PROJECT REPORT

GreenCheck: Leaf Health Classification Using CNN Team

Supervisor:

Pradeep Gopalakrishnan

Professor of Practice, School of Engineering.

Members:

1. Bharath Kumar J (CU22BSC003A)
2. Kousumi Paul (CU22BCA001A)
3. Aditi Balaji (CU22BCA005A)

In partial fulfilment of the course requirements

Semester Project (PRJ 302)

BCA Semester 6

School of Engineering

ABSTRACT

This project focuses on the development of a deep learning-based system for automatic plant leaf disease detection and classification. Leveraging a custom-organized image dataset containing various plant species and their respective diseases, the model aims to accurately identify plant health conditions from leaf images. A custom convolutional neural network (CNN) architecture was trained on over 40,000 high-resolution images categorized by plant type and disease severity. The dataset was structured in a nested folder format to support fine-grained classification. The model achieved high accuracy by incorporating advanced image preprocessing, data augmentation techniques, and a carefully tuned training pipeline. This system offers a promising solution for early disease diagnosis in crops, potentially aiding farmers and agricultural experts in timely interventions, reducing crop loss, and promoting sustainable farming practices.

INTRODUCTION

The world very much depends on agriculture. India is also 80% an agricultural country. It therefore becomes important to take smart and efficient measures to have a good agricultural produce. Machine learning and technology is developing every day which could be made use of so that its more reliable. Traditionally, plant disease diagnosis relies on manual inspection by experts, a process that is often time-consuming, subjective, and limited by human error and availability of resources.

In recent years, advancements in computer vision and deep learning have opened up new possibilities for automating disease detection in plants. By analyzing leaf images, machine learning models especially convolutional neural networks (CNNs) can learn to recognize subtle visual symptoms of various diseases with high accuracy. This project aims to harness the power of deep learning to develop a robust and scalable system for automatic leaf disease detection and classification.

The project utilizes a custom-organized dataset containing images of leaves from multiple plant species, categorized by disease type and condition. The dataset is structured in a nested format, allowing the model to learn both plant-specific and disease-specific features. A custom CNN architecture is designed and trained on over 40,000 labeled images using image preprocessing, data augmentation, and performance tuning techniques.

The ultimate goal of this project is to provide a tool that can assist farmers, researchers, and agronomists in diagnosing plant diseases quickly and accurately, enabling timely intervention and promoting sustainable agriculture.

INITIAL REQUIREMENTS

To successfully implement this model, there are some basic requirements that will help us in the process:

1. **GPU (Graphics Processing Unit):**
   Training deep learning models, especially on large image datasets, requires high computational power. A GPU significantly accelerates the training process by parallelizing matrix operations, which are fundamental to neural network computations.

2. **PlantVillage Dataset:**
   The project uses the PlantVillage dataset, a well-known collection of plant leaf images labeled by species and disease. This dataset provides a diverse and comprehensive set of examples essential for training and evaluating the model effectively.

3. **Understanding of Neural Networks:**
   A foundational understanding of how neural networks work including concepts like layers, activation functions, forward and backward propagation, loss functions, and optimization is necessary for designing, training, and fine-tuning the model.

4. **Familiarity with Python and Its Submodules:**
   Python is the primary programming language used in the project. Familiarity with the following libraries is required:

   PyTorch: For building and training the CNN model.

   OS: For navigating the dataset directory structure.

   NumPy: For efficient numerical operations and array manipulations.

   PIL (Python Imaging Library): For image loading and basic preprocessing.

   Matplotlib: For visualizing data, model performance, and sample predictions.

5. **Data Preprocessing Techniques:**
   Knowledge of image preprocessing methods such as resizing, normalization, and augmentation (rotation, flipping, zooming, etc.) to enhance model generalization and reduce overfitting.

6. **Model Evaluation Metrics:**
   Familiarity with evaluation metrics such as accuracy, confusion matrix, precision, recall, and F1-score to assess the model's performance effectively.

METHODOLOGY

1. Dataset Collection

We first downloaded the Plant village dataset from kaggle.

