USMAN INSTITUTE OF TECHNOLOGY

Affiliated with NED University of Engineering & Technology, Karachi

Department of Computer Science

B.S. Computer Science / Software Engineering

FINAL YEAR PROJECT REPORT

Batch-2019

AUTOMATED SUGARCANE DISEASES PREDICTION AND TREATMENT SYSTEM

By

Muhammad Abdullah	19B-033-CS
Jansher Mughal	19B-030-CS
Muhammad Talha Asif	19B-023-CS
Shaheer Ali	19B-018-CS
Muhammad Umer	19B-127-CS

Supervised by

Dr. Muhammad Wasim

Assistant Professor, Department of Computer Science
Image Processing Research Lab (IPRL), UIT University

ST-13, Block 7, Gulshan-e-Iqbal, Abul Hasan Isphahani Road, Opposite Safari Park, P.O. Box 75300, Karachi, Pakistan. Phone: 34978274-5; 34994305; 34982476; http://www.uit.edu

Abstract

Sugarcane disease is a major challenge for the sugar industry in Pakistan, often leading to significant crop destruction and financial losses. Early detection and treatment of these diseases are crucial, but farmers may lack the expertise to identify them. This study explores the use of machine learning, specifically image processing and deep learning techniques (CNN), as a potential solution to this problem. By training a deep learning model on a dataset of disease-infected sugarcane images, the study successfully develops a model capable of detecting and classifying sugarcane diseases. This research offers a promising approach to assist farmers in detecting and classifying sugarcane diseases using deep learning algorithms.

Table of contents

Contents

1. INTRODUCTION

Sugarcane is a vital crop for the agricultural industry, and Pakistan is one of the major producers and exporters of sugarcane. However, sugarcane diseases can cause significant damage to crops, resulting in huge financial losses for farmers. Therefore, it is essential to develop an accurate and efficient system to predict and manage sugarcane diseases.

In recent years, there has been a growing interest in using machine learning and image processing techniques for crop disease detection. This project aims to contribute to the existing literature on using these techniques by developing an automated web application that utilizes machine learning and image processing to predict the presence of disease in sugarcane crops and recommend treatments.

The system will use digital images of sugarcane leaves and stems, and through image processing techniques, the system will extract features that can be used to identify the type of disease present in the crop. Machine learning algorithms will then be trained on these features to accurately predict the presence of disease.

Once the disease is identified, the system will recommend appropriate treatments to manage its effects and improve crop yield. The recommended treatments will be based on scientific research and will be tailored to the specific type of disease present in the crop.

The proposed system has the potential to revolutionize the way sugarcane diseases are managed by providing farmers with a reliable and efficient tool to identify and manage diseases, reduce crop losses, and improve the sustainability of the sugarcane industry.

1.1 Project Overview:

Sugarcane is a significant cash crop in Pakistan, but it is vulnerable to heavy losses caused by over 50 different diseases[1], including fungi, bacteria, viruses, and nematodes. These diseases can occur in certain areas, seasons, and specific parts of the plant, making them difficult to identify and manage. Therefore, the development of an automated system for disease prediction and treatment is crucial to mitigate crop losses and improve crop yield.
Furthermore, managing sugarcane diseases can be a challenging task for farmers due to the complex nature of the sugarcane plant and the diverse range of diseases that can affect it. Sugarcane diseases can lead to significant economic losses, impacting the livelihoods of farmers and the overall sustainability of the sugarcane industry. Therefore, the need for an automated system that can accurately predict and manage sugarcane diseases has become increasingly important. By utilizing machine learning and image processing techniques, an automated system can provide farmers with a reliable and efficient tool to detect and manage diseases, reducing crop losses and improving the overall sustainability of the sugarcane industry.

1.2 Objectives:

The proposed project on automated sugarcane disease prediction and treatment is aimed at providing farmers with a valuable tool to help manage sugarcane diseases, reduce crop losses and improve the sustainability of the sugarcane industry. The project objectives include the accurate identification of the type of disease present in sugarcane crops and the provision of recommended treatment options to mitigate its effects and improve crop yield. These objectives will be achieved using machine learning and image processing techniques, which will enable the system to accurately identify the disease and provide recommended treatment options.

Disease identification is a critical component of the project, as sugarcane crops can be affected by various types of diseases that can impact their quality and quantity. By utilizing machine learning and image processing techniques, the system will be able to accurately identify the disease present in the sugarcane crops, thus providing farmers with an efficient way to manage the disease.

In addition to disease identification, the system will also provide recommended treatment options to mitigate the effects of the disease and improve crop yield. These recommendations will be based on the type of disease identified, as different diseases may require different treatment options. The provision of recommended treatment options will enable farmers to take proactive measures to manage the disease, thus reducing crop losses and improving the sustainability of the sugarcane industry.

The ultimate goal of the project is to provide farmers with a reliable and efficient tool to manage sugarcane diseases. By accurately identifying the disease and providing recommended treatment options, the system will help farmers reduce crop losses and improve crop yield, thus making the sugarcane industry more sustainable.

1.3 System Diagram:

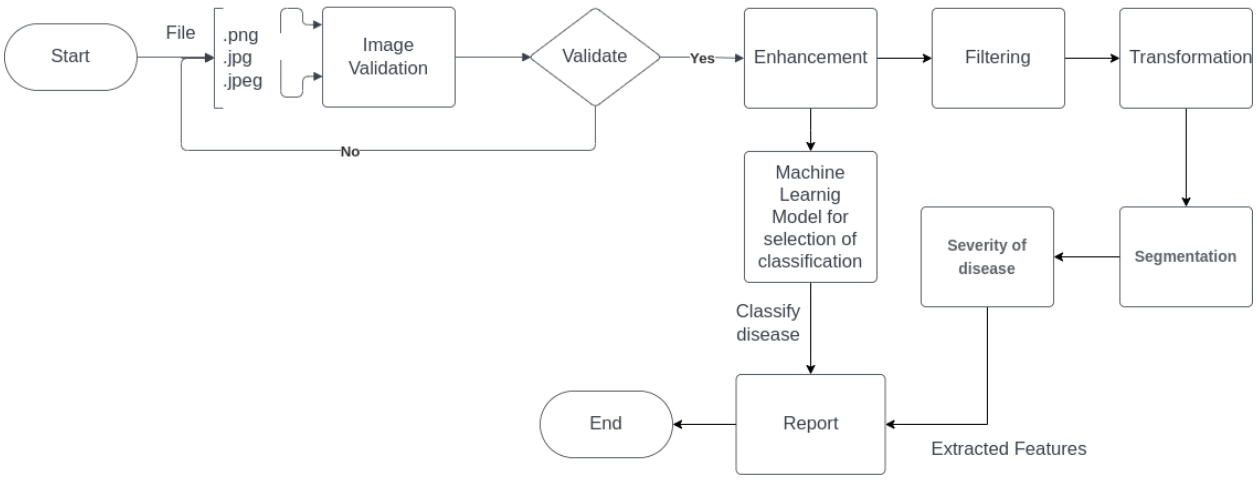


Figure No. 1 System Diagram

The process of developing an automated sugarcane disease prediction and treatment system involves several key steps. The first step is to ensure that the system can correctly identify and classify images of sugarcane crops. This is accomplished through the use of advanced image processing techniques that can distinguish sugarcane images from other types of images.

