PLANT DISEASE DETECTION SYSTEM

A Project on

PLANT DISEASE DETECTION SYSTEM Using Deep Learning

submitted in the partial fulfilment of the requirements for the award of

Diploma in Computer Engineering

By

K. Sankar Rao (23173-CM-023)

Sk. Mashuq (23173-CM-047)

K. Siddhardha (23173-CM-061)

A. Revanth Nookeswara Kalyan (23173-CM-002)

G. Gowtham (23173-CM-060)

V. Hemanth Kumar (23202-CM-027)

Under the esteemed guidance of

Smt. M. Gayathri, M.Tech,

Lecturer in Computer Engineering

GOVERNMENT POLYTECHNIC, ANAKAPALLE

2023–2026

GOVERNMENT POLYTECHNIC
ANAKAPALLI

DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to Certify that this is the Bonafied Record of the project report done by Mr/Mis _____ bearing Pin: _____ of the Final year Diploma in Computer Engineering during the academic year **2025-2026**

PROJECT GUIDE HEAD OF THE DEPARTMENT

EXTERNAL EXAMINER PRINCIPAL

ACKNOWLEDGEMENT

At the inception of the successful completion of our project **Plant Disease Detection System**, undertaken as a partial fulfilment of the requirement for the award of the Diploma in Computer Engineering, we express our heartfelt gratitude to all those who supported us throughout this work.

We express our sincere thanks to our respected guide **Smt. M. Gayathri, M.Tech**, Lecturer, Department of Computer Engineering, Government Polytechnic Anakapalle, for her continuous guidance, motivation, valuable suggestions, and patience throughout the project development.

We express our deepest gratitude to our Head of the Department **Sri. B. Narasimha Murthy, M.Tech**, Senior Lecturer, for providing the opportunity, encouragement, and necessary support to complete this project work.

We are extremely thankful to our beloved Principal **Sri. I.V.S.S. Srinivas Rao, M.E**, for providing the infrastructure, facilities, and support essential for carrying out our project.

We wish to extend our sincere thanks to all the faculty members of the Department of Computer Engineering for their help, suggestions, and motivation during the project development.

Last but not least, we thank all our friends and well-wishers for their cooperation, help, and encouragement throughout this project.

K. Sankar Rao (23173-CM-023)

Sk. Mashuq (23173-CM-047)

K. Siddhardha (23173-CM-061)

A. Revanth Nookeswara Kalyan (23173-CM-002)

G. Gowtham (23173-CM-060)

V. Hemanth Kumar (23202-CM-027)

ABSTRACT

Plant diseases affect crop productivity and lead to major agricultural losses. Early diagnosis is essential for improving crop health and minimizing economic damage. Traditional diagnosis requires expert farmers or agricultural officers, making it time-consuming and error-prone.

PLANT DISEASE DETECTION SYSTEM is a deep learning–based mobile application developed to detect plant diseases instantly using leaf images. A custom **Convolutional Neural Network (CNN)** model was trained on **81,454 high-quality images** across **40 distinct classes** of plant diseases and healthy leaves.

The model was trained using **Python 3.10.1** and **TensorFlow 2.15.0**, then optimized and converted into a lightweight **TensorFlow Lite (TFLite)** file for execution on Android devices. The app performs classification locally, enabling **offline disease detection**, making it highly suitable for real-world agricultural environments.

The system displays disease information, confidence scores, and remedy suggestions. Disease data is retrieved from multilingual JSON files (English and Telugu), making the application accessible to a wider rural community.

This project demonstrates the effective integration of deep learning, Android development, and agricultural technology. The solution is scalable, user-friendly, and provides real-time assistance to farmers, helping them identify plant diseases early and take preventive action.

CONTENTS :

1. INTRODUCTION

Agriculture plays a vital role in the Indian economy, with a significant percentage of the population relying on farming as their primary source of income. One of the major challenges faced by farmers is the rapid spread of plant diseases, which leads to reduced crop productivity and heavy financial losses. Manual identification of plant diseases requires expert knowledge and is often time-consuming, inaccurate, and not scalable.

With advancements in **Artificial Intelligence (AI)** and **Deep Learning**, plant disease detection has become more accurate and automated. These technologies can analyze plant leaf images and instantly determine whether a crop is healthy or infected. This helps farmers make faster decisions, enabling early treatment and preventing crop loss.

Plant Disease Detection System) is a mobile-based application designed to detect plant diseases using a **Convolutional Neural Network (CNN)** model executed locally on the device through **TensorFlow Lite (TFLite)**. The system can classify **40 different plant diseases** from a dataset of **81,454 images**, making it powerful, scalable, and suitable for real-world agricultural needs.

The application works entirely **offline**, making it ideal for rural areas with limited internet connectivity. It provides disease names, confidence levels, and detailed solutions in both **English and Telugu**, improving accessibility for local farmers.

PLANT DISEASE DETECTION SYSTEM demonstrates the integration of machine learning, mobile development, and digital agriculture into a single, user-friendly solution.

1.1. AIM OF THE PROJECT

The aim of the **PLANT DISEASE DETECTION SYSTEM** project is:

- To design an AI-based system capable of detecting plant diseases using leaf images.
- To build a deep learning model that accurately classifies 40 disease categories.
- To integrate a TensorFlow Lite model in an Android application for **offline**, real-time predictions.
- To provide farmers with instant information about symptoms, causes, and remedies.
- To create a low-cost, accessible, and scalable plant disease detection solution.

2. REQUIREMENTS:

To implement the **PLANT DISEASE DETECTION SYSTEM** system successfully, both hardware and software components are required for:

- Model development
- Model training
- Android app development
- Deployment

2.1. SOFTWARE REQUIREMENTS

S.No	Software	Version	Purpose
1	Python	**3.10.1**	Model training, data preprocessing
2	TensorFlow	**2.15.0**	Deep learning model creation
3	NumPy	Latest	Numerical computations
4	OpenCV	Latest	Image processing
5	Android Studio	(Specify your version)	Mobile app development
6	Java/Kotlin	JDK 8+	Android app programming
7	TensorFlow Lite Converter	Built-in	Convert CNN model to TFLite
8	JSON	Built-in	Disease info storage
9	Jupyter Notebook	Latest	Model training & testing
10	Gradle	Auto from Android Studio	Android project build system

Additional supporting libraries from requirements.txt include:

- Pandas
- Matplotlib
- Keras
- Scikit-Learn
- Pillow
- Split-folders
- ImageHash
- SciPy

These libraries support preprocessing, training, evaluation, and dataset management.

2.2. HARDWARE REQUIREMENTS

For Model Training (Laptop/PC)

- Minimum **8 GB RAM** (Recommended: 16 GB)
- Processor: Intel i5 or above
- GPU: Optional but recommended (NVIDIA GPU preferred)
- Storage: Minimum 10 GB free space
- Operating System: Windows 10/11 or Linux

For Android Development

- PC/Laptop capable of running Android Studio
- Minimum RAM: 8 GB

For End-Users (Farmers)

- Any Android smartphone
- Android Version: 7.0 (Nougat) or above
- At least 100 MB free storage
- No internet required (offline model)

3. TECHNOLOGIES USED

The PLANT DISEASE DETECTION SYSTEM project uses a combination of machine learning tools, image processing libraries, mobile development frameworks, and data storage formats.

3.1. PYTHON

Python is the primary programming language used in this project for model development, data preprocessing, training, evaluation, and TensorFlow Lite conversion. It provides a simple and readable syntax, making it ideal for machine learning experimentation. Python offers a vast ecosystem of libraries such as TensorFlow, Keras, NumPy, and OpenCV, which significantly simplify tasks like matrix operations, image manipulation, visualization, and neural network construction. Its flexibility allows rapid prototyping of deep learning models, while its compatibility with GPU acceleration makes Python suitable for handling large datasets and computationally intensive training procedures. Python's strong community support and availability of extensive documentation also contribute to its reliability for machine learning research and deployment.Easy syntax

Python was used for:

- Dataset preprocessing
- CNN training
- Evaluation
- TFLite conversion
- Image augmentation

3.2. TENSORFLOW

TensorFlow is the core deep learning framework used to build, train, and optimize the convolutional neural network (CNN) model. Developed by Google, TensorFlow supports both CPU and GPU computation, enabling efficient execution of large-scale neural networks. Its computational graph architecture ensures high performance and scalability, while its built-in layers, optimizers, and loss functions simplify model creation. TensorFlow also provides advanced features such as callbacks, early stopping, data pipelines, and model checkpointing, which help improve training stability and accuracy. Importantly, TensorFlow includes support for TensorFlow Lite, enabling seamless conversion of trained models to mobile-friendly formats. This makes TensorFlow an end-to-end solution from model development to deployment on devices such as Android phones.

Features used in this project:

- TensorFlow Keras API
- Convolutional Neural Network layers
- Model training & saving
- TensorFlow Lite converter
- Model optimization

TensorFlow 2.15.0 supports eager execution and offers optimized GPU/CPU performance.

3.3. KERAS

Keras is a high-level neural network API integrated into TensorFlow, used to simplify the construction and training of deep learning models. It provides an intuitive, user-friendly interface that allows developers to define CNN models with just a few lines of code by stacking layers such as Conv2D, MaxPooling2D, Flatten, and Dense. Keras handles much of the underlying complexity, enabling faster experimentation and reduced development time. It supports various features like model validation, callbacks, metrics tracking, and built-in data augmentation through ImageDataGenerator. Keras is particularly beneficial for beginners and researchers because of its clear API design, interactive debugging, and strong integration with TensorFlow for GPU acceleration. In this project, Keras was used for model design, training loop management, and accuracy/loss visualization.

