PLANT DISEASE CLASSIFIER

AI/ML Model - Technical Documentation

Generated: November 17, 2025

Project Type: Deep Learning Classification System

1. PROJECT OVERVIEW

A **Plant Disease Classification System** using Deep Learning (Convolutional Neural Networks) to identify 38 different plant diseases from leaf images. The system uses TensorFlow/Keras for model training and Streamlit for web-based deployment. The model processes leaf images and provides real-time disease classification with probability scores for accurate plant health diagnosis.

Metric	Value
Total Classes	38 plant diseases
Dataset Source	PlantVillage (Kaggle)
Training Samples	54,305 images
Input Image Size	224×224 pixels (RGB)
Model Architecture	Convolutional Neural Network (CNN)
Training Epochs	5
Batch Size	32
Train/Validation Split	80%/20%

2. MACHINE LEARNING CONCEPTS

2.1 Deep Learning Architecture - CNN (Convolutional Neural Network)

A Convolutional Neural Network (CNN) is a specialized deep learning architecture designed for processing gridded data like images. It automatically learns spatial hierarchies of features through convolutional layers, making it ideal for image classification tasks. The model consists of:

Layer	Parameters	Purpose
Conv2D (Layer 1)	32 filters, 3×3 kernel	Extract low-level features (edges, textures)
MaxPooling2D	2×2 pool size	Reduce spatial dimensions, retain features
Conv2D (Layer 2)	64 filters, 3×3 kernel	Extract higher-level features (patterns, shapes)
MaxPooling2D	2×2 pool size	Further dimensionality reduction
Flatten	N/A	Convert 2D feature maps to 1D vector
Dense	256 neurons, ReLU activation	Learn non-linear combinations of features
Output Dense	38 neurons, Softmax activation	38-class probability distribution

Why CNN? CNNs are superior for image classification because they: (1) Automatically learn spatial hierarchies of features, (2) Exhibit shift and rotation invariance through convolution operations, (3) Share parameters across the image reducing model complexity, (4) Require fewer parameters than fully connected networks, (5) Perform exceptionally well on image recognition tasks.

2.2 Key Machine Learning Concepts

Concept	Explanation	Application
Convolution	Applies filters across image to detect patterns	Feature extraction from leaf images
Pooling	Reduces dimensionality while preserving features	Prevents overfitting, improves efficiency
Activation Function (ReLU)	Introduces non-linearity (max(0, x))	Enables learning complex boundaries
Softmax	Converts logits to probability distribution	Multi-class classification output
Categorical Cross-Entropy	Loss function for multi-class problems	Measures prediction error
Optimization (Adam)	Adaptive learning rate optimizer	Efficiently updates network weights
Epochs	Complete passes through training data	5 epochs used in this model
Batch Size	Number of samples processed together	32 samples per batch
Validation Split	Percentage of data for validation	20% used for testing generalization

3. IMAGE PREPROCESSING PIPELINE

Image preprocessing is a critical step that prepares raw input data for the neural network. The pipeline follows this sequence: **Input Image (JPG/PNG) → Resize (224×224) → Convert to Array → Add Batch Dimension → Normalize (0-1) → CNN Model**

3.1 Preprocessing Steps (Detailed)

Step	Operation	Reason/Purpose
1. Load Image	PIL.Image opens JPG/PNG file	Initial input from user or disk
2. Resize	Scale to 224×224 pixels	Standard CNN input size; consistent dimensions
3. Convert to Array	NumPy array format	Required for tensor operations in TensorFlow
4. Add Batch Dimension	Shape: (224, 224, 3) → (1, 224, 224, 3)	Model expects batch of images
5. Normalize	Divide pixel values by 255	Scale to [0, 1]; prevents gradient explosion
6. Forward Pass	Feed through CNN layers	Extract features and classify

Why Normalization? Networks train significantly better with normalized inputs because: (1) Prevents vanishing/exploding gradients, (2) Accelerates training convergence, (3) Improves numerical stability, (4) Reduces internal covariate shift, (5) Standardizes input scale across all features.

4. MODEL TRAINING FLOW (DETAILED)

The training process involves multiple stages from data preparation through model optimization and evaluation. Each stage is critical for creating an accurate, generalizable model.

