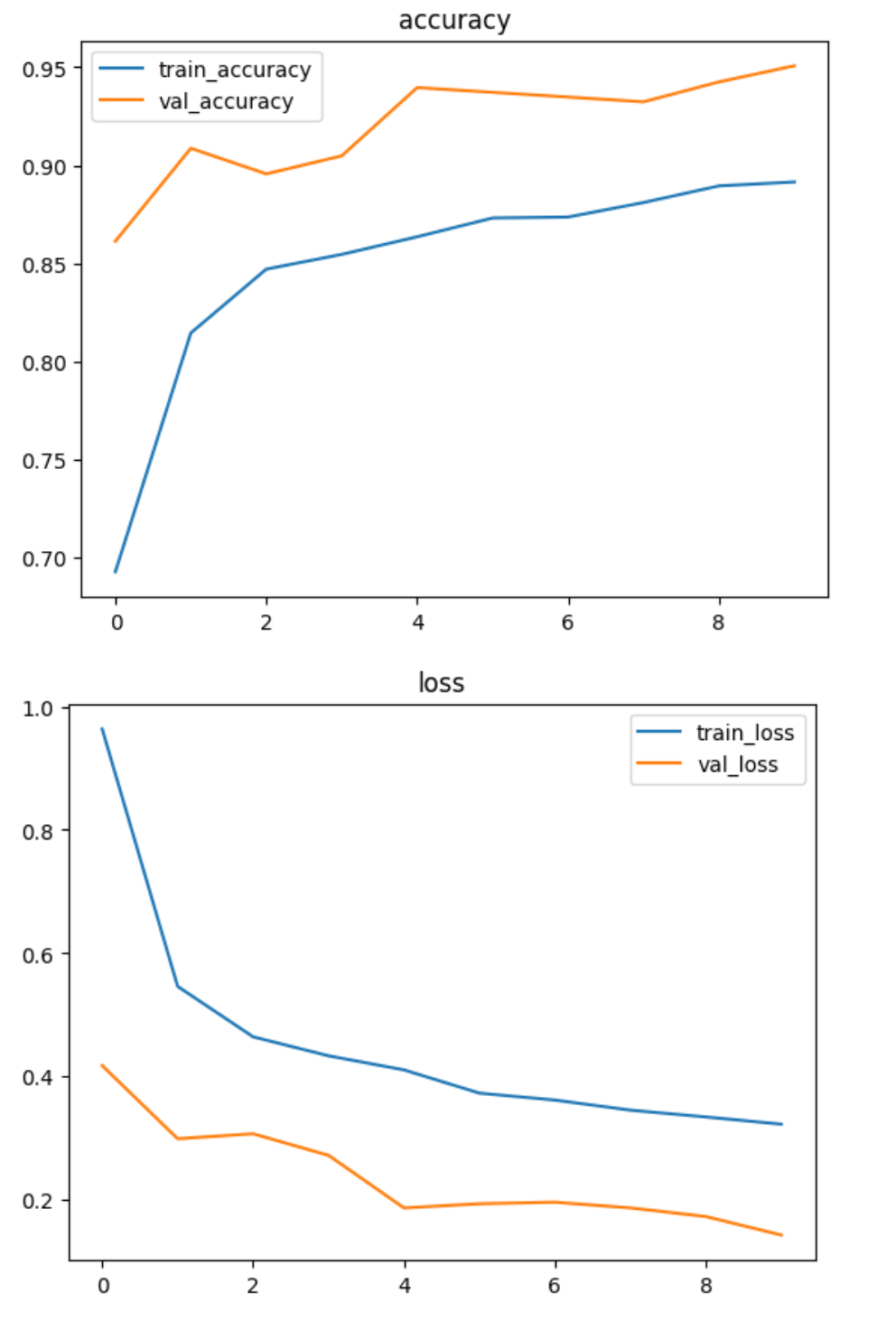
Next, we loaded the EfficientNetB0 model with include\_top=False to remove its final classification layers, allowing us to add our own custom classification head. The model was initialized with imagenet weights, and we froze all layers in the EfficientNet backbone to begin with. A new classification head was added consisting of a global average pooling layer, a dense layer with 256 ReLU units, a dropout layer (rate: 0.5), and a final dense layer with 15 softmax units representing the target classes.

