KANCHAN (02601192022)

B.TECH (AI-ML)

🌿 Project : Plant Disease Detection using CNN

This notebook shows how to create a custom Convolutional Neural Network (CNN) using PyTorch to classify different plant diseases from images. It uses the PlantVillage dataset, which includes images of both healthy and diseased leaves from various plant species.


PLANT DISEASE
DETECTION

📋 Table of Contents

🔍 Introduction:

Plant diseases lead to major crop losses globally. By using deep learning for early detection, farmers can identify and manage diseases before they spread, helping to boost crop production and ensure food security.

In [1]:
```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import matplotlib.pyplot as plt
import json
import pickle
import warnings
```

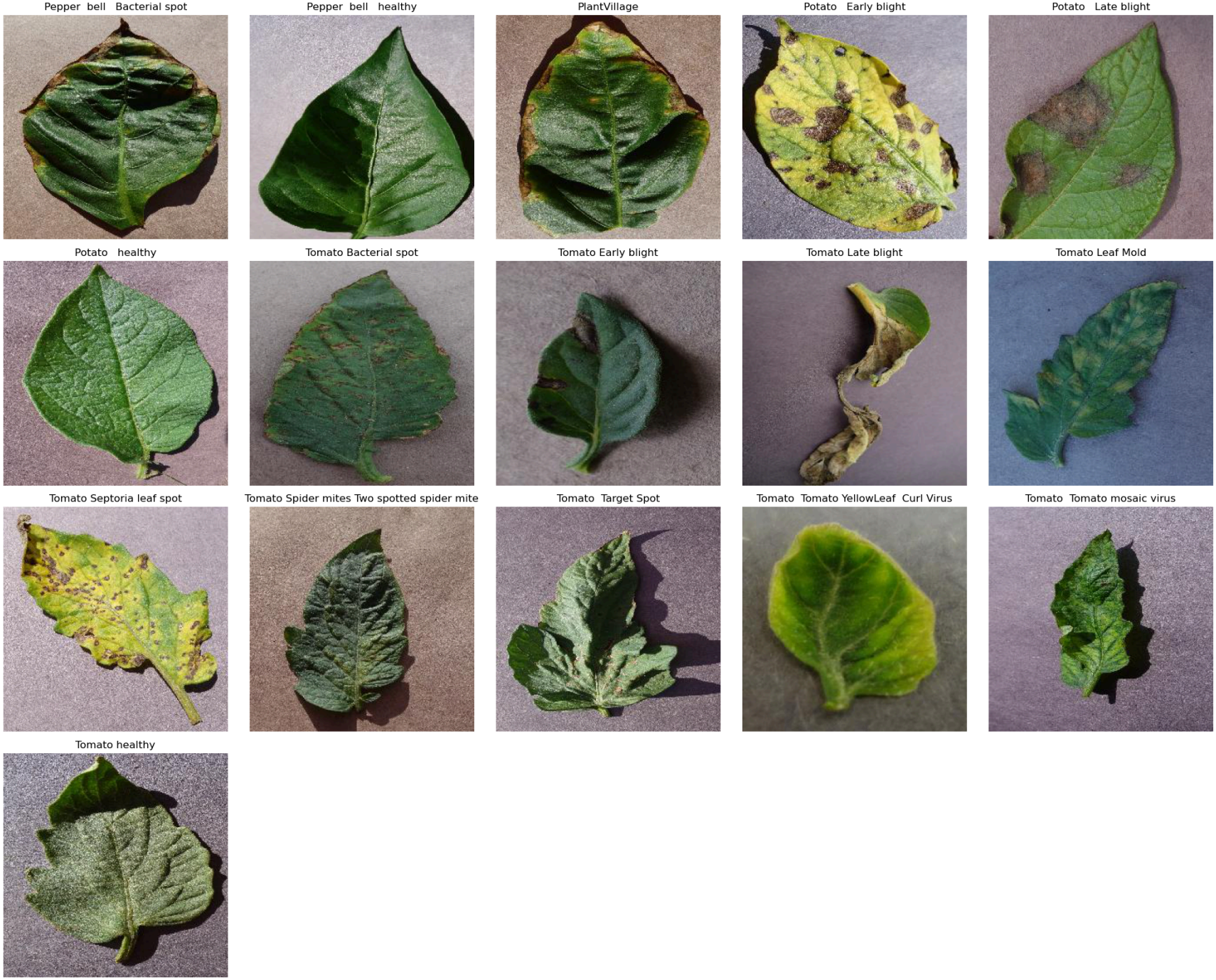
```
warnings.filterwarnings('ignore')
```

In [2]:

```
import matplotlib.pyplot as plt
import random
import os
from PIL import Image
import numpy as np

def display_disease_samples(data_dir, plants=None, num_cols=5):
    disease_folders = sorted([f for f in os.listdir(data_dir) if os.path.isdir(os.path.join(data_dir, f))])

    if plants is not None:
        disease_folders = [f for f in disease_folders if any(p in f for p in plants)]

    # Preload a fallback image from any folder
    fallback_image = None
    for folder in disease_folders:
        folder_path = os.path.join(data_dir, folder)
        img_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        if img_files:
            fallback_image = Image.open(os.path.join(folder_path, random.choice(img_files))).convert('RGB')
            break

    num_diseases = len(disease_folders)
    num_rows = (num_diseases + num_cols - 1) // num_cols
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(20, 4 * num_rows))
    axes = axes.flatten() if num_rows > 1 else [axes]

    for i, disease_folder in enumerate(disease_folders):
        folder_path = os.path.join(data_dir, disease_folder)
        img_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        disease_name = disease_folder.replace('_', ' ')

        if img_files:
            img_path = os.path.join(folder_path, random.choice(img_files))
            img = Image.open(img_path).convert('RGB')
        else:
            img = fallback_image if fallback_image else Image.new('RGB', (224, 224), color='gray')

        axes[i].imshow(img)
        axes[i].set_title(disease_name, fontsize=12)
        axes[i].axis('off')

    for j in range(i + 1, len(axes)):
```

```
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()

print("🌿 Sample images from different plant disease categories:")
display_disease_samples(r"C:\Users\Kanchan Yadav\Downloads\Plant Village Dataset\PlantVillage")
```

```
🌿 Sample images from different plant disease categories:
```


Pepper bell   Bacterial spot
Pepper bell   healthy
PlantVillage
Potato   Early blight
Potato   Late blight
Potato   healthy
Tomato Bacterial spot
Tomato Early blight
Tomato Late blight
Tomato Leaf Mold
Tomato Septoria leaf spot
Tomato Spider mites Two spotted spider mite
Tomato Target Spot
Tomato Tomato YellowLeaf Curl Virus
Tomato Tomato mosaic virus
Tomato healthy

Custom Dataset Class for Plant Disease Images

The `PlantDiseaseDataset` class is a custom dataset handler designed to load and preprocess plant disease images. It extends PyTorch's `Dataset` class and provides a structured way to manage both image data and their corresponding labels.

🔑 Key Features:

1. **Initialization (`__init__`)** Accepts image file paths, labels, and optional transformations. This allows flexible preprocessing and augmentation of the data.
2. **Length Method (`__len__`)** Returns the total number of samples in the dataset. This is useful for batching and iteration.
3. **Get Item Method (`__getitem__`)** Retrieves a single image and its corresponding label by index. If any transformations are provided, they are applied to the image before returning.

✅ Why This Class Is Important

This custom dataset class is essential for efficiently handling and preparing image data during training and evaluation. It ensures that the data pipeline integrates smoothly with PyTorch's DataLoader, enabling effective model training for the plant disease classification task.

In [3]:
```python
import torch
from torch.utils.data import Dataset
from PIL import Image

class PlantDiseaseDataset(Dataset):
    """
    Custom Dataset for loading plant disease images.

    Args:
        image_paths (list): List of image file paths.
        labels (list): Corresponding list of labels.
        transform (callable, optional): Optional transform to be applied on a sample.
    """
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform
```

```
    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        label = self.labels[idx]

        # Load image and convert to RGB format
        img = Image.open(img_path).convert("RGB")

        # Apply transformations
        if self.transform:
            img = self.transform(img)

        # Return image tensor and label as long tensor
        return img, torch.tensor(label, dtype=torch.long)
```

🌱 Convolutional Neural Network (CNN) for Plant Disease Detection

This section presents our custom Convolutional Neural Network (CNN) architecture, `PlantDiseaseModel`, specifically built for classifying plant diseases. The model uses a series of convolutional layers to extract features from images, followed by fully connected layers for classification.

🧠 Architecture Overview

The `PlantDiseaseModel` consists of the following main components:

1. **Convolutional Blocks** The model includes five convolutional blocks. Each block contains a convolutional layer, batch normalization, ReLU activation, and max pooling. These layers extract and refine visual features at increasing levels of complexity.

2. **Global Average Pooling** This layer follows the convolutional blocks, reducing each feature map to a single value. It compresses the spatial information into a feature vector while preserving important patterns.

3. **Fully Connected Layers** The pooled features are passed through fully connected (dense) layers to output class probabilities. A dropout layer is added to prevent overfitting by randomly disabling some neurons during training.

✅ Why This Architecture?

This CNN architecture is designed to strike a balance between simplicity and effectiveness. The stacked convolutional layers help the model learn detailed patterns in plant leaf images, while dropout regularization ensures better generalization to new, unseen data. This makes it highly suitable for plant disease classification tasks.

In [4]:
```python
import torch
import torch.nn as nn
from torchinfo import summary

class PlantDiseaseModel(nn.Module):
    """
    Convolutional Neural Network for Plant Disease Classification.

    Args:
        num_classes (int): Number of output classes.
        dropout_rate (float): Dropout rate for regularization.
    """
    def __init__(self, num_classes, dropout_rate=0.5):
        super(PlantDiseaseModel, self).__init__()

        # Convolutional Block 1
        self.conv_block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding="same"),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        # Block 2
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding="same"),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        # Block 3
        self.conv_block3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding="same"),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
```

```
        # Block 4
        self.conv_block4 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, padding="same"),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        # Block 5
        self.conv_block5 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, padding="same"),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        # Global Average Pooling
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))

        # Fully Connected Layers
        self.fc_block = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)
        x = self.conv_block5(x)
        x = self.global_avg_pool(x)
        x = self.fc_block(x)
        return x

# Example: Model summary for 15 classes with 224x224 input images
if __name__ == "__main__":
    model = PlantDiseaseModel(num_classes=15)
    print(summary(model, input_size=(1, 3, 224, 224)))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PlantDiseaseModel                        [1, 15]                   --
├─Sequential: 1-1                        [1, 64, 112, 112]         --
│    └─Conv2d: 2-1                       [1, 64, 224, 224]         1,792
│    └─BatchNorm2d: 2-2                  [1, 64, 224, 224]         128
│    └─ReLU: 2-3                         [1, 64, 224, 224]         --
│    └─MaxPool2d: 2-4                    [1, 64, 112, 112]         --
├─Sequential: 1-2                        [1, 128, 56, 56]          --
│    └─Conv2d: 2-5                       [1, 128, 112, 112]        73,856
│    └─BatchNorm2d: 2-6                  [1, 128, 112, 112]        256
│    └─ReLU: 2-7                         [1, 128, 112, 112]        --
│    └─MaxPool2d: 2-8                    [1, 128, 56, 56]          --
├─Sequential: 1-3                        [1, 256, 28, 28]          --
│    └─Conv2d: 2-9                       [1, 256, 56, 56]          295,168
│    └─BatchNorm2d: 2-10                 [1, 256, 56, 56]          512
│    └─ReLU: 2-11                        [1, 256, 56, 56]          --
│    └─MaxPool2d: 2-12                   [1, 256, 28, 28]          --
├─Sequential: 1-4                        [1, 512, 14, 14]          --
│    └─Conv2d: 2-13                      [1, 512, 28, 28]          1,180,160
│    └─BatchNorm2d: 2-14                 [1, 512, 28, 28]          1,024
│    └─ReLU: 2-15                        [1, 512, 28, 28]          --
│    └─MaxPool2d: 2-16                   [1, 512, 14, 14]          --
├─Sequential: 1-5                        [1, 512, 7, 7]            --
│    └─Conv2d: 2-17                      [1, 512, 14, 14]          2,359,808
│    └─BatchNorm2d: 2-18                 [1, 512, 14, 14]          1,024
│    └─ReLU: 2-19                        [1, 512, 14, 14]          --
│    └─MaxPool2d: 2-20                   [1, 512, 7, 7]            --
├─AdaptiveAvgPool2d: 1-6                 [1, 512, 1, 1]            --
├─Sequential: 1-7                        [1, 15]                   --
│    └─Flatten: 2-21                     [1, 512]                  --
│    └─Linear: 2-22                      [1, 256]                  131,328
│    └─ReLU: 2-23                        [1, 256]                  --
│    └─Dropout: 2-24                     [1, 256]                  --
│    └─Linear: 2-25                      [1, 15]                   3,855
==========================================================================================
Total params: 4,048,911
Trainable params: 4,048,911
Non-trainable params: 0
Total mult-adds (G): 3.33
==========================================================================================
Input size (MB): 0.60
Forward/backward pass size (MB): 97.95
Params size (MB): 16.20
```

```
Estimated Total Size (MB): 114.74
==========================================================================================
```

🔲 Early Stopping Utility

To improve training efficiency and avoid overfitting, we use an EarlyStopping utility. This mechanism monitors the model's validation loss and stops training once the loss ceases to improve, ensuring optimal performance without unnecessary training.