1. DATA CURATION

Download PlantVillage dataset from Kaggle (54,305 images)
Extract 38 disease classes
Use "color" variant images for better feature extraction

2. DATA PREPARATION

Create ImageDataGenerator for preprocessing
Split: 80% training, 20% validation
Batch size: 32 samples
Target size: 224×224 pixels
Rescale: divide by 255 to normalize

3. MODEL ARCHITECTURE

Build Sequential model (linear layer stack)
Conv2D(32) + MaxPooling → Conv2D(64) + MaxPooling
Flatten → Dense(256, ReLU) → Output Dense(38, Softmax)
Total parameters: ~1.3M

4. COMPILATION

Optimizer: Adam (adaptive moment estimation)
Loss Function: Categorical cross-entropy
Metrics: Accuracy for monitoring performance

5. TRAINING

Forward pass: Input → through all layers → output probabilities
Backward propagation: Calculate gradients
Weight updates: Adam optimizer adjusts all parameters
Validation: Evaluate on held-out validation data
Repeated for 5 epochs (5 complete passes through dataset)

6. EVALUATION & STORAGE

Calculate validation accuracy
Plot training vs validation accuracy curves
Plot training vs validation loss curves
Save model as plant_disease_model.h5 (HDF5 format)

5. INFERENCE PIPELINE (DEPLOYMENT - main.py)

The inference pipeline is used when the model is deployed and makes predictions on new, unseen data. It follows this sequence: **User Upload Image** → **Load & Preprocess** → **Model Prediction** → **Class Mapping** → **Display Result**

1. STREAMLIT UI INITIALIZATION

Display title: "Plant Disease Classifier"
Create file uploader widget
Accept: JPG, JPEG, PNG formats

2. MODEL & CLASS LOADING

Load pre-trained model: plant_disease_model.h5
Load class indices: class_indices.json (38 disease mappings)
Model loaded once during app startup for efficiency

3. IMAGE UPLOAD

User selects leaf image through web interface
Image stored in temporary memory
No direct file system access required

4. PREPROCESSING

Open image using PIL
Resize to 224×224 pixels
Convert to NumPy array
Add batch dimension: (1, 224, 224, 3)
Normalize: divide by 255 → [0, 1] range

5. MODEL INFERENCE

Forward pass through 7 layers
Output: 38 probability values (sum = 1.0)
Computation time: ~100-500ms depending on hardware

6. CLASS PREDICTION

Use np.argmax() to find highest probability index
Map index to disease name using class_indices.json
Example: Index 0 → "Apple___Apple_scab"

7. RESULT DISPLAY

Display original image (150×150 px)
Show prediction result using st.success()
Format: "Prediction: [Disease_Name]"
Probability scores could be added for confidence level

6. DATASET - 38 PLANT DISEASES

The model is trained on the PlantVillage dataset containing 54,305 high-quality leaf images across 14 crop types and 38 disease/health classes. Below is the complete classification scheme:

Crop	Classes	Count
Apple	Apple_scab, Black_rot, Cedar_apple_rust, healthy	4
Blueberry	healthy	1
Cherry	Powdery_mildew, healthy	2
Corn (Maize)	Cercospora, Common_rust, Northern_Leaf_Blight, healthy	4
Grape	Black_rot, Esca, Leaf_blight, healthy	4
Orange	Haunglongbing (Citrus_greening)	1
Peach	Bacterial_spot, healthy	2
Pepper, Bell	Bacterial_spot, healthy	2
Potato	Early_blight, Late_blight, healthy	3
Raspberry	healthy	1
Soybean	healthy	1
Squash	Powdery_mildew	1
Strawberry	Leaf_scorch, healthy	2
Tomato	Bacterial_spot, Early_blight, Late_blight, Leaf_Mold, Septoria_leaf_spot, Spider_mites, Target_Spot, YLCV, TMV, h	10
TOTAL		38

7. TECHNICAL STACK & DEPENDENCIES

Component	Technology	Version	Purpose
Deep Learning Framework	TensorFlow/Keras	Latest	Model training & inference, layer abstractions
Image Processing	PIL (Pillow)	Latest	Load, resize, manipulate images
Numerical Computing	NumPy	Latest	Array operations, matrix computations
Web UI Framework	Streamlit	Latest	Interactive web interface, rapid prototyping
Data Source	PlantVillage Dataset	Kaggle	54,305 plant leaf images, 38 classes
Containerization	Docker	20.10+	Consistent deployment across environments
Configuration	TOML	Python standard	Server settings, browser configuration
Serialization	HDF5 (.h5)	TensorFlow standard	Model architecture & weights storage