The model was compiled using the Adam optimizer with a learning rate of 0.001, categorical crossentropy loss, and accuracy as the evaluation metric. Freezing the backbone ensures that only the new classification head is trained initially, which is ideal for adapting pre-trained features to the new task without distorting them early in the training process.

**Total Parameters:** 4,381,362  
**Trainable Parameters (head-only):** 331,791

## Initial Training (Head Only)

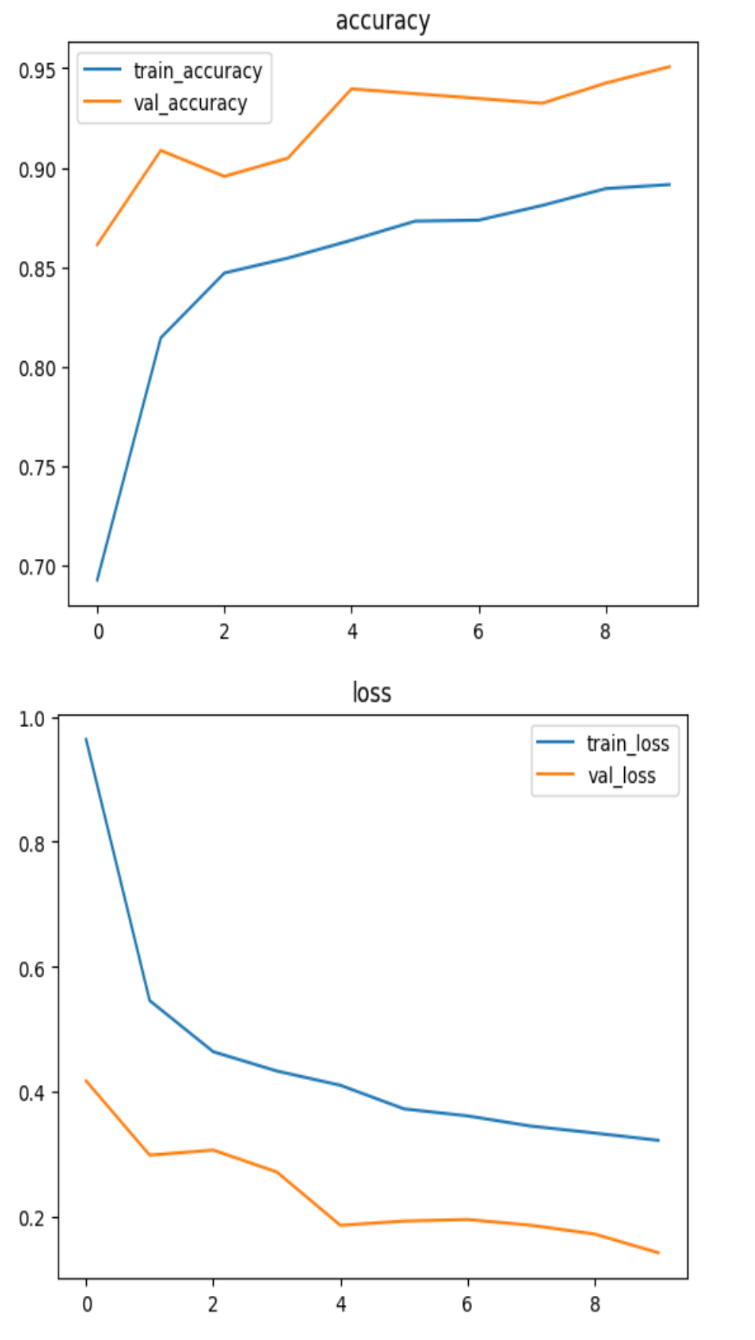