🛠️ How Early Stopping Works

1. **Patience** The `patience` parameter defines how many consecutive epochs the training should wait for a decrease in validation loss. If no significant improvement occurs within this interval, training is stopped early.
2. **Minimum Delta** The `min_delta` parameter specifies the smallest change in validation loss that qualifies as an improvement. This helps to filter out insignificant fluctuations that don't reflect real progress.
3. **Model Checkpointing** Whenever a new lowest validation loss is achieved, the current model state is saved to a specified `save_path`. This ensures the best-performing version of the model is preserved, even if later epochs lead to overfitting.

In [5]:
```python
import torch

class EarlyStopping:
    """
    Early stopping handler to stop training when validation loss stops improving.

    Args:
        patience (int): Number of epochs to wait after last improvement.
        min_delta (float): Minimum change in validation loss to qualify as improvement.
        save_path (str): Path to save the best model.
    """
    def __init__(self, patience=5, min_delta=0.001, save_path="best_model.pth"):
        self.patience = patience
        self.min_delta = min_delta
        self.save_path = save_path
        self.best_loss = float('inf')
        self.counter = 0

    def __call__(self, val_loss, model):
```

```
        """
        Check if early stopping condition is met.

        Args:
            val_loss (float): Current validation loss.
            model (torch.nn.Module): Current model.

        Returns:
            bool: True if training should stop, False otherwise.
        """
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
            # Save the best model state
            torch.save(model.state_dict(), self.save_path)
            print(f"[INFO] Model checkpoint saved to {self.save_path}")
            return False  # Continue training
        else:
            self.counter += 1
            print(f"[INFO] No improvement. Patience counter: {self.counter}/{self.patience}")
            if self.counter >= self.patience:
                print("[INFO] Early stopping triggered.")
                return True  # Stop training
        return False  # Continue training
```

Data Loading and Preparation

This part has the code to get images from folders and get them ready for training. It also converts labels into numbers and splits the data into training, validation, and testing groups.

In [6]:
```
# Data Loading and Preparation Functions
def load_images(directory_root):
    """Load images and their labels from directory structure"""
    image_list, label_list = [], []
    print("[INFO] Loading images...")

    for disease_folder in os.listdir(directory_root):
        disease_folder_path = os.path.join(directory_root, disease_folder)
        if not os.path.isdir(disease_folder_path):
            continue

        for img_name in os.listdir(disease_folder_path):
```

```
            if img_name.startswith("."):
                continue
            img_path = os.path.join(disease_folder_path, img_name)
            if img_path.lower().endswith(('.jpg', '.jpeg', '.png')):
                image_list.append(img_path)
                label_list.append(disease_folder)

    print("[INFO] Image loading completed")
    print(f"Total images: {len(image_list)}")
    return image_list, label_list

def prepare_data(directory_root, image_size=(256, 256), batch_size=32, test_size=0.3, valid_ratio=0.5, random_state=42)
    """Prepare data loaders and label encoder"""
    # Load images and labels
    image_paths, labels = load_images(directory_root)

    # Encode labels as integers
    label_encoder = LabelEncoder()
    labels_encoded = label_encoder.fit_transform(labels)

    # Save label encoder for inference
    with open('label_encoder.pkl', 'wb') as f:
        pickle.dump(label_encoder, f)

    # Save class names for reference
    class_names = list(label_encoder.classes_)
    with open('class_names.json', 'w') as f:
        json.dump(class_names, f)

    # Train, validation, and test splits
    train_paths, temp_paths, train_labels, temp_labels = train_test_split(
        image_paths, labels_encoded, test_size=test_size, random_state=random_state, stratify=labels_encoded
    )
    valid_paths, test_paths, valid_labels, test_labels = train_test_split(
        temp_paths, temp_labels, test_size=valid_ratio, random_state=random_state, stratify=temp_labels
    )

    print(f"Training samples: {len(train_paths)}")
    print(f"Validation samples: {len(valid_paths)}")
    print(f"Test samples: {len(test_paths)}")

    # Data Transformations
    train_transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.RandomHorizontalFlip(),
```

```
        transforms.RandomVerticalFlip(),
        transforms.RandomRotation(30),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    valid_test_transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    # Save image transformation for inference
    with open('inference_transform.pkl', 'wb') as f:
        pickle.dump(valid_test_transform, f)

    # Create datasets with appropriate transformations
    train_dataset = PlantDiseaseDataset(train_paths, train_labels, transform=train_transform)
    valid_dataset = PlantDiseaseDataset(valid_paths, valid_labels, transform=valid_test_transform)
    test_dataset = PlantDiseaseDataset(test_paths, test_labels, transform=valid_test_transform)

    # Create dataloaders
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, valid_loader, test_loader, len(class_names)
```

In [7]:

```
import os
import matplotlib.pyplot as plt

def plot_dataset_distribution(data_dir):
    folders = sorted([f for f in os.listdir(data_dir) if os.path.isdir(os.path.join(data_dir, f))])

    counts = {}
    for folder in folders:
        folder_path = os.path.join(data_dir, folder)
        image_count = len([f for f in os.listdir(folder_path) if f.lower().endswith(('.jpg', '.jpeg', '.png'))])
        counts[folder] = image_count

    plant_data = {}
    for folder, count in counts.items():
        # Extract plant type (e.g., "Tomato" from "Tomato_Bacterial_spot")
        if "__" in folder:
```

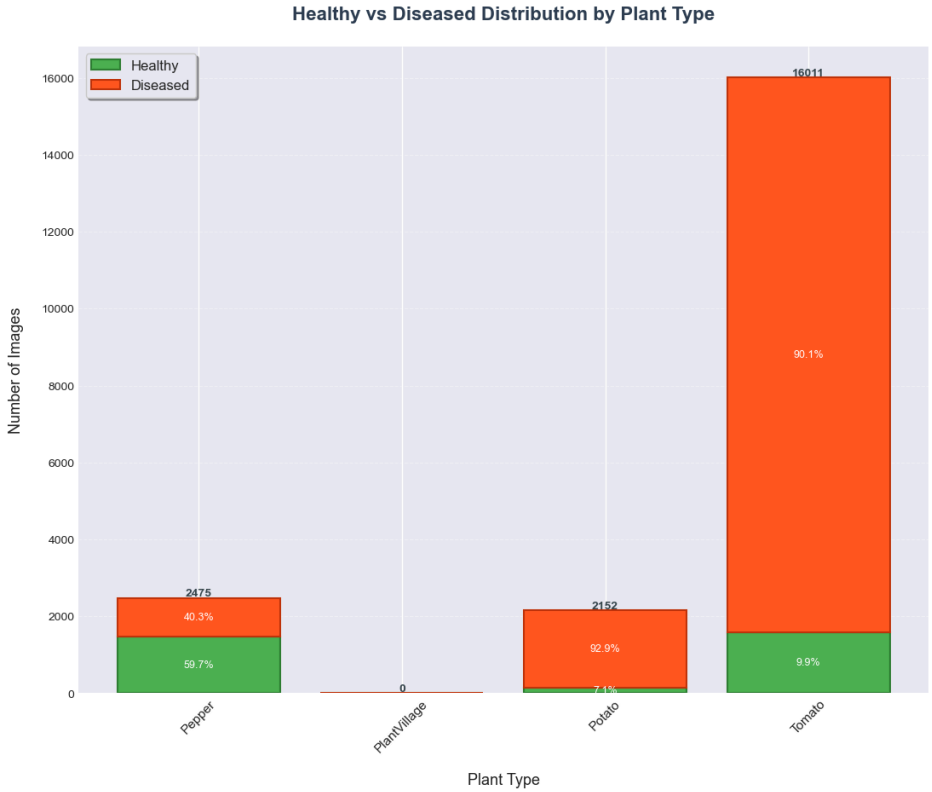
```
            parts = folder.split("__")
            plant = parts[0].replace("_", " ")
        else:
            plant = folder.split("_")[0]

        # Check if healthy or diseased
        if "healthy" in folder.lower():
            status = "Healthy"
        else:
            status = "Diseased"

        # Organize data by plant type and status
        if plant not in plant_data:
            plant_data[plant] = {"Healthy": 0, "Diseased": 0}
        plant_data[plant][status] += count

    # Create plot with enhanced styling
    plt.style.use('seaborn-darkgrid')
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(22, 10))

    # Color palette
    healthy_color = '#4CAF50'  # Modern green
    diseased_color = '#FF5722'  # Vibrant orange
    pie_colors = plt.cm.tab20.colors  # More distinct colors

    # Plot 1: Enhanced stacked bar chart
    plants = list(plant_data.keys())
    healthy_counts = [plant_data[p]["Healthy"] for p in plants]
    diseased_counts = [plant_data[p]["Diseased"] for p in plants]

    # Create gradient effect for bars
    bar1 = ax1.bar(plants, healthy_counts, label='Healthy',
                   color=healthy_color, edgecolor='#2E7D32', linewidth=1.5)
    bar2 = ax1.bar(plants, diseased_counts, bottom=healthy_counts,
                   label='Diseased', color=diseased_color, edgecolor='#BF360C', linewidth=1.5)

    # Enhanced annotations with zero-check
    for i, plant in enumerate(plants):
        total = healthy_counts[i] + diseased_counts[i]
        ax1.text(i, total + 50, f'{total}', ha='center',
                 fontsize=10, fontweight='bold', color='#37474F')
        if total > 0:
            healthy_pct = healthy_counts[i] / total * 100
            diseased_pct = diseased_counts[i] / total * 100
            ax1.text(i, healthy_counts[i]/2, f'{healthy_pct:.1f}%',
```