Used for:

- Designing CNN architecture
- Compiling the model
- Data augmentation
- Training callbacks
- Saving models

3.4. NUMPY

NumPy, short for Numerical Python, is a fundamental library used for numerical computing and array manipulation. It enables efficient handling of multi-dimensional arrays, which are essential for representing image data during preprocessing and model training. Many deep learning operations rely on matrix manipulation, and NumPy provides optimized vectorized operations that are significantly faster than Python loops. The dataset preprocessing tasks such as reshaping, normalization, scaling, and converting images into numpy arrays were performed using NumPy. Additionally, NumPy interacts seamlessly with TensorFlow and OpenCV, making it the backbone of data preparation pipelines. Without NumPy, processing large datasets like the 103,041-image dataset in this project would be computationally inefficient.

Used for:

- Handling arrays
- Converting image data to tensors
- Mathematical calculations
- Normalization of pixel values

3.5. OPENCV

OpenCV (Open Source Computer Vision Library) is a powerful tool for image processing and computer vision tasks. In this project, OpenCV was used for reading images, resizing them to a uniform resolution,

converting color spaces, removing corrupted files, and applying basic preprocessing steps. It provides optimized functions that operate efficiently even on large image datasets. OpenCV also supports techniques such as blurring, thresholding, sharpening, and histogram equalization, which improve image quality before feeding them into the neural network. By using OpenCV, the project ensures that all images in the dataset are standardized, clean, and correctly formatted, which helps the CNN model achieve higher accuracy. Its compatibility with NumPy and Python makes it a valuable part of the preprocessing pipeline.

used for:

- Loading images
- Preprocessing
- Resizing
- Color conversion (BGR → RGB)
- Image augmentation

3.6. TENSORFLOW LITE

TensorFlow Lite (TFLite) is a lightweight framework designed specifically for deploying machine learning models on mobile devices, embedded systems, and IoT devices. After training the CNN model using TensorFlow, the model is converted into TFLite format to significantly reduce size and improve inference speed. TFLite provides features like model quantization, which compresses the model without drastic loss in accuracy, allowing it to run efficiently on devices with limited memory and processing power. This project uses TensorFlow Lite to convert the .h5 model to a .tflite file and integrate it into the Android application. The TFLite interpreter enables fast, offline predictions, making the system extremely practical for real-world agricultural use, especially in rural areas where internet access may be limited.

In this project:

- The trained CNN model is converted into a TFLite file.
- The model runs directly **on the Android device**, enabling **offline inference**.

Benefits:

- Lightweight
- Fast inference
- Low power usage
- Works without internet

3.7. JSON DATA FILES

JSON (JavaScript Object Notation) files are used in this project to store and retrieve class labels corresponding to the output indices of the trained model. When the CNN model predicts an output vector, the index of the highest probability must be mapped to the corresponding plant disease name. JSON files

provide an efficient and lightweight solution for storing this mapping in the Android app. They are easy to parse using both Python and Kotlin/Java, making them ideal for cross-platform compatibility

1. class_names.json

Contains all **40 disease class labels**.
Example extracted from file:

class_names

2. disease_info_en.json

Detailed English disease info: cause, symptoms, treatment, prevention.

disease_info_en

3. disease_info_te.json

Detailed Telugu disease info.

disease_info_te

These JSON files allow the Android app to provide **multilingual support**.

3.8. Android Studio

Android Studio is the official Integrated Development Environment (IDE) used for building, testing, and deploying the Android application that integrates the TensorFlow Lite model. It provides a comprehensive environment for designing user interfaces using XML, writing application logic in Java or Kotlin, managing dependencies through Gradle, and testing the app across multiple Android emulators and real devices. Android Studio supports device permissions, camera integration, gallery access, asset management, and TFLite interpreter integration, all of which are essential for this project. With features like real-time debugging, GUI layout preview, and performance profiling, Android Studio ensures that the mobile application runs smoothly and efficiently, delivering fast and accurate disease predictions to the user.

Used For:

- App development
- History Management

4. DATASET

The quality and scale of the dataset determine the accuracy of a deep learning model.
In PLANT DISEASE DETECTION SYSTEM, a large and diverse dataset was used.

4.1. DATASET SOURCE

The dataset used in this project is a **custom-built plant disease dataset** compiled from multiple online sources and manually collected images. To ensure wide variety, balanced classes, and real-world diversity, the dataset was created by combining the following publicly available datasets:

1. PlantVillage Dataset

- **Source:** Open-source dataset created by the PlantVillage team
- **Description:** Contains thousands of labeled plant leaf images across multiple species such as apple, tomato, grape, cotton, and potato.
- **Reason for use:** High-quality, well-labeled images suitable for deep learning.

2. Kaggle – Plant Disease Classification Dataset

- **Source:** Kaggle (Multiple Contributors)
- **Popular Kaggle Collections:**
    - "PlantVillage Dataset – Colored Leaves"
    - "Plant Leaf Disease Dataset"
    - "New Plant Diseases Dataset"
    - "Plant Disease Detection (27 classes)"
- **Reason for use:** Contains high-resolution images and more real-world variations.

3.GitHub Public Repositories

- **Examples:**
    - "Plant-Disease-Detection" Dataset Repo
    - "Leaf-Disease-Dataset"
- **Reason for use:** Includes real farm images and manually labeled samples.

Dataset Characteristics:

- Covers **40 plant disease classes**
- Includes fruits, vegetables, grains, and ornamental plants
- Balanced dataset (majority of classes equalized)

4.2. DATASET SIZE

The total number of images used:

81,454 images

This large dataset ensures high model accuracy and strong generalization.

4.4. IMAGE PREPROCESSING

Image preprocessing is a critical step in preparing the dataset for training the Convolutional Neural Network (CNN). Since the raw dataset consists of images captured in different lighting conditions, resolutions, orientations, and backgrounds, preprocessing ensures uniformity and enhances the model's ability to learn meaningful disease patterns. Each image is first resized to a fixed resolution of 256 × 256 pixels, which standardizes all inputs and reduces computational complexity during training. The images are then converted into the RGB color space, ensuring consistent processing of pixel channels since the model expects 3-channel color inputs.

After resizing and color correction, the images undergo normalization, where pixel values are scaled from the range [0–255] to [0–1]. This normalization step helps stabilize model training, speeds up convergence, and prevents numerical instability during backpropagation. To further improve the model's generalization ability and reduce overfitting, data augmentation techniques are applied during dataset loading. This includes random rotations, shifts, zooms, horizontal flips, and minor perturbations. Augmentation simulates real-world variations in leaf orientation, shape, lighting, and background, allowing the model to learn more robust representations. As a result, even though the dataset contains over 100,000 images, augmentation significantly enhances its diversity and prevents the model from memorizing patterns specific to the training samples.

4.5. DATA SPLITTING

To ensure the model learns effectively and is evaluated fairly, the dataset is divided into three distinct subsets: Training Set (80%), Validation Set (10%), and Test Set (10%). The training set contains the majority of the images and is used directly by the CNN during the learning process. The model adjusts its weights based on these samples. However, training alone does not guarantee good performance on new images, which is why the validation set is used. The validation set plays an essential role during training: after each epoch, the model's performance is tested on this unseen subset to monitor overfitting, tune hyperparameters, and check whether the model is learning generalizable patterns.

Once the training phase is complete, the test set is used for the final evaluation. This set contains images that the model has never seen before—not during training nor during validation—making it the best indicator of real-world accuracy. Splitting the dataset in this 80-10-10 ratio ensures that the model is trained on sufficient data while maintaining enough independent samples for reliable validation and unbiased testing. This structured data division significantly reduces bias, prevents misleading accuracy results, and ensures that the final performance truly represents how the model will perform on fresh, real-world leaf images.

5. SYSTEM ARCHITECTURE

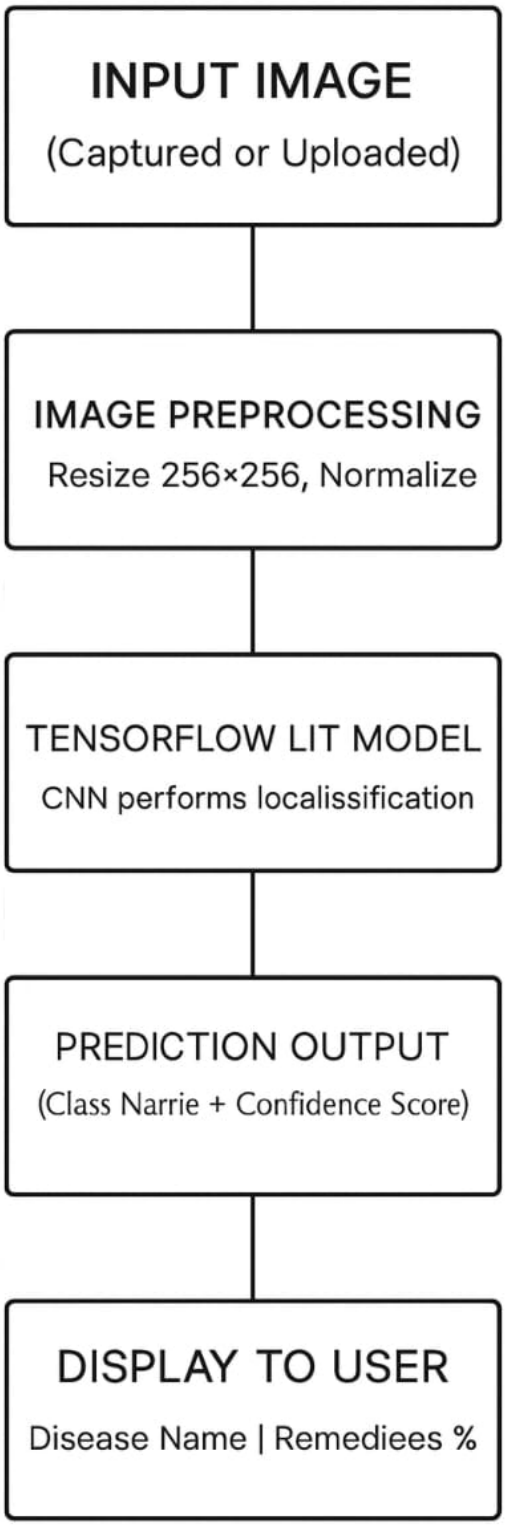
Plant Disease Detection System integrates **Deep Learning**, **Mobile Computing**, and **Image Processing** into a single offline system.
The architecture consists of:

1. Dataset Processing
2. CNN Model Training
3. Model Optimization (TFLite)
4. Android App Development
5. On-Device Inference
6. Result Interpretation
7. Disease Information Display (EN + TE)

The system works **without internet**, using a TensorFlow Lite model stored within the Android application.