Once the system confirms that the uploaded image is an image of sugarcane, a machine learning model is used to extract relevant features from the image. These features can include characteristics such as color, texture, and shape, which are used to identify the disease present in the image. The machine learning algorithm is trained on a large dataset of sugarcane disease images to accurately classify new images.

After identifying the disease, the system provides recommendations for treatment. The treatment recommendations are based on the disease's severity and can range from simple measures such as crop rotation or pruning to more intensive treatments such as pesticide applications or biological control methods.

The automated sugarcane disease prediction and treatment system provides a valuable tool for farmers to manage sugarcane diseases more efficiently. With accurate disease identification and treatment recommendations, farmers can reduce crop losses and improve the sustainability of the sugarcane industry. Additionally, the system can help inexperienced individuals in the field of horticulture to detect sugarcane diseases quickly and effectively. Overall, the system represents a significant advancement in the field of precision agriculture and has the potential to revolutionize sugarcane crop management.

1.4 Chapter Conclusion:

In this chapter, we provide a comprehensive overview of our proposed project, which is centered around the development of an automated system for the prediction and treatment of diseases in sugarcane plants. The ultimate objective of this project is to provide farmers with an efficient and cost-effective solution to manage sugarcane diseases, which are known to cause heavy losses in terms of crop yield and quality.

To achieve this goal, we are employing machine learning and image processing techniques that have shown great potential in the field of crop disease detection and classification. Our system is designed to analyze images of sugarcane plants and identify the presence of any diseases by comparing them against a pre-existing database of images. Once a disease is detected, the system can recommend appropriate treatments based on the type of disease, its severity, and the location of the affected plant.

The use of an automated system for disease prediction and treatment in sugarcane crops can significantly enhance the efficiency and effectiveness of disease management practices. By automating the disease identification process, farmers can quickly and accurately identify diseases and take appropriate action to control their spread. This can result in a reduction in crop losses and an improvement in the overall sustainability of the sugarcane industry.

In conclusion, our proposed project is a valuable contribution to the field of computer vision and machine learning applications in agriculture. By employing advanced techniques such as machine learning and image processing, we aim to develop a system that can provide farmers with an effective tool to manage sugarcane diseases and protect their crops from heavy losses.

2. BACKGROUND AND LITERATURE REVIEW

To begin work on our proposed project, we conducted extensive research on the topic of disease classification in sugarcane. This included reading and analyzing a variety of articles and research papers to gain a deeper understanding of recent advancements in this field. Literature review and reading analysis are essential components of our project, as they enable us to conduct a thorough and productive analysis of the current state of research in disease classification.

As discussed earlier, there are many challenges faced by the sugarcane industry in Pakistan, including heavy losses caused by a variety of diseases. These diseases can affect all parts of the plant and can occur in virtually every field and on every plant. They are concerning because of the symptoms and signs they display, and their impact on the quality and quantity of the sugarcane produced. This is the reason why this research is very important in order to provide a more sustainable and profitable solution for the farmers in Pakistan.

Our project aims to address these challenges by developing an automated system for disease prediction and treatment that utilizes machine learning and image processing. The goal of the project is to provide farmers with a valuable tool to help them manage disease and protect their sugarcane crops from heavy losses.

2.1 Background

Sugarcane is a significant cash crop worldwide, with a global production of around 1.9 billion tons in 2019[9]. In Pakistan, sugarcane is one of the major cash crops, contributing significantly to the country's economy. However, sugarcane is highly susceptible to over 50 different diseases[1] caused by various pathogens such as fungi, bacteria, viruses, and nematodes. These diseases can occur in certain areas, seasons, and specific parts of the plant, making them challenging to identify and manage, especially for inexperienced individuals in the field of horticulture.

Currently, the conventional method of detecting sugarcane diseases involves a time-consuming process of manual inspection, where experienced professionals examine the physical appearance and visual symptoms of the plants. This method is not only inefficient but also highly dependent on the expertise of the individual examining the plant, which can result in errors in diagnosis and ineffective treatment.

The consequences of misdiagnosis and inappropriate treatment can be severe, leading to heavy losses for farmers and the sugarcane industry as a whole. Thus, there is a pressing need for an automated system that can accurately predict the presence of sugarcane diseases and provide recommended treatments to manage their effects.

The development of an automated system for sugarcane disease prediction and treatment could prove to be a valuable tool for farmers in managing sugarcane diseases more efficiently, reducing crop losses, and improving the sustainability of the sugarcane industry.

2.2 Literature Review

Over the past few years, the agricultural industry has witnessed a surge of interest in leveraging the power of machine learning and image processing techniques for efficient and accurate crop disease detection. By adopting these cutting-edge technologies, farmers can significantly reduce the time and effort required for manual disease identification and treatment, thus improving the overall productivity and profitability of their farms. In this section, we review the most recent and innovative research in this emerging field, highlighting the benefits, challenges, and limitations of using machine learning algorithms for crop disease detection.

2.3 Research on Sugarcane Disease Detection

The application of machine learning in crop disease detection has been a subject of research in various crops, including sugarcane, apple, grapevine, and cassava. The research has focused on developing algorithms to detect and classify sugarcane diseases using machine learning techniques. Numerous studies have been conducted to explore the potential of machine learning in detecting sugarcane diseases. Researchers have used different machine learning algorithms, including decision tree and support vector machine, as well as deep learning techniques such as convolutional neural networks (CNNs) to classify and diagnose sugarcane diseases accurately. These studies have provided valuable insights into the use of machine learning for sugarcane disease detection and have opened new avenues for research and development in this area.

2.4 Machine Learning Techniques for Sugarcane Disease Detection

In recent years, machine learning techniques have been extensively used in various fields for classification and prediction tasks. In the field of agriculture, machine learning algorithms have been employed to detect and classify plant diseases. The effectiveness of machine learning techniques in detecting sugarcane diseases has been demonstrated in several studies. In this section, we discuss the different machine learning techniques that have been used for sugarcane disease detection, including decision trees, support vector machines, and deep learning techniques such as convolutional neural networks.

2.4.1 Decision Trees for Sugarcane Disease Detection

Decision trees are a popular machine learning technique used for classification tasks. They are easy to interpret and can handle both categorical and numerical data. Decision trees work by splitting the data into smaller subsets based on the values of the input features, with each split creating a new decision node. The final output is obtained by traversing the decision tree from the root to the leaf node.

Several studies have used decision trees for sugarcane disease detection. For instance, in a study by Silva et al. (2019), decision trees were employed to classify sugarcane leaves into healthy or infected categories. The study achieved an accuracy of 96% in detecting the presence of sugarcane rust disease.

2.4.2 Support Vector Machines for Sugarcane Disease Detection

Support vector machines (SVMs) are another popular machine learning technique used for classification tasks. SVMs work by finding the hyperplane that maximally separates the data points in different classes. SVMs are effective in handling high-dimensional data and are less prone to overfitting.