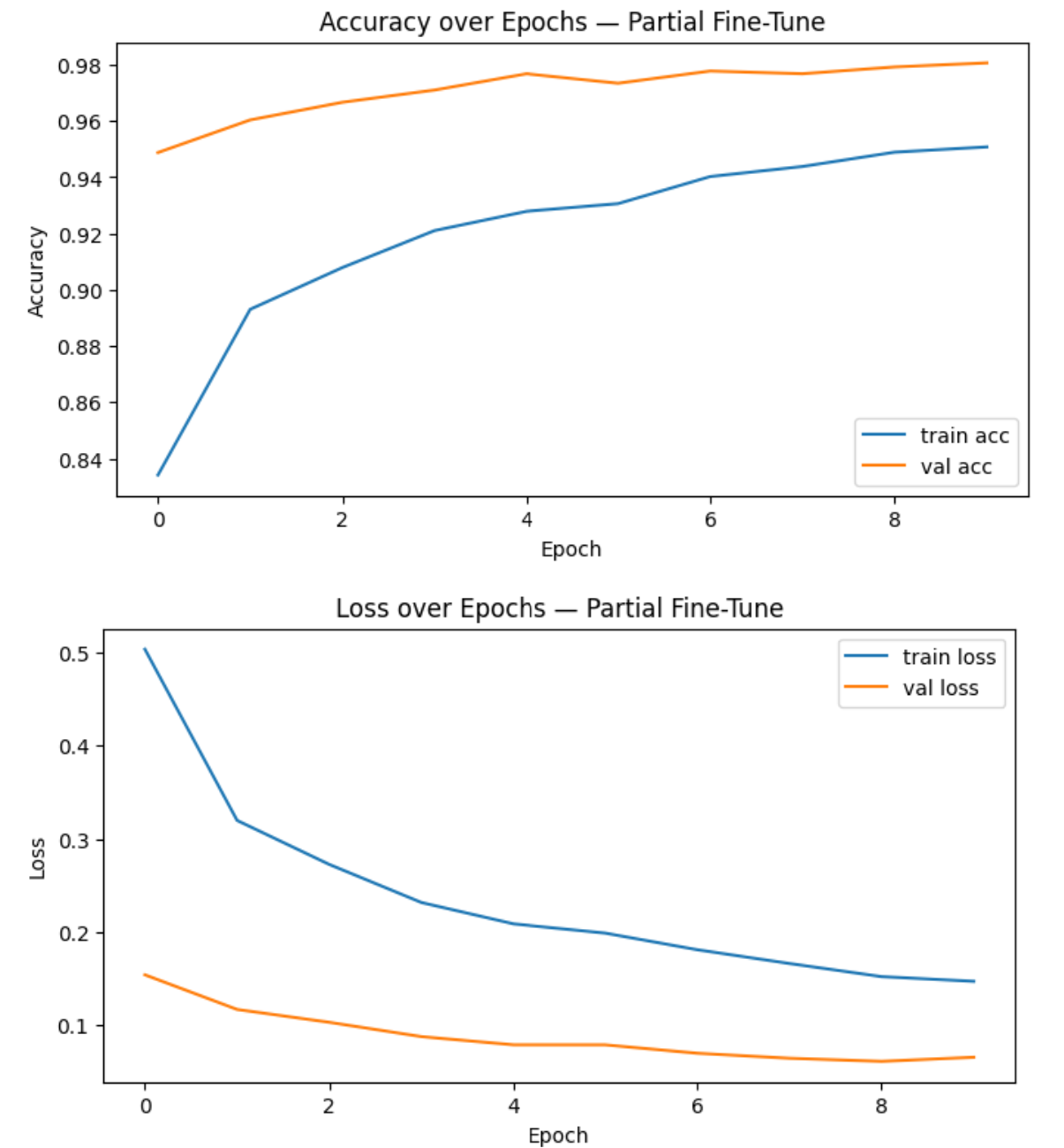
We trained the model for 10 epochs, tracking the val\_accuracy for early stopping and best model checkpointing. The training showed rapid improvement in validation accuracy, rising from an initial 86.1% to a peak of **95.07%** by the 10th epoch. This demonstrated that the classification head was successfully adapting to the leaf disease patterns using the frozen EfficientNet features.



