CAPSTONE PROJECT 2

Final Year Project Report

Integrated Tomato Plant Disease Detection System with Attention Mechanism

by

TAN JAVIER
19017219

BSc (Hons) in Computer Science

Supervisor: Dr. Farrukh Hassan

Semester: April 2024
Date: 15 July, 2024

Department of Computing and Information Systems
School of Engineering and Technology
Sunway University

Abstract

This study presents an advanced deep learning model for tomato plant disease detection, addressing critical challenges in precision agriculture and contributing to global food security efforts. The proposed model integrates an EfficientNet-B3 architecture with a Convolutional Block Attention Module (CBAM), leveraging transfer learning from ImageNet to enhance feature extraction and classification performance. Utilizing the comprehensive PlantVillage dataset, the model was meticulously trained and evaluated on 10 distinct classes of tomato plant conditions, encompassing 9 specific diseases and healthy samples, totaling approximately 10,000 images. The methodology employed a multifaceted approach, including extensive data preprocessing techniques such as augmentation, resizing, and normalization to enhance model generalization. The model development process involved careful hyperparameter tuning and the implementation of custom callback functions for optimal training dynamics. Comparative analyses were rigorously conducted against established CNN architectures including VGG16, DenseNet121, ResNet50, and the base EfficientNet-B3, providing a comprehensive benchmark of performance. The proposed EfficientNet-B3 with CBAM model achieved exceptional performance, demonstrating 99.55% accuracy, 99.60% average precision, and 99.50% for both average recall and F1-score across all disease classes. This performance notably surpassed that of baseline models, albeit by small margins in some cases. Ablation studies were performed to elucidate the individual contributions of transfer learning and the attention mechanism, revealing their synergistic effect on model performance. Grad-CAM visualizations further illustrated the model's enhanced ability to localize disease-relevant features, particularly for complex conditions like late blight. To demonstrate practical applicability, a Flask-based web application was developed, integrating the trained model for real-time disease detection. This implementation revealed both the potential and limitations of deploying such systems in real-world scenarios, highlighting areas for future improvement, particularly in handling non-plant images. This research makes significant contributions to the field of automated plant disease detection, offering valuable information into the application of advanced deep learning within agriculture. The findings pave the way for future advancements in crop management systems, potentially revolutionizing disease monitoring practices. However, the study also acknowledges limitations and proposes future directions, including the incorporation of disease severity assessment and enhanced robustness to real-world variability, to further bridge the gap between laboratory performance and practical agricultural applications.

Table of Content

List of Figures

List of Tables

1.0 INTRODUCTION

The purpose of this chapter is to provide a short but crucial briefing for this project. There are six subsections in this introductory chapter. In section 1.1, the brief history and background knowledge about this project is provided whereas in section 1.2, the problem statements that are being addressed in this project were discussed in detail. Moving on, section 1.3 contains the objectives of this project, and in section 1.4, the research as well as the development scope of this project is outlined.

1.1 Project Overview & Background

The majority of living beings on Earth have a relationship with plants, albeit close or distant, and humans are extremely reliant on plants in our daily lives because they provide us with nutrients and also produce oxygen for us to breathe. This means that extensive measures must be taken in order to properly care for and prevent any possible threats from happening to them such as diseases (Rizzo, 2021). From a different perspective, the agriculture industry is one of the highest contributors to the economy for the majority of countries worldwide, and in some cases, is also the primary revenue source for numerous developing and underdeveloped countries. According to Li and Solaymani (2021), the agriculture industry is one of the forces that is contributing to the economic growth within Malaysia. Most people are able to earn a living and more through the act of farming, and farmers usually cultivate a large variety of plants in the form of fruits, vegetables, and other crops as well. However, there are many threats that are currently plaguing plants which are in the form of pests, weather, and plant diseases. These threats have a severe impact on the quality and quantity of the overall crop production yield and are also responsible for the loss of $220 billion dollars annually for plant disease treatment (FAO, 2019). In response to plant health challenges, numerous agricultural producers have turned to intensive application of chemical agents, including pesticides and herbicides. While this approach aims to control plant diseases and boost crop yields, it often results in a trade-off: increased production volume at the expense of crop quality. This widespread chemical usage, though effective in managing diseases, can lead to a paradoxical outcome where quantity rises but overall crop quality diminishes.

Using tomatoes as an example, they are one of the most widely cultivated and consumed vegetable crops globally. In 2021, the worldwide production of tomatoes reached a staggering 187.12 million tonnes (FAO, 2023). Tomatoes are not only a vital source of nutrition but also a significant economic contributor for many countries. They are rich in essential nutrients

such as vitamin C, potassium, folate, and vitamin K, as well as antioxidants like lycopene, which has been linked to various health benefits, including reduced risk of heart disease and certain types of cancer (Chaudhary et al., 2018). Moreover, tomatoes are versatile and are able to be consumed fresh, cooked, or processed into various products like sauces, juices, and ketchup, making them an integral part of many culinary traditions worldwide. However, tomato production faces numerous challenges, particularly from plant diseases. According to recent estimates, tomato diseases cause yield losses ranging from 30% to 100% depending on the severity of the infection (Shoaib et al., 2023). Some of the most common and destructive tomato diseases include late blight, early blight, bacterial spot, and tomato yellow leaf curl virus (Barbedo, 2019). These diseases not only reduce the quantity and quality of tomato yields but also pose significant economic burdens for farmers in terms of disease management costs. Farmers often resort to the extensive use of pesticides and fungicides to control tomato diseases, which leads to increased production costs, environmental pollution, and potential health risks for consumers (Shamshiri et al., 2018). Moreover, the development of pathogen resistance to commonly used chemicals further complicates disease management strategies (Parnell et al., 2017). Consequently, there is a pressing need for more sustainable, efficient, and accurate methods for detecting and diagnosing tomato diseases, which could help farmers optimize disease management practices and minimize the reliance on chemical interventions.

The most effective method in preventing plants from contracting diseases is to practice early detection and provide effective treatments as soon as possible. Traditionally, the only method to confirm the condition of a plant as to whether it had a disease was to check it by hand or with the naked eye meaning, it required knowledgeable and experienced professionals in order to perform this task. This task is usually considered to be extremely stressful, tedious, and time-consuming. However, there are numerous types of research being conducted in order to provide proper solutions that are able to reduce the complexity of this task and aid farmers in easily carrying out their duties. A significant portion of the proposed approaches draws from the domain of visual computing, leveraging a combination of traditional image manipulation techniques, machine learning methodologies, and advanced deep learning frameworks. These computational strategies form the backbone of many current solutions in the field. For instance, Fuentes et al. (2017) developed a deep learning-based system using convolutional neural networks (CNNs) for real-time tomato disease detection. Their system achieved an average accuracy of 83% in identifying nine different tomato diseases. Similarly,

Rangarajan et al. (2018) proposed a CNN-based approach for tomato leaf disease classification, which attained an accuracy of 97.49%. While these solutions demonstrate the potential of deep learning in tomato disease detection, there remain opportunities for improvement in terms of model accuracy, efficiency, and user-friendliness.

The aim of this project is to develop an enhanced deep learning model for tomato disease detection that addresses some of the limitations of existing solutions. By leveraging advanced CNN architectures and attention mechanisms, this project seeks to improve the accuracy as well as the robustness of disease detection. Furthermore, by also enhancing the model's ability to focus on relevant features and suppress irrelevant ones, the incorporation of attention modules such as Convolutional Block Attention Module (CBAM) which could improve disease detection performance. In addition to improving the model's performance, this project aims to make the developed solution more accessible and user-friendly by integrating it into a web application. The web application will allow users to easily detect diseases from tomato plants via image uploads and receive disease predictions, along with relevant information such as disease symptoms, causes, and prevention measures. Moreover, this project aims to contribute to the broader field of plant disease detection by exploring the effectiveness of advanced deep learning techniques and attention mechanisms in the context of tomato diseases. The insights gained from this project could potentially be applied to other crops and plant diseases, furthering the development of accurate, efficient, and accessible disease detection systems.

1.2 Problem Statement

The overall problem statement that this project is addressing is associated with constructing an application that is able to detect and identify tomato plant diseases for the user. However, many sub-issues should be resolved beforehand because they could affect the effectiveness of the proposed application. Three problems have been identified by analyzing previous works done by researchers or from pre-existing systems available on the market.

1.2.1 Tomato Plant Diseases Identification Requires High Amount of Effort & Time

The diagnosis of plant diseases is a critical aspect of agriculture, as it directly impacts crop productivity and, consequently, food security (Midhunraj et al., 2023). Traditionally, this process has been manual, requiring significant effort and time from individuals who must visually inspect plants and identify diseases based on their symptoms. However, this

approach presents several challenges. Given the vast diversity and intricate nature of agricultural vegetation, which result in a myriad of diseases, making it difficult for most farmers to identify plant diseases within a timely manner and also requires them to exert more effort to diagnose all of plants (Midhunraj et al., 2023). Plant ailments often manifest initial indicators before fully developing. Once established, many pathogens can rapidly proliferate across an entire plantation. This necessitates vigilant and frequent crop monitoring, as early intervention can significantly impede disease progression and limit its spread throughout the cultivation area. There are some deep learning applications that have shown promise in automating and simplifying the process of plant disease diagnosis (Prabha. 2021). These applications leverage advanced algorithms to analyze plant images and identify diseases, significantly reducing the time and effort required (Liu and Wang, 2021).

1.2.2 Inaccurate Knowledge Regarding Tomato Plant Disease

The traditional method of diagnosing plant diseases is not only labor-intensive but also prone to inaccuracies (Midhunraj et al., 2023). The accuracy of disease identification is compromised by various factors, including the observer's expertise and the quality of the visual data such as lighting conditions or image resolution. According to Kaur and Sharma in 2021, the complexity and volume of both cultivated plants and crops is able to result in various illnesses, and as a result, a pathologist may frequently misdiagnose an illness. Moreover, using basic observation does not always provide enough information in order to diagnose a plant disease and this is because some diseases are difficult to identify which results in scientists in laboratories to cut off the plant in order to analyze it but this method is expensive and unaffordable to some people. Both machine learning (ML) and deep learning (DL) methods have shown adequate results in the field of plant disease detection due to the reason that these methods have been tested and created by reliable people to identify plant diseases. The model proposed by Jayswal and Chaudhari (2023) outperformed any traditional deep learning algorithms, scoring 96 percent during evaluation and another model achieved an 88 percent training accuracy (Varshney et al., 2022).

1.2.3 Limitations in Existing Tomato Disease Detection Models

While deep learning models have shown promise in automating tomato disease detection, there remain significant challenges in developing systems that are both highly accurate and computationally efficient. Current models often struggle to maintain consistent performance across diverse real-world conditions, where factors such as varying light conditions, image

quality, and disease symptom presentation are able to impact detection accuracy (Ferentinos, 2018). One key challenge is the need to capture subtle visual features that distinguish between different tomato diseases, some of which may have similar symptoms or appear at different stages of progression. This requires models with strong feature extraction capabilities, but improving these capabilities often results in higher cost of computational requirements (Barbedo, 2018). Furthermore, the balance between model accuracy and efficiency is crucial, especially considering the potential deployment of these systems in resource-constrained agricultural environments. While more complex models might offer improved accuracy, they may not be practical for real-world practical applications as well as on devices with limited processing capabilities (Boulent et al., 2019). There is also the challenge of developing models that are able to adapt well to new and unseen data. Tomato diseases present differently based on factors such as plant variety, growth stage, and environmental conditions. Creating a model that performs consistently across these variables remains an active area of research (Too et al., 2019). Addressing these challenges requires creative approaches in model architecture design, data preprocessing, and training strategies. The goal is to develop tomato disease detection models that not only achieve high accuracy but also offer improved efficiency, robustness, and generalizability, making them more suitable for practical applications in diverse agricultural settings.

1.3 Project Objectives

The primary objective of this project is to develop an enhanced deep learning model for accurate and efficient tomato plant disease detection and to integrate it into a user-friendly web application. This overarching goal is broken down into several sub-objectives:

1. **Investigate the effectiveness of incorporating attention mechanisms into CNN architectures for tomato disease detection**

   This sub-objective aims to explore the potential benefits of integrating attention mechanisms, specifically the Convolutional Block Attention Module (CBAM), into state-of-the-art CNN architectures such as EfficientNet-B3. By enhancing the model's ability to focus on relevant features and suppress irrelevant ones, the incorporation of attention modules is expected to improve disease detection performance.

2. **Optimize the model's hyperparameters and training process**

   To achieve the best possible performance, it is crucial to fine-tune the model's hyperparameters and optimize the training process. This sub-objective involves experimenting with different hyperparameter configurations, such as learning rate, batch size, and regularization techniques, to find the optimal settings that maximize the model's accuracy and efficiency.

3. **Develop a user-friendly web application for easy access to the disease detection model**

   To make the enhanced tomato disease detection model accessible to a wider audience, this sub-objective focuses on developing a Flask-based web application. The application will provide a user-friendly interface for uploading tomato plant images and displaying disease predictions, along with relevant information such as disease symptoms, causes, and prevention measures.

4. **Compare the performance of the proposed model with other state-of-the-art models**

   To evaluate the effectiveness of the proposed deep learning model, it is essential to compare its performance with other popular and well-known models for tomato disease detection. This sub-objective involves benchmarking the proposed model against existing models based on their accuracy, precision, recall, and F1 score, using a standardized dataset. The reason behind this comparison is to provide further insights into the relative strengths and weaknesses of the proposed model as well as identify certain parts that may need further improvements.

1.4 Project Scope

The scope of this project encompasses the development, evaluation, and deployment of a deep learning model for tomato plant disease detection, as well as the creation of a web application to facilitate user interaction with the model. The proposed model will then be trained as well as tested using a dataset of tomato plant images captured under proper lighting conditions and with clear quality. The model will be capable of detecting and classifying the following tomato diseases and conditions which are, bacterial spot, early blight, late blight, leaf mold, septoria leaf spot, two-spotted spider mite, target spot, mosaic virus, yellow leaf curl virus, and healthy tomatoes with no diseases.

However, the project has certain limitations and restrictions. The model's performance may be affected by factors such as extreme lighting conditions, low image resolution, or the presence of multiple diseases in a single image. Additionally, the model will not be able to provide recommendations for disease treatment or prevention beyond the general information displayed in the web application. The project does not include real-time monitoring of tomato plants or the ability to detect diseases in other plant species.

To ensure the feasibility of the project within the constraints, resources and timeframe given, the following restrictions and limitations will be applied:

1. The web application will be developed using the Flask framework and will include a user-friendly interface for image upload and result display. However, the application will not include advanced features such as user authentication, data storage, or integration with external systems.

2. The model will be trained using publicly available datasets and pre-trained CNN architectures to leverage transfer learning. However, the project will not collect new tomato plant images or create a custom dataset via amalgamation of multiple datasets.

3. The evaluation of the model's performance will be limited to making comparisons with a selected set of well-known models, rather than an exhaustive comparison with all available models.

4. The project will focus on the detection and classification of tomato plant diseases based on visual symptoms present in the images. It will not involve the development of models for disease severity estimation, yield prediction, or other related tasks.

5. The web application will provide general information about the predicted diseases, such as symptoms, causes, and prevention measures. However, it will not offer personalized treatment recommendations or integrate with existing disease management systems.

1.5 Chapter Summary

This introductory chapter provided an overview of the project which discussed the significance of tomato production and the challenges posed by diseases. It highlighted the limitations of current disease detection methods and outlined the project's aim to develop an enhanced deep learning model for tomato disease detection, incorporating advanced CNN architectures and attention mechanisms. The primary objective is to create an accurate and efficient solution accessible through customizations of an advanced CNN architecture. Several sub-objectives were identified, including investigating attention mechanisms, optimizing model performance, comparing the proposed model with other popular as well as well-known models, and providing disease information to users. Besides that, the project scope was defined, specifying the diseases and conditions the model will focus on detecting and classifying. The restrictions and limitations were outlined to ensure the project's feasibility within the given resources and timeframe.

2.0 LITERATURE REVIEW

In this section of the report, the background knowledge regarding several topics related to the project will be discussed in detail as well as conducting an analysis and review of related studies in order to identify the research gaps within their results. Within this literature review, the key deep learning stages will be analyzed in detail regarding their purpose, and also contain discussions on the overview of current materials provided by other similar studies or even credible sources. The review will mainly discuss findings and knowledge in terms of tomato disease detection as well as agricultural concepts.

2.1 Data Acquisition

The stage of data acquisition is characterized as the act of collecting data from a variety of system sources (Choudhuri and Mangrulkar, 2021). This stage is vital for the deep learning-based process and this is because high-quality data is commonly considered as the foundation of any successful deep learning project, as the accuracy and generalizability of the resulting models heavily depend on the quality and quantity of the data used. According to Roh et al. (2019), most deep learning-based systems allocate their time and resources in order to gather, clean, and analyze the necessary data. Therefore, it is mandatory to prepare the data needed for the deep learning model to perform its task efficiently and effectively. In addition, similar studies related to plant disease detection created unique datasets by capturing photographs of the plants (Gutierrez et al., 2019). However, this has its drawbacks due to the fact that it requires the usage of manual labor in order to collect the images, and the majority of the time, these datasets will result in smaller data compared to other resources and lead to the underdevelopment of the deep learning model in terms of its accuracy and effectiveness.

On the other hand, some studies opted to use online resources (Singh et al., 2020; Arsenovic et al., 2019) such as search engines in order to create their own datasets from scratch. This type of data collection method is able to achieve a larger dataset but might contain unwanted images that would need to be filtered out, and also require some images to be evaluated by agricultural experts for confirmation of the plant diseases. Hence, this section of the literature review will provide an in-depth discussion of the current popular plant disease datasets used by others in order to develop the model for plant disease detection and will also validate the importance of both data that is collected in a controlled environment as well as an uncontrolled environment.

2.1.1 Plant Disease Datasets

There are several public datasets being used within similar studies that focus on identifying and classifying various plant-related diseases. The PlantVillage, PlantDoc, FieldPlant, and PDD271 datasets are some of the available datasets that are currently able to be accessed publicly. These public datasets will be evaluated on their overall characteristics in terms of image size and plant disease classification for each individual species of plant. The overall findings, characteristics, and important remarks of all the datasets are summarized in table format which is able to be seen in Table 1.

Firstly, the PlantVillage (Hughes and Salathé, 2015) dataset was created with the aid of Penn State University, Florida State University, and Cornell University, and contains more than 54,000 images of plant diseases as well as healthy images across 14 plant species with each plant species having at least 200 plus images that resulted in 38 distinct categories. The images were taken in a controlled environment with a static background which included only having a singular leaf of the plant with the disease at the center of the image, and occasionally separated via the disease severity using keywords of "late" and "early" before the disease name. By doing so, it aids researchers and professionals in obtaining a thorough grasp of the variations in plant growth and progress (Shoaib et al., 2023). Moreover, every image has a fixed dimension of 256 by 256 for height and width respectively. The image collection was assembled from diverse sources, including web-based repositories, academic research centers, and individual contributors. This compiled dataset was subsequently divided into three distinct groups: a primary set for model training, a separate set for validation purposes, and a final set for testing. The largest portion of images was allocated to the training subset.

Next, the PlantDoc (Singh et al., 2020) dataset was introduced in 2019 by the Indian Institute of Technology Gandhinagar India and contains more than 2,500 images of various plant diseases and healthy plants which contains 13 plant species that has a total of 27 distinct classes comprising both disease and health plants. Each of these images from this dataset depicts the diseased plants in the field instead of in a laboratory, however, the dataset was not created via proper photography but was taken from online search engines such as Google and Ecosia in order to save both time and effort (Singh et al., 2020). Additionally, the images also capture not just the leaves of the diseased plant, but other parts of the plant as well, and the

composition of each of the captured plants varies from one another with some appearing larger or smaller than others due to the distance of the camera.

The third dataset is the FieldPlant (Moupojou et al., 2023) dataset has a total of 5,170 plant disease images collected directly from various plantations that resulted in 27 disease classes across three plant species which are cassava, corn, and tomato. The images were all taken from multiple fields with each having different lighting and reflections on the leaves of the plant, and the images were then labeled by professionals in order to verify their authenticity. The labeling or annotation was done using the online platform RoboFlow and certain plants required multiple annotations due to having many leaves while others only required a single annotation because they had one leaf. The images within this dataset were captured with a resolution of 4608 by 3456 (4:3) pixel resolution.

The PDD271 dataset was created by Liu et al. (2021) and this dataset contains 220,592 images of many plant diseases across 271 classes. Each of the diseased plant categories contains a minimum of 500 images and each plant disease for a specific species was captured from multiple angles in order to increase the overall changes in the image classification. The dataset consists of images of vegetables, grains, and plants taken from the real fields. The downside of this dataset is that it is currently being managed by Beijing Puhui Sannong Technology Co. Ltd, meaning the entire dataset is not available publicly, and only a small sample consisting of 10 images for each of the 271 classes is allowed to be used.

Dataset	No. of Images	No. of Classes	Data Acquisition
PlantVillage (Hughes and Salathé, 2015)	54,305	38	Laboratory
PlantDoc (Singh et al., 2020)	2,598	17	In-Field
FieldPlant (Moupojou et al., 2023)	5,170	27	In-Field
PDD271 (Liu et al., 2021)	220,592	271	In-Field

Table 1 Plant Disease Dataset

2.1.2 Data Collection Methods

The performance of any deep learning model is mostly influenced by the training and testing of the data from a dataset, meaning that the quality, as well as the quantity, is extremely vital in the creation of a proper deep learning model. This means that the method in which the images are collected is also important because images acquired in a controlled environment will have mixed outcomes when compared to images acquired in the fields. However, images acquired within a controlled environment are much easier to handle and classify than images collected from the actual fields.

The images that were acquired within a controlled environment such as a laboratory are able to achieve a more than adequate performance in terms of accuracy (Hughes and Salathé, 2015) however, these types of datasets tend to have significant drawbacks when testing with real diseased plants in the actual environment. One example is the PlantVillage dataset by Hughes and Salathé (2015) which only consists of images of both healthy and diseased plants that were captured in a controlled environment with a static background.

When the controlled environment datasets were tested against field images, the performance of the deep learning model dropped significantly. This is due to the fact that in contrast to images collected from laboratories, real fields have a higher level of complexity due to the reason that there are many leaves captured in a single image, multiple plant organs are captured as well, different lighting and reflections on the leaves, and differences in the background (Singh et al., 2020). This study by Ngugi et al. in 2020 shows that the decrease in performance of deep learning models is because of field images continuing non-static backgrounds, but by removing the backgrounds, the accuracy of the models increases. Therefore, plant disease detection systems that are being trained with images from a controlled environment, such as in a laboratory, are not useful in practical applications (Ahmad et al., 2021; Ngugi et al., 2020; Wang et al., 2022). Furthermore, Li et al. (2021) stated that there needs to be an established large dataset of plant diseases that were captured within fields and the images must also contain a high amount of complexities in order to be used for plant disease detection systems, that are able to assist farmers by providing them a practical alternative to resolve crop issues (Nagaraju and Chawla, 2020).

2.2 Data Pre-Processing

The pre-processing stage is one of the most common steps that is performed before passing the images through to the deep learning model. This stage is traditionally executed via computer vision techniques such as noise reduction, image resizing, contrast enhancements, feature extraction, and etcetera. The main purpose of this stage is to increase the deep learning model output. According to Militante et al. (2019), “the pre-processing stage is usually required to be applied to the images in order to reduce the computational costs and standardize the image resolutions to a specific standard benchmark”. Therefore, this section will discuss the various data pre-processing techniques that are the most popularly used by researchers when developing plant disease detection systems.

2.2.1 Noise Reduction

A variety of filtering techniques are employed to minimize noise, resulting in enhanced image clarity and smoothness. Examples of these filters are Gausian, median, and mean filters. All these filters have the effect of blurring the images, and this occurs in order to eliminate unwanted details from the images, however this also leads to the loss of additional information from the images in terms of the pixels (Hÿtch and Hawkes, 2020). To perform the necessary steps in reducing noise from images, two important morphological image operations are commonly applied onto either binary images or grayscale images, which are erosion and dilation. The erosion operation is used to remove floating pixels as well as thin lines so that only important objects remain, meaning it eliminates unnecessary foreground objects whereas, the dilation operation is used to add additional pixels to the boundary of an object which makes it more pronoun, and also fills in empty spaces within the object (Hÿtch and Hawkes, 2020). Furthermore, most images are usually captured in the RGB (Red; Green; Blue) format but this format is not suitable for operations related to noise reductions. Therefore, the images must convert from RGB to other color formats such as HSV, and the L*a*b color space. The hue, saturation, and value (HSV) color method is the most popular noise reduction color format because it somewhat follows human perception but other similar color format alternatives are the HSI (Hue-Saturation-Intensity) and the HSL (Hue-Saturation-Lighting) color methods. In the “L*a*b” color model, three distinct components represent different aspects of color. The 'L' parameter denotes brightness or luminosity. The 'a' dimension captures the spectrum between red and green hues, while the 'b' dimension represents the range from blue to yellow tones. Garcia-Lamont et al. stated in 2018 that these color spaces contain different brightness based on its hue, meaning images are able

to be have different lighting conditions when processed, and this is vital because the images captured of diseased plants under very different lighting conditions which is due to variables such as time and weather.

2.2.2 Feature Extraction

The process of converting input information in the form of images into a set of features is known as feature extraction. The main features that are considered during the extraction process of an image are its color, texture, and shape. The way feature extraction is being used in plant disease detection is by extracting unique qualities within the images in order to determine the condition of the plant and classify them as either healthy or diseased. Several algorithms are available to be implemented for this image pre-processing stage and the most popular algorithms are the Scale Invariant Feature Transform (SIFT), the Speeded Up Robust Features (SURF), the Pyramid Histogram of Visual Words (PHOW), and the Histogram of Oriented Gradient (HOG) (Kirti and Rajpal, 2020). The Histogram of Oriented Gradients (HOG) technique primarily focuses on capturing object shapes within images. It accomplishes this by identifying edge patterns across multiple orientations in the trained image data, whereas the SIFT algorithm is used to locate the scale and rotation invariant local features throughout the entire image in order to obtain a set of image locations which is also known as the key points of an image. Moving on, the SURF algorithm is theoretically similar to the SIFT algorithm but has the advantage of being much faster, which makes it even better for implementations related to real-time applications. Furthermore, the image feature extraction pre-processing is also able to be conducted within the gray-scale color space as well, and in this color space, other computing methods are being used, for example, the Grey Level Co-occurrence Matrix (GLCM). The GLCM computes various images via their texture properties, and the texture properties of an image are important for this project because both diseased as well as healthy leaves have unique textures such as diseased leaves having a more uneven texture while healthy leaves have smoother textures, meaning extracting the texture features from plants will allow for it to be classified as healthy or diseased.

2.2.3 Data Manipulation

In the domain of deep learning pre-processing, the emphasis shifts away from conventional feature extraction due to the innate ability of deep learning models to autonomously generate features (Sarker, 2021). Consequently, pre-processing predominantly centers on tasks like data augmentation and resizing input images to align with the model's parameters (Shorten

and Khoshgoftaar, 2019). The initial phase involves meticulous cleaning of both images and annotations, expunging any irrelevant or redundant elements (Suzuki et al., 2021). Subsequently, data augmentation techniques come into play, introducing subtle modifications to the dataset, a practice known to elevate model accuracy by diversifying the training set (Yang et al., 2022). Data cleaning serves as the foundational step, requiring the elimination of unnecessary images and annotations (Whang et al., 2023). Although primarily a manual process, certain aspects are able to be automated to enhance efficiency. This phase addresses issues such as images lacking corresponding annotations, annotations devoid of corresponding images, and the removal of duplicate or extraneous images (Whang et al., 2023). Moving on, data augmentation assumes a critical role in enhancing the model's ability to detect objects across varying scales. Minor alterations, encompassing rotations, changes in viewpoint, or adjustments in size, contribute to refining the model's accuracy by exposing it to diverse scenarios. Synthetically generated images become integral components of the dataset, especially considering that both greenhouse and cultivation chamber images are often captured within limited environmental conditions (Nikolenko, 2021). The inspection task, however, may necessitate adaptability to different scales, brightness levels, illumination, and orientations. Training the model to navigate these diverse conditions is facilitated through the inclusion of synthetically modified images, enriching the training dataset to ensure robust performance in real-world scenarios (Nikolenko, 2021).

2.3 Convolutional Neural Network

Deep Learning (DL) emerges as a specialized domain within deep learning, leveraging the capabilities of Artificial Neural Networks (ANN). DL introduces a notable advantage by eliminating the need for the manual creation of a new feature extractor for each problem, with ANN acting as the foundational element for diverse deep learning methodologies. Notably, Convolutional Neural Networks (CNN), a subset within the realm of deep learning, revolutionize the training of classifiers by directly utilizing raw images, circumventing human intervention in the feature extraction process.

Specifically applied to image classification tasks, CNNs employ convolutional layers that facilitate a hierarchical extraction of features. In these layers, initial stages concentrate on extracting simpler features such as edges, progressively advancing to deeper layers where more intricate and specific features are identified. The incorporation of pooling layers plays a crucial role in reducing the input’s dimensions. Following the convolutional and pooling

layers, fully connected neural networks take on the role of classifiers, leveraging the high-level features extracted during the earlier stages to make informed decisions. This seamless integration of ANN and CNN exemplifies the efficacy of deep learning in handling complex tasks, particularly in image-related domains.

Convolutional Neural Networks (CNNs) operate as a type of forward-propagating neural network, where information flows from the initial input to the final output. As illustrated in Figure 1, the structure of a CNN typically comprises three distinct sections which are, an input layer, multiple hidden layers, and a fully-connected output layer. The hidden portion of the network is composed of several specialized layers, including Convolutional layers, Rectified Linear Unit (ReLU) activation layers, and Pooling layers. These distinct layer types are combined to create a unified neural network architecture.

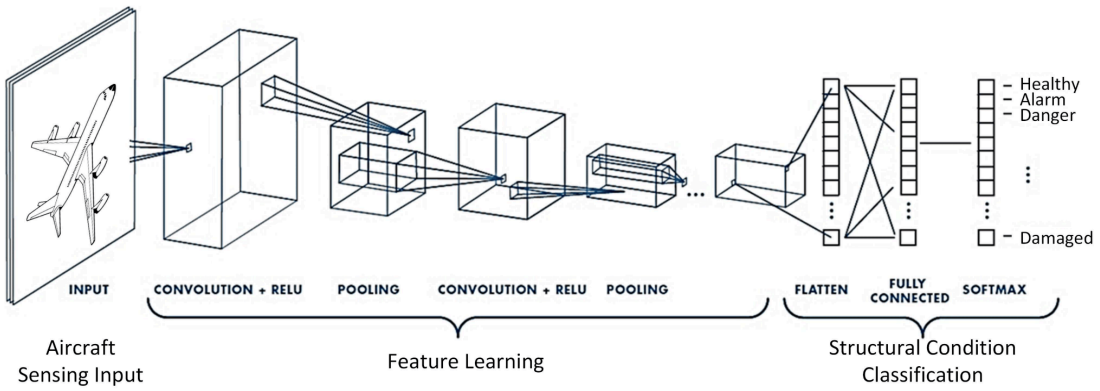


Figure 1 Convolutional Neural Network (Tabian et al., 2019)

The Input layer processes the image's pixel values arranged in an array. Subsequent to this, the hidden layers autonomously engage in feature extraction, with the convolutional layers serving as specialized feature extractors. Through convolution operations between the image matrix and filters, these layers efficiently carry out this crucial task. The introduction of non-linearity is facilitated by the Rectified Linear Unit (ReLU) activation function, which nullifies negative pixels during this process.

Moving forward, the Pooling layer strategically deploys various filters to reduce the dimensionality of the features. The high-dimensional feature map undergoes a flattening process, transforming it into a single linear vector before progressing to the fully connected

output layer. This resultant feature map is then directed to the fully connected layer for the final classification of the image into predefined classes.

The effectiveness of Convolutional Neural Networks (CNNs) heavily depends on several key configuration variables, known as hyperparameters. Among these, the number of training iterations, the quantity of samples processed in each training step, and the magnitude of weight updates are particularly crucial. These variables significantly shape how well the model trains and how quickly it acquires knowledge. Furthermore, the choice of optimization algorithm is another vital factor that can substantially enhance both the efficiency of the training process and the speed at which the model improves its performance.

In projects leveraging the Convolutional Neural Network (CNN) architecture, the significance of CNN frameworks cannot be overstated. Numerous frameworks have emerged, designed to streamline the development of tools that provide advanced capabilities while alleviating the complexities associated with intricate programming tasks. Each framework is uniquely crafted to cater to specific purposes, offering researchers a diverse array of tools and platforms for the seamless execution of Deep Learning (DL) experimental studies. Table 2 outlines the most renowned among these tools, presenting a comprehensive overview of their features and functionalities.

Framework	**Released Year**	**Programming Language**	**Operating System**	**CUDA Supported**
TensorFlow	2015	C++, Python	Windows, Linux, macOS	Yes
Keras	2015	Python	Windows, Linux, macOS	Yes
PyTorch	2016	Python, C	Windows, Linux, macOS	Yes
Caffe	2013	C++	Windows, Linux, macOS	Yes
Deeplearning4j	2014	C++, Java, Python	Windows, Linux, macOS	Yes

Table 2 CNN Framework (Mathew et al., 2021)

TensorFlow, developed by Google Brain, stands out as a multifaceted deep learning framework supporting multiple programming languages, including Python, C++, and R. Its versatility extends to hardware compatibility, enabling the deployment of models across various computational platforms such as CPUs and GPUs.

Built on top of TensorFlow, Keras offers a user-friendly Python interface for neural network development. Designed for rapid prototyping, Keras simplifies the construction of complex neural architectures, accommodating a wide range of models from convolutional networks (CNNs) to recurrent networks (RNNs). Its seamless operation across both CPU and GPU environments further enhances its utility.

Another prominent tool in the machine learning landscape is PyTorch, a Python-based framework known for its powerful tensor computations and GPU acceleration capabilities. PyTorch's distinguishing feature lies in its dynamic computational graph creation, facilitated by a tape-based autograd system. This flexibility makes it particularly well-suited for handling variable-length inputs and sequential data structures.

In the realm of computer vision, Caffe has carved out a niche for its exceptional speed in image processing and learning tasks. Developed by Yangqing Jia, this open-source framework has gained widespread adoption in both industrial and research settings. A notable aspect of Caffe is its Model Zoo, a repository of pre-trained models that significantly streamlines the problem-solving process for various vision-related challenges.

Tailored for enterprise environments, Deeplearning4j emerges as a robust option for Java and Java Virtual Machine (JVM) users. This framework prioritizes operational efficiency, emphasizing speed and reliability over pure research applications. Deeplearning4j leverages the ND4J tensor library for handling multi-dimensional arrays and supports a diverse range of data formats, including images, CSV, and plaintext. Its compatibility with both CPU and GPU platforms, coupled with seamless integration capabilities with big data technologies like Hadoop and Apache Spark, positions Deeplearning4j as a compelling choice for large-scale machine learning projects.

2.3.1 VGGNet

VGGNet is a deep convolutional neural network (CNN) architecture introduced by Simonyan and Zisserman in their 2014 paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" (Simonyan & Zisserman, 2014). The architecture was designed to investigate the impact of network depth on performance in large-scale image recognition tasks. The key feature of VGGNet is its simplicity and uniformity in architecture design. It consists of 16 or 19 weight layers, with all convolutional layers using 3x3 filters and all max-pooling layers using 2x2 filters with a stride of 2. The network follows a straightforward structure which comprises a stack of convolutional layers and later accompanied by fully connected layers. The use of small 3x3 filters throughout the network allows for a significant increase in depth while keeping the number of parameters relatively low compared to larger filter sizes.

VGGNet is designed to process RGB images of 224x224 pixels. Its architecture features a series of 3x3 convolutional layers, with the number of filters increasing from 64 to 512 as the network deepens. These layers use a 1-pixel stride and padding to maintain spatial dimensions. Interspersed 2x2 max-pooling layers with a stride of 2 reduce feature map size, lowering computational demands. The network culminates in three fully connected layers: two with 4096 neurons each, employing 50% dropout for regularization, and a final 1000-neuron layer corresponding to ImageNet classes, followed by a softmax activation. The two primary VGGNet variants are VGG16 and VGG19 with VGG16 comprising 13 convolutional and 3 fully connected layers. Its structure progresses from two 64-filter convolutional layers, through pairs of 128-, 256-, and 512-filter layers, separated by max-pooling. VGG19 expands on this with 16 convolutional layers, adding depth in the later stages. Specifically, it includes four 256-filter layers and two sets of four 512-filter layers before the final pooling and fully connected sections. These architectures are visually represented in Figures 2 and 3, with a summary provided in Table 3.

Model	Depth	Input Resolution	Convolutional Layers	Parameters (M)
VGG16	16	224 x 224	13	138.4
VGG19	19	224 x 224	16	143.7

Table 3 VGGNet Architecture Model Variations

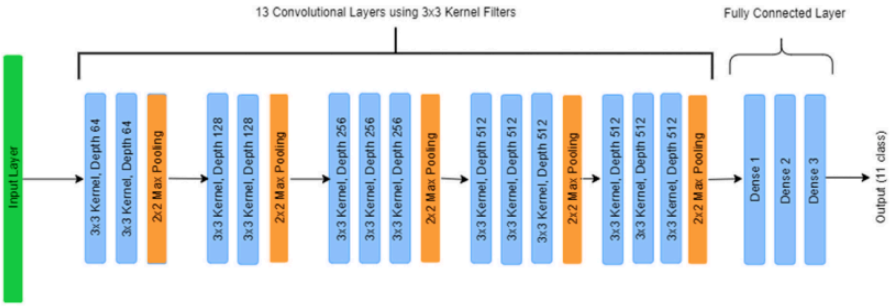


Figure 2 VGG16 Architecture Model Diagram (Paul et al., 2023)

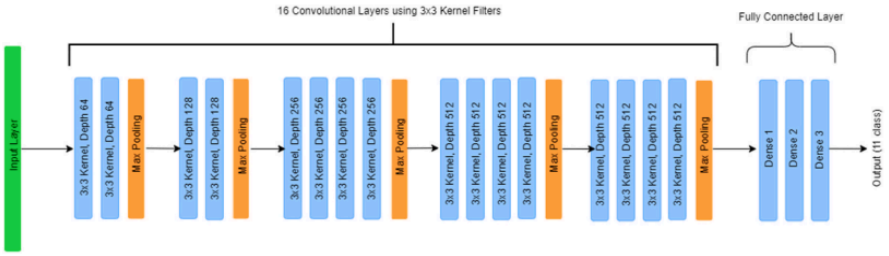


Figure 3 VGG19 Architecture Model Diagram (Paul et al., 2023)

The simplicity and uniformity of VGGNet's architecture make it easy to understand and implement. The use of small 3x3 filters throughout the network allows for a significant increase in depth while keeping the number of parameters relatively low. This design choice was inspired by the idea that multiple stacked smaller filters can have the same effective receptive field as a larger filter while using fewer parameters and being more discriminative. One of the main advantages of VGGNet is its ability to work well with other datasets and tasks which is due to its deep architecture and its uses of small filters which led to it having widespread adoption of VGGNet as a base model for transfer learning especially for plant disease detection. Transfer learning has been shown to be effective in reducing the amount of training data and computational resources required, as well as improving the performance of the model on the target task (Pan & Yang, 2009). However, VGGNet also has some limitations. One of the main drawbacks is its large number of parameters, which can make it

computationally expensive to train and deploy, especially on resource-constrained devices. Additionally, the deep architecture of VGGNet can make it prone to overfitting, especially when trained on smaller datasets. To mitigate this, techniques such as data augmentation, regularization, and transfer learning are often employed.

2.3.2 ResNet

ResNet, short for Residual Networks, is a CNN architecture introduced by He et al. in their 2016 paper "Deep Residual Learning for Image Recognition" (He et al., 2016). ResNet implements the theory of residual learning, meaning that it is able to conduct training for deeper networks without suffering from a decrease in performance. The key innovation in ResNet is the introduction of "identity shortcut connections," also known as skip connections or residual connections. ResNet introduces skip connections that bypass one or more layers, facilitating direct gradient flow through the network. This innovative approach addresses the vanishing gradient issue, enabling the creation of significantly deeper architectures - some extending to hundreds of layers. The core building block of ResNet is the residual unit, which incorporates a series of convolutions alongside a shortcut path. The unit's output combines the processed data from the convolutional layers with the unaltered input from the shortcut. This design allows the network to focus on learning residual mappings relative to the input, which proves more efficient than learning complete transformations from scratch.

The architecture of ResNet starts with the network beginning with an initial convolutional layer and then a max-pooling layer. Then, a series of residual blocks are stacked, with the number of blocks varying depending on the depth of the network. Each residual block usually consists of at least 2 convolutional layers, with the addition of batch normalization and ReLU activation applied after each convolution. The shortcut connection performs identity mapping and is added to the output of the convolutional layers. The network's terminal structure, following the residual modules, incorporates a global mean pooling operation. This is followed by a densely interconnected layer that executes the ultimate categorization step.

Several variants of ResNet have been proposed, differing in the number of layers and the type of residual blocks used. ResNet18 and ResNet34 are the shallower variants, consisting of 18 and 34 layers, respectively. These variants use basic residual blocks, which consist of 2 convolutional layers with 3x3 filters, and the first convolutional layer in each block is accompanied by the batch normalization as well as the ReLU activation, while the second

convolutional layer is followed only by batch normalization. After processing through two convolutional stages, the resulting feature maps are combined with the block's initial input via the skip pathway. This summation then undergoes activation through a ReLU function, completing the residual unit's forward pass.

ResNet50, ResNet101, and ResNet152 are the deeper variants, consisting of 50, 101, and 152 layers respectively. These variants use bottleneck residual blocks, which consist of three convolutional layers, a 1x1 convolution for dimensionality reduction, a 3x3 convolution, and another 1x1 convolution for dimensionality restoration. The use of bottleneck blocks allows for a significant reduction in the number of parameters and computational cost compared to basic blocks. In a bottleneck block, the first 1x1 convolution reduces the number of channels, the 3x3 convolution operates on the reduced channels, and the second 1x1 convolution restores the number of channels. Batch normalization as well as ReLU activation are applied to each convolutional layer afterwards, except for the last 1x1 convolution, which is followed only by batch normalization. The output of the last 1x1 convolution is then added to the input of the block via the shortcut connection, and ReLU activation is applied to the sum. A summarized description of the variations for the ResNet architecture is shown in Table 4 and the most popular ResNet architecture, ResNet50, is shown in Figure 4.

Model	Depth	Input Resolution	Residual Blocks	Parameters (M)
ResNet18	18	224 x 224	8 (2 layers each)	11.7
ResNet34	34	224 x 224	16 (2 layers each)	21.8
ResNet50	50	224 x 224	16 (3 layers each)	25.6
ResNet101	101	224 x 224	33 (3 layers each)	44.5
ResNet152	152	224 x 224	50 (3 layers each)	60.2

Table 4 ResNet Architecture Model Variations

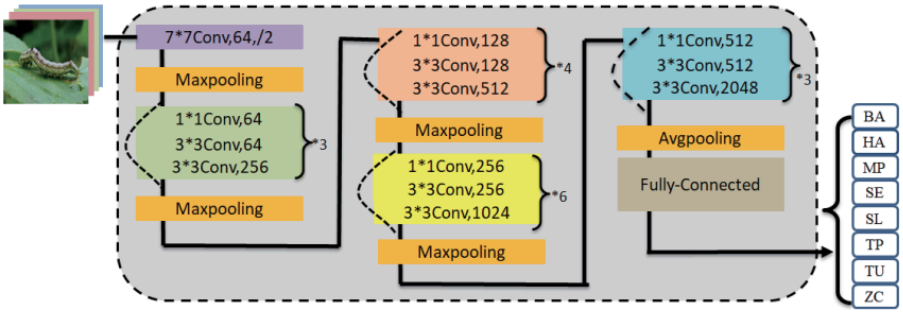


Figure 4 ResNet50 Architecture Model Diagram (Huang & Chen, 2024)

ResNet's effectiveness in identifying plant diseases stems from its capacity to construct and train exceptionally deep neural architectures without performance decline. Its skip connections enhance gradient propagation, allowing the network to capture more intricate and distinctive features. The incorporation of batch normalization and global average pooling further refines the model, mitigating overfitting issues. Despite these advantages, deeper ResNet variants like ResNet101 and ResNet152 can be computationally intensive. This poses challenges for deployment on devices with limited resources or in scenarios requiring real-time processing. To tackle this, the research community has explored various optimization strategies. These include compressing the model, applying quantization techniques, and leveraging knowledge distillation. Such approaches aim to reduce computational demands while preserving high levels of accuracy.

2.3.3 EfficientNet

EfficientNet is a family of CNN architectures introduced by Tan and Le in their 2019 paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks" (Tan & Le, 2019). The main goal of EfficientNet is to develop a more efficient and scalable network architecture that “balances network depth, width, and resolution to achieve better performance with fewer parameters" (William Edwards II & Dinc, 2020). EfficientNet's fundamental innovation lies in its uniform scaling approach. This method simultaneously adjusts three key network dimensions of depth, width, and input resolution using a consistent set of scaling factors. This approach allows for the development of a family of models, from the base EfficientNet-B0 to the larger EfficientNet-B7, with each model achieving better performance than the previous one while maintaining a balance between network depth, width, and resolution. EfficientNet architectures are built using mobile inverted bottleneck

(MBConv) blocks, which were introduced in the MobileNetV2 architecture (Sandler et al., 2018). MBConv blocks consist of a sequence of depthwise and pointwise convolutions, with squeeze-and-excitation (SE) blocks added to improve channel interdependencies. The architecture of EfficientNet can be summarized as follows: the network begins with a convolutional layer followed by multiple layers of MBConv blocks. The number of MBConv blocks and their configurations vary depending on the specific EfficientNet model. To conclude the network architecture, following the MBConv modules, a global average pooling operation is applied. This is then succeeded by a densely connected layer that performs the final classification task.

The architecture of a general EfficientNet model consists of several key components. The input layer accepts images with a specified resolution, which varies depending on the specific EfficientNet model. The stem layer, which is the first convolutional layer in the network, has a larger kernel size and a higher number of output channels compared to the subsequent layers, helping to capture low-level features and reduce the spatial dimensions of the input image. The main body of the EfficientNet architecture consists of a series of mobile inverted bottleneck (MBConv) blocks, each composed of an expansion layer (1x1 pointwise convolution), a depthwise convolution (3x3), a squeeze-and-excitation (SE) block, and a pointwise convolution (1x1). The SE block improves channel interdependencies by adaptively recalibrating channel-wise feature responses. Skip connections are added within the MBConv blocks to allow for faster convergence and better gradient flow. Between the MBConv blocks, reduction blocks are used to decrease the spatial dimensions of the features and increase the number of channels, typically consisting of a depthwise convolution with a stride of 2, followed by a pointwise convolution. The network's concluding section follows the final MBConv block. It incorporates a global average pooling operation, succeeded by a densely connected layer. This layer employs softmax activation to transform the extracted features into class probability distributions, facilitating the final classification step.

The EfficientNet family includes eight models, from B0 to B7, with increasing complexity and performance. The main differences between these models lie in the number of MBConv blocks, the depth of the network, the width of the channels, and the input image resolution. The compound scaling method is used to determine the optimal scaling coefficients for each of these dimensions, ensuring a balance between performance and efficiency. A summarized

description of the variations of EfficientNet architecture is shown in Table 5, and in Figure 5, the general variation of the EfficientNet architecture, which is EfficientNet-B0, is shown.

Model	Depth	Width Multiplier	Input Resolution	MBConv Blocks	Parameters (M)
EfficientNet-B0	18	1.0	224 x 224	7	5.3
EfficientNet-B1	23	1.0	240 x 240	8	7.8
EfficientNet-B2	28	1.1	260 x 260	9	9.2
EfficientNet-B3	33	1.2	300 x 300	10	12.0
EfficientNet-B4	38	1.4	380 x 380	11	19.0
EfficientNet-B5	43	1.6	456 x 456	12	30.0
EfficientNet-B6	51	1.8	528 x 528	14	43.0
EfficientNet-B7	66	2.0	600 x 600	18	66.0

Table 5 EfficientNet Architecture Model Variations

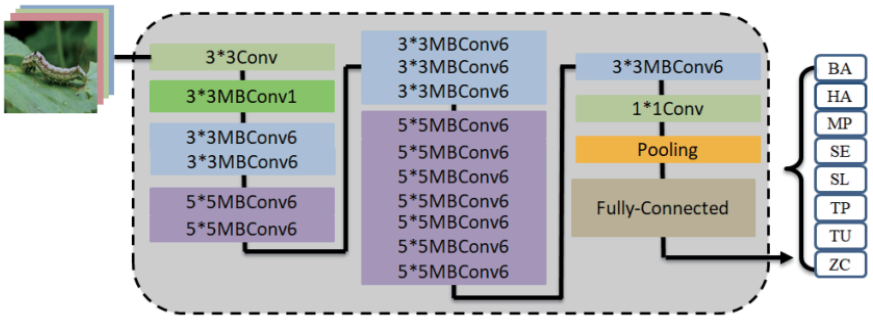


Figure 5 EfficientNet-B0 Architecture Model Diagram (Huang & Chen, 2024)

The compound scaling method used in EfficientNet is based on the observation that scaling up any single dimension of network depth, width, or resolution improves performance, but the improvement diminishes beyond a certain point. Therefore, to achieve better performance with limited resources, it is more effective to scale up all three dimensions in a balanced way. The compound scaling method determines the optimal scaling coefficients for depth, width, and resolution based on a small grid search, and these coefficients are then used to scale up the base model to obtain the larger EfficientNet models. EfficientNet has been shown to achieve state-of-the-art performance on various image classification tasks, including the

ImageNet dataset, while being much smaller and more efficient than other CNN architectures. The efficiency of EfficientNet makes it particularly well-suited for resource-constrained environments and applications that require real-time performance.

The success of EfficientNet in plant disease detection can be attributed to its efficient and scalable architecture, which allows for the development of models with varying complexity and performance. The compound scaling method used in EfficientNet ensures a balance between network depth, width, and resolution, enabling the models to learn rich and discriminative features while maintaining computational efficiency. Furthermore, the use of MBConv blocks in EfficientNet architectures helps to reduce the number of parameters and computational cost compared to traditional convolutional blocks. The depthwise and pointwise convolutions in MBConv blocks allow for efficient feature extraction and channel mixing, while the squeeze-and-excitation blocks help to improve channel interdependencies and feature recalibration. However, EfficientNet can still be prone to overfitting, especially when trained on small or imbalanced datasets. Techniques such as data augmentation, regularization, and transfer learning can be employed to mitigate overfitting and improve the generalization ability of EfficientNet models in plant disease detection tasks.

2.3.4 DenseNet

DenseNet was introduced by Huang et al. in their 2017 paper "Densely Connected Convolutional Networks" (Huang et al., 2017). The main idea behind DenseNet is to improve information flow and gradient propagation through the network by connecting each layer to the rest of the other layers in a straight direction. DenseNet's unique structure enables each layer to access feature maps generated by all prior layers. In turn, it forwards its own output to every subsequent layer. This comprehensive interconnection strategy promotes the reuse of features, leading to a more efficient use of parameters. Additionally, this dense connectivity helps alleviate the issue of vanishing gradients often encountered in deep networks. Additionally, DenseNet encourages feature propagation and exploration, as each layer has access to a collective knowledge of the network.

DenseNet's structure is built around interconnected modules known as dense blocks, with transition layers serving as intermediaries. The dense blocks feature a unique connectivity pattern where each layer links to all others within the block. A key parameter, the growth rate (k), defines the number of feature maps each layer produces. The network begins with an

input stage designed for specific image dimensions, often 224x224 pixels. This is followed by an initial convolution using large 7x7 filters with a stride of 2, complemented by normalization, ReLU activation, and max pooling operations. The core of DenseNet comprises a sequence of dense blocks. Within these, each layer utilizes the combined output from all preceding layers as its input. Transition layers positioned between blocks serve to adjust both the spatial dimensions and channel count of the feature maps, typically employing 1x1 convolutions and 2x2 average pooling. The network concludes with a global average pooling operation, condensing the spatial information. A final fully connected layer then performs the classification task. This comprehensive connectivity approach allows DenseNet to efficiently reuse features, minimize parameter count, and effectively combat the vanishing gradient problem often encountered in deep architectures.

Several DenseNet variants have been proposed, differing in the number of layers, growth rate, and the presence of bottleneck layers. DenseNet-121 is a variant with 121 layers, including 4 dense blocks with growth rate k=32. It also incorporates bottleneck layers before each 3x3 convolution to reduce computational complexity. DenseNet-169 is a variant with 169 layers, including 4 dense blocks with growth rate k=32 and bottleneck layers. DenseNet-201 is a variant with 201 layers, including 4 dense blocks with growth rate k=32 and bottleneck layers. DenseNet-264 is a variant with 264 layers, including 4 dense blocks with growth rate k=48 and bottleneck layers. A summarized description of the variations of DenseNet architecture is shown in Table 6 and Figure 6 showcases the common DenseNet architecture model variation which is DenseNet-121.

Model	Depth	Dense Block	Growth Rate (k)	Bottleneck Layers	Parameters (M)
DenseNet-121	121	4	32	Yes	7.95
DenseNet-169	169	4	32	Yes	14.15
DenseNet-201	201	4	32	Yes	20.01
DenseNet-264	264	4	48	Yes	33.34

Table 6 DenseNet Architecture Model Variations

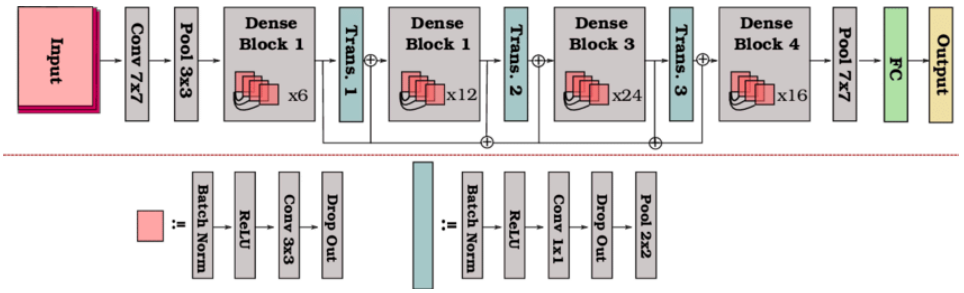


Figure 6 DenseNet-121 Architecture Model Diagram (Radwan, 2019)

The success of DenseNet in plant disease detection is due to it having a dense connectivity pattern that enables efficient feature reuse and gradient propagation. By connecting each layer to every other layer, DenseNet allows for the effective exploitation of features from different levels of abstraction, leading to improved performance and reduced overfitting. Furthermore, the use of dense blocks and transition layers in DenseNet architecture helps to manage the number of parameters and computational power needed to execute it, while the growth rate (k) manages the number of feature maps inserted from each layer, and the transition layers help to reduce the spatial dimensions and number of channels between dense blocks. However, like other deep architectures, DenseNet can be computationally expensive and memory-intensive in cases when dealing with images formatted to have high resolutions or datasets that are large, and using techniques such as model compression or pruning can help to mitigate these challenges.

2.4 Attention Mechanisms

In recent years, the deep learning community has shown growing interest in attention-based approaches, especially for visual computing tasks like classifying images and identifying objects. These techniques enhance neural networks by enabling them to prioritize the most pertinent input features, leading to improved effectiveness and computational efficiency. Among the various attention strategies, three have gained particular prominence which are, SENet, ECANet, and CBAM. These mechanisms show promise for advancing the field of automated plant disease identification.

2.4.1 SENet

Squeeze-and-Excitation Networks (SENet) introduced by Hu et al. in 2018 are a type of attention mechanism that adaptively recalibrates channel-wise feature responses by explicitly modeling the interdependencies between channels. The SENet architecture consists of two

key components which are a squeeze operation and an excitation operation. The squeeze operation aggregates the spatial information of each feature map into a single numerical value, efficiently summarizing the overall pattern of feature activations across channels. The excitation operation then uses these aggregated values to learn a set of channel-wise weights, which are subsequently used to recalibrate the original feature maps. By incorporating the SENet attention mechanism into existing CNN architectures, such as ResNet (Hu et al., 2018), significant improvements in performance have been observed across a wide range of computer vision tasks. In the context of plant disease detection, the ability of SENet to focus on the most informative channels and to model channel interdependencies can potentially enhance the network's ability to identify and localize disease symptoms, which may be manifest in specific color or texture changes (Too et al., 2019). The architecture model is illustrated in Figure 7.

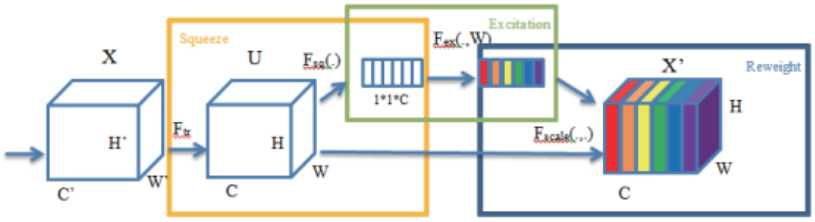


Figure 7 SENet Architecture Model Diagram (Huang & Chen, 2024)

2.4.2 ECANet

Efficient Channel Attention Networks (ECANet) was introduced by Wang et al. in 2020 are an extension of the SENet architecture that aims to improve the efficiency and effectiveness of channel attention. ECANet introduces a local cross-channel interaction strategy that captures the local dependencies between channels, in contrast to the global average pooling used in SENet. This local interaction is achieved through a 1D convolution operation along the channel dimension, allowing the network to learn a more fine-grained set of channel-wise weights. The ECANet attention mechanism has been shown to outperform SENet in terms of both accuracy and efficiency on several benchmark datasets (Wang et al., 2020). In the context of plant disease detection, ECANet's ability to model local channel dependencies may be particularly useful for capturing subtle variations in disease symptoms that may be localized to specific regions of the plant or leaf (Li et al., 2021). The ECANet model architecture is shown in Figure 8.

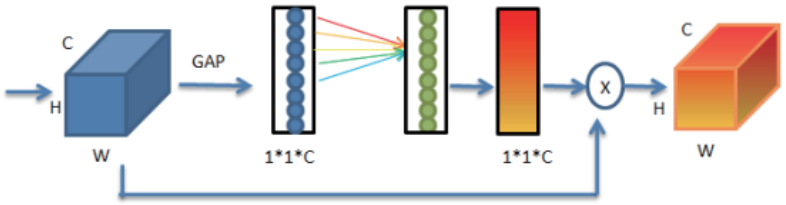


Figure 8 ECANet Architecture Model Diagram (Huang & Chen, 2024)

2.4.3 CBAM

The Convolutional Block Attention Module (CBAM) introduced by Woo et al. in 2018 is a more comprehensive attention mechanism that combines both channel-wise and spatial attention. CBAM consists of a channel attention module and a spatial attention module, which are applied sequentially to the input feature maps. The channel attention module is similar to SENet, using global average pooling and a multilayer perceptron to learn a set of channel-wise weights. The spatial attention module, on the other hand, uses a 2D convolution operation to learn a set of spatial weights, effectively focusing the network's attention on the most informative regions of the input. This attention mechanism has been applied to a wide range of computer vision tasks, demonstrating significant improvements in performance compared to baseline CNN architectures (Woo et al., 2018). The CBAM model is shown in Figure 9. In the context of plant disease detection, CBAM's ability to simultaneously model both channel-wise and spatial dependencies may be particularly beneficial for identifying and localizing disease symptoms that exhibit both color and texture variations (Li & Razi, 2019).

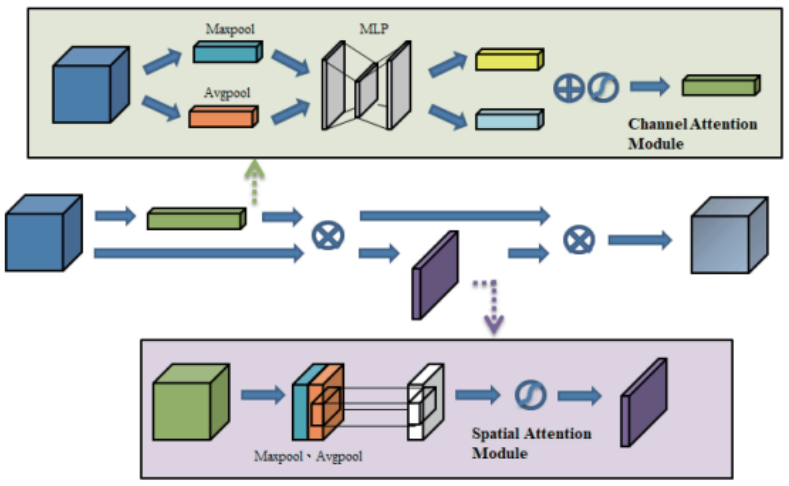


Figure 9 CBAM Architecture Model Diagram (Huang & Chen, 2024)

2.5 Evaluation Metrics

Numerous evaluation metrics are integral for assessing the performance of vision-based and classification techniques. Common performance indicators for assessing classification models encompass accuracy, precision, recall, and the F1-score. These measures collectively provide a thorough evaluation of a model's classification capabilities. These metrics derive from the outcomes of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). In instances of true positives, positive samples are correctly identified, while true negatives involve the accurate prediction of negative samples. Conversely, false positives occur when negative samples are erroneously classified as positive, and false negatives arise when positive samples are misclassified as negative. By leveraging these metrics, a nuanced evaluation of the classification model's performance becomes feasible, enabling a thorough understanding of its strengths and areas for improvement.

Accuracy stands as a foundational metric, calculated as the ratio of correct predictions to the total number of samples, which is equivalent to the summation of all possible outcomes. It is expressed as the total correctly recognized examples divided by the total examples in the dataset. This metric proves robust in assessing classifier performance, as it remains unaffected by the individual performance of specific classes. By quantifying the proportion of accurately predicted samples relative to the entire dataset, accuracy offers a straightforward yet insightful gauge of a model's overall effectiveness. Visualization of a classifier's performance becomes more accessible when its accuracy in predicting specific data groups is known. The formula for computing the accuracy of a classifier model is provided below, encapsulating its fundamental role in performance evaluation.

$$Accuracy = \frac{(TP + TN)}{TP + TN + FP + FN} = \frac{Total\ correctly\ classified\ data}{Total\ sample\ of\ data}$$

Precision quantifies a classifier's accuracy in identifying positive samples. It's determined by comparing the correctly identified positive instances to the total number of samples the model classified as positive. A high precision score suggests that when the model predicts a positive outcome, it's often correct. This metric is particularly useful in contexts where false positives are costly or undesirable. The mathematical expression for precision provides a concrete way to evaluate how well a model avoids labeling negative instances as positive. By focusing on

the proportion of true positives among all positive predictions, precision offers insight into the reliability of a model's positive classifications.

$$Precision = \frac{TP}{(TP + FP)} = \frac{Correctly\ classified\ positive\ data}{Total\ sample\ of\ positive\ data}$$

Recall, also known as sensitivity or true positive rate, is characterized by the ratio of correctly predicted positives to the actual positives, as delineated in the equation below. This metric centers on evaluating the classifier's efficacy in recognizing positive examples, gauging the relevance of the total positively labeled instances. A high recall value signifies that the classifier excels in accurately classifying positive labels. The recall formula, provided below, encapsulates this pivotal metric, offering a quantitative assessment of the model's ability to comprehensively identify positive instances.

$$Recall = \frac{TP}{(TP + FN)} = \frac{Correctly\ classified\ positive\ data}{Total\ actual\ sample\ of\ positive\ data}$$

The F1-score, a pivotal metric in performance evaluation, encapsulates the harmonic mean of precision and recall. The formula presented below illustrates how the F1-score synthesizes precision and recall metrics. This composite measure offers a nuanced evaluation, taking into account both the model's ability to identify positive instances and its tendency to avoid false positives. By balancing these two aspects, the F1-score provides a comprehensive assessment of the classifier's performance. The F1-score, a variant of the F-Score when β equals 1, acts as the harmonic mean of precision and recall. In the context of the baseline paper, which assesses classifier performance using F1-score for both disease and healthy classes, maintaining consistency, this metric is chosen for meaningful result comparisons. Additionally, the mean of the F1-score for both classes is calculated and recorded to facilitate a comprehensive comparison of classifier performance with models proposed in related studies. Precision and recall, presented alongside, contribute to a comprehensive understanding of classifier performance.

$$F1 = \frac{2TP}{2TP + FN + FP} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

In summary, this project employs a comprehensive set of performance metrics, including F1-score, Precision, Recall, and Accuracy, to offer a thorough evaluation of classifier performance. Despite the inclusion of multiple metrics, the comparative analysis specifically relies on F1 score when assessing the performances of different classifiers. This strategic focus on F1 score enables a nuanced and effective comparison, emphasizing a balanced consideration of both precision and recall in the evaluation process.

2.6 Related Works

Picon et al. (2019) curated a comprehensive dataset comprising 121,955 images from five distinct in-field crops. They employed ResNet50-based CNN models, enabling parallel processing of image information, resulting in an impressive balanced accuracy of 98% across the five crop datasets. Geetharamani and Pandian (2019) took a different approach, utilizing a compact CNN model featuring six layers. Through the application of various data augmentation techniques, they achieved a noteworthy classification accuracy of 96.46% across 39 classes. In a soybean disease classification context, Karlekar and Seal (2020) implemented a ten-layer CNN model after eliminating background interference from leaf images. Their meticulous approach yielded a remarkable test accuracy of 98.14%. Agarwal et al. (2020) directed their focus to tomato disease diagnosis within the PlantVillage dataset, employing a fundamental eight-layer CNN model. Their model demonstrated exceptional accuracy, reaching 98.4%. Hernández and López (2020) brought forth an innovative dimension by introducing probabilistic programming through Bayesian deep learning. Leveraging uncertainty with fine-tuning of the VGG16 model, they reported an accuracy of 96% using stochastic gradient descent (SGD) on the PlantVillage dataset, which comprised a substantial 54,306 images. This collective body of research underscores the diverse methodologies and substantial achievements in leveraging CNN models for crop disease classification.

Chen et al. (2020) devised a novel approach by amalgamating the initial two blocks of VGG16 with pre-trained weights from ImageNet and the Inception v3 module, initialized with random weights. Their model showcased remarkable accuracy across various datasets: 84.25% on the maize dataset from PlantVillage, 92% on the rice dataset, and 80.38% on the custom maize dataset. In a study by Sethy et al. (2020), a collection of 5932 in-field images of rice, categorized into four classes, became the focal point. The performance of 11 CNN

models, coupled with a SVM classifier, was systematically compared. ResNet50 paired with SVM emerged as the most successful model, boasting an impressive accuracy of 98.38%.

Scientists investigating plant pathology classification have employed a range of strategies centered around convolutional neural networks. For instance, a study by Radhakrishnan (2020) utilized AlexNet for feature extraction coupled with an SVM classifier to categorize rice diseases. This method, trained on an extensive set of 60,000 rice leaf images, achieved a 96.8% accuracy rate in testing. In a different approach, Barman et al. (2020) developed a custom CNN architecture that demonstrated impressive performance, reaching 99% accuracy when applied to a three-class citrus disease dataset. Taking yet another route, Ji et al. (2020) introduced a hybrid model that combined Inception v3 and ResNet 50 architectures. Their innovative fusion approach yielded a remarkable 98.57% accuracy in classifying grape diseases, setting a new benchmark in the field. These diverse studies highlight the versatility and effectiveness of CNN-based models in tackling the complex challenge of identifying crop diseases across various plant species.

Rangarajan et al. (2021) harnessed the capabilities of VGG16 for image feature extraction, coupled with a multi-class SVM for classification in the context of plant disease detection. Their model demonstrated significant prowess, achieving an accuracy of 91.3% when applied to a dataset comprising five classes within the eggplant domain. In addition, Alguliyev et al. (2021) delved into the fusion of CNN and gated recurrent unit (GRU) models, presenting a comprehensive approach to plant disease detection tasks. Their model, evaluated on the extensive PlantVillage dataset comprising 87,867 images, demonstrated an impressive accuracy of 91.19%. In a focused study on Vigna mungo disease severity diagnosis, Joshi et al. (2021) contributed three unique CNN models, namely VirLeafNet-1, VirLeafNet-2, and VirLeafNet-3. Their dedicated efforts resulted in impressive accuracies of 91.23%, 96.42%, and 97.40%, respectively. These achievements were recorded through meticulous evaluations on images sourced from Unmanned Aerial Vehicles (UAV), showcasing the versatility of CNN models in addressing diverse plant disease detection challenges.

Author	Classification Model	Dataset	Accuracy (%)
Picon et al. (2019)	ResNet50	Self	98.00
Geetharamani and Pandian (2019)	CNN	PlantVillage	96.46
Karlekar and Seal (2020)	CNN	PDDB	98.14
Agarwal et al. (2020)	CNN	PlantVillage	98.40
Hernández and López (2020)	VGG16	PlantVillage	96.00
Chen et al. (2020)	VGG19, InceptionV3	Self	92.00
Sethy et al. (2020)	CNN, MobileNetV2, ResNet50	Self	98.38
Radhakrishnan (2020)	AlexNet	Self	96.80
Barment et al. (2020)	CNN, MobileNetV2	Self	99.00
Ji et al. (2020)	InceptionV3, ResNet50	Self	98.57
Rangarajan et al. (2021)	VGG16	Self	94.30
Alguliyev et al. (2021)	CNN	PlantVillage	91.19
Joshi et al. (2021)	CNN	Self	97.40

Table 7 Deep Learning Classification Models

An extensive examination of the literature unveils that CNN models for plant disease detection draw inspiration predominantly from established architectures such as VGGNet, ResNet, DenseNet, and EfficientNet. Notably, numerous models integrate image pre-processing techniques in conjunction with CNNs. In addition to relying on CNNs for feature extraction, a subset of models incorporates additional techniques for feature selection. Remarkably, the prevailing trend in CNN model development involves the utilization of either subsets or the entirety of the PlantVillage dataset. Some researchers have gone a step further, contributing to the creation of novel datasets specifically tailored for plant disease detection while concurrently developing and refining their CNN models. This diverse approach showcases the adaptability and continual evolution of CNN-based methodologies in the dynamic landscape of plant disease detection research.

2.7 Chapter Summary

This literature review provided an in-depth exploration of the key topics and research related to developing a deep learning-based system for tomato plant disease detection. The review began by discussing data acquisition methods in Section 2.1, highlighting the importance of high-quality datasets for successful deep learning projects. It examined several publicly available plant disease datasets, such as PlantVillage, PlantDoc, FieldPlant, and PDD271, comparing their characteristics and suitability for training deep learning models. The review also emphasized the significance of data collection methods, contrasting the advantages and limitations of capturing images in controlled laboratory environments versus real field conditions.

Section 2.2 focused on data pre-processing techniques, which play a crucial role in enhancing the performance of deep learning models. The review covered various aspects of pre-processing, including noise reduction using filters and morphological operations, and feature extraction methods such as HOG, SIFT, and SURF. It also discussed the importance of data manipulation techniques, such as data cleaning and augmentation, in the context of deep learning. In Section 2.3, the review delved into Convolutional Neural Networks (CNNs), a powerful deep learning architecture for image classification tasks. It provided an overview of the key components and functioning of CNNs, as well as a comparison of popular CNN frameworks like TensorFlow, Keras, PyTorch, Caffe, and Deeplearning4j. The review then examined several prominent CNN architectures, including VGGNet, ResNet, EfficientNet, and DenseNet, discussing their unique features, advantages, and potential applications in plant disease detection.

Section 2.4 discussed the evaluation metrics commonly used to assess the performance of classification models, such as accuracy, precision, recall, and F1-score. It provided detailed explanations and formulas for each metric, emphasizing the importance of the F1-score as a balanced measure that combines precision and recall. Attention mechanisms, which allow neural networks to focus on the most relevant features of an input, were covered in Section 2.5. The review explored three prominent attention mechanisms: SENet, ECANet, and CBAM, discussing their architectures, functioning, and potential benefits for plant disease detection tasks.

Finally, Section 2.6 presented a comprehensive review of related works in the field of plant disease detection using deep learning techniques. It summarized the findings and methodologies of numerous studies, highlighting the diverse range of CNN architectures, datasets, and pre-processing techniques employed by researchers. The review identified the prevalent trend of utilizing established CNN architectures, such as VGGNet and ResNet, and the growing interest in developing novel datasets specifically created for plant disease detection. In conclusion, this literature review provided a solid foundation for understanding the current state of research in deep learning-based tomato plant disease detection. The study highlighted critical areas requiring further investigation, including the demand for extensive, varied data collections that reflect authentic field environments. Additionally, it underscored the promising avenue of incorporating attention-based techniques into convolutional neural network structures to enhance their efficacy in identifying plant diseases. The insights gained from this review will inform the methodology and approach for developing an enhanced deep learning model for accurate and efficient tomato plant disease detection.

3.0 METHODOLOGY

In this section, the various development steps taken to successfully carry out this project according to the scopes, objectives and problems stated beforehand will be discussed as well as elaborated in detail including proper justifications for the methods used in order to properly convey their importance when completing the project without any major issues. The majority of the development choices selected were based on the literature review discussions and anything that was not discussed in the literature review will be properly supported and justified as to the reason they were selected. There are two parts for this project which consists of the deep learning portion and the system development portion, and each of them will have their respective methodologies that are not necessarily related to one another. In Figure 10, the illustration of the proposed methodology used to develop the deep learning proposed model that is later used in the web application for the user to detect selected tomato plant diseases is shown, whereas in Figure 11, the diagram showcasing the flowchart of the proposed web application that allows users to upload images and test out the model's capabilities is displayed. As a disclaimer, some of the steps in the flowchart are not complete and detailed due to the reason that they are merely used to represent significant milestones that must be achieved in order to complete the whole project.

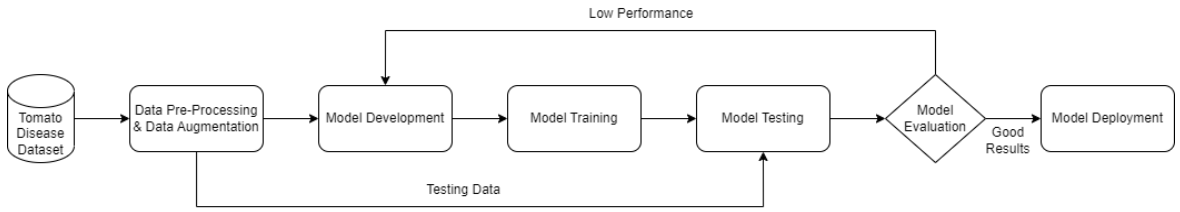


Figure 10 Proposed Deep Learning Methodology

Based on Figure 10, the first step is to use an existing or create a new tomato plant dataset and afterwards perform data pre-processing as well as data augmentation techniques on the images. The image dataset will be split into two distinct groups with one being used to train the model and the other used for testing the trained model. Next, the proposed CNN model is developed according to the specifications of the image dataset as well as the curated choices of CNN architectures and when it has been fully developed, the model will then be trained with the images assigned for training purposes. Afterwards, the trained model will be tested with the predetermined images for model testing and the testing result will also be evaluated according to the selected metrics. If the model is not able to produce the expected results

when evaluating, the model will be configured again until the required or satisfactory results are obtained. The final stage would be to deploy or save the model that manages to produce the best overall results in order to use it in the web application later.

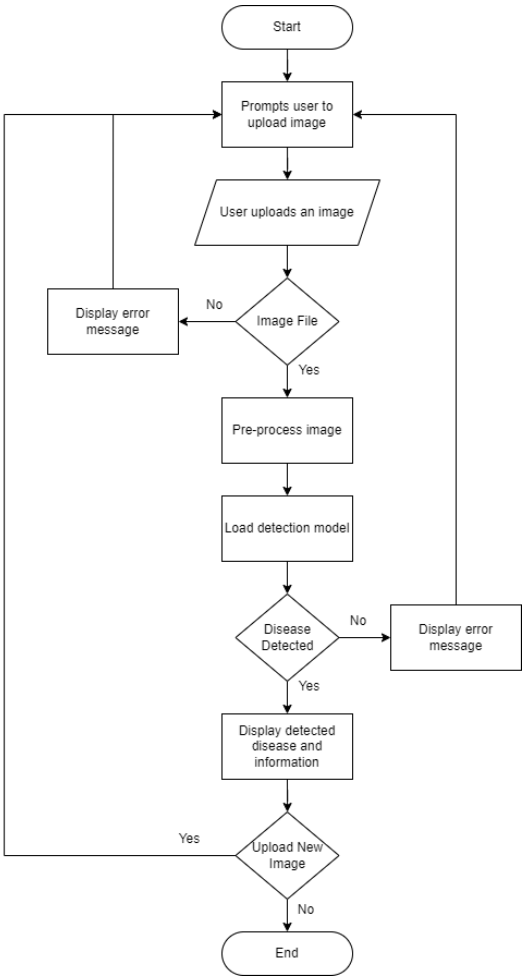


Figure 11 Proposed Web Application Methodology

In Figure 11, the general flowchart of the web application system is shown and it starts with prompting the user to upload an image of a tomato plant that contains a disease into the web application. The system will then validate the user's input and ensure that it is indeed an image file by sending an error message to notify the user that the uploaded file is incorrect or proceed to the next step of the flowchart. When the image has been uploaded, the system will then perform some simple image pre-processing techniques on the image such as resizing it to better suit the model dimension requirements. Moving on, the system will load the CNN model and use it to detect the disease on the tomato plant, and if there are not proper plant images detected on the image, the system will send another error message to inform the user however, if there is a plant leaf on the image, the system will proceed as intent. The results of

the CNN model will then be displayed to the user including some additional information regarding the disease as well as other information related to the model itself. At the end, the user is allowed to upload a new image into the web application again or terminate the web application.

3.1 Environment & Hyperparamater Specifications

The main programming language used in this project is the Python programming language for both the deep learning tasks and the web application development. Both of them used certain frameworks and libraries that were imported using Python such as the TensorFlow and Keras library for the deep learning portion, and the Flask library for the web application portion. Throughout the entire project, the main operating system used to carry out all the tasks were done in Windows 10 and the hardware specifications are an Intel Core i5-6500 CPU with 3.20 GHz for the CPU processor and 16 GB of RAM however, when training the CNN model, the Google Collab IDE was used due to its free allocation of 15 GB of GPU resources for their users. Table 8 represents the experimental setup of both the deep learning and web application portion of this project including the versions of the main frameworks and libraries used, whereas Table 9 represents the hyperparameters used in the developing the model.

Environmental Setup	Specifications
Operating System	Windows 10
CPU	Intel Core i5-6500, 3.20 GHz
Memory	16 GB
GPU Model	Tesla T4
GPU Memory	15 GB
Python Version	3.8.16
TensorFlow Version	2.13.0
Keras Version	2.13.0
Flask Version	3.0.2

Table 8 Environmental Setup Specifications

Hyperparameters Setup	Specifications
Number of Classes	10
Model Input Dimensions	(224 x 224 x 3)
Batch Size	40
Learning Rate	0.001
Dropout Rate	0.45
Epochs	5
Optimizer	Adamax
Loss Function	Categorical Cross-Entropy
Horizontal Flip	True
Vertical Flip	True
Rotation Range	30
Width Shift Range	0.2
Height Shift Range	0.2
Shear Range	0.2
Zoom Range	0.2

Table 9 Hyperparameters Setup Specifications

3.2 Dataset Acquisition

The tomato dataset selection for this project is from the popular PlantVillage dataset. The PlantVillage dataset contains images from various plant types, including the tomato plant, meaning that by isolating the required plant type and removing the other unnecessary plant types, it is possible to use this dataset for training as well as testing purposes for the proposed model. There are roughly 10,000 combined images across 10 classes of diseased tomato plants in the dataset, with each class having roughly 1,000 images. Out of the 10 tomato plant classes in the dataset, 9 are disease classes, while the remaining class contains images of healthy tomato plant leaves. Each of the images in the dataset contains a singular tomato leaf that is or is not infected with a disease, and the tomato leaves predominantly occupy the center position in the image.

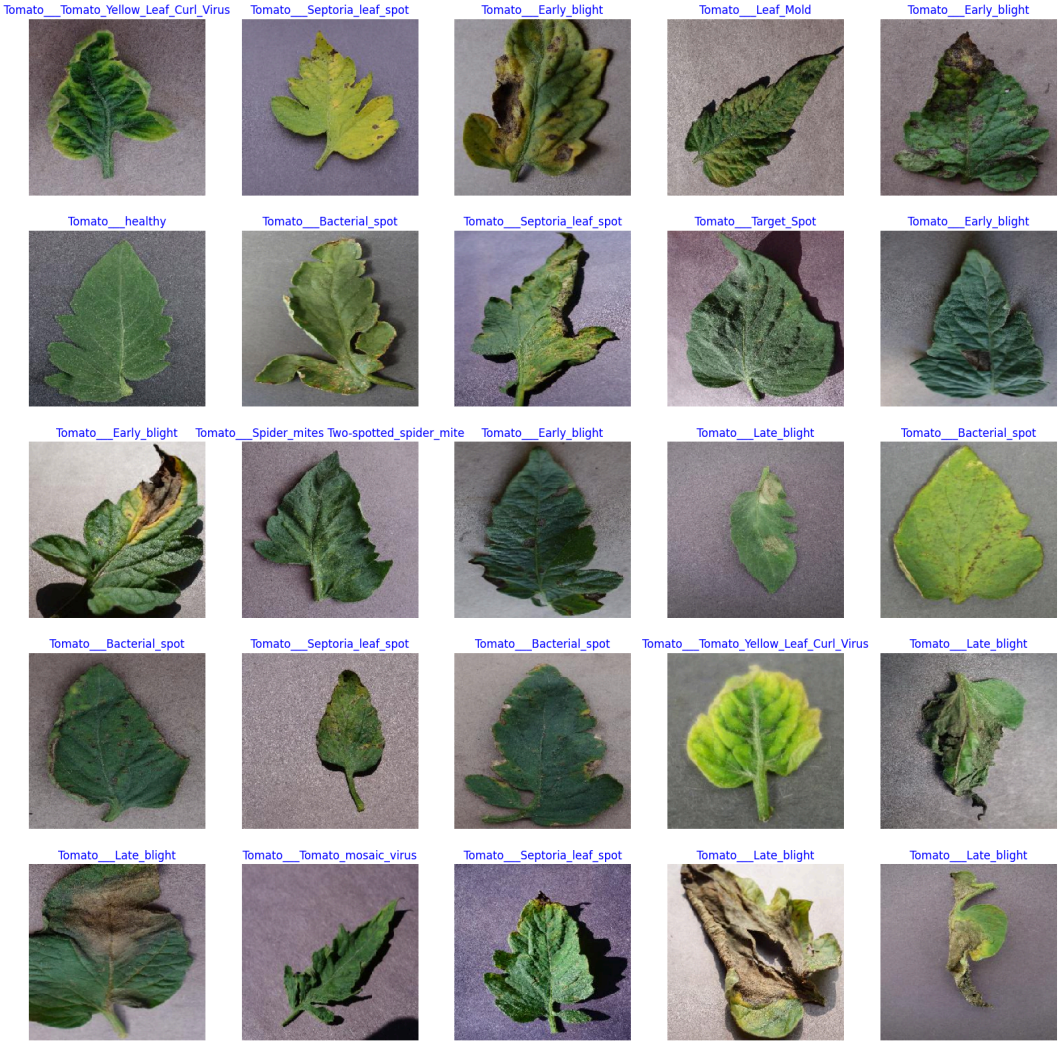


Figure 12 Dataset Class Sample Images

The 9 tomato plant disease class images that are within the dataset are the “tomato bacterial spot”, the ”tomato early blight”, the ”tomato late blight”, the ”tomato leaf mold”, the ”tomato septoria leaf spot”, the ”tomato two-spotted spider mites”, the ”tomato target spot”, the ”tomato mosaic virus”, and the ”tomato yellow leaf curl virus”. As stated previously, the last remaining class is labeled as “tomato healthy” which contains images of healthy tomato leaves. The sample images from the 10 classes are shown in Figure 12 in no particular order or quantity and were taken from the dataset at random. Furthermore, the dataset was then split into two categories with one being used to train the model and the other was used to test the trained model respectively. Approximately 60% of the images in the dataset was used to validate the model while the remaining 40% was used to test the model.

3.3 Data Pre-Processing

Moving on, this project employed various data pre-processing techniques in order to prepare the tomato plant disease dataset images for training the model. These techniques include data collection, DataFrame creation, data augmentation, image resizing, dataset splitting, and the use of data generators. The first pre-processing technique that is being utilized is to create the proper pathway to the dataset as well as creating the DataFrames needed to load the dataset images and classes. This is important as it sets up the necessary foundation so that other pre-processing techniques are able to be applied to the images.

```
def define_paths(dir):
    filepaths = []
    labels = []
    folds = os.listdir(dir)
    for fold in folds:
        foldpath = os.path.join(dir, fold)
        filelist = os.listdir(foldpath)
        for file in filelist:
            fpath = os.path.join(foldpath, file)
            filepaths.append(fpath)
            labels.append(fold)
    return filepaths, labels
```

Figure 13 define_paths Function

To isolate the required tomato plant images, the “define_paths” function is implemented to retrieve the file paths and corresponding labels for each image in the dataset. The “define_paths" function takes a directory path as input and recursively traverses through the subdirectories to collect the file paths and labels of the images. The file paths are later stored in the “filepaths” list variable, and the corresponding labels are stored in the “labels” list variable. The relevant code is displayed in Figure 13.

```
def define_df(files, classes):
    Fseries = pd.Series(files, name= 'filepaths')
    Lseries = pd.Series(classes, name='labels')
    return pd.concat([Fseries, Lseries], axis= 1)
```

Figure 14 define_df Function

After collecting the file paths and labels, the next step is to create a DataFrame to organize the data. The “define_df” function is used to create a DataFrame by combining the file paths and labels into a single structure. The “define_df” function takes the lists of file paths from the “files” local variables and labels from the “classes” local variables as input and creates two Pandas Series objects named “Fseries” for file paths and “Lseries” for labels. These series are then concatenated along the column axis to form a DataFrame. The corresponding relevant code is shown in Figure 14.

```
def create_df(tr_dir, val_dir):
    # train dataframe
    files, classes = define_paths(tr_dir)
    train_df = define_df(files, classes)

    # valid and test dataframe
    files, classes = define_paths(tr_dir)
    dummy_df = define_df(files, classes)
    strat = dummy_df['labels']
    valid_df, test_df = train_test_split(dummy_df,  train_size= 0.6, shuffle= True, random_state= 123, stratify= strat)
    return train_df, valid_df, test_df
```

Figure 15 create_df Function

The “create_df” function is then used to create separate DataFrames for training, validation, and testing. It utilizes the “define_paths” and “define_df” functions to collect the data and create the DataFrames. The training DataFrame named “train_df” is created using the entire dataset, while the validation and test DataFrames named “valid_df” and “test_df” respectively are created by splitting the remaining data using the “train_test_split” function from the scikit-learn library. The “create_df” function takes the paths to the training and validation directories known as “tr_dir” and “val_dir” respectively as input. It also creates the “train_df” using the entire training directory and creates a “dummy_df” using the same directory. The “dummy_df” is then split into “valid_df” and “test_df” using the “train_test_split” function, with 60% of the data used for validation and the remaining 40% used for testing. The stratify parameter is set to the labels column of the “dummy_df” to ensure that the class distribution is preserved during the splitting process. The entire code for the “create_df” function is shown in Figure 15.

```
def create_gens(train_df, valid_df, test_df, batch_size):
    img_size = (224, 224)
    channels = 3
    img_shape = (img_size[0], img_size[1], channels)
    ts_length = len(test_df)
    test_batch_size = test_batch_size = max(sorted([ts_length // n for n in range(1, ts_length + 1) if ts_length%n == 0 and ts_length/n <= 80]))
    test_steps = ts_length // test_batch_size
    def scalar(img):
        return img
    # Added more augmentations
    tr_gen = ImageDataGenerator(preprocessing_function= scalar,
                                horizontal_flip=True,
                                vertical_flip=True,
                                rotation_range=30,
                                width_shift_range=0.2,
                                height_shift_range=0.2,
                                shear_range=0.2,
                                zoom_range=0.2,
                                fill_mode='nearest')
    ts_gen = ImageDataGenerator(preprocessing_function= scalar)
    train_gen = tr_gen.flow_from_dataframe( train_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                color_mode= 'rgb', shuffle= True, batch_size= batch_size)
    valid_gen = ts_gen.flow_from_dataframe( valid_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                color_mode= 'rgb', shuffle= True, batch_size= batch_size)
    test_gen = ts_gen.flow_from_dataframe( test_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                color_mode= 'rgb', shuffle= False, batch_size= test_batch_size)
    return train_gen, valid_gen, test_gen
```

Figure 16 create_gens Function

The next important step of the pre-processing stage occurs in the “create_gens” function which is mainly responsible for creating data generators for training, validation, and testing. It takes the training, validation, and test DataFrames stored in the local variables named “train_df”, “valid_df”, “test_df” and the batch size under the local variable named “batch_size” as input parameters. The function starts by defining the image size as 224 by 224 and stores it under the variable named “img_size”. The “channels” variable is set to 3 in order to indicate that the images have three color channels or commonly known as the Red-Blue-Green (RGB) color format. This means that the “img_shape” variable is created as a tuple combining the “img_size” and “channels” values. Furthermore, the length of the “test_df” test DataFrame is calculated and stored in the “ts_length” variable, and this length is used to determine the appropriate batch size for the test generator. The third stage is to calculate the “test_batch_size” variable by using a list comprehension and the “max” function. It finds the maximum value among the factors of the “ts_length” variable that result in a batch size less than or equal to 80. This ensures that the test batch size is a factor of the test set length and is not too large. The “test_steps” variable is then calculated by dividing “ts_length” by “test_batch_size”, and this represents the number of steps or batches needed to cover the entire test set. The scalar function is defined as a preprocessing function that simply returns the input image without any modifications and it is used as the “preprocessing_function” parameter in the ‘ImageDataGenerator’ instances. Afterwards, two instances of the ‘ImageDataGenerator’ class are created named “tr_gen” for training and “ts_gen” for validation and testing respectively. The “tr_gen” instance is configured with

various data augmentation techniques such as horizontal and vertical flips, rotation, width and height shifts, shearing, zooming, and filling mode, whereas the “ts_gen” instance only applies the scalar preprocessing function without any data augmentation. The “train_gen” generator is later created using the “flow_from_dataframe” method of the “tr_gen” instance by taking the “train_df” DataFrame as input in order to specify the file paths column and labels column or also known as “x_col” and “y_col” respectively. The “target_size” is set to “img_size”, “class_mode” is set to 'categorical' for multi-class classification, “color_mode” is set to 'rgb' for color images, “shuffle” is set to ‘True’ to shuffle the data after each epoch, and “batch_size” is set to the input “batch_size”. Similarly, the “valid_gen” generator is created using the “flow_from_dataframe” method of the “ts_gen” instance and it uses the “valid_df” DataFrame as input as well as having the same parameters as “train_gen”. The “test_gen” generator is also created using the “flow_from_dataframe” method of the “ts_gen” instance and it also uses the “test_df” DataFrame as input and has similar parameters as “train_gen” and “valid_gen” with the exception that “shuffle” is set to ‘False’ to preserve the order of the test set, and “batch_size” is set to “test_batch_size”. Finally, the function returns the “train_gen”, “valid_gen”, and “test_gen generators”. The code within the “create_gens” function is displayed in Figure 16.

3.4 Convolutional Neural Network Structure

In this project, the proposed model architecture for tomato plant disease detection is based on the EfficientNet-B3 convolutional neural network (CNN). EfficientNet is a family of CNN architectures introduced by Tan and Le in their 2019 paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks" (Tan & Le, 2019). The main goal of EfficientNet is to develop a more efficient and scalable network architecture that balances network depth, width, and resolution to achieve better performance with fewer parameters. The choice of EfficientNet-B3 as the base architecture for this project is motivated by several factors. Firstly, EfficientNet models have demonstrated adequate results on various image classification tasks, including plant disease detection, while being more efficient in terms of computational resources compared to other CNN architectures (Tan & Le, 2019). This efficiency is particularly important when considering the deployment of the model in resource-constrained environments or real-time applications.

In addition to the base EfficientNet-B3 architecture, this project incorporates transfer learning to leverage the knowledge gained from pre-training on a large-scale dataset. Transfer learning is a technique where a model trained on a source task is repurposed and fine-tuned for a target task, often resulting in improved performance and faster convergence (Pan & Yang, 2009). By using pre-trained weights from the ImageNet dataset, which contains a vast collection of diverse images, the EfficientNet-B3 model can benefit from the learned features and representations, reducing the need for extensive training data and computational resources. In addition to transfer learning, the proposed methodology incorporates the CBAM attention mechanism introduced by Woo et al. in 2018 in order to improve the model's ability to focus on discriminative regions and features even further. CBAM is an attention mechanism that adaptively recalibrates the importance of features along both the channel and spatial dimensions. By integrating CBAM into the EfficientNet-B3 architecture, the model can learn to emphasize informative features and suppress irrelevant ones, leading to enhanced disease detection performance.

```python
def cbam_block(input_tensor, ratio=8):
    # Channel Attention Module
    channel_axis = 1 if K.image_data_format() == "channels_first" else -1
    channel = input_tensor.shape[channel_axis]

    shared_layer_one = Conv2D(channel // ratio, 1, activation='relu', kernel_initializer='he_normal', use_bias=True, bias_initializer='zeros')
    shared_layer_two = Conv2D(channel, 1, kernel_initializer='he_normal', use_bias=True, bias_initializer='zeros')

    avg_pool = GlobalAveragePooling2D()(input_tensor)
    avg_pool = Reshape((1, 1, channel))(avg_pool)
    avg_pool = shared_layer_one(avg_pool)
    avg_pool = shared_layer_two(avg_pool)

    max_pool = GlobalMaxPooling2D()(input_tensor)
    max_pool = Reshape((1, 1, channel))(max_pool)
    max_pool = shared_layer_one(max_pool)
    max_pool = shared_layer_two(max_pool)

    cbam_feature = Add()([avg_pool, max_pool])
    cbam_feature = Activation('sigmoid')(cbam_feature)

    if K.image_data_format() == "channels_first":
        cbam_feature = Permute((3, 1, 2))(cbam_feature)

    x = Multiply()([input_tensor, cbam_feature])

    # Spatial Attention Module
    avg_pool = Lambda(lambda x: K.mean(x, axis=3, keepdims=True))(x)
    max_pool = Lambda(lambda x: K.max(x, axis=3, keepdims=True))(x)
    spatial_attention = Concatenate(axis=3)([avg_pool, max_pool])
    spatial_attention = Conv2D(1, 7, strides=1, padding='same', activation='sigmoid', kernel_initializer='he_normal', use_bias=False)(spatial_attention)

    x = Multiply()([x, spatial_attention])

    return x
```

Figure 17 cbam_block Function

The “cbam_block” function shown in Figure 17 has two main components which are the channel attention module and the spatial attention module. The function starts by determining the channel axis based on the image data format. If the format is "channels_first", the channel axis is set to 1, otherwise, it is set to -1. The number of channels in the input tensor is

obtained using “input_tensor.shape[channel_axis]” and stored in the “channel” variable. Two shared layers, “shared_layer_one” and “shared_layer_two”, are defined using ‘Conv2D’ with a kernel size of 1. These layers are used to learn the channel attention weights. The first layer reduces the number of channels by a ratio, commonly defaulted as 8, and applies a ReLU activation function, while the second layer restores the number of channels. Afterwards, the global average pooling is applied to the input tensor using the “GlobalAveragePooling2D” function, which reduces the spatial dimensions and produces a tensor of shape and the resulting tensor is then reshaped using the “Reshape” function. The reshaped average-pooled tensor is passed through the shared layers of “shared_layer_one” and “shared_layer_two” respectively in order for them to store the channel attention weights. Similarly, global max pooling is applied to the input tensor using the “GlobalMaxPooling2D” function, and the resulting tensor is reshaped and passed through the shared layers. The outputs of the shared layers for both average and max pooling are added element-wise using the “Add” function, and the output then undergoes processing through a sigmoid activation, yielding channel-specific attention coefficients. If the image data format is "channels_first", the channel attention weights are permuted using the “Permute” function in order to match the shape of the input tensor. Furthermore, the channel attention weights are multiplied element-wise with the input tensor using the “Multiply” function to apply the channel-wise attention. Moreover, the spatial attention module starts by applying average pooling and max pooling along the spatial dimensions of the input tensor using the “Lambda” function layers. The “Lambda” function layers compute the mean and max values along the channel axis while keeping the dimensions intact. The average-pooled and max-pooled tensors are concatenated along the channel axis which is then passed through a convolutional layer with a kernel size of 7, stride of 1, "same" padding, and the layer calculates the spatial attention weights as well as applies a sigmoid activation function. The spatial attention weights are then multiplied element-wise with the input tensor and finally, the attended feature maps, which have been refined by both channel-wise and spatial attention, are returned.

```
def create_model(input_shape, class_count):
    base_model = tf.keras.applications.efficientnet.EfficientNetB3(include_top=False, weights="imagenet", input_shape=input_shape)

    # Adding CBAM block after the last convolutional layer of the base model
    x = base_model.output
    x = cbam_block(x)
    x = GlobalAveragePooling2D()(x)

    x = BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001)(x)
    x = Dense(256, kernel_regularizer=regularizers.l2(0.016), activity_regularizer=regularizers.l1(0.006),
              bias_regularizer=regularizers.l1(0.006), activation='relu')(x)
    x = Dropout(rate=0.45, seed=123)(x)
    output = Dense(class_count, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=output)
    model.compile(Adamax(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

    return model
```

Figure 18 create_model Function

The “create_model” function in Figure 18 takes the input shape and the number of classes as parameters named as “input_shape” and “class_count” respectively. Inside the function, the pre-trained EfficientNet-B3 model is instantiated from the TensorFlow and Keras libraries. The “include_top” parameter is set to ‘False’ in order to remove the last classification layer, allowing us to add custom layers on top. Next, the “weights” parameter is set to ‘imagenet’ to load the pre-trained weights from the ImageNet dataset and the “input_shape” is specified to match the shape of the input images. The output of the base model is obtained using “base_model.output”, which represents the features extracted by the EfficientNet-B3 network and the output of the base model is then passed through a CBAM function named “cbam_block” to apply channel-wise and spatial-wise attention. The implementation of the CBAM function has already been explained further in detail previously and the code snippet is also shown in Figure 17. Later, the output of the CBAM function is then passed through a global average pooling layer using the “GlobalAveragePooling2D” function to reduce the spatial dimensions and obtain a fixed-length feature vector. Moving on, batch normalization is applied using the “BatchNormalization” layer to normalize the activations and improve the stability of the network. The “BatchNormalization” layer has the “axis” parameter is set to -1 to perform normalization along the feature dimension while the “momentum” and “epsilon” parameters are set to their default values. Furthermore, a dense layer with 256 units and ReLU activation is added, along with L2 kernel regularization, L1 activity regularization, and L1 bias regularization. These regularization techniques help in preventing overfitting and improving the model's generalization ability. A dropout layer with a rate of 0.45 is also added to further regularize the model and reduce overfitting, and the “seed” parameter is set to 123 for reproducibility. The final dense layer with “class_count” units and softmax activation is added to produce the class probabilities for the tomato plant disease classes. The actual model

is created using the “Model” class by specifying the input and output layers, and is later compiled with the Adamax optimizer, a variant of the Adam optimizer with improved stability, using a learning rate of 0.001. The loss function is set to categorical cross-entropy, which is suitable for multi-class classification tasks, and accuracy is used as the evaluation metric before finally returning the compiled model.

```
# Create Model Structure
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)
class_count = len(list(train_gen.class_indices.keys())) # to define number of classes in dense layer

# create model with CBAM
model = create_model(img_shape, class_count)
model.summary()
```

Figure 19 Proposed Model Construction Code

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0         []

 rescaling (Rescaling)       (None, 224, 224, 3)       0         ['input_1[0][0]']

 normalization (Normalizati  (None, 224, 224, 3)       7         ['rescaling[0][0]']
 on)

 rescaling_1 (Rescaling)     (None, 224, 224, 3)       0         ['normalization[0][0]']

 stem_conv_pad (ZeroPadding  (None, 225, 225, 3)       0         ['rescaling_1[0][0]']
 2D)

 stem_conv (Conv2D)          (None, 112, 112, 40)      1080      ['stem_conv_pad[0][0]']

 stem_bn (BatchNormalizatio  (None, 112, 112, 40)      160       ['stem_conv[0][0]']
 n)

 stem_activation (Activatio  (None, 112, 112, 40)      0         ['stem_bn[0][0]']
 n)
...
Total params: 11777273 (44.93 MB)
Trainable params: 11686898 (44.58 MB)
Non-trainable params: 90375 (353.03 KB)
__________________________________________________________________________________________________
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 20 Proposed Model Construction Code Output

Figure 20 depicts the summary of the proposed model generated by executing the code shown in Figure 19. The visible portion of the proposed model summary shows the input layer with an output shape of (None, 224, 224, 3), indicating that the proposed model expects input

images of size 224x224 with 3 color channels. Subsequent layers such as rescaling, normalization, and additional rescaling are applied to the input, maintaining the same output shape. Due to the limited space in the image, the entire summary of the proposed model is not visible, however the proposed model contains additional layers such as the CBAM block and dense layers for classification, which are not currently visible in the figure. At the bottom of the proposed model summary, the total number of parameters in the model is displayed. The model has a total of 11,777,273 parameters with the size of 44.93 MB, out of which 11,686,898 with the size 44.58 MB are trainable parameters, and 90,375 with the size of 353.03 KB are non-trainable parameters. Trainable and non-trainable parameters in a model serve distinct purposes. Trainable parameters are adjustable values that the model refines during the learning process. These parameters evolve as the model is exposed to data, allowing it to adapt and improve its performance. In contrast, non-trainable parameters remain constant throughout the training phase. These fixed values are typically pre-determined and do not change in response to the training data. The distinction between these two types of parameters is crucial for understanding how a model learns and which aspects of its structure are malleable versus static during the training process.

By leveraging the pre-trained EfficientNet-B3 architecture and fine-tuning it for the specific task of tomato plant disease detection, this project aims to achieve high accuracy and efficiency. The transfer learning approach allows the model to benefit from the knowledge gained from a large-scale dataset, reducing the need for extensive training data and computational resources. In addition, the additional layers added on top of the base model, such as global average pooling, batch normalization, dense layers with regularization, and dropout, help in adapting the model to the specific characteristics of the tomato plant disease dataset and preventing overfitting. Additionally, by also incorporating the CBAM attention mechanism, the model is able to adaptively recalibrate the importance of features along both the channel and spatial dimensions. This enables the model to focus on the most informative regions and suppress irrelevant ones, leading to improved disease detection performance. The reason behind choosing to integrating CBAM into the EfficientNet-B3 architecture is supported by the literature review, which highlights the effectiveness of incorporating an attention mechanism in enhancing the performance of CNN models in various computer vision tasks, including plant disease detection (Woo et al., 2018). Overall, the proposed EfficientNet-B3 model with transfer learning as well as CBAM attention mechanism provides a powerful and efficient solution for detecting and classifying tomato plant diseases.

The combination of the EfficientNet architecture, which balances network depth, width, and resolution, along with the benefits of transfer learning, positions this model as a promising approach for accurate and reliable plant disease detection.

3.5 Data Training

This section outlines the data training process for the proposed model and focuses on two main aspects which are the implementation of a custom callback class and the execution of the model training. The custom callback is designed to monitor training progress, adjust learning rates, and provide detailed logging, while the training execution involves setting up key variables and using Keras's "fit" function to train the model.

```
class MyCallback(keras.callbacks.Callback):
    def __init__(self, model, base_model, patience, stop_patience, threshold, factor, batches, initial_epoch, epochs):
        super(MyCallback, self).__init__()
        self.model = model
        self.base_model = base_model
        self.patience = patience # specifies how many epochs without improvement before learning rate is adjusted
        self.stop_patience = stop_patience # specifies how many times to adjust lr without improvement to stop training
        self.threshold = threshold # specifies training accuracy threshold when lr will be adjusted based on validation loss
        self.factor = factor # factor by which to reduce the learning rate
        self.batches = batches # number of training batch to runn per epoch
        self.initial_epoch = initial_epoch
        self.epochs = epochs
        # callback variables
        self.count = 0 # how many times lr has been reduced without improvement
        self.stop_count = 0
        self.best_epoch = 1   # epoch with the lowest loss
        self.initial_lr = float(tf.keras.backend.get_value(model.optimizer.lr)) # get the initial learning rate and save it
        self.highest_tracc = 0.0 # set highest training accuracy to 0 initially
        self.lowest_vloss = np.inf # set lowest validation loss to infinity initially
        self.best_weights = self.model.get_weights() # set best weights to model's initial weights
        self.initial_weights = self.model.get_weights()   # save initial weights if they have to get restored

    # Define a function that will run when train begins
    def on_train_begin(self, logs= None):
        msg = '{0:^8s}{1:^10s}{2:^9s}{3:^9s}{4:^9s}{5:^9s}{6:^9s}{7:^10s}{8:10s}{9:^8s}'.format('Epoch', 'Loss', 'Accuracy', 'V_loss', 'V_acc', 'LR', 'Next LR', 'Monitor','% Improv', 'Duration')
        print(msg)
        self.start_time = time.time()

    def on_train_end(self, logs= None):
        stop_time = time.time()
        tr_duration = stop_time - self.start_time
        hours = tr_duration // 3600
        minutes = (tr_duration - (hours * 3600)) // 60
        seconds = tr_duration - ((hours * 3600) + (minutes * 60))
        msg = f'training elapsed time was {str(hours)} hours, {minutes:4.1f} minutes, {seconds:4.2f} seconds)'
        print(msg)
        self.model.set_weights(self.best_weights) # set the weights of the model to the best weights

    def on_train_batch_end(self, batch, logs= None):
        acc = logs.get('accuracy') * 100 # get batch accuracy
        loss = logs.get('loss')
        msg = '{0:20s}processing batch {1:} of {2:5s}-   accuracy=  {3:5.3f}   -   loss: {4:8.5f}'.format(' ', str(batch), str(self.batches), acc, loss)
        print(msg, '\r', end= '') # prints over on the same line to show running batch count

    def on_epoch_begin(self, epoch, logs= None):
        self.ep_start = time.time()

    # Define method runs on the end of each epoch
    def on_epoch_end(self, epoch, logs= None):
        ep_end = time.time()
        duration = ep_end - self.ep_start

        lr = float(tf.keras.backend.get_value(self.model.optimizer.lr)) # get the current learning rate
        current_lr = lr
        acc = logs.get('accuracy')  # get training accuracy
        v_acc = logs.get('val_accuracy')  # get validation accuracy
        loss = logs.get('loss')  # get training loss for this epoch
        v_loss = logs.get('val_loss')  # get the validation loss for this epoch

        if acc < self.threshold: # if training accuracy is below threshold adjust lr based on training accuracy
            monitor = 'accuracy'
            if epoch == 0:
                pimprov = 0.0
            else:
                pimprov = (acc - self.highest_tracc ) * 100 / self.highest_tracc # define improvement of model progres

            if acc > self.highest_tracc: # training accuracy improved in the epoch
                self.highest_tracc = acc # set new highest training accuracy
                self.best_weights = self.model.get_weights() # training accuracy improved so save the weights
                self.count = 0 # set count to 0 since training accuracy improved
                self.stop_count = 0 # set stop counter to 0
                if v_loss < self.lowest_vloss:
                    self.lowest_vloss = v_loss
                self.best_epoch = epoch + 1  # set the value of best epoch for this epoch

            else:
                # training accuracy did not improve check if this has happened for patience number of epochs
                # if so adjust learning rate
                if self.count >= self.patience - 1: # lr should be adjusted
                    lr = lr * self.factor # adjust the learning by factor
                    tf.keras.backend.set_value(self.model.optimizer.lr, lr) # set the learning rate in the optimizer
                    self.count = 0 # reset the count to 0
                    self.stop_count = self.stop_count + 1 # count the number of consecutive lr adjustments
                    self.count = 0 # reset counter
                    if v_loss < self.lowest_vloss:
                        self.lowest_vloss = v_loss
                else:
                    self.count = self.count + 1 # increment patience counter

        else: # training accuracy is above threshold so adjust learning rate based on validation loss
            monitor = 'val_loss'
            if epoch == 0:
                pimprov = 0.0
            else:
                pimprov = (self.lowest_vloss - v_loss ) * 100 / self.lowest_vloss
            if v_loss < self.lowest_vloss: # check if the validation loss improved
                self.lowest_vloss = v_loss # replace lowest validation loss with new validation loss
                self.best_weights = self.model.get_weights() # validation loss improved so save the weights
                self.count = 0 # reset count since validation loss improved
                self.stop_count = 0
                self.best_epoch = epoch + 1 # set the value of the best epoch to this epoch
            else: # validation loss did not improve
                if self.count >= self.patience - 1: # need to adjust lr
                    lr = lr * self.factor # adjust the learning rate
                    self.stop_count = self.stop_count + 1 # increment stop counter because lr was adjusted
                    self.count = 0 # reset counter
                    tf.keras.backend.set_value(self.model.optimizer.lr, lr) # set the learning rate in the optimizer
                else:
                    self.count = self.count + 1 # increment the patience counter
                if acc > self.highest_tracc:
                    self.highest_tracc = acc

        msg = f'{str(epoch + 1):^3s}/{str(self.epochs):4s} {loss:^9.3f}{acc * 100:^9.3f}{v_loss:^9.5f}{v_acc * 100:^9.3f}{current_lr:^9.5f}{lr:^9.5f}{monitor:^11s}{pimprov:^10.2f}{duration:^8.2f}'
        print(msg)

        if self.stop_count > self.stop_patience - 1: # check if learning rate has been adjusted stop_count times with no improvement
            msg = f' training has been halted at epoch {epoch + 1} after {self.stop_patience} adjustments of learning rate with no improvement'
            print(msg)
            self.model.stop_training = True # stop training
```

Figure 21 MyCallback Class Method

The “MyCallback” class depicted in Figure 21 is a custom implementation that extends the built-in Callback class provided by the Keras library. This class is designed to monitor and control the training process, providing detailed logging and adaptive learning rate adjustments. The class starts by initializing several variables through the use of an

initialization method and in this initialization method, the class stores references to the model and base model, which are used throughout the training process. The “patience” variable determines how many epochs without improvement are allowed before adjusting the learning rate, while “stop_patience” sets the number of learning rate adjustments without improvement before halting training. The “threshold” is used to decide whether to monitor accuracy or validation loss, and “factor” is the multiplier used to reduce the learning rate. The “batches”, “initial_epoch”, and “epochs” variables provide information about the training process structure. Several instance variables are initialized to track the training progress. The “count” and “stop_count” variables keep track of epochs without improvement and the number of learning rate adjustments, respectively. The “best_epoch” is initialized to 1, and will be updated as training progresses. The initial learning rate is retrieved from the model's optimizer and stored. The “highest_tracc” and “lowest_vloss” variables are initialized to track the best training accuracy and validation loss, respectively. Finally, the current model weights are stored as both the best weights and initial weights, providing a starting point for comparison and potential restoration.

Besides that, the “on_train_begin” method is called at the start of the training process by printing a formatted header for the training log, providing column names for the data that will be logged during training. The header includes information such as the epoch number, loss, accuracy, validation loss, validation accuracy, current learning rate, next learning rate, monitored metric, percentage improvement, and duration. After printing the header, the method records the start time of the training process, which will be used later to calculate the total training duration.

Next, the “on_train_end” method is called when the training process concludes and it calculates as well as prints the total duration of the training process. It subtracts the start time, stored in the “on_train_begin” variable, from the current time to get the total duration in seconds. This duration is then converted into hours, minutes, and seconds for a more readable format.

After printing the duration, the method sets the model weights to the best weights observed during training, ensuring that the model retains its best performance. In addition, the "on_train_batch_end" method is called after each training batch and provides real-time feedback on the training progress at the batch level. It retrieves the current accuracy and loss from the logs, formats them into a string, and prints this information. The accuracy is multiplied by 100 to convert it to a percentage. The printed message includes the current batch number, total number of batches, accuracy, and loss.

Additionally, the "on_epoch_begin" method is called at the start of each epoch and its purpose is to simply record the start time of the epoch. This timestamp will be used later in the method to calculate the duration of the epoch and it also contains the main logic for monitoring training progress and adjusting the learning rate. This method begins by calculating the duration of the epoch and retrieving the current learning rate. It then fetches the current accuracy, validation accuracy, loss, and validation loss from the logs. The method then decides whether to monitor accuracy or validation loss based on the "threshold" value.

If the accuracy is below the threshold, the method focuses on improving accuracy. It calculates the percentage improvement in accuracy compared to the highest accuracy seen so far. If the current accuracy is the highest seen, it updates the "highest_tracc", saves the current model weights as the best weights, resets the "count" and "stop_count", and updates the "best_epoch". If the accuracy hasn't improved, it increments the "count", whereas if the "count" reaches the "patience" value, it reduces the learning rate by multiplying it with the "factor", resets the "count", and increments the "stop_count" by 1. If the accuracy is above the threshold, the method focuses on reducing validation loss. The logic is similar to the accuracy case, but it monitors the validation loss instead. It calculates the percentage improvement in validation loss, updates the "lowest_vloss" and "best_weights" if the current validation loss is the lowest seen, and adjusts the learning rate if there's no improvement for "patience" number of epochs. After this logic, the method prints a formatted string containing all the relevant information for the epoch, including the epoch number, loss, accuracy, validation loss, validation accuracy, current learning rate, next learning rate, monitored metric, percentage improvement, and duration. Finally, if the "stop_count" exceeds the "stop_patience" which indicates that there has been no improvement despite multiple learning rate adjustments, the method prints a message and sets "model.stop_training" to 'True' and halts the training process.

```
batch_size = 40
epochs = 5
patience = 1
stop_patience = 3
threshold = 0.9
factor = 0.5
freeze = False
batches = int(np.ceil(len(train_gen.labels) / batch_size))

callbacks = [MyCallback(model=model, base_model=model, patience=patience,
             stop_patience=stop_patience, threshold=threshold, factor=factor,
             batches=batches, initial_epoch=0, epochs=epochs)]

history = model.fit(x=train_gen, epochs=epochs, verbose=1, callbacks=callbacks, validation_data=valid_gen, validation_steps=None, shuffle=False, initial_epoch=0)
```

Figure 22 Proposed Model Training Execution Code

After the “MyCallback” class is defined, the actual training process is initiated using the “fit” method of the Keras model and this entire process is shown in Figure 22. To begin the training process, the training variables are defined first with the “batch_size” being set to 40, meaning that 40 samples will be processed before the model weights are updated. The number of “epochs” is set to 5, indicating that the entire dataset will be passed through the model 5 times during training. The “patience” variable is set to 1, meaning the learning rate will be adjusted if there's no improvement after 1 epoch. The “stop_patience” is set to 3, indicating that the training will be halted if there's no improvement after 3 learning rate adjustments. The “threshold” is set to 0.9, which is used in the MyCallback class to determine whether to monitor accuracy or validation loss. The “factor” is set to 0.5, which is the factor by which the learning rate is reduced when there's no improvement. The “freeze” variable is set to ‘False’, and the “batches” variable is calculated by dividing the number of training samples by the batch size and rounding up to the nearest integer.

A list of callbacks is created, containing a single instance of the MyCallback class. This instance is initialized with the model, base model, and all the variables defined previously. Later, the “fit” method is called on the model to start the training process. It takes the training data generator as input, along with the number of epochs, verbosity level, the callbacks list, validation data generator, and other variables. The “shuffle” variable is set to ‘False’, meaning the order of samples will not be randomized between epochs, and the “initial_epoch” is set to 0 in order to indicate that the training should start from the first epoch. Finally, the training history returned by the “fit” method is stored in the “history” variable and the history object contains information about the training process which also includes the loss and accuracy values for each epoch which is useful for analysis and visualization of the training progress.

3.6 Evaluation Metrics

After the proposed model training is completed, various evaluation metrics are calculated to assess the proposed model's performance in order to provide comprehension into the state of the proposed model when testing on the test dataset. The evaluation process includes calculating accuracy and loss for the training, validation, and test sets, as well as generating a confusion matrix and a classification report.

```
ts_length = len(test_df)
test_batch_size = test_batch_size = max(sorted([ts_length // n for n in range(1, ts_length + 1) if ts_length%n == 0 and ts_length/n <= 80]))
test_steps = ts_length // test_batch_size
train_score = model.evaluate(train_gen, steps= test_steps, verbose= 1)
valid_score = model.evaluate(valid_gen, steps= test_steps, verbose= 1)
test_score = model.evaluate(test_gen, steps= test_steps, verbose= 1)

print("Train Loss: ", train_score[0])
print("Train Accuracy: ", train_score[1])
print('-' * 20)
print("Validation Loss: ", valid_score[0])
print("Validation Accuracy: ", valid_score[1])
print('-' * 20)
print("Test Loss: ", test_score[0])
print("Test Accuracy: ", test_score[1])

preds = model.predict_generator(test_gen)
y_pred = np.argmax(preds, axis=1)
print(y_pred)

target_names = ['Bacterial_spot', 'Early_blight', 'Late_blight', 'Leaf_Mold', 'Septoria_leaf_spot', 'Spider_mites', 'Target_Spot', 'Tomato_Yellow_Leaf_Curl_Virus', 'Tomato_mosaic_virus', 'healthy']
# Confusion matrix
cm = confusion_matrix(test_gen.classes, y_pred)
plot_confusion_matrix(cm= cm, classes= target_names, title = 'Confusion Matrix')
# Classification report
print(classification_report(test_gen.classes, y_pred, target_names= target_names))
```

Figure 23 Proposed Model Evaluation Code

Firstly, the evaluation of the proposed model's performance on the training, validation, and test sets is conducted. In Figure 23, the length of the test dataset is first determined. The test batch size is then calculated as the largest factor of the test dataset length that results in a batch size less than or equal to 80. This ensures efficient use of memory during evaluation. The number of test steps is calculated by dividing the test dataset length by the batch size. The proposed model's “evaluate” method is then called on the training, validation, and test generators. This method returns a list containing the loss and accuracy for each dataset. These scores are then printed, providing a clear comparison of the proposed model's performance across the three datasets. Next, predictions are generated for the test set, and the predicted classes are extracted. The “predict_generator” method is used to generate predictions for the test set. These predictions are probability distributions over the classes. The “np.argmax” function is then used to convert these probabilities into class predictions by selecting the class with the highest probability for each sample. The resulting predictions are printed. Following the prediction generation, a confusion matrix is created and visualized using the “plot_confusion_matrix” function shown in Figure 24. Finally, a classification report is generated with the “classification_report” function from scikit-learn showing the main classification metrics such as precision, recall, f1-score, and support for each class.

```python
def plot_confusion_matrix(cm, classes, normalize= False, title= 'Confusion Matrix', cmap=plt.cm.viridis):
    sns.set_style('white')
    plt.figure(figsize= (10, 10))
    plt.imshow(cm, interpolation= 'nearest', cmap= cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation= 'vertical')
    plt.yticks(tick_marks, classes)
    if normalize:
        cm = cm.astype('float') / cm.sum(axis= 1)[:, np.newaxis]
        print('Normalized Confusion Matrix')
    else:
        print('Confusion Matrix, Without Normalization')
    print(cm)
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j], horizontalalignment= 'center', color= 'purple' if cm[i, j] > thresh else 'yellow')
    plt.tight_layout()
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
```

Figure 24 plot_confusion_matrix Function

The “confusion_matrix” function from scikit-learn is used to create the confusion matrix, comparing the true labels with the predicted labels. The resulting matrix is then visualized using a custom “plot_confusion_matrix” function illustrated in Figure 24. This function creates a visual representation of the confusion matrix. It uses matplotlib to create a heatmap of the matrix, with each cell color based on its value. The function also adds text labels to each cell, showing the number of samples in that category. The color of the text is adjusted for readability based on the cell's value.

```python
def plot_training(hist):
    sns.set_style('darkgrid')
    tr_acc = hist.history['accuracy']
    tr_loss = hist.history['loss']
    val_acc = hist.history['val_accuracy']
    val_loss = hist.history['val_loss']
    index_loss = np.argmin(val_loss)
    val_lowest = val_loss[index_loss]
    index_acc = np.argmax(val_acc)
    acc_highest = val_acc[index_acc]

    plt.figure(figsize= (20, 8))
    plt.style.use('fivethirtyeight')
    Epochs = [i+1 for i in range(len(tr_acc))]
    loss_label = f'best epoch= {str(index_loss + 1)}'
    acc_label = f'best epoch= {str(index_acc + 1)}'
    plt.subplot(1, 2, 1)
    plt.plot(Epochs, tr_loss, 'r', label= 'Training loss')
    plt.plot(Epochs, val_loss, 'g', label= 'Validation loss')
    plt.scatter(index_loss + 1, val_lowest, s= 150, c= 'blue', label= loss_label)
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(Epochs, tr_acc, 'r', label= 'Training Accuracy')
    plt.plot(Epochs, val_acc, 'g', label= 'Validation Accuracy')
    plt.scatter(index_acc + 1 , acc_highest, s= 150, c= 'blue', label= acc_label)
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.tight_layout
    plt.show()
```

Figure 25 plot_training Function

In addition to these evaluation metrics, the training history is visualized using a custom “plot_training” function and this function creates two subplots in which one for loss and one for accuracy. Each subplot shows the training and validation metrics over the course of the training epochs. The best epoch, with lowest validation loss and highest validation accuracy, is highlighted with a blue dot. This visualization allows for easy interpretation of the proposed model's learning progress and potential overfitting.

3.7 Web Application Development

The web application for this project was developed as a means to demonstrate the capabilities of the proposed tomato plant disease detection model. It is important to note that while functional, the application is not intended to be a comprehensive or production-ready solution, but rather a proof-of-concept to showcase the model's performance. The web application was built using the Flask framework, a lightweight and flexible Python web framework. Flask was chosen for its simplicity and ease of integration with deep learning models. The application's structure follows a typical Flask project layout, with separate files for routing, templates, and static assets. The main structure of the web application is defined in the “base.html” template, which serves as the foundation for all pages. This template includes the necessary HTML structure, Bootstrap CSS for styling, and JavaScript libraries for enhanced functionality. The “base.html” file also defines a navigation bar that allows users to easily move between different sections of the application. This navigation bar provides links to the home page, the disease detection page, and about page, allowing users to easily navigate through the application. The display of the home page, disease detection page, and about page are shown in Figure 26, Figure 27, and Figure 28 respectively.

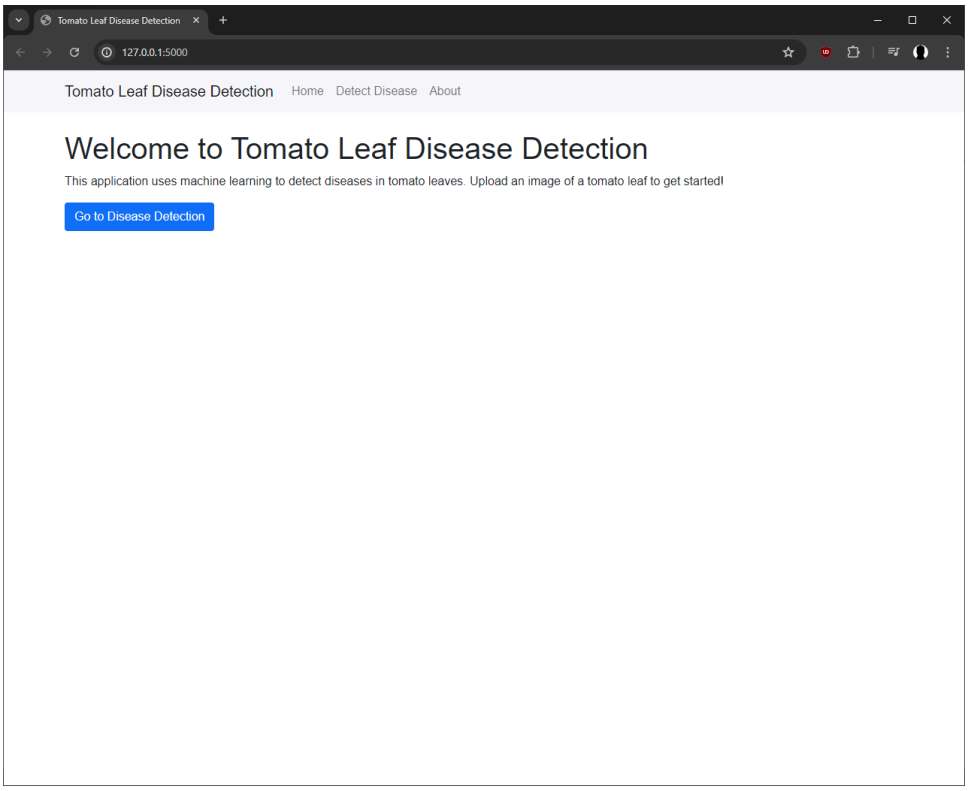


Figure 26 Home Screen Display

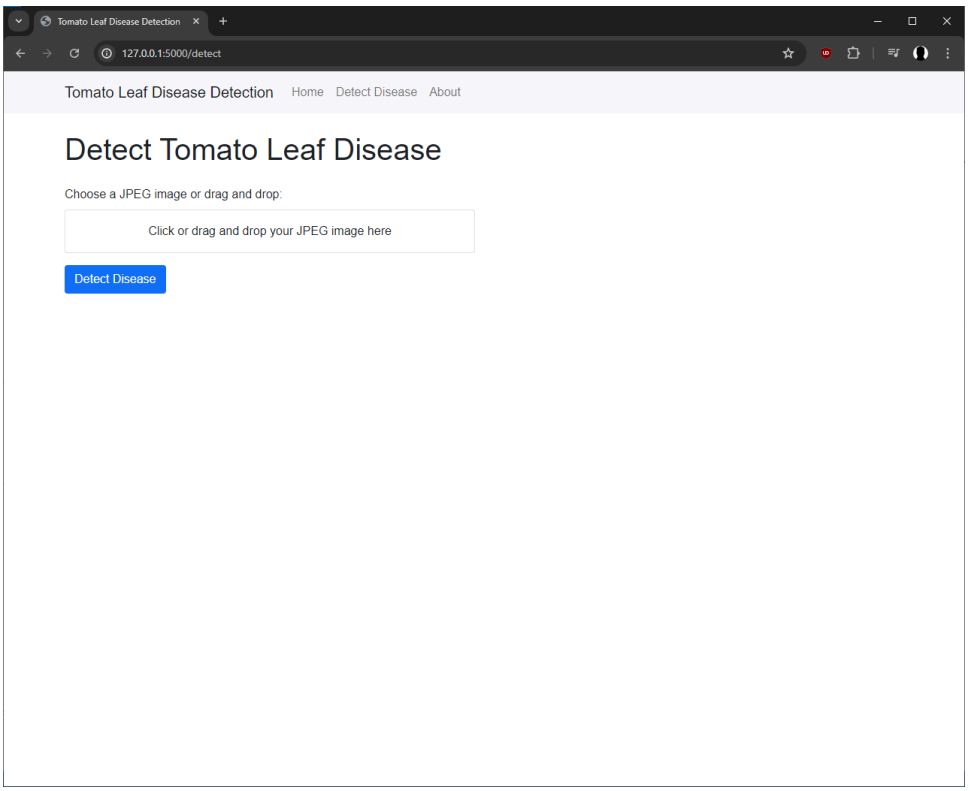


Figure 27 Detect Screen Display

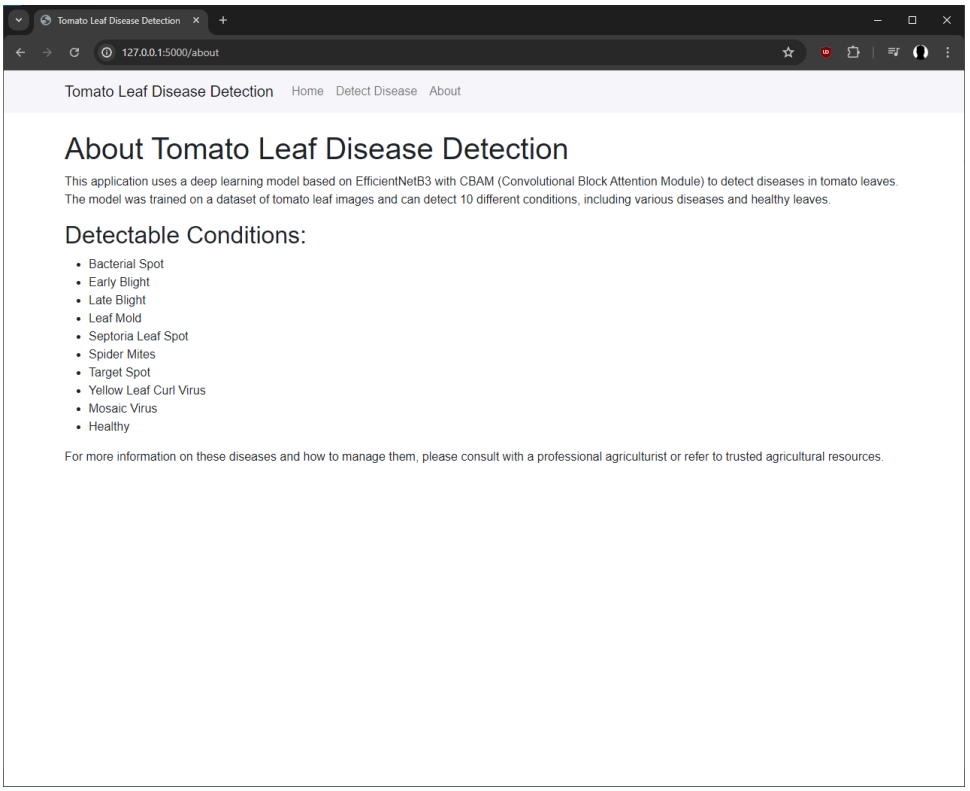


Figure 28 About Screen Display

The core functionality of the web application is implemented in the “detect.html” template and the associated JavaScript file named “main.js”. The “detect.html” template defines the structure of the disease detection page, including the file upload form and the results display area. This form allows users to upload an image file either by clicking to select a file or by dragging and dropping the file onto the designated area. The application supports drag-and-drop functionality through JavaScript event listeners defined in the “main.js” file. These event listeners prevent the default browser behavior for drag events and handle the file drop, allowing for a smooth user experience. Once an image is uploaded, the application provides an image preview to the user as demonstrated in Figure 29. If the uploaded file is a JPEG image, the web application will display a preview of the image to the user, else if the file is not a JPEG, an error message is shown which is depicted in Figure 30.

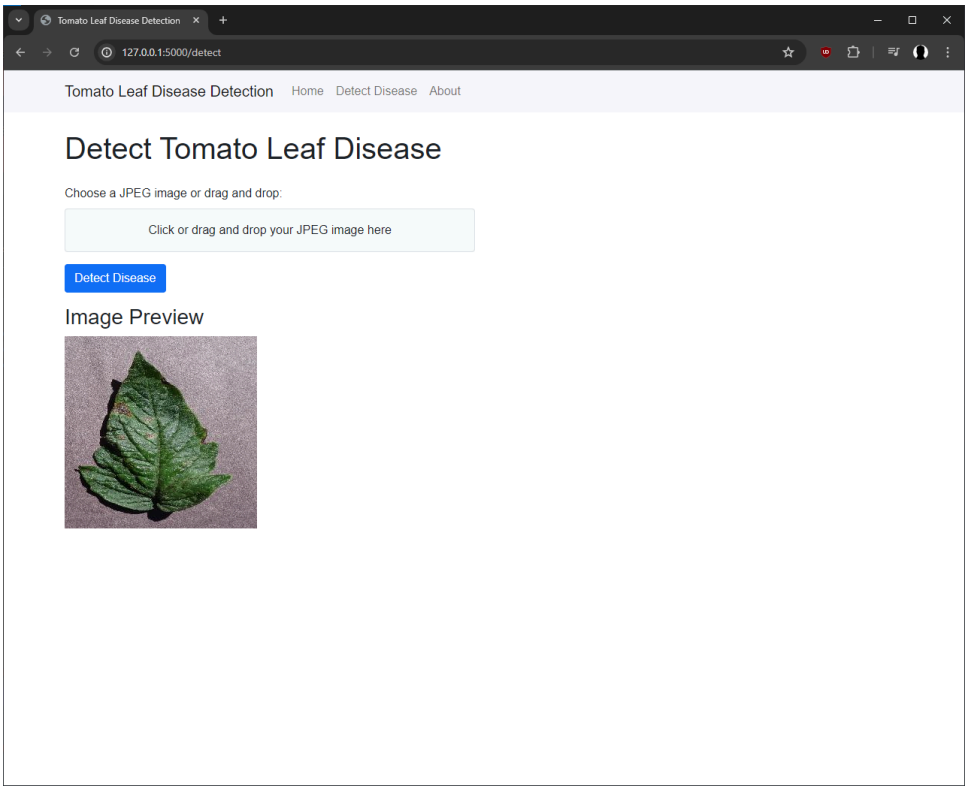


Figure 29 Upload Image Preview Display

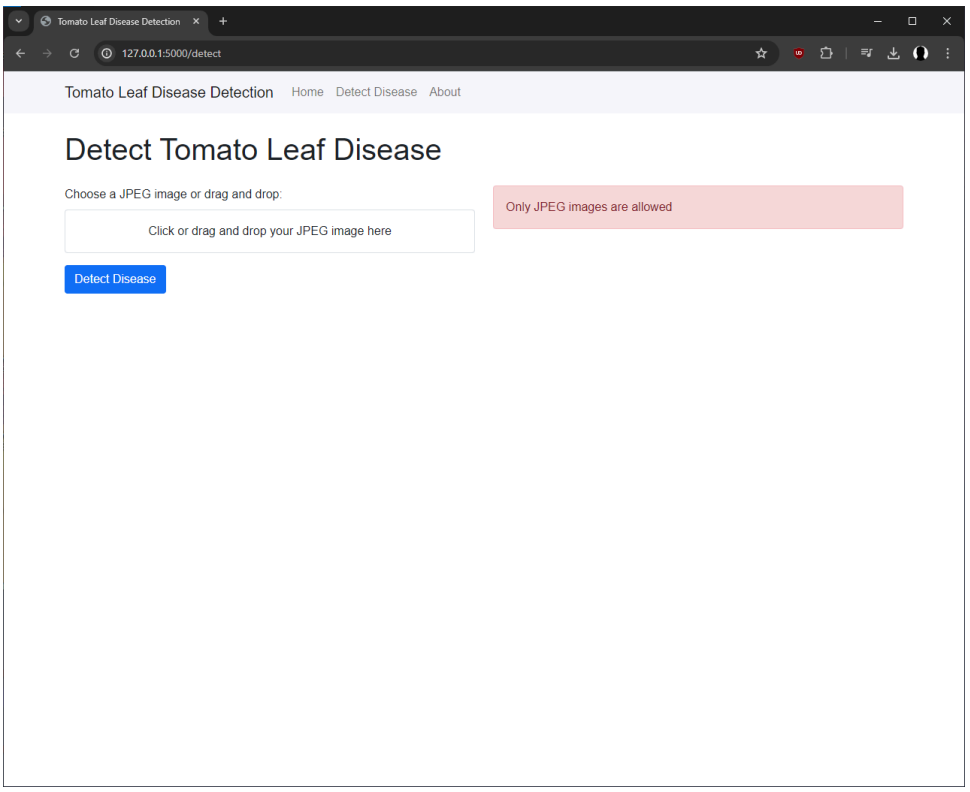


Figure 30 File Upload Error Display

When the user submits the form to detect diseases, the application sends the image to the server for processing. During this time, a loading indicator is displayed to provide feedback to the user and the indicator is shown in Figure 31. This loading indicator helps improve the user experience by providing visual feedback during the potentially time-consuming disease detection process.

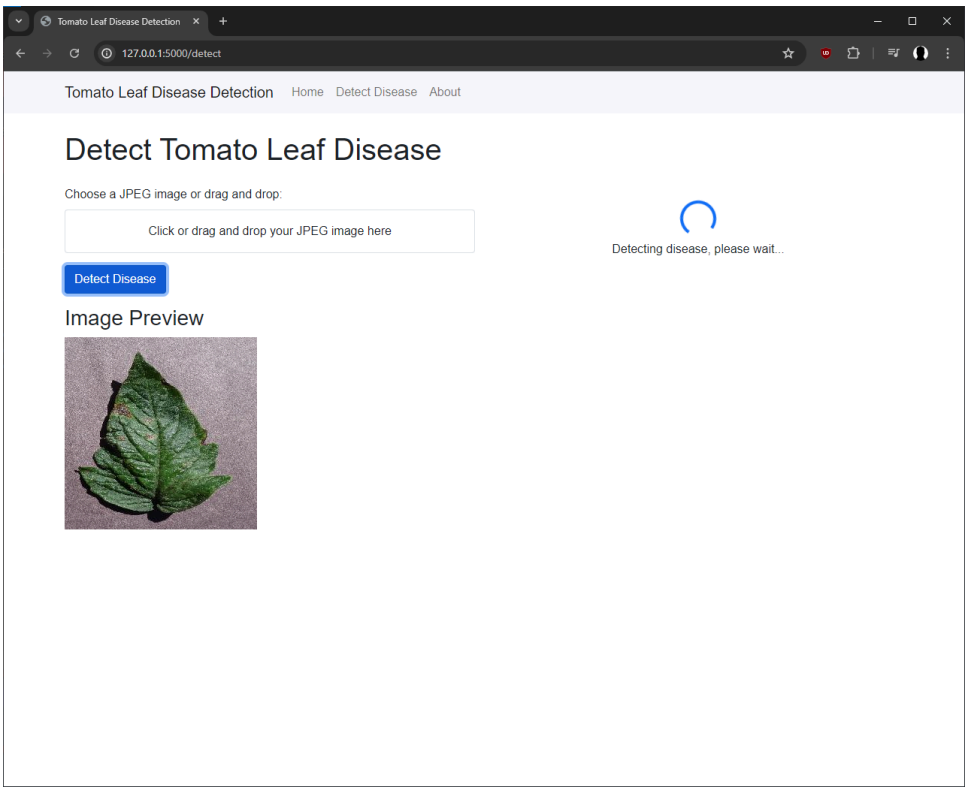


Figure 31 Disease Detection Loading Display

Once the server processes the image and returns the results, the application displays these results as shown in Figure 32. This includes showing the detected disease, the confidence level, a Grad-CAM heatmap highlighting the diseased area, and a pie chart showing the probability distribution across all possible diseases. The web application creates a pie chart using 'Chart.js' to visualize the probability distribution with the visual representation of the model's confidence across all the possible classes. Additionally, the application provides users with more detailed information about the detected disease, including symptoms, causes, and preventive measures in order to aid users understand the detected disease and provides guidance on how to manage it, and this is seen in Figure 33.

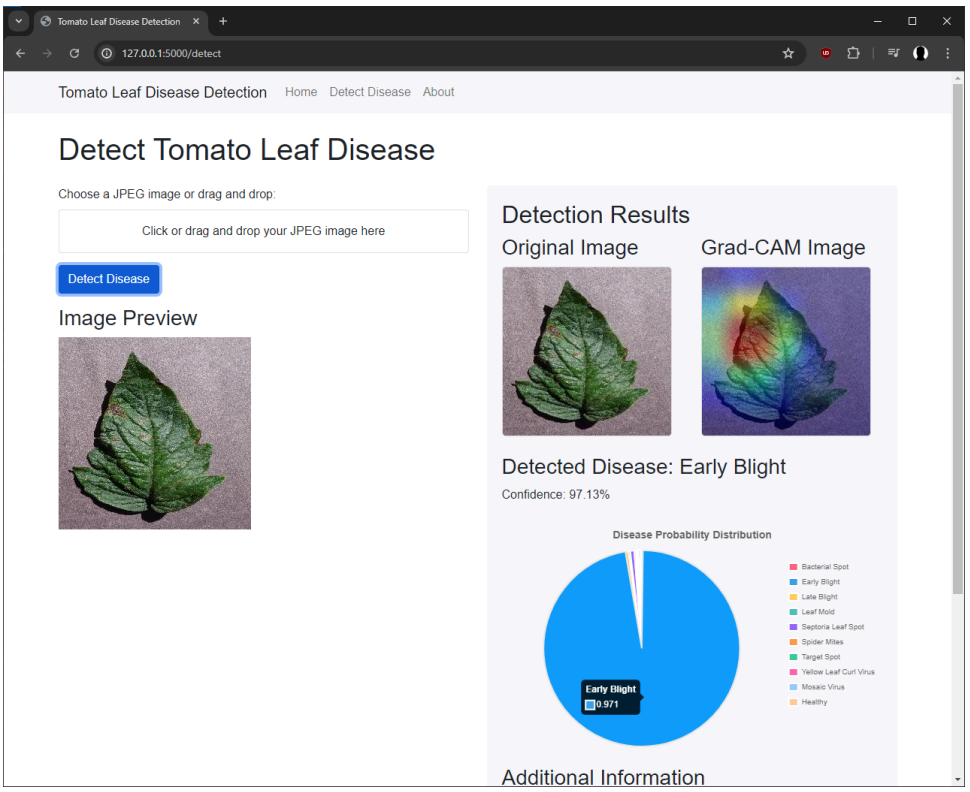


Figure 32 Disease Detection Results 1 Display

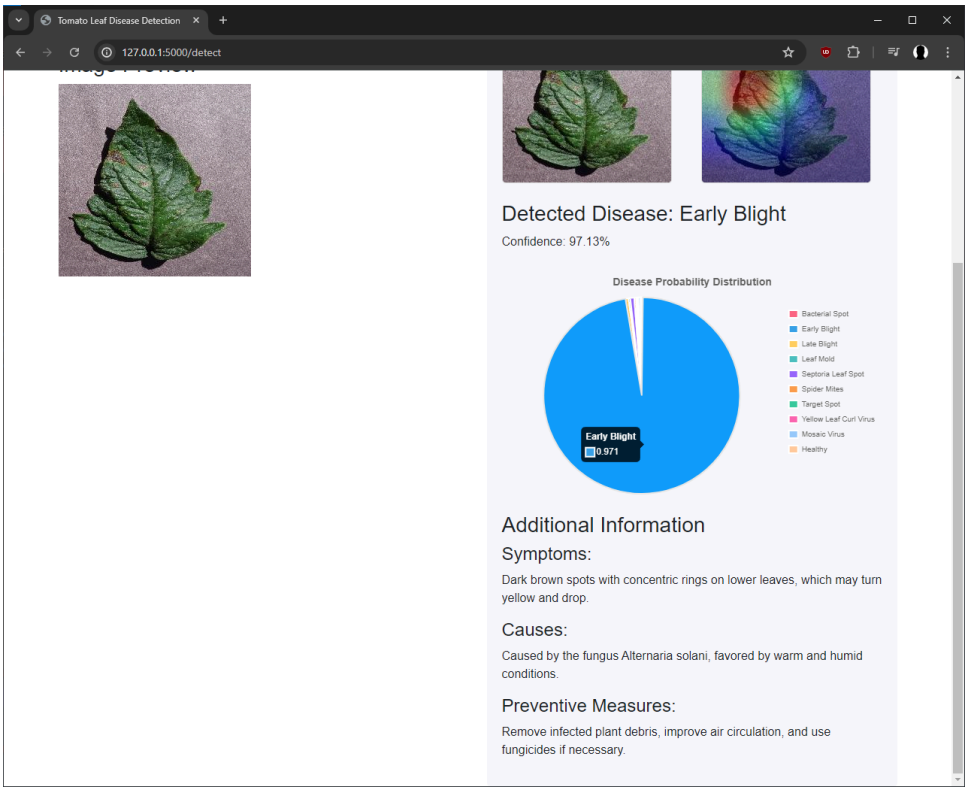


Figure 33 Disease Detection Results 2 Display

In addition to handling file type errors, the web application also implements error handling for cases where the uploaded image does not contain a tomato plant or when the disease is out of scope for the detection model. Based on Figure 34, after the image is processed by the model, there is a confidence threshold check where it will ensure that the model only returns a result when it's reasonably confident in its prediction. If the confidence is below the threshold, it suggests that the image might not contain a tomato plant, or the disease might be out of scope for the model.

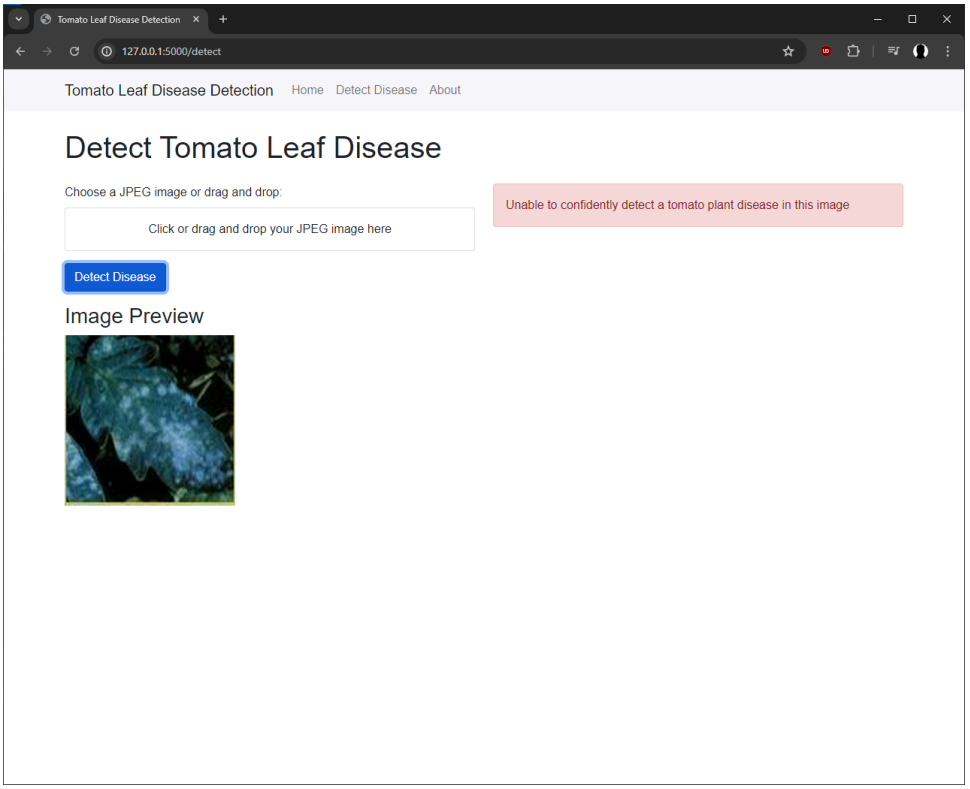


Figure 34 Disease Detection Error Display

3.8 Chapter Summary

This chapter presented a detailed methodology for developing an enhanced deep learning model for tomato plant disease detection and its integration into a web application. The methodology encompassed several key stages, including environment setup, dataset acquisition, data pre-processing, model development, training, evaluation, and web application development. The project utilized Python as the primary programming language, employing TensorFlow and Keras for deep learning tasks and Flask for web application development. The experimental setup, including hardware specifications and software versions, was clearly delineated to ensure reproducibility. Hyperparameters for the model were carefully selected and documented, including the number of classes, input dimensions, batch size, learning rate, and data augmentation techniques. For dataset acquisition, the project leveraged the PlantVillage dataset, specifically isolating tomato plant images across ten classes, including nine disease classes and one healthy class. The dataset was also split into training, validation, and testing sets in order to properly evaluate the proposed model. Additionally, various data pre-processing techniques were used to prepare the images for model training and the techniques used included creating proper file paths, generating DataFrames, implementing data augmentation, resizing images, and utilizing data generators. Besides that, the main focus of this project methodology focused on the development of a Convolutional Neural Network (CNN) based on the EfficientNet-B3 architecture. The model was later enhanced with transfer learning by using the weight from the popular ImageNet dataset, and integrated with the Convolutional Block Attention Module (CBAM) to improve feature focus. Moreover, the training process highlighted the use of a custom callback class for monitoring training progress, adjusting learning rates, and providing detailed logging. Furthermore, the evaluation metrics for assessing the model's performance included accuracy and loss calculations for training, validation, and test sets, as well as the generation of confusion matrices and classification reports. Finally, the chapter described the development of a Flask-based web application to demonstrate the model's capabilities.

4.0 RESULTS & DISCUSSION

This section presents the results obtained from training and evaluating the proposed EfficientNet-B3 model with CBAM for tomato plant disease detection. The performance of the model is analyzed in detail, including training metrics, evaluation results, comparisons with other architectures, and ablation experiments. Additionally, this section discusses the implications of these results and their significance for the project's objectives and overall effectiveness in similar natured projects.

4.1 Proposed Model Results

Epoch	Loss	Accuracy	V_loss	V_acc	LR	Next LR	Monitor	% Improv	Duration
1 /5	5.734	80.230	3.60356	95.117	0.00100	0.00100	accuracy	0.00	6730.64
2 /5	2.653	95.770	1.77797	98.100	0.00100	0.00100	val_loss	50.66	278.51
3 /5	1.352	97.370	0.91629	98.300	0.00100	0.00100	val_loss	48.46	286.38
4 /5	0.741	97.950	0.50608	99.233	0.00100	0.00100	val_loss	44.77	277.74
5 /5	0.474	98.150	0.33446	99.600	0.00100	0.00100	val_loss	33.91	286.28

training elapsed time was 2.0 hours, 11.0 minutes, 49.34 seconds)

Figure 35 Proposed Model Training Tabular Results

Based on the result shown in Figure 35, the training process of the proposed EfficientNet-B3 model with CBAM was conducted over 5 epochs, with the results displayed in the provided image. Throughout this process, the model demonstrated rapid and consistent improvement across all metrics. Initially, the training accuracy started at 80.23% in the first epoch. As the training progressed, this metric steadily increased, ultimately reaching 98.15% in the final epoch. This significant improvement illustrates the model's growing ability to effectively differentiate between various tomato plant diseases. Simultaneously, the validation accuracy showed a parallel improvement, rising from 95.12% to an impressive 99.60%. This concurrent enhancement in validation accuracy strongly suggests that the model is performing well on unseen data, rather than simply memorizing the training set. Furthermore, both training and validation loss exhibited substantial decreases. Specifically, the training loss dropped from 5.734 to 0.474, while the validation loss reduced from 3.60356 to 0.33446. The consistent reduction in both these metrics further supports the notion that the model was learning effectively without overfitting to the training data. Interestingly, the learning rate remained constant at 0.00100 throughout the entire training process. This stability indicates that the initial hyperparameters were well-chosen and did not require adjustment by the adaptive callback mechanism implemented in the code. Another noteworthy observation is the significant decrease in training duration per epoch. The first epoch took 6730.64 seconds,

whereas the final epoch only required 286.28 seconds. This substantial reduction in training time can be attributed to GPU caching and optimization processes. Thus, these results clearly demonstrate the effectiveness of the proposed model architecture in classifying tomato plant diseases. The rapid convergence to high accuracy within just 5 epochs suggests that the combination of EfficientNet-B3 and CBAM is particularly well-suited for this task. Moreover, the consistent improvement in both training and validation metrics indicates good generalization, however there may be some potential limitations due to the high accuracy results that could indicate cases of overfitting.

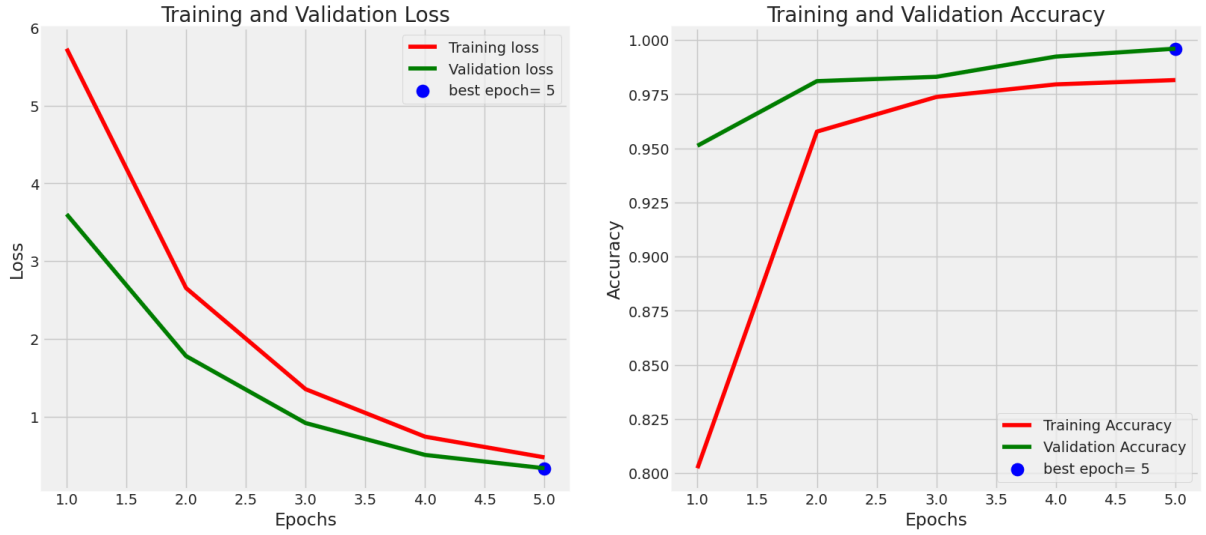


Figure 36 Proposed Model Training Plot Results

Building upon the previous analysis, the graphical representation in Figure 36 provides a visual interpretation of the model's performance over the 5 epochs of training. This plot directly corresponds to the numerical data presented in the earlier tabular format, offering a clearer visualization of the training dynamics. The left panel of Figure 36 displays the Training and Validation Loss curves. Both curves show a sharp initial decrease, particularly between epochs 1 and 2, followed by a more gradual decline in subsequent epochs. The training loss, indicated by the red line, starts higher than the validation loss, indicated by the green line, but converges to a similar value by the end of training. This convergence suggests that the model is generalizing well, with minimal overfitting. The right panel illustrates the Training and Validation Accuracy curves. Both accuracy measures show a steep increase in the early epochs, with the validation accuracy consistently outperforming the training accuracy. The blue dots on both panels indicate the best epoch in terms of validation

performance. For both loss and accuracy, the best epoch appears to be the final 5th epoch which suggests that the model continued to improve throughout the entire training process.

```
50/50 [==============================] - 37s 733ms/step - loss: 0.3307 - accuracy: 0.9975
50/50 [==============================] - 11s 218ms/step - loss: 0.3382 - accuracy: 0.9955
50/50 [==============================] - 27s 467ms/step - loss: 0.3357 - accuracy: 0.9955
Train Loss:  0.33069297671318054
Train Accuracy:  0.9975000023841858
--------------------
Validation Loss:  0.33824509382247925
Validation Accuracy:  0.9955000281333923
--------------------
Test Loss:  0.33566296100616455
Test Accuracy:  0.9955000281333923
```

Figure 37 Proposed Model Evaluation Results

Figure 37 displays the final evaluation results of the proposed model, encompassing metrics for the training, validation, and test sets. This comprehensive information offers a detailed view of the performance of the proposed model across different data subsets, allowing for a thorough analysis of its effectiveness in classifying tomato plant diseases. The training set performance is notably strong, with a train loss of 0.33069 and a train accuracy of 0.9975. These figures indicate that the model has learned to classify the training data with exceptional precision. The low training loss and high accuracy, reaching 99.75%, suggest that the model has effectively captured the patterns and features necessary for distinguishing between different tomato plant diseases within the training set. Moving on to the validation set performance, the metrics are closely aligned with the training results. The validation loss is 0.3382, and the validation accuracy is 0.9955. These values are very close to the training metrics, with only a slightly higher loss and marginally lower accuracy of 99.55%. This close alignment between training and validation performance is a positive indicator due to the reason that the model performs well to unseen data and is not overfitting to the training set. Besides that, the test set performance has a test loss of 0.3357 and a test accuracy of 0.9955. Interestingly, the test set accuracy is identical to the validation set at 99.55%, and the loss is very close as well. This consistency across validation and test sets further reinforces the model's robustness and its ability to generalize effectively to new, unseen data.

	precision	recall	f1-score	support
Bacterial_spot	0.99	0.99	0.99	400
Early_blight	0.99	0.99	0.99	400
Late_blight	0.99	0.99	0.99	400
Leaf_Mold	1.00	1.00	1.00	400
Septoria_leaf_spot	1.00	1.00	1.00	400
Spider_mites	1.00	0.99	0.99	400
Target_Spot	0.99	1.00	0.99	400
Tomato_Yellow_Leaf_Curl_Virus	1.00	0.99	1.00	400
Tomato_mosaic_virus	1.00	1.00	1.00	400
healthy	1.00	1.00	1.00	400
accuracy			1.00	4000
macro avg	1.00	1.00	1.00	4000
weighted avg	1.00	1.00	1.00	4000

Figure 38 Proposed Model Classification Results

Figure 38 presents a detailed classification result for the proposed model's performance on the tomato plant disease detection task. This table provides precision, recall, F1-score, and support for each disease class, as well as overall accuracy and weighted averages. The result lists ten distinct classes, which correspond to different tomato plant diseases and a healthy class. For most classes, the model achieves perfect scores across all metrics, with precision, recall, and F1-score all at 1.00. This indicates that for these classes, the model correctly identified all instances without any false positives or false negatives, and the support column shows that each class has 400 samples in the test set. The overall accuracy of the model is calculated as 1.00 and the macro average, which gives equal weight to each class regardless of its support, shows perfect scores of 1.00 for precision, recall, and F1-score. Lastly, the weighted average, which takes into account the support of each class, also shows perfect scores across all metrics.

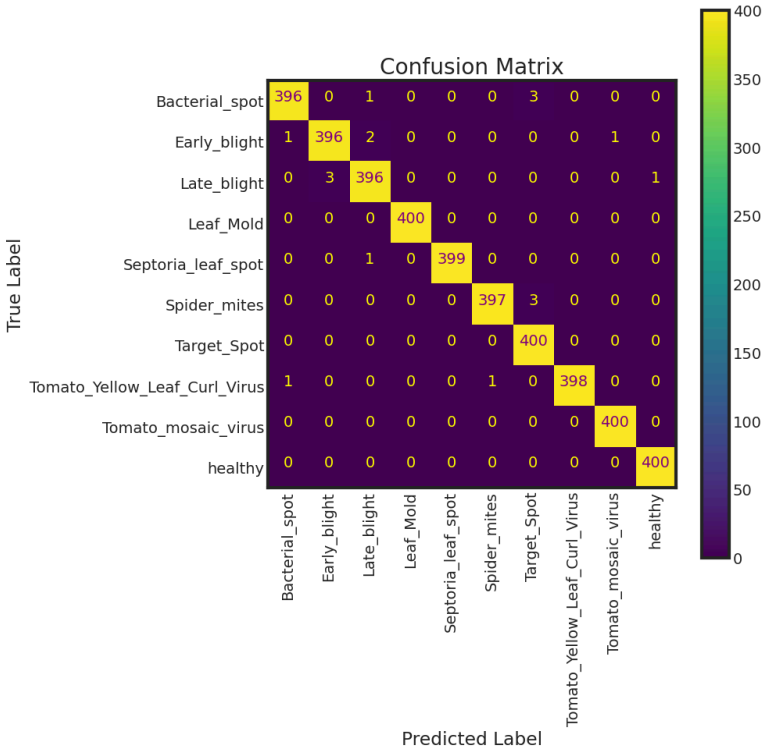


Figure 39 Proposed Model Confusion Matrix Results

Figure 39 displays a 10 by 10 confusion matrix illustrating the performance of the proposed model across ten classes of tomato plant conditions. This tabular representation illustrates the model's predictive performance, detailing both accurate classifications and misclassifications across all categories with the diagonal elements representing the number of correct classifications for each class while elements not within the diagonal line indicate misclassifications. The confusion matrix reveals exceptional performance across all classes, with the vast majority of samples being correctly classified with the majority being close to or are equal to 400 and this indicates that the images are being correctly predicted with no major misclassifications.

4.2 Neural Network Comparison Results

This section presents a comparative analysis of the proposed model against other prominent CNN architectures for tomato plant disease detection. This comparison encompasses five models: VGG16, DenseNet121, ResNet50, EfficientNet-B3, and the proposed model. Their performance is evaluated using standard metrics including average precision, average recall, average F1-score, and overall accuracy. Additionally, the section examines the training dynamics of each model through accuracy and loss plots, as well as their classification performance via confusion matrices.

Model	Avg. Precision	Avg. Recall	Avg. F1-Score	Accuracy
VGG16	79.70%	73.70%	72.80%	73.55%
DenseNet121	98.80%	98.70%	98.70%	98.65%
ResNet50	99.60%	99.40%	99.60%	99.53%
EfficientNet-B3	99.20%	99.20%	99.30%	99.26%
Proposed Model	**99.60%**	**99.50%**	**99.50%**	**99.55%**

Table 10 Model Evaluation Results

Table 10 showcases the evaluation results for five different CNN models which are VGG16, DenseNet121, ResNet50, EfficientNet-B3, and the proposed model. The performance evaluation incorporates several key indicators such as the mean precision, mean recall, average F1-score, and global accuracy. Among these models, the proposed model demonstrates the highest performance across all metrics. Specifically, it achieves an average precision of 99.60%, average recall of 99.50%, average F1-score of 99.50%, and an impressive accuracy of 99.55%. These results represent a slight improvement over the next best performing model, ResNet50, which shows comparable performance with 99.60% average precision, 99.40% average recall, 99.60% average F1-score, and 99.53% accuracy. Furthermore, the EfficientNet-B3 and DenseNet121 architectures also exhibit strong performance, with accuracies of 99.26% and 98.65% respectively. However, the VGG16 model lags significantly behind the other architectures, achieving only 73.55% accuracy.

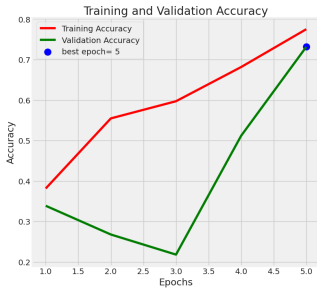
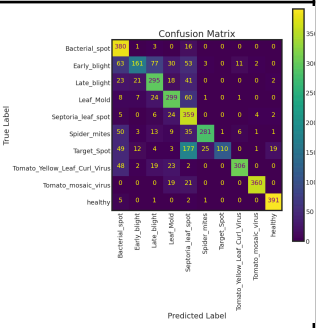
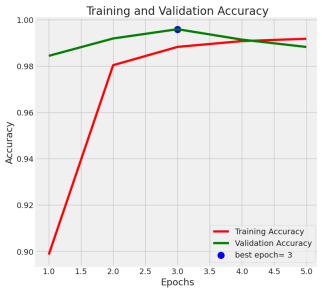
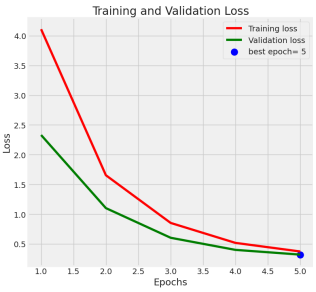
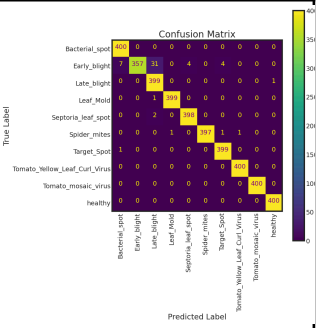
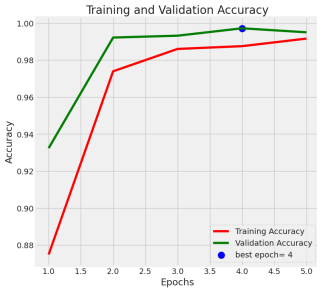
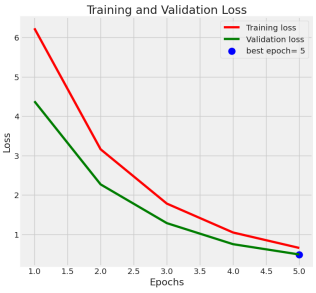
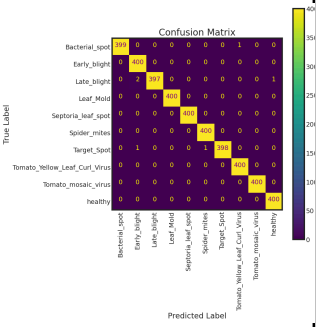
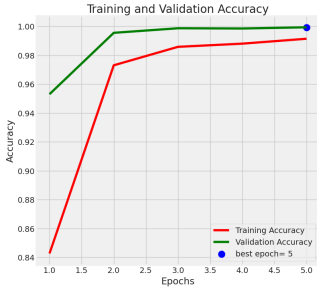
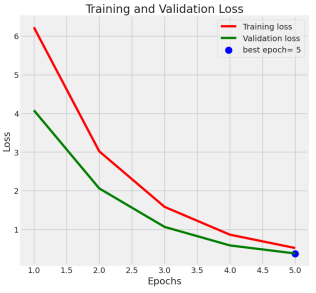
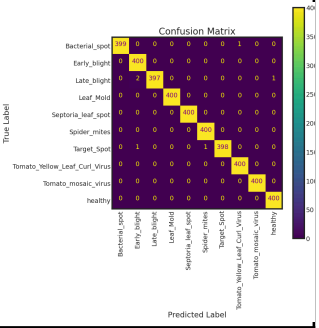
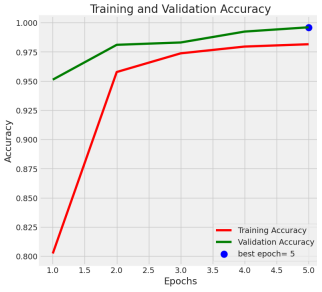
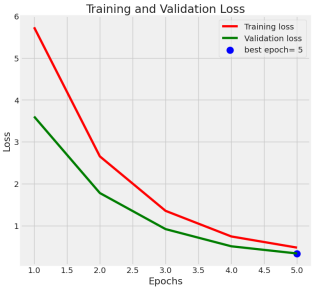
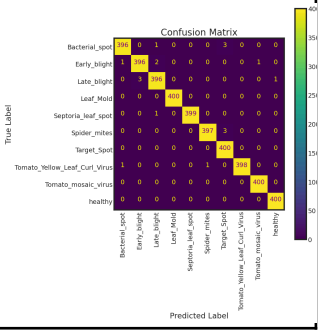
Model	Accuracy	Loss	Confusion Matrix
VGG16			
DenseNet121			
ResNet50			
EfficientNet-B3			
Proposed Model			

Table 11 Model Performance Results

Table 11 depicts the visualization of the training dynamics and classification performance of each model through accuracy plots, loss plots, and confusion matrices. The accuracy plots reveal that all models, with the exception of VGG16, achieve rapid convergence to high accuracy within the initial few epochs. Particularly, the proposed model and ResNet50 display remarkably stable and high accuracy curves for both training and validation sets, indicating robust learning and good generalization. Complementing the accuracy plots, the loss curves further corroborate these findings. The proposed model, ResNet50, EfficientNet-B3, and DenseNet121 all exhibit a rapid decrease in loss and maintain close alignment between training and validation losses. This alignment suggests that these models are learning effectively without overfitting to the training data. The confusion matrices presented in Table 11 visually confirm the high classification performance of the top-performing models. The proposed model, ResNet50, EfficientNet-B3, and DenseNet121 all display strong diagonal patterns, indicating a high number of correct classifications across all classes while, VGG16's confusion matrix shows more off-diagonal elements, meaning that the model has lower accuracy and higher misclassification rate.

These comparative results have several implications for the project. Firstly, they demonstrate that the proposed model achieves as well as exceeds the expected performance for tomato plant disease detection and manages to outperform well-established baseline architectures. This suggests that the modifications made to the base EfficientNet-B3 architecture with the addition of the CBAM attention mechanism, provide meaningful improvement in classification performance. However, the minimal difference in performance between the top models also indicates that the project may be approaching the limits of performance improvement possible with the current dataset and problem formulation. This observation suggests that further significant gains might require more sophisticated techniques or larger diverse datasets. Additionally, the extremely high accuracy achieved by multiple models does create doubts about the complexity and representativeness of the test set. It may be beneficial to evaluate these models on a more challenging or real-world dataset to ensure their performance generalizes well to diverse scenarios. Thus, the proposed model achieves the highest accuracy when compared to other well-known models but the differences with ResNet50 and EfficientNet-B3 are minimal which suggests that further work must be done to improve the proposed model.

4.3 Ablation Experiment Results

This section presents the results of ablation experiments conducted to analyze the impact of different components in the proposed model architecture for tomato plant disease detection. Ablation studies systematically remove or modify specific elements of the model to understand their individual contributions to overall performance. The experiments compare four variations of the EfficientNet-B3 architecture which are, the base model, the base model with Convolutional Block Attention Module, the base model with transfer learning, and the full proposed model combining transfer learning and CBAM.

Model	Avg. Precision	Avg. Recall	Avg. F1-Score	Accuracy
EfficientNet-B3	69.80%	66.10%	65.00%	66.22%
EfficientNet-B3-CBAM	55.90%	43.10%	42.10%	43.20%
EfficientNet-B3-TL	99.20%	99.20%	99.30%	99.26%
EfficientNet-B3-TL-CBAM	**99.60%**	**99.50%**	**99.50%**	**99.55%**

Table 12 Ablation Experiment Evaluation Results

Table 12 displays the evaluation metrics for each model variation. The base EfficientNet-B3 model achieves moderate performance with 66.22% accuracy, 69.80% average precision, 66.10% average recall, and 65.00% average F1-score. Interestingly, the addition of CBAM alone leads to a decrease in performance across all metrics, with accuracy dropping to 43.20%. This unexpected result suggests that the attention mechanism may not be effective without proper initialization or training. In contrast, the introduction of transfer learning dramatically improves performance, achieving 99.26% accuracy and similarly high scores for other metrics. This substantial improvement highlights the critical role of transfer learning in leveraging pre-trained knowledge for the specific task of tomato plant disease detection. The full proposed model combines transfer learning with CBAM and achieves the highest performance across all metrics, with 99.55% accuracy, 99.60% average precision, and 99.50% for both the average recall as well as F1-score. This result proves that the attention mechanism, when combined with transfer learning, provides a meaningful improvement over transfer learning alone.

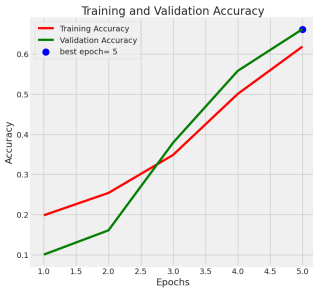
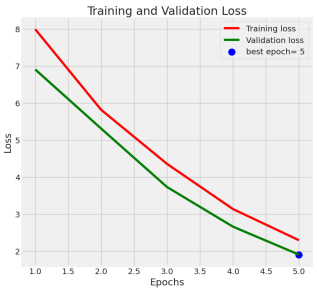
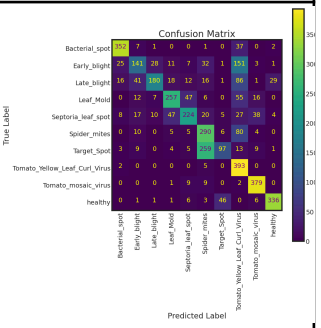
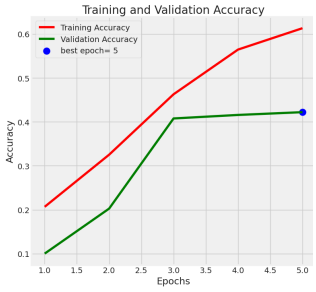
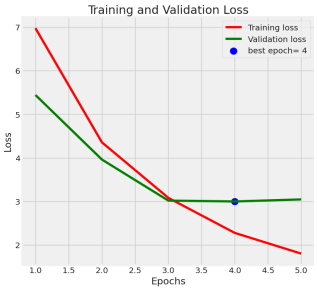
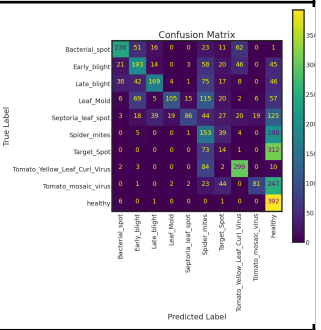
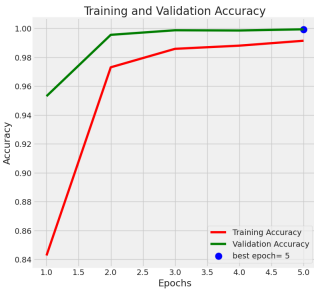
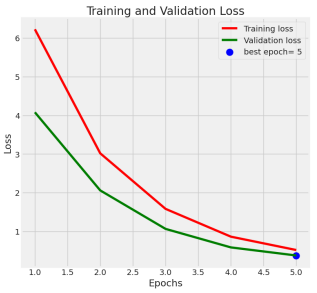
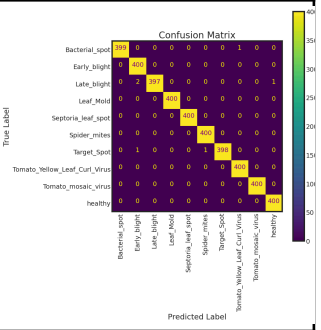
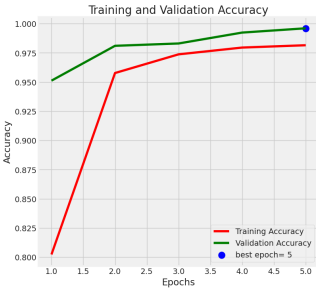
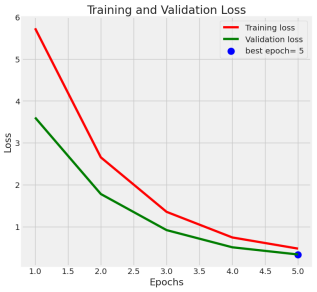
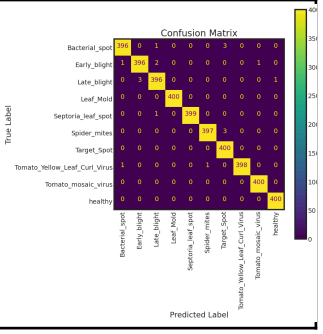
Model	Accuracy	Loss	Confusion Matrix
EfficientNet-B3			
EfficientNet-B3 -CBAM			
EfficientNet-B3 -TL			
EfficientNet-B3 -TL-CBAM			

Table 13 Ablation Experiment Performance Results

In Table 13, the performance plots and confusion matrices, provides further insights into the behavior of each model variation. The accuracy and loss plots for the base EfficientNet-B3 model show gradual improvement over epochs, but with a significant gap between training and validation performance, indicating potential overfitting. The EfficientNet-B3-CBAM model's plots reveal unstable training dynamics, with fluctuating accuracy and loss curves that fail to converge effectively. In contrast, both EfficientNet-B3-TL and

EfficientNet-B3-TL-CBAM demonstrate rapid convergence and stable training, with closely aligned training and validation curves. This behavior suggests effective learning and good generalization. The confusion matrices further corroborate these findings, with the transfer learning-based models showing strong diagonal patterns indicative of high classification accuracy across all classes.

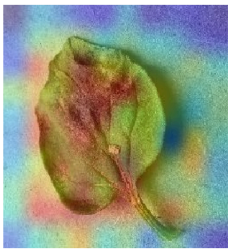
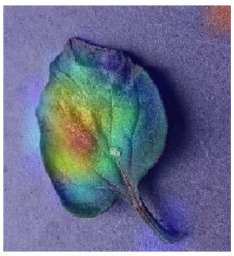
Original	EfficientNet-B3	EfficientNet-B3-CBAM	EfficientNet-B3-TL	EfficientNet-B3-TL-CBAM
				

Table 14 Ablation Experiment Grad-CAM Results

The Grad-CAM visualizations in Table 14 provide valuable insights into how the different model variations focus on relevant features for tomato plant disease detection. The original image shows a tomato leaf affected by late blight. For the base EfficientNet-B3 model, the Grad-CAM heatmap shows a somewhat diffuse focus across the leaf, with moderate activation on the diseased areas but also highlighting parts of the healthy tissue and leaf edges. This suggests that while the model is able to identify some relevant features, it may not be optimally concentrating on the most important disease indicators. The EfficientNet-B3-CBAM variation displays a more concentrated activation pattern, with higher intensity on the diseased spots. However, the focus appears slightly shifted towards the leaf's edge, potentially indicating that the attention mechanism without proper initialization may be emphasizing less relevant features. Additionally, the EfficientNet-B3-TL model's Grad-CAM heatmap shows a more focused activation pattern compared to the base model, but it is not as precise in highlighting the full diseased area as initially described. The visualization indicates that the model is concentrating on a small portion of the diseased region while missing the full extent of the infection. This suggests that while transfer learning improves the model's ability to identify relevant features, it may not be sufficient on its own to capture the complete disease presentation. Finally, the EfficientNet-B3-TL-CBAM model, which combines transfer learning and the attention mechanism, shows the most refined and targeted activation. The heatmap intensely highlights the diseased spots while almost

completely ignoring the healthy parts of the leaf. This suggests that the full proposed model achieves the best performance in pinpointing the most relevant features for disease detection.

These Grad-CAM results have several implications for the project. Firstly, they provide visual confirmation of the quantitative improvements observed in the ablation study. The progression from the base model to the full proposed model clearly shows how each component contributes to more precise feature localization. The results also highlight the critical role of transfer learning in this task. The stark difference between the non-transfer learning models and those with transfer learning underscores how pre-trained weights significantly boost the model's ability to identify disease-relevant features. This is particularly important in the context of plant disease detection, where the ability to distinguish subtle variations in leaf appearance is crucial. Furthermore, the visualizations demonstrate the synergistic effect of combining transfer learning with the attention mechanism. While CBAM alone without transfer learning shows some improvement in focus, the combination with transfer learning yields the most precise disease localization. This suggests that the attention mechanism is most effective when it can operate on features that have already been refined through transfer learning.

4.4 Web Application Development Results

The web application developed for tomato leaf disease detection demonstrates the ability to carry out its core functionalities as intended. However, there are some notable limitations that require further discussion. One significant limitation is the application's tendency to produce high-confidence predictions for images that do not contain tomato plants or any plants at all. As evidenced in Figure 40, the system incorrectly identified a photograph of a human face as having "Early Blight" with a confidence level of 93.24%. This misclassification highlights a critical flaw in the model's ability to differentiate between plant and non-plant images.

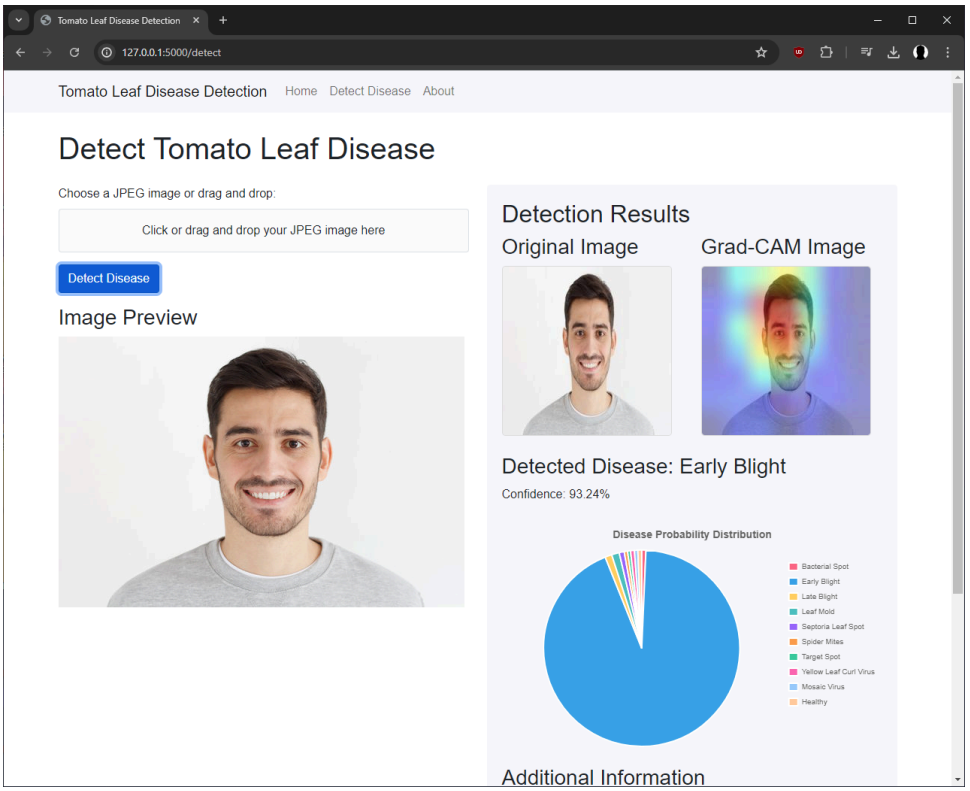


Figure 40 Web Application Disease Detection Limitation

This limitation stems from the model's training process which may have focused exclusively on distinguishing between different types of tomato leaf diseases without incorporating sufficient negative examples of non-plant images. Consequently, the model lacks the ability to recognize when an input falls outside its domain of expertise. This issue underscores the importance of diverse training data that includes not only positive examples of various diseases but also a wide range of negative examples to help the model learn appropriate decision boundaries. This limitation emphasizes the need for additional safeguards in the system, such as an initial plant detection step or a confidence threshold below which the

system should indicate uncertainty rather than providing a potentially misleading diagnosis. Another potential limitation of the web application is its reliance on user-uploaded images due to the reason that images tend to vary significantly in quality, lighting conditions, and framing. While this flexibility is necessary for practical use, it may introduce inconsistencies in the model's performance compared to the controlled conditions under which it was likely trained.

Despite these limitations, the web application serves as a valuable proof of concept for the proposed tomato leaf disease detection model. It demonstrates the potential for deploying deep learning models in user-friendly interfaces accessible to non-expert users. The application successfully showcases the core functionality of disease detection and provides visual aids such as Grad-CAM heatmaps and probability distributions which is quite informative for users. Moving forward, addressing the identified limitations through further research and development could significantly enhance the reliability and practical utility of the system. This might involve expanding the training dataset as well as implementing more robust error handling. Nonetheless, the current iteration of the web application provides a solid foundation for future improvements and serves as a tangible demonstration of the potential for AI-assisted plant disease diagnosis.

4.5 Chapter Summary

This chapter presented a comprehensive analysis of the proposed EfficientNet-B3 model with Convolutional Block Attention Module (CBAM) for tomato plant disease detection. The analysis covered model training, evaluation, comparative studies, and ablation experiments. Firstly, the training process of the proposed model was examined over 5 epochs. The model showed consistent improvement, with training accuracy increasing from 80.23% to 98.15% and validation accuracy rising from 95.12% to 99.60%. Concurrently, both training and validation loss decreased substantially, indicating effective learning without overfitting. Following the training analysis, evaluation results on the test set were presented. The proposed model achieved impressive performance with 99.55% accuracy, 99.60% average precision, and 99.50% for both average recall and F1-score. Additionally, the confusion matrix displayed strong diagonal patterns, further confirming high classification accuracy across all disease classes.

Next, a comparative analysis was conducted against other prominent CNN architectures. The proposed model outperformed VGG16, DenseNet121, ResNet50, and the base EfficientNet-B3, albeit by small margins in some cases. Notably, VGG16 underperformed significantly compared to the other models. To understand the contribution of individual components, ablation experiments were then performed. These experiments revealed the crucial role of transfer learning in improving model performance, while the addition of CBAM provided further refinement. Interestingly, the base EfficientNet-B3 model and the variant with only CBAM showed limited performance, emphasizing the importance of transfer learning in this task. Furthermore, Grad-CAM visualizations were used to provide insights into feature localization across different model variations. These visualizations demonstrated the progression from the base model to the full proposed model, with the combination of transfer learning and CBAM showing the most precise focus on disease-relevant areas. However, it was observed that the transfer learning model alone had limitations in capturing the full extent of the diseased area, particularly for late blight.

Additionally, this chapter examined the results of the web application development. The application successfully demonstrated the core functionality of the proposed model, providing an intuitive interface for users to upload images and receive disease detection results. However, some limitations were identified, particularly the system's tendency to produce high-confidence predictions for non-plant images. This highlighted the need for improved discrimination between plant and non-plant inputs. Despite these limitations, the web application served as an effective proof of concept, showcasing the potential for deploying deep learning models in user-friendly interfaces for plant disease diagnosis.

The chapter concluded by discussing the implications of all results, including the synergistic effect of combining transfer learning with attention mechanisms, challenges in comprehensive disease detection, and areas for improvement in both the model and web application. The high accuracy achieved across multiple models, coupled with the web application's limitations, emphasized the need for evaluation on more challenging, real-world datasets. Overall, this chapter provided a thorough examination of the proposed model's performance and practical implementation, demonstrating its effectiveness in tomato plant disease detection while also identifying areas for future research and enhancement.

5.0 CONCLUSION

This project has successfully developed and evaluated an advanced deep learning model for tomato plant disease detection, marking a significant advancement in precision agriculture. The proposed model, based on an EfficientNet-B3 architecture enhanced with a Convolutional Block Attention Module (CBAM), demonstrated exceptional performance in classifying various tomato plant diseases. With an overall accuracy of 99.55%, the model consistently outperformed other well-known as well as popular CNN architectures such as VGG16, DenseNet121, and ResNet50 across ten distinct disease categories.

The integration of transfer learning and attention mechanisms was a key innovation in this project. By leveraging the weights from the ImageNet dataset, the proposed model efficiently adapted to the specific task of tomato disease detection, reducing the need for extensive training data and computational resources. The incorporation of the CBAM attention mechanism in the proposed model is to increase the model's ability to focus on relevant features, as evidenced by Grad-CAM visualizations. Ablation studies provided valuable insights into the contribution of individual components, highlighting the critical role of transfer learning and its synergistic effect with the attention mechanism. These findings not only validate the design choices but also offer guidance for future research in tomato plant disease detection via deep learning techniques. The development of a web application demonstrated the practical applicability of the proposed model as well as showcase the feasibility of integrating advanced deep learning techniques and end-users in the agricultural sector. However, limitations identified in the application, particularly its tendency to produce high-confidence predictions for non-plant images, underscore the need for further refinements in real-world deployment scenarios.

Looking ahead, several avenues for future research and improvement emerge from this project. Firstly, incorporating a disease severity measurement component into the detection model would significantly enhance its practical value. Currently, the model focuses on identifying the presence of diseases, but quantifying the severity of the infection could provide more nuanced and actionable information to farmers and agricultural professionals. This could involve developing a multi-task learning approach where the model simultaneously classifies the disease type and estimates its severity on a continuous scale. Such an enhancement would require careful annotation of training data to include severity ratings, potentially using standardized scales like the Horsfall-Barratt scale or

percentage-based assessments. This addition would allow for more precise monitoring of disease progression over time and could inform more targeted treatment strategies.

Secondly, enhancing the model's ability to handle real-world variability and edge cases is crucial for its practical deployment. This could involve developing a more robust pre-processing pipeline that can handle various image quality issues such as poor lighting, blurring, or partial occlusion of leaves. Additionally, implementing advanced data augmentation techniques could improve the model's generalization to unseen data. Another aspect of this improvement could be the integration of active learning techniques, where the model identifies and flags uncertain predictions for human expert review, gradually improving its performance on challenging cases over time.

Thirdly, exploring the integration of the tomato plant disease detection model with broader agricultural management systems could significantly increase its impact. This could involve developing APIs to connect the model with existing farm management software, IoT sensors, or automated greenhouse systems. Such integration could enable real-time monitoring and alert systems, where the disease detection model continuously analyzes data from various sources to provide early warnings of disease outbreaks. Furthermore, combining the disease detection results with other relevant data such as the weather forecast, soil conditions, and historical crop performance could lead to more comprehensive predictive models for crop health and yield optimization. These future directions aim to not only improve the technical performance of the model but also to enhance its practical utility and integration into broader agricultural ecosystems, potentially revolutionizing how plant diseases are monitored and managed in modern farming practices.

In conclusion, this project has made substantial contributions to the field of automated plant disease detection, demonstrating the potential of deep learning techniques in addressing critical agricultural challenges. The high performance achieved by the proposed model, coupled with its practical implementation through a web application, lays a strong foundation for future advancements in precision agriculture. The ongoing advancement of technology paves the way for incorporating smart systems into farming practices. These innovations show potential to revolutionize crop care, mitigate disease-related yield losses, and play a crucial role in strengthening worldwide food stability.

REFERENCES

Agarwal, M., Gupta, S. K., & Biswas, K. K. (2020). Development of Efficient CNN model for Tomato crop disease identification. *Sustainable Computing: Informatics and Systems*, *28*, 100407.

Ahmad, M., Abdullah, M., Moon, H., & Han, D. (2021). Plant disease detection in imbalanced datasets using efficient convolutional neural networks with stepwise transfer learning. *IEEE Access*, *9*, 140565-140580.

Alguliyev, R., Imamverdiyev, Y., Sukhostat, L., & Bayramov, R. (2021). Plant disease detection based on a deep model. *Soft Computing*, *25*(21), 13229-13242.

Arsenovic, M., Karanovic, M., Sladojevic, S., Anderla, A., & Stefanovic, D. (2019). Solving current limitations of deep learning based approaches for plant disease detection. *Symmetry*, *11*(7), 939.

Barbedo, J. G. (2018). Factors influencing the use of deep learning for plant disease recognition. *Biosystems engineering*, *172*, 84-91.

Barbedo, J. G. A. (2019). Plant disease identification from individual lesions and spots using deep learning. *Biosystems engineering*, *180*, 96-107.

Barman, U., Choudhury, R. D., Sahu, D., & Barman, G. G. (2020). Comparison of convolution neural networks for smartphone image based real time classification of citrus leaf disease. *Computers and Electronics in Agriculture*, *177*, 105661.

Boulent, J., Foucher, S., Théau, J., & St-Charles, P. L. (2019). Convolutional neural networks for the automatic identification of plant diseases. *Frontiers in plant science*, *10*, 464450.

Chaudhary, P., Sharma, A., Singh, B., & Nagpal, A. K. (2018). Bioactivities of phytochemicals present in tomato. *Journal of food science and technology*, *55*, 2833-2849.

Chen, J., Chen, J., Zhang, D., Sun, Y., & Nanehkaran, Y. A. (2020). Using deep transfer learning for image-based plant disease identification. *Computers and Electronics in Agriculture*, *173*, 105393.

Choudhuri, K. B. R., & Mangrulkar, R. S. (2021). Data Acquisition and Preparation for Artificial Intelligence and Machine Learning Applications. In *Design of Intelligent Applications Using Machine Learning and Deep Learning Techniques* (pp. 1-11). Chapman and Hall/CRC.

FAO. (2024). *FAOSTAT: Crops and livestock products.* Fao.org. https://www.fao.org/faostat/en/#data/QCL

FAO. (2019). *New standards to curb the global spread of plant pests and diseases*. https://www.fao.org/news/story/en/item/1187738/icode/

Ferentinos, K. P. (2018). Deep learning models for plant disease detection and diagnosis. *Computers and electronics in agriculture*, *145*, 311-318.

Fuentes, A., Yoon, S., Kim, S. C., & Park, D. S. (2017). A robust deep-learning-based detector for real-time tomato plant diseases and pests recognition. *Sensors*, *17*(9), 2022.

Garcia-Lamont, F., Cervantes, J., López, A., & Rodriguez, L. (2018). Segmentation of images by color features: A survey. *Neurocomputing*, *292*, 1-27.

Geetharamani, G., & Pandian, A. (2019). Identification of plant leaf diseases using a nine-layer deep convolutional neural network. *Computers & Electrical Engineering*, *76*, 323-338.

Gutierrez, A., Ansuategi, A., Susperregi, L., Tubío, C., Rankić, I., & Lenža, L. (2019). A benchmarking of learning strategies for pest detection and identification on tomato plants for autonomous scouting robots using internal databases. *Journal of Sensors*, *2019*, 1-15.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Hernández, S., & López, J. L. (2020). Uncertainty quantification for plant disease detection using Bayesian deep learning. *Applied Soft Computing*, *96*, 106597.

Huang, M. L., & Chen, Y. A. (2024). Automatic Identification of Tomato Pests Using Parallel Deep Learning Models. *Sensors & Materials*, *36*.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700-4708).

Hughes, D., & Salathé, M. (2015). An open access repository of images on plant health to enable the development of mobile disease diagnostics. *arXiv preprint arXiv:1511.08060*.

Hÿtch, M., & Hawkes, P. W. (2020). *Morphological image operators*. Academic Press.

Jayswal, H. S., & Chaudhari, J. P. (2023). Plant Diseases Detection and Classification Using Machine Learning, Deep Learning, Spectroscopy. In *International Conference on Smart Computing and Communication* (pp. 593-603). Singapore: Springer Nature Singapore.

Ji, M., Zhang, L., & Wu, Q. (2020). Automatic grape leaf diseases identification via UnitedModel based on multiple convolutional neural networks. *Information Processing in Agriculture*, *7*(3), 418-426.

Joshi, R. C., Kaushik, M., Dutta, M. K., Srivastava, A., & Choudhary, N. (2021). VirLeafNet: Automatic analysis and viral disease diagnosis using deep-learning in Vigna mungo plant. *Ecological Informatics*, *61*, 101197.

Karlekar, A., & Seal, A. (2020). SoyNet: Soybean leaf diseases classification. *Computers and Electronics in Agriculture*, *172*, 105342.

Kaur, L., & Sharma, S. G. (2021). Identification of plant diseases and distinct approaches for their management. *Bulletin of the National Research Centre*, *45*, 1-10.

Kirti & Rajpal, N. (2020). Black rot disease detection in grape plant (Vitis vinifera) using colour based segmentation & machine learning. In *2020 2nd international conference on advances in computing, communication control and networking (ICACCCN)* (pp. 976-979). IEEE.

Li, H., & Razi, A. (2019). MEDA: Multi-output Encoder-Decoder for Spatial Attention in Convolutional Neural Networks. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers* (pp. 2087-2091). IEEE.

Li, L., Zhang, S., & Wang, B. (2021). Plant disease detection and classification by deep learning—a review. *IEEE Access*, *9*, 56683-56698.

Li, Y., & Solaymani, S. (2021). Energy consumption, technology innovation and economic growth nexuses in Malaysian. *Energy*, *232*, 121040.

Liu, J., & Wang, X. (2021). Plant diseases and pests detection based on deep learning: a review. *Plant Methods*, *17*, 1-18.

Liu, X., Min, W., Mei, S., Wang, L., & Jiang, S. (2021). Plant disease recognition: A large-scale benchmark dataset and a visual region and loss reweighting approach. *IEEE Transactions on Image Processing*, *30*, 2003-2015.

Mathew, A., Amudha, P., & Sivakumari, S. (2021). Deep learning techniques: an overview. *Advanced Machine Learning Technologies and Applications: Proceedings of AMLTA 2020*, 599-608.

Midhunraj, P. K., Thivya, K. S., & Anand, M. (2023). An Analysis of Plant Diseases on Detection and Classification: From Machine Learning to Deep Learning Techniques. *Multimedia Tools and Applications*, 1-24.

Militante, S. V., Gerardo, B. D., & Dionisio, N. V. (2019). Plant leaf detection and disease recognition using deep learning. In *2019 IEEE Eurasia conference on IOT, communication and engineering (ECICE)* (pp. 579-582). IEEE.

Moupojou, E., Tagne, A., Retraint, F., Tadonkemwa, A., Wilfried, D., Tapamo, H., & Nkenlifack, M. (2023). FieldPlant: A Dataset of Field Plant Images for Plant Disease Detection and Classification With Deep Learning. *IEEE Access*, *11*, 35398-35410.

Nagaraju, M., & Chawla, P. (2020). Systematic review of deep learning techniques in plant disease detection. *International journal of system assurance engineering and management*, *11*, 547-560.

Nikolenko, S. I. (2021). *Synthetic data for deep learning* (Vol. 174). Springer Nature.

Ngugi, L. C., Abdelwahab, M., & Abo-Zahhad, M. (2020). Tomato leaf segmentation algorithms for mobile phone applications using deep learning. *Computers and Electronics in Agriculture*, *178*, 105788.

Pan, S. J., & Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, *22*(10), 1345-1359.

Parnell, S., van den Bosch, F., Gottwald, T., & Gilligan, C. A. (2017). Surveillance to inform control of emerging plant diseases: an epidemiological perspective. *Annual review of phytopathology*, *55*(1), 591-610.

Paul, S. G., Biswas, A. A., Saha, A., Zulfiker, M. S., Ritu, N. A., Zahan, I., ... & Islam, M. A. (2023). A real-time application-based convolutional neural network approach for tomato leaf disease classification. *Array*, *19*, 100313.

Picon, A., Seitz, M., Alvarez-Gila, A., Mohnke, P., Ortiz-Barredo, A., & Echazarra, J. (2019). Crop conditional Convolutional Neural Networks for massive multi-crop plant disease classification over cell phone acquired images taken on real field conditions. *Computers and Electronics in Agriculture*, *167*, 105093.

Prabha, K. (2021). Disease sniffing robots to apps fixing plant diseases: applications of artificial intelligence in plant pathology—a mini review. *Indian Phytopathology*, *74*(1), 13-20.

Radhakrishnan, S. (2020). An improved machine learning algorithm for predicting blast disease in paddy crop. *Materials Today: Proceedings*, *33*, 682-686.

Radwan, N. (2019). *Leveraging sparse and dense features for reliable state estimation in urban environments* (Doctoral dissertation, University of Freiburg, Freiburg im Breisgau, Germany).

Rangarajan, A. K., Purushothaman, R., & Pérez-Ruiz, M. (2021). Disease classification in aubergine with local symptomatic region using deep learning models. *Biosystems Engineering*, *209*, 139-153.

Rangarajan, A. K., Purushothaman, R., & Ramesh, A. (2018). Tomato crop disease classification using pre-trained deep learning algorithm. *Procedia computer science*, *133*, 1040-1047.

Rizzo, D. M., Lichtveld, M., Mazet, J. A., Togami, E., & Miller, S. A. (2021). Plant health and its effects on food safety and security in a One Health framework: Four case studies. *One health outlook*, *3*, 1-9.

Roh, Y., Heo, G., & Whang, S. E. (2019). A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, *33*(4), 1328-1347.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4510-4520).

Sarker, I. H. (2021). Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, *2*(6), 420.

Sethy, P. K., Barpanda, N. K., Rath, A. K., & Behera, S. K. (2020). Deep feature based rice leaf disease identification using support vector machine. *Computers and Electronics in Agriculture*, *175*, 105527.

Shamshiri, R., Weltzien, C., Hameed, I. A., J Yule, I., E Grift, T., Balasundram, S. K., ... & Chowdhary, G. (2018). Research and development in agricultural robotics: A perspective of digital farming.

Shoaib, M., Shah, B., Ei-Sappagh, S., Ali, A., Ullah, A., Alenezi, F., ... & Ali, F. (2023). An advanced deep learning models-based plant disease detection: A review of recent research. *Frontiers in Plant Science*, *14*, 1158933.

Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of big data*, *6*(1), 1-48.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Singh, D., Jain, N., Jain, P., Kayal, P., Kumawat, S., & Batra, N. (2020). PlantDoc: A dataset for visual plant disease detection. In *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD* (pp. 249-253).

Suzuki, K., Kobayashi, Y., & Narihira, T. (2021). Data cleansing for deep neural networks with storage-efficient approximation of influence functions. *arXiv preprint arXiv:2103.11807*.

Tabian, I., Fu, H., & Sharif Khodaei, Z. (2019). A convolutional neural network for impact detection and characterization of complex composite structures. *Sensors*, *19*(22), 4933.

Tan, M., & Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning* (pp. 6105-6114). PMLR.

Too, E. C., Yujian, L., Njuki, S., & Yingchun, L. (2019). A comparative study of fine-tuning deep learning models for plant disease identification. *Computers and Electronics in Agriculture*, *161*, 272-279.

Varshney, D., Babukhanwala, B., Khan, J., Saxena, D., & Singh, A. K. (2022). Plant disease detection using machine learning techniques. In *2022 3rd International Conference for Emerging Technology (INCET)* (pp. 1-5). IEEE.

Wang, D., Wang, J., Ren, Z., & Li, W. (2022). DHBP: A dual-stream hierarchical bilinear pooling model for plant disease multi-task classification. *Computers and Electronics in Agriculture*, *195*, 106788.

Wang, Q., Wu, B., Zhu, P., Li, P., Zuo, W., & Hu, Q. (2020). ECA-Net: Efficient channel attention for deep convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 11534-11542).

Whang, S. E., Roh, Y., Song, H., & Lee, J. G. (2023). Data collection and quality challenges in deep learning: A data-centric ai perspective. *The VLDB Journal*, *32*(4), 791-813.

William Edwards II, D., & Dinc, I. (2020). Classification of protein crystallization images using EfficientNet with data augmentation. In *CSBio'20: Proceedings of the Eleventh International Conference on Computational Systems-Biology and Bioinformatics* (pp. 54-60).

Woo, S., Park, J., Lee, J. Y., & Kweon, I. S. (2018). Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 3-19).

Yang, S., Xiao, W., Zhang, M., Guo, S., Zhao, J., & Shen, F. (2022). Image data augmentation for deep learning: A survey. *arXiv preprint arXiv:2204.08610*.