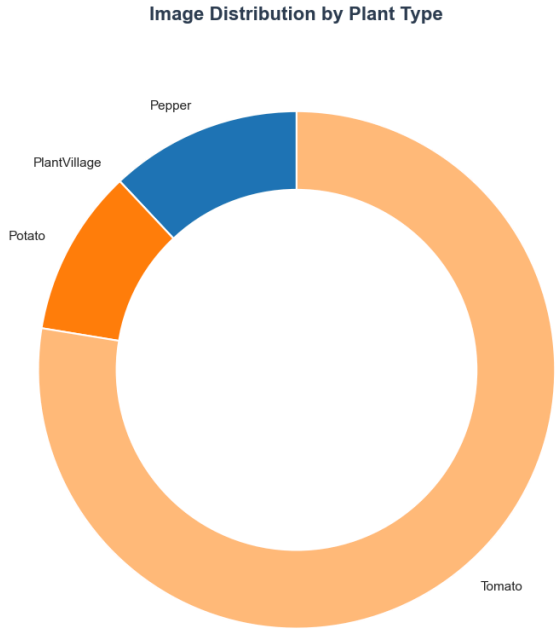
```
                ha='center', va='center', color='white', fontsize=9)
        ax1.text(i, healthy_counts[i] + diseased_counts[i]/2, f'{diseased_pct:.1f}%',
                ha='center', va='center', color='white', fontsize=9)

ax1.set_title('Healthy vs Diseased Distribution by Plant Type\n',
              fontsize=16, fontweight='bold', color='#2E4053')
ax1.set_xlabel('Plant Type', fontsize=13, labelpad=15)
ax1.set_ylabel('Number of Images', fontsize=13, labelpad=15)
ax1.tick_params(axis='x', rotation=45, labelsize=11)
ax1.legend(frameon=True, shadow=True, fontsize=12)

# Add subtle grid
ax1.yaxis.grid(True, linestyle='--', alpha=0.4)

# Plot 2: Enhanced pie chart
plant_totals = {p: plant_data[p]["Healthy"] + plant_data[p]["Diseased"] for p in plants}
wedges, texts, autotexts = ax2.pie(plant_totals.values(), labels=plant_totals.keys(),
                                   autopct='%1.1f%%', startangle=90,
                                   colors=pie_colors,
                                   wedgeprops={'linewidth': 1.5, 'edgecolor': 'white'},
                                   textprops={'fontsize': 11})

# Improve percentage formatting
for autotext in autotexts:
    autotext.set_color('white')
    autotext.set_fontweight('bold')

# Add donut effect
centre_circle = plt.Circle((0,0), 0.70, fc='white')
ax2.add_artist(centre_circle)

ax2.set_title('Image Distribution by Plant Type\n',
              fontsize=16, fontweight='bold', color='#2E4053')

plt.tight_layout(pad=3.0)

# Summary statistics with enhanced formatting
total_images = sum(plant_totals.values())
total_healthy = sum(healthy_counts)
total_diseased = sum(diseased_counts)

print(f"\n{'📊'*3} Dataset Summary {'📊'*3}")
print(f"\n🔸 Total images: \033[1m{total_images:,}\033[0m")
print(f"🔸 Healthy samples: \033[1m{total_healthy:,}\033[0m ({(total_healthy/total_images if total_images > 0 else
print(f"🔸 Diseased samples: \033[1m{total_diseased:,}\033[0m ({(total_diseased/total_images if total_images > 0 el
```

```
    print(f" 🔸 Number of plant types: \033[1m{len(plants)}\033[0m")
    print(f" 🔸 Number of disease categories: \033[1m{len(folders) - len(healthy_counts)}\033[0m\n")

# 👇 Dataset path
plot_dataset_distribution(r"C:\Users\Kanchan Yadav\Downloads\Plant Village Dataset\PlantVillage")
```

📊📊📊 Dataset Summary 📊📊📊

- 🔸 Total images: **20,638**
- 🔸 Healthy samples: **3,221** (15.6%)
- 🔸 Diseased samples: **17,417** (84.4%)
- 🔸 Number of plant types: **4**
- 🔸 Number of disease categories: **12**



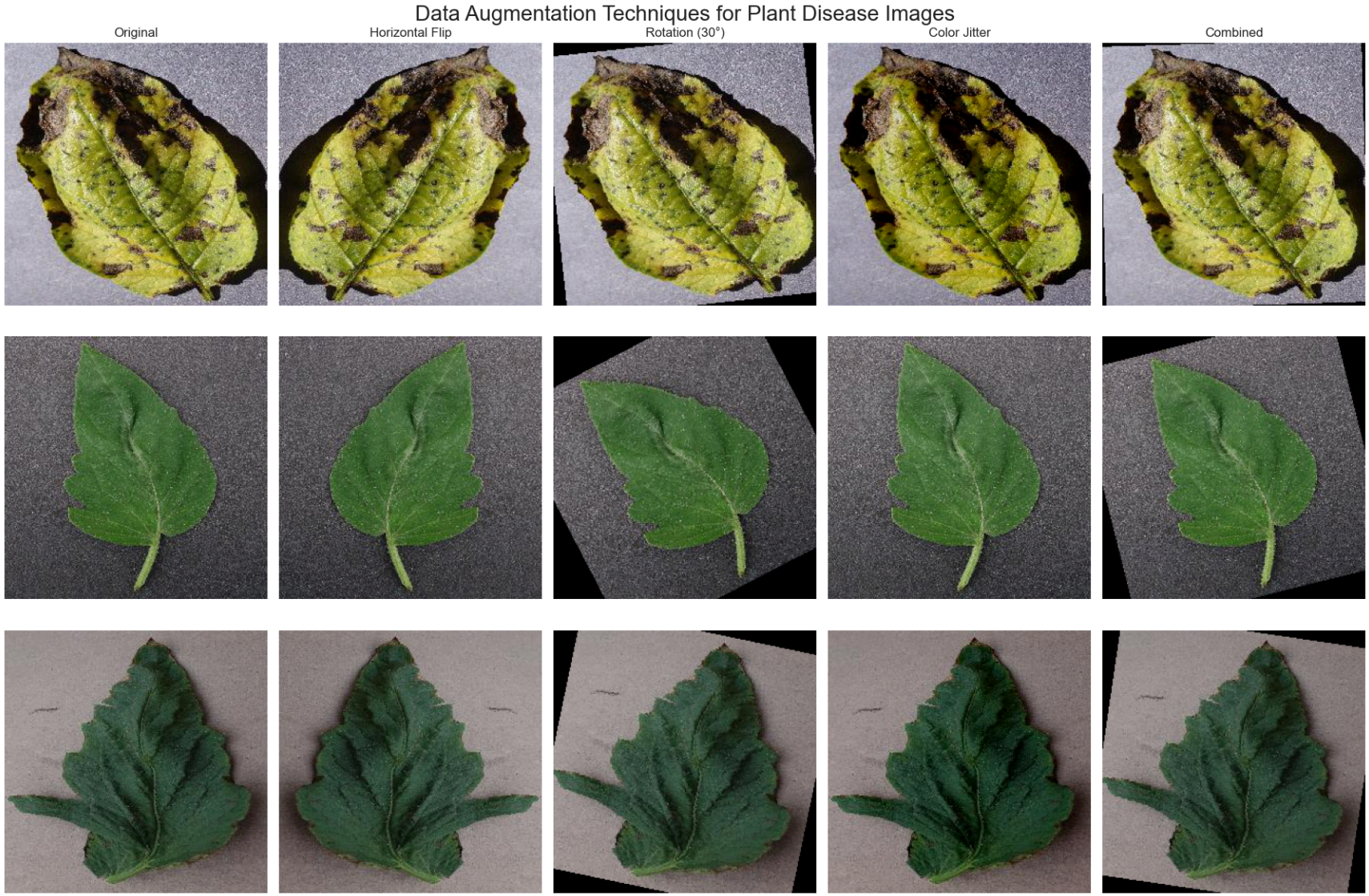


In [8]:

```
def show_augmentations(data_dir, num_plants=3):
    disease_folders = [f for f in os.listdir(data_dir) if os.path.isdir(os.path.join(data_dir, f))]
    selected_folders = random.sample(disease_folders, min(num_plants, len(disease_folders)))

    # Define augmentations to display like the training used one.
    augmentations = [
```

```
        ("Original", transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ToTensor()
        ])),
        ("Horizontal Flip", transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.RandomHorizontalFlip(p=1.0),
            transforms.ToTensor()
        ])),
        ("Rotation (30°)", transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.RandomRotation(30),
            transforms.ToTensor()
        ])),
        ("Color Jitter", transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
            transforms.ToTensor()
        ])),
        ("Combined", transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(20),
            transforms.ColorJitter(brightness=0.1, contrast=0.1),
            transforms.ToTensor()
        ]))
    ]

    fig, axes = plt.subplots(len(selected_folders), len(augmentations), figsize=(18, 4 * len(selected_folders)))

    for i, folder in enumerate(selected_folders):
        folder_path = os.path.join(data_dir, folder)

        img_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        if not img_files:
            continue

        img_path = os.path.join(folder_path, random.choice(img_files))
        original_img = Image.open(img_path).convert('RGB')

        for j, (aug_name, transform) in enumerate(augmentations):
            img_tensor = transform(original_img)

            img_np = img_tensor.permute(1, 2, 0).numpy()
```

```
            ax = axes[i, j] if len(selected_folders) > 1 else axes[j]
            ax.imshow(img_np)

            if i == 0:
                ax.set_title(aug_name, fontsize=12)

            if j == 0:
                disease_name = folder.replace('_', ' ')
                ax.set_ylabel(disease_name, fontsize=10)

            ax.axis('off')

    plt.tight_layout()
    plt.suptitle("Data Augmentation Techniques for Plant Disease Images", fontsize=20, y=1.0)
    plt.show()

show_augmentations(r"C:\Users\Kanchan Yadav\Downloads\Plant Village Dataset\PlantVillage")
```



Training and Evaluation Functions

These functions take care of training the model and checking how well it performs. They also include tools to calculate the error and accuracy, and save graphs that show how the model learned over time.

In [9]:

```python
# Training and Evaluation Functions
def evaluate_model(model, data_loader, criterion, device):
    """Evaluate model on validation or test set"""
    model.eval()
    val_loss = 0.0
    correct, total = 0, 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        progress_bar = tqdm(enumerate(data_loader), desc="Evaluating", total=len(data_loader))
        for batch_idx, (inputs, labels) in progress_bar:
            inputs, labels = inputs.to(device), labels.to(device)
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

            progress_bar.set_postfix({"Val Loss": loss.item(), "Accuracy": correct / total * 100})

    val_loss /= len(data_loader)
    accuracy = correct / total * 100
    return val_loss, accuracy, np.array(all_preds), np.array(all_labels)

def train_model(model, train_loader, valid_loader, criterion, optimizer, scheduler=None,
                epochs=10, early_stopping=None, device="cpu"):
    """Train the model with optional early stopping and learning rate scheduler"""
    model.to(device)
    train_losses, valid_losses, valid_accuracies = [], [], []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        progress_bar = tqdm(enumerate(train_loader), desc=f"Epoch {epoch+1}/{epochs}",
                            total=len(train_loader))

        for batch_idx, (inputs, labels) in progress_bar:
```

```
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            progress_bar.set_postfix({"Train Loss": loss.item()})

        # Record training loss
        train_loss = running_loss / len(train_loader)
        train_losses.append(train_loss)

        # Validation step
        val_loss, val_accuracy, _, _ = evaluate_model(model, valid_loader, criterion, device)
        valid_losses.append(val_loss)
        valid_accuracies.append(val_accuracy)

        # Print epoch summary
        print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, "
              f"Val Accuracy = {val_accuracy:.2f}%")

        # Learning rate scheduler step
        if scheduler:
            scheduler.step(val_loss)

        # Early stopping
        if early_stopping and early_stopping(val_loss, model):
            print("[INFO] Early stopping triggered.")
            break

    # Save the learning curves
    save_learning_curves(train_losses, valid_losses, valid_accuracies)

    return train_losses, valid_losses, valid_accuracies

def save_learning_curves(train_losses, valid_losses, valid_accuracies):
    """Save learning curves as a plot"""
    plt.figure(figsize=(12, 5))
```

```
    # Plot training and validation loss
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Training Loss')
    plt.plot(valid_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')

    # Plot validation accuracy
    plt.subplot(1, 2, 2)
    plt.plot(valid_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.title('Validation Accuracy')

    plt.tight_layout()
    plt.savefig('learning_curves.png')
    plt.close()
```

Prediction and Main Training Function

The prediction function lets you make guesses on individual images. The main training function runs the whole process—getting the data ready, training the model, and checking how well it works.

