## **Plant Disease Classification Model: Code Explanation and Output Analysis**

### **BSc Computer Science - Group 13 Final Year Project**

This document provides a detailed explanation of the Python code used to develop a plant disease classification model and an in-depth analysis of its performance outputs. The goal is to present a comprehensive understanding of the project's methodology, results, and implications.

### **1. Introduction to the Project**

This project focuses on developing an efficient and highly accurate Convolutional Neural Network (CNN) for classifying plant diseases from images. Early and accurate detection of plant diseases is crucial for agriculture, helping to prevent crop loss and ensuring food security. This report will cover the technical implementation of our model, the training process, and a detailed breakdown of its performance, validated through various advanced visualizations.

### **2. Code Explanation: Building the Classification System**

Our project's code is structured into several logical blocks, each responsible for a specific phase of the machine learning pipeline, from data preparation to model evaluation and visualization.

#### **2.1. Environment Setup and Data Handling**

The initial part of the code sets up the Python environment, installs necessary libraries, and handles the dataset loading.

* **Package Installation:**  
  !pip install -q torch torchvision matplotlib seaborn scikit-learn umap-learn  
    
  This line installs all the essential Python libraries.
  + torch and torchvision: Core libraries for building and working with neural networks in PyTorch, and for image datasets/transforms.
  + matplotlib and seaborn: Used for creating various types of plots and visualizations.
  + scikit-learn: Provides a wide range of machine learning tools, including metrics for evaluating model performance.
  + umap-learn: For Uniform Manifold Approximation and Projection, a dimensionality reduction technique.
* **Google Drive Integration:**  
  from google.colab import drive  
  drive.mount('/content/drive', force\_remount=True)  
    
  This section mounts Google Drive, allowing the Colab environment to access files stored in your Drive, specifically the zipped dataset.
* **Dataset Extraction:**  
  zip\_path = '/content/drive/My Drive/Plant\_leaf\_diseases\_dataset\_with\_augmentation.zip'  
  extracted\_base = '/content/extracted\_dataset'  
  os.makedirs(extracted\_base, exist\_ok=True)  
  # ... (zip extraction logic) ...  
    
  The code defines paths for the zipped dataset in Google Drive and a local directory for extraction. It then extracts the dataset, making the image files accessible for processing.
* **Dataset Root Detection (find\_dataset\_root):**  
  def find\_dataset\_root(start\_path):  
   # ... (logic to find the actual directory containing class folders) ...  
    
  This crucial function intelligently navigates the extracted directory structure to locate the precise folder where the image classification classes (e.g., 'Apple\_\_\_healthy', 'Tomato\_\_\_Late\_blight') reside. This ensures the data loading process (ImageFolder) works correctly regardless of minor variations in the zip file's internal structure.
* **Dataset Exploration (explore\_dataset):**  
  def explore\_dataset(dataset\_root):  
   # ... (prints class names and image counts) ...  
    
  Once the root is found, this function provides a quick overview of the dataset, listing a sample of classes and the total number of images, confirming successful loading.

#### **2.2. Device Configuration and Image Transformations**

To ensure efficient training, especially on hardware accelerators like GPUs, and to prepare images for the neural network, specific configurations and transformations are applied.

* **Device Selection:**  
  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
  print(f"Using device: {device}")  
    
  This line automatically detects if a CUDA-enabled GPU is available. If so, it uses the GPU for computations, which significantly speeds up training. Otherwise, it defaults to the CPU.
* **Optimal Settings:**  
  if device.type == 'cuda':  
   torch.backends.cudnn.benchmark = True  
    
  When using a GPU, torch.backends.cudnn.benchmark = True enables cuDNN's auto-tuner to find the most efficient algorithms for convolutional operations, further boosting performance.
* **Image Transformations:**  
  transform\_train = transforms.Compose([  
   transforms.Resize(img\_size),  
   transforms.CenterCrop(crop\_size),  
   transforms.RandomHorizontalFlip(0.3),  
   transforms.RandomRotation(15),  
   transforms.ColorJitter(brightness=0.2, contrast=0.2),  
   transforms.ToTensor(),  
   transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  
  ])  
  transform\_val = transforms.Compose([  
   transforms.Resize(img\_size),  
   transforms.CenterCrop(crop\_size),  
   transforms.ToTensor(),  
   transforms.Normalize(...)  
  ])  
    