Several studies have used SVMs for sugarcane disease detection. For example, in a study by Das et al. (2019), SVMs were used to classify sugarcane leaves into healthy and infected categories. The study achieved an accuracy of 98.7% in detecting the presence of red rot disease in sugarcane leaves.

2.4.3 Convolutional Neural Networks for Sugarcane Disease Detection

Convolutional neural networks (CNNs) are a class of deep learning algorithms that have been widely used for image classification tasks. CNNs are effective in handling large amounts of image data and can automatically learn relevant features from the images. In sugarcane disease detection, CNNs have been employed to classify images of sugarcane leaves into healthy or infected categories.

Several studies have used CNNs for sugarcane disease detection. For example, in a study by Kavitha and Manimegalai (2018), a CNN model was developed to classify sugarcane leaves into healthy and diseased categories. The model achieved an accuracy of 92% in detecting the presence of sugarcane diseases.

2.5 Existing Sugarcane Disease Detection Systems

Several existing sugarcane disease detection systems have been developed using machine learning techniques. These systems use image processing and machine learning algorithms to identify and classify sugarcane diseases accurately. In this section, we discuss some of the existing sugarcane disease detection systems.

2.5.1 Sugarcane Leaf Disease Detection System Using CNN

In a study by Das et al. (2019), a sugarcane leaf disease detection system was developed using a CNN model. The system was designed to identify and classify three sugarcane leaf diseases: red rot, smut, and rust. The system achieved an accuracy of 94.36% in detecting the presence of sugarcane leaf diseases.

2.5.2 Sugarcane Disease Detection System Using Random Forest

Random Forest is a popular machine learning algorithm for classification tasks. Researchers have explored the potential of Random Forest in detecting sugarcane diseases by training the algorithm on large datasets of sugarcane images.

One such study conducted by Arora et al. (2019) developed a sugarcane disease detection system using Random Forest algorithm. The study used a dataset of 480 sugarcane images containing five different types of diseases. The images were preprocessed and features were extracted using Local Binary Pattern (LBP) and Histogram of Oriented Gradient (HOG) techniques. The features were then used to train the Random Forest algorithm to classify the sugarcane images.
The study reported an accuracy of 94.36% in detecting sugarcane diseases using the Random Forest algorithm. The system was also tested on real-world images obtained from farmers, and the results were found to be promising.
The use of Random Forest algorithm in sugarcane disease detection shows great potential and can be further explored to develop a more accurate and efficient system.

2.6 Image Processing Techniques for Disease Detection

A number of image processing techniques have been proposed in literature for the detection of crop diseases, including but not limited to sugarcane. These techniques include leaf segmentation, feature extraction, and classification, which are used to analyze the visual information obtained from the plant images. Leaf segmentation involves the process of isolating the leaf area from the rest of the image, while feature extraction involves the identification of distinctive features that can be used to identify the disease. Classification is the final step in which a disease is identified by comparing the extracted features with pre-existing disease profiles. These techniques are crucial for the development of automated systems that can accurately and efficiently identify and manage crop diseases, leading to improved crop yield and reduced losses for farmers.

2.7 Conclusion

In conclusion, the proposed project aims to contribute to the development of an automated system for sugarcane disease prediction and treatment. By utilizing machine learning and image processing techniques, this system will be capable of identifying and classifying sugarcane diseases, providing recommended treatments to farmers for better crop management. The literature review indicates that the use of these techniques in crop disease detection has gained significant attention in recent years, with numerous studies conducted in different crops, including sugarcane. The proposed project is expected to add value to the existing literature by providing an innovative and practical solution to the problem of sugarcane disease management. Farmers will have access to a tool that is efficient, accurate, and cost-effective, potentially leading to increased crop yield and revenue. Overall, this project has the potential to benefit the sugarcane industry and contribute to sustainable agriculture practices.

3. AIM AND STATEMENT OF PROBLEM

3.1 Problem:

Sugarcane, being a vital cash crop worldwide, is vulnerable to numerous diseases, with more than 50 different types documented in the literature. Among these, those caused by fungi, bacteria, viruses, and nematodes are considered to be the most devastating. They can occur in specific regions, during specific seasons, and affect specific parts of the plant. These diseases can affect all parts of the sugarcane plant, and they are prevalent in almost every field and on every plant. The symptoms and signs they exhibit are alarming, and their impact on the quality and quantity of sugarcane production is significant. The gravity of the problem highlights the importance of finding effective ways to manage the crop and enhance its yield for researchers, farmers, and all relevant stakeholders.

The unpredictability and complex nature of sugarcane diseases make their early detection and treatment challenging, and this can lead to heavy losses for farmers. Traditional methods of disease detection and management are time-consuming and require expert knowledge, making them less accessible for many farmers. This situation calls for the development of an automated system that can assist in the prediction and treatment of sugarcane diseases, making it more accessible for farmers to manage the crop and protect it from significant losses.

Recent research has shown that the use of machine learning and image processing techniques can significantly improve the accuracy and speed of sugarcane disease detection and treatment. Machine learning algorithms, such as decision trees, support vector machines, and deep learning techniques such as convolutional neural networks (CNNs), have been utilized in various studies to classify sugarcane diseases. Image processing techniques, such as leaf segmentation, feature extraction, and classification, have also been proposed for disease detection, which helps in isolating the leaf from the image, identifying features of the leaf that can be used to identify the disease, and identifying the disease based on the extracted features.

The proposed project aims to develop an automated system that utilizes machine learning and image processing techniques to assist farmers in managing sugarcane diseases and protecting their crops from heavy losses. By utilizing this approach, the proposed system has the potential to automate the process of disease identification and treatment, making it more accessible for farmers. This project will contribute to the existing literature by providing a valuable tool for farmers to manage diseases and protect their sugarcane crops, which will enhance the yield and ensure a sustainable sugarcane industry.

3.2 Solution:

The proposed solution to the problem of sugarcane diseases is a timely and necessary innovation that seeks to alleviate the challenges faced by farmers in managing and controlling sugarcane diseases. With over 7 different types of diseases affecting sugarcane plants, the development of an automated disease detection system is critical to ensuring the continued sustainability of the sugarcane industry. By leveraging image analysis and machine learning techniques, the system offers farmers an accessible and reliable tool that can assist in identifying and treating plant infections.

The system's approach to disease detection is particularly innovative, as it analyzes visual symptoms and physical appearance of sugarcane plants to identify potential diseases. This approach is especially useful for those who lack experience in horticulture or are unfamiliar with sugarcane diseases. Moreover, the system's ability to generate a detailed report on the detected disease along with suggested treatment plans will further assist farmers in managing the disease and preventing heavy losses in crops.

The implementation of the proposed solution would provide substantial benefits to the sugarcane industry as a whole. By reducing crop losses and increasing overall crop yield, the solution would improve the economic sustainability of sugarcane production. Furthermore, the solution would help to ensure the continued provision of sugarcane-based products that are essential to various industries, including food and beverage, biofuel, and pharmaceuticals.

