# Cover Page:

Contents:

[Cover Page: 1](#_Toc163843359)

[Contents: 2](#_Toc163843360)

[Abstract: 2](#_Toc163843361)

[Acknowledgement: 4](#_Toc163843362)

[List of Figures: 5](#_Toc163843363)

[List of Tables: 6](#_Toc163843364)

[List of Abbreviations: 7](#_Toc163843365)

[Introduction: 8](#_Toc163843366)

[Background: 8](#_Toc163843367)

[Objectives: 9](#_Toc163843368)

[Limitations: 10](#_Toc163843369)

[Literature Review: 12](#_Toc163843370)

[Methodology: 15](#_Toc163843371)

[Introduction: 15](#_Toc163843372)

[Dataset: 15](#_Toc163843373)

[Software and Hardware Specifications: 16](#_Toc163843374)

[Deep Learning Architectures: 17](#_Toc163843375)

[Deep Learning Optimizers: 25](#_Toc163843376)

[Training Specifications: 25](#_Toc163843377)

[Code implementation and explanation: 27](#_Toc163843378)

[Results: 33](#_Toc163843379)

[Introduction to Results: 33](#_Toc163843380)

[Training Progress: 34](#_Toc163843381)

[Training, Validation, and Testing Performance: 36](#_Toc163843382)

# Abstract:

Recently, various state-of-the-art deep learning (DL) algorithms have been used to classify plant diseases on publicly available/author-generated datasets. In this research, a deep learning-based comparative analysis was proposed for plant disease classification. First, a comparative analysis of well-known CNN architectures was proposed. The research found a convolutional neural network (CNN) a best Second, which attempted to improve the performance of the realized model by training on various deep learning optimizers have been developed. The comparison between different CNNs was based on performance metrics such as validation accuracy/loss, F1 score, and number of required epochs All the selected DL architectures were trained on the PlantVillage dataset with 26 different diseases in 14 plant species. Keras with a TensorFlow backend was used to train the deep learning algorithm. It is concluded that the Xception architecture trained with Adam optimizer obtained the highest verification accuracy and F1-score of 99.87% and 0.9978, respectively, better than the previous methods and indicating the novelty of the work Therefore, the method proposed in this study can be applied in other agricultural applications to identify its explicit classification.

# Acknowledgement:

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Bappaditya Mandal, for his continuous support and guidance throughout my BSc project. This project would not have been possible without his direction and constant assistance.

Secondly, I would like to thank Ash leake for his technical support and for providing timely resource arrangement.

Finally, I would like to thank my family for continually believing in me. They have always encouraged me never to give up in pursuing my goals. Without their love and presence, the successes as well as the failures would have been meaningless.

# List of Figures:

[Figure 1.EfficientNetB3-Architecture 18](#_Toc163843315)

[Figure 2.MobileNetV1-Architecture 19](#_Toc163843316)

[Figure 3.ResNet50-Architecture 20](#_Toc163843317)

[Figure 4.VGG16-Architecture 21](#_Toc163843318)

[Figure 5.Xception-Architecture 22](#_Toc163843319)

[Figure 6. The methodology of this research. 24](#_Toc163843320)

[Figure 7. Some of the plant diseases from the PlantVillage dataset. 24](#_Toc163843321)

[Figure 8. Data Splitting - Code Snippet 27](#_Toc163843322)

[Figure 9. Data Augmentation - Code Snippet 29](#_Toc163843323)

[Figure 10. Custom Callback - Code Snippet 32](#_Toc163843324)

# List of Tables:

[Table 1.Architecture Metrics 23](#_Toc163843334)

[Table 2. Hyperparameters of the deep learning optimizers 26](#_Toc163843335)

[Table 3.Training Time Comparison 35](#_Toc163843336)

[Table 4. Training Metrics 37](#_Toc163843337)

[Table 5.Validation Metrics 39](#_Toc163843338)

[Table 6. Testing Metrics 41](#_Toc163843339)

# List of Abbreviations:

# Introduction:

## Background:

Agriculture is considered to play an important role in sustaining human life and ensuring food security around the world. With the global population projected to reach 9 billion by 2050, there is increasing pressure to increase agricultural productivity and reduce challenges such as plant diseases. Plant diseases have economic and environmental impacts, leading to crop losses, reduced yields, and increased use of pesticides.

Traditional approaches to diagnosis and management are often relied upon, which can be time-consuming, subjective, and limited in feasibility. Modern technology combinations, especially computer vision and machine learning, are seen as promising solutions to address these challenges. By using big data and advanced algorithms, automated systems for rapid, accurate diagnosis and targeted therapy in plants can be developed.

The availability of datasets such as the Plant Village dataset has changed the landscape of plant disease research. The Plant Village dataset contains a high-quality collection of images showing a variety of plant diseases including fungi, viruses, and pathogens as well as samples of healthy plants. These labelled images serve as valuable training data for training and evaluation of machine learning models, especially convolutional neural networks (CNNs).

The use of pre-existing CNN architectures such as ResNet, VGG, and Inception is seen to provide a structured approach to modelling, allowing researchers to focus on optimizing and generalizing model performance. It also gives flexibility and improves convergence speed during training.

## Objectives:

This study aims to evaluate and compare the performance of various pre-existing convolutional neural network (CNN) architectures for accurate plant disease classification using the Plant Village dataset. The objective of the study is to aid insights in the selection of optimal models to optimize agricultural applications. To achieve this objective, the following objectives will be pursued.

First, several CNN algorithms, including ResNet, VGG, and Inception etc, will be trained on the Plant Village dataset. Standardized preprocessing techniques and hyperparameters will be used to ensure the accuracy of the models. The trained models will then be evaluated and compared based on their classification accuracy, robustness, and computational efficiency.

Second, various optimization algorithms such as Adamax, Adadelta, and RMSProp will be used to evaluate their impact on model convergence performance and trade-off analysis between convergence speed and accuracy. The results obtained by each optimizer will be analysed to understand their effectiveness in training CNN models for classifying plant diseases.

Third, model performance will be evaluated using standard metrics including accuracy, precision, recall, and F1-score. The generation and analysis of confusion matrices will provide insight into the ability of different models to correctly identify different plant diseases and will further contribute to more detailed studies of classification efficiencies.

Then, the study aims to identify and compare the best-performing CNN designs and optimization algorithms based on established evaluation metrics and criteria. This requires intensive research to understand the strengths and limitations of each model in a real-world agricultural setting.

Finally, research findings will be discussed in terms of their practical implications for agricultural stakeholders. The potential benefits of applying advanced machine learning techniques to plant disease diagnosis and management will be highlighted, as well as suggestions for new research directions such as increasing the quality of data collection, sample preparation techniques, and potential applications in precision agriculture.