To begin with we needed to do the basic setup of our environment by installing and importing necessary packages and modules:

```
import torch
import torchvision
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import os
torchvision.utils.make_grid
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

The next step is to divide the dataset into train, test and val groups but was readily available in the downloaded dataset. There are 39 classes in total of various species of leaves. Initially, we chose to work with mainly 5 groups namely, Apple, Tomato, Bell Pepper and Grapes for simplicity.

Each of these groups have further classifications which are the classes.

Apple

Apple Scab

Black Rot

Cedar Apple Rust

Healthy

Tomato

Bacterial Spot

Early Blight

Late Blight

Septoria Leaf Spot

Yellow Leaf Curl Virus

Healthy

Bell Pepper

Bacterial Spot

Healthy

Grapes

Black Rot

Esca (Black Measles)

Leaf Blight

Healthy

2. Data Pre-processing

The next step is to perform pre-processing techniques on the images. Augmentation techniques such as flipping, rotation, brightness adjustment, cropping, etc were applied so that the model learns better. We wanted to understand how the different augmentation techniques work. Here are some results which help us differentiate the process.

Original image:

Augmentation:

```
transform=transforms.Compose([
            transforms.RandomHorizontalFlip(0.9),
            transforms.Resize((255,255)),
            transforms.RandomAffine(90, (0.3, 0.3), (1.0, 2.0)),
            transforms.Resize((155,155)),
            transforms.RandomAffine(90, (0.3, 0.3), (1.0, 2.0)),
            transforms.ToTensor()
        ])
```

Transformed image:

3. **Image-to-Tensor Conversion and Dataset Splitting:**
   Converting images into **tensors** is a foundational step in preparing data for deep learning models. A **tensor** is a multi-dimensional array that serves as the basic data structure in PyTorch (and other deep learning frameworks), allowing efficient numerical computation and automatic differentiation. Images, which are typically represented as pixel matrices, must be converted into tensors to be processed by neural networks.

```
transform = transforms.Compose([
    transforms.RandomRotation(degrees=30),
    transforms.RandomHorizontalFlip(p=0.5),            # 50% chance to flip horizontally
    transforms.ColorJitter(brightness=0.2),
    transforms.Resize((224, 224)),  # Resize to 224x224 pixels
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # Normalize in the range [-1, 1]
])
```

Additionally, to ensure proper model training and evaluation, the dataset is divided into three subsets:

- **Training Set:** Used to train the model and adjust weights.
- **Validation Set:** Used to fine-tune hyperparameters and monitor performance during training, helping to prevent overfitting.
- **Test Set:** Used to evaluate the final model's accuracy and generalization on unseen data.

But our dataset has predefined separation of the data into these three categories. This structured approach to data preparation is critical for building a robust and accurate classification system.

4. Model Development and Training Workflow

To train a reliable deep learning model for leaf disease detection, several critical steps are followed during implementation:

a. **Building the CNN Neural Network Model:**
   Convolutional Neural Networks (CNNs) are specifically designed to process and classify image data. The architecture typically includes layers such as convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected layers for final classification. In this project, a custom CNN model was built using PyTorch, carefully tuned for the specific characteristics of leaf images.

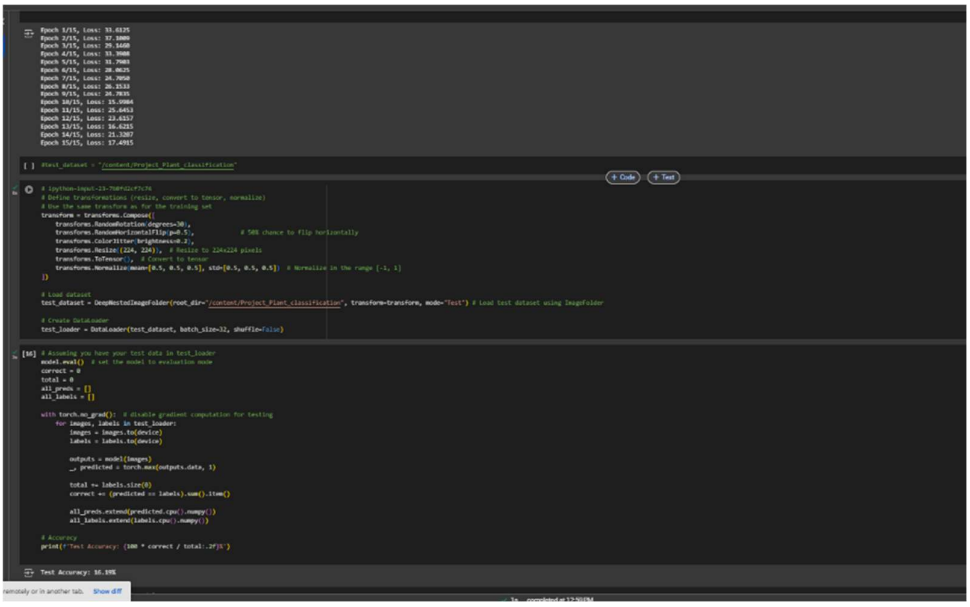
   Initially, the model was designed with a simple architecture consisting of **2 convolutional layers** to evaluate its performance on a small subset of the dataset, specifically targeting **two classes: Apple Scab and Black Rot**. This preliminary

experiment was conducted over **5 epochs** with a **batch size of 32**. The model achieved an impressive **accuracy of 99.01%**, indicating that the CNN was able to effectively learn distinguishing features between these two diseases even with a minimal architecture.

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)   # Input: RGB (3 channels) format (channels, number of kernels, size of kernel, padding)

        #size of kernel is taken randomly, there are specific values for certain extraction
        #print(self.conv1)
        self.pool = nn.MaxPool2d(2, 2) # Takes a 2*2 pooling filter to reduce size.
        #print(self.pool)


        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        #print(self.conv2)
        self.fc1 = nn.Linear(32 * 56 * 56, 128)       # Based on 224x224 resized input and flattens the value to 128 features
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))   # [B, 16, 112, 112]

        x = self.pool(F.relu(self.conv2(x)))   # [B, 32, 56, 56]

        x = x.view(-1, 32 * 56 * 56)           # Flatten
        #print(x)
        x = F.relu(self.fc1(x))

        x = self.fc2(x)
        return x
```

```
Test Accuracy: 99.01%
```