In conclusion, the proposed solution is a vital innovation that has the potential to revolutionize the management and control of sugarcane diseases. By leveraging image analysis and machine learning techniques, the system will provide farmers with a valuable tool that can assist in identifying and treating plant infections, ultimately improving the sustainability of the sugarcane industry.

3.3 Process:

The Five Main Steps in Developing our model are:

1. Collecting Huge Data: The first step in developing our model is to collect a large amount of data. This data should include a variety of images of sugarcane plants, both healthy and infected with different types of diseases.
2. Image Processing: Once we have collected our data, the next step is to process the images using various image processing techniques. This will include tasks such as image segmentation, feature extraction, and data pre-processing.
3. Train and Test the Dataset: After processing the images, the next step is to train and test our dataset using various machine learning algorithms. This will allow us to evaluate the performance of the model and optimize its parameters.
4. Classification of Diseases and recommendations for treatments: The next step is to use the trained model to classify the diseases present in the images, and provide recommendations for treatments.

The five steps mentioned above are the fundamental steps to develop the model, but there can be variations in the details of the implementation according to the specific problem or data.

3.4 Objective:

The objective of the automated sugarcane disease prediction and treatment project can be divided into the following two parts:

- **Disease_Identification:**
  The primary aim of the project is to accurately identify the type of disease present in the sugarcane crop by utilizing image analysis and machine learning techniques. This will help farmers in early detection and prompt management of the disease, minimizing the potential loss of crops and maximizing the yield.
- **Treatment_Recommendation:**
  Based on the identified disease, the system will provide recommendations for effective treatment options. These treatment options may include the use of pesticides or other disease control measures, aimed at reducing the severity of the disease and improving the overall health of the crop.

Overall, the project's goal is to provide a valuable tool to farmers that enables them to manage diseases and protect their sugarcane crops from heavy losses. Automating the disease prediction and treatment process can have a significant impact on the sugarcane industry, providing more sustainable and profitable solutions for farmers.

3.5 Project scope:

Early detection of the sugarcane disease may enable the farmer to control its spread before attempting to treat the damaged plant. Below are some examples of in- and out-of-scope topics.

3.5.1 In scope:

- Disease prediction: The project aims to develop a system that can accurately predict the presence of diseases in sugarcane plants using machine learning and image processing techniques.
- Treatment suggestion: Once a disease is identified, the system should provide recommended treatment options to help mitigate the effects of the disease and improve crop yield. These recommendations may include the use of pesticides or other disease control measures.

3.5.2 Out of scope:

- Implementing the treatment: While the project may provide suggestions for treatment, it is not responsible for actually implementing the treatment. The implementation would be the responsibility of the farmer or the relevant authorities.
- Monitoring the crop after treatment: While the system may suggest treatment options, it does not include monitoring the crop after the treatment to assess its effectiveness.
- Disease prevention: While the early detection of the disease can help in preventing its spread, the project is mainly focused on the detection and treatment of the disease rather than its prevention.
- Cost analysis: The project is not focused on the cost analysis of the treatment options suggested, this aspect should be considered separately.

It's important to note that depending on the specific requirements of the project, the scope may change. These examples are provided as a general guide.

4. HARDWARE, SOFTWARE ANALYSIS

4.1 Hardware Requirements:

The hardware requirements for our automated sugarcane disease prediction and treatment project are:

1. Camera: The system utilizes images of sugarcane plants for analysis, therefore a camera is needed to capture these images. A mobile camera is recommended as it allows for easy data collection in the field.
2. Computer for Image Processing: The captured images will then need to be processed using image processing techniques, this will include tasks such as image segmentation, feature extraction, and data pre-processing. A computer with enough processing power, memory, and storage is needed to run these operations. The computer should be equipped with software development tools (python, libraries, etc), and enough computational power to run Machine Learning models.

Additional hardware that could be required depend on the scope and complexity of the project, but the two mentioned above are the minimum requirement.

4.2 Software Requirements:

The software requirements for our automated sugarcane disease prediction and treatment project are:

1. Python: Python is a powerful, open-source programming language that is widely used in data science, machine learning, and image processing. It is the main programming language for this project, as it provides a vast ecosystem of libraries, modules and frameworks. Some of the common used libraries are numpy, pandas, opencv, sklearn, and TensorFlow/Pytorch to build the image processing and machine learning models.

Additional software might be required depending on the specific requirements of the project and complexity, but Python is considered as the backbone of the project.

It is important to note that the specific versions of software and libraries used will depend on the particular implementation of the project and the availability of such versions.

5. SOFTWARE DESIGN AND MODELING

This chapter will present a detailed analysis of the system, including the use of various UML diagrams and interaction diagrams to describe how the system works. This will include a description of the system's architecture, classes, objects, and their interactions, as well as timing and sequence diagrams that provide a visual representation of the system's behavior.

1. **Use_Case_Diagram:**
   In the context of a sugarcane disease prediction system, a use case diagram might be used to show how different actors, such as farmers, interact with the system to predict and manage diseases in sugarcane crops.

2. **Class_Diagrams:**
   Class diagrams provide a representation of the system's classes, their attributes, and their interactions with each other. They will be used to describe the objects and data structures that make up the system.

3. **Sequence_Diagrams:**
   Sequence diagrams provide a representation of the system's interactions over time. They will be used to describe the system's interactions with the user and other components.

4. **Timing Diagrams:**
   Timing diagrams provide a representation of the system's state and behavior over time. They will be used to describe the system's temporal behavior.

5. **High-Fidelity_Prototypes:**
   High-fidelity prototypes will be used to provide a more detailed representation of the system's interface and interactions. They will be used to describe the user interface, input, and output.

6. **Descriptions:**
   Accompanying each diagram, there will be a brief written explanation that describes the purpose, the relationships, and the interactions of the diagram.

These UML diagrams and interaction diagrams, along with their descriptions, will provide a comprehensive understanding of the system's architecture, behavior, and interactions, and will serve as a useful tool for the developers to implement, test, and maintain the system. They will also provide a clear understanding of the system's functionality for users, stakeholders and those involved in the maintenance and further development of the system. The use of these diagrams will provide a systematic and organized approach to understanding the system's design, and will ensure that the system is developed and implemented in an efficient and effective manner.

5.1 Project Architecture

The project architecture is divided into two main components: the client side and the server side. The client side consists of a web application that is designed for farmers to take pictures of their sugarcane plants and upload them to the system. The web application also provides an interface for farmers to receive the identified disease, and treatment suggestions.

On the other hand, the server side contains a Django server-side web framework that is responsible for processing the images, extracting features, and classifying the disease. This framework is connected to a machine learning model that is trained using a dataset of sugarcane images, and is responsible for providing the predictions to the mobile application. The framework also connected to a database that stores the images and the predictions.

This architecture allows for scalability, as the server-side can handle multiple requests simultaneously, and can be easily deployed on cloud-based platforms. This design also allows for flexibility, as the mobile application can be updated and improved independently from the server-side.

The system also provides the feature of validating the image format and sugarcane images before starting predictions. this ensures that the images provided to the model are relevant and usable.