8. MODEL ARCHITECTURE VISUALIZATION

Architecture Flow: INPUT LAYER (224×224×3 RGB Image) ↓ Convolutional Layer 1 (Conv2D with 32 filters, 3×3 kernel) + ReLU Activation ↓ Max Pooling Layer 1 (2×2 pool size) [Reduces spatial dimensions to ~112×112] ↓ Convolutional Layer 2 (Conv2D with 64 filters, 3×3 kernel) + ReLU Activation ↓ Max Pooling Layer 2 (2×2 pool size) [Further reduces to ~56×56] ↓ Flatten Layer [Converts all feature maps to 1D vector] ↓ Fully Connected (Dense) Layer 1 (256 neurons, ReLU activation) [Non-linear feature combinations] ↓ Output Dense Layer (38 neurons, Softmax activation) [Probability distribution across diseases] ↓ OUTPUT [38 disease probability scores, sum = 1.0]

9. HYPERPARAMETERS & CONFIGURATION

Parameter	Value	Rationale
Image Input Size	224×224 pixels	Standard for modern CNNs; balance between detai and computation
Batch Size	32 samples	Trade-off between memory usage and training stability
Training Epochs	5	Limited epochs in demo; production uses 50-200 for better convergence
Validation Split	20%	Sufficient for monitoring generalization without reducing training data
Conv2D Filters (Layer 1)	32	Captures low-level features; reduced complexity
Conv2D Filters (Layer 2)	64	Captures higher-level patterns; doubles filter count
Kernel Size	3×3	Standard for feature extraction; balance between receptive field and parameters
Pooling Size	2×2	Reduces spatial dimensions by 50% per pooling layer
Dense Layer Neurons	256	Sufficient capacity for learning feature combinations
Output Layer Neurons	38	One neuron per disease class
ReLU Activation	max(0, x)	Introduces non-linearity; standard for hidden layers
Softmax Activation	e^x / Σ(e^x)	Normalizes outputs to probabilities for multi-class
Optimizer	Adam	Adaptive learning rate; faster convergence than SGD
Learning Rate (Adam)	~0.001 (default)	Balanced convergence speed and stability
Loss Function	Categorical Cross-Entropy	Standard for multi-class classification

10. LOSS FUNCTIONS & OPTIMIZATION

10.1 Categorical Cross-Entropy Loss

The loss function quantifies the difference between predicted and actual distributions. Categorical Cross-Entropy is the standard for multi-class classification: **Formula: L = -Σ(y_i × log(■_i))** Where: • y_i = true label (one-hot encoded, 0 or 1) • ■_i = predicted probability for class i • Σ = sum across all 38 classes **Why Cross-Entropy?** It heavily penalizes confident wrong predictions, encouraging the model to learn discriminative features. It has nice mathematical properties for backpropagation and gradient descent optimization.

10.2 Adam Optimizer

Adam (Adaptive Moment Estimation) is an advanced optimization algorithm that adapts learning rates for each parameter: **Key Features:** • Maintains exponential moving average of gradients (momentum term) • Maintains exponential moving average of squared gradients (RMSprop term) • Adapts learning rate per parameter based on these moving averages • Default learning rate: 0.001 (typically requires no tuning) • Computationally efficient and memory-friendly **Advantages over Standard SGD:** • Faster convergence with fewer iterations • Handles sparse gradients well • Robust to different hyperparameter choices • Works well with mini-batches (batch size: 32)

11. DEPLOYMENT ARCHITECTURE (DOCKER)

The application is containerized using Docker for consistent deployment across different environments. The container includes all dependencies, the trained model, and configuration files.

Docker Container Structure:

Base Image: python:3.10-slim (lightweight Python runtime) **Container Setup:** • COPY: Copy all application files to /app directory • WORKDIR: Set /app as working directory • RUN: Execute pip install for dependencies • EXPOSE: Port 80 (HTTP traffic) **Configuration:** • Create ~/.streamlit directory • Copy config.toml (server settings) • Copy credentials.toml (authentication) **Entrypoint:** streamlit run main.py • Starts Streamlit server on 0.0.0.0:80 • Accessible from any network interface • Port 80: Standard HTTP port (no port forwarding needed) **Advantages:** • Reproducible deployments across servers • Isolation from host system dependencies • Easy scaling and orchestration • Version control for entire environment

12. CRITICAL ML CONCEPTS FOR DEVELOPMENT

Overfitting Prevention

Problem: Model memorizes training data instead of learning generalizable patterns.
Solution: Use validation set (20%) to monitor generalization. If val_loss increases while train_loss decreases = overfitting.
Techniques: Early stopping, dropout layers, regularization, data augmentation.