Encouraged by these results, we proceeded to scale the model for full multiclass classification using the **entire dataset**, which includes multiple plant types and disease categories. For this stage, the network was expanded to **4 convolutional layers**, and the **batch size was increased to 64** to accommodate the larger dataset. However, the model achieved a significantly lower **accuracy of 16.19%**, suggesting that it was **underfitting**—failing to learn meaningful representations from the data due to insufficient model complexity.



To address this issue, we concluded that a deeper network is required to extract and learn the more complex and diverse features present in the full dataset. Therefore, the next phase involves implementing a more sophisticated architecture with **8 convolutional layers**, which is expected to improve learning capacity and overall model performance.

This is when we realized that we had to improve our model suspecting that this kind of result was due to overfitting. We created a fresh new class called GreenCheck having 3 layers. This model was taking an input of all the 39 classes.

```
class GreenCheck(nn.Module):
    def __init__(self, num_classes):
        super(GreenCheck, self).__init__()
        # convolution layers
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64*28*28, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # forward pass
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))


        #x = x.view(-1, 64 * 28 * 28)
        x = x.view(x.size(0), -1)

        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

b. Choosing the Right Loss Function and Optimizer:
The choice of loss function and optimizer significantly influences how well the model learns.

- **Loss Function:** Cross-Entropy Loss was used, as it is well-suited for multi-class classification problems. It measures the difference between the predicted class probabilities and the actual labels.
  Cross-entropy loss measures the difference between the predicted probability distribution and the actual class labels. It is commonly used for multi-class classification tasks, penalizing incorrect predictions more heavily as the predicted probability diverges from the true label.
- **Optimizer:** The Adam optimizer was selected due to its adaptive learning rate and efficiency in handling sparse gradients, making it ideal for complex image classification tasks.

  Adam (Adaptive Moment Estimation) is an optimization algorithm that combines the advantages of two other methods—AdaGrad and RMSProp. It adapts the learning rate for each parameter dynamically, making it efficient and well-suited for training deep neural networks with large datasets

c. **Passing the Training and Validation Datasets to the Model:**
The dataset is split into training and validation sets:

- The **training set** is used to teach the model by updating its weights based on the calculated loss.
- The **validation set** is used to monitor the model's performance on unseen data after each epoch. This helps detect overfitting and adjust parameters accordingly.

A custom data loader was implemented using PyTorch's DataLoader class to feed batches of tensors into the model efficiently during both training and validation phases.

5. Passing the Test Dataset to Check Model Accuracy:

Once the model training is complete, it is evaluated on the **test set**, which contains completely unseen images. This final step helps determine the model's ability to generalize and perform in real-world conditions. Accuracy and other metrics such as confusion matrix and precision/recall scores are used to analyze performance.

6. Verifying Using Trial-and-Error Methods:

Model performance is rarely optimal on the first try. Throughout the project, trial-and-error methods were employed:

- Experimenting with different CNN architectures (number of layers, filter sizes, etc.).
- Trying different batch sizes, learning rates, and epochs.
- Testing various data augmentation techniques to improve generalization.
- Fine-tuning dropout rates and normalization layers to reduce overfitting.

This iterative approach led to progressive improvements in accuracy and robustness of the model.

CODE