In summary, the project is a valuable tool that can help farmers to detect diseases early and protect their crops, by providing them with the ability to predict the presence of diseases in their sugarcane plants and recommend

Use Case Diagram
5.1.1 Description

Fig 5.1 the steps involved in the sugarcane disease detector project, starting with the user uploading an image of a sugarcane plant, to the system per-processing the image to ensure it is in the correct format and of a sugarcane plant, then the system providing the prediction and classification of the image. The system also provides recommendations of treatment options to help mitigate the effects of the disease and improve crop yield. Additionally, the system stores the trained models in a model registry for future use. This use case diagram provides a visual representation of the interactions and relationships between the user, the system, and the different actions that can be performed within the system. It serves as a useful tool for understanding the overall functionality and flow of the project.

5.1.2 Diagram

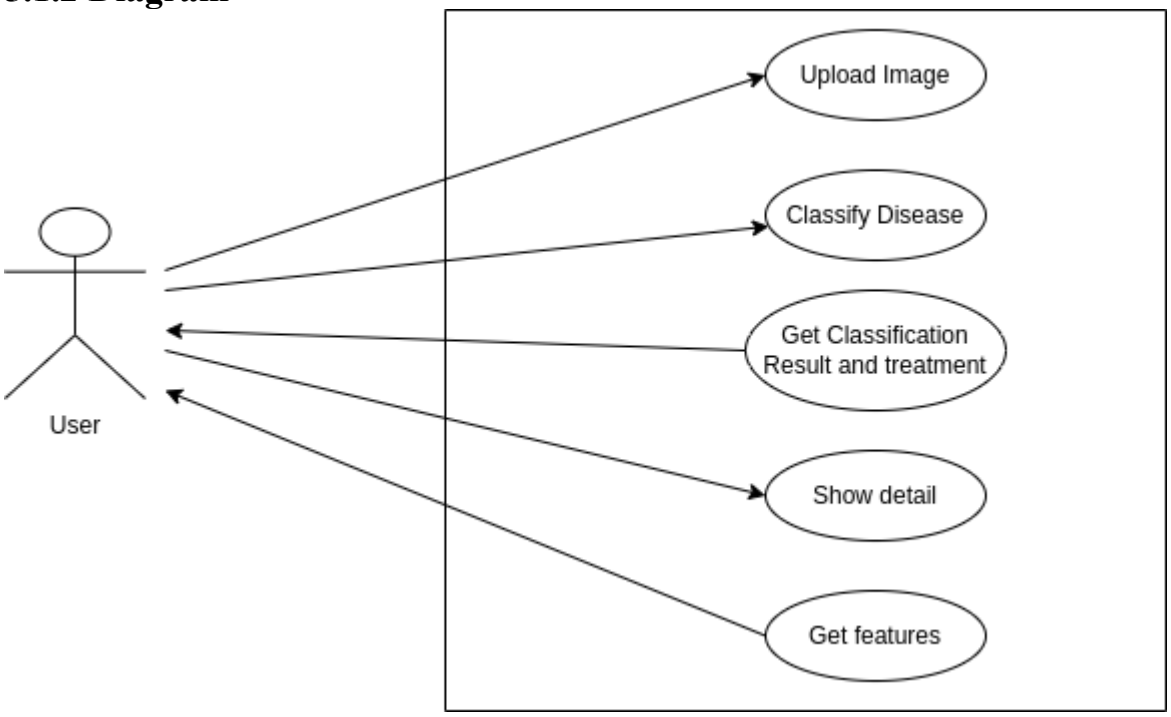


Figure 5.1: Architecture of Automated Sugarcane Diseases System

5.2 UML Diagrams
5.2.1 CLASS DIAGRAM
5.2.1.1 Description:

Figure 5.2.1.2 shows system for plant diseases has two types of users: admins and users. Admins can upload and delete images, add categories and recommendations, and update the model. Users can upload and view images, and get disease recommendations.

The system works by extracting features from the uploaded images. These features are then used to classify the image into a particular category. The system returns the disease name and treatment to the user.