  These transformations prepare the raw images for input into the neural network:
  + transforms.Resize and transforms.CenterCrop: Standardize image dimensions.
  + transforms.RandomHorizontalFlip, transforms.RandomRotation, transforms.ColorJitter: These are data augmentation techniques applied *only* during training. They introduce minor variations to the training images (e.g., flipping, rotating, changing brightness/contrast) to make the model more robust and prevent it from overfitting to specific image orientations or lighting conditions.
  + transforms.ToTensor(): Converts images to PyTorch tensors and scales pixel values from 0-255 to 0-1.
  + transforms.Normalize(): Standardizes the pixel values across all images using pre-calculated mean and standard deviation for typical image datasets. This helps the neural network learn more effectively.
* **Dataset Loading and Splitting:**  
  train\_dataset = datasets.ImageFolder(dataset\_root, transform=transform\_train)  
  val\_dataset = datasets.ImageFolder(dataset\_root, transform=transform\_val)  
  test\_dataset = datasets.ImageFolder(dataset\_root, transform=transform\_val)  
  # ... (splitting into train, validation, test indices) ...  
    
  The ImageFolder class automatically organizes images into classes based on their folder structure. The dataset is then split into three sets:
  + **Training Set (70%):** Used to train the model.
  + **Validation Set (15%):** Used during training to monitor the model's performance on unseen data and adjust hyperparameters or trigger early stopping.
  + **Test Set (15%):** A completely held-out set used *only* at the very end to evaluate the model's final, unbiased performance.

#### **2.3. Model Architecture (ColabOptimizedCNN)**

The core of the project is the custom Convolutional Neural Network (CNN) designed for efficiency and accuracy.

* **Class Definition:**  
  class ColabOptimizedCNN(nn.Module):  
   def \_\_init\_\_(self, num\_classes, input\_size=112):  
   super(ColabOptimizedCNN, self).\_\_init\_\_()  
   # ... (layers defined here) ...  
   def forward(self, x):  
   # ... (forward pass logic) ...  
   def get\_features(self, x):  
   # ... (feature extraction logic) ...  
    
  This class inherits from nn.Module, the base class for all neural network modules in PyTorch.
* **Feature Extractor (self.features):** This sequential block of layers is responsible for extracting meaningful features from the input images.
  + nn.Conv2d: Convolutional layers apply filters to images to detect patterns.
  + nn.BatchNorm2d: Batch normalization layers normalize the activations of previous layers, which helps in faster and more stable training.
  + nn.ReLU6: A common activation function that introduces non-linearity, allowing the network to learn complex patterns. ReLU6 caps the output at 6, which can be beneficial in certain mobile-optimized architectures.
  + \_make\_depthwise\_block: This is the key to efficiency. It creates a "depthwise separable convolution" block. Instead of a single convolution that processes all input channels and outputs all channels simultaneously, it performs:
    1. **Depthwise Convolution:** A separate filter for each input channel. This is computationally much cheaper.
    2. Pointwise Convolution (1x1 Conv): A 1x1 convolution to combine the outputs of the depthwise convolution across channels. This allows for efficient channel mixing.  
       This design significantly reduces the number of parameters and computations compared to traditional convolutions.
  + nn.AdaptiveAvgPool2d((1, 1)): This layer efficiently reduces the spatial dimensions of the feature maps to 1x1, preparing them for the fully connected (linear) layers.
* **Classifier (self.classifier):** This block takes the extracted features and maps them to the final class predictions.
  + nn.Dropout: Randomly sets a fraction of input units to zero during training. This prevents co-adaptation of neurons and helps in regularization, reducing overfitting.
  + nn.Linear: Fully connected layers, which perform linear transformations. The final linear layer outputs a score for each class.
* **Weight Initialization (\_initialize\_weights):** This method sets initial values for the network's weights and biases. Proper initialization is crucial for stable and effective training.
* **forward method:** Defines the forward pass of the model, specifying how input data x flows through the layers to produce an output.
* **get\_features method:** A helper function specifically designed to extract the learned features *before* they pass through the final classification layers. This is essential for visualizing the model's learned representation using techniques like PCA, t-SNE, and UMAP.

#### **2.4. Training Setup**

With the model and data ready, the next step is to configure the training process.

* **Loss Function (criterion):**  
  criterion = nn.CrossEntropyLoss()  
  ```nn.CrossEntropyLoss` is commonly used for multi-class classification problems. It measures the difference between the model's predicted probabilities and the true labels.
* **Optimizer (optimizer):**  
  optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight\_decay=1e-4)  
  ```AdamW` is an optimization algorithm that adjusts the model's weights during training to minimize the loss. `lr` (learning rate) controls the step size of these adjustments, and `weight\_decay` adds a regularization term to prevent overfitting.
* **Learning Rate Scheduler (scheduler):**  
  scheduler = StepLR(optimizer, step\_size=7, gamma=0.5)  
    
  This scheduler automatically reduces the learning rate by a factor of gamma (0.5) every step\_size (7) epochs. This strategy allows the model to learn quickly at the beginning and then fine-tune its parameters more precisely as training progresses.
* **Data Loaders (DataLoader):**  
  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, sampler=SubsetRandomSampler(train\_indices), num\_workers=num\_workers, pin\_memory=True)  
  val\_loader = DataLoader(val\_dataset, ...)  
  test\_loader = DataLoader(test\_dataset, ...)  
  ```DataLoader` batches the data, shuffles it, and provides it to the model during training and evaluation.  
  \* `batch\_size`: The number of samples processed at once.  
  \* `SubsetRandomSampler`: Ensures that the data is randomly sampled from the specified indices (train, val, test).  
  \* `num\_workers`: Number of subprocesses to use for data loading, speeding up the process.  
  \* `pin\_memory=True`: Copies Tensors into CUDA pinned memory before returning them, which can speed up data transfer to GPU.

#### **2.5. Training Loop (train\_model)**

This function orchestrates the entire training process over multiple epochs.

* **Epoch Iteration:**  
  for epoch in range(epochs):  
   model.train() # Set model to training mode  
   # ... (training steps) ...  
   model.eval() # Set model to evaluation mode  
   # ... (validation steps) ...  
   scheduler.step() # Update learning rate  
    
  Each "epoch" represents one complete pass through the entire training dataset.
  + model.train(): Sets the model to training mode, enabling features like dropout and batch normalization to behave as expected during training.
  + **Forward Pass:** outputs = model(inputs) feeds the input images through the model to get predictions.
  + **Loss and Backpropagation:** loss = criterion(outputs, targets) calculates the loss. loss.backward() computes gradients (how much each parameter contributed to the error). optimizer.step() updates the model's weights using these gradients.
  + model.eval(): Sets the model to evaluation mode, disabling dropout and ensuring batch normalization uses accumulated statistics, which is appropriate for consistent predictions.
  + **Validation:** During each epoch, the model is evaluated on the separate validation set to monitor its performance on unseen data.
* **Early Stopping:**  
  if val\_acc > best\_val\_acc:  
   best\_val\_acc = val\_acc  
   counter = 0  
   torch.save(model.state\_dict(), '/content/best\_model.pt')  
  else:  
   counter += 1  
   if counter >= patience:  
   print(f"Early stopping at epoch {epoch+1}")  
   break  
    
  This mechanism prevents overfitting. If the validation accuracy doesn't improve for a certain number of epochs (patience), training is stopped early, and the model with the best validation accuracy is saved.

#### **2.6. Evaluation Function (evaluate\_model)**

After training, this function rigorously assesses the model's performance on the test set.

* **Prediction Collection:**  
  model.eval()  
  with torch.no\_grad(): # No gradient calculations needed for evaluation  
   for inputs, targets in test\_loader:  
   # ... (collects predictions, true labels, and probabilities) ...  
    
  The function iterates through the test dataset, collects the model's predictions (preds), the true labels (targets), and the confidence scores for each class (probs).
* **Metric Calculation:**  
  accuracy = accuracy\_score(all\_targets, all\_preds)  
  precision\_macro = precision\_score(all\_targets, all\_preds, average='macro', zero\_division=0)  
  recall\_macro = recall\_score(all\_targets, all\_preds, average='macro', zero\_division=0)  
  f1\_macro = f1\_score(all\_targets, all\_preds, average='macro', zero\_division=0)  
  # ... (micro averages also calculated) ...  
    
  It calculates various standard classification metrics from sklearn.metrics:
  + **Accuracy:** The proportion of correctly classified samples.
  + **Precision:** Out of all positive predictions for a class, how many were actually correct.
  + **Recall (Sensitivity):** Out of all actual positive samples for a class, how many were correctly identified.
  + **F1-Score:** The harmonic mean of precision and recall, providing a balanced measure.
  + **Macro Average:** Calculates the metric independently for each class and then takes the average. This treats all classes equally, regardless of their size.
  + **Micro Average:** Calculates the metric globally by counting the total true positives, false negatives, and false positives. This is influenced by class imbalance.

#### **2.7. Visualization Functions**

A significant part of the project is dedicated to visually interpreting the model's behavior and performance.

* **plot\_precision\_recall\_curves:**  
  def plot\_precision\_recall\_curves(all\_targets, all\_probs, classes):  
   # ... (plots micro, macro, and per-class PR curves) ...  
    
  These curves illustrate the trade-off between precision and recall at various classification thresholds. A curve closer to the top-right corner indicates better performance. Micro-average combines the true positives, false positives, and false negatives from all classes to compute the overall precision-recall. Macro-average calculates precision-recall for each class independently and then averages them.
* **plot\_roc\_curves:**  
  def plot\_roc\_curves(all\_targets, all\_probs, classes):  
   # ... (plots micro, macro, and per-class ROC curves) ...  
    
  The Receiver Operating Characteristic (ROC) curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The Area Under the Curve (AUC) summarizes the curve's performance. An AUC of 1.0 represents a perfect classifier, while 0.5 represents random chance. These curves provide insights into the model's ability to distinguish between classes.
* **plot\_feature\_clusters:**  
  def plot\_feature\_clusters(model, test\_loader, classes, device, max\_samples=1000):  
   # ... (extracts features, applies PCA, t-SNE, UMAP, and plots) ...  
    
  This function visualizes the high-dimensional feature space learned by the model using dimensionality reduction techniques:
  + **PCA (Principal Component Analysis):** A linear method that transforms data to a new set of orthogonal variables, capturing the maximum variance. It's good for understanding global data structure.
  + **t-SNE (t-Distributed Stochastic Neighbor Embedding):** A non-linear method that is particularly good at visualizing high-dimensional data by giving each datapoint a location in a two or three-dimensional map. It excels at preserving local neighborhoods (clusters).
  + UMAP (Uniform Manifold Approximation and Projection): Another non-linear technique for dimensionality reduction, often faster than t-SNE and better at preserving global structure while still revealing local clusters.  
    These plots help us see if the model effectively clusters images belonging to the same disease class, visually confirming the quality of its learned representations.
* **analyze\_model\_complexity:**  
  def analyze\_model\_complexity(model):  
   # ... (counts parameters, estimates memory, plots distributions) ...  
    
  This function provides a detailed breakdown of the model's architecture, including:
  + **Total Parameters:** The total number of adjustable weights and biases in the model.
  + **Trainable Parameters:** Parameters that are updated during training.
  + **Model Size:** An estimation of the model's memory footprint.
  + **Parameter Distribution:** Visualizes how parameters are distributed across different layer types (e.g., convolutional, linear).
  + Memory Usage Breakdown: Estimates memory used by parameters, gradients, and activations.  
    This analysis is vital for understanding the model's efficiency and suitability for resource-constrained environments.

### **3. Analysis of Model Output (Based on Provided Images)**

The generated output images provide comprehensive insights into our model's training process and final performance.

#### **3.1. Overall Performance Metrics**

The output from the evaluate\_model function and the "Test Set Performance Metrics" chart (from downloadchart2.png) summarize the model's effectiveness.

* **Key Observations (from Screenshot 2025-06-02 224414.png and Screenshot 2025-06-02 224913.png):**
  + **Test Loss: 0.0364** (Very low, indicating minimal error on unseen data).
  + **Accuracy: 0.9873 (98.73%)** This is an exceptionally high accuracy, meaning the model correctly classified nearly 99% of the plant disease images in the independent test set.
  + **Macro Precision: 0.9849**
  + **Macro Recall: 0.9839**
  + **Macro F1 Score: 0.9842**
  + **Micro Precision: 0.9873**
  + **Micro Recall: 0.9873**
  + **Micro F1 Score: 0.9873**
* Analysis:  
  The high accuracy is the primary indicator of success. The macro-averaged metrics (precision, recall, F1-score) are particularly important in multi-class classification, especially when dealing with potentially imbalanced datasets. High macro scores demonstrate that the model performs well across all individual classes, not just the majority classes. The micro-averaged scores are identical to the overall accuracy in this case, which is expected for classification problems where each sample is assigned to exactly one class. These results confirm the model's strong classification capabilities and its ability to generalize to new, unseen plant images.

#### **3.2. Model Complexity Analysis**

The analyze\_model\_complexity function provides a detailed look at the model's size and parameter distribution.

* **Key Observations (from Screenshot 2025-06-02 224556.png and downloadchart1.png):**
  + **Total Parameters: 410,471**
  + **Trainable Parameters: 410,471** (All parameters are trainable, meaning the model was trained from scratch or fine-tuned entirely).
  + **Model Size: 1.57 MB**
  + **Parameter Distribution by Layer Type (Pie Chart):** The majority of parameters (64.5%) are in Conv2d layers, followed by Linear layers (34.4%), and a small portion in BatchNorm2d (1.1%). This is typical for CNNs where convolutional layers perform feature extraction and linear layers perform classification.
  + **Memory Usage Breakdown (Bar Chart):** Shows the estimated memory used by parameters, gradients, and activations. Parameters and gradients each consume about 1.5 MB, while estimated activations (based on batch size and image dimensions) also take up a significant chunk of memory. This analysis is crucial for understanding the model's resource requirements.
* Analysis:  
  A total parameter count of 410K and a model size of 1.57 MB are remarkably small for a deep learning model achieving such high accuracy. This is a direct benefit of using depthwise separable convolutions in the ColabOptimizedCNN architecture, which are known for their efficiency. This lightweight nature makes the model highly suitable for deployment on resource-constrained devices, such as mobile phones or embedded systems, which could be used for on-site plant disease detection.

#### **3.3. Training Progress and Convergence**

The "Training and Validation Loss," "Validation Accuracy," "Learning Rate Schedule," and "Overfitting Indicator" charts (from downloadchart2.png) illustrate the model's learning process over 20 epochs.

* **Training and Validation Loss:** Both training and validation losses decrease steadily and converge to very low values. Importantly, they track each other closely, indicating that the model is learning effectively without a significant gap between training and validation performance, which would suggest overfitting.
* **Validation Accuracy:** The validation accuracy increases rapidly in the initial epochs and then stabilizes at a high level, reaching a best of 0.9882. The "Convergence Analysis" plot (also in downloadchart2.png) shows a smooth curve, indicating stable learning.
* **Learning Rate Schedule:** The chart clearly shows the step-wise reduction of the learning rate. This strategy (reducing LR every 7 epochs) helps the model escape local minima and fine-tune its weights more precisely in later stages, contributing to better final performance.
* **Overfitting Indicator (Validation Loss - Train Loss):** This plot shows the difference between validation and training loss. A consistently positive and increasing difference could indicate overfitting. In our case, the difference remains relatively low and stable, suggesting good generalization and minimal overfitting.
* Analysis:  
  The training curves demonstrate that the model learned efficiently and converged effectively. The consistent performance on both training and validation sets, coupled with the application of early stopping (triggered at Epoch 20/20 in Screenshot 2025-06-02 224414.png), highlights the robustness of the training process and the model's ability to generalize well to unseen data.

#### **3.4. Precision-Recall Curves**

The "Micro-averaged Precision-Recall Curve," "Macro-averaged Precision-Recall Curve," and "Per-class Precision-Recall (Top 5)" charts (from downloadchart3.png) provide a detailed view of the model's trade-off between precision and recall.

* **Key Observations:**
  + Both Micro-averaged and Macro-averaged curves are very close to the top-right corner of the plot, indicating high precision for high recall values across all classes.
  + The "Per-class Precision-Recall (Top 5)" plot shows that even for individual classes (e.g., 'Apple\_Ap', 'Apple\_Bl'), the curves are nearly perfect, residing very close to the (1,1) point.
* Analysis:  
  These curves are excellent indicators of the model's performance in a multi-class setting. High precision and recall across the board (as shown by the near-perfect curves) mean that the model not only correctly identifies positive cases (high recall) but also has very few false positives (high precision). This is crucial for a diagnostic tool like plant disease classification, where both missing a disease and misdiagnosing a healthy plant can have significant consequences.

#### **3.5. ROC Curves**

The "Micro-averaged ROC Curve," "Macro-averaged ROC Curve," and "Per-class ROC Curves (Top 5)" charts (from downloadchart4.png) assess the model's ability to distinguish between classes.

* **Key Observations (from downloadchart4.png and Screenshot 2025-06-02 224822.png):**
  + **Micro-averaged ROC AUC: 1.0000**
  + **Macro-averaged ROC AUC: 0.9999**
  + All ROC curves (micro, macro, and per-class) hug the top-left corner of the plot, indicating a near-perfect classifier. The dashed line represents random chance, and our model's curves are far superior.
* Analysis:  
  An AUC score very close to 1.0 (with the micro-average being exactly 1.0000 and macro-average 0.9999) is an outstanding result. It signifies that the model has an almost perfect ability to discriminate between healthy and diseased plants, as well as between different types of diseases. This means the model produces high true positive rates while keeping false positive rates extremely low, which is ideal for real-world application.

#### **3.6. Feature Space Clustering**

The PCA, t-SNE, and UMAP scatter plots (from downloadchart5.jpg) visualize the high-dimensional features extracted by the model, projected into a 2D space. Each point represents an image, and its color indicates its true disease class.

* **Key Observations:**
  + **PCA (Explained Variance: 0.281):** While PCA shows some separation, the clusters are relatively spread out, and there's significant overlap between different colored groups. The explained variance is low, meaning 2D PCA only captures a small portion of the total information.
  + **t-SNE:** This plot shows much tighter and more distinct clusters. Images of the same color (same disease class) are grouped together, and different colors are largely separated, with minimal overlap.
  + **UMAP:** Similar to t-SNE, UMAP also produces well-defined and separated clusters, suggesting that the model has learned robust and discriminative features.
* Analysis:  
  The feature clustering visualizations, particularly t-SNE and UMAP, provide compelling visual evidence of the model's learning success. The clear separation of different disease classes in the learned feature space indicates that the CNN effectively extracts distinct and meaningful patterns for each plant disease. This ability to group similar diseases and separate dissimilar ones at the feature level is fundamental to its high classification accuracy. PCA's lower separation is expected, as it is a linear technique and might not capture the complex non-linear relationships present in the data as effectively as t-SNE or UMAP.

#### **3.7. Sample Predictions**

The sample predictions provide concrete examples of the model's output.

* **Key Observations (from Screenshot 2025-06-02 224938.png):**
  + For "Sample 1," the true label was 'Orange\_\_Haunglongbing\_(Citrus\_greening)', and the model correctly predicted 'Orange\_\_Haunglongbing\_(Citrus\_greening)' with a confidence of 1.000 (100%). It also correctly identified the top 3 predictions as mostly variations of the correct class, showing high confidence.
* Analysis:  
  These sample predictions confirm the model's ability to make accurate and highly confident classifications. A confidence score of 1.000 (100%) for the correct class is a strong indicator of the model's certainty, which is vital for practical applications where precise diagnosis is critical.

### **4. Conclusion**

This project successfully developed and rigorously evaluated a custom-optimized CNN for plant disease classification. The results demonstrate a highly accurate, efficient, and robust model capable of distinguishing between various plant diseases with exceptional performance.

**Key Achievements:**

* **Exceptional Accuracy:** Achieved a remarkable 98.73% accuracy on unseen test data, indicating its reliability for real-world application.
* **High Efficiency:** The model is remarkably lightweight at 1.57 MB, making it highly suitable for deployment on edge devices or mobile platforms.
* **Excellent Generalization:** Training and validation curves, coupled with robust test metrics, confirm that the model learns effectively without overfitting, ensuring strong performance on new data.
* **Discriminative Feature Learning:** Feature space visualizations using t-SNE and UMAP clearly show that the model has learned distinct and separable representations for different plant disease classes.

This project contributes a valuable tool for automated plant disease diagnosis, potentially aiding farmers and agricultural experts in maintaining crop health more effectively. The combination of high accuracy and low computational footprint makes this model a strong candidate for practical deployment.