```
# Import necessary libraries
import torch
import torch.nn as nn
from torch.optim import Adam

from torch.utils.data import DataLoader, Dataset, Subset
from torchvision import datasets, transforms
from torchsummary import summary
# Basic Libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from collections import Counter

torch.manual_seed(42)
np.random.seed(42)
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
print(f'Using device: {device} ')
#Data Transformations
train_transforms = transforms.Compose([
            transforms.Resize((224,224)),
            transforms.RandomHorizontalFlip(0.9),
            transforms.RandomVerticalFlip(),
            transforms.RandomRotation(90),
            transforms.RandomAffine(90, (0.3, 0.3), (1.0, 2.0)),
            transforms.ToTensor(),
```

```
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])

        ])
val_test_transforms = transforms.Compose([
                        transforms.Resize((224,224)),
                        transforms.ToTensor(),
                       transforms.Normalize(mean=[0.485, 0.456,
                      0.406], std=[0.229, 0.224, 0.225])
])
#Setting path
from google.colab import drive
drive.mount('/content/drive')
# Set paths to the inner ZIP files
base_path = "/content/drive/MyDrive"
inner_zip_1 =
f"{base_path}/Plant_leaf_diseases_dataset_without_augmentation.zip"
# Unzip them to separate folders
!unzip -q "{inner_zip_1}" -d "{base_path}/without_augmentation"
print("Both inner ZIP files extracted!")


data_dir =
'/content/drive/MyDrive/without_augmentation/Plant_leave_diseases_da
taset_without_augmentation'
dataset = datasets.ImageFolder(root=data_dir,
transform=val_test_transforms)
# Lets get class names and number of class
class_names = dataset.classes
num_classes = len(class_names)
print(f'Number of classes: {num_classes}')
print(f'Classes: {class_names}')
# lets split Plant Village in Train, Val and Test
# Train -> 70%   Val -> 15%  Test -> 15%
indices = list(range(len(dataset)))
labels = [dataset.targets[i] for i in range(len(dataset))]
```

```
# 1st Split  Train(70%) + (Val+Test) (30%)

train_idx, temp_idx, train_labels, temp_labels =
train_test_split(indices, labels, test_size=0.2, stratify=labels,
random_state=42)


# 2nd Split  Val (15%) + Test (15%) from the 30%

val_idx, test_idx, val_labels, test_labels =
train_test_split(temp_idx, temp_labels, test_size=0.6,
stratify=temp_labels, random_state=42)


# Create subset of Village Datasets

train_dataset = Subset(dataset, train_idx)

val_dataset = Subset(dataset, val_idx)

test_dataset = Subset(dataset, test_idx)


