Plant Disease Classification Using Convolutional Neural Networks

Kuan-Ting Chen
Department of Biomechatronics Engineering, National Taiwan University, Taipei, Taiwan

ABSTRACT

With the increasing impact of plant diseases on global agricultural production, rapid and accurate diagnosis of diseases has become a significant challenge [3]. This study integrates deep learning technology and utilizes convolutional neural networks (CNN) [4] to classify plant leaf diseases into three categories: healthy, powdery mildew, and rust spots. The dataset used in this study is sourced from publicly available data on Kaggle [2], comprising a total of 1,472 high-resolution images, which are divided into training, testing, and validation sets. To enhance classification efficiency, preprocessing of the original images is required, including data augmentation and space resolution adjustments. Through experiments with different parameters, we analyzed the impact of number of CNN layers, convolutional kernel size, activation functions, optimizer, dropout rate, training epochs and batch size on model performance. The results show that appropriate parameter settings can effectively improve classification accuracy, achieving high F1 scores in distinguishing healthy leaves and disease categories. This study demonstrates the potential of deep learning in plant disease classification.

Keywords: Plant Phenotyping, Plant Disease Recognition, Image Classification, Convolutional Neural Networks, Image Preprocessing, Gaussian Noise

1. INTRODUCTION

Plant diseases pose a significant challenge to agricultural production, especially as global population growth exacerbates food security concerns. Traditional methods for diagnosing plant diseases rely heavily on manual observation and expert experience, which are time-consuming and prone to subjective bias [3]. This study leverages deep learning technology and utilizes convolutional neural networks (CNNs) [4] for plant disease image classification, providing a solution for automated and efficient disease detection.

The dataset used in this study consists of 1,472 high-resolution images labeled as Healthy, Powdery Mildew, or Rust [2], capturing leaf features under various plant conditions (Figure 1). To improve model performance, various experiments were conducted to examine the effects of different convolutional layers (e.g., two layers, four layers), kernel sizes (e.g., 3, 5, 7), activation functions (e.g., ReLU, ELU), batch sizes, and training epochs on classification outcomes. Additionally, preprocessing techniques such as data augmentation were employed to optimize input data, reduce overfitting, and enhance the model's generalization capability.

The results demonstrate that a four layer convolutional neural network effectively balances computational efficiency and feature extraction capability. The model performed exceptionally well in classifying Healthy, Powdery Mildew and Rust categories, achieving both F1 scores and Accuracy close to 1.

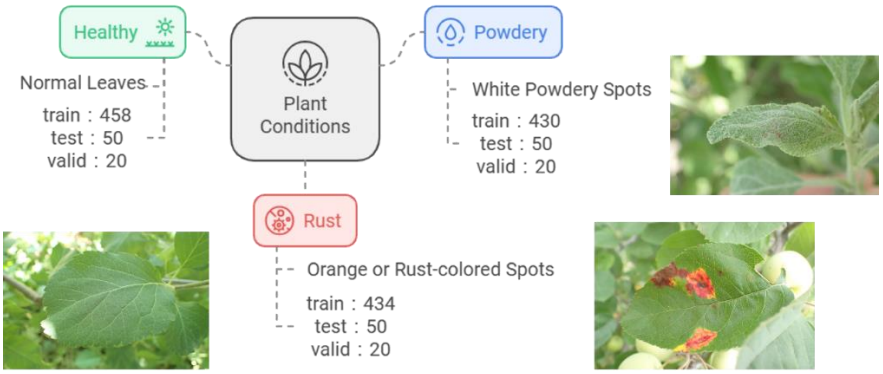


Figure 1. The dataset [2] includes three labels: "Healthy," "Powdery Mildew," and "Rust," referring to plant conditions. A total of 1,472 images are divided into a training set (1,322), a test set (150), and a validation set (60).

2. METHODOLOGY

2.1 Algorithm Overview

Two-layer Convolutional Neural Network:

In plant phenotyping analysis, image classification models serve as key tools for quantifying the morphology, structure, and functional characteristics of plants under specific environmental conditions. This study employs a two-layer convolutional neural network (Two-layer CNN) built on the TensorFlow framework to perform three-class classification of plant diseases, focusing on the health status of leaves: Healthy, Powdery Mildew, and Rust.

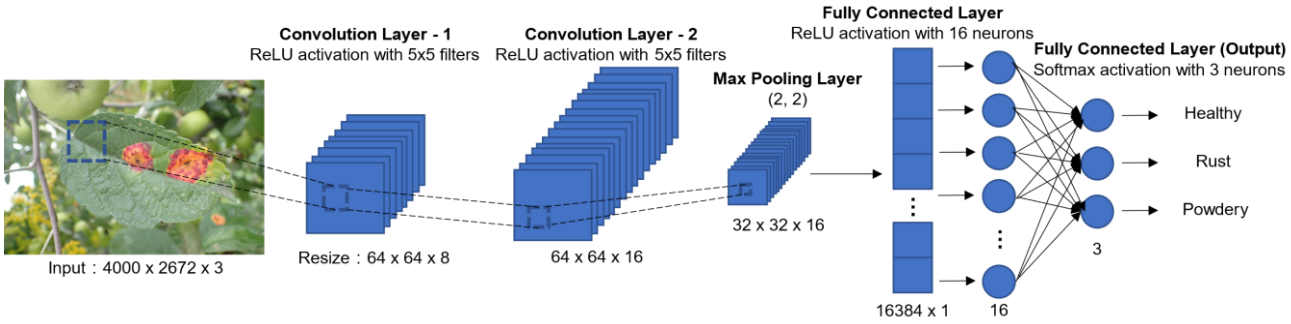


Figure 2. This is a Two-layer Convolutional Neural Network (Two-layer CNN) architecture designed for image classification. It extracts features through convolutional layers, reduces dimensionality via pooling layers, flattens into fully connected layers, and classifies results into Healthy, Rust, or Powdery Mildew categories.

Four-layer Convolutional Neural Network:

To achieve higher classification accuracy, the model complexity is increased with the following modifications:

1. The number of convolutional layers is increased to 4, the input image size is enlarged to 256×256, and the kernel size is increased to 64 to extract finer image features.
2. A max-pooling layer of size 2×2 is added after each convolutional layer, and a Dropout layer is applied after each fully connected layer to prevent overfitting.
3. The number of fully connected layers is increased to match the number of convolutional layers, with the number of neurons set to 500, 500, 100, and 3, respectively.
4. Batch size and training epochs will be optimized based on experimental procedures to determine the parameters that achieve the best model accuracy.

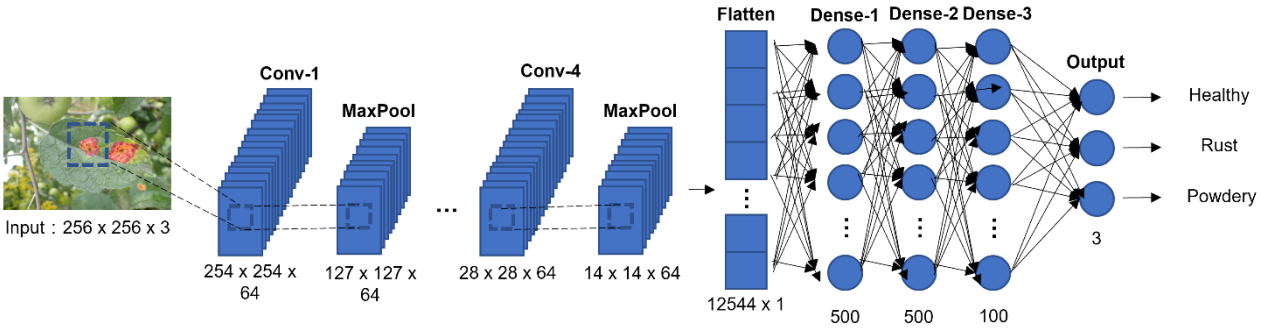


Figure 3. Four-Layer Convolutional Neural Network Architecture for Plant Disease Classification