Data Augmentation

Creates variations of training images (rotations, flips, zooms, color changes).
Benefits: Increases effective training data size, improves model robustness, prevents overfitting.
Implementation: ImageDataGenerator in current code supports multiple augmentation techniques.

Feature Extraction

CNN layers learn hierarchical features: Layer 1 (edges) → Layer 2 (textures) → Dense layers (object parts).
Transfer Learning: Use pre-trained weights (ImageNet) instead of training from scratch for faster convergence.

Vanishing Gradients

Problem: Gradients become too small during backpropagation in deep networks.
Solution: ReLU activation, batch normalization, proper weight initialization.
Impact: Without mitigation, network cannot learn effectively.

Model Serialization

Save/load models for reuse without retraining. HDF5 format (.h5) stores:
• Model architecture (layer definitions)
• Weights (learned parameters)
• Training configuration
Benefit: Inference uses pre-trained weights, ~1000x faster than training.

Gradient Descent & Backpropagation

Forward Pass: Input → layers → output predictions
Backward Pass: Calculate $\partial L/\partial w$ (gradient of loss w.r.t. weights)
Update: $w_{new} = w_{old} - \alpha \times \partial L/\partial w$ (where α = learning rate)
Repeated for all layers in reverse order (hence "back" propagation).

13. PERFORMANCE METRICS & EVALUATION

The model's performance is evaluated using multiple metrics to ensure accuracy and generalization:

Accuracy

Percentage of correct predictions: (True Positives + True Negatives) / Total
Range: 0-100% (higher is better)
Usage: Overall performance metric across all classes.

Loss

Categorical Cross-Entropy loss value
Range: 0 to ∞ (lower is better)
Training Loss: Monitors learning on training data
Validation Loss: Monitors generalization on unseen data
If val_loss > train_loss consistently → overfitting.

Training vs Validation Curves

Plotted against epochs to visualize learning
Ideal: Both curves decrease smoothly and converge
Warning Signs: Validation plateau while training continues = overfitting
Metric in Code: Used to detect model performance trends.

Confusion Matrix (Optional)

Shows true/false positives/negatives per class
Identifies which diseases are confused with others
Useful for understanding model weaknesses.
Could enhance current implementation.

Per-Class Metrics (Optional)

Precision: TP / (TP + FP) - accuracy for each disease
Recall: TP / (TP + FN) - disease detection rate
F1-Score: Harmonic mean of precision & recall
Useful for imbalanced datasets.

14. COMPLETE WORKFLOW SEQUENCE

Training Phase (app.py):

① Download PlantVillage Dataset from Kaggle (54,305 images) ② Extract and organize 38 disease classes ③ Create ImageDataGenerator with rescaling (÷255) ④ Split data: 80% training, 20% validation ⑤ Build Sequential CNN model (7 layers total) ⑥ Compile with Adam optimizer & cross-entropy loss ⑦ Train for 5 epochs with batch size 32 ⑧ Monitor training/validation accuracy and loss ⑨ Evaluate on validation set ⑩ Plot performance curves (accuracy & loss) ■ Save model as plant_disease_model.h5 (HDF5 format) ■ Save class indices mapping as class_indices.json

Inference Phase (main.py):

① Load pre-trained model from plant_disease_model.h5 ② Load class indices mapping from class_indices.json ③ Initialize Streamlit web interface ④ Display title and file uploader widget ⑤ User uploads leaf image (JPG/JPEG/PNG) ⑥ Display uploaded image (150×150 px preview) ⑦ User clicks "Classify" button ⑧ Preprocess image: resize to 224×224, normalize to [0,1] ⑨ Run forward pass through CNN layers (~100-500ms) ⑩ Model outputs 38 probability values ■ Extract class with highest probability using argmax() ■ Map class index to disease name ■ Display prediction result to user ■ Ready for next image classification

15. FUTURE ENHANCEMENTS & IMPROVEMENTS

Transfer Learning

Use pre-trained ImageNet weights as starting point instead of random initialization.
Benefit: Faster training, better accuracy with fewer data, leverages general image features.
Example: Use MobileNetV2 or ResNet50 as backbone.

Data Augmentation

Apply random rotations, flips, zoom, brightness changes to training images.
Benefit: Simulates different lighting/angles, improves robustness.
Current Status: Infrastructure exists (ImageDataGenerator) but not fully utilized.

Confidence Scores

Display probability for predicted class and top-3 alternatives.
Benefit: Users understand model confidence, easier to validate predictions.
Implementation: Extract top_3 indices from prediction array.