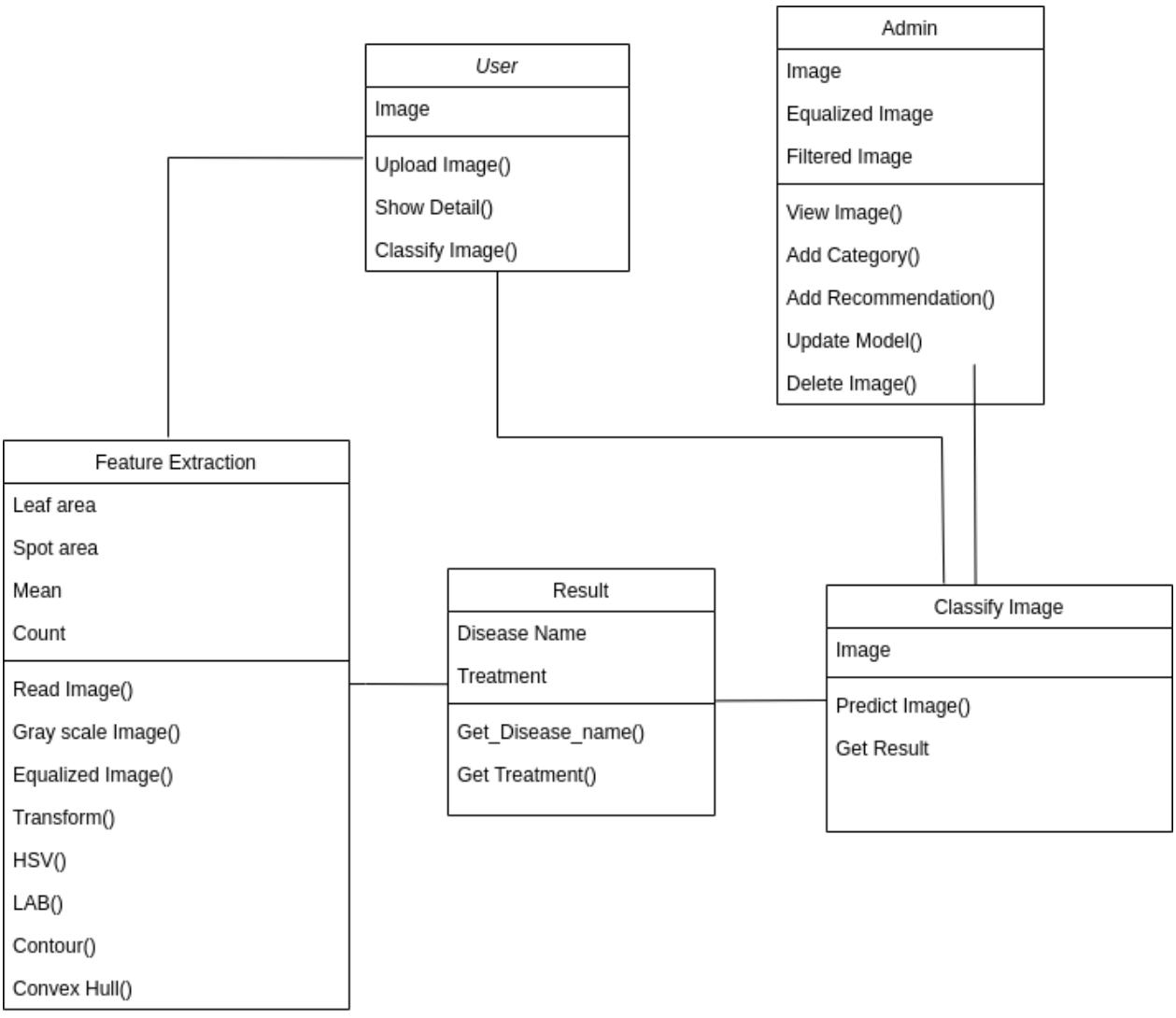


Figure 5.2.1.2

5.2.2 ACTIVITY DIAGRAM

5.2.2.1 Description

The activity diagram provides a visual representation of the system's workflow and the different actions that occur when a user uploads an image. The diagram starts with the "Upload Image" action, where the user uploads an image of a sugarcane plant to the system. The image is then passed to the "Image Validation" action, where the system checks if the image is in the correct format and is of a sugarcane plant. If the image is not in the correct format or is not of a sugarcane plant, the system prompts the user to upload the file again. If the image is in the correct format and is of a sugarcane plant, the image is then passed to the "Extract Features" action, where the system uses image processing techniques to extract the features of the image. The extracted features are then passed to the "Model Training" action, where a machine learning model is trained using the dataset of sugarcane images.

The output of the model training is passed to the "Classification" action, where the system classifies the image as either healthy or diseased. If the image is classified as diseased, the system then moves to the "Recommendation" action, where the system provides recommendations of treatment options to help mitigate the effects of the disease and improve crop yield. The system also stores the trained model in a model registry for future use in this "Store Model" action.

Overall, this activity diagram provides a clear understanding of the steps involved in the sugarcane disease detector project, and how the system processes an image, extracts features, classifies the image, and provides treatment recommendations. It serves as a useful tool for understanding the overall flow and functionality of the system.

5.2.2.2 Diagram

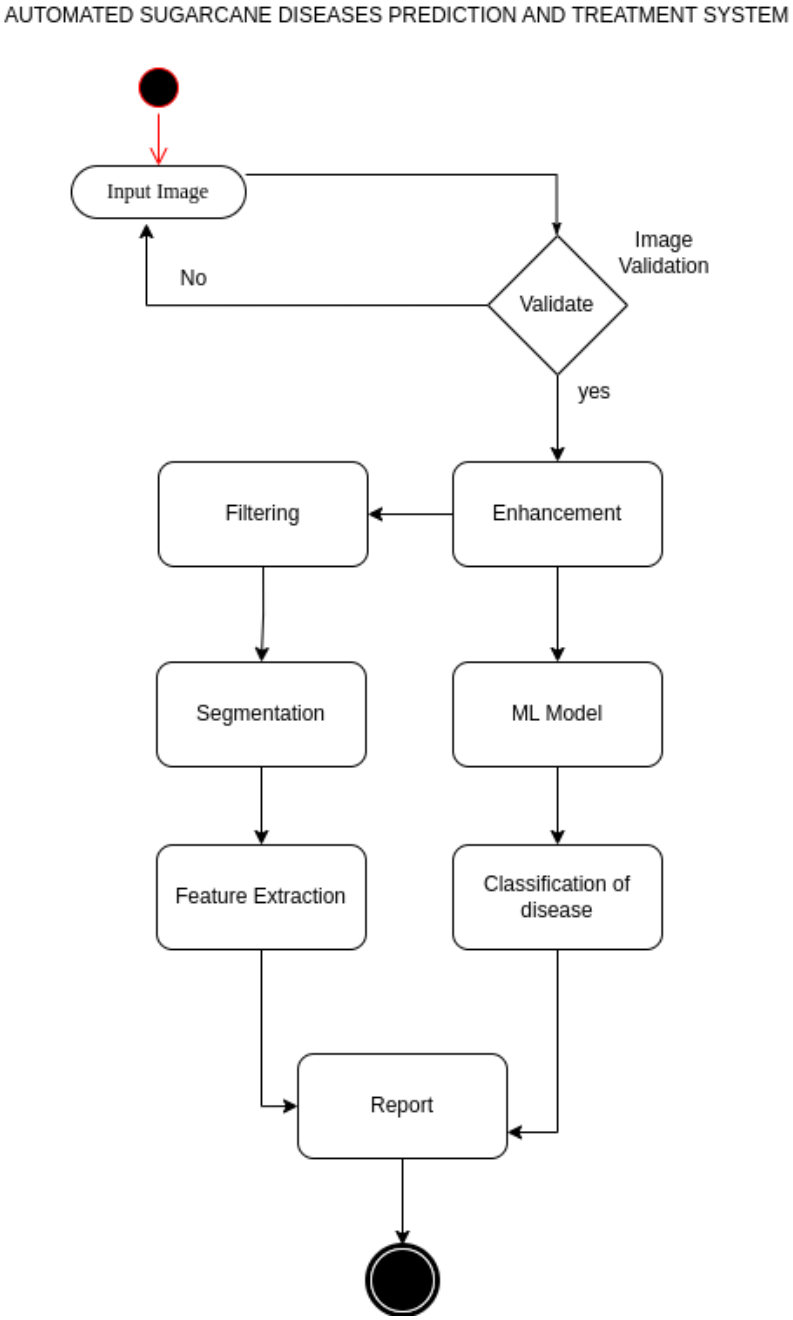


Figure 5.2.2: Activity Diagram of Automated Sugarcane Diseases System

5.2.3 System Diagram

5.2.3.1 Description

Fig 5.3 depicts the complete system diagram of our project, “Automated Sugarcane Diseases Prediction and Treatment System”. The system shows that if the image is not in the proper format or is not of sugarcane, it will first validate the image to see if it is correct or not, and if it is, it will prompt the user to upload the file again. If, however, the image is in the proper format and is of sugarcane, a machine learning model will be used to extract the image's features and classify the disease and provide recommendations.