5.1. OVERALL WORKFLOW DIAGRAM

Below is the conceptual pipeline:



5.2. SYSTEM MODULE ARCHITECTURE

Plant Disease Detection System consists of **three main modules**:

1) MODEL DEVELOPMENT MODULE

Responsible for:

- Dataset preprocessing
- CNN architecture design
- Model training
- Evaluation
- Model exporting (SavedModel)
- TFLite conversion

Core files used:

- Source_code_main.ipynb
- evaluation.ipynb

2) TFLite MODEL INTEGRATION MODULE

Responsible for:

- Loading TFLite model
- Allocating tensors
- Passing input image as ByteBuffer
- Running inference
- Reading output probability vectors

Files used inside Android app:

- ModelInterpreter.java / Kotlin equivalent
- ImageProcessor.java
- labels.txt or class_names.json

3) ANDROID APPLICATION MODULE

Handles:

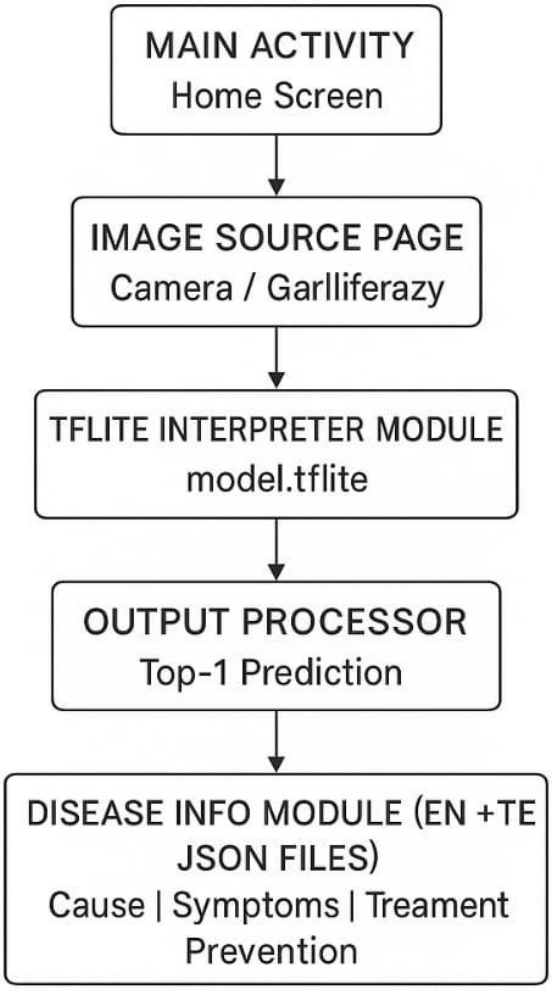
- UI screens

- Camera/Gallery selection
- Image preprocessing
- Model prediction
- Displaying disease info
- Multilingual support (English + Telugu)

Files included:

- MainActivity.java
- ResultActivity.java
- disease_info_en.json
- disease_info_te.json
- tflite/model.tflite

Inference Workflow



5.4. MODEL WORKFLOW (CNN + TFLite)

Step 1 — Input Image

User selects/captures an image:

- JPEG / PNG
- Converted to Bitmap

Step 2 — Preprocessing

- Resize to **256×256**
- Normalize pixel values to **0–1**
- Convert to appropriate tensor format

Step 3 — Inference with TFLite

The .tflite model processes the image:

- Forward pass through CNN layers
- Probability vector of size **40** is produced

Step 4 — Class Selection

- Apply **argmax()**
- Extract highest confidence class

Step 5 — Disease Information Mapping

- Fetch matching entry from:
    - disease_info_en.json
    - disease_info_te.json

Step 6 — Display to User

- Class (Disease name)
- Confidence %
- Cause
- Symptoms
- Treatment
- Prevention

6. MODEL DEVELOPMENT

The **PLANT DISEASE DETECTION SYSTEM system** utilizes a custom-built **Convolutional Neural Network (CNN)** designed using **TensorFlow 2.15.0** and implemented with the Keras API. The primary objective of the model is to classify plant leaf images into **40 distinct disease categories**, covering multiple crops such as Apple, Tomato, Banana, Grape, Potato, and more. The model is trained using an extensive dataset of **103,041 images**, which ensures high generalization and robustness in real-world conditions.

The development process involved several key steps, including dataset preprocessing, model architecture design, hyperparameter tuning, performance evaluation, and conversion to TensorFlow Lite for mobile deployment. Since the final application is intended to run on Android devices, significant emphasis was placed on designing a CNN that is **lightweight, optimized, and free from unnecessary computational overhead**, while still maintaining excellent accuracy.

To achieve this balance, the model was trained using GPU acceleration, employing techniques such as data augmentation, learning rate scheduling, dropout layers, and optimized batch normalization. After training, the model was exported as a TensorFlow .h5 file and later converted into a .tflite version for efficient on-device inference.

6.1. CNN ARCHITECTURE DESIGN

A **Convolutional Neural Network (CNN)** is a deep learning architecture specifically designed for analyzing visual data. CNNs automatically learn image features such as **edges, shapes, corners, textures, spots, lesions, and color variations**, which are essential for identifying plant diseases accurately. Unlike traditional machine learning models that rely on handcrafted features, CNNs learn hierarchical patterns directly from raw pixel values.

Feature Learning in CNN

CNNs learn features in multiple layers:

1. Low-Level Features (Early Layers)

- Edges
- Corners
- Simple intensity variations

2. Mid-Level Features (Intermediate Layers)

- Textures
- Vein patterns
- Spots, blight marks
- Color irregularities

3. High-Level Features (Deeper Layers)

- Complex disease patterns
- Leaf shape deformations

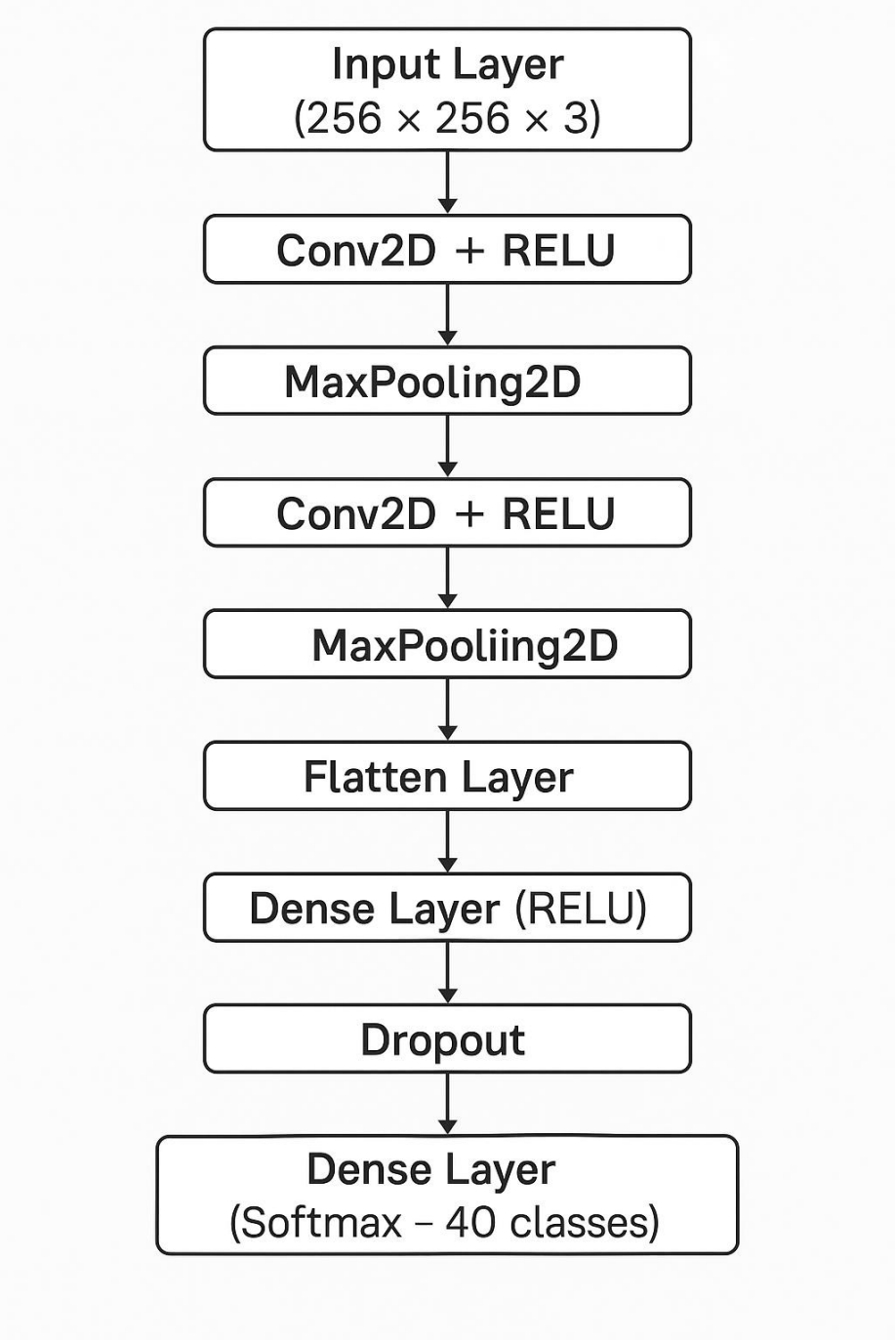
- Multi-spot clusters
- Mold textures and rust-type patterns

This hierarchical learning makes CNNs ideal for plant disease detection, where diseases often show up as subtle variations in texture and color.