## Fine-Tuning: Unfreezing Last 30 Layers

After the initial convergence, we unfroze the last 30 layers of the EfficientNetB0 backbone to allow partial fine-tuning. The learning rate was reduced to 5e-5 to prevent drastic updates that might overwrite valuable pre-trained weights. We retrained the model for 10 more epochs using the same callbacks.

Validation accuracy improved steadily, reaching a new peak of **98.06%** by the 10th epoch. This stage allowed the model to refine its high-level features to better suit the visual traits of diseased leaves in the PlantVillage dataset.



## Final Fine-Tuning: Unfreeze All Layers

For the final training phase, we unfroze the entire model and compiled it again with an even smaller learning rate of 1e-6. This ensures that all layers could participate in fine-tuning but at a pace slow enough to preserve previously learned patterns. We trained for 10 epochs, but early stopping halted the training at epoch 4 due to no further improvement in validation accuracy.

Though no improvement over 98.06% was achieved, the model maintained strong validation performance and generalization ability throughout.

**Final Validation Accuracy:** 94.39%  
**Test Accuracy:** 91.62%



## Final Model Saving

After training, we saved two versions of the model:

1. The best model (saved automatically by ModelCheckpoint during training).
2. The final model after full fine-tuning, saved explicitly as both .keras and .h5.

✅ Final transfer model saved to /kaggle/working/final\_transfer\_model.keras

# Final Evaluation and Performance Analysis

After training and saving the final transfer learning model, we conducted a comprehensive evaluation on the test dataset to assess generalization performance. The evaluation included calculating key metrics such as test accuracy and loss, plotting a confusion matrix, and generating a detailed classification report for each of the 15 classes.

## 1. Overall Model Evaluation

The final model was evaluated using the evaluate() function on the unseen test data. This provided the overall **categorical cross-entropy loss** and **classification accuracy**.

* **Test Loss**: 0.2507
* **Test Accuracy**: 91.62% (0.9162)

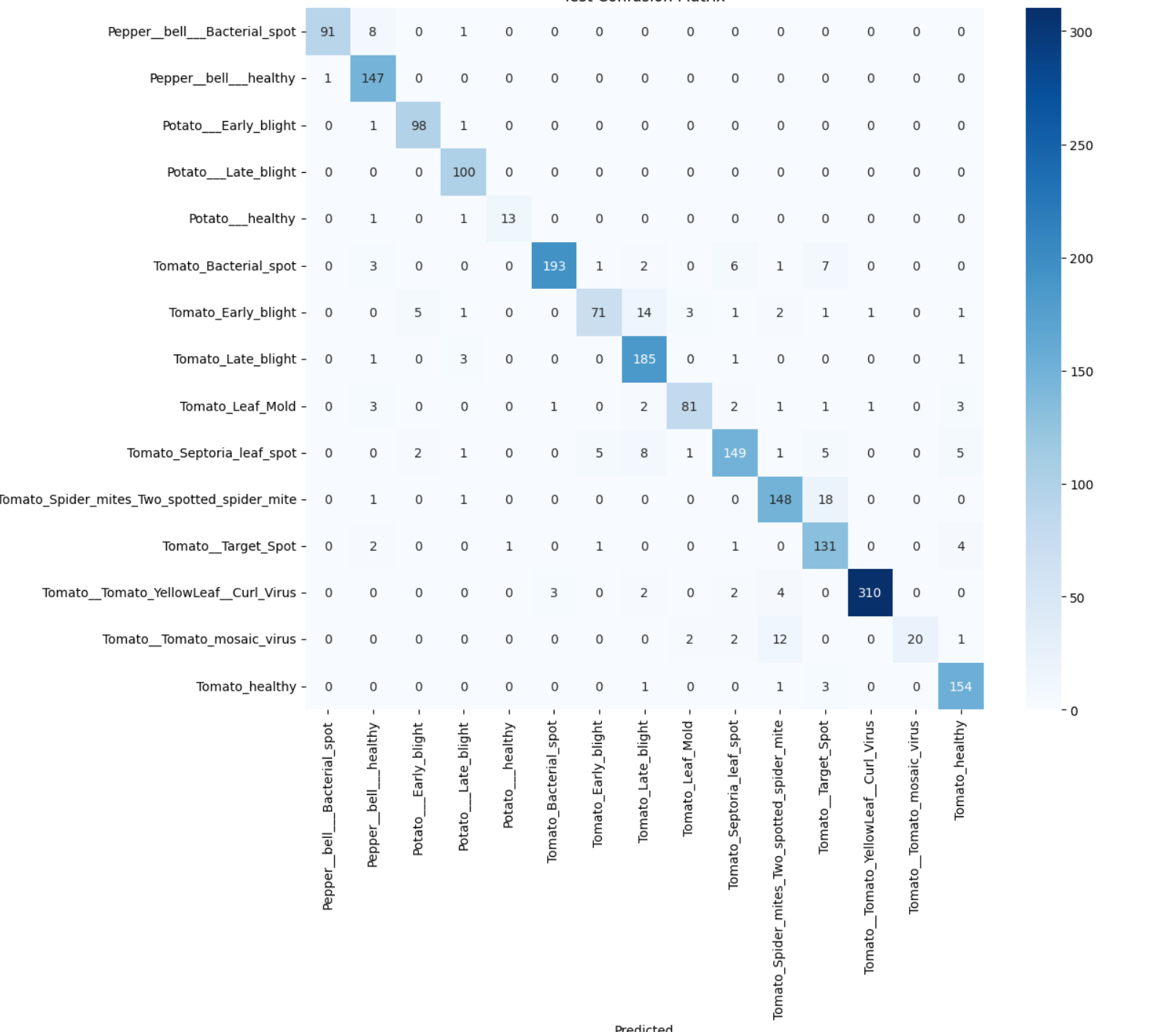
These results confirm that the model not only performed well during training and validation but also generalized effectively to completely unseen data.