# Apply train transforms to training dataset

train_dataset.dataset.transform = train_transforms


# DataLoaders

batch_size = 32

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)

test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

# N Visualize Class Distribution

class_counts = Counter(labels)

class_counts_df = pd.DataFrame.from_dict(class_counts,
orient='index', columns=['count'])

class_counts_df['class'] = [class_names[i] for i in
class_counts_df.index]

class_counts_df = class_counts_df.sort_values('count',
ascending=False)


plt.figure(figsize=(12,6))
```

```
sns.barplot(data=class_counts_df, x='class', y='count')
plt.title("Class Distribution in Plant Village Dataset")
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
plt.savefig('plnt_vlg_class_distribution.png')
plt.close()
class GreenCheck(nn.Module):
    def __init__(self, num_classes):
        super(GreenCheck, self).__init__()
        # convolution layers
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64*28*28, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # forward pass
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))

        #x = x.view(-1, 64 * 28 * 28)
        x = x.view(x.size(0), -1)

        x = self.relu(self.fc1(x))
        x = self.dropout(x)
```

```
        x = self.fc2(x)
        return x
model = GreenCheck(num_classes=num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=0.001)

#Training
num_epochs = 10
train_losses, val_losses = [], []
train_accurates, val_accurates = [], []

for epoch in range(num_epochs):
    # Here we train the model
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    train_loss = running_loss / len(train_loader)
    train_acc = 100 * correct / total
    train_losses.append(train_loss)
    train_accurates.append(train_acc)
```

```
    # Performing Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    val_loss = val_loss / len(val_loader)
    val_acc = 100 * correct / total
    val_losses.append(val_loss)
    val_accurates.append(val_acc)

    print(f'Epoch [{epoch+1}/{num_epochs}]')
    print(f'Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.2f}%')
    print(f'Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%')



# Plot Training Metrics
epochs = range(1, num_epochs+1)
plt.figure(figsize=(12,5))

# Loss Plot
plt.subplot(1,2,1)
sns.lineplot(x=epochs, y=train_losses, label='Train Loss')
sns.lineplot(x=epochs, y=val_losses, label='Val Loss')
```

```
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')

# Accuracy Plot
plt.subplot(1, 2, 2)
sns.lineplot(x=epochs, y=train_accurates, label='Train Acc')
sns.lineplot(x=epochs, y=val_accurates, label='Val Acc')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')

plt.tight_layout()
plt.show()
plt.savefig('training_metrics.png')
plt.close()
#  Test Evaluation
model.eval()
test_correct = 0
test_total = 0
all_preds = []
all_labels = []
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
```

```
test_acc = 100 * test_correct / test_total

print(f'Test Accuracy: {test_acc:.2f}%')


# Confusion Matrix

cm = confusion_matrix(all_labels, all_preds)

plt.figure(figsize=(12, 10))

sns.heatmap(cm, annot=True, fmt='d', xticklabels=class_names,
yticklabels=class_names, cmap='Blues')

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.xticks(rotation=90)

plt.yticks(rotation=0)

plt.tight_layout()

plt.show()

plt.savefig('confusion_matrix.png')

plt.close()

# Classification Report

print('\nClassification Report:')

print(classification_report(all_labels, all_preds,
target_names=class_names))
```

RESULTS

The training and validation process result:

```
Epoch [1/10]
Train Loss: 2.1674, Train Acc: 39.36%
Val Loss: 1.3021, Val Acc: 60.66%
Epoch [2/10]
Train Loss: 1.2681, Train Acc: 62.37%
Val Loss: 0.8904, Val Acc: 73.24%
Epoch [3/10]
Train Loss: 0.9778, Train Acc: 70.46%
Val Loss: 0.7043, Val Acc: 77.57%
Epoch [4/10]
Train Loss: 0.8381, Train Acc: 74.63%
Val Loss: 0.5460, Val Acc: 82.91%
Epoch [5/10]
Train Loss: 0.7579, Train Acc: 77.00%
Val Loss: 0.5448, Val Acc: 83.05%
Epoch [6/10]
Train Loss: 0.6982, Train Acc: 78.57%
Val Loss: 0.4833, Val Acc: 84.69%
Epoch [7/10]
Train Loss: 0.6518, Train Acc: 80.15%
Val Loss: 0.4721, Val Acc: 86.27%
Epoch [8/10]
Train Loss: 0.6160, Train Acc: 81.04%
Val Loss: 0.4294, Val Acc: 86.34%
Epoch [9/10]
Train Loss: 0.5937, Train Acc: 81.77%
Val Loss: 0.4048, Val Acc: 87.22%
Epoch [10/10]
Train Loss: 0.5775, Train Acc: 82.33%
Val Loss: 0.4063, Val Acc: 86.90%
```

The final accuracy received was 91%. We focused on the f-1 metric which resulted to 0.92.

Classification Report:

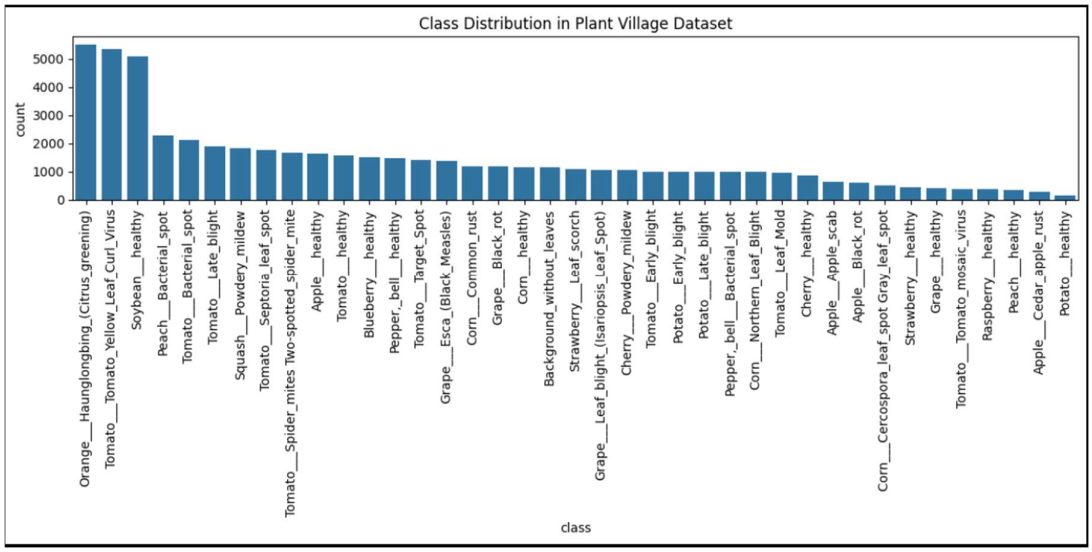
	precision	recall	f1-score	support
Apple___Apple_scab	0.91	0.70	0.79	76
Apple___Black_rot	0.93	0.74	0.83	74
Apple___Cedar_apple_rust	0.94	0.91	0.92	33
Apple___healthy	0.90	0.88	0.89	198
Background_without_leaves	0.98	0.96	0.97	137
Blueberry___healthy	0.96	0.84	0.90	180
Cherry___Powdery_mildew	0.98	0.94	0.96	126
Cherry___healthy	0.91	0.93	0.92	103
Corn___Cercospora_leaf_spot Gray_leaf_spot	0.76	0.81	0.78	62
Corn___Common_rust	0.93	0.96	0.94	143
Corn___Northern_Leaf_Blight	0.92	0.85	0.88	118
Corn___healthy	0.96	0.96	0.96	139
Grape___Black_rot	0.97	0.87	0.91	142
Grape___Esca_(Black_Measles)	0.97	0.95	0.96	167
Grape___Leaf_blight_(Isariopsis_Leaf_Spot)	0.93	0.98	0.95	129
Grape___healthy	0.83	0.76	0.80	51
Orange___Haunglongbing_(Citrus_greening)	0.99	0.97	0.98	661
Peach___Bacterial_spot	0.97	0.91	0.94	276
Peach___healthy	0.95	0.93	0.94	43
Pepper,_bell___Bacterial_spot	0.89	0.92	0.91	119
Pepper,_bell___healthy	0.85	0.88	0.86	178
...				
accuracy			0.92	6656
macro avg	0.90	0.88	0.89	6656
weighted avg	0.92	0.92	0.92	6656

VISUALIZATION

A. Dataset

Visualization plays a crucial role in understanding the performance of a machine learning model. It provides insights into how well the model is learning, identifies potential issues such as overfitting or underfitting, and helps us evaluate the quality of predictions.

The dataset being huge is difficult to maintain and check the number of images for 39 classes. Plotting a bar graph helps us understand the population of the images.

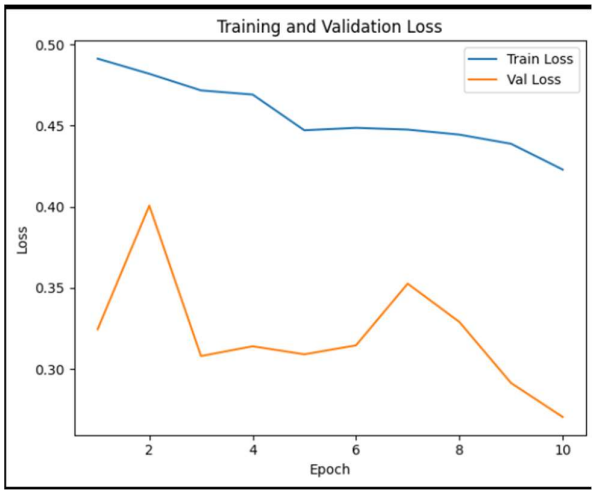


This graph indicates that particularly the 3 classes: ‘Orange__Haunglongbing’, ‘Tomato_Yellow_Leaf_Curl_Virus’ and ‘Soyabean__Healthy’ are very large in number.