In [10]:
```
# Prediction function for inference
def predict_image(model, image_path, transform, device, label_encoder=None):
    """Make prediction on a single image"""
    model.eval()

    # Open and transform the image
    image = Image.open(image_path).convert("RGB")
    image_tensor = transform(image).unsqueeze(0).to(device)

    # Make prediction
    with torch.no_grad():
        outputs = model(image_tensor)
        _, predicted = torch.max(outputs, 1)
        probabilities = torch.nn.functional.softmax(outputs, dim=1)

    predicted_idx = predicted.item()
```

```
    confidence = probabilities[0][predicted_idx].item() * 100

    if label_encoder:
        predicted_class = label_encoder.inverse_transform([predicted_idx])[0]
        return predicted_class, confidence, probabilities[0].cpu().numpy()
    else:
        return predicted_idx, confidence, probabilities[0].cpu().numpy()

# Main function to train the model
def train(data_dir, model_save_path="best_model.pth", batch_size=32,
          epochs=40, learning_rate=0.01, image_size=(256, 256)):
    """Main function to train and save the model and necessary files for deployment"""
    # Prepare data
    train_loader, valid_loader, test_loader, num_classes = prepare_data(
        data_dir, image_size=image_size, batch_size=batch_size
    )

    # Setup device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Initialize model, loss function, optimizer and scheduler
    model = PlantDiseaseModel(num_classes=num_classes, dropout_rate=0.5)
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=0.1, patience=3, verbose=True
    )
    early_stopping = EarlyStopping(patience=7, min_delta=0.001, save_path=model_save_path)

    # Print model summary
    print(f"Model created with {num_classes} output classes")

    # Train the model
    train_model(
        model=model,
        train_loader=train_loader,
        valid_loader=valid_loader,
        criterion=criterion,
        optimizer=optimizer,
        scheduler=scheduler,
        epochs=epochs,
        early_stopping=early_stopping,
        device=device
```

```
    )

    # Load the best model
    model.load_state_dict(torch.load(model_save_path))

    # Evaluate on test set
    print("\n[INFO] Evaluating the model on the test set...")
    test_loss, test_accuracy, predictions, true_labels = evaluate_model(
        model, test_loader, criterion, device
    )
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%")

    # Save model architecture for inference
    dummy_input = torch.randn(1, 3, *image_size).to(device)
    torch.onnx.export(model, dummy_input, "plant_disease_model.onnx")

    # Save model config
    model_config = {
        "image_size": image_size,
        "num_classes": num_classes,
        "model_path": model_save_path,
        "label_encoder_path": "label_encoder.pkl",
        "transform_path": "inference_transform.pkl",
        "class_names_path": "class_names.json"
    }

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

    print("[INFO] Training completed and all necessary files saved for deployment.")
    return model, model_config
```

Main Execution Block:

This part sets up the main part of the script. It defines important settings and then calls the training function to start training the plant disease classification model.

The settings include:

1. **data_dir:** Where the dataset is stored.
2. **model_path:** Where the best model will be saved.

3. **batch_size:** How many samples the model looks at before updating.
4. **epochs:** How many times the model will go through the whole training data.
5. **learning_rate:** How big each step is when the model learns to reduce errors.

This block makes sure the training starts when you run the script directly.

In [11]:
```
if __name__ == "__main__":
    data_dir =  r"C:\Users\Kanchan Yadav\Downloads\Plant Village Dataset\PlantVillage"
    model_path = "best_model.pth"
    batch_size = 32
    epochs = 40
    learning_rate = 0.01

    model, model_config = train(
        data_dir=data_dir,
        model_save_path=model_path,
        batch_size=batch_size,
        epochs=epochs,
        learning_rate=learning_rate
    )
```

```
[INFO] Loading images...
[INFO] Image loading completed
Total images: 20638
Training samples: 14446
Validation samples: 3096
Test samples: 3096
Using device: cpu
Model created with 15 output classes
Epoch 1/40: 100%|██████████| 452/452 [3:13:01<00:00, 25.62s/it, Train Loss=1.77]
Evaluating: 100%|██████████| 97/97 [05:53<00:00,  3.64s/it, Val Loss=1.63, Accuracy=42.4]
Epoch 1: Train Loss = 2.2445, Val Loss = 1.7756, Val Accuracy = 42.41%
[INFO] Model checkpoint saved to best_model.pth
Epoch 2/40: 100%|██████████| 452/452 [1:05:13<00:00,  8.66s/it, Train Loss=1.72]
Evaluating: 100%|██████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=1.4, Accuracy=49.6]
Epoch 2: Train Loss = 1.9389, Val Loss = 1.5120, Val Accuracy = 49.64%
[INFO] Model checkpoint saved to best_model.pth
Epoch 3/40: 100%|██████████| 452/452 [1:04:06<00:00,  8.51s/it, Train Loss=1.93]
Evaluating: 100%|██████████| 97/97 [05:12<00:00,  3.23s/it, Val Loss=1.33, Accuracy=51.1]
Epoch 3: Train Loss = 1.8396, Val Loss = 1.4659, Val Accuracy = 51.07%
[INFO] Model checkpoint saved to best_model.pth
```

```
Epoch 4/40: 100%|████████████████████████████████████████████| 452/452 [1:03:14<00:00,  8.40s/it, Train Loss=1.34]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:09<00:00,  3.19s/it, Val Loss=1.24, Accuracy=58]
Epoch 4: Train Loss = 1.7400, Val Loss = 1.3266, Val Accuracy = 58.04%
[INFO] Model checkpoint saved to best_model.pth
Epoch 5/40: 100%|████████████████████████████████████████████| 452/452 [1:05:12<00:00,  8.66s/it, Train Loss=1.83]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:42<00:00,  3.53s/it, Val Loss=1.08, Accuracy=61.7]
Epoch 5: Train Loss = 1.6468, Val Loss = 1.1752, Val Accuracy = 61.69%
[INFO] Model checkpoint saved to best_model.pth
Epoch 6/40: 100%|████████████████████████████████████████████| 452/452 [1:04:15<00:00,  8.53s/it, Train Loss=1.43]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:13<00:00,  3.23s/it, Val Loss=1.12, Accuracy=61.2]
Epoch 6: Train Loss = 1.5683, Val Loss = 1.1353, Val Accuracy = 61.21%
[INFO] Model checkpoint saved to best_model.pth
Epoch 7/40: 100%|████████████████████████████████████████████| 452/452 [1:03:44<00:00,  8.46s/it, Train Loss=1.29]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:15<00:00,  3.26s/it, Val Loss=0.958, Accuracy=67.3]
Epoch 7: Train Loss = 1.4962, Val Loss = 0.9982, Val Accuracy = 67.34%
[INFO] Model checkpoint saved to best_model.pth
Epoch 8/40: 100%|████████████████████████████████████████████| 452/452 [1:04:29<00:00,  8.56s/it, Train Loss=1.84]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.97, Accuracy=64.7]
Epoch 8: Train Loss = 1.4429, Val Loss = 0.9895, Val Accuracy = 64.70%
[INFO] Model checkpoint saved to best_model.pth
Epoch 9/40: 100%|████████████████████████████████████████████| 452/452 [1:06:16<00:00,  8.80s/it, Train Loss=1.28]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:53<00:00,  3.64s/it, Val Loss=0.946, Accuracy=67.8]
Epoch 9: Train Loss = 1.3610, Val Loss = 0.9629, Val Accuracy = 67.76%
[INFO] Model checkpoint saved to best_model.pth
Epoch 10/40: 100%|███████████████████████████████████████████| 452/452 [1:04:35<00:00,  8.57s/it, Train Loss=0.885]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.804, Accuracy=70.9]
Epoch 10: Train Loss = 1.2733, Val Loss = 0.8750, Val Accuracy = 70.93%
[INFO] Model checkpoint saved to best_model.pth
Epoch 11/40: 100%|███████████████████████████████████████████| 452/452 [1:03:08<00:00,  8.38s/it, Train Loss=1.28]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.636, Accuracy=78.5]
Epoch 11: Train Loss = 1.1902, Val Loss = 0.6669, Val Accuracy = 78.49%
[INFO] Model checkpoint saved to best_model.pth
Epoch 12/40: 100%|███████████████████████████████████████████| 452/452 [1:03:27<00:00,  8.42s/it, Train Loss=1.02]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.665, Accuracy=79.3]
Epoch 12: Train Loss = 1.1171, Val Loss = 0.6780, Val Accuracy = 79.26%
[INFO] No improvement. Patience counter: 1/7
Epoch 13/40: 100%|███████████████████████████████████████████| 452/452 [1:03:32<00:00,  8.43s/it, Train Loss=1.44]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.467, Accuracy=82.7]
Epoch 13: Train Loss = 1.0577, Val Loss = 0.5642, Val Accuracy = 82.72%
[INFO] Model checkpoint saved to best_model.pth
Epoch 14/40: 100%|███████████████████████████████████████████| 452/452 [1:03:28<00:00,  8.43s/it, Train Loss=0.715]
Evaluating: 100%|████████████████████████████████████████████| 97/97 [05:08<00:00,  3.19s/it, Val Loss=0.539, Accuracy=82.2]
```

```
Epoch 14: Train Loss = 1.0102, Val Loss = 0.5364, Val Accuracy = 82.20%
[INFO] Model checkpoint saved to best_model.pth
Epoch 15/40: 100%|████████████████████████████████████████| 452/452 [1:03:32<00:00,  8.43s/it, Train Loss=0.793]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:11<00:00,  3.21s/it, Val Loss=0.861, Accuracy=80.6]
Epoch 15: Train Loss = 0.9557, Val Loss = 0.5588, Val Accuracy = 80.62%
[INFO] No improvement. Patience counter: 1/7
Epoch 16/40: 100%|████████████████████████████████████████| 452/452 [1:03:48<00:00,  8.47s/it, Train Loss=0.411]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:13<00:00,  3.23s/it, Val Loss=0.319, Accuracy=85.8]
Epoch 16: Train Loss = 0.9042, Val Loss = 0.4534, Val Accuracy = 85.76%
[INFO] Model checkpoint saved to best_model.pth
Epoch 17/40: 100%|████████████████████████████████████████| 452/452 [1:03:25<00:00,  8.42s/it, Train Loss=0.561]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:09<00:00,  3.19s/it, Val Loss=0.511, Accuracy=85.3]
Epoch 17: Train Loss = 0.8553, Val Loss = 0.4794, Val Accuracy = 85.34%
[INFO] No improvement. Patience counter: 1/7
Epoch 18/40: 100%|████████████████████████████████████████| 452/452 [1:03:55<00:00,  8.48s/it, Train Loss=0.823]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:14<00:00,  3.24s/it, Val Loss=0.433, Accuracy=87]
Epoch 18: Train Loss = 0.7762, Val Loss = 0.3996, Val Accuracy = 86.98%
[INFO] Model checkpoint saved to best_model.pth
Epoch 19/40: 100%|████████████████████████████████████████| 452/452 [1:06:47<00:00,  8.87s/it, Train Loss=0.391]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:40<00:00,  3.51s/it, Val Loss=0.181, Accuracy=90.3]
Epoch 19: Train Loss = 0.7168, Val Loss = 0.3119, Val Accuracy = 90.28%
[INFO] Model checkpoint saved to best_model.pth
Epoch 20/40: 100%|████████████████████████████████████████| 452/452 [1:06:45<00:00,  8.86s/it, Train Loss=1.42]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:54<00:00,  3.66s/it, Val Loss=0.297, Accuracy=89]
Epoch 20: Train Loss = 0.7069, Val Loss = 0.3184, Val Accuracy = 88.95%
[INFO] No improvement. Patience counter: 1/7
Epoch 21/40: 100%|████████████████████████████████████████| 452/452 [1:07:07<00:00,  8.91s/it, Train Loss=0.844]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:56<00:00,  3.68s/it, Val Loss=0.36, Accuracy=87.8]
Epoch 21: Train Loss = 0.6561, Val Loss = 0.3690, Val Accuracy = 87.82%
[INFO] No improvement. Patience counter: 2/7
Epoch 22/40: 100%|████████████████████████████████████████| 452/452 [2:04:22<00:00, 16.51s/it, Train Loss=0.653]
Evaluating: 100%|████████████████████████████████████████| 97/97 [10:49<00:00,  6.69s/it, Val Loss=0.33, Accuracy=92]
Epoch 22: Train Loss = 0.6085, Val Loss = 0.2417, Val Accuracy = 92.02%
[INFO] Model checkpoint saved to best_model.pth
Epoch 23/40: 100%|████████████████████████████████████████| 452/452 [1:45:12<00:00, 13.96s/it, Train Loss=0.56]
Evaluating: 100%|████████████████████████████████████████| 97/97 [06:23<00:00,  3.96s/it, Val Loss=0.122, Accuracy=92.4]
Epoch 23: Train Loss = 0.5875, Val Loss = 0.2368, Val Accuracy = 92.38%
[INFO] Model checkpoint saved to best_model.pth
Epoch 24/40: 100%|████████████████████████████████████████| 452/452 [1:12:49<00:00,  9.67s/it, Train Loss=0.554]
Evaluating: 100%|████████████████████████████████████████| 97/97 [06:30<00:00,  4.03s/it, Val Loss=0.147, Accuracy=92.5]
Epoch 24: Train Loss = 0.5358, Val Loss = 0.2427, Val Accuracy = 92.54%
[INFO] No improvement. Patience counter: 1/7
```