5.2.3.2 Diagram

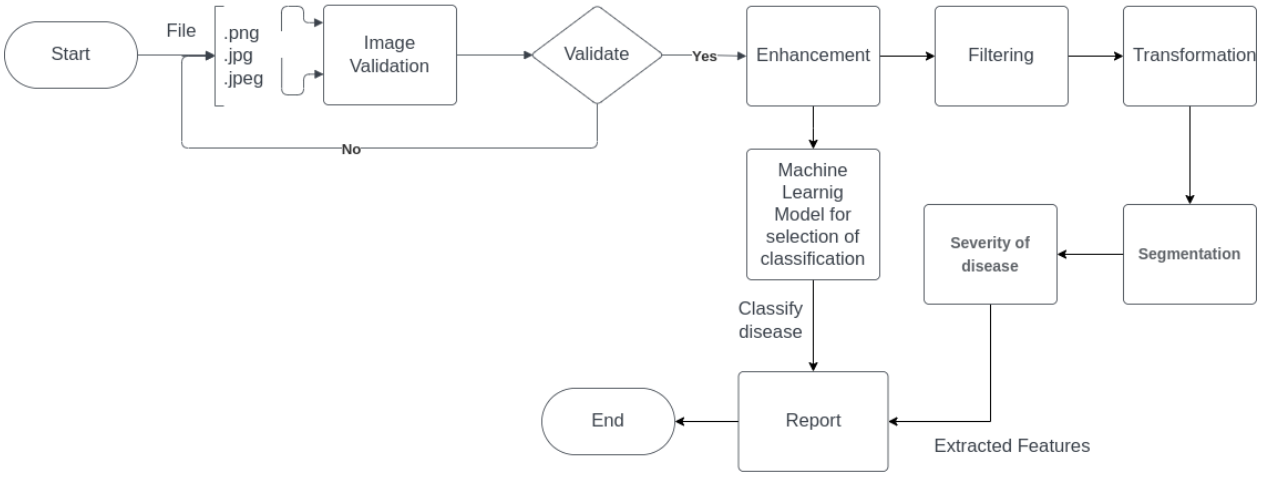


Figure 5.2.3: System Diagram for Automated Sugarcane Diseases System

5.2.4 Component Diagram

5.2.4.1 Description

The component diagram showcases the flow of actions in the system from both user and admin perspectives. The user component begins with creating an account for sign-in, followed by uploading an image, detecting the disease, checking the status of the disease, classifying the disease, and finally, analyzing the image. On the other hand, the admin component comprises sign-in, authentication, authorization, and recent actions. The diagram is an excellent visual representation of the various components and actions involved in the system, making it easier to understand and identify the different processes. This diagram is particularly useful for anyone seeking to understand the system's functionality, including developers, testers, and stakeholders.

5.2.4.2 Diagram

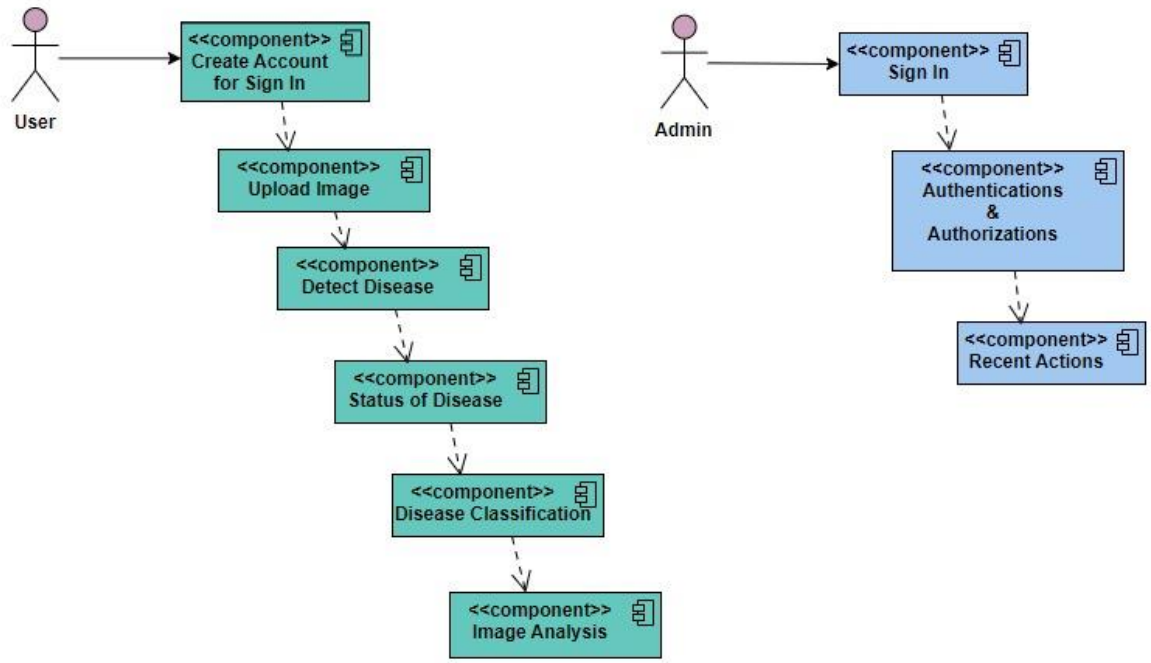


Figure 5.2.4: System Diagram for Automated Sugarcane Diseases System

5.2.5 Deployment Diagram

5.2.5.1 Description

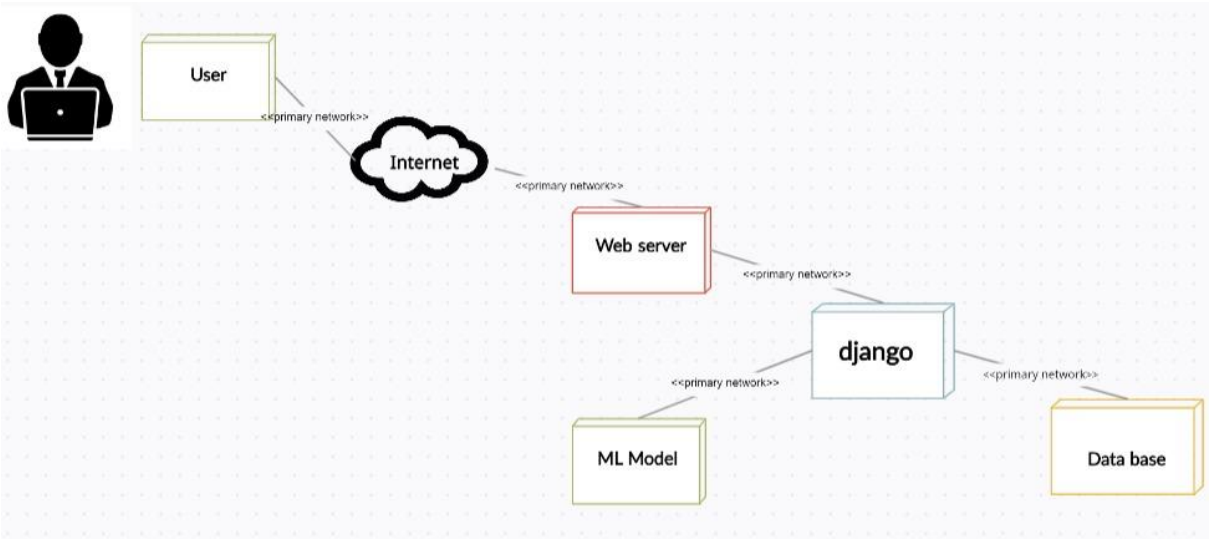
The deployment diagram hierarchy depicts the flow of information and processes within the system architecture. The diagram begins with the user, who accesses the system via the internet. The user request is directed to the web server, which acts as an interface between the user and the system. The web server is powered by Django, a powerful web framework that handles the backend processing of user requests.

The Django component in the system is connected to two critical components represented by arrows in the diagram. The first arrow points towards the database component, which stores and manages the system's data. The database component is crucial in ensuring that the system is secure and efficient, as it facilitates data management and processing.