Model Validation Metrics

Add confusion matrix, precision/recall per class, ROC curves.
Benefit: Detailed performance analysis, identify problematic classes.
Tools: scikit-learn confusion_matrix, classification_report.

Batch Prediction

Allow uploading multiple images at once.
Benefit: Process folders of farm images, generate reports.
Implementation: Loop through multiple files, collect predictions.

Mobile Deployment

Convert to TensorFlow Lite for mobile apps (Android/iOS).
Benefit: On-device inference, no internet required, faster response.
Tools: tf.lite.TFLiteConverter.

API Development

Create REST API (FastAPI/Flask) for programmatic access.
Benefit: Integration with other systems, batch processing workflows.
Endpoint: POST /predict with image → returns JSON with classification.

Explainability

Visualize which image regions most influence predictions (Grad-CAM, LIME).
Benefit: Build trust, understand model reasoning, debugging.
Tools: tf-explain, grad-cam libraries.

16. COMMON ISSUES & TROUBLESHOOTING

OutOfMemory Error

Symptom: "CUDA out of memory" or "Cannot allocate memory"
Causes: Batch size too large, model too large for GPU/RAM
Solutions: Reduce batch size (try 16 or 8), use smaller model, enable gradient checkpointing.

Model Accuracy Low (<70%)

Symptom: Validation accuracy plateau at low value
Causes: Insufficient training epochs, learning rate too high/low, bad data quality
Solutions: Train longer (50+ epochs), adjust learning rate, verify image quality.

Severe Overfitting

Symptom: Train accuracy 95%+ but validation accuracy 50%
Causes: Model too complex, insufficient data, no augmentation
Solutions: Add dropout layers, use data augmentation, collect more data, reduce model size.

Slow Inference

Symptom: Predictions take 5+ seconds
Causes: CPU inference (no GPU), model too large, disk I/O delays
Solutions: Use GPU (CUDA), quantize model, optimize preprocessing.

Image Preprocessing Errors

Symptom: "Image size mismatch" or dimension errors
Causes: Inconsistent image sizes, wrong color channels
Solutions: Always resize to 224×224, ensure RGB format (not RGBA or grayscale).

Class Index Mismatch

Symptom: Predictions show wrong disease names
Causes: class_indices.json doesn't match model training
Solutions: Regenerate class_indices.json from train_generator.class_indices.

Streamlit Not Loading

Symptom: Page keeps loading, no error message
Causes: Model loading takes too long, file path incorrect
Solutions: Add caching (@st.cache), verify file paths, check console logs.

17. CONCLUSION & KEY TAKEAWAYS

Summary This Plant Disease Classifier demonstrates a complete end-to-end deep learning pipeline for practical agricultural applications. The system combines CNNs' powerful feature extraction capabilities with Streamlit's user-friendly interface for accessible plant disease diagnosis. **Key Technical Achievements:** ✓ Implemented 7-layer CNN architecture optimized for 38-class disease classification ✓ Preprocesses images to consistent format ([0,1] normalized, 224×224 pixels) ✓ Uses Adam optimizer with categorical cross-entropy loss for stable training ✓ Achieves reasonable accuracy within 5 training epochs ✓ Deployed via Docker for reproducible, scalable deployment ✓ Provides web interface for non-technical users ✓ Leverages Kaggle's PlantVillage dataset with 54,305 high-quality images **ML Concepts Demonstrated:** • Convolutional Neural Networks and feature hierarchies • Data preprocessing and normalization techniques • Train/validation split and overfitting prevention • Backpropagation and gradient descent optimization • Model serialization and inference pipelines • Web interface development for ML models • Containerization for consistent deployment **Real-World Applications:** This system can be deployed on farms for: ✓ Early disease detection for crop management ✓ Reduce pesticide waste through targeted treatment ✓ Prevent large-scale crop failures ✓ Enable data-driven agricultural decisions ✓ Lower farming costs and increase yields **Next Steps for Production Deployment:** 1. Collect more diverse training data from different farms/regions 2. Implement data augmentation for robustness 3. Use transfer learning to improve accuracy 4. Add confidence scores for predictions 5. Create mobile app for field use 6. Integrate with farm management systems 7. Implement user feedback loop for continuous improvement 8. Deploy on edge devices for offline operation Understanding these technical details is crucial for maintaining, improving, and extending this system for practical agricultural applications.

Generated: November 17, 2025 at 14:50:55
Plant Disease Classifier - Technical Documentation