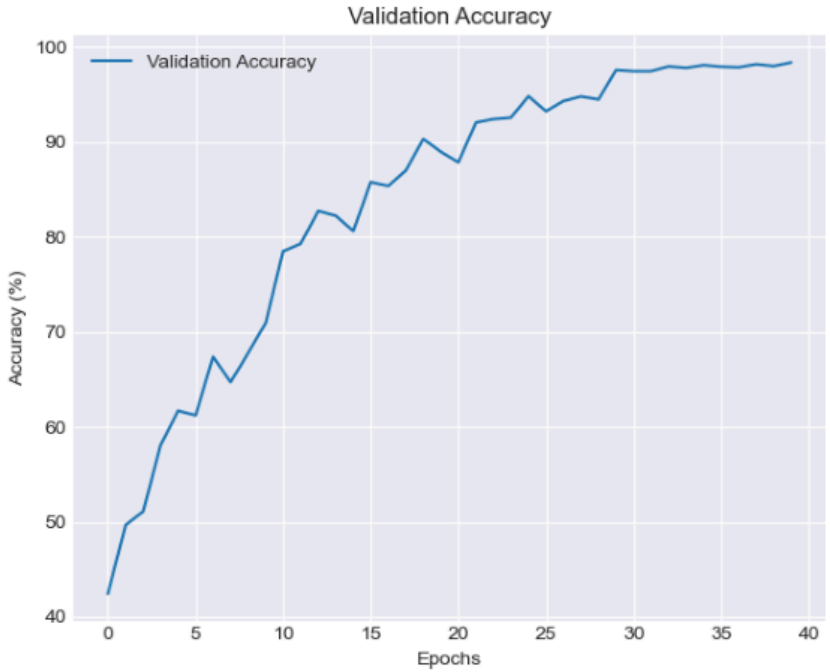
```
Epoch 25/40: 100%|████████████████████████████████████████| 452/452 [1:08:53<00:00,  9.14s/it, Train Loss=0.639]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:36<00:00,  3.47s/it, Val Loss=0.1, Accuracy=94.8]
Epoch 25: Train Loss = 0.5046, Val Loss = 0.1605, Val Accuracy = 94.80%
[INFO] Model checkpoint saved to best_model.pth
Epoch 26/40: 100%|████████████████████████████████████████| 452/452 [1:05:38<00:00,  8.71s/it, Train Loss=0.182]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:35<00:00,  3.46s/it, Val Loss=0.188, Accuracy=93.2]
Epoch 26: Train Loss = 0.4981, Val Loss = 0.2152, Val Accuracy = 93.22%
[INFO] No improvement. Patience counter: 1/7
Epoch 27/40: 100%|████████████████████████████████████████| 452/452 [1:06:28<00:00,  8.82s/it, Train Loss=1.11]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:52<00:00,  3.64s/it, Val Loss=0.174, Accuracy=94.3]
Epoch 27: Train Loss = 0.4977, Val Loss = 0.1701, Val Accuracy = 94.28%
[INFO] No improvement. Patience counter: 2/7
Epoch 28/40: 100%|████████████████████████████████████████| 452/452 [1:06:53<00:00,  8.88s/it, Train Loss=0.858]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:49<00:00,  3.61s/it, Val Loss=0.14, Accuracy=94.8]
Epoch 28: Train Loss = 0.4731, Val Loss = 0.1667, Val Accuracy = 94.77%
[INFO] No improvement. Patience counter: 3/7
Epoch 29/40: 100%|████████████████████████████████████████| 452/452 [1:06:16<00:00,  8.80s/it, Train Loss=0.621]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:11<00:00,  3.21s/it, Val Loss=0.167, Accuracy=94.4]
Epoch 29: Train Loss = 0.4515, Val Loss = 0.1746, Val Accuracy = 94.44%
[INFO] No improvement. Patience counter: 4/7
Epoch 30/40: 100%|████████████████████████████████████████| 452/452 [1:04:05<00:00,  8.51s/it, Train Loss=0.347]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.0843, Accuracy=97.5]
Epoch 30: Train Loss = 0.3481, Val Loss = 0.0786, Val Accuracy = 97.55%
[INFO] Model checkpoint saved to best_model.pth
Epoch 31/40: 100%|████████████████████████████████████████| 452/452 [1:04:35<00:00,  8.57s/it, Train Loss=0.218]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:14<00:00,  3.24s/it, Val Loss=0.0676, Accuracy=97.4]
Epoch 31: Train Loss = 0.3121, Val Loss = 0.0735, Val Accuracy = 97.42%
[INFO] Model checkpoint saved to best_model.pth
Epoch 32/40: 100%|████████████████████████████████████████| 452/452 [1:04:38<00:00,  8.58s/it, Train Loss=0.116]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:13<00:00,  3.23s/it, Val Loss=0.0626, Accuracy=97.4]
Epoch 32: Train Loss = 0.2980, Val Loss = 0.0740, Val Accuracy = 97.42%
[INFO] No improvement. Patience counter: 1/7
Epoch 33/40: 100%|████████████████████████████████████████| 452/452 [1:04:26<00:00,  8.55s/it, Train Loss=0.801]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:15<00:00,  3.25s/it, Val Loss=0.059, Accuracy=97.9]
Epoch 33: Train Loss = 0.2832, Val Loss = 0.0644, Val Accuracy = 97.93%
[INFO] Model checkpoint saved to best_model.pth
Epoch 34/40: 100%|████████████████████████████████████████| 452/452 [1:04:44<00:00,  8.60s/it, Train Loss=0.161]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:14<00:00,  3.24s/it, Val Loss=0.0688, Accuracy=97.8]
Epoch 34: Train Loss = 0.2899, Val Loss = 0.0700, Val Accuracy = 97.77%
[INFO] No improvement. Patience counter: 1/7
Epoch 35/40: 100%|████████████████████████████████████████| 452/452 [1:04:29<00:00,  8.56s/it, Train Loss=0.0464]
Evaluating: 100%|████████████████████████████████████████| 97/97 [05:13<00:00,  3.23s/it, Val Loss=0.0467, Accuracy=98.1]
```

```
Epoch 35: Train Loss = 0.2822, Val Loss = 0.0659, Val Accuracy = 98.06%
[INFO] No improvement. Patience counter: 2/7
Epoch 36/40: 100%|████████████████████████████████████████| 452/452 [1:04:46<00:00,  8.60s/it, Train Loss=0.295]
Evaluating: 100%|████████████████████████████████████| 97/97 [05:22<00:00,  3.33s/it, Val Loss=0.0384, Accuracy=97.9]
Epoch 36: Train Loss = 0.2797, Val Loss = 0.0595, Val Accuracy = 97.90%
[INFO] Model checkpoint saved to best_model.pth
Epoch 37/40: 100%|████████████████████████████████████████| 452/452 [1:05:06<00:00,  8.64s/it, Train Loss=0.184]
Evaluating: 100%|████████████████████████████████████| 97/97 [05:12<00:00,  3.22s/it, Val Loss=0.0334, Accuracy=97.8]
Epoch 37: Train Loss = 0.2559, Val Loss = 0.0646, Val Accuracy = 97.84%
[INFO] No improvement. Patience counter: 1/7
Epoch 38/40: 100%|████████████████████████████████████████| 452/452 [1:05:27<00:00,  8.69s/it, Train Loss=0.181]
Evaluating: 100%|████████████████████████████████████| 97/97 [05:17<00:00,  3.27s/it, Val Loss=0.0503, Accuracy=98.2]
Epoch 38: Train Loss = 0.2662, Val Loss = 0.0584, Val Accuracy = 98.16%
[INFO] Model checkpoint saved to best_model.pth
Epoch 39/40: 100%|████████████████████████████████████████| 452/452 [1:06:58<00:00,  8.89s/it, Train Loss=0.385]
Evaluating: 100%|████████████████████████████████████| 97/97 [06:02<00:00,  3.74s/it, Val Loss=0.0331, Accuracy=98]
Epoch 39: Train Loss = 0.2560, Val Loss = 0.0603, Val Accuracy = 97.97%
[INFO] No improvement. Patience counter: 1/7
Epoch 40/40: 100%|████████████████████████████████████████| 452/452 [1:07:03<00:00,  8.90s/it, Train Loss=0.619]
Evaluating: 100%|████████████████████████████████████| 97/97 [05:46<00:00,  3.57s/it, Val Loss=0.0448, Accuracy=98.4]
Epoch 40: Train Loss = 0.2592, Val Loss = 0.0544, Val Accuracy = 98.35%
[INFO] Model checkpoint saved to best_model.pth

[INFO] Evaluating the model on the test set...
Evaluating: 100%|████████████████████████████████████| 97/97 [05:51<00:00,  3.62s/it, Val Loss=0.0859, Accuracy=97.6]
Test Loss: 0.0710, Test Accuracy: 97.61%
[INFO] Training completed and all necessary files saved for deployment.
```

In [12]:

```python
def visualize_learning_curves():
    try:
        # Load training history
        with open('learning_curves.png', 'rb') as f:
            plt.figure(figsize=(12, 5))
            img = plt.imread('learning_curves.png')
            plt.imshow(img)
            plt.axis('off')
            plt.tight_layout()
            plt.show()
    except FileNotFoundError:
        print("Learning curves not found. Train the model first.")

print("📊 Visualizing Model Training Progress:")
visualize_learning_curves()
```

📊 Visualizing Model Training Progress:



In [13]:

```python
def evaluate_by_plant_type(model, test_loader, label_encoder, device):
    """Evaluate model performance separately for each plant type with enhanced visualization"""
    model.eval()

    # Prepare containers for per-class metrics
    class_correct = {}
    class_total = {}

    # Get all predictions
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            # Store predictions and true labels
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
```

```
        # Update per-class counts
        for i, label in enumerate(labels):
            label_idx = label.item()
            label_name = label_encoder.inverse_transform([label_idx])[0]

            if label_name not in class_correct:
                class_correct[label_name] = 0
                class_total[label_name] = 0

            class_total[label_name] += 1
            if preds[i] == label:
                class_correct[label_name] += 1

# Extract plant types from class names
plants = {}
for class_name in class_correct.keys():
    if "__" in class_name:
        plant = class_name.split("__")[0].replace("_", " ")
    else:
        plant = class_name.split("_")[0]

    if plant not in plants:
        plants[plant] = {"correct": 0, "total": 0}

    plants[plant]["correct"] += class_correct[class_name]
    plants[plant]["total"] += class_total[class_name]

# Compute accuracy per plant type
plant_accuracy = {p: (stats["correct"] / stats["total"]) * 100
                  for p, stats in plants.items()}

# Sort plants by accuracy for better visual comparison
sorted_plants = dict(sorted(plant_accuracy.items(), key=lambda x: x[1], reverse=True))

# Enhanced Visualization
plt.style.use('ggplot')
fig = plt.figure(figsize=(16, 10))
ax = fig.add_subplot(111)

# Improved color configuration
colors = plt.cm.RdYlGn(np.linspace(0.2, 0.8, len(sorted_plants)))

# Calculate average accuracy
avg_accuracy = np.mean(list(plant_accuracy.values()))
```

```python
# Create bars with enhanced styling
plants_list = list(sorted_plants.keys())
accuracies = list(sorted_plants.values())
totals = [plants[p]["total"] for p in plants_list]

# Create gradient background
ax.set_facecolor('#f8f9fa')
fig.patch.set_facecolor('#ffffff')

# Create enhanced bars
bars = ax.bar(plants_list, accuracies, color=colors, edgecolor='#505050',
              linewidth=1, alpha=0.85, width=0.7)

# Add drop shadow effect to bars
for bar in bars:
    x, y = bar.get_xy()
    w, h = bar.get_width(), bar.get_height()
    shadow = plt.Rectangle((x+0.03, y-0.03), w, h, color='#00000022', zorder=0)
    ax.add_patch(shadow)

# Add annotations with improved styling
for bar, acc, total in zip(bars, accuracies, totals):
    height = bar.get_height()
    # Add accuracy labels
    ax.text(bar.get_x() + bar.get_width()/2., height + 1,
            f'{acc:.1f}%',
            ha='center', va='bottom',
            fontsize=11, fontweight='bold',
            bbox=dict(boxstyle="round,pad=0.3", fc='white', ec="grey", alpha=0.8))

    # Add sample size labels
    ax.text(bar.get_x() + bar.get_width()/2., height/2,
            f'n={total}',
            ha='center', va='center',
            fontsize=10, color='#303030',
            fontweight='bold', rotation=0)

# Add reference lines and styling
ax.axhline(avg_accuracy, color='#e74c3c', linestyle='-', linewidth=2.5, alpha=0.7)
ax.axhline(avg_accuracy, color='#c0392b', linestyle='-', linewidth=1, alpha=1)

# Add average line label with enhanced styling
ax.text(len(plants_list)-0.5, avg_accuracy + 3,
        f' Average: {avg_accuracy:.1f}%',
```