The structure of the two layer CNN model includes the following layers :

2.1.1 Convolutional Layer :

The first convolutional layer has an output depth (number of convolutional kernels) of 8, generating 8 feature maps to extract basic features, while the second layer generates 16 feature maps to capture higher-level abstract features. Each convolutional kernel is set to a size of 5×5, sliding over the input image to extract features within a 5×5 region. The activation function used is ReLU (Rectified Linear Unit), defined as $Relu(x) = Max(0, x)$, where positive input values x are retained as output, and zero or negative input values are set to 0. This non-linear transformation enables the model to learn complex features. The input shape is specified as 64×64 RGB images using input_shape=(64, 64, 3). The parameter data_format='channels_last' indicates that the data is arranged in the order (height, width, channels), consistent with TensorFlow's default data format.

```
# Add CNN with 8 filters, input images have a shape of 64x64 with 3 color channels (RGB).
self.model.add(Conv2D(8, hidden_kernel_size, activation=hidden_activation,
                      input_shape=(64, 64, 3), data_format='channels_last'))
# Add another CNN layer with 16 filters.
self.model.add(Conv2D(16, hidden_kernel_size, activation=hidden_activation))
```

Figure 4. This code adds two CNN layers to the model, one with 8 filters and the other with 16 filters. Parameters such as kernel size, activation function, and input image shape can be retrieved from the backend, allowing dynamic adjustment of the model.

2.1.2 Pooling Layer :

The max-pooling operation extracts the maximum value from each 2×2 region, achieving the following objectives: reducing the size of feature maps and lowering computational costs; preserving essential features to enhance noise robustness; preventing overfitting and improving the model's generalization capability. For instance, in this study, an input feature map of size 64×64 is reduced to 32×32 after this operation.

```
# Add a pooling layer to reduce the spatial dimensions of the output volume.
self.model.add(MaxPooling2D((2, 2)))
```

Figure 5. This code adds a max-pooling layer (MaxPooling2D) to the model, reducing the number of parameters and preventing overfitting. By setting (2,2) as the pooling size, the width and height of the feature maps are halved. ◦

2.1.3 Flattening Layer :

The flattening layer converts multi-dimensional data into a one-dimensional vector, enabling the output from the pooling layer to serve as input for subsequent fully connected layers. For instance, in this study, an input pooling layer feature map of size 32×32×16 is transformed into a one-dimensional vector of length 16,384 through this operation.

2.1.4 Fully Connected Layer :

This layer contains 16 neurons, controlling the model's learning capacity and computational complexity. Each neuron is connected to all outputs of the previous layer, performing weighted summation and activation operations. The output features are further transformed into a 16-dimensional feature representation. The activation function used is ReLU, defined as:

$$y_i = \text{Relu}\left(\sum_{j=1}^{n} w_{ij} x_j + b_i\right)$$

where w_{ij} represents the weights, b_i represents the biases, and x_j is the output from the previous layer. This enables the layer to exhibit non-linear behavior, which is essential for learning complex patterns. Positioned after convolutional or pooling layers, the fully connected layer serves as a feature classifier.

2.1.5 Dropout Layer :

During training, the dropout layer randomly disables a portion of the neurons, making the network more robust by preventing over-reliance on specific neuron weights and reducing the risk of overfitting. For instance, when the dropout rate is set to 0.4, 40% of the neuron outputs are randomly set to 0, while the remaining 60% continue to participate in the computations.

```
self.model.add(Flatten())
self.model.add(Dense(16, activation=hidden_activation))
self.model.add(Dropout(dropout_rate))
```

Figure 6. This code adds three layers to the model: Flatten layer - flattens multi-dimensional feature maps into a one-dimensional vector. Dense layer - a fully connected layer with 16 neurons, allowing the selection of the desired activation function. Dropout layer - randomly masks elements in the neural network to prevent overfitting.

2.1.6 Output Layer :

This layer is set to have 3 neurons, corresponding to the 3 classification labels. As the output layer, it produces the predicted probabilities for each class. The **Softmax function** is applied, defined as:

$$Softmax(z_i) = e^{z_i} / \sum_{j=1}^{n} e^{z_j}$$

where z_i represents the linear output value (logits) of the i-th neuron. The exponential function e^{z_i} transforms the linear outputs into non-negative values, ensuring no negative numbers in the final probability distribution. The term $\sum_{j=1}^{n} e^{z_j}$ calculates the sum of the exponential values for all classes, normalizing each class output so that the sum of all outputs equals 1.

In summary, the Softmax function converts the linear outputs into a probability distribution where all values sum to 1, and each output lies between 0 and 1. Combined with the fully connected layer, it produces the percentage probabilities for the three-class classification results.

```
# Classification task: The number of neurons in the last layer is equal to the number of categories.
# Output layer use softmax activation function, 3 categories : Rust, Healthy, Powdery.
self.model.add(Dense(3, activation='softmax'))
```

Figure 7. This code adds an output layer to the model, which contains 3 neurons corresponding to the 3 classification targets (Rust, Healthy, Powdery). The output layer uses the Softmax activation function to ensure that the output values are between 0 and 1, and their sum equals 1, thereby providing the probability predictions for each class.

2.1.7 Compiling the Model :

After constructing the entire model architecture, configurations are prepared for training. For a K-class classification problem, the model's output is a K-dimensional vector representing the predicted probabilities for each class. The true labels are encoded as a one-hot vector, where only the correct class position is 1, and all other positions are 0.

The categorical_crossentropy loss function computes the cross-entropy between the model's output and the true labels. A smaller cross-entropy indicates that the model's predicted probability distribution is closer to the true labels, resulting in better model performance.

The Adam (Adaptive Moment Estimation) optimizer is used, which is a popular optimization algorithm for training deep learning models. It is an adaptive learning rate optimization method that combines the advantages of Momentum and the RMSProp algorithms.

```
# Compile the model with categorical crossentropy loss function and the default optimizer is adam.
self.model.compile(loss='categorical_crossentropy', optimizer=self.param['optimizer'])
```

Figure 8. This code compiles the model using the categorical_crossentropy loss function and sets Adam as the default optimizer.

The parameter settings of the algorithm consider a balance between model performance and computational resources. For example, a 64×64 image resolution is chosen to retain sufficient features while reducing computational requirements. The ReLU activation function is used to accelerate convergence, and the Adam optimizer is selected for its advantages in adaptive learning rate adjustment, enabling fast and efficient convergence. The following experimental steps will provide a detailed explanation of data preprocessing and the parameter settings of the algorithm.

2.2 Experimental Procedure

2.2.1 Data Preprocessing (Adjusting Spatial Resolution and Grayscale Conversion) :

Due to the excessively high resolution of the original images (e.g., 4000×2697), which results in longer computation time, the experiment attempts to reduce the image size to 0.1 times the original resolution and convert the images to grayscale.

```
def batch_adjust_resolution(self):  1 usage
    input_folder = QFileDialog.getExistingDirectory(self, parent: 'Select Input Folder')
    if not input_folder:
        return

    output_folder = QFileDialog.getExistingDirectory(self, parent: 'Select Output Folder')
    if not output_folder:
        return
```

Figure 9. The code defines the batch_adjust_resolution function, which processes all images in an input folder in batches and outputs them to an output folder after preprocessing.

However, this simplified approach weakens the model's ability to extract features, leading to meaningless classification results (e.g., all images in Figure 10 are classified as healthy). The conclusion is that grayscale conversion may not be suitable for this dataset, as the color characteristics of diseases are crucial for accurate identification. It is recommended to retain RGB color information and apply data augmentation (e.g., flipping, scaling, adding noise, etc.) to enhance the model's generalization capability.