## Limitations:

This research project has several limitations that define the context and the boundaries of the study. First, the reliance on plant village dataset for model training and analysis poses limitations on dataset representativeness. Although the list covers a wide range of plant diseases, it may not include all possible variations and environmental factors encountered in actual agricultural conditions. These limitations require caution when attributing outcomes are moving into broader agricultural contexts or diversity contexts.

Second, computer hardware plays an important role in the size and complexity of the experiments conducted in this study. Limited technical capacity may prevent analysis of large datasets, complex model architectures, or detailed hyperparameter tuning, which may affect the depth of analysis and optimization achieved.

Furthermore, the choice of the overall findings are specific choices related to CNN architectures, optimization algorithms, dataset preprocessing techniques, although these choices are made deliberately to maintain the accuracy of the experiment, they can hinder limit the applicability of the results to other models or data structures.

Furthermore, time constraints of the experimental timeline may limit sample replication, thorough sensitivity assessment, or long-term performance monitoring These time constraints may affect thorough sample analysis and finding alternative routes or methods.

Finally, external factors such as changes in disease prevalence, changing environmental factors, or extracurricular technological advances may affect the relevance and consumption of research findings in the long run.

# Literature Review:

Plant pathologists classify plant diseases by examining several plant sections, such as the root, kernel, stem, and leaf. In a report he presented, Ferentinos used various categorization systems for plant diseases. High accuracy disease architectures are designed by CNN models. Researchers Brahimi et al. examined 14828 photos of tomato leaves with nine distinct illnesses on them. 99.18% accuracy was achieved by the CNN model with the help of isolated sick leaf patches and a large enough dataset. Plant pests may seriously hurt crops and cost farmers a lot of money. In his study, Dawei et al. employed transfer learning to increase the precision of pest identification and showed encouraging outcomes. Singh and others. The suggested classifier outperformed transfer learning techniques (AlexNet, ResNet, VGG16, Inception V3) and other machine learning algorithms (SVM, Decision Tree, Logistic Regression, and KNN), achieving 96.46% accuracy. Utilizing a region-based CNN model, Hasan et al. estimated wheat output with an average accuracy ranging from 88% to 94%. The model was used to calculate the production yield of several wheat cultivars. Patil and Bodhe collected sugarcane leaf shape features in order to identify plant diseases. Using threshold segmentation and triangle segmentation, they were able to identify the leaf and lesioning region with 98.6% accuracy. In a research study titled "Using deep learning for image-based potato tuber disease detection," Oppenheim et al. employed a CNN model to classify potato diseases using computer vision. Oppenheim et al. divided tubers into five groups using the CNN technique. In a study report published in 2016, Barbedo, Jiang, and colleagues determined the image processing research needs for autonomously diagnosing plant illnesses. Jiang et al. [17] presented the real-time apple leaf disease classification method based on rainbow concatenation and the VGGInception model. Using the Apple Leaf Dataset, the suggested method demonstrated a 97.14% test accuracy. The suggested method fared better than other well-known pre-trained models. In Atole et al.'s research publication [18], rice plants were classified into three classes with an accuracy of 91.23% utilizing a pretrained AlexNet deep network. In a research study, A. P. Singh demonstrated how to use an artificial bee colony for feature selection in order to identify the ideal feature set needed for a support vector machine classifier to identify diseases in photos of grape plants. The accuracy of the suggested algorithm was 92.14%. For the purpose of classifying plant species, Zhu et al. used Inception V2 with batch normalization and found that this method produced results that were more accurate than Faster RCNN. Zhang put out a categorization system that he called varied region-based CNN, which is able to extract features and encode context-aware representation. Image segmentation and image classification are the two steps in the two-step method for plant leaf disease identification that Nanehkaran et al. proposed. It was suggested to segment images using hue, saturation, and intensity. In order to attain greater accuracy, the segmented images were classed using the CNN model. Using the deep learning model R-CNN, W. Shaohua and S. Goudos (2020) exhibited faster and more accurate object detection. Rani et al. introduced D-Leaf, a deep learning-based technique for identifying plant species. Machine learning approaches were employed to handle the task of picture classification, while pre-trained models were utilized for feature extraction. The accuracy of the D-leaf model using the ANN classifier was 94.88%. Zhao et al. presented an image classification strategy for improving the Support Vector Machine (SVM) model's hyper-parameters using an enhanced artificial bee colony nature-inspired optimization algorithm. Particle swarm optimization (PSO-SVM) and Genetic algorithm (GA)—SVM techniques were examined for a comparative analysis. With an accuracy of 98.28%, the suggested technique outperformed alternative comparable methods for the classification of hyperspectral pictures. Algorithms inspired by nature employ practical optimization techniques. Genetic optimization, three-parent genetic optimization, parallel three-parent genetic algorithm, artificial bee colony, particle swarm optimization, cuckoo search, ant colony optimization, big bang big crunch, parallel big bang big crunch, and other popular nature-inspired optimization techniques are examples. Numerous fields have successfully implemented deep learning techniques. Gao et al. proposed IDONET, an intrusion detection system that uses improved mask R-CNN for power grid surveillance. The suggested model outperformed the other cutting-edge object identification models when it came to small item detection after being trained for it. A method to classify photos of rice blast, brown mark disease, and bacterial blight (a total of 1500 images) with 94% accuracy was proposed by Bhattacharya et al. This work presents a deep learning-based approach whose hyper-parameters can be adjusted through the use of optimization algorithms inspired by nature.

# Methodology:

## Introduction:

Convolutional Neural Networks (CNNs) have been widely utilized for image classification tasks, prompting this research to investigate the performance of several novel CNN architectures for classifying plant diseases. All 5 DL architectures considered for this study were recently published in large research articles on plant disease classification, as depicted in Figure 1.

The overall methodology employed in this study was to train the 5 DL architectures on the Plant Village dataset using 3 different optimizers for each. The convergence of the 5 CNN architectures to the final training/validation values was observed to update the hyperparameters. Subsequently, the CNN models were compared in terms of training, validation, and testing accuracy/loss, as well as F1-score. This comparison led to the utilization of DL optimization algorithms to further enhance the performance of their CNN architectures, resulting in the highest F1 score in their specific category.

## Dataset:

All DL models were trained on the publicly available Plant Village dataset, which contains 54,306 images with 38 different healthy/diseased leaves associated with their 14 plant species (some of the plant diseases are shown in Figure 3), image size Converted to 224 × 3 and normalized by dividing the pixels values ​​by 255 to match the initial values ​​of the models To avoid overfitting, 70%, 20% and 10% data sets were allocated of the training, validation and test data sets, respectively.