## 2. Confusion Matrix Analysis

A confusion matrix was generated to analyze class-wise predictions and identify patterns of misclassification. The matrix compared the true class labels (y\_true) against the predicted labels (y\_pred) across all test images.

**Key Observations from Confusion Matrix**:

* Most diagonal cells (true positives) had high counts, indicating correct classifications.
* Minor misclassifications occurred between visually similar diseases (e.g., *Tomato\_Leaf\_Mold* vs *Tomato\_Septoria\_leaf\_spot*).
* Class *Tomato\_Tomato\_YellowLeaf\_Curl\_Virus* had the highest correct predictions (310/321).
* Class *Tomato\_Tomato\_mosaic\_virus* showed lower recall, with some images being predicted as other classes.



## 3. Classification Report (Precision, Recall, F1-score)

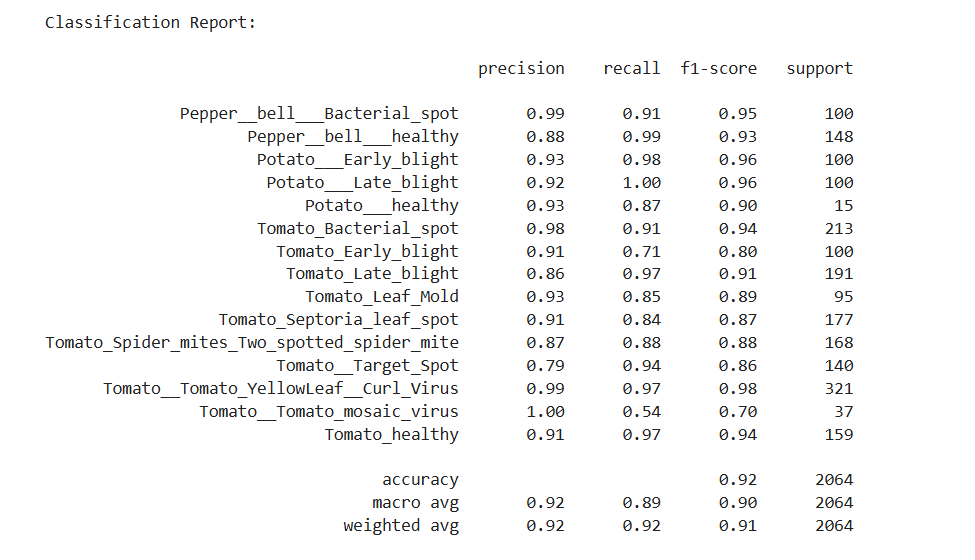
A classification report was generated to evaluate precision, recall, and F1-score for each class.

**Highlights from the Report**:

* **Precision**: Highest (≥ 0.99) for *Tomato\_Tomato\_mosaic\_virus* and *Tomato\_Tomato\_YellowLeaf\_Curl\_Virus*.
* **Recall**: Slightly lower for *Tomato\_Early\_blight* (0.71), indicating some confusion with similar diseases.
* **Macro Avg**:
  + Precision: 0.92
  + Recall: 0.89
  + F1-score: 0.90
* **Weighted Avg** (accounts for support size):
  + Precision: 0.92
  + Recall: 0.92
  + F1-score: 0.91

**Most Challenging Classes**:

* *Tomato\_Early\_blight* – low recall (71%), needs improvement.
* *Tomato\_Tomato\_mosaic\_virus* – perfect precision but low recall (54%), indicating it's often confused with other viral diseases.