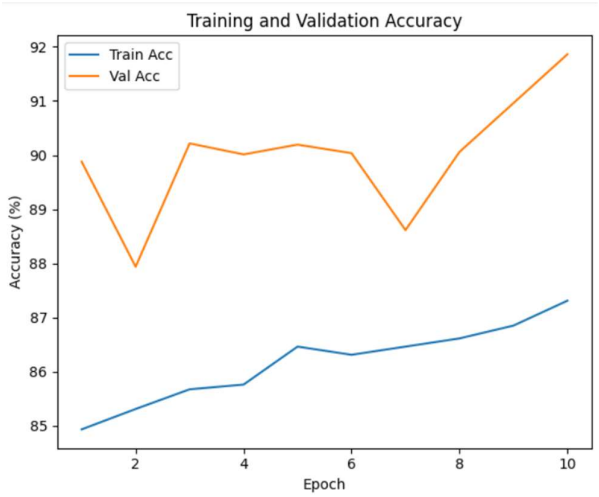
B. Training and validation

The training and validation plots shown above illustrate the loss and accuracy metrics over 10 epochs.

- The graph below shows the **Training and Validation Loss**. We observe a general decline in both training and validation loss as epochs increase, which suggests that the model is learning and generalizing better with time after overfitting in the beginning. The validation loss is constantly fluctuating but resulting less with time indicating good learning of the model.



- The graph below shows the **Training and Validation Accuracy**. We see a steady improvement in both training and validation accuracy. Notably, the **validation accuracy is higher than the training accuracy**, suggesting good learning. It is either a good model or the augmentation or noise in the validation data is easier for the model than the train data.



These trends indicate that the model is improving with training and is not showing signs of severe overfitting or underfitting. Continued training or fine-tuning might further improve performance.

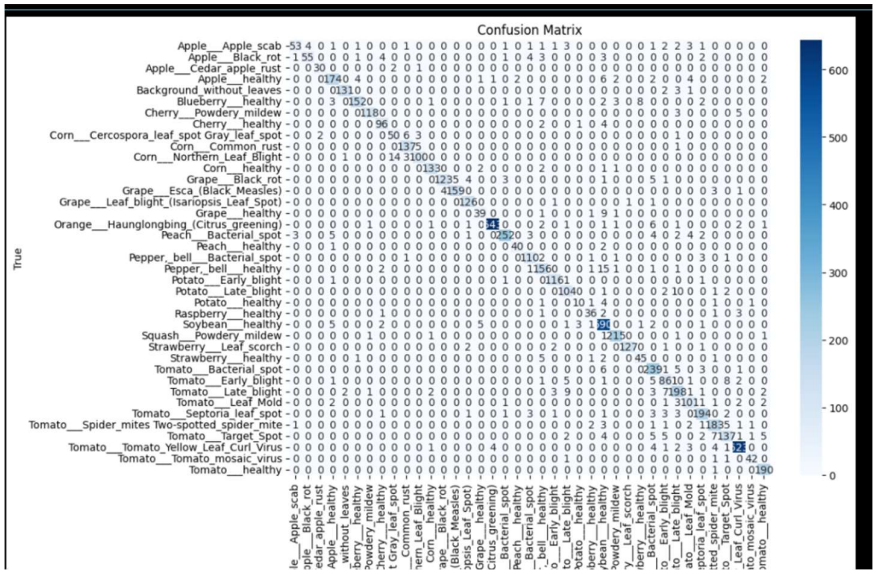
C. Confusion Matrix

To gain a deeper understanding of the model's classification performance, a **confusion matrix** was generated. A confusion matrix is a summary table used to evaluate the performance of a classification model. It compares the actual target values with those predicted by the model, providing detailed insight into the types of errors the model is making.

Each row of the confusion matrix represents the actual class, while each column represents the predicted class. The **diagonal elements** indicate correct predictions (i.e., true positives), whereas the **off-diagonal elements** represent misclassifications.

Key Observations:

- The majority of values are concentrated along the **diagonal**, indicating a high number of correct predictions across almost all classes.
- We can notice that the most correct predictions are from the classes 'Orange__Haunglongbing', 'Tomato_Yellow_Leaf_Curl_Virus' and 'Soyabean__Healthy' which has the highest number of samples indicating more the data, better is the learning compared to other classes.
- We can notice more misclassifications in Tomato and Apple related classes which could be due to the model requiring more learning. The leaves of Tomato and apple are more fine-grained which require more learning for correct classification.
- 'Background_without_leaves', a challenging and visually distinct class, still gets misclassified occasionally, but with relatively few false positives or negatives.



ISSUES FACED

During the course of the project, we encountered several challenges that required time and effort to resolve. Initially, we faced difficulty in connecting **Google Drive to the Google Colab notebook** to access and organize our dataset. It took some time to fully understand the integration process and how to efficiently load data from the drive into the training environment.

As we progressed to building and training the neural network, we encountered another significant limitation: **lack of GPU support on our local systems**. Training deep learning models without GPU acceleration proved to be time-consuming and inefficient, especially with large datasets. This limited our ability to iterate quickly and test different architectures effectively.

To address this, we shifted our focus to working with **GPU-enabled environments** such as Google Colab, which allowed us to significantly reduce training time and improve overall performance. Further, we aim to experiment using GPU support to enhance model accuracy and efficiency. The accuracy can further be improved by fine-tuning the hyperparameters such as number of layers, batch size, learning rate, etc.

CONCLUSION

This project successfully demonstrates the application of deep learning techniques, particularly Convolutional Neural Networks (CNNs), for the detection and classification of plant leaf diseases. Through systematic experimentation, starting with a simple two-class model and progressing toward a complex multiclass architecture, we explored the challenges and intricacies of scaling a model for real-world agricultural data.

The initial results on a small dataset showed promising accuracy, confirming the potential of CNNs in capturing disease-specific patterns. However, when applied to the full dataset, performance dropped due to underfitting, highlighting the need for deeper architecture and more robust training strategies. This iterative development process emphasized the importance of network depth, dataset quality, appropriate loss functions (such as Cross-Entropy), and optimizers like Adam in building an effective classification model.

After doing more changes to the classes, training process and other parameters, we received a good result of 91.84%.

Overall, this project lays a strong foundation for a practical and scalable leaf disease detection system. With further improvements such as hyperparameter tuning, data balancing, and real-time deployment, this solution can be extended to assist farmers, researchers, and agronomists in promoting early diagnosis, reducing crop losses, and contributing to smarter, technology-driven agriculture.

RESOURCES

1. Kaggle for ‘Plant Village’ dataset

2. Different repositories for code insight

3. Articles and Blogs for conceptual understanding