## Software and Hardware Specifications:

The software environment for this research work is developed using Python 3.11.4 in compatible operating systems such as Linux, Windows, or macOS. The primary integrated development environment (IDE) used for coding and testing is Google Colab and Kaggle. These environments provide a robust platform for the development and successful implementation of machine learning models.

This research relies on several core libraries and resources for data binding, deep learning model development, and visualization. System libraries such as os, time, shutil, and pathlib facilitate file management, time-sensitive operations, and access management in the operating system. For data processing and transformation, the OpenCV library (cv2) is used for graphical representation, while numpy and pandas are used for numerical calculations and data analysis tasks, respectively.

Deep learning functions are implemented using TensorFlow version 2.9.1 and Keras libraries. TensorFlow provides the backbone for building and training deep learning models, while Keras provides a high-level API for building simple neural networks The Keras.preprocessing.image module supports image data processing and enhancement through the ImageDataGenerator class. Various layers (e.g., Conv2D, MaxPooling2D, Dense) and optimizers (e.g., Adamax, RMSprop, Adadelta) from Keras are used for model architecture and optimization.

The software stack also includes tools for model evaluation and visualization. The sklearn library is used for classification reports, and model evaluation metrics. In addition, data visualization is facilitated by the use of seaborn and matplotlib.pyplot to create informative plot graphs.

All experiments were performed on a graphical processing unit (2 x NVIDIA T4) with specifications: 16GB GDDR6 memory, 2560 CUDA cores and 320 Tensor cores, 1590 MHz core clock, and 320 GB/sec memory bandwidth.

## Deep Learning Architectures:

Following the creation of the AlexNet architecture, a groundbreaking era of cutting-edge CNN architectures was initiated for several image categorization applications. Thus, we examined well-liked and effective CNN models including VGG-16, ResNet-50, MobileNet, EfficientNetB3, and Xception in this article.

Using information taken from the Keras website, the study provides a thorough analysis of a number of well-known convolutional neural network (CNN) models, including EfficientNetB3, MobileNet, ResNet50, VGG16, and Xception. The features of every model are examined to offer a thorough comprehension of their performance in relation to different metrics.

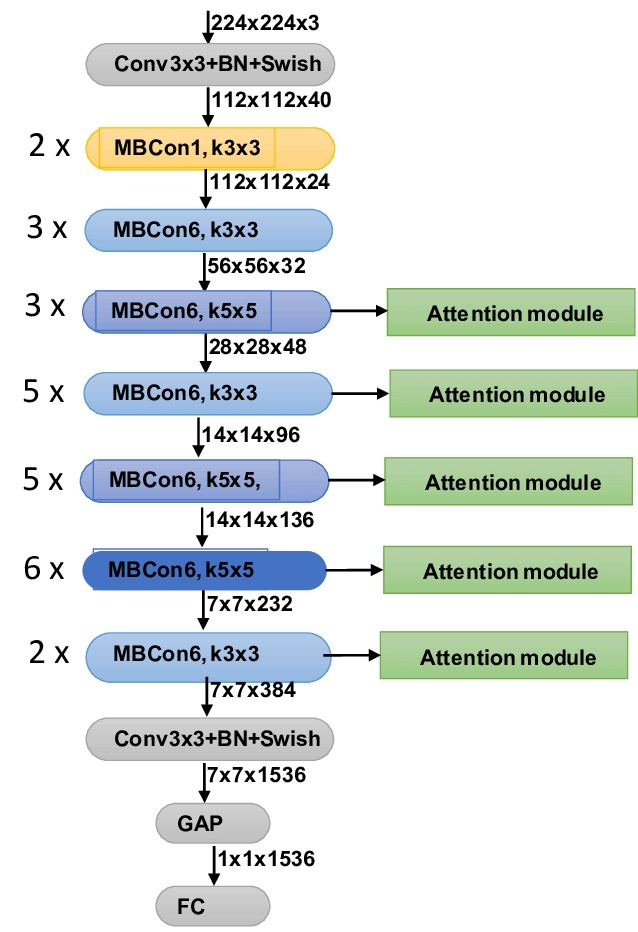
The EfficientNetB3 architecture is renowned for its effectiveness in striking a balance between model size and performance. It is composed of several convolutional layer blocks, each of which is followed by activation functions and batch normalization. A stem convolutional layer is the first layer in the network, and it uses the input images to extract fundamental information. Then, a sequence of blocks comprised of inverted bottleneck layers, squeeze-and-excitation modules, and depth wise separable convolutions are stacked to capture progressively more complicated characteristics. Lastly, fully connected layers and global average pooling are used to carry out classification tasks. Because of its focus on scalability and parameter efficiency, EfficientNetB3 is an excellent choice for environments with limited resources. EfficientNetB3 stands out because to its moderate size of 48 MB. It achieves impressive performance with a top-1 accuracy of 81.6% and a top-5 accuracy of 95.7%, while having a comparatively compact architecture. The number of parameters in the model—12.3 million—shows how well it uses computational power. Furthermore, its significant depth of 210 layers demonstrates its ability to extract complex features from the data. EfficientNetB3 exhibits efficiency in inference time, requiring 140.0 milliseconds on CPU and 8.8 milliseconds on GPU for each inference step.

Figure 1.EfficientNetB3-Architecture

A lightweight CNN architecture called MobileNetV1 was created for embedded and mobile devices. It has depth wise separable convolutions, which reduce computational complexity and model size by splitting typical convolutions into distinct depth wise and pointwise convolutions. A sequence of convolutional layers precedes depth-wise separable convolutional blocks in the network's architecture. To effectively capture spatial and channel-wise information, pointwise convolutional layers are inserted across these blocks. The last two layers are a fully connected layer for classification and global average pooling. MobileNetV1 is suited for real-time applications on systems with limited resources because of its architecture, which places a high priority on computational efficiency while retaining competitive accuracy. Even with its significantly reduced size of 16 MB, MobileNet shows good performance metrics. Despite its decreased complexity, its top-1 accuracy of 70.4% and top-5 accuracy of 89.5% demonstrate its competence. Combining computational efficiency and performance, MobileNet has a low parameter count of 4.3 million and a deep layer count of 55. Its acceleration to 22.6 milliseconds per step on the CPU and GPU further confirms that it is suitable for contexts with limited resources.

A diagram of a computer

Description automatically generated

Figure 2.MobileNetV1-Architecture

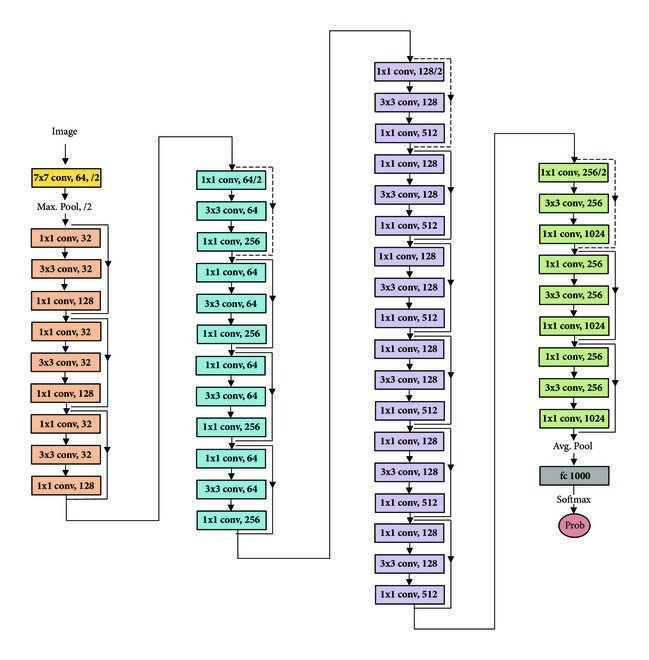
Renowned for its residual learning framework, ResNet50 is a deep CNN architecture that helps to train very deep networks by mitigating the vanishing gradient issue. The network consists of numerous residual blocks, each of which has identity shortcuts after a number of convolutional layers. By facilitating the gradient flow during training, these shortcuts enable deep network optimization that is both efficient and effective. A stem convolutional layer and successive blocks of residual units are the architecture's foundation. The last two layers are a fully connected layer for classification and global average pooling. Because to ResNet50's architecture, deep networks with hundreds of layers may be trained with effective optimization and top-notch performance. ResNet50 has a bigger, 98 MB architecture and is more reliable. Its 74.9% top-1 accuracy and 92.1% top-5 accuracy are praiseworthy, but they come with a hefty 25.6 million parameter count. Nonetheless, the model's 107 layers of depth enable it to identify intricate patterns in the data. ResNet50 shows competitive inference times despite its high processing requirements, needing 4.6 milliseconds on GPU and 58.2 milliseconds on CPU each step.

Figure 3.ResNet50-Architecture

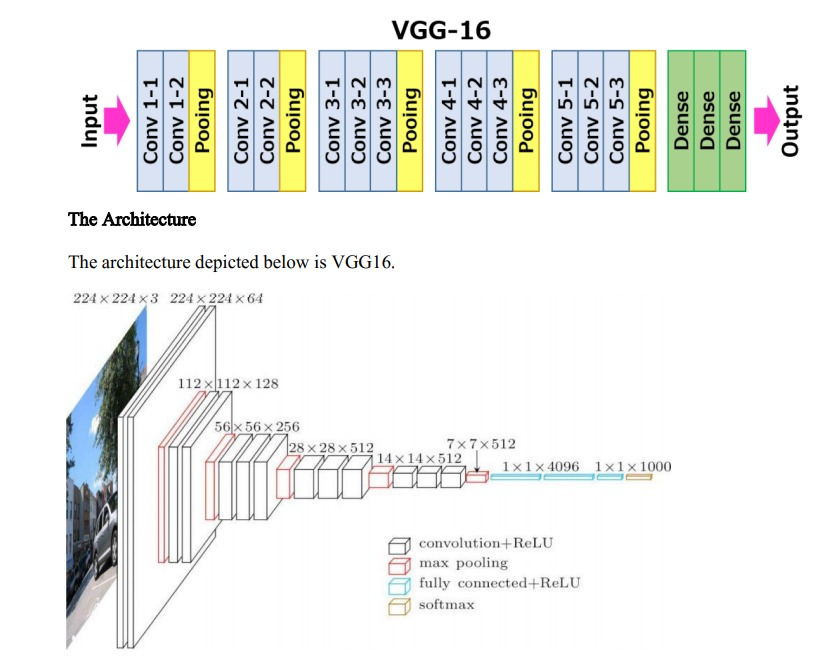
The traditional VGG16 architecture is renowned for its simplicity. It is composed of several convolutional layers, then layers for spatial down sampling called max pooling. Repeated blocks of convolutional layers, which get deeper as the network grows, define the architecture of the network. VGG16 is made up of five convolutional blocks, each of which has a max-pooling layer after two or three convolutional layers. The completely connected categorization layers are among the last layers. Because of its simple yet effective architecture, VGG16 is a well-liked option for a variety of computer vision applications. On the other hand, VGG16 is notable for having a large number of parameters—138.4 million—and a large model size (528 MB). Although its top-5 accuracy of 90.1% and top-1 accuracy of 71.3% are impressive, they come at the cost of processing power. Compared to its competitors, VGG16 has a comparatively shallow design with a depth of 16 layers. Its inference time, which requires 4.2 milliseconds on the GPU and 69.5 milliseconds on the CPU per step, is still competitive.

Figure 4.VGG16-Architecture

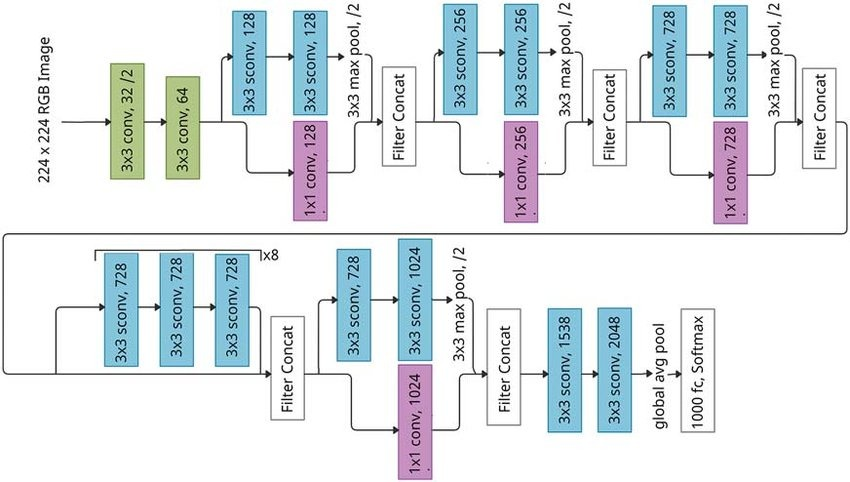
Xception is an expansion of the Inception architecture, which separates spatial and channel-wise convolutions using depth wise separable convolutions. The goal of this architecture is to reduce both the size of the model and computational complexity while capturing complicated features. Each depth wise separable convolutional block in Xception is followed by activation functions and batch normalization. The network is composed of numerous depth wise separable convolutional blocks stacked on top of a stem convolutional layer. Global average pooling and fully connected categorization layers are the last layers. Comparing Xception's architecture to conventional convolutional architectures, the latter require more parameters to reach state-of-the-art performance. Xception provides a balance of efficiency and performance, even at 88 MB in size. It is positioned well among its contemporaries thanks to its top-1 accuracy of 79.0% and top-5 accuracy of 94.5%. With 81 layers and 22.9 million parameters, Xception is able to reconcile computational efficiency with model complexity. It’s fit for real-time applications is demonstrated by its inference time of 8.1 milliseconds on the GPU and 109.4 milliseconds per step on the CPU.