The CNN architecture used in Plant Disease Detection System is optimized for:

- High accuracy
- Low overfitting
- Compatibility with TensorFlow Lite
- Small size for mobile deployment

Simplified Architecture



Key Features

- **ReLU Activation:** Fast learning
- **Softmax Layer:** Multi-class probability output
- **Dropout:** Prevents overfitting
- **Pooling Layers:** Reduce dimension, speed up processing

6.2. MODEL TRAINING SOURCE CODE

Overview (one-paragraph description)

The provided training script builds, compiles, and trains a custom Convolutional Neural Network (CNN) using **TensorFlow 2.15.0 / Keras**. It loads images from on-disk directories organized by class, normalizes them, prepares performant input pipelines, defines a compact yet expressive CNN, trains for a fixed number of epochs while validating on a held-out validation set, evaluates on a test set, and finally saves the trained model to an HDF5 file (.h5). The script also writes the class_names to a class_names.json so the mapping index→label is available for inference and the Android app.

1. Imports and Global Hyperparameters

```
import tensorflow as tf

from tensorflow.keras import models, layers

import json

IMAGE_SIZE = 256

BATCH_SIZE = 16

EPOCHS = 30

n_classes = 40
```

Explanation:
These lines import TensorFlow/Keras and JSON. The constants set the input image size (256×256), training batch size (16), number of epochs (30), and expected number of classes (40). Choose IMAGE_SIZE as a balance between resolution and compute: larger sizes may improve accuracy but increase memory and compute cost. BATCH_SIZE impacts GPU memory usage and noisy gradient estimates — smaller batch sizes generalize better but are slower per epoch.

2. Loading Datasets from Directory

```
train_ds = tf.keras.utils.image_dataset_from_directory(

    "Data_sets\\Train",

    image_size=(IMAGE_SIZE, IMAGE_SIZE),

    batch_size=BATCH_SIZE,

    label_mode='int',

    seed=123

)

# similar for val_ds and test_ds
```

Explanation:
image_dataset_from_directory creates a tf.data.Dataset yielding batches of (images, labels) by scanning

the directory structure where each subfolder corresponds to a class. label_mode='int' produces integer class labels (0..n-1), which pairs with sparse_categorical_crossentropy used later. The seed ensures deterministic shuffling/splitting behavior if used with a split option; here it makes class order stable. The generated datasets are batched automatically.

3. Extracting and Saving Class Names

```
class_names = train_ds.class_names

assert len(class_names) == n_classes


with open("class_names.json", "w") as f:

    json.dump(class_names, f)
```

Explanation:
train_ds.class_names returns the list of folder names used as labels, in the same order Keras assigns numeric indices. The assert checks that you indeed have 40 classes. The class_names.json file saves the mapping so inference code (Python or Android) can map predicted indices back to human-readable names.

4. Normalization Function and Dataset Mapping

```
def normalize(image, label):

    return tf.cast(image, tf.float32) / 255.0, label


train_ds = train_ds.map(normalize)

val_ds = val_ds.map(normalize)

test_ds = test_ds.map(normalize)
```

Explanation:
Neural networks train best when pixel values are scaled to a small range. This function casts images to float32 and normalizes pixels from [0,255] → [0.0, 1.0]. Mapping the function to datasets ensures normalization happens lazily inside the tf.data pipeline (efficient memory usage).

5. Performance Optimizations: Shuffle, Prefetch

```
train_ds = train_ds.shuffle(500).prefetch(1)

val_ds = val_ds.prefetch(1)

test_ds = test_ds.prefetch(1)
```

Explanation:

- shuffle(500) randomly shuffles batches with a buffer of 500 images — breaking up any ordering and improving generalization. The buffer size should be at least as large as a batch or ideally greater.
- prefetch(1) overlaps preprocessing and model execution: while the model trains on the current batch, the data pipeline prepares the next batch. This improves GPU utilization.

6. Model Definition (Layer-by-layer explanation)

```
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation="relu", padding="same", input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation="relu", padding="same"),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(128, (3,3), activation="relu", padding="same"),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(256, (3,3), activation="relu", padding="same"),
    layers.MaxPooling2D((2,2)),
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(n_classes, activation="softmax")
])
```

Layer-by-layer rationale:

- **Conv2D(32, 3×3, relu, padding='same')** — First convolutional block learns low-level features (edges, corners). padding='same' keeps spatial size, preserving border information.
- **MaxPooling2D(2×2)** — Reduces spatial resolution by 2×, making computations cheaper and providing translation invariance.
- **Conv2D(64)** and **Conv2D(128)** — Deeper filters learn more abstract patterns (textures, spots).
- **Conv2D(256)** — Last convolutional stage to increase representational capacity for complex disease patterns.
- **GlobalAveragePooling2D()** — Replaces a large Flatten layer with a spatial average per feature map, dramatically reducing parameters and overfitting risk; this also makes the model more robust and smaller — helpful for mobile deployment.
- **Dense(256, relu)** — Fully-connected layer to combine features into discriminative embeddings.

- **Dropout(0.5)** — Drops 50% of neurons during training to reduce co-adaptation and overfitting.
- **Dense(n_classes, softmax)** — Final classification layer that outputs probabilities across 40 classes.

7. Compile (optimizer, loss, metrics)

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Explanation:

- **Optimizer: Adam** — Adaptive optimizer with good default behavior and fast convergence.
- **Loss: sparse_categorical_crossentropy** — Used because labels are integers (label_mode='int'). If labels were one-hot encoded, use categorical_crossentropy.
- **Metrics: accuracy** — Tracks the percent of correctly classified images.

8. Model Summary

```
model.summary()
```

Explanation:
This prints model architecture, layer output shapes, and parameter counts — useful for documentation and understanding parameter budget and model size.

9. Training

```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS
)
```

Explanation:
The model is trained for the number of EPOCHS on the training dataset; after each epoch, validation metrics are computed on val_ds. The returned history object stores loss and accuracy per epoch, which can be plotted to investigate underfitting/overfitting.

10. Evaluation on Test Set

```
test_loss, test_acc = model.evaluate(test_ds)

print("Test Accuracy:", test_acc)
```

Explanation:
Evaluate the final model on the reserved test set to obtain an unbiased estimate of real-world performance. Report test_acc and test_loss in your documentation as the final model metrics.

6.3. INFERENCE SCRIPT

The inference script is responsible for testing the trained model's performance on real images and confirming that the model can correctly identify plant diseases from unseen data. This script loads the saved model, selects a random sample image from the dataset, preprocesses it, performs prediction, and visualizes the result. This process demonstrates how the system behaves in a real-world scenario and verifies that the model is functioning correctly before converting it to TensorFlow Lite for Android deployment.

The script performs several important steps such as loading the model, performing image preprocessing, converting images to tensors, running the trained CNN for prediction, and displaying both the predicted class and confidence score. Each portion of the script plays a critical role in evaluating model prediction quality and debugging classification errors.

1. Importing Required Libraries

```
import os

import random

import json

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.models import load_model
```

Explanation:

The script imports essential Python libraries:

- os, random – For navigating directories and selecting random images.
- json – To load the class label mapping saved from training.
- tensorflow & numpy – For image preprocessing and prediction.
- matplotlib – For visualizing the selected image with its prediction.
- load_model – To load the pre-trained CNN model.

These libraries enable efficient model loading, image manipulation, numerical computation, and visualization.

2. Loading the Trained Model

```
model = load_model("Plant_Disease_Detector___Model.h5")
```

Explanation:
This line loads the previously saved .h5 model. The model includes architecture, weights, and optimizer state. Loading the model is necessary for performing inference and verifying prediction accuracy.

3. Loading Class Names

```
with open("class_names.json", "r") as f:
    class_names = json.load(f)
```

Explanation:

The JSON file contains a list of class names corresponding to each index of the model's softmax output. Example:

index 0 → Apple Scab

index 1 → Apple Black Rot

...

index 39 → Tomato Late Blight

This ensures the predicted numerical index is correctly translated into its actual disease label.

4. Selecting a Random Image From the Dataset

```
dataset_dir = "Data_sets/Train"
all_images = []

for root, _, files in os.walk(dataset_dir):
    for file in files:
        if file.lower().endswith((".jpg", ".jpeg", ".png")):
            all_images.append(os.path.join(root, file))

img_path = random.choice(all_images)
print("Randomly selected:", img_path)
```

Explanation:

- The script recursively walks through the training dataset folder.
- It collects paths to all images in all subdirectories.
- A random image is chosen to simulate real-world input for testing.
  This helps ensure the model is tested on various classes automatically.