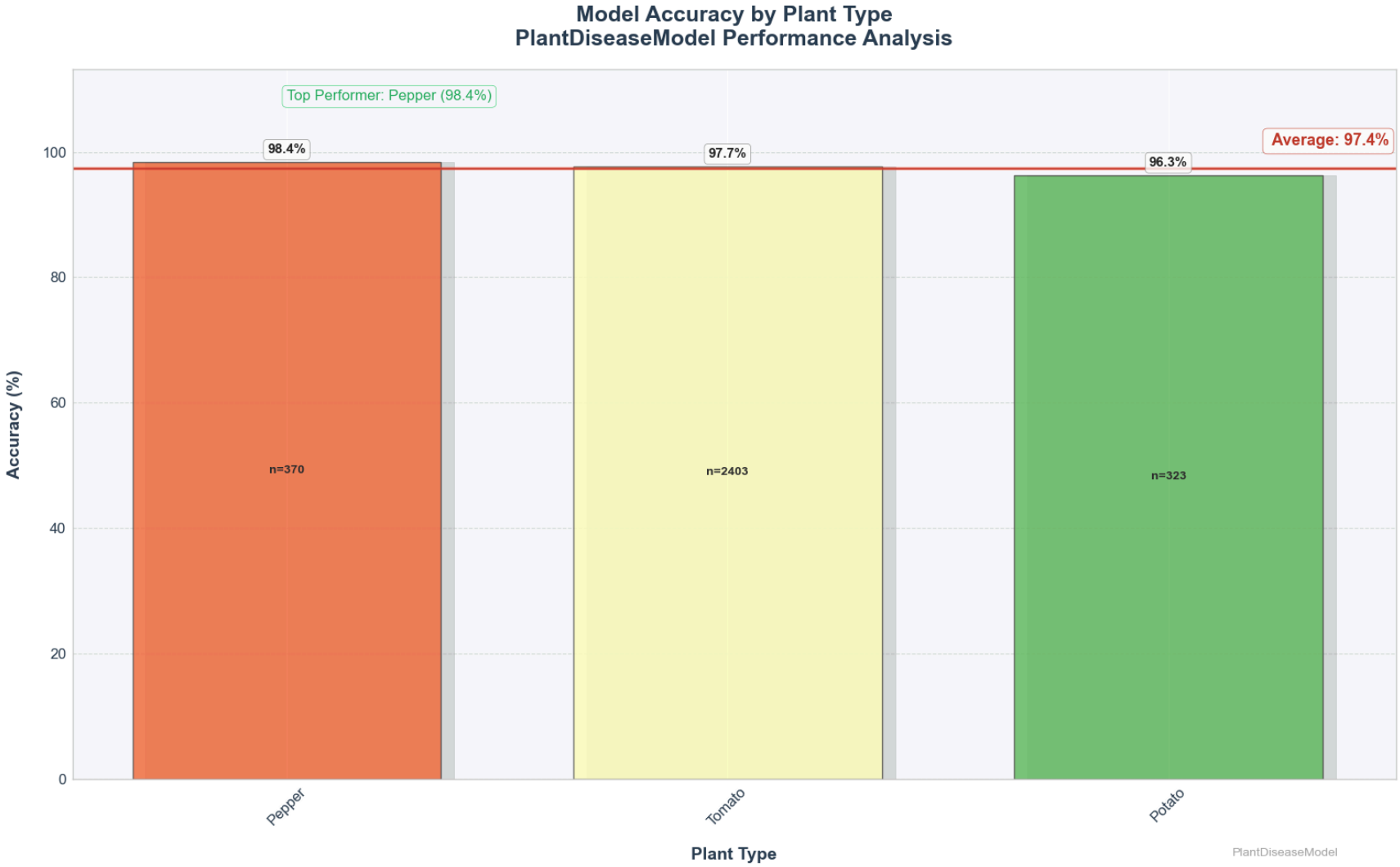
```
            color='#c0392b', fontsize=13, ha='right', va='bottom',
            fontweight='bold',
            bbox=dict(boxstyle="round,pad=0.3", fc='white', ec="#c0392b", alpha=0.8))

    # Configure axes and labels with enhanced styling
    ax.set_title(f'Model Accuracy by Plant Type\n{model.__class__.__name__} Performance Analysis',
                 fontsize=18, pad=20, fontweight='bold', color='#2c3e50')

    ax.set_xlabel('Plant Type', fontsize=14, labelpad=15, fontweight='bold', color='#2c3e50')
    ax.set_ylabel('Accuracy (%)', fontsize=14, labelpad=15, fontweight='bold', color='#2c3e50')

    # Add a subtle box around the plot
    for spine in ax.spines.values():
        spine.set_visible(True)
        spine.set_color('#cccccc')
        spine.set_linewidth(1)

    # Enhanced tick parameters
    ax.tick_params(axis='x', rotation=45, labelsize=12, pad=5, colors='#2c3e50')
    ax.tick_params(axis='y', labelsize=12, pad=5, colors='#2c3e50')
    ax.set_ylim(0, max(accuracies) * 1.15)

    # Add customized grid
    ax.yaxis.grid(True, linestyle='--', alpha=0.4, color='#95a5a6')
    ax.set_axisbelow(True)

    # Add a subtle top performance indicator
    top_performer = plants_list[0]
    top_accuracy = accuracies[0]
    ax.text(0, max(accuracies) * 1.1,
            f"Top Performer: {top_performer} ({top_accuracy:.1f}%)",
            fontsize=12, ha='left', color='#27ae60',
            bbox=dict(boxstyle="round,pad=0.3", fc='#f8f9fa', ec="#2ecc71", alpha=0.8))

    # Add watermark or model info
    fig.text(0.95, 0.02, f"{model.__class__.__name__}",
             fontsize=10, color='gray', ha='right', va='bottom', alpha=0.7)

    plt.tight_layout()
    plt.show()

    return plant_accuracy
```

In [14]:
```
import torch
import pickle
```

```python
import os
import random
from PIL import Image

# Load model and necessary components
model_path = "best_model.pth"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load class names and create model
with open('class_names.json', 'r') as f:
    class_names = json.load(f)
num_classes = len(class_names)

model = PlantDiseaseModel(num_classes=num_classes)
model.load_state_dict(torch.load(model_path))
model.to(device)

# Load label encoder and transform
with open('label_encoder.pkl', 'rb') as f:
    label_encoder = pickle.load(f)

with open('inference_transform.pkl', 'rb') as f:
    transform = pickle.load(f)

print(f"✅ Model loaded with {num_classes} classes")
print(f"✅ Using device: {device}")

# Cell 3: Evaluate Model Performance by Plant Type
data_dir = r"C:\Users\Kanchan Yadav\Downloads\Plant Village Dataset\PlantVillage"
batch_size = 32

_, _, test_loader, _ = prepare_data(
    data_dir,
    image_size=(256, 256),
    batch_size=batch_size,
    test_size=0.3,
    valid_ratio=0.5
)

print("\n📊 Evaluating model performance by plant type...")
plant_accuracy = evaluate_by_plant_type(model, test_loader, label_encoder, device)
```

✅ Model loaded with 15 classes
✅ Using device: cpu
[INFO] Loading images...
[INFO] Image loading completed
Total images: 20638
Training samples: 14446
Validation samples: 3096
Test samples: 3096

📊 Evaluating model performance by plant type...



In [15]:

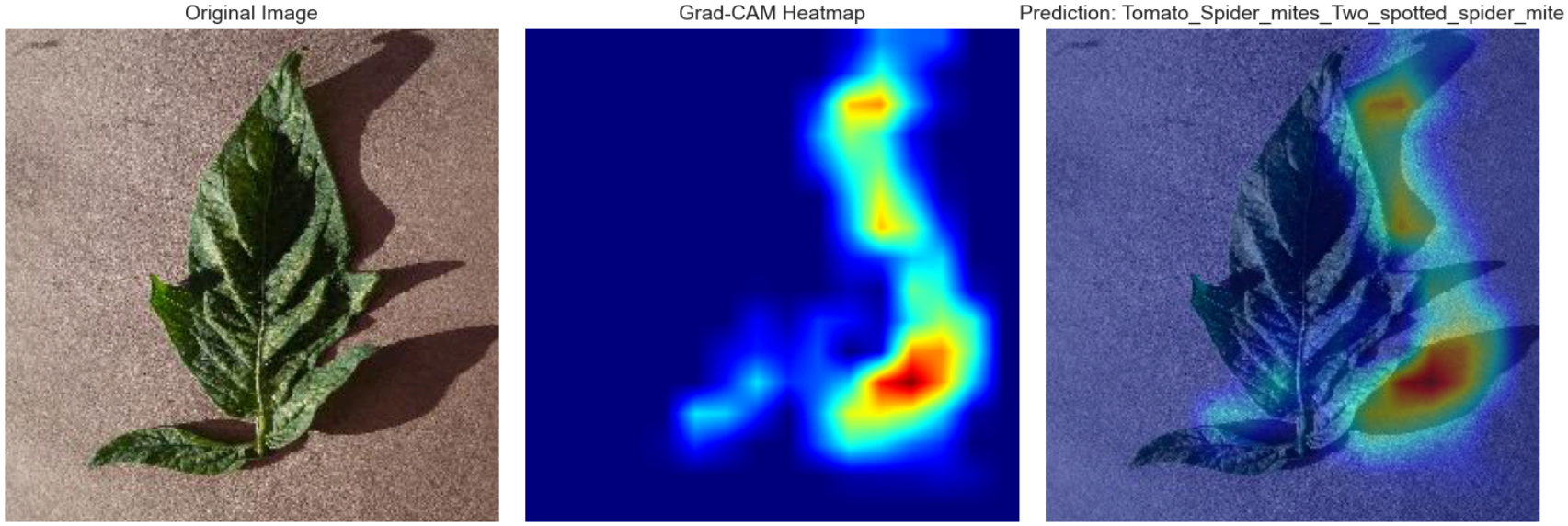
```python
def apply_gradcam(model, img_path, transform, label_encoder, device, layer_name='conv_block5'):
    try:
        import cv2
    except ImportError:
        print("OpenCV (cv2) is required for Grad-CAM visualization. Please install it with: !pip install opencv-python"
        return

    model.eval()

    # Hook for the selected layer
    activations = None
    gradients = None

    def forward_hook(module, input, output):
        nonlocal activations
        activations = output.detach()

    def backward_hook(module, grad_input, grad_output):
        nonlocal gradients
        gradients = grad_output[0].detach()

    # Register hooks
    if layer_name == 'conv_block5':
        target_layer = model.conv_block5[0]  # First conv layer of the last block
    elif layer_name == 'conv_block4':
        target_layer = model.conv_block4[0]
    else:
        target_layer = model.conv_block3[0]

    forward_handle = target_layer.register_forward_hook(forward_hook)
    backward_handle = target_layer.register_backward_hook(backward_hook)

    try:
        # Load and preprocess image
        img = Image.open(img_path).convert('RGB')
        input_tensor = transform(img).unsqueeze(0).to(device)

        # Forward pass
        output = model(input_tensor)
        pred_idx = output.argmax(dim=1).item()
        pred_class = label_encoder.inverse_transform([pred_idx])[0]

        # Backward pass for the predicted class
        model.zero_grad()
```

```
        output[:, pred_idx].backward()

        # Generate Grad-CAM
        if activations is not None and gradients is not None:
            # Pool gradients across the channels
            pooled_gradients = torch.mean(gradients, dim=[0, 2, 3])

            # Weight activation maps by gradients
            for i in range(activations.size(1)):
                activations[:, i, :, :] *= pooled_gradients[i]

            # Average over channels
            heatmap = torch.mean(activations, dim=1).squeeze().cpu().numpy()

            # ReLU on heatmap
            heatmap = np.maximum(heatmap, 0)

            # Normalize heatmap
            if np.max(heatmap) > 0:
                heatmap = heatmap / np.max(heatmap)

            # Resize heatmap to original image size
            original_img = np.array(img)
            heatmap = cv2.resize(heatmap, (original_img.shape[1], original_img.shape[0]))

            # Apply colormap to heatmap
            heatmap = np.uint8(255 * heatmap)
            heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

            # Superimpose heatmap on original image
            superimposed = cv2.addWeighted(original_img, 0.6, heatmap, 0.4, 0)

            # Create figure with original and heatmap
            plt.figure(figsize=(15, 5))

            # Plot original image
            plt.subplot(1, 3, 1)
            plt.imshow(original_img)
            plt.title("Original Image", fontsize=14)
            plt.axis('off')

            # Plot heatmap
            plt.subplot(1, 3, 2)
            plt.imshow(cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB))
            plt.title("Grad-CAM Heatmap", fontsize=14)
```