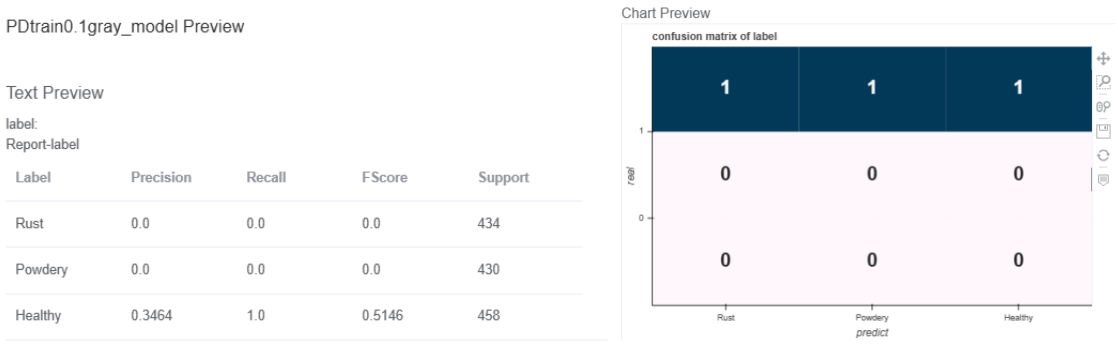


Figure 10. The training results of the model after reducing the image size to 0.1 times and converting to grayscale show poor classification performance. The model only identifies healthy leaves, while the other two categories (Rust and Powdery) are not correctly predicted, resulting in Precision, Recall, and F1 Score all being 0.

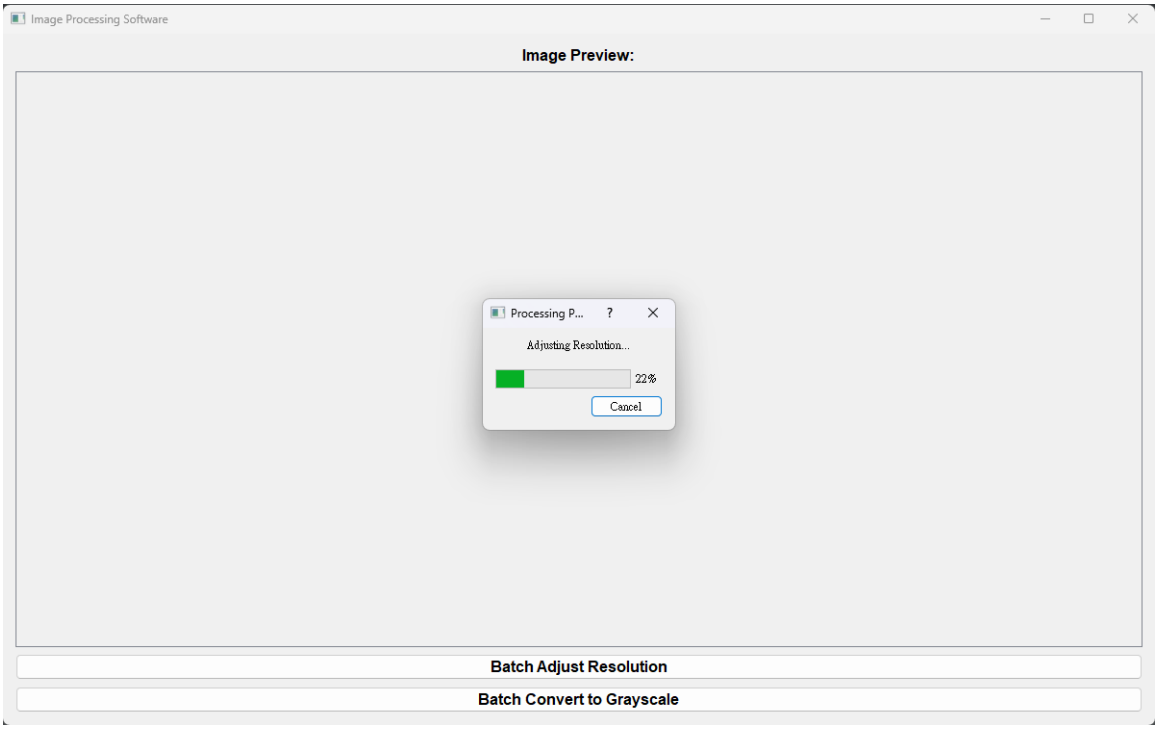


Figure 11. A UI interface was designed using PyQt5 for this batch processing program. It allows batch adjustments of spatial resolution and grayscale conversion while displaying the processing progress.

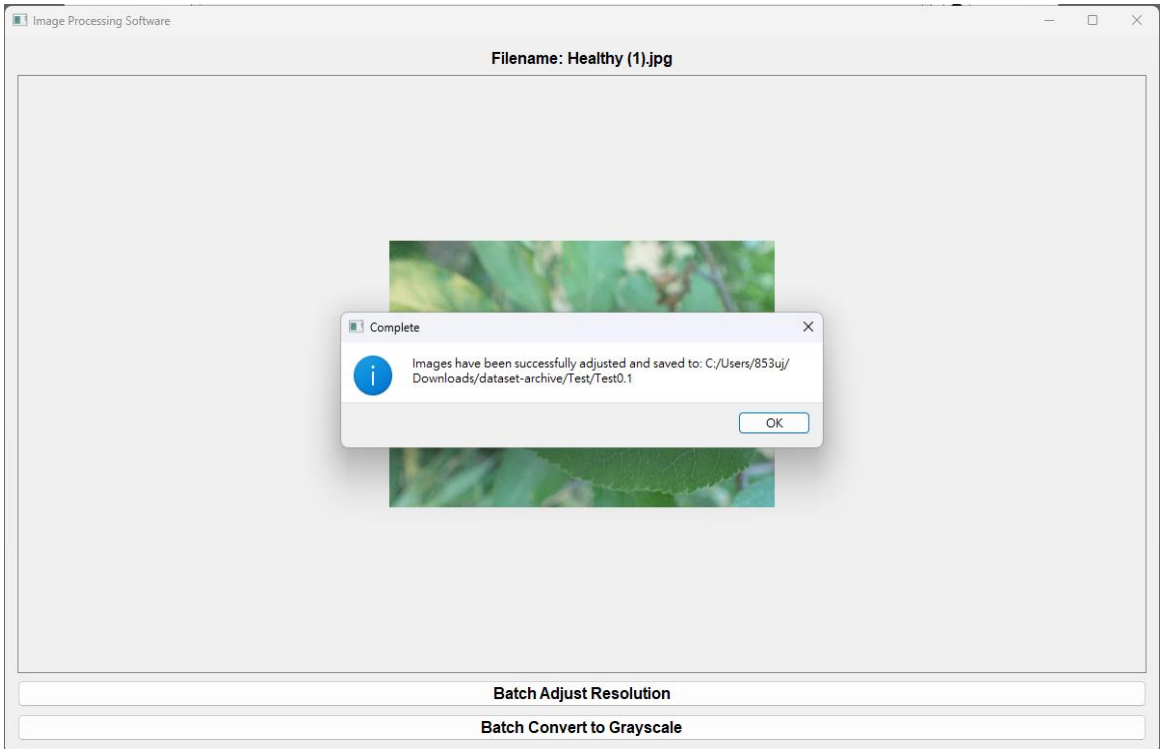


Figure 12. After processing is complete, the UI enables previewing of the results. A popup window notifies the user of the success or failure of the batch processing and displays the output folder path.

2.2.2 Data Preprocessing (Data Augmentation) :

Resizing and Rescaling: Images are resized to a uniform size, and pixel values are normalized to provide consistent and efficient data input for the model.

Data Augmentation: Random transformations are applied to generate more diverse training samples, improving the model's generalization ability. This approach is particularly useful for scenarios with limited data, as in this study, or when robustness needs to be enhanced.

Data Augmentation

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Resizing(256,256),
    tf.keras.layers.Rescaling(1.0/255),
    tf.keras.layers.RandomContrast(0.3),
    tf.keras.layers.RandomFlip('horizontal_and_vertical'),
    tf.keras.layers.RandomZoom(0.3),
    tf.keras.layers.RandomRotation(0.2),
    tf.keras.layers.Lambda(lambda x: add_gaussian_noise(x, mean=0.0, stddev=0.05))
])
```

Figure 13. The figure illustrates a sequential data augmentation pipeline implemented using TensorFlow/Keras.

Gaussian noise : Generated with a specified mean and standard deviation using tf.random.normal and added to the original image. To ensure pixel values remain valid, the resulting noisy image is clipped to a range between 0.0 and 1.0 using tf.clip_by_value. This approach enhances the model's robustness by enabling it to handle noisy data, improving generalization in real-world scenarios.

Gaussian Noise

```
def add_gaussian_noise(images, mean, stddev):
    noise =tf.random.normal(shape=tf.shape(images), mean=mean, stddev=stddev, dtype=tf.float32)
    noisy_images = images + noise
    return tf.clip_by_value(noisy_images, 0.0, 1.0)
```

Figure 14. A custom function `add_gaussian_noise` implemented to introduce Gaussian noise to images.

These preprocessing techniques effectively improve model performance, especially in situations with limited datasets, by enabling the model to learn features across a wider range of variations.

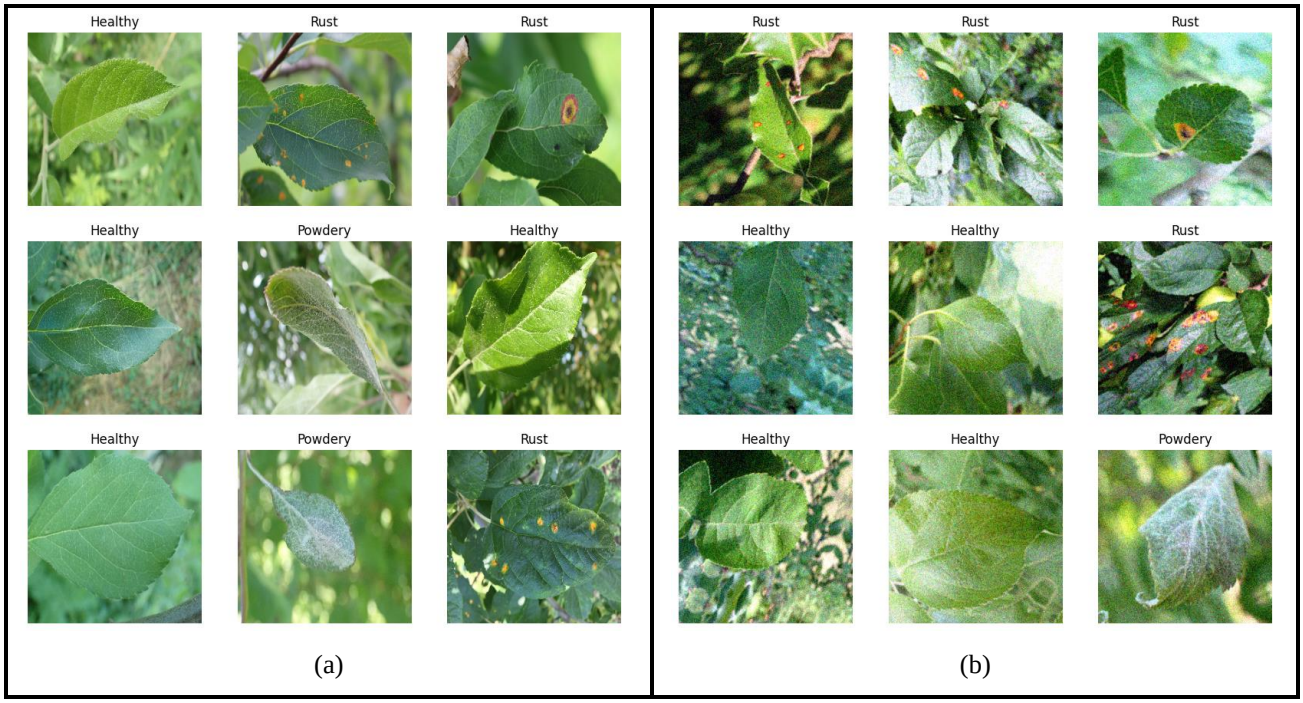


Figure 15. (a) displays the original image in train dataset. (b) displays the results of data augmentation applied to the training dataset, showcasing various transformations such as resizing, random contrast adjustments, flipping, and Gaussian noising. The images are labeled with their corresponding classes: Rust, Healthy, and Powdery. Each transformed image is rescaled back to the range of [0,255] and converted to uint8 for visualization. The augmentation introduces variability in the dataset, enhancing the model's ability to generalize by simulating different real-world conditions, such as changes in orientation, lighting, and perspective.

2.2.3 Parameter Adjustment :

Inanalysis Platform:

The platform allows adjustments to parameters such as convolution kernel size, activation function, optimizer, dropout rate, training epochs, and batch size as needed.

```
{
    "dataType":"cv",
    "projectType":"classification",
    "algoName":"r13631011_FourLayerCNN",
    "description": "Four layers Convolutional Neural Network.",
    "lib":"keras",
    "param":[
        {
            "name":"hidden_kernel_size",
            "description":"The size of the convolutional filter (kernel)",
            "type":"int",
            "lowerBound":3,
            "upperBound":7,
            "default":3
        },
```

Figure 16. Parameters can be added to a JSON file, specifying the parameter name, description, type, range, and default value. When uploaded to the backend system, as shown in Figure 17, these parameters can be adjusted dynamically via the UI interface for model configuration.

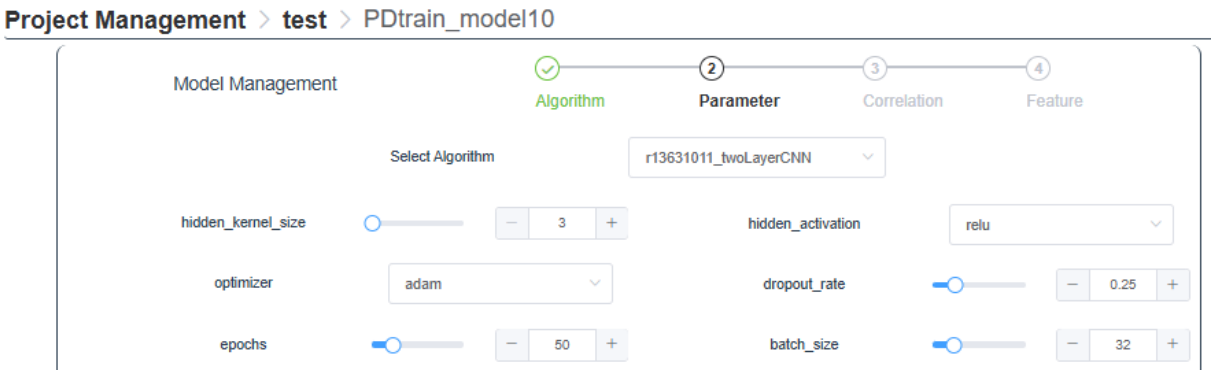


Figure 17. The Inanalysis model management system [1] allows users to adjust parameters such as convolution kernel size, activation function, optimizer, dropout rate, training epochs, and batch size as needed.

Kaggle Platform :

After multiple model training experiments on the Inanalysis platform, insights are gained on efficiently fine-tuning the optimal parameter combination. Parameters can also be adjusted directly by modifying the code. Combined with other program blocks, such as data augmentation and result analysis charts, the best-performing training model can be achieved.

```
IMAGE_SIZE=256
CHANNELS=3
BATCH_SIZE=32
EPOCHS=30

input_shape=(BATCH_SIZE , IMAGE_SIZE, IMAGE_SIZE, CHANNELS)

model= tf.keras.models.Sequential([
  data_augmentation,
  # Convolution layer 1
  tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),padding='valid',activation='relu',input_shape=input_shape),
  tf.keras.layers.MaxPool2D(pool_size=(2,2)),

  # Convolution layer 2
  tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),strides=(1,1),padding='valid',activation='relu'),
  tf.keras.layers.MaxPool2D(pool_size=(2,2)),

  # Convolution layer 3
  tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),strides=(1,1),padding='valid',activation='relu'),
  tf.keras.layers.MaxPool2D(pool_size=(2,2)),

  # Convolution layer 4
  tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),strides=(1,1),padding='valid',activation='relu'),
  tf.keras.layers.MaxPool2D(pool_size=(2,2)),

  # Flatten Layers
  tf.keras.layers.Flatten(),

  # Dense layers
  tf.keras.layers.Dense(units=500,activation='relu'),
  tf.keras.layers.Dropout(0.4),
  tf.keras.layers.Dense(units=500,activation='relu'),
  tf.keras.layers.Dropout(0.3),
  tf.keras.layers.Dense(units=100,activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(units=3,activation='softmax')

])

model.build(input_shape=input_shape)
```

Figure 18. Four layer CNN model architecture implementation on Kaggle

3. RESULTS

The model performance results for the dataset classified into Healthy, Powdery, and Rust will be discussed, focusing on the effects of parameter adjustments. Multiple experiments for result analysis.

3.1 2LCNN - Experiment 1: Determining Input Size

Using 150 test images (a total of 120 MB) for quick training, the training process takes approximately 15 minutes on Inanalysis by GPU 1080Ti. Each class contains 50 images, and the input size (input_shape) in the algorithm is tested.

model	resize input	kernel	activation	optimizer	dropout	epochs	batch_size	H_Fscore	P_Fscore	R_Fscore	Avg_Fscore	Outcome
150-1	32*32	3	relu	0.25	0.25	50	32	0.7792	0.8571	0.75	0.795433	Normal
150-2	64*64	3	relu	0.25	0.25	50	32	1	1	1	1	Overfitting

Table 1. The F1 scores for Healthy, Powdery, and Rust classes (H_Fscore, P_Fscore, R_Fscore) are used to evaluate the model's classification performance for each category, with the average F1 score (Avg_Fscore) serving as the overall performance evaluation metric.

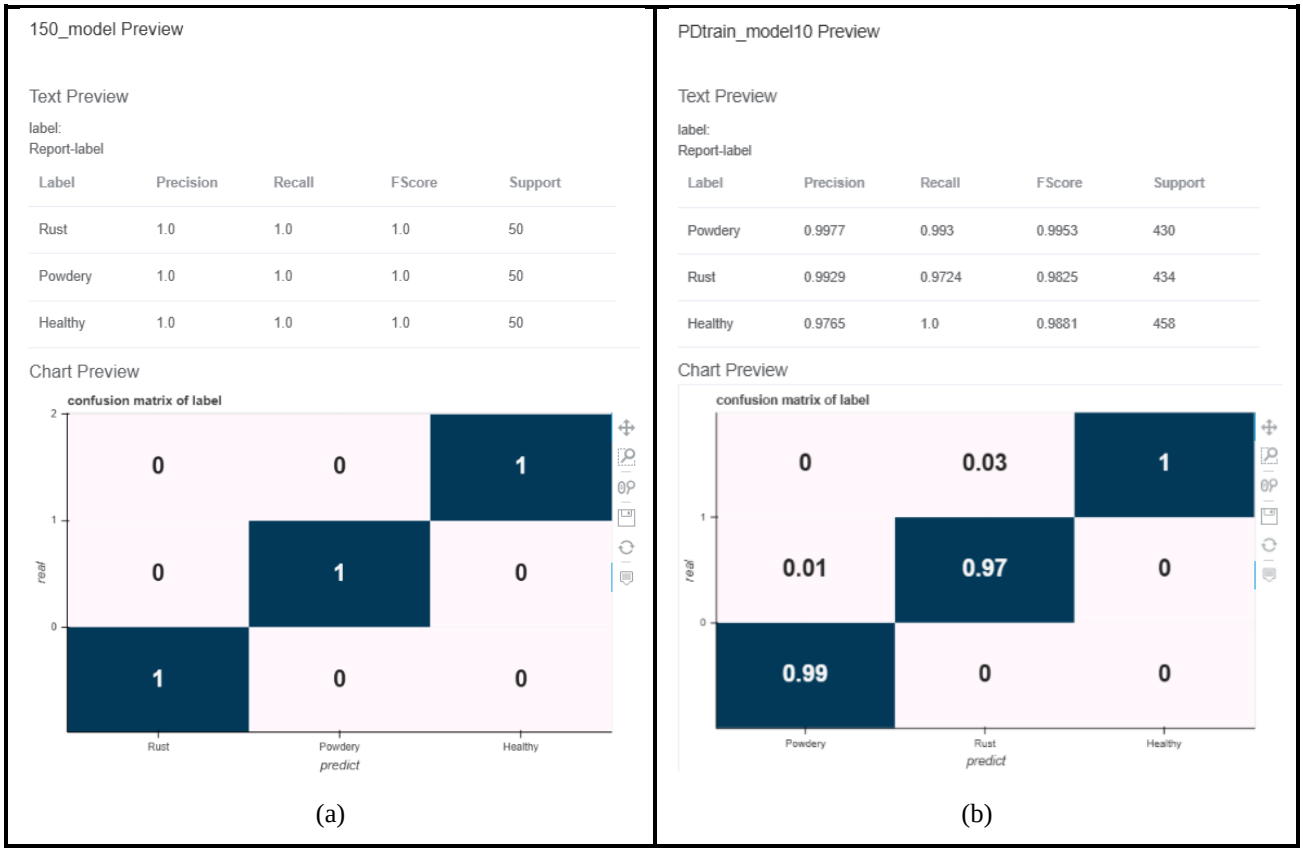


Figure 19. (a) shows the overfitting training result of the model 150-2 using 150 test images, while (b) shows the model 10 using 1,322 training images

Model 150-1 (input size 32×32) achieved an Avg_Fscore of 0.795433, indicating a relatively balanced performance across the three classes.Model 150-2 (input size 64×64) achieved an Avg_Fscore of 1, suggesting perfect performance on the training data but exhibiting clear overfitting, which may limit its generalization ability on test data.

The possible reasons for this include:The dataset is relatively small, and using an input size of 64×64 leads to overfitting quickly. To balance computational efficiency and prevent overfitting, larger datasets make overfitting less likely. Therefore, 64×64 is temporary chosen as the input size for the formal training dataset of 1,322 images.

3.2 2LCNN - Experiment 2: Determining Kernel Size

The formal training uses 1,322 images (a total of 1.09 GB) with the Two-Layer CNN. Training takes approximately 2 hours on Inanalysis, and the kernel size in the algorithm is tested.

model	kernel	activation	optimizer	dropout	epochs	batch_size	H_Fscore	P_Fscore	R_Fscore	Avg_Fscore	Outcome
1	3	relu	adam	0.25	50	32	0.9956	0.9919	0.9895	0.992333	Overfitting
2	5	relu	adam	0.4	50	32	0.9309	0.9882	0.916	0.945033	Normal
3	7	relu	adam	0.4	75	32	0.9956	0.9872	0.985	0.989267	Normal

Table 2. Tests were conducted using different convolution kernel sizes, with adjustments made to other parameters such as dropout rate, training epochs, and batch size.

Increasing the hidden_kernel size (e.g., 5, 7), when combined with appropriate settings for other parameters (dropout rate, training epochs, and batch size), has a positive impact on model performance.

3.3 2LCNN - Experiment 3: Determining the Activation Function

The formal training uses 1,322 images (a total of 1.09 GB), testing the ELU (Exponential Linear Unit) activation function. ELU is an improved version of ReLU, with its output range including negative values, which helps reduce bias issues, mitigates vanishing gradients, and accelerates convergence.

model	kernel	activation	optimizer	dropout	epochs	batch_size	H_Fscore	P_Fscore	R_Fscore	Avg_Fscore	Outcome
4	5	elu	adam	0.4	50	32	1	1	1	1	Overfitting
5	3	elu	adam	0.25	50	32	1	1	1	1	Overfitting
6	7	elu	adam	0.4	75	32	1	1	1	1	Overfitting

Table 3. Using the ELU activation function demonstrates its clear convergence-accelerating properties; however, the results indicate overfitting.

The ELU activation function fails to achieve better training outcomes, leading to the selection of ReLU as the primary activation function for this study.

3.4 2LCNN - Experiment 4: Finding the Optimal Parameter Combination

The formal training uses 1,322 images (a total of 1.09 GB), testing various combinations of parameters such as kernel size, dropout rate, batch size, and training epochs to determine the optimal settings. Although ELU was tested for its improved characteristics, including its ability to output negative values, mitigate bias issues, reduce vanishing gradients, and accelerate convergence, ReLU remains the preferred activation function due to its superior performance in this task.

model	kernel	activation	optimizer	dropout	epochs	batch_size	Avg_Fscore	Avg_Test_Fscore	Avg_Predict_Fscore	Outcome
7	5	relu	adam	0.4	100	64	0.9726	0.6952	0.8073	Best Predict
8	5	relu	adam	0.4	75	32	0.997733	0.7875	0.6286	Best Test

Table 4. The two best parameter combinations from multiple model training experiments are presented: model 7 achieves the best Avg_Predict_Fscore, while Model 8 achieves the best Avg_Test_Fscore.

After dozens of model training experiments, the two optimal parameter combinations were identified. However, both testing and validation results did not exceed 90%, indicating the need for further optimization of the algorithm and preprocessing of the dataset.

3.5 4LCNN - Experiment 1: Determining the Optimal Training Epochs and Batch Size

This experiment focuses on identifying the best combination of training epochs and batch size for the four-layer convolutional neural network (4LCNN), training takes approximately 30 minutes on Kaggle by GPU T4.

model	epochs	batch_size	Test_Avg_FScore	Predict_Avg_FScore	Description
1	10	32	0.939767	0.931233	
2	30	32	0.953333	0.949967	Best Test
3	50	32	0.940567	0.9666	Best Predict

Table 5. Training with different numbers of epochs, and a comparison was made between the average F1 score on the test dataset and the average F1 score on the predicted dataset.

The table summarizes the performance of three models trained with different numbers of epochs and a batch size of 32, evaluated using the average F1 scores on the test and predicted datasets. Model 2, trained for 30 epochs, achieves the highest Test_Avg_FScore of 0.953333, indicating the best generalization performance on the test dataset. In contrast, model 3, trained for 50 epochs, achieves the highest Predict_Avg_FScore of 0.9666, demonstrating the best fit to the training data. Model 1, with 10 epochs, shows balanced performance but slightly lower F1 scores compared to the other models. This comparison highlights the trade-off between generalization and overfitting when selecting the number of training epochs.

3.6 4LCNN - Experiment 2: Refining Training Epochs and Batch Size

The experiment continues from the results of the first experiment to determine the optimal combination of training epochs and batch size. The results are summarized below:

model	hidden_kernel	epochs	batch_size	Test_Avg_FScore	Predict_Avg_FScore
4	3	50	64	0.919833	0.949567
5	3	40	32	0.921233	0.9666
6	5	40	32	0.9275	0.933733
7	7	40	32	0.166667	0.166667
8	3	45	32	0.914433	0.898867
9	3	75	64	0.891833	0.931567
10	3	60	32	0.9072	0.916067

Table 6. Training with different numbers of epochs and batch sizes, and a comparison was made between the average F1 score on the test dataset and the average F1 score on the predicted dataset.

It doesn't better than the result in first experiment. So I chose to use the parameters of 4L CNN model 2 as the main parameters of the model trained on the Kaggle platform.

3.7 Confusion Matrix for Training Result Analysis

The Confusion Matrix (Table 7) is a tool used to evaluate the performance of classification models. It is a square matrix that compares the model's predicted results with the actual labels. Each element of the matrix represents the number of instances for different classification outcomes. The structure and relevant metrics of the confusion matrix are explained as follows:

The rows of the confusion matrix represent the actual labels of the model.

The columns of the confusion matrix represent the predicted labels of the model.

Key elements in the confusion matrix include:

TP (True Positive): Correctly classified positive samples.

TN (True Negative): Correctly classified negative samples.

FP (False Positive): Negative samples incorrectly classified as positive.

FN (False Negative): Positive samples incorrectly classified as negative.

In this study, a multi-class confusion matrix is used, which increases the complexity of the structure. As the number of categories increases, the dimensions of the confusion matrix also expand. For example, a three-class problem results in a 3×3 confusion matrix, which contains more elements. In addition to TP, TN, FP, and FN, there may be additional combinations.

Regarding evaluation metrics, while accuracy, precision, and recall can still be used, these metrics are typically calculated for each class and then averaged. In this study, the average F1 Score (Equation 3) across the three categories is used as the primary performaonce evaluation metric.

	Predicted: Rust	**Predicted: Powdery**	**Predicted: Healthy**
Actual: Healthy	FP(H → R)	FP(H → P)	TP (Healthy)
Actual: Powdery	FP(P → R)	TP (Powdery)	FP(P → H)
Actual: Rust	TP (Rust)	FP(R → P)	FP(R → H)

Table 7. This confusion matrix summarizes the prediction results and actual categories for a three-class classification problem. The classes include Rust, Powdery, and Healthy. The table provides the values for True Positive (TP), False Positive (FP), and False Negative (FN) for each class. These values are used to calculate various performance metrics, such as Precision (Equation 1), Recall (Equation 2), and F1 Score (Equation 3).

$$Presion = \frac{TP}{TP + FP}$$

Equation 1. Precision formula

$$Recall = \frac{TP}{TP + FN}$$

Equation 2. Recall formula

$$F1\ Score = \frac{2 * Precision * Recall}{Precision + Recall}$$

Equation 3. F1 Score formula, which represents the harmonic mean of Precision and Recall.

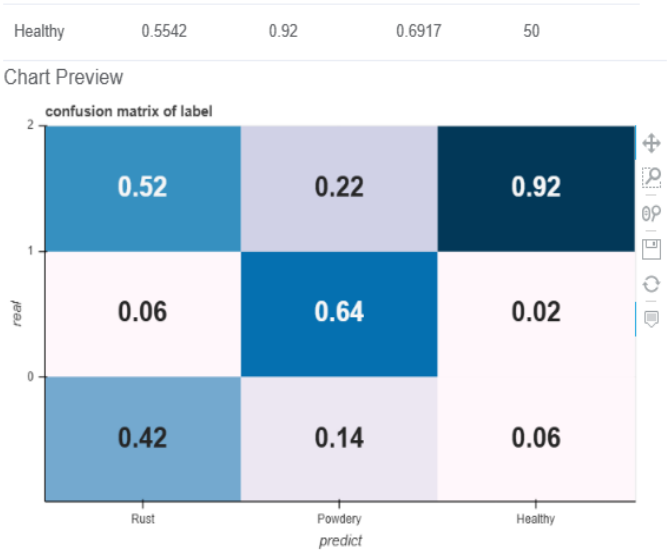
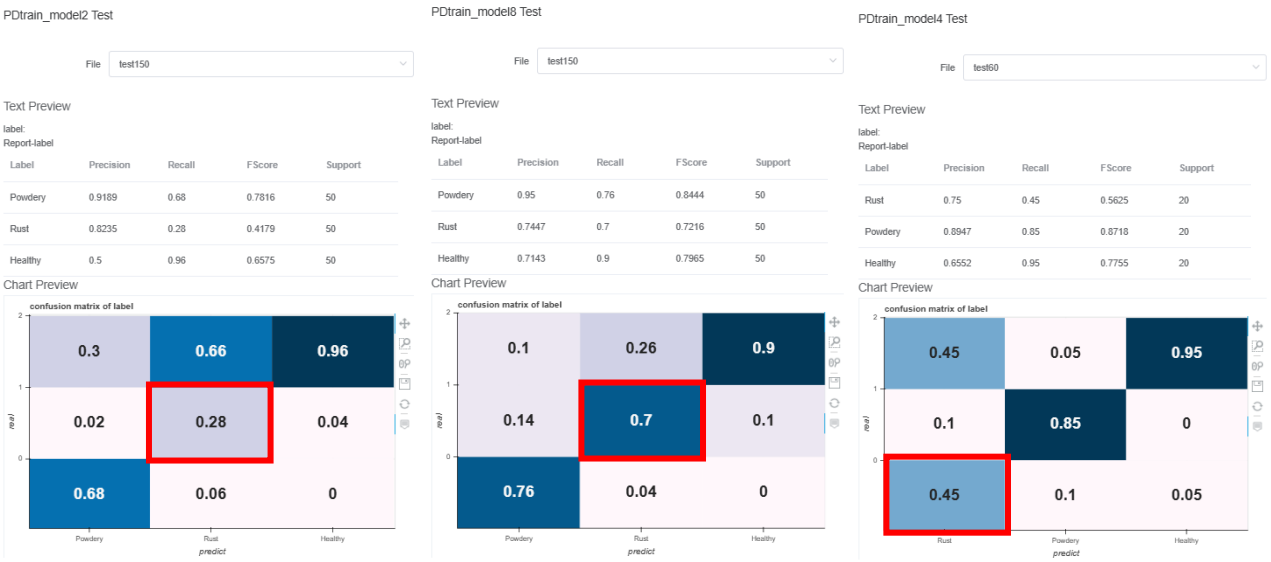


Figure 20. The confusion matrix results demonstrate an example calculation for the Healthy class. The calculated metrics are as follows: ， Recall = 0.92 ， Precision = 0.92 / 0.92+0.22+0.52 = 0.5542 ， F1 Score = 2*0.5542*0.92 / 0.5542+0.92 = 0.6917



(a) (b) (c)

Figure 21. Results from three experiments show that the classification accuracy for Rust is significantly lower on two layer CNN training by GPU 1080Ti. (a) presents the test set recognition results for Model 2, (b) shows the test set recognition results for Model 8, and (c) displays the validation set recognition results for Model 4.

3.8 Accuracy on Train and Test

Using `matplotlib.pyplot`, a chart can be plotted with the x-axis representing epochs and the y-axis representing accuracy, illustrating the changes in model accuracy as the number of training epochs progresses.

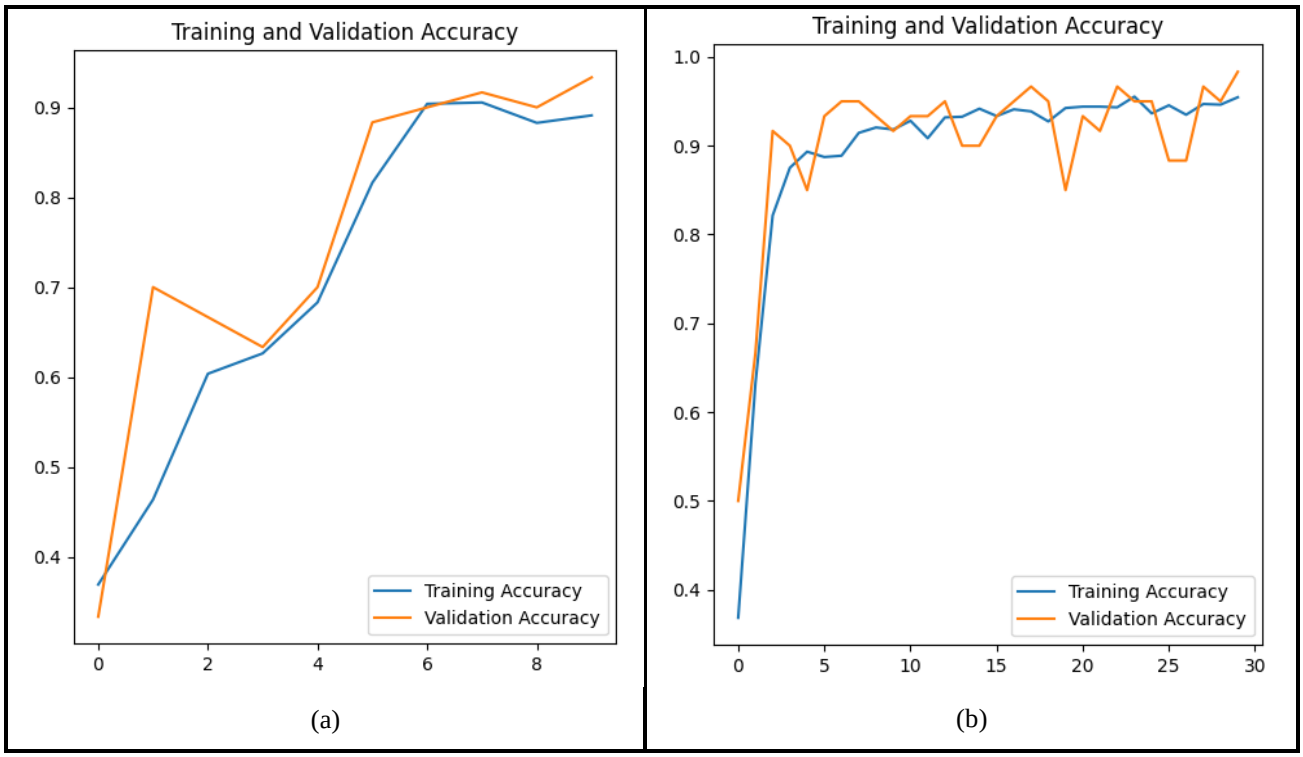


Figure 22. (a) shows epochs=10 and (b) shows epochs=30

Next, a comparison is made between model without Gaussian noise(Figure 22(a)) and with Gaussian noise(Figure 22(b)). The results show that the model with Gaussian noise achieves higher accuracy, demonstrating that incorporating Gaussian noise as part of data augmentation in preprocessing effectively improves recognition accuracy.

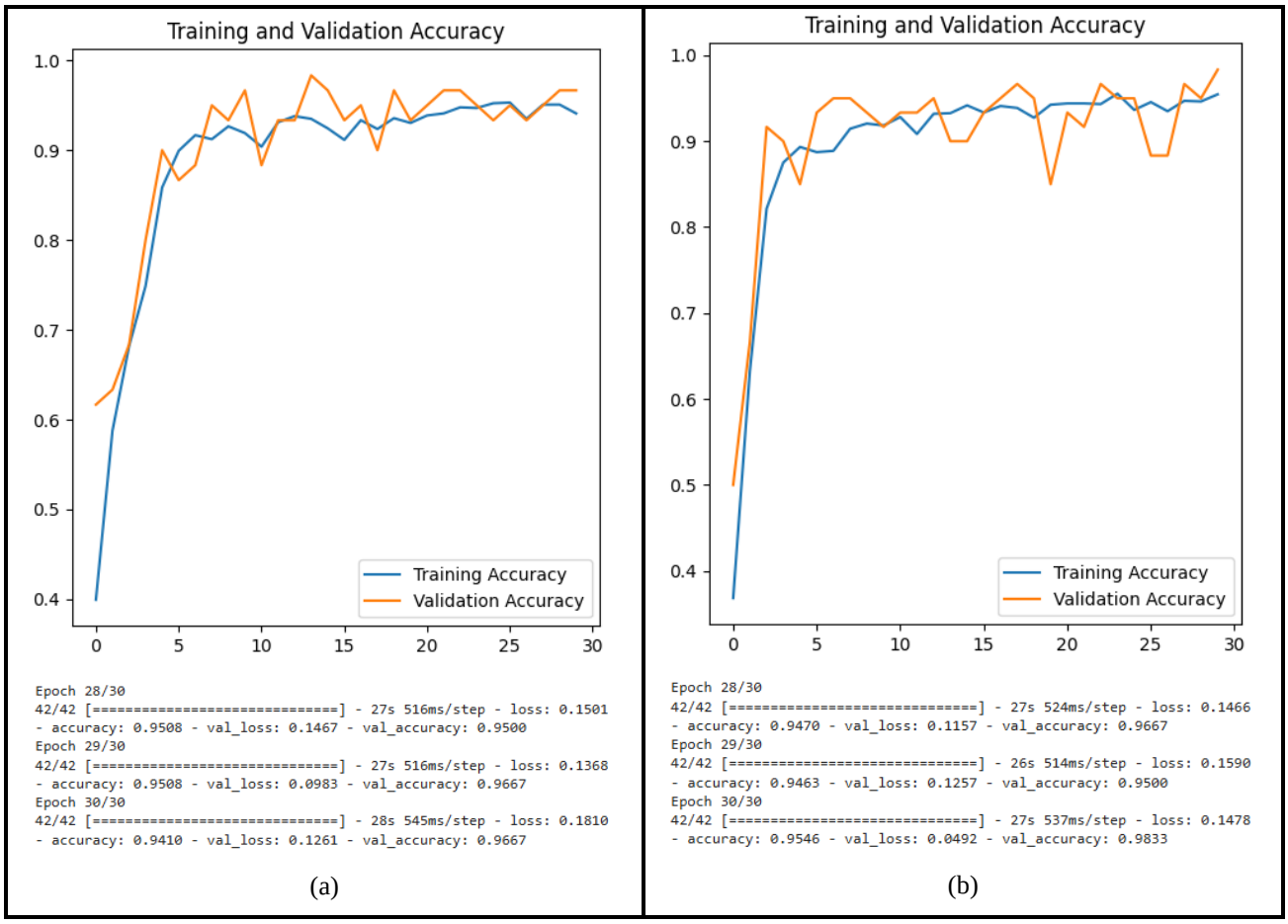


Figure 23. (a) model without Gaussian noise and (b) model with Gaussian noise

3.9 Image Predictions on Test Data

The Figure 24. displays 16 images from the test dataset, each annotated with the actual class, predicted class, and the prediction confidence generated by the model. The code uses a custom prediction function to determine the class and confidence level for each input image. The images are organized into a 4×4 grid, showcasing predictions for three categories: Rust, Healthy, and Powdery.

The model's performance is visually represented, demonstrating high accuracy with confidence levels nearing 100% for most predictions. Correct classifications are observed for various leaf conditions, with clear indications of the model's ability to distinguish between disease categories and healthy leaves. This visualization effectively validates the model's performance and highlights its potential for real-world applications.

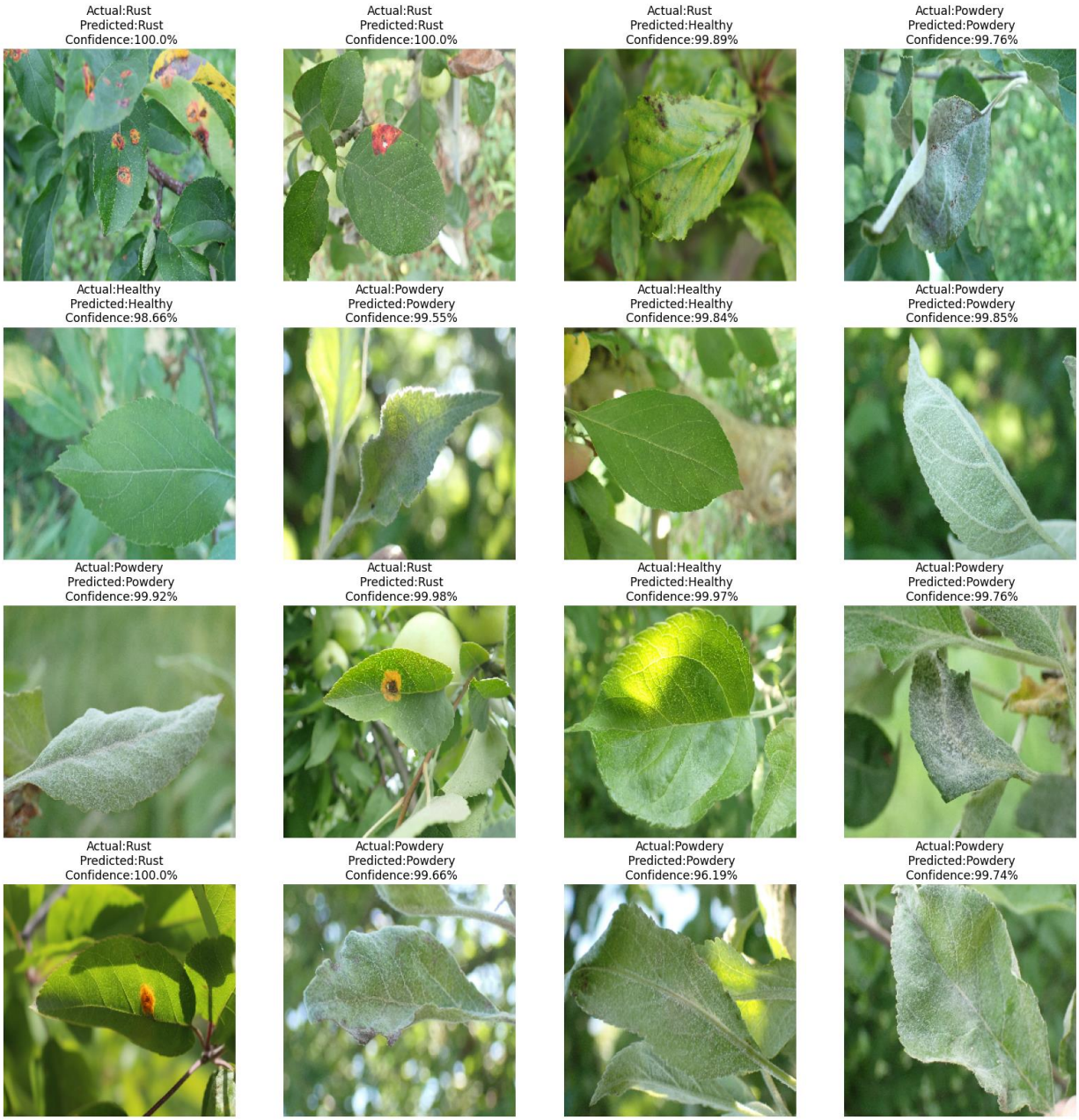


Figure 24. Visualization of Model Predictions on Test Dataset

4. CONCLUSIONS

This study demonstrates the effective application of convolutional neural networks (CNNs) for classifying plant leaf conditions into three categories: Healthy, Powdery Mildew, and Rust. A systematic exploration of various parameters, including input resolution, convolutional kernel size, activation functions, dropout rates, and batch sizes, revealed optimal configurations that enhance model accuracy. Preprocessing techniques such as image resizing, data augmentation, and Gaussian noise addition significantly improved the model's generalization capabilities, robustness and accuracy.

The findings highlight the potential of deep learning in addressing the challenges of automated plant disease classification. This work serves as a foundation for further advancements in model design and preprocessing methods, contributing to the development of reliable, scalable, and efficient systems for agricultural disease management. Future research could explore integrating more complex architectures and diverse datasets to further improve classification performance and practical applicability.

Data Availability Statement:
All the codes, datasets and files can be viewed and downloaded via the links : Digital Image Processing Final Project

REFERENCES

[1] Inanalysis Platform https://github.com/inanalysis
[2] Kaggle Dataset: Plant disease recognition dataset https://www.kaggle.com/datasets/rashikrahmanpritom/plant-disease-recognition-dataset
[3] Murphy KM, Ludwig E, Gutierrez J, Gehan MA. Deep Learning in Image-Based Plant Phenotyping. Annu Rev Plant Biol. (2024) Jul;75(1):771-795. Epub 2024 Jul 2. PMID: 38382904. https://doi.org/10.1146/annurev-arplant-070523-042828
[4] Alzubaidi, L., Zhang, J., Humaidi, A.J. et al. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. J Big Data 8, 53 (2021). https://doi.org/10.1186/s40537-021-00444-8