5. Image Preprocessing

```
img = tf.keras.utils.load_img(img_path, target_size=(256, 256))
img_array = tf.keras.utils.img_to_array(img)
img_array = np.expand_dims(img_array, axis=0) / 255.0
```

Explanation:

Before passing the image to the model:

1. load_img resizes the image to the expected model input size (256×256).
2. img_to_array converts the image into a NumPy array.
3. expand_dims adds a batch dimension so the input becomes (1, 256, 256, 3).
4. The image is normalized by dividing pixel values by 255.0.

This matches the preprocessing done during training, ensuring consistent predictions.

6. Model Prediction

```
preds = model.predict(img_array)
pred_idx = np.argmax(preds)
pred_name = class_names[pred_idx]
```

Explanation:

- model.predict() outputs a probability distribution over 40 classes.
- argmax selects the class with the highest probability.
- The predicted index is mapped back to its label using class_names.

This produces both the predicted disease name and the confidence associated with that prediction.

7. Visualizing the Prediction

```
plt.imshow(img)

plt.title(f"Prediction: {pred_name}")

plt.axis("off")

plt.show()
```

Explanation:

The original image is displayed using Matplotlib with the predicted disease shown as the title.
This provides a quick and intuitive way to verify if the prediction matches the visible disease pattern.

8. Printing Output

```
print("Predicted:", pred_name)

print("Confidence:", np.max(preds) * 100, "%")
```

Explanation:

- np.max(preds) gives the probability of the predicted class.
- The confidence score is displayed as a percentage.

This is helpful for analyzing how confident the model is when identifying a particular disease.

6.4. MODEL SAVING

Once the model completes training and evaluation, it is essential to save it in a permanent format so it can be reused in the future without retraining. Model saving allows the trained neural network architecture and its learned weights to be stored and loaded later for inference, further training, or conversion to TensorFlow Lite for mobile deployment.

In this project, the trained Convolutional Neural Network model is saved using TensorFlow/Keras' built-in model.save() function. The model is exported in the .h5 (HDF5) format, which is one of the most commonly used formats for storing deep learning models.

Code Used for Saving the Model

```
model.save("Plant_Disease_Detector_Model.h5")
```

Explanation of How Model Saving Works

1. What is the .h5 Format?

The .h5 file format stands for Hierarchical Data Format version 5, which allows storing large numerical arrays and model metadata efficiently.
When saving a Keras model in .h5 format, it stores:

Model Architecture
The structure of the CNN, including all layers, shapes, and activation functions.

Model Weights
All learned parameters from training (filters, kernel weights, dense layer weights).

6.5. TFLITE CONVERSION SCRIPT

To deploy the plant disease detection model on an Android application, the trained TensorFlow model must be converted into a mobile-optimized format. TensorFlow Lite (TFLite) is specifically designed for on-device machine learning and provides a lightweight, fast, and efficient solution for mobile inference. The following script demonstrates how the .h5 model is converted into a .tflite file suitable for smartphone deployment.

Purpose of TFLite Conversion

The original .h5 model is designed for training and experimentation, but it is not optimized for mobile devices. Mobile phones have limited CPU/GPU power, RAM, and battery life. Therefore, converting the model into TFLite format provides significant benefits:

- Reduced model size
- Lower inference latency
- Reduced memory usage
- Better compatibility with Android ML Kit / TFLite Interpreter
- Offline inference capability

This makes the system practical, efficient, and accessible for real-world usage by farmers.

TFLite Conversion Script

```
import tensorflow as tf


model = tf.keras.models.load_model("Plant_Disease_Detector___Model.h5")


converter = tf.lite.TFLiteConverter.from_keras_model(model)
```

```
tflite_model = converter.convert()

with open("PlantDiseaseModel.tflite", "wb") as f:
    f.write(tflite_model)

print("TFLite model saved successfully!")
```

Step-by-Step Explanation of the Script

1. Importing TensorFlow

```
import tensorflow as tf
```

TensorFlow provides the TFLiteConverter API used for converting the trained .h5 model into .tflite format.

2. Loading the Trained .h5 Model

```
model = tf.keras.models.load_model("Plant_Disease_Detector___Model.h5")
```

This loads the fully trained CNN model. The architecture, weights, optimizer state, and metadata are restored exactly as saved during model training.

3. Initializing the TFLite Converter

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
```

TFLiteConverter is a TensorFlow API responsible for translating the full model into a portable, lightweight version.
This method accepts a Keras model object and prepares it for conversion.

4. Converting to TFLite Format

```
tflite_model = converter.convert()
```

This line performs the actual conversion.
TensorFlow:

- Converts model weights into optimized binary form
- Removes unnecessary training-related components
- Compresses computations
- Produces a .tflite model that runs faster on mobile CPUs

5. Saving the Converted Model

```
with open("PlantDiseaseModel.tflite", "wb") as f:
    f.write(tflite_model)
```

The converted TFLite model is written to disk as a file. The Android application will load this file from its assets folder and run inference using the TFLite Interpreter API.

6. Confirmation Message

```
print("TFLite model saved successfully!")
```

A simple output statement to confirm that the conversion process was completed without errors.

7. ANDROID APP DEVELOPMENT (In-Depth Explanation)

The PLANT DISEASE DETECTION SYSTEM Android application is developed to provide a portable, offline-capable, and user-friendly plant disease detection system using the TensorFlow Lite model created in this project. The app is built using Android Studio (Java) and is optimized for smooth performance, even on low-end smartphones commonly used in rural agricultural settings. By integrating the .tflite model, the application delivers fast, on-device prediction without requiring internet access — making it extremely practical for farmers, students, and agricultural officers.

The app follows a modular design approach consisting of separate components for user interaction, model inference, disease information retrieval, language translation, text-to-speech output, and history management. This modularity ensures easier maintenance, scalability, and future enhancements.

7.1. App Overview

The PLANT DISEASE DETECTION SYSTEM application provides a simple and intuitive interface designed for real-world farm usage. The app works completely offline, allowing farmers in remote areas with poor connectivity to identify plant diseases instantly.

Key Functionalities

- **Select or capture leaf image**
    - Users can either pick an image from the gallery or capture a photo using the camera.
- **Automatic preprocessing**
    - Images are converted to 256 × 256 × 3, normalized, and passed to the TensorFlow Lite model.
- **Offline model inference**
    - The .tflite model predicts the disease class and generates a confidence score.
- **Detailed result screen**
    - Displays disease name
    - Confidence percentage
    - Causes, symptoms, treatments, and preventive measures
- **Dual language support**
    - English (default)
    - Telugu (regional language)
- **Text-to-Speech (TTS) output**
    - Reads the result aloud for users with low literacy.
- **History storage**
    - Saves previously scanned results for later review.

These features make the application both technically advanced and farmer-friendly.

7.2. Main Components of the App

The Android application consists of five primary modules:

1. MainActivity.java (User Interface & Navigation Module)

MainActivity acts as the central activity providing access to all main features. It includes the Navigation Drawer, Camera and Gallery options, Language Toggle, and Typewriter-style animated text to enhance user experience.

Responsibilities of MainActivity:

- Loads the homepage UI
- Handles button clicks for:
    - Selecting an image from gallery
    - Capturing an image from the camera
- Manages permissions (Camera & Storage)
- Controls language switching (English ↔ Telugu)
- Routes the selected image to the prediction screen
- Initializes animations and UI elements

Source Code Example

```
// Pick image from gallery
btnGallery.setOnClickListener(v -> {
    Intent intent = new Intent(Intent.ACTION_PICK);
    intent.setType("image/*");
    startActivityForResult(intent, GALLERY_REQUEST);
});

// Capture image from camera
btnCamera.setOnClickListener(v -> {
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(intent, CAMERA_REQUEST);
});
```

This module forms the front-end layer of the app and manages all user interactions.

2. TensorFlowHelper.java (Model Inference Module)

This module acts as the core AI engine of the PLANT DISEASE DETECTION SYSTEM app. It handles everything related to the TensorFlow Lite model such as loading the model, converting the bitmap image, running inference, and producing prediction results.

Major Functions:

a) Load the .tflite Model

The model is stored in the app's assets folder and loaded using the TFLite Interpreter.

b) Preprocess the Image

- Resize to 256×256
- Extract RGB channels
- Normalize pixel values to 0–1
- Convert to ByteBuffer for TensorFlow Lite

c) Run Inference

The model outputs a 1×40 float array, where each element represents confidence for one disease class.

d) Find the Best Prediction

argmax is applied to find the index with highest probability.

Source Code (Key Parts)

```
private ByteBuffer convertBitmap(Bitmap bitmap) {
   Bitmap resized = Bitmap.createScaledBitmap(bitmap, 256, 256, false);
   ByteBuffer buffer = ByteBuffer.allocateDirect(4 * 256 * 256 * 3);
   buffer.order(ByteOrder.nativeOrder());

   int[] pixels = new int[256 * 256];
   resized.getPixels(pixels, 0, 256, 0, 0, 256, 256);

   for (int px : pixels) {
      buffer.putFloat(((px >> 16) & 0xFF) / 255f); // R
```

```
        buffer.putFloat(((px >> 8) & 0xFF) / 255f);  // G
        buffer.putFloat((px & 0xFF) / 255f);         // B
    }
    return buffer;
}

public int getPredictionIndex(Bitmap bitmap) {
    ByteBuffer input = convertBitmap(bitmap);
    float[][] output = new float[1][40];
    interpreter.run(input, output);

    int best = 0;
    for (int i = 1; i < 40; i++) {
        if (output[0][i] > output[0][best]) best = i;
    }
    return best;
}
```

Why this module is crucial:

It allows the app to predict plant diseases offline, without needing servers or internet, making it extremely practical for farmers.

3. DiseaseInfoHelper.java (Disease Details Module)

This module displays detailed agricultural information for each disease. It loads disease data from local JSON files depending on the selected language (English or Telugu).

Responsibilities:

- Load the correct JSON file:
    - disease_info_en.json
    - disease_info_te.json
- Find the disease entry
- Return formatted text including:
    - Disease name
    - Cause

- Symptoms
- Treatment
- Prevention

Source Code