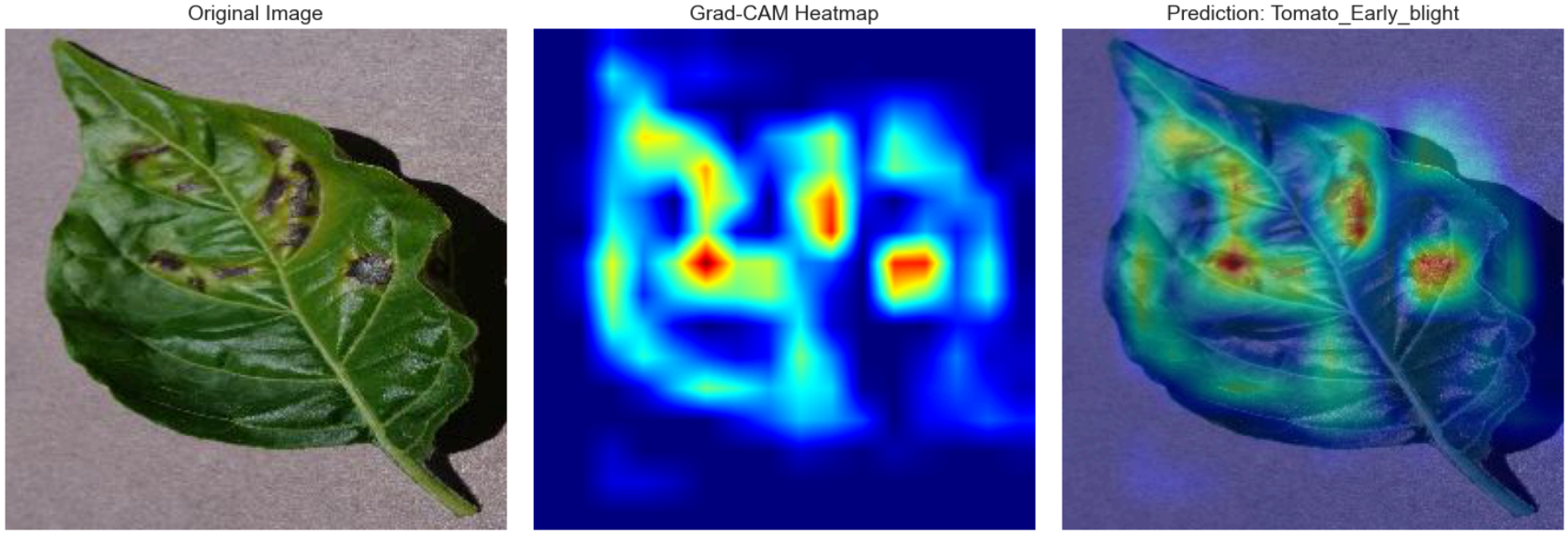
```
            plt.axis('off')

            # Plot superimposed
            plt.subplot(1, 3, 3)
            plt.imshow(cv2.cvtColor(superimposed, cv2.COLOR_BGR2RGB))
            plt.title(f"Prediction: {pred_class}", fontsize=14)
            plt.axis('off')

            plt.tight_layout()
            plt.show()
        else:
            print("Could not generate activations or gradients")
    finally:
        # Always remove hooks to prevent memory leaks
        forward_handle.remove()
        backward_handle.remove()

# Cell 5: Apply Grad-CAM to sample images
def generate_gradcam_visualizations(num_samples=2):
    print("\n🔍 Generating Grad-CAM visualizations...")
    sample_images = []

    for disease_folder in os.listdir(data_dir):
        disease_folder_path = os.path.join(data_dir, disease_folder)
        if not os.path.isdir(disease_folder_path):
            continue

        img_files = [f for f in os.listdir(disease_folder_path)
                     if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        if img_files:
            selected_img = os.path.join(disease_folder_path, random.choice(img_files))
            sample_images.append((selected_img, disease_folder))

    # Apply GradCAM to a couple of sample images
    if sample_images:
        samples_to_visualize = random.sample(sample_images, min(num_samples, len(sample_images)))
        for i, (img_path, true_label) in enumerate(samples_to_visualize):
            print(f"\nVisualizing sample {i+1} - {true_label}...")
            apply_gradcam(model, img_path, transform, label_encoder, device)
    else:
        print("No sample images found.")

# Generate visualizations for 2 random samples
generate_gradcam_visualizations(num_samples=2)
```

🔍 Generating Grad-CAM visualizations...

Visualizing sample 1 - Tomato_Spider_mites_Two_spotted_spider_mite...



Visualizing sample 2 - Pepper__bell___Bacterial_spot...