Figure 5.Xception-Architecture

In summary, the study carefully analyses each CNN model's features, highlighting its advantages and disadvantages in terms of several performance measures and the reasons for which they were chosen for this research.

Table 1.Architecture Metrics

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Size (MB)** | **Top-1**  **Accuracy** | **Top-5**  **Accuracy** | **Parameters** | **Depth** | **Time (ms) per inference step (CPU)** | **Time (ms) per**  **inference step (GPU)** |
| [EfficientNetB3](https://keras.io/api/applications/efficientnet/#efficientnetb3-function) | 48 | 81.6% | 95.7% | 12.3M | 210 | 140.0 | 8.8 |
| [MobileNet](https://keras.io/api/applications/mobilenet) | 16 | 70.4% | 89.5% | 4.3M | 55 | 22.6 | 22.6 |
| [ResNet50](https://keras.io/api/applications/resnet/#resnet50-function) | 98 | 74.9% | 92.1% | 25.6M | 107 | 58.2 | 4.6 |
| [VGG16](https://keras.io/api/applications/vgg/#vgg16-function) | 528 | 71.3% | 90.1% | 138.4M | 16 | 69.5 | 4.2 |
| [Xception](https://keras.io/api/applications/xception) | 88 | 79.0% | 94.5% | 22.9M | 81 | 109.4 | 8.1 |

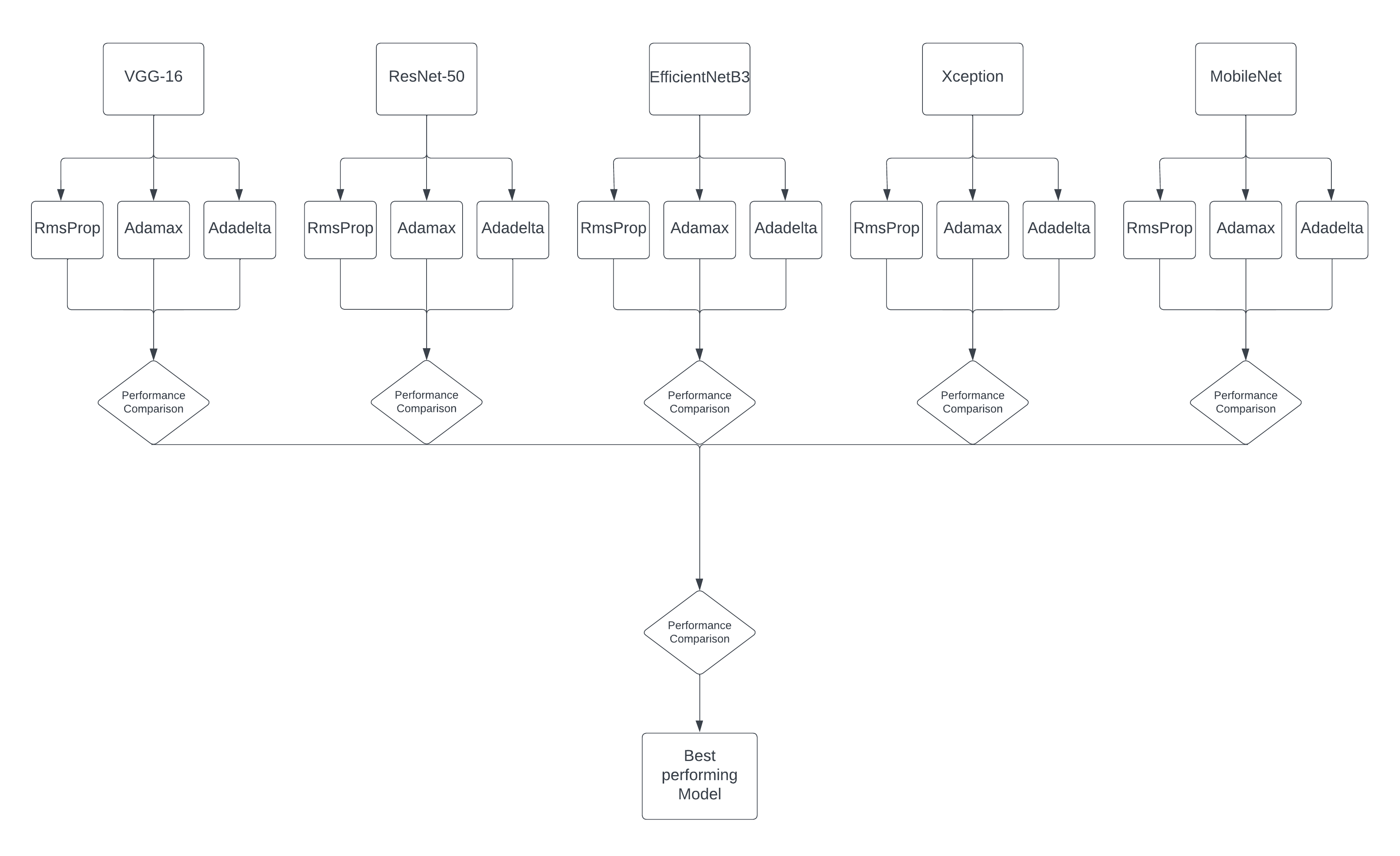


Figure 6. The methodology of this research.

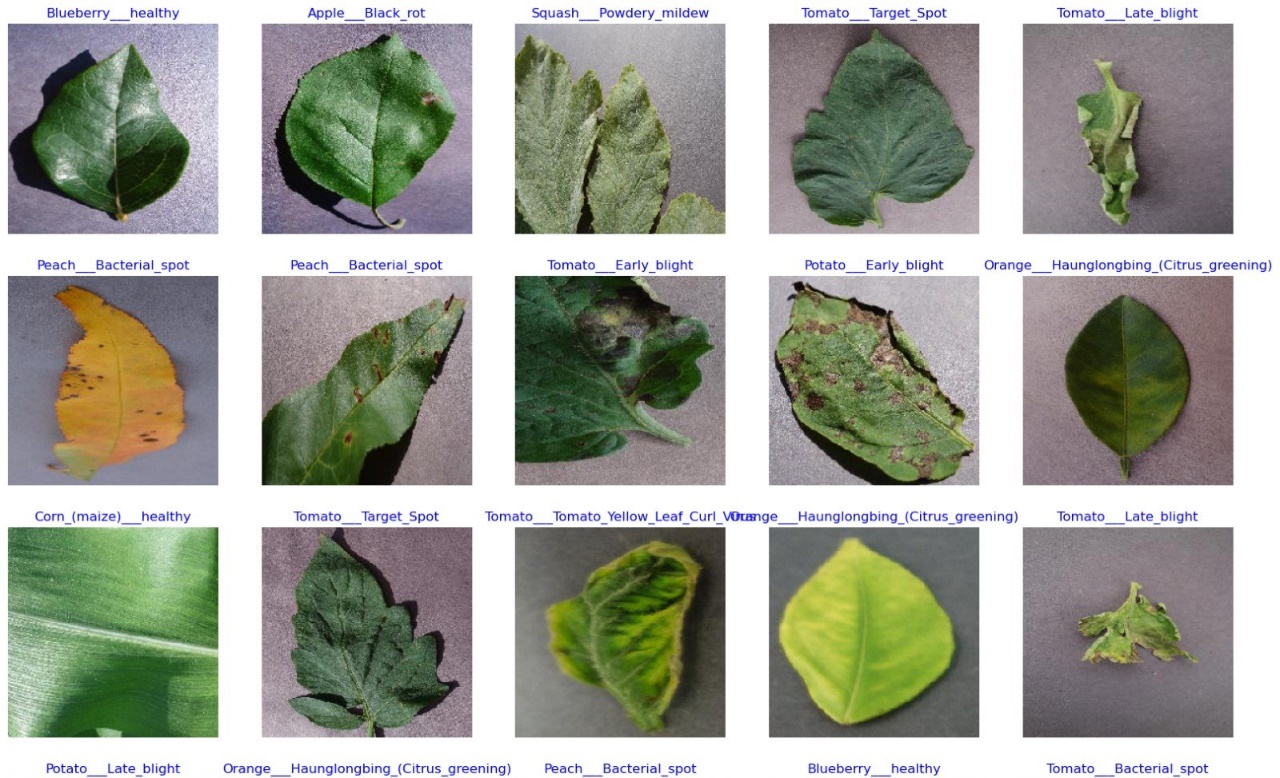


Figure 7. Some of the plant diseases from the PlantVillage dataset.

## Deep Learning Optimizers:

All the DL models were trained using 3 state-of-the-art deep learning optimizers. Some characteristics of these optimizers are provided as follows:

1. RMSProp: Learning rates were adjusted individually for each parameter during training. This was achieved by maintaining a moving average of squared gradients for each parameter, thus normalizing updates. This adaptability enabled faster convergence and stability in training deep neural networks.
2. Adadelta: Adadelta, an extension of the Adagrad algorithm, further addressed its tendency to decrease the learning rate monotonically. This was accomplished by using a sliding window of the squared gradients to compute adaptive learning rates, allowing for more stable and efficient training.
3. RMSProp: Adaptive learning rates for each parameter were computed by maintaining a moving average of the squared gradients. This normalization of learning rates allowed for more stable and efficient training of deep neural networks.

## Training Specifications:

All DL models were trained from scratch on the PlantVillage dataset. The hyperparameters were tuned using the random search method. The internal covariate shift problem was observed in neural networks due to variations in the input data distribution caused by changes in the number of parameters in the previous layers. This problem was addressed using Batch Normalization, a method known for its effectiveness in handling the high learning rate. The ReLU activation function was employed to train all DL models, chosen for its computational efficiency and its ability to mitigate the vanishing gradient problem. The specifications of all DL optimizers are summarized in Table 1.

Table 2. Hyperparameters of the deep learning optimizers

|  |  |
| --- | --- |
| Optimizers | Specifications |
| Adamax | Learning rate = 0.002, beta1 = 0.9, beta2 = 0.999, epsilon = 1 \* 10-8 |
| Adadelta | Learning rate = 1.0, rho = 0.95, epsilon = 1 \* 10-6 |
| RMSProp | Learning rate = 0.001, rho = 0.9, epsilon = 1 \* 10-7 |

## Code implementation and explanation:

#### Data Preparation and Splitting:

A screen shot of a computer program

Description automatically generated

Figure 8. Data Splitting - Code Snippet

1. define\_paths(data\_dir)

* This function takes the directory path where the dataset is stored as input.
* It iterates through each fold (subdirectory) in the dataset directory and collects file paths and their corresponding labels.
* File paths and labels are stored in separate lists (file paths and labels).
* Returns the list of file paths and corresponding labels.

2. define\_df (files, classes)

* This function takes two lists as input: files (list of file paths) and classes (list of corresponding labels).
* It creates two pandas Series objects: Fseries for file paths and Lseries for labels.
* Then it concatenates these two Series along the columns axis to form a DataFrame.
* Returns the DataFrame containing file paths and labels.

3. split\_data (data\_dir)

* This function orchestrates the process of splitting the dataset into training, validation, and testing sets.
* It utilizes the previously defined functions define\_paths () and define\_df () to obtain the DataFrame containing file paths and labels.
* Firstly, it creates the training DataFrame by splitting the entire dataset into 80% for training and 20% for validation and testing combined.
* Then, it splits the remaining data (validation and testing combined) equally into validation and testing DataFrames, each containing 10% of the original dataset.
* Stratified splitting is applied to maintain the distribution of classes across splits.
* Returns three DataFrames: train\_df for training, valid\_df for validation, and test\_df for testing.

#### Data Augmentation and Preprocessing using Image Data Generators:

A screen shot of a computer program

Description automatically generated

Figure 9. Data Augmentation - Code Snippet

Function: create\_gens (train\_df, valid\_df, test\_df, batch\_size)

This function prepares data generators for training, validation, and testing data. It utilizes the Keras ImageDataGenerator to perform data augmentation and preprocessing.

1. Defining Model Parameters:

* img\_size: The desired size for input images after resizing.
* channels: The number of color channels in the images (e.g., 3 for RGB).
* color: The colour mode for images ('RGB' or 'grayscale').
* img\_shape: The shape of input images (height, width, channels).

1. Customizing Test Batch Size:

* It calculates a suitable batch size for the test data based on the length of the test DataFrame to ensure efficient memory usage.
* The batch size is dynamically adjusted to fit the test data while keeping it within a reasonable range.

1. Defining Data Augmentation Function:

* A custom function scalar(img) is defined to perform minimal preprocessing on images. In this case, it simply returns the input image unchanged.

1. Creating ImageDataGenerators:

* Two ImageDataGenerator objects (tr\_gen for training and ts\_gen for validation and testing) are instantiated.
* tr\_gen is configured with data augmentation parameters such as horizontal flipping to augment training data.
* ts\_gen is used for validation and testing and does not perform data augmentation.

1. Flowing Data from DataFrames:

* Training, validation, and testing data are fed into the corresponding ImageDataGenerator objects using flow\_from\_dataframe () method.
* DataFrames (train\_df, valid\_df, test\_df) are passed along with relevant parameters such as file paths, labels, target size, colour mode, class mode, shuffle, and batch size.
* For training and validation, shuffle is set to True to shuffle the data for each epoch. For testing, shuffle is set to False to maintain the order of data.

1. Returning Data Generators:

* Returns three data generators: train\_gen, valid\_gen, and test\_gen containing augmented and pre-processed data ready to be fed into the deep learning model.

#### Utilizing Custom Callback in Model Training:

The custom callback class, named as MyCallback, constitutes a pivotal component within the research, offering a suite of customizable functionalities aimed at optimizing the training process. It facilitates real-time monitoring of crucial training metrics, such as accuracy and validation loss, both at the culmination of each epoch and batch iteration. Through this continuous evaluation, insights were gained into the model's performance, enabling informed decisions regarding learning rate adjustments and overall model optimization strategies. Moreover, MyCallback affords a level of flexibility that allows for user interaction during training, enabling intervention, parameter adjustments, or decision making based on real-time observations or experimental requirements.

A computer screen shot of a program

Description automatically generated

Figure 10. Custom Callback - Code Snippet

1. Setting up Callback Parameters:

* The parameters such as patience, stop\_patience, threshold, factor, ask\_epoch, and batches are predefined to configure the behaviour of the custom callback during training.

2. Instantiating the Callback:

* The custom callback MyCallback is instantiated with the defined parameters and passed to the callbacks list.
* This list holds the callback objects that will be executed during model training.

3. Training the Model with Callbacks:

* During model training, the callbacks list is passed to the fit () method of the model.
* This ensures that the defined callback functions are executed at specific points during training.

# Results:

## Introduction to Results:

The study's findings are discussed in this part, with an emphasis on comparing the effectiveness of many convolutional neural network (CNN) architectures trained with various optimizers for the goal of predicting leaf disease using the Plant Village dataset. The study set out to identify which architecture and optimizer combination produced the greatest overall performance and maximum accuracy in classifying leaf diseases through extensive experimentation and analysis.   
The investigation's main focuses were covered by the results that were given. First, per-epoch data analysis revealed information about each model's training progress, including adjustments to learning rate, accuracy, and loss. Through this study, it was possible to see how well each model learnt from and adjusted to the training set over the course of several epochs.

Each model's performance in training, validating, and testing was also evaluated, and measures like accuracy and loss were looked at to see how well the models generalized to new data. Important insights into the robustness and generalization capacities of the trained models were obtained by comparing performance across several datasets.   
In addition, a thorough examination of the classification reports produced by each model was given, examining parameters like F1-score, precision, and recall for distinct classes. This research allowed for the discovery of each model's strengths and weaknesses by providing a detailed understanding of how well it classified particular leaf diseases.

The study sought to determine which CNN architecture(s) and optimizer(s) provided the most promising outcomes for leaf disease prediction by thoroughly comparing the models. The study of plant pathology gained important insights by clarifying the advantages and disadvantages of each strategy, which enhanced knowledge of the use of deep learning methods in agricultural research.   
The findings were thoroughly analysed, their ramifications were explored, and possible directions for further research in this area were indicated in the sections that followed.

## Training Progress:

The dynamics of the training process can be better understood by examining the training times for each architecture using a variety of optimizers. EfficenetNetB3 showed significant differences in training times among the architectures based on the optimizer that was used. Notably, Adadelta yielded the least training time of 101 minutes, whereas RMSProp produced the longest training time of 151 minutes. This discrepancy implies that each optimizer has different optimization efficiencies and convergence rates, which affect the total training time needed for model convergence.

By comparison, MobileNet showed impressive uniformity in training durations for all optimizers, with the lowest variation found to be between 50 and 54 minutes. This consistency points to a strong optimization process in which the intrinsic properties of the design support steady and effective model convergence. ResNet50 also showed a similar pattern, albeit Adadelta had the longest training period at 140 minutes, as opposed to 116 minutes for Adamax and 108 minutes for RMSProp. This disparity highlights the impact of the optimizer on training effectiveness, with some optimization techniques showing better convergence performance for topologies.

In addition, VGG-16 demonstrated comparatively consistent training times for all optimizers, spanning from 118 to 126 minutes. This consistency points to a well-balanced optimization process involving many optimization techniques, wherein the optimization of parameters and the complexity of the architecture converge effectively. On the other hand, out of all the architectures, Xception showed the most differences in training times; RMSProp produced the longest training period, at 236 minutes, followed by Adadelta (210 minutes) and Adamax (154 minutes). This discrepancy highlights how sensitive some structures are to optimization techniques; more complex models necessitate longer optimization procedures to reach convergence.

All things considered, the training time study clarifies the complex relationship among architecture complexity, optimizer choice, and training effectiveness. A thorough analysis of training times for different optimizers and designs provides important information about the parameters affecting computing resource needs and model convergence. These results advance our knowledge of the deep learning optimization process and guide future attempts to design models with the goal of improving training effectiveness and convergence rates.

Table 3.Training Time Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| Architecture | Training time with Adamax | Training time with Adadelta | Training time with RMSProp |
| EfficenetNetB3 | 111 | 101 | 151 |
| MobileNet | 50 | 54 | 53 |
| ResNet50 | 116 | 140 | 108 |
| VGG-16 | 119 | 126 | 118 |
| Xception | 154 | 210 | 236 |

## Training, Validation, and Testing Performance:

The training accuracy and loss metrics for every CNN architecture that was trained using various optimizers are displayed in Table 3. With scores ranging from 0.99952 to 0.99991, Adamax consistently produced high training accuracy across architectures. Excellent results were also obtained using Adadelta, whose training accuracy ranged from 0.90652 to 0.99984. With values ranging from 0.95424 to 0.99979, RMSProp shows a slightly worse training accuracy than the other optimizers.

All optimizers and architectures show a similar pattern in terms of training loss. Lower training loss values were typically the outcome of using Adadelta and Adamax, indicating successful optimization and model convergence. For some designs, like VGG-16, where the loss value reached an exceptionally high value of 1,498,191.375, RMSProp showed larger training loss values. This anomaly points to possible problems with the stability or convergence of the optimization for the VGG-16 when trained using RMSProp.

All things considered; the training performance measurements shed light on how well various optimizers support model convergence. While Adadelta and Adamax consistently produce minimal training loss and excellent training accuracy across architectures, RMSProp's performance is inconsistent, with rare disparities seen, especially in VGG-16 situations. These results highlight how crucial it is to choose optimizers carefully in order to achieve the best model performance possible during training. To clarify the underlying causes of these variances and anomalies in training performance, more examination and research may be necessary.

Table 4. Training Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Architectures | Optimizers | Training Accuracy | Training Loss |  |
| EfficientNetB3 | Adadelta | 0.99984 | 0.09715 |  |
| Adamax | 0.99982 | 0.12424 |  |
| RMSProp | 0.99965 | 0.06087 |  |
| MobileNet | Adadelta | 0.99961 | 0.10478 |  |
| Adamax | 0.99991 | 0.12012 |  |
| RMSProp | 0.99726 | 0.10817 |  |
| ResNet50 | Adadelta | 0.99965 | 0.11693 |  |
| Adamax | 0.99991 | 0.14739 |  |
| RMSProp | 0.99846 | 0.10754 |  |
| VGG-16 | Adadelta | 0.98340 | 0.27617 |  |
| Adamax | 0.99636 | 0.21265 |  |
| RMSProp | 0.95424 | 1498191.37500 |  |
| Xception | Adadelta | 0.90652 | 7.82996 |  |
| Adamax | 0.99952 | 0.12464 |  |
| RMSProp | 0.99979 | 0.05484 |  |

The validation accuracy and loss metrics for every CNN architecture trained with various optimizers are shown in Table 4. Adadelta and Adamax consistently produced good validation accuracy results across architectures, with values ranging from 0.89448 to 0.99908. With results ranging from 0.94328 to 0.99687, RMSProp likewise delivered acceptable validation accuracy, albeit marginally lower than Adadelta and Adamax.

For validation loss, all optimizers and architectures follow a similar pattern. Lower validation loss values were typically obtained by Adadelta and Adamax, suggesting strong model generalization and resilience to new data. But on occasion, RMSProp produced larger validation loss values; this was especially noticeable in architectures like VGG-16 and Xception. This may point to overfitting tendencies and possible problems with model generalization when trained with RMSProp.

A comparison of training and validation performance can be used to determine if overfitting or underfitting is present. Overfitting, in which the model learns to memorize the training data instead of generalizing to unseen data, is suggested if the training accuracy is noticeably higher than the validation accuracy. On the other hand, low training, and validation accuracy points to underfitting, in which the model is unable to identify the underlying patterns in the data.

After examination, it seems that the models trained with Adamax and Adadelta performed rather well overall, with training and validation accuracy nearly matching. On the other hand, disparities between training and validation performance were seen in a few models, most notably VGG-16 and Xception trained with RMSProp, indicating possible overfitting. The intricacy of these architectures and the optimization dynamics of RMSProp could be to blame for this mismatch. The latter could have caused unduly aggressive parameter updates during training, which would have overfitted the training set. To reduce overfitting and enhance these models' generalization capabilities, more research and regularization strategies would be necessary.

Table 5.Validation Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Architectures | Optimizers | Validation Accuracy | Validation Loss |  |
| EfficientNetB3 | Adadelta | 0.99908 | 0.10204 |  |
| Adamax | 0.99834 | 0.12952 |  |
| RMSProp | 0.99834 | 0.06790 |  |
| MobileNet | Adadelta | 0.99632 | 0.11667 |  |
| Adamax | 0.99613 | 0.13330 |  |
| RMSProp | 0.99079 | 0.13398 |  |
| ResNet50 | Adadelta | 0.99466 | 0.13368 |  |
| Adamax | 0.99687 | 0.15594 |  |
| RMSProp | 0.98950 | 0.12866 |  |
| VGG-16 | Adadelta | 0.97864 | 0.27381 |  |
| Adamax | 0.99006 | 0.22947 |  |
| RMSProp | 0.94328 | 0.33017 |  |
| Xception | Adadelta | 0.89448 | 7.85990 |  |
| Adamax | 0.99576 | 0.14055 |  |
| RMSProp | 0.99687 | 0.06476 |  |

The testing accuracy and loss metrics for each CNN architecture trained with various optimizers are shown in Table 5. With scores ranging from 0.89413 to 0.99871, all optimizers consistently produced good testing accuracy across architectures. Comparably, testing loss values were generally modest, indicating that the model performed well on untested data.

On the other hand, assessing loss values for several architectures trained using RMSProp show significant differences. Testing loss values for VGG-16 and Xception are abnormally high, reaching 9,670,691 and 7.85561, respectively. These anomalies point to possible problems with the stability and generalization of the model after training with RMSProp, which calls for more research into the regularization strategies and optimization dynamics used.

After examination, it seems that most models perform in a balanced manner, with measures for training and validation accuracy closely matching testing accuracy. On the other hand, disparities between training and testing performance are seen in VGG-16 and Xception trained with RMSProp, indicating possible overfitting tendencies. These architectures' intricacy and RMSProp's optimization dynamics may have caused excessively aggressive parameter updates during training, which overfit the training set.   
In summary, even though most models perform well on the testing dataset, more research into regularization techniques and optimization strategies is necessary to address overfitting and enhance generalization performance on untested data, given the anomalies found in VGG-16 and Xception trained with RMSProp.

Table 6. Testing Metrics

|  |  |  |  |
| --- | --- | --- | --- |
| Architectures | Optimizers | Testing Accuracy | Testing Loss |
| EfficientNetB3 | Adadelta | 0.99871 | 0.10284 |
| Adamax | 0.99816 | 0.13038 |
| RMSProp | 0.99853 | 0.06678 |
| MobileNet | Adadelta | 0.99816 | 0.11370 |
| Adamax | 0.99705 | 0.13106 |
| RMSProp | 0.99871 | 0.12558 |
| ResNet50 | Adadelta | 0.99705 | 0.12745 |
| Adamax | 0.99632 | 0.15948 |
| RMSProp | 0.99319 | 0.12347 |
| VGG-16 | Adadelta | 0.97606 | 0.27861 |
| Adamax | 0.99282 | 0.22509 |
| RMSProp | 0.93795 | 9.67069 |
| Xception | Adadelta | 0.89413 | 7.85561 |
| Adamax | 0.99761 | 0.13095 |
| RMSProp | 0.99871 | 0.06189 |
|  |  |  |  |

## Classification Report Analysis:

Table 7.Classification Report

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Architectures | Optimizers | Precision | Recall | F1-Score |
| EfficientNetB3 | Adadelta |  |  |  |
| Adamax |  |  |  |
| RMSProp |  |  |  |
| MobileNet | Adadelta |  |  |  |
| Adamax |  |  |  |
| RMSProp |  |  |  |
| ResNet50 | Adadelta |  |  |  |
| Adamax |  |  |  |
| RMSProp |  |  |  |
| VGG-16 | Adadelta |  |  |  |
| Adamax |  |  |  |
| RMSProp |  |  |  |
| Xception | Adadelta |  |  |  |
| Adamax |  |  |  |
| RMSProp |  |  |  |
|  |  |  |  |  |