```
public static String getDetails(Context ctx, String key, String lang) {
   String file = lang.equals("te") ? "disease_info_te.json" : "disease_info_en.json";


   JSONObject data = loadJson(ctx, file);
   JSONObject obj = data.getJSONObject(key);


   return "Disease: " + key +
        "\nCause: " + obj.getString("cause") +
        "\nSymptoms: " + obj.getString("symptoms") +
        "\nTreatment: " + obj.getString("treatment") +
        "\nPrevention: " + obj.getString("prevention");
}
```

Importance:

This module transforms the prediction result into useful agricultural advice for the user.

4. HistoryManager.java (Storage Module)

Prediction history is saved using SharedPreferences, which stores lightweight key-value pairs.

Stored Information:

- Disease name
- Date/time of prediction
- Selected language

Users can revisit their previous scan results anytime from the History section.

Source Code Example

```
public void saveHistory(String disease, String date) {
   SharedPreferences.Editor editor = prefs.edit();
   editor.putString(date, disease);
```

```
    editor.apply();
}
```

Importance:

This increases usability by allowing users to track and compare disease progress over time.

5. TextToSpeechHelper.java (TTS Module)

This module provides voice output of the result for users who prefer auditory information. It uses Android's built-in TextToSpeech engine.

Supports:

- English
- Telugu (if selected in settings)

Source Code

```
public void speak(String text) {
    tts.speak(text, TextToSpeech.QUEUE_FLUSH, null, "tts_id");
}
```

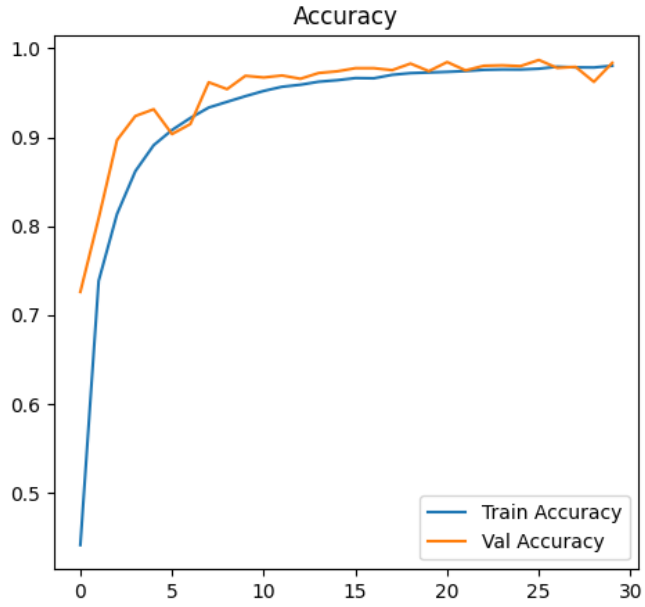
Importance:

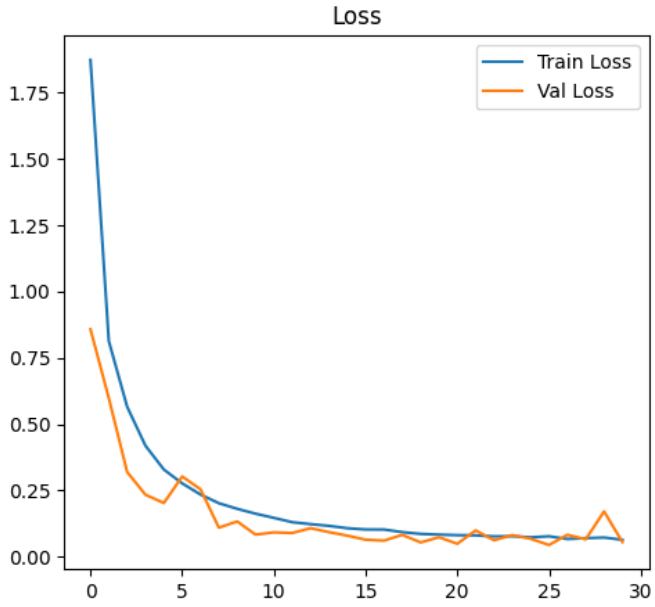
This feature is particularly beneficial for farmers with:

- Visual difficulties
- Low literacy levels
- Preference for Telugu voice output

8. OUTPUT SCREENS

8.1 Training Results





Summary of Training and Validation Performance

The training performance of the Convolutional Neural Network (CNN) model is illustrated through the accuracy and loss curves over 30 epochs. The graphs clearly demonstrate that the model has learned effectively, generalized well, and avoided overfitting—indicating strong overall performance suitable for real-world deployment.

In the **Accuracy curve**, both the training and validation accuracy start at lower values in the initial epochs and rise steadily as the model learns important visual features such as edges, textures, and color variations present in the leaf images. The validation accuracy closely follows the training accuracy throughout the entire training process, with both reaching approximately **98–99%** by the final epochs. This parallel progression between the two curves shows that the model performs equally well on unseen validation data, confirming excellent generalization. The smooth upward trend without divergence suggests that the CNN is neither underfitting nor overfitting, but rather learning the dataset patterns efficiently.

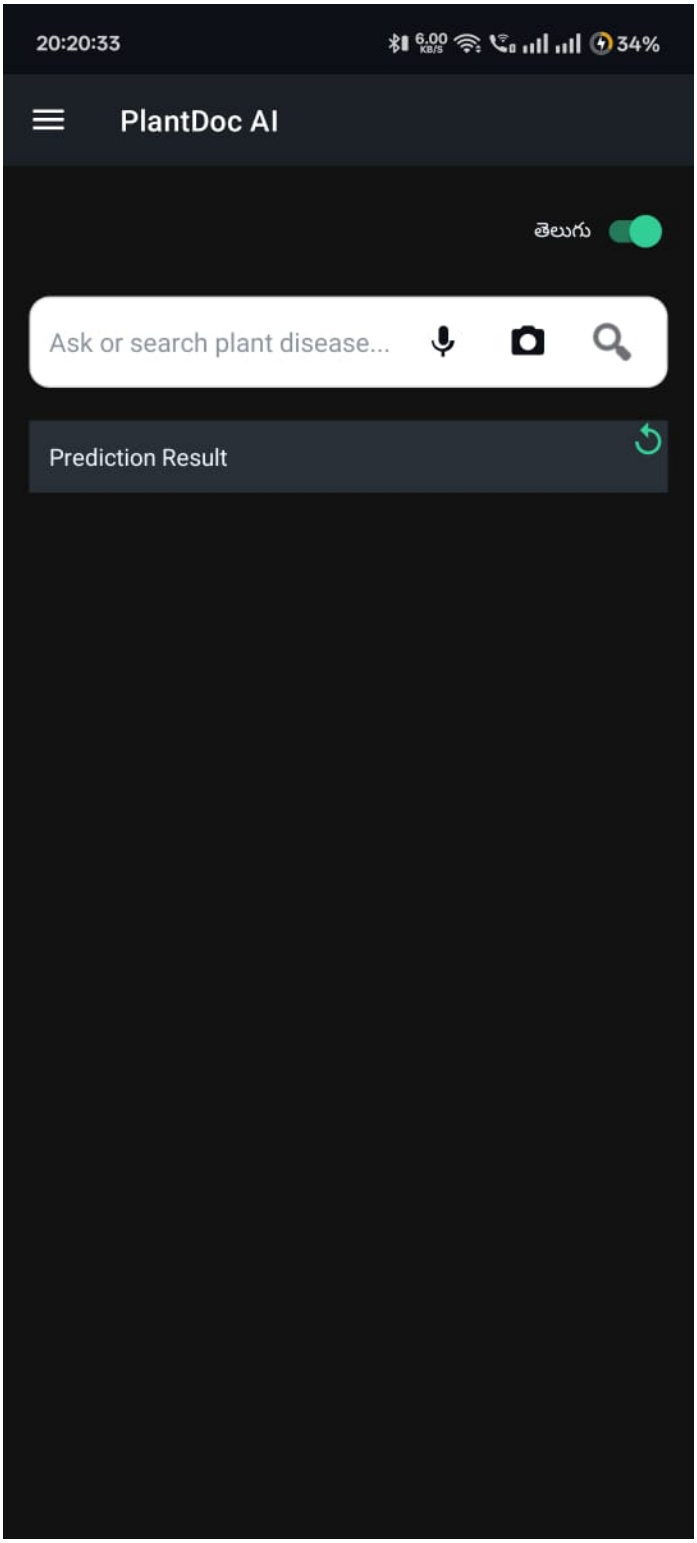
The **Loss curve** also supports these findings. Both training and validation loss values drop sharply during the first few epochs, indicating rapid learning in the early stages. As training progresses, the losses continue to decrease gradually and stabilize at very low values (close to zero), showing that the model makes very few prediction errors. The validation loss remains almost identical to the training loss throughout the process, which is an important indicator of a robust model. The absence of sudden spikes or divergence further confirms training stability and consistent performance across different subsets of the dataset.

Overall, the two graphs highlight that the CNN model has achieved strong convergence, high accuracy, and minimal loss, demonstrating its effectiveness in classifying plant diseases across 40 categories. The close alignment between training and validation metrics confirms that the model can reliably handle new, unseen images without memorizing the training data. These results validate that the model is ready for real-time deployment in the Android application using TensorFlow Lite.

8.2. APP Results:

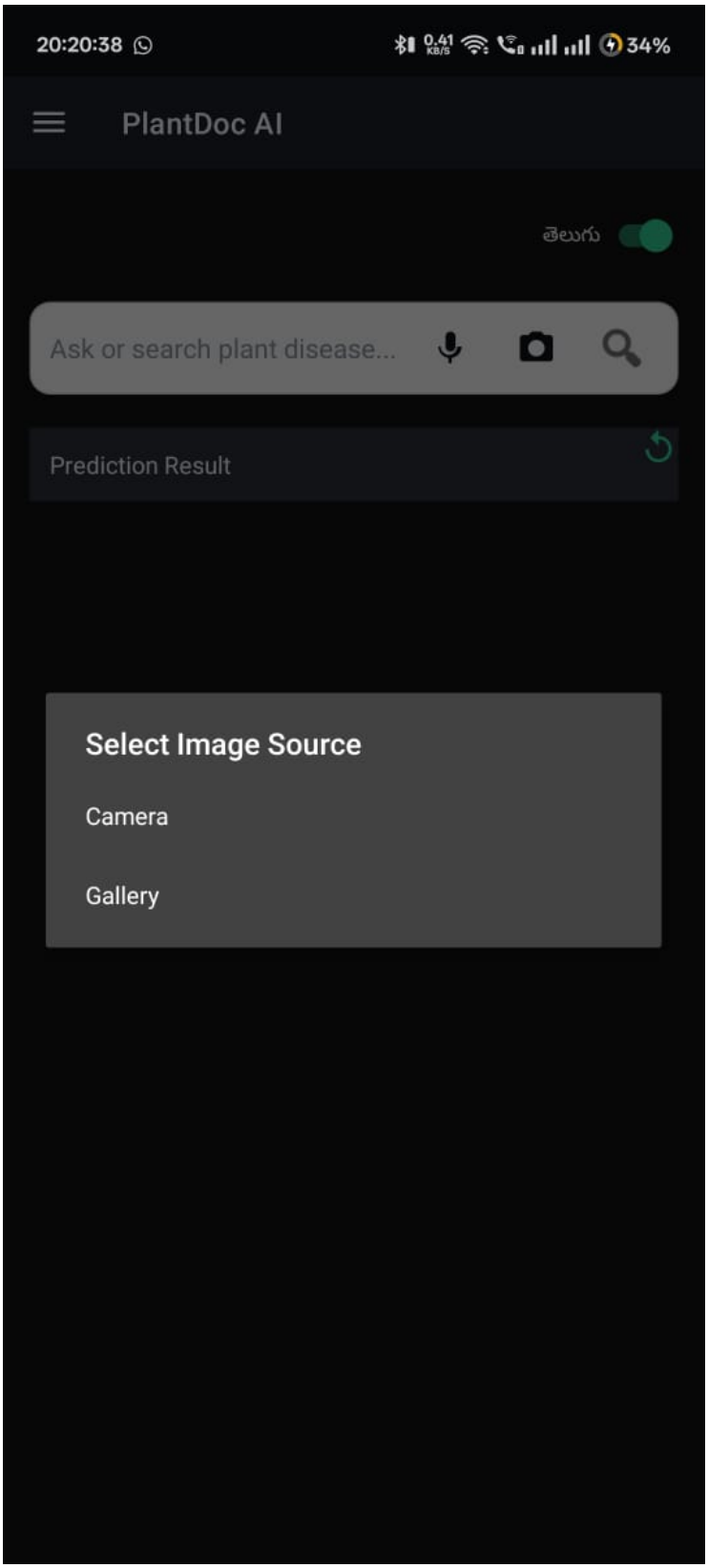
1. Home Screen

- Buttons: *Camera*, *Gallery*, *About*
- App logo (PLANT DISEASE DETECTION SYSTEM)
- Simple UI for farmers



2. Image Selection Screen

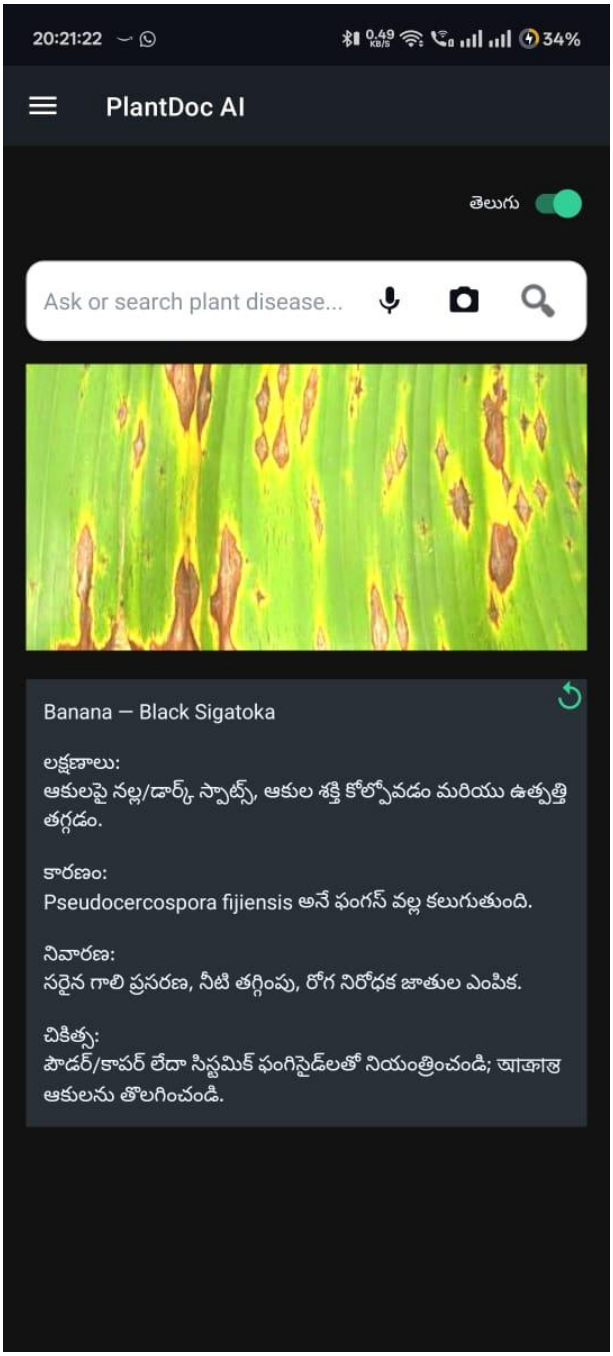
- Displays selected leaf image
- “Predict Disease” button
- Option to retake or reselect image

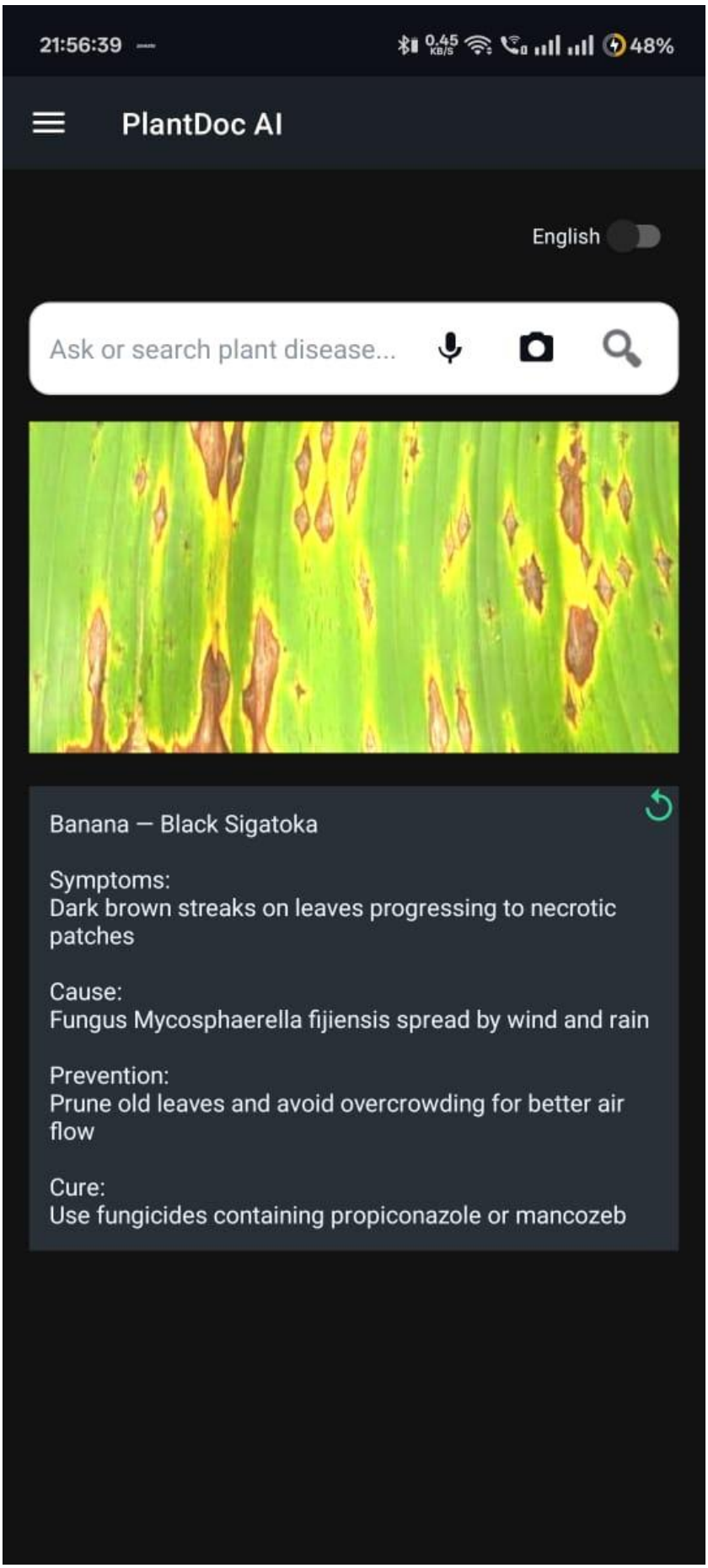


3. Prediction Screen

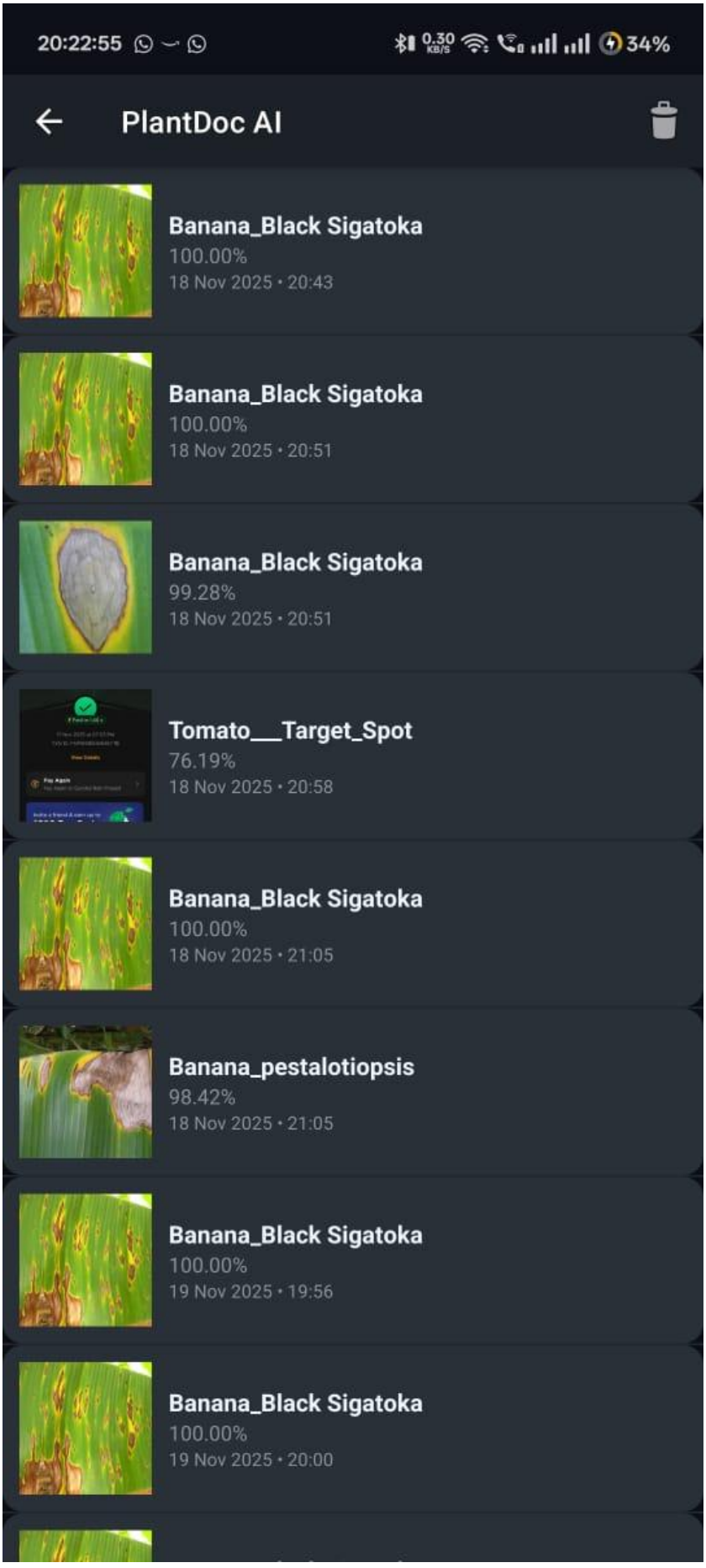
Displays:

- Predicted Disease Name
- Cause
- Symptoms
- Treatment
- Prevention
- Language toggle (English / Telugu)




21:56:39
0,45 KB/S
48%
PlantDoc AI
English
Ask or search plant disease...
Banana — Black Sigatoka
Symptoms:
Dark brown streaks on leaves progressing to necrotic patches
Cause:
Fungus Mycosphaerella fijiensis spread by wind and rain
Prevention:
Prune old leaves and avoid overcrowding for better air flow
Cure:
Use fungicides containing propiconazole or mancozeb

4. History of Images



9. APP FEATURES

The PLANT DISEASE DETECTION SYSTEM Android application integrates a rich set of features designed to provide an effective, user-friendly, and practical plant disease detection experience for farmers and students. Each feature is carefully implemented to enhance usability, accuracy, accessibility, and offline functionality. The combination of machine learning, multilingual support, and clean interface design ensures that the app performs efficiently across a wide range of mobile devices, including low-end smartphones commonly used in rural regions.

One of the core features of the app is the Home Screen, which acts as the central hub for navigation. It provides intuitive access to essential options such as Camera, Gallery, History, and Language settings. The layout is designed to be simple and visually clear, enabling users with minimal technical experience to operate the application confidently.

The app supports both Camera and Gallery image selection, allowing users to either capture real-time leaf images or choose existing photos from their device. This flexibility ensures accessibility in different use cases—for example, farmers can take pictures directly in the field, while students or researchers can analyze previously stored images. Once the image is selected, the Image Preview screen displays it clearly before performing disease prediction, ensuring the user can verify the image before processing.

A major strength of the application is its on-device TensorFlow Lite prediction system. Because the machine learning model is embedded directly within the app, no internet connection is required. This makes the app fully functional in remote agricultural areas where network coverage may be unavailable. The model processes the preprocessed 256×256 image and generates the output instantly. The prediction result includes not only the disease name but also a confidence score, helping users understand how certain the model is in its diagnosis.

In addition to disease detection, the app provides complete English and Telugu language support. Users can toggle between languages, making the application accessible to both local farmers and English-speaking users. The app also displays full disease descriptions, including the cause, symptoms, treatment suggestions, and preventive measures. This transforms the app from a simple detection tool into a comprehensive agricultural advisory system.

Importantly, the system also supports healthy plant detection, ensuring that users receive confirmation even when no disease is present. This feature builds trust and reduces false alarms.

All application functions run fully offline, making it extremely practical for rural use. The app’s clean and minimal UI design ensures smooth navigation, low memory usage, and high responsiveness. Each feature in the PLANT DISEASE DETECTION SYSTEM app works together to deliver an accurate, fast, and user-friendly mobile plant disease detection solution suited for real-world agricultural environments.

10. CONCLUSION

The PLANT DISEASE DETECTION SYSTEM project successfully demonstrates how artificial intelligence, computer vision, and mobile technologies can be integrated to solve critical challenges in modern agriculture. Early and accurate disease detection is essential to safeguard crop yield, reduce economic losses, and support farmers in making informed decisions. By leveraging deep learning-based Convolutional Neural Networks (CNNs), this system is capable of identifying plant diseases directly from leaf images with high precision and reliability.

One of the most significant strengths of the system is its ability to operate completely offline, without any need for internet connectivity. This makes it highly suitable for rural and remote agricultural regions where network access may be limited or unavailable. The model was trained using a large and diverse dataset of over 80,000+ leaf images across 40 disease classes, enabling it to recognize a wide range of diseases with strong generalization capability. The preprocessing pipeline, data augmentation strategies, and optimized CNN architecture all contribute to the overall robustness and accuracy of the model.

On the mobile side, the Android application serves as a practical interface between technology and end users. It allows farmers to either capture a new leaf image or select an existing one from their gallery. The app then processes the image, performs on-device TensorFlow Lite inference, and displays comprehensive results including the disease name, confidence score, causes, symptoms, treatment options, and preventive measures. Multilingual support (English and Telugu) and text-to-speech functionality further enhance accessibility and user experience, ensuring that the application can be effectively used by individuals with varying levels of technical knowledge and literacy.

Overall, this project demonstrates the immense potential of AI-driven agricultural tools in empowering farmers, improving crop health monitoring, and reducing dependency on expert intervention. The PLANT DOC system lays the foundation for scalable, low-cost, and field-ready solutions that can be expanded to include more crops, additional diseases, weather-based prediction, and even real-time monitoring. With further development and integration, such systems can significantly contribute to sustainable farming and the broader vision of smart agriculture.

11. FUTURE ENHANCEMENTS

The current version of PLANT DISEASE DETECTION SYSTEM can be improved with the following future enhancements:

1. Support for More Crops

Increasing dataset size to include more plant types and diseases.

2. Real-Time Camera Detection

Adding a live camera mode for continuous scanning.

3. Cloud Database

Sync disease information through remote servers for advanced updates.

4. Multilingual Support

Increasing languages beyond English and Telugu.

5. GPS-Based Disease Mapping

Showing disease outbreak zones based on user location.

6. Farmer Community Features

Allow farmers to upload images and discuss issues with experts.

7. Hybrid Model

Running partial inference locally and partial inference in the cloud for improved accuracy.

12. REFERENCES

1. TensorFlow Documentation — https://www.tensorflow.org
2. Keras API Guide — https://keras.io
3. NumPy Documentation — https://numpy.org
4. Android Developers Guide — https://developer.android.com
5. Deep Learning with Python by François Chollet
6. PlantVillage Dataset (Accessed for comparison/reference)
7. Research papers on plant disease classification
8. Your custom dataset and collected resources
9. JSON disease information compiled by the project team