Inference

In [16]:

```python
def interactive_disease_diagnosis(model_path, label_encoder_path, transform_path, sample_images_dir):
    """
    Create an interactive display showing disease diagnosis and treatment recommendations

    Arguments:
    model_path -- path to the trained model
    label_encoder_path -- path to the saved label encoder
    transform_path -- path to the saved transform
    sample_images_dir -- directory containing sample images
    """
    import json
    import os
    import pickle
    import random
    import torch
    import matplotlib.pyplot as plt
    import numpy as np
    from PIL import Image
    from matplotlib.gridspec import GridSpec

    # Load model
    with open('class_names.json', 'r') as f:
        class_names = json.load(f)
    num_classes = len(class_names)

    model = PlantDiseaseModel(num_classes=num_classes)
    model.load_state_dict(torch.load(model_path))
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()  # Set model to evaluation mode

    # Load label encoder and transform
    with open(label_encoder_path, 'rb') as f:
        label_encoder = pickle.load(f)

    with open(transform_path, 'rb') as f:
        transform = pickle.load(f)

    # Treatment recommendations for common plant diseases
    treatment_recommendations = {
        "Tomato_Bacterial_spot": [
            "Remove and destroy infected plants",
            "Rotate crops (avoid planting tomatoes in the same location for 2-3 years)",
            "Use copper-based fungicides",
```

```
            "Ensure proper spacing between plants for good air circulation"
        ],
        "Tomato_Early_blight": [
            "Remove infected leaves immediately",
            "Apply fungicides containing chlorothalonil or copper",
            "Mulch around the base of plants",
            "Water at soil level rather than on foliage"
        ],
        "Tomato_Late_blight": [
            "Remove and destroy infected plants",
            "Apply fungicides proactively before symptoms appear",
            "Improve air circulation around plants",
            "Avoid overhead irrigation"
        ],
        "Tomato_Leaf_Mold": [
            "Increase spacing between plants to improve air circulation",
            "Apply fungicides containing chlorothalonil or copper",
            "Remove infected leaves",
            "Keep foliage dry by watering at the base"
        ],
        "Tomato_Septoria_leaf_spot": [
            "Remove infected leaves",
            "Apply fungicides containing chlorothalonil or copper",
            "Rotate crops",
            "Mulch around plants to prevent spores splashing from soil"
        ],
        "Tomato_Spider_mites_Two_spotted_spider_mite": [
            "Spray plants with strong streams of water to dislodge mites",
            "Apply insecticidal soap or neem oil",
            "Introduce predatory mites",
            "Increase humidity around plants"
        ],
        "Tomato__Target_Spot": [
            "Remove infected plant debris",
            "Apply fungicides",
            "Improve air circulation",
            "Avoid overhead watering"
        ],
        "Tomato__Tomato_YellowLeaf__Curl_Virus": [
            "No cure available - remove and destroy infected plants",
            "Control whitefly populations (vectors)",
            "Use reflective mulches to repel whiteflies",
            "Plant resistant varieties"
        ],
        "Tomato__Tomato_mosaic_virus": [
```

```
            "No cure available - remove and destroy infected plants",
            "Wash hands and tools after handling infected plants",
            "Control aphid populations (vectors)",
            "Plant resistant varieties"
        ],
        "Potato___Early_blight": [
            "Remove infected leaves",
            "Apply fungicides containing chlorothalonil",
            "Maintain good soil fertility",
            "Ensure proper hilling to protect tubers"
        ],
        "Potato___Late_blight": [
            "Apply fungicides preventatively",
            "Remove volunteer potato plants",
            "Harvest tubers during dry weather",
            "Ensure proper storage conditions for harvested potatoes"
        ],
        "Pepper__bell___Bacterial_spot": [
            "Remove infected plant debris",
            "Rotate crops",
            "Apply copper-based sprays",
            "Use disease-free seeds"
        ]
    }

    # Default recommendation for healthy plants
    default_healthy_practices = [
        "Maintain proper watering schedule",
        "Ensure adequate sunlight",
        "Fertilize appropriately for plant type",
        "Monitor regularly for signs of disease"
    ]

    # Helper function for prediction
    def predict_image(model, img_path, transform, device, label_encoder):
        # Load and preprocess image
        image = Image.open(img_path).convert('RGB')
        image_tensor = transform(image).unsqueeze(0).to(device)

        # Get prediction
        with torch.no_grad():
            outputs = model(image_tensor)
            probabilities = torch.nn.functional.softmax(outputs, dim=1)[0]

        # Get top prediction
```

```
        top_prob, top_class = torch.max(probabilities, 0)
        predicted_class = label_encoder.inverse_transform([top_class.item()])[0]
        confidence = float(top_prob.item()) * 100

        return predicted_class, confidence, probabilities.cpu().numpy()

    # Find test images
    test_images = []
    for disease_folder in os.listdir(sample_images_dir):
        disease_folder_path = os.path.join(sample_images_dir, disease_folder)
        if not os.path.isdir(disease_folder_path):
            continue

        img_files = [f for f in os.listdir(disease_folder_path)
                     if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        if img_files:
            # Take a random image from this disease category
            selected_img = os.path.join(disease_folder_path, random.choice(img_files))
            test_images.append((selected_img, disease_folder))

    # Randomly select images for demonstration
    num_images = min(6, len(test_images))  # Increased from 4 to 6 images
    selected_test_images = random.sample(test_images, num_images) if len(test_images) > num_images else test_images

    # Set up the figure with GridSpec for better layout control
    plt.rcParams.update({'font.size': 12})
    fig = plt.figure(figsize=(24, 20))
    gs = GridSpec(3, 2, figure=fig)

    # Add a stylish title with improved formatting
    fig.suptitle("🌿 Plant Disease Diagnosis & Treatment Recommendations",
                 fontsize=28, fontweight='bold', y=0.98,
                 bbox=dict(facecolor='#e8f4ea', edgecolor='green', boxstyle='round,pad=0.5'))

    # Add a subtitle with system info
    fig.text(0.5, 0.94, f"Running on: {device} | Model: PlantDiseaseModel | Classes: {num_classes}",
             ha='center', fontsize=14, fontstyle='italic', color='#555555')

    # Color palette for recommendations
    treatment_colors = {
        'healthy': '#e8f4ea',  # Light green
        'disease': '#f9e8ea'   # Light red
    }

    # Add a legend for confidence
```

```
    cmap = plt.cm.RdYlGn
    confidence_gradient = np.linspace(0, 1, 100)
    confidence_bar = np.vstack((confidence_gradient, confidence_gradient))

    # Add confidence colorbar at the bottom
    cax = fig.add_axes([0.3, 0.05, 0.4, 0.02])
    cb = plt.colorbar(plt.imshow(confidence_bar, cmap=cmap), cax=cax, orientation='horizontal')
    cb.set_label('Prediction Confidence', fontsize=14)
    cb.set_ticks([0, 0.25, 0.5, 0.75, 1])
    cb.set_ticklabels(['0%', '25%', '50%', '75%', '100%'])

    # Create grid layout based on number of images
    rows = 2 if num_images <= 4 else 3
    cols = 2

    # Process each selected image
    all_predictions = []  # Store prediction results
    for i, (img_path, true_label) in enumerate(selected_test_images):
        if i >= rows * cols:
            break

        # Calculate grid position
        row = i // cols
        col = i % cols

        # Make prediction
        predicted_class, confidence, probabilities = predict_image(
            model, img_path, transform, device, label_encoder)

        # Store prediction result
        all_predictions.append((os.path.basename(img_path), true_label, predicted_class, confidence))

        # Create subplot with better positioning
        ax = fig.add_subplot(gs[row, col])

        # Load and display image
        img = Image.open(img_path).convert('RGB')
        ax.imshow(img)
        ax.axis('off')

        # Determine if prediction is correct
        is_correct = predicted_class == true_label

        # Format disease name for display
        display_pred = predicted_class.replace('_', ' ')
```

```
        display_true = true_label.replace('_', ' ')

        # Apply color based on confidence
        title_color = cmap(confidence/100)

        # Create a styled title box
        title_box = dict(
            boxstyle='round,pad=0.5',
            facecolor=cmap(confidence/100),
            alpha=0.8,
            edgecolor='gray'
        )

        # Set title with prediction info in a box
        ax.set_title(f"Prediction: {display_pred}\nConfidence: {confidence:.1f}%",
                     fontsize=16, fontweight='bold', color='white',
                     bbox=title_box)

        # Add actual label in smaller text
        ax.text(0.5, -0.05, f"Actual: {display_true}",
                transform=ax.transAxes, ha='center', fontsize=14,
                color='black' if is_correct else 'darkred',
                fontweight='bold' if not is_correct else 'normal')

        # Get treatment recommendations
        is_healthy = "healthy" in predicted_class.lower()
        if is_healthy:
            recommendations = default_healthy_practices
            recommendation_title = "Healthy Plant Care:"
            box_color = treatment_colors['healthy']
        else:
            recommendations = treatment_recommendations.get(
                predicted_class, ["No specific recommendations available"])
            recommendation_title = "Treatment Recommendations:"
            box_color = treatment_colors['disease']

        # Create a styled box for recommendations
        rec_box_props = dict(
            boxstyle='round,pad=0.6',
            facecolor=box_color,
            alpha=0.85,
            edgecolor='gray'
        )

        # Add disease severity indicator
```

```
        if not is_healthy:
            severity = "High" if confidence > 85 else "Medium" if confidence > 65 else "Low"
            severity_color = "red" if severity == "High" else "orange" if severity == "Medium" else "green"
            severity_text = f"Severity: {severity}"
        else:
            severity_text = "Status: Healthy"
            severity_color = "green"

        # Build recommendation text with formatting
        recommendation_text = f"{recommendation_title}\n"
        for rec in recommendations:
            recommendation_text += f"• {rec}\n"

        recommendation_text += f"\n{severity_text}"

        # Place recommendations in a better position
        plt.figtext(0.5 + col * 0.5 - 0.48,
                    0.9 - row * 0.33 - 0.13,
                    recommendation_text,
                    fontsize=14,
                    color='black',
                    bbox=rec_box_props,
                    verticalalignment='top')

        # Add severity indicator dot
        plt.figtext(0.5 + col * 0.5 - 0.15,
                    0.9 - row * 0.33 - 0.33,
                    "●",
                    fontsize=30,
                    color=severity_color,
                    ha='right')

        # Add top 3 probable diseases as small text (if not healthy)
        if not is_healthy and len(class_names) > 1:
            # Get top 3 predictions
            top_indices = np.argsort(probabilities)[-3:][::-1]
            top_classes = [label_encoder.inverse_transform([idx])[0].replace('_', ' ') for idx in top_indices]
            top_probs = [probabilities[idx] * 100 for idx in top_indices]

            # Format alternatives text
            alt_text = "Alternative diagnoses:\n"
            for j, (cls, prob) in enumerate(zip(top_classes, top_probs)):
                if j == 0:  # Skip the top prediction (already shown)
                    continue
                alt_text += f"{cls}: {prob:.1f}%\n"
```

```
            # Add alternatives in small text
            plt.figtext(0.5 + col * 0.5 - 0.48,
                        0.9 - row * 0.33 - 0.3,
                        alt_text,
                        fontsize=10,
                        color='#555555',
                        verticalalignment='top')

    plt.tight_layout(rect=[0, 0.08, 1, 0.93])

    # Add a footer with additional information
    footer_text = (
        "Note: This is an AI-assisted diagnosis tool and should be used as a guide only. "
        "For conclusive diagnosis, consult with a professional plant pathologist."
    )
    fig.text(0.5, 0.01, footer_text, ha='center', fontsize=12, fontstyle='italic')

    plt.show()

    # Return results summary for further use if needed
    results = {
        "images_analyzed": len(selected_test_images),
        "predictions": all_predictions,
        "model_device": str(device)
    }

    return results

print("\n🌿 Running interactive disease diagnosis with treatment recommendations...")
results = interactive_disease_diagnosis(
    model_path=model_path,
    label_encoder_path="label_encoder.pkl",
    transform_path="inference_transform.pkl",
    sample_images_dir=data_dir
)

# Print a summary of results
print(f"\nAnalysis complete! Examined {results['images_analyzed']} plant images")
print(f"Model running on: {results['model_device']}")
print("\nSummary of diagnoses:")
for img, true, pred, conf in results['predictions']:
    match = "✓" if true.replace("_", " ") == pred.replace("_", " ") else "✗"
    print(f"- {img}: {pred.replace('_', ' ')} ({conf:.1f}%) {match}")
```

🌿 Running interactive disease diagnosis with treatment recommendations...

Plant Disease Diagnosis & Treatment Recommendations

Running on: cpu | Model: PlantDiseaseModel | Classes: 15

Prediction: Tomato Tomato mosaic virus
Confidence: 100.0%

Actual: Tomato Tomato mosaic virus

Treatment Recommendations:
• No cure available - remove and destroy infected plants
• Wash hands and tools after handling infected plants
• Control aphid populations (vectors)
• Plant resistant varieties

Severity: High

Prediction: Pepper bell Bacterial spot
Confidence: 98.7%

Actual: Pepper bell Bacterial spot

Treatment Recommendations:
• Remove infected plant debris
• Rotate crops
• Apply copper-based sprays
• Use disease-free seeds

Severity: High

Prediction: Pepper bell healthy
Confidence: 100.0%

Alternative diagnoses:
Tomato Leaf Mold: 0.0%
Tomato Spider mites Two spotted spider mite: 0.0%

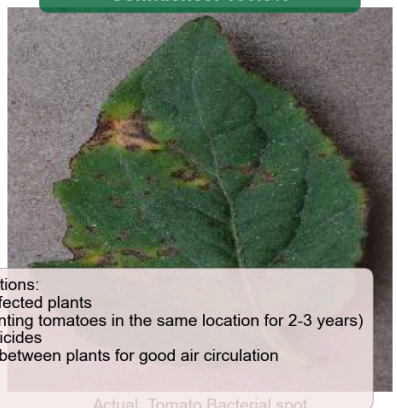
Actual: Pepper bell healthy

Healthy Plant Care:
• Maintain proper watering schedule
• Ensure adequate sunlight
• Fertilize appropriately for plant type
• Monitor regularly for signs of disease

Status: Healthy

Prediction: Tomato Bacterial spot
Confidence: 100.0%

Alternative diagnoses:
Pepper bell healthy: 1.3%
Tomato Septoria leaf spot: 0.0%

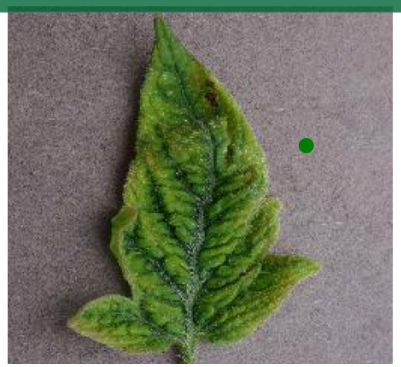


Actual: Tomato Bacterial spot

Treatment Recommendations:
• Remove and destroy infected plants
• Rotate crops (avoid planting tomatoes in the same location for 2-3 years)
• Use copper-based fungicides
• Ensure proper spacing between plants for good air circulation

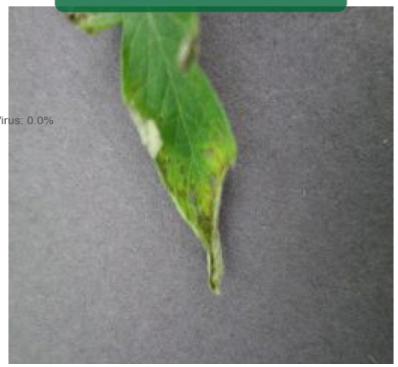
Severity: High

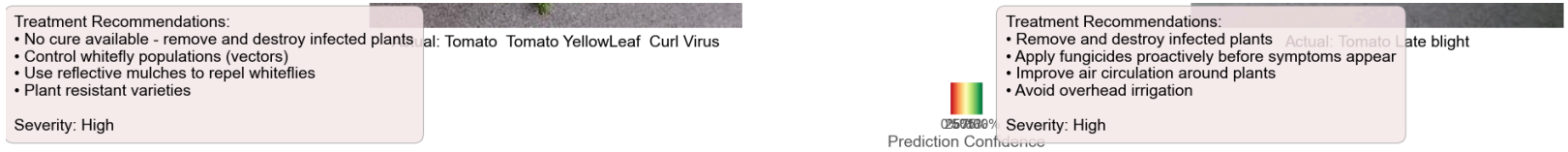
Prediction: Tomato Tomato YellowLeaf Curl Virus
Confidence: 100.0%



Prediction: Tomato Late blight
Confidence: 99.9%

Alternative diagnoses:
Tomato Tomato YellowLeaf Curl Virus: 0.0%
Tomato Septoria leaf spot: 0.0%





Note: This is an AI-assisted diagnosis tool and should be used as a guide only. For conclusive diagnosis, consult with a professional plant pathologist.

```
Analysis complete! Examined 6 plant images
Model running on: cpu

Summary of diagnoses:
- 38748c94-80b0-4a2b-b330-61d2619f5913___PSU_CG 2081.JPG: Tomato  Tomato mosaic virus (100.0%) ✓
- 2700e5b5-295c-4378-b829-0e5989864380___NREC_B.Spot 9091.JPG: Pepper  bell  Bacterial spot (98.7%) ✓
- fb7d12ae-ac5d-4593-8f17-1f248a6ff331___JR_HL 8539.JPG: Pepper  bell  healthy (100.0%) ✓
- 6757fb05-9503-42bc-aa7e-08d083b780aa___GCREC_Bact.Sp 3201.JPG: Tomato Bacterial spot (100.0%) ✓
- 1e891a18-662b-41d6-9bf3-a50eb7f58fbe___YLCV_NREC 0231.JPG: Tomato  Tomato YellowLeaf  Curl Virus (100.0%) ✓
- d8674a0f-a63f-4d10-852a-4913ba977823___GHLB2 Leaf 8568.JPG: Tomato Late blight (99.9%) ✓
```

In [ ]: