Plant Diseases Detection System Using Deep Learning

Submitted by:

Sibsankar Maity

Roll No.- 23MA60R29

M.Tech Computer Science and Data Processing.

Supervised by:

Prof. Pratima Panigrahi

Project-2 (MA67023)

Indian Institute of Technology Kharagpur, India

Department of Mathematics

Submitted on:

November 27, 2024

Acknowledgement

First of all, I owe my sincere gratitude to IIT KHARAGPUR for allowing me to make this project work. With great pleasure, I offer my heartfelt gratitude & indebtedness to my project Guide **Prof. Pratima Panigrahi**, Professor, Department of Mathematics, IIT Kharagpur, Kharagpur-721302, West Bengal, India who gave me the golden opportunity to do this interesting project work on "**Plant Diseases Detection System Using Deep Learning**."

Sibsankar Maity
Department of Mathematics,
IIT KHARAGPUR,
Kharagpur-721302,

Certification

This is to certify that the project titled **"Plant Diseases Detection System Using Deep Learning"** submitted by **Sibsankar Maity, Roll-No : 23MA60R29**, Department of Mathematics, IIT Kharagpur, has been carried out under my supervision.

This project work is a part of the Master's thesis project for fulfillment of the award of the degree of Master of Technology in Computer Science and Data Processing. It is a record of bonafide project work carried out by him in the department of Mathematics, with my supervision and guidance.

Prof. Pratima Panigrahi
Professor
Department of Mathematics
IIT Kharagpur.

Contents

1 Abstract

Plant disease detection is a critical aspect of modern agriculture, aiming to improve crop health, yield, and food security. Traditional methods often rely on manual inspections by experts, which can be time-consuming, labour-intensive, and prone to human error. With the advent of advanced machine learning techniques and the increasing availability of image data, automated plant disease detection has emerged as a promising solution.

This project focuses on developing a system that utilizes image processing and deep learning to identify and classify plant diseases from images. The system is designed to take a picture of a plant as input, preprocess the image, and apply a trained convolutional neural network (CNN) to detect the presence and type of disease. The model leverages large datasets of labelled plant images to achieve high accuracy in disease classification.

The proposed approach offers several advantages over traditional methods, including faster diagnosis, scalability, and the ability to process images in real time. By integrating this system into mobile applications or agricultural management platforms, farmers can quickly and accurately identify diseases in their crops, enabling timely interventions and reducing the spread of disease.

This work contributes to the broader field of precision agriculture, providing a tool that can help farmers improve crop management, reduce losses, and ultimately enhance food production efficiency. Future research will focus on expanding the system to include a wider range of crops and diseases, improving model accuracy under diverse environmental conditions, and exploring the integration of other data sources, such as environmental sensors, to enhance detection capabilities.

2 Introduction

The prevalence of plant diseases poses a significant threat to agricultural productivity, directly impacting global food security. Effective and timely detection of these diseases is critical for preventing and managing outbreaks, serving as a fundamental aspect of agricultural decision-making processes. Traditionally, plant diseases have been identified through visual inspections carried out by agricultural experts or farmers. However, this method is inherently subjective and, depending on the observer's experience, can be both time-consuming and error-prone. Such limitations often result in misdiagnoses, leading to unnecessary economic losses and potential environmental harm due to the misuse of treatment chemicals.

Historically, traditional image processing techniques have been employed to automate the identification of plant diseases. These methods typically involve extracting features such as color, texture, and shape from images, followed by the application of various classifiers like Support Vector Machines (SVM) and Principal Component Analysis (PCA). While these approaches have achieved notable successes, they are labor-intensive and heavily reliant on manual feature extraction, which is not only subjective but also less effective in complex agricultural environments. Consequently, their practical application remains limited on a larger scale.

In response to these challenges, deep learning, particularly through the use of Convolutional Neural Networks (CNNs), has emerged as a powerful solution. Unlike traditional methods, CNNs can automatically learn to extract relevant features from images and perform disease classification with minimal human intervention, significantly enhancing both efficiency and accuracy. However, the performance of deep learning models largely depends on the availability of large and diverse datasets, which are essential for training robust models. To mitigate this, transfer learning has been increasingly utilized, enabling the adaptation of pre-trained CNNs to plant disease detection tasks using smaller, domain-specific datasets.

Despite significant advancements facilitated by deep learning, gaps remain in the current research landscape. Many studies have yet to fully explore the potential of new visualization techniques and the adaptation of deep learning models for early disease detection, particularly when working with limited samples. This project addresses these research gaps by reviewing the latest advancements in plant leaf disease recognition, utilizing a combination of image processing, hyper-spectral imaging, and deep learning techniques. The aim is to provide a comprehensive overview of the field's current state and outline potential areas for future research, focusing on early detection challenges and the constraints imposed by smaller datasets.

3 Literature Review

Machine learning (ML) and deep learning (DL) have become increasingly popular in plant disease detection, offering innovative ways to identify diseases from plant images. Traditionally, ML methods involved manually extracting features such as color, texture, and shape from images. These features were then used to train models to distinguish between healthy and diseased plants, achieving success in detecting diseases like leaf blotch and powdery mildew. However, these methods often faced challenges in early disease detection and struggled with processing images from complex agricultural environments.

To address these limitations, deep learning techniques, particularly Convolutional Neural Networks (CNNs), have been introduced. Unlike traditional methods, CNNs automatically learn significant features directly from the images, significantly enhancing their ability to detect subtle disease symptoms early in their development. They are particularly effective with high-resolution images, which are crucial for detailed plant health analysis. Despite their advantages, DL models require substantial amounts of labeled data for training and are computationally intensive, which can be prohibitive in resource-limited settings.

Researchers have sought to mitigate these challenges through techniques such as data augmentation, which increases the diversity of training data, and transfer learning, which leverages pre-trained models on new, smaller datasets specific to particular plant diseases. These strategies have improved the robustness and generalizability of DL models, demonstrating enhanced performance across various conditions and plant types.

Despite these advancements, most existing research has concentrated on specific types of plants or diseases, which narrows the scope of applicability of these models. Recent studies have underscored the necessity for models capable of detecting a wide range of diseases across different plant species. Additionally, ensemble methods that combine the predictions of multiple DL models are being investigated to boost diagnostic accuracy and reliability. These methods hold promise for creating more versatile and accurate plant disease detection systems.

While significant progress has been made in applying ML and DL to plant disease detection, there remain substantial challenges, such as the need for more diverse datasets that represent various plant species and environmental conditions, and the high computational demands of training complex DL models. Future research should focus on these areas to develop more effective and universally applicable plant disease detection systems. Moreover, integrating these models with IoT devices for real-time monitoring and diagnosis could revolutionize precision agriculture, providing farmers with timely, accurate data to make informed decisions about disease management.

4 Methodology

4.1 Dataset Description

In our project, we utilized the publicly accessible PlantVillage dataset, originally compiled by Sharada P. Mohanty and colleagues. This dataset comprises approximately 87,000 RGB images of crop leaves, which are categorized into 38 distinct classes reflecting various plant diseases and health conditions. To facilitate the machine learning process, the dataset is systematically divided into three main parts:

- **Training Set:** This set includes 70,295 images, making up about 80% of the total dataset, and is used to train the machine learning models.

- **Validation Set:** Consisting of 17,572 images, this segment accounts for the remaining 20% of the dataset and is used to validate the model's accuracy during training.

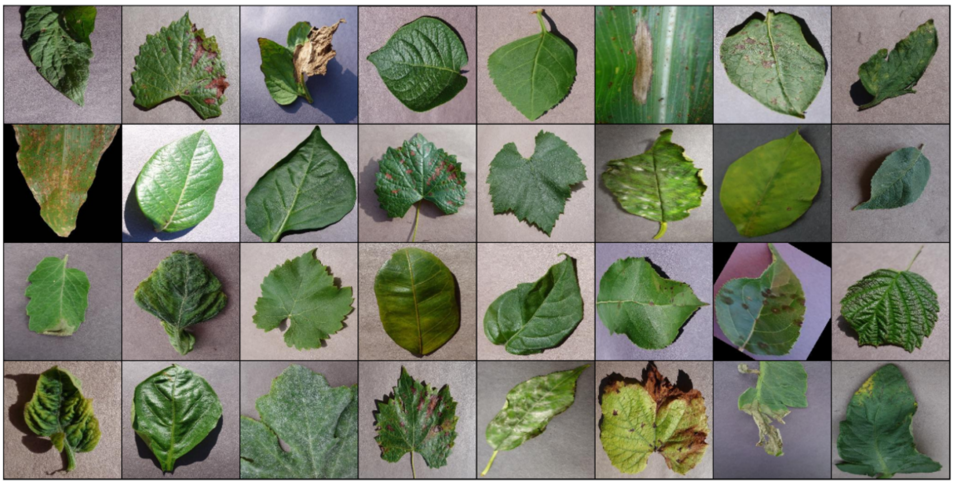


Figure 1: Caption

- **Test Set:** A smaller, distinct group of 33 images is used to test the model's performance on new, unseen data after training is complete.

4.2 Data Preprocessing

1. **Image Augmentation:** To increase the robustness of our model and prevent overfitting, we employed image augmentation techniques on the training dataset. This involved applying a variety of transformations to artificially expand our dataset. Common augmentations included:
   - Rotation: Images were rotated by angles (e.g., ±10°, ±20°) to simulate different orientations of leaves.
   - Flipping: Horizontal and vertical flips were used to further increase the dataset variability.
   - Scaling and Cropping: We applied random scaling and cropping to simulate different distances and angles from which photos might be taken in real-world scenarios.
   - Color Adjustments: Brightness, contrast, and saturation adjustments were made to account for varying lighting conditions that might affect image capture in agricultural settings.
2. **Image Resizing:**All images were resized to a consistent dimension (e.g., 256x256 pixels) to ensure uniformity across the dataset. This step is important because convolutional neural networks (CNNs) require input images of the same size.
3. **Normalization:**To help the CNN model learn more efficiently, we normalized the image pixel values. Typically, pixel values range from 0 to 255; we scaled these to a range of 0 to 1 by dividing each pixel by 255. This normalization process aids in speeding up the learning process and leads to faster convergence.

4. **Data Batching:**During training, the images were batched (e.g., batches of 32 or 64 images) to optimize the training process. Batching allows the model to update its weights incrementally, which is less memory-intensive and can lead to better model generalization.

5. **Label Encoding:**Since the dataset contains 38 different classes, we encoded these classes into a numerical format using one-hot encoding. This transformation converts categorical labels into a binary matrix representation that is suitable for classification tasks with neural networks.

4.3 Model Architecture

Convolutional Neural Networks (CNNs) are deep learning models that extract features from images using convolutional layers, followed by pooling and fully connected layers for tasks like image classification. They excel in capturing spatial hierarchies and patterns, making them ideal for analyzing visual data.

There are two main parts to a CNN architecture

- A convolution tool that separates and identifies the various features of the image for analysis in a process called as Feature Extraction.
- The network of feature extraction consists of many pairs of convolutional or pooling layers.
- A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the features extracted in previous stages.
- This CNN model of feature extraction aims to reduce the number of features present in a dataset. It creates new features which summarises the existing features contained in an original set of features. There are many CNN layers as shown in the basic CNN architecture with diagram.

There are three types of CNN architecture which are the convolutional layers, pooling layers, and fully-connected (FC) layers. When these layers are stacked, a CNN architecture will be formed. In addition to these three layers, there are two more important parameters which are the dropout layer and the activation function which are defined below.

1. **Convolutional Layer:**
   - The convolutional layer is the core building block of a CNN. It applies filters to the input image to create feature maps that capture essential visual features such as edges and textures.
   - A filter or kernel slides over the image, and at each position, a dot product is computed between the filter and the image pixels covered by the filter. This process transforms the original image data into feature maps that represent different aspects of the image.
2. **Pooling Layer:**

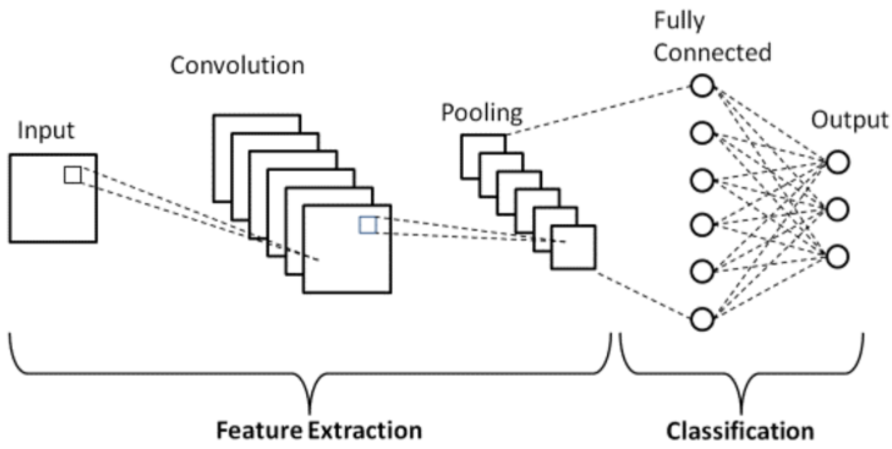


Figure 2: CNN Architecture

- Often following the convolutional layer, the pooling layer serves to reduce the spatial dimensions (width and height) of the input volume for the subsequent layers. This reduction helps decrease the computational load and the number of parameters in the network.
- Common pooling operations include Max Pooling (selecting the maximum value from a set of values in the filter region) and Average Pooling (calculating the average of the values). This summarization helps to make the detection of features invariant to scale and orientation changes.

3. **Fully Connected Layer:**

- After several convolutional and pooling layers, the high-level reasoning in the neural network is performed via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks.
- The input to these layers is typically flattened into a one-dimensional vector of features. These layers are usually placed towards the end of a CNN architecture and are responsible for classifying the input image into various classes based on the combination of features identified by the previous layers.

4. **Dropout Layer:**

- Dropout is a regularization technique used to prevent overfitting in neural networks. By randomly setting a fraction of the neurons to zero during the training process, dropout forces the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Dropout is typically used when training very large neural networks to effectively increase model generalization.

5. **Activation Functions:** Activation functions introduce non-linearity to neural networks, enabling them to learn and model complex data patterns. These functions are applied element-wise to the output from the convolutional layers. Some of the most commonly used activation functions include the ReLU, Sigmoid, and TanH. Each plays a crucial role in the neural network's ability to process and make predictions from the input data.

   (a) **Linear Function:**

   - The linear activation function is straightforward; it produces an output that is a direct linear correlation of the input. Mathematically, it is expressed as:$f(x) = x$
   - This function essentially passes the input as is to the output, maintaining the proportionality of the input values.

   (b) **Sigmoid Function:**

   - The sigmoid function outputs values between 0 and 1, making it especially suitable for binary classification problems. It is defined by the equation: $f(x) = \frac{1}{1+e^{-x}}$
   - This function is useful when the model needs to predict the probability as an output since the probability of anything exists only between the range of 0 and 1.

   (c) **Hyperbolic Tangent Function (TanH):**

   - The hyperbolic tangent function, or TanH, outputs values between -1 and 1. It is computed using the formula: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
   - TanH is similar to the sigmoid function but varies in that it can output negative values as well, making it more balanced and therefore often more effective in the hidden layers of a neural network.

6. **Flattening** Flattening is a key step in preparing data for fully connected layers in a Convolutional Neural Network (CNN). After an image passes through several convolutional and pooling layers, its multi-dimensional output is transformed into a one-dimensional vector. This vector contains all the extracted features necessary for the network to perform classification. Flattening allows the network to transition smoothly from feature extraction to high-level reasoning and prediction.

7. **Output Layer**The final layer of fully connected neurons provides the network's output. For classification tasks, this layer typically comprises as many neurons as there are classes and utilizes a softmax activation function to transform the network's output into probability scores for each class.

Optimization Algorithms for CNNs

1. **Stochastic Gradient Descent (SGD):** SGD updates the model's parameters by computing the gradient of the loss function concerning the parameters for a single sample at a time. The model's parameters are adjusted in the opposite direction of the gradient, with the magnitude of the adjustment dictated by the learning rate.

2. **Adam (Adaptive Moment Estimation):**Adam optimizer is a combination of momentum and RMSprop techniques. It tracks an exponentially decaying average of past gradients and an exponentially decaying average of the squares of past gradients. Adam adjusts the learning rate based on this information, helping to navigate the contours of the loss function more effectively than SGD, especially in complex optimization landscapes.

Model Details:

4.4 Model Performance

The convolutional neural network (CNN) demonstrated substantial progress in learning and generalization over nine training epochs, optimizing its performance for an image classification task. Starting with initial accuracy figures of 79.85% for training and 75.55% for validation, the model showed a steady improvement in accuracy and a reduction in loss with each epoch. Notably, by the ninth epoch, training accuracy soared to 98.96% with a training loss reduced to 0.0113, while validation accuracy peaked at 98.57% with a loss of 0.0223. This reflects the model's capability to effectively adapt and refine its parameters for high precision in image classification, aided by dynamic adjustments in the learning rate which started at 0.00828 and was gradually reduced to nearly zero.Then the training history of the model is given below:

```
Epoch [0], last_lr: 0.00280, train_loss: 0.6890, train_acc: 0.7985, val_loss: 0.9683, val_acc: 0.7555
Epoch [1], last_lr: 0.00760, train_loss: 0.3712, train_acc: 0.8872, val_loss: 0.6858, val_acc: 0.7907
Epoch [2], last_lr: 0.01000, train_loss: 0.3513, train_acc: 0.8898, val_loss: 0.9802, val_acc: 0.7263
Epoch [3], last_lr: 0.00950, train_loss: 0.2611, train_acc: 0.9161, val_loss: 0.4038, val_acc: 0.8659
Epoch [4], last_lr: 0.00812, train_loss: 0.2007, train_acc: 0.9349, val_loss: 0.2892, val_acc: 0.9047
Epoch [5], last_lr: 0.00611, train_loss: 0.1538, train_acc: 0.9495, val_loss: 0.2020, val_acc: 0.9343
Epoch [6], last_lr: 0.00389, train_loss: 0.1060, train_acc: 0.9656, val_loss: 0.0816, val_acc: 0.9726
Epoch [7], last_lr: 0.00188, train_loss: 0.0616, train_acc: 0.9795, val_loss: 0.0673, val_acc: 0.9770
Epoch [8], last_lr: 0.00050, train_loss: 0.0258, train_acc: 0.9822, val_loss: 0.0275, val_acc: 0.9802
Epoch [9], last_lr: 0.00000, train_loss: 0.0113, train_acc: 0.9896, val_loss: 0.0223, val_acc: 0.9857
CPU times: user 1h 29min 18s, sys: 4min 22s, total: 1h 33min 41s
Wall time: 1h 27min 4s
```

Figure 3: Caption

In our project report, we illustrate the model's performance with graphs showing training and validation results over several epochs. Figure 1 displays a steady decrease in both training and validation losses, indicating that the model is learning effectively. Figure 2 shows increasing training and validation accuracies, reaching up to 98.96% and 98.57% respectively by the ninth epoch, demonstrating strong generalization. These graphs confirm that the model is well-tuned, achieving high accuracy without overfitting, and is reliable for practical image classification tasks.

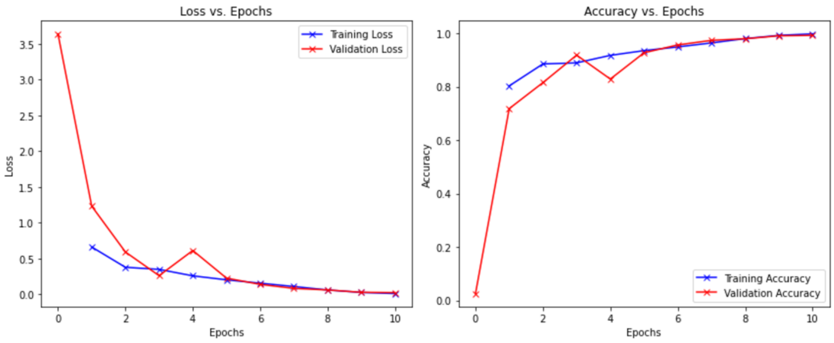


Figure 4: Caption

The confusion matrix provided gives a comprehensive overview of the model's performance across the different classes in the validation dataset. This matrix is essential for visualizing the accuracy of predictions and identifying classes where the model may be confusing one label for another.

The matrix dimensions are 38x38, corresponding to the 38 classes the model classifies. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class. The diagonal cells, which are highlighted, show the number of correct predictions for each class, indicating how often the model was correct for a particular class.

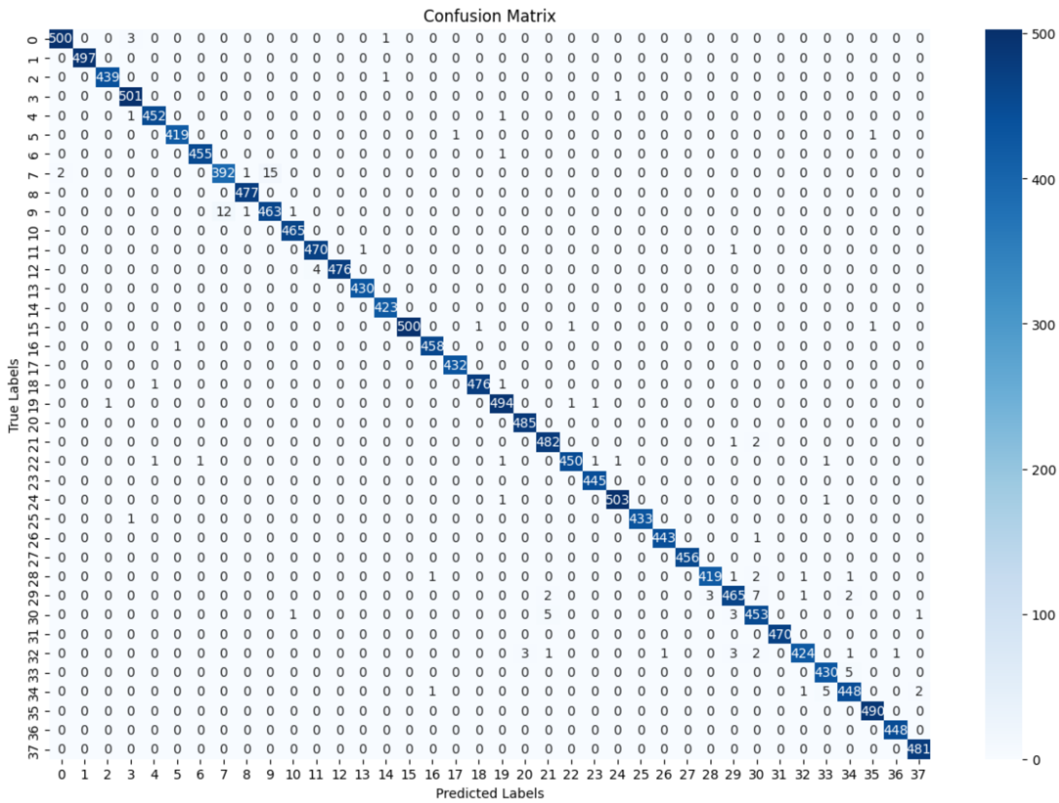


Figure 5: confusion matrix

This confusion matrix is crucial for identifying specific weaknesses in the model's classification abilities and provides clear direction for future improvements. Enhancements could include increasing the diversity of training examples for underperforming classes or tweaking the model architecture to better handle the features of those classes.

Model Testing

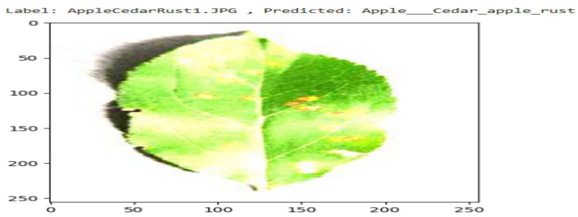
The model's performance on the validation dataset is impressive across precision, recall, and F1-score metrics, with nearly all values reaching perfect or near-perfect scores. These metrics confirm that the model is not only accurate in its predictions (precision) but also thorough in capturing relevant instances (recall) for almost every class. While most classes achieved scores of 1.00, a few showed slightly lower scores around 0.97 to 0.99, suggesting minor areas for improvement.

	precision	recall	f1-score	support
0	1.00	0.99	0.99	504
1	1.00	1.00	1.00	497
2	1.00	1.00	1.00	440
3	0.99	1.00	0.99	502
4	1.00	1.00	1.00	454
5	1.00	1.00	1.00	421
6	1.00	1.00	1.00	456
7	0.97	0.96	0.96	410
8	1.00	1.00	1.00	477
9	0.97	0.97	0.97	477
10	1.00	1.00	1.00	465
11	0.99	1.00	0.99	472
12	1.00	0.99	1.00	480
13	1.00	1.00	1.00	430
14	1.00	1.00	1.00	423
15	1.00	0.99	1.00	503
16	1.00	1.00	1.00	459
17	1.00	1.00	1.00	432
18	1.00	1.00	1.00	478
19	0.99	0.99	0.99	497
20	0.99	1.00	1.00	485
21	0.98	0.99	0.99	485
22	1.00	0.99	0.99	456
23	1.00	1.00	1.00	445
24	1.00	1.00	1.00	505
25	1.00	1.00	1.00	434
26	1.00	1.00	1.00	444
27	1.00	1.00	1.00	456
28	0.99	0.99	0.99	425
29	0.98	0.97	0.97	480
30	0.97	0.98	0.97	463
31	1.00	1.00	1.00	470
32	0.99	0.97	0.98	436
33	0.98	0.99	0.99	435
34	0.98	0.98	0.98	457
35	1.00	1.00	1.00	490
36	1.00	1.00	1.00	448
37	0.99	1.00	1.00	481
accuracy			0.99	17572
macro avg	0.99	0.99	0.99	17572
weighted avg	0.99	0.99	0.99	17572

Figure 6: Model's Performance

Overall, the model demonstrates excellent generalization with both macro and weighted averages for precision, recall, and F1-score at 0.99, and an overall accuracy of 99%.

In our project report, the testing stage of the model on a new dataset showcased its strong ability to recognize a variety of plant diseases accurately.The test outcomes of the model are given below:



The test outcomes revealed consistent and correct predictions across several conditions, such as "Apple Cedar apple rust," "Corn Common rust," and "Tomato Yellow Leaf Curl Virus," demonstrating the model's effective generalization from training to practical applications. This high level of accuracy in the test results confirms the model's reliability for real-world agricultural deployments. The minimal misclassifications observed suggest the model is well-tuned, though continued improvements could further refine its diagnostic capabilities. Also the result of the test dataset which conatins the 33 different image are given below:

```
Label: AppleCedarRust1.JPG , Predicted: Apple___Cedar_apple_rust
Label: AppleCedarRust2.JPG , Predicted: Apple___Cedar_apple_rust
Label: AppleCedarRust3.JPG , Predicted: Apple___Cedar_apple_rust
Label: AppleCedarRust4.JPG , Predicted: Apple___Cedar_apple_rust
Label: AppleScab1.JPG , Predicted: Apple___Apple_scab
Label: AppleScab2.JPG , Predicted: Apple___Apple_scab
Label: AppleScab3.JPG , Predicted: Apple___Apple_scab
Label: CornCommonRust1.JPG , Predicted: Corn_(maize)___Common_rust_
Label: CornCommonRust2.JPG , Predicted: Corn_(maize)___Common_rust_
Label: CornCommonRust3.JPG , Predicted: Corn_(maize)___Common_rust_
Label: PotatoEarlyBlight1.JPG , Predicted: Potato___Early_blight
Label: PotatoEarlyBlight2.JPG , Predicted: Potato___Early_blight
Label: PotatoEarlyBlight3.JPG , Predicted: Potato___Early_blight
Label: PotatoEarlyBlight4.JPG , Predicted: Potato___Early_blight
Label: PotatoEarlyBlight5.JPG , Predicted: Potato___Early_blight
Label: PotatoHealthy1.JPG , Predicted: Potato___healthy
Label: PotatoHealthy2.JPG , Predicted: Potato___healthy
Label: TomatoEarlyBlight1.JPG , Predicted: Tomato___Early_blight
Label: TomatoEarlyBlight2.JPG , Predicted: Tomato___Early_blight
Label: TomatoEarlyBlight3.JPG , Predicted: Tomato___Early_blight
Label: TomatoEarlyBlight4.JPG , Predicted: Tomato___Early_blight
Label: TomatoEarlyBlight5.JPG , Predicted: Tomato___Early_blight
Label: TomatoEarlyBlight6.JPG , Predicted: Tomato___Early_blight
Label: TomatoHealthy1.JPG , Predicted: Tomato___healthy
Label: TomatoHealthy2.JPG , Predicted: Tomato___healthy
Label: TomatoHealthy3.JPG , Predicted: Tomato___healthy
Label: TomatoHealthy4.JPG , Predicted: Tomato___healthy
Label: TomatoYellowCurlVirus1.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
Label: TomatoYellowCurlVirus2.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
Label: TomatoYellowCurlVirus3.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
Label: TomatoYellowCurlVirus4.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
Label: TomatoYellowCurlVirus5.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
Label: TomatoYellowCurlVirus6.JPG , Predicted: Tomato___Tomato_Yellow_Leaf_Curl_Virus
```

Figure 7: Test Result

5 Conclusion

In conclusion, our project has effectively demonstrated the substantial capabilities of Convolutional Neural Networks (CNNs) for complex image processing tasks, specifically in automatic plant disease detection. CNNs excel in environments with large datasets, leveraging their deep learning frameworks to perform robust feature extraction and achieve impressive generalization. However, challenges such as overfitting can arise when CNNs are applied to smaller datasets. To address this, we employed techniques such as hyperparameter optimization and data augmentation to increase the dataset's diversity and volume, significantly enhancing model performance.

Moreover, the application of transfer learning has been pivotal in our project, enabling us to harness pre-trained models to circumvent the limitations posed by the scarcity of extensive training data. This approach not only saved valuable resources but also maintained high accuracy levels, showcasing the adaptability and efficiency of CNNs. The results of our study affirm that CNNs are not only adaptable but also extremely powerful in handling detailed and nuanced tasks like disease detection in plants, making them invaluable tools in fields requiring high precision and reliability. This project lays a solid foundation for further exploration and application of CNNs in real-world scenarios, promising continued advancements in automated disease detection systems.

6 Future Work

The next stage of our crop disease detection project will focus on detailed mapping of disease distribution on individual leaves through advanced image segmentation techniques. This approach will allow us not only to identify but also to visually delineate the affected areas on the leaves. We plan to further leverage deep learning by incorporating sophisticated CNN architectures like ResNet50, enhanced by transfer learning from pre-trained models to boost our system's accuracy. Ultimately, our goal is to integrate this advanced model into a web-based application, creating an easy-to-use platform for farmers and agricultural professionals to monitor plant health in real time. This development is poised to transform agricultural disease management into a more proactive and data-driven practice.

7 References

1. Panchal AV, Patel SC, Bagyalakshmi K, Kumar P, Khan IR, Soni M. Image-based plant diseases detection using deep learning. Materials Today: Proceedings. 2023 Jan 1;80:3500-6.

2. Mohanty, Sharada P., David P. Hughes, and Marcel Salath´ e. ”Using deep learning for image-based plant disease detection.” Frontiers in plant science 7 (2016): 1419.

3. Saleem, Muhammad Hammad, Johan Potgieter, and Khalid Mahmood Arif. 2019. ”Plant Disease Detection and Classification by Deep Learning” Plants 8, no. 11: 468.

4. M. A. Jasim and J. M. AL-Tuwaijari, ”Plant Leaf Diseases Detection and Classification Using Image Processing and Deep Learning Techniques,” 2020 International Conference on Computer Science and Software Engineering (CSASE), Duhok, Iraq, 2020, pp. 259-265.

5. P. B R, A. Ashok and S. H. A V, ”Plant Disease Detection and Classification Using Deep Learning Model,” 2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA), Coimbatore, India, 2021, pp. 1285-1291,

6. Deeplearning from Campusx youtube channel.

Implementation Code

```
import os                        # for working with files
import numpy as np               # for numerical computationss
import pandas as pd              # for working with dataframes
import torch                     # Pytorch module
import matplotlib.pyplot as plt # for plotting informations on graph and images using tensors
import torch.nn as nn            # for creating  neural networks
from torch.utils.data import DataLoader # for dataloaders
from PIL import Image            # for checking images
import torch.nn.functional as F # for functions for calculating loss
import torchvision.transforms as transforms   # for transforming images into tensors
from torchvision.utils import make_grid       # for data checking
from torchvision.datasets import ImageFolder  # for working with classes and images
from torchsummary import summary              # for getting the summary of our model

data_dir = "../input/new-plant-diseases-dataset/New Plant Diseases Dataset(Augmented)/New Plant Dise
train_dir = data_dir + "/train"
valid_dir = data_dir + "/valid"
diseases = os.listdir(train_dir)

train_dir
```

```

valid_dir

diseases=sorted(diseases)
diseases

print("Total disease classes are: {}".format(len(diseases)))

plants = []
NumberOfDiseases = 0
for plant in diseases:
    if plant.split('___')[0] not in plants:
        plants.append(plant.split('___')[0])
    if plant.split('___')[1] != 'healthy':
        NumberOfDiseases += 1

# unique plants in the dataset
print(f"Unique Plants are: \n{plants}")

# number of unique plants
print("Number of plants: {}".format(len(plants)))

# number of unique diseases
print("Number of diseases: {}".format(NumberOfDiseases))

# Number of images for each disease
nums = {}
for disease in diseases:
    nums[disease] = len(os.listdir(train_dir + '/' + disease))

# converting the nums dictionary to pandas dataframe passing index as plant name and number of image

img_per_class = pd.DataFrame(nums.values(), index=nums.keys(), columns=["no. of images"])
img_per_class

# plotting number of images available for each disease
index = [n for n in range(38)]
plt.figure(figsize=(20, 5))
plt.bar(index, [n for n in nums.values()], width=0.3)
plt.xlabel('Plants/Diseases', fontsize=10)
plt.ylabel('No of images available', fontsize=10)
plt.xticks(index, diseases, fontsize=5, rotation=90)
plt.title('Images per each class of plant disease')

n_train = 0
for value in nums.values():
```

```
    n_train += value
print(f"There are {n_train} images for training")

# datasets for validation and training
train = ImageFolder(train_dir, transform=transforms.ToTensor())
valid = ImageFolder(valid_dir, transform=transforms.ToTensor())

train

valid

# The shape of the image
img, label = train[0]
print(img.shape, label)

# total number of classes in train set
len(train.classes)

# for checking some images from training dataset
def show_image(image, label):
    print("Label :" + train.classes[label] + "(" + str(label) + ")")
    plt.imshow(image.permute(1, 2, 0))

show_image(*train[64000])

show_image(*train[9000])

show_image(*train[30000])

# Setting the seed value
random_seed = 42
torch.manual_seed(random_seed)
# setting the batch size
batch_size = 32
# DataLoaders for training and validation
train_dl = DataLoader(train, batch_size, shuffle=True, num_workers=2, pin_memory=True)
valid_dl = DataLoader(valid, batch_size, num_workers=2, pin_memory=True)

# helper function to show a batch of training instances
def show_batch(data):
    for images, labels in data:
        fig, ax = plt.subplots(figsize=(30, 30))
        ax.set_xticks([]); ax.set_yticks([])
        ax.imshow(make_grid(images, nrow=8).permute(1, 2, 0))
        break

```

```
# for moving data into GPU (if available)
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available:
        return torch.device("cuda")
    else:
        return torch.device("cpu")

# for moving data to device (CPU or GPU)
def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

# for loading in the device (GPU if available else CPU)
class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)


device = get_default_device()
device

# Moving data into GPU
train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)

def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)  # Get the index of the max log-probability
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationBase(nn.Module):

    def training_step(self, batch):
```

```
        images, labels = batch
        out = self(images)  # Generate predictions
        loss = F.cross_entropy(out, labels)  # Calculate loss
        return loss  # Return loss as a tensor, not a dictionary

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                   # Generate predictions
        loss = F.cross_entropy(out, labels)  # Calculate loss
        acc = accuracy(out, labels)          # Calculate accuracy
        return {"val_loss": loss.detach(), "val_accuracy": acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x["val_loss"] for x in outputs]
        batch_accuracies = [x["val_accuracy"] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()       # Combine loss
        epoch_accuracy = torch.stack(batch_accuracies).mean()
        return {"val_loss": epoch_loss, "val_accuracy": epoch_accuracy} # Combine accuracies

    def training_epoch_end(self, outputs):
        batch_losses = [x["train_loss"] for x in outputs]
        batch_accuracies = [x["train_accuracy"] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()       # Combine loss
        epoch_accuracy = torch.stack(batch_accuracies).mean()
        return {"train_loss": epoch_loss, "train_accuracy": epoch_accuracy} # Combine accuracies

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, train_acc: {:.4f}, val_loss: {:.4f},
            epoch, result['lrs'][-1], result['train_loss'], result['train_accuracy'], result['val_lo

# Architecture for training

# Convolutional block with BatchNormalization
def ConvBlock(in_channels, out_channels, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True)]
    if pool:
        layers.append(nn.MaxPool2d(4))
    return nn.Sequential(*layers)


class ResNet14(ImageClassificationBase):
    def __init__(self, in_channels, num_diseases):
        super().__init__()

```

```
        # Initial convolutional layers
        self.conv1 = ConvBlock(in_channels, 64)                # Output: 64 x H x W
        self.conv2 = ConvBlock(64, 128, pool=True)             # Output: 128 x H/4 x W/4

        # Residual block 1
        self.res1 = nn.Sequential(
            ConvBlock(128, 128),
            ConvBlock(128, 128)
        )

        # Additional convolutional layers
        self.conv3 = ConvBlock(128, 256, pool=True)            # Output: 256 x H/16 x W/16
        self.conv4 = ConvBlock(256, 512, pool=True)            # Output: 512 x H/64 x W/64

        # Residual block 2
        self.res2 = nn.Sequential(
            ConvBlock(512, 512),
            ConvBlock(512, 512)
        )

        # New convolutional layer for deeper structure
        self.conv5 = ConvBlock(512, 1024, pool=True)           # Output: 1024 x H/256 x W/256

        # Residual block 3
        self.res3 = nn.Sequential(
            ConvBlock(1024, 1024),
            ConvBlock(1024, 1024)
        )

        # Updated Classifier with Global Average Pooling
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),                      # Global pooling to 1x1 spatial dimen
            nn.Flatten(),                                      # Flatten features
            nn.Linear(1024, num_diseases)                      # Fully connected layer
        )

    def forward(self, xb):
        out = self.conv1(xb)                                   # Conv1
        out = self.conv2(out)                                  # Conv2
        out = self.res1(out) + out                             # Residual block 1
        out = self.conv3(out)                                  # Conv3
        out = self.conv4(out)                                  # Conv4
        out = self.res2(out) + out                             # Residual block 2
        out = self.conv5(out)                                  # Conv5
        out = self.res3(out) + out                             # Residual block 3
        out = self.classifier(out)                             # Classifier
```

```python
        return out

# defining the model and moving it to the GPU
model = to_device(ResNet14(3, len(train.classes)), device)
model

from torchsummary import summary

# Define input shape (RGB images of size 256x256)
INPUT_SHAPE = (3, 256, 256)

# Print summary of the model
print(summary(model.cuda(), INPUT_SHAPE))

# Function to evaluate the model
@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

# Function to get learning rate
def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

# Function to train the model using the One Cycle LR Policy
def fit_OneCycle(epochs, max_lr, model, train_loader, val_loader, weight_decay=0,
                grad_clip=None, opt_func=torch.optim.SGD):
    torch.cuda.empty_cache()
    history = []

    optimizer = opt_func(model.parameters(), max_lr, weight_decay=weight_decay)
    # Scheduler for one-cycle learning rate
    sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs, steps_per_epoch=le

    for epoch in range(epochs):
        # Training
        model.train()
        train_losses = []
        train_accuracies = []  # Collect training accuracies
        lrs = []

        for batch in train_loader:
            images, labels = batch
            images, labels = images.to(device), labels.to(device)
```

```
            # Forward pass and loss calculation
            out = model(images)
            loss = F.cross_entropy(out, labels)
            train_losses.append(loss)

            # Calculate accuracy
            acc = accuracy(out, labels)
            train_accuracies.append(acc)

            # Backpropagation
            loss.backward()

            # Gradient clipping
            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

            # Record learning rates
            lrs.append(get_lr(optimizer))
            sched.step()

        # Validation
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['train_accuracy'] = torch.stack(train_accuracies).mean().item()  # Calculate mean tra
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)

    return history

%%time
history = [evaluate(model, valid_dl)]
history

epochs = 10
max_lr = 0.01
grad_clip = 0.1
weight_decay = 1e-4
opt_func = torch.optim.Adam

%%time
history += fit_OneCycle(epochs, max_lr, model, train_dl, valid_dl,
```

```
                              grad_clip=grad_clip,
                              weight_decay=1e-4,
                              opt_func=opt_func)
# Get the last epoch's training and validation accuracy
last_epoch_data = history[-1]
train_accuracy = last_epoch_data['train_accuracy']
valid_accuracy = last_epoch_data['val_accuracy']

print(f"Training Accuracy: {train_accuracy*100:.2f}%")
print(f"Validation Accuracy: {valid_accuracy*100:.2f}%")

def plot_accuracies(history):
    accuracies = [x['val_accuracy'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');
def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs');
def plot_lrs(history):
    lrs = np.concatenate([x.get('lrs', []) for x in history])
    plt.plot(lrs)
    plt.xlabel('Batch no.')
    plt.ylabel('Learning rate')
    plt.title('Learning Rate vs. Batch no.');
plot_accuracies(history)

plot_losses(history)

plot_lrs(history)

def plot_metrics(history):
    # Extract metrics from history
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x.get('val_loss') for x in history]
    train_accuracies = [x.get('train_accuracy') for x in history]
    val_accuracies = [x.get('val_accuracy') for x in history]

    # Plot Loss curves
```

```
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, '-bx', label='Training Loss')
    plt.plot(val_losses, '-rx', label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Loss vs. Epochs')

    # Plot Accuracy curves
    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, '-bx', label='Training Accuracy')
    plt.plot(val_accuracies, '-rx', label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Accuracy vs. Epochs')

    plt.tight_layout()
    plt.show()

plot_metrics(history)

test_dir = "../input/new-plant-diseases-dataset/test"
test = ImageFolder(test_dir, transform=transforms.ToTensor())

test_images = sorted(os.listdir(test_dir + '/test')) # since images in test folder are in alphabetic
test_images

def predict_image(img, model):
    """Converts image to array and return the predicted class
        with highest probability"""
    # Convert to a batch of 1
    xb = to_device(img.unsqueeze(0), device)
    # Get predictions from model
    yb = model(xb)
    # Pick index with highest probability
    _, preds  = torch.max(yb, dim=1)
    # Retrieve the class label

    return train.classes[preds[0].item()]

# predicting first image
img, label = test[0]
plt.imshow(img.permute(1, 2, 0))
print('Label:', test_images[0], ', Predicted:', predict_image(img, model))
```

```python


# getting all predictions (actual label vs predicted)
for i, (img, label) in enumerate(test):
    print('Label:', test_images[i], ', Predicted:', predict_image(img, model))


```