## 4. Final Remarks on Model Performance

The model demonstrates excellent performance overall:

* **Generalization**: Strong accuracy on test set confirms minimal overfitting.
* **Balanced Performance**: Metrics indicate consistent results across majority and minority classes.
* **Class-Specific Strengths**: Viral and bacterial tomato diseases were classified with high reliability.
* **Limitations**: Some classes need improvement, particularly *Tomato\_Early\_blight* and *Tomato\_mosaic\_virus*.

The use of **EfficientNetB0** combined with **progressive fine-tuning** proved highly effective for this multi-class classification task.

Stage 3: Final Evaluation, Visual Testing & Web Deployment

## Section 1: Final Evaluation on Full Test Dataset

**Purpose**

The aim of this section was to perform a **final and full-scale evaluation** of the trained EfficientNet model using the test dataset. This step ensures the model's performance is stable, reliable, and generalizable to unseen images.

* Used the test\_gen generator to load **2,064 images** spread across **15 different plant disease and health categories**.
* Ensured **no shuffling** to preserve class order, which is critical for accurate evaluation and confusion matrix generation.
* Evaluated the final trained model on this dataset using Keras' evaluate() method.
* This was the **final check** for model performance before deployment.
* Provided accurate test loss and accuracy metrics to measure how well the model generalizes beyond training and validation data.

**Output**

* **Test Loss**: 0.2507
* **Test Accuracy**: 91.62%

**Interpretation**

* A low loss value and high accuracy indicate **excellent performance** and **minimal overfitting**, especially considering the complexity of plant diseases and image variability.

## Section 2: Visual Testing — Sample Predictions

**Purpose**

This section visually demonstrates how well the model performs on actual test images by comparing **predicted vs actual labels** for individual samples.

* Displayed **10 test images** alongside their **true and predicted labels**.
* Each image was taken directly from the test dataset and fed into the model individually.
* Observed whether the predictions matched the ground truth.
* Helps **visually validate** model performance.
* Reveals any patterns in misclassification that may not be obvious through metrics alone.
* Useful for presentation or user demonstration.

**Key Observations**

* Majority of predictions matched the actual labels, supporting the reported accuracy.
* Occasional misclassifications showed similar-looking diseases (e.g., **bacterial vs fungal spots**) being confused — common even for human observers.



## Section 3: Visual Testing — One Image from Each Class

**Purpose**

This test aimed to extract **one representative image from each of the 15 classes** and display the model's prediction to ensure consistent class-level performance.

* Iterated through the test generator until **one image per class** was obtained.
* Ran each image through the model and compared predicted and true labels.
* Ensured that all classes, especially the **less frequent or more complex ones**, were handled accurately.
* Allowed class-wise debugging and error analysis.

**Key Insights**

* Predictions for most classes were accurate, indicating strong **class coverage**.
* Minor misclassifications occurred in visually similar categories, such as:
  + Tomato Leaf Mold vs Septoria Leaf Spot.
  + Potato Early Blight vs Late Blight.



## Final Section: Gradio-Based Web Interface Deployment

**Purpose**

To make the trained model **accessible to users via a simple web interface**, allowing anyone to upload a plant leaf image and receive real-time disease predictions.

* Integrated the Keras model into a **Gradio UI**.
* Created a clean interface with:
  + **Input**: Leaf image (uploaded by the user).
  + **Output**: Predicted class and model confidence score.
* The model resizes and preprocesses the image using **EfficientNet-compatible methods**, then performs prediction.
* Bridged the gap between development and usability.
* Made the model accessible to **farmers, agronomists, or researchers** without coding expertise.
* Demonstrated practical application in **real-world agriculture**.

**🔹 Example Output**

*“Tomato\_\_Tomato\_YellowLeaf\_\_Curl\_Virus (98.22%)”*

**🔹 Strengths**

* High confidence scores, typically 90%+, on clean leaf images.
* Fast and responsive interface with minimal delay.
* Simple and intuitive for non-technical users.