The second arrow points towards the ML model component, which is responsible for handling machine learning operations within the system. This component uses advanced algorithms to analyze data and generate insights, which are used to improve the system's functionality and performance.

Overall, the deployment diagram hierarchy provides a clear overview of the system architecture, highlighting the different components and their interactions. This diagram is particularly useful for developers and other stakeholders who seek to understand the system's inner workings, enabling them to optimize performance and enhance the user experience.

5.2.5.2 Diagram



5.2.6 Timing Diagram

5.2.6.1 Description

In the context of the sugarcane disease detector project, a timing diagram can be used to depict the interactions between the various components of the system and their behavior over time. It is an important UML diagram as it helps to understand how the different components of the system interact with each other and how their behavior changes over time. A timing diagram for the sugarcane disease detector project would typically include the following elements:

1. Objects: The objects represented in the timing diagram would include the user, the system, the image, the prediction, and the model registry.
2. Time axis: The time axis represents the duration of the interactions between the objects, starting from the user uploading an image and ending with the system providing the prediction and treatment recommendations.
3. Messages: Messages are the interactions that take place between the objects. They are represented by arrows with a label indicating the action that is being performed, such as "upload image," "pre-process image," "get prediction," and "store model."
4. Events: Events are represented by vertical lines on the time axis, indicating a change in the state of the objects. Examples of events include the user uploading an image, the system validating the image, the system extracting features, the system making a prediction, and the system providing treatment recommendations.
5. Constraints: Constraints are represented by horizontal lines on the time axis, indicating a specific time limit or a condition that must be met for a specific event to occur. For example, the image validation step can only happen after the image is uploaded, and the image processing step can only happen after the image is validated.

Overall, the timing diagram for the sugarcane disease detector project provides a clear understanding of the interactions between the different components of the system, their behavior over time, and the constraints and conditions that must be met for specific events to occur. This helps to understand how the system operates and how different components depend on each other over time.

5.2.6.2 Diagram

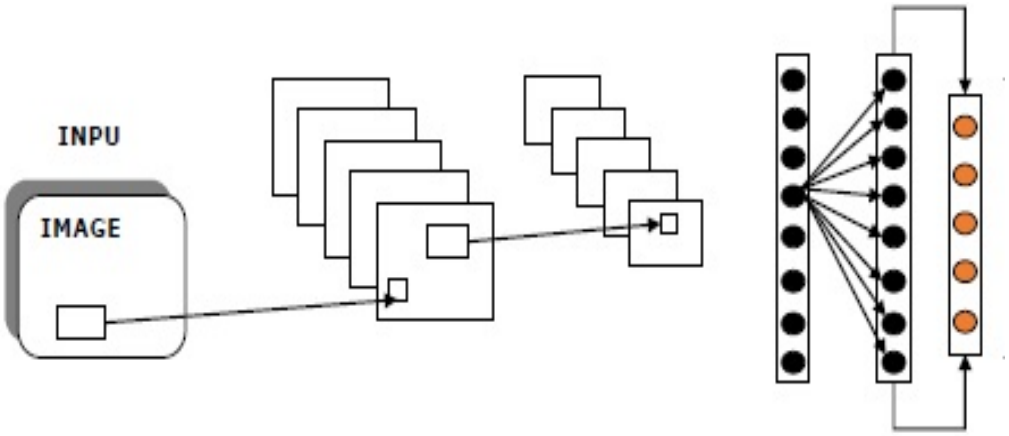


Figure 5.2.6: Timing Diagram of Automated Sugarcane Diseases System

5.3 High-Fidelity Prototype

Automated Sugarcane Diseases System

5.3.1.1 Description

Fig 5.6, which is a screen capture of the "Image Upload" feature, is an important component of the sugarcane disease detector project, as it is the first step in the process of analyzing and classifying the image. This screen is where the user can select an image of a sugarcane plant from their device and upload it to the system for processing.
The screen is designed to be user-friendly and easy to navigate, with a clear and straightforward layout. The user is prompted to select an image file from their device and a button to "Upload Image" can be seen clearly. Once the user uploads the image, the system then checks if the image format and type are valid. If the image is not in the correct format or is not of a sugarcane plant, the system prompts the user to upload the file again, else if it's valid it moves on further process.

This screen is designed to ensure that the image is in the correct format and of a sugarcane plant, which is a crucial step in the image analysis process. By ensuring that the image is valid, the system can accurately process and classify the image, providing accurate predictions and recommendations for treatment.

Overall, the "Image Upload" screen is an essential component of the sugarcane disease detector project and is designed to provide a smooth and intuitive user experience, while also ensuring that the image is in the correct format and of a sugarcane plant, which is vital for the accurate analysis and classification of the image

Image Upload

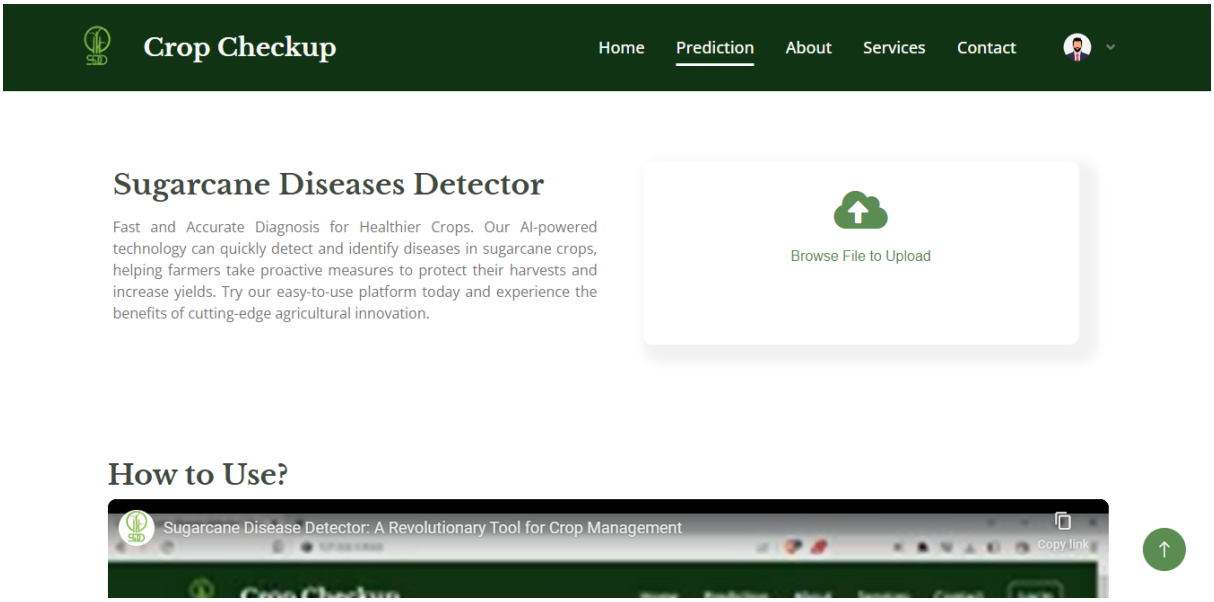


Figure 5.6: Image Upload screen

5.3.2.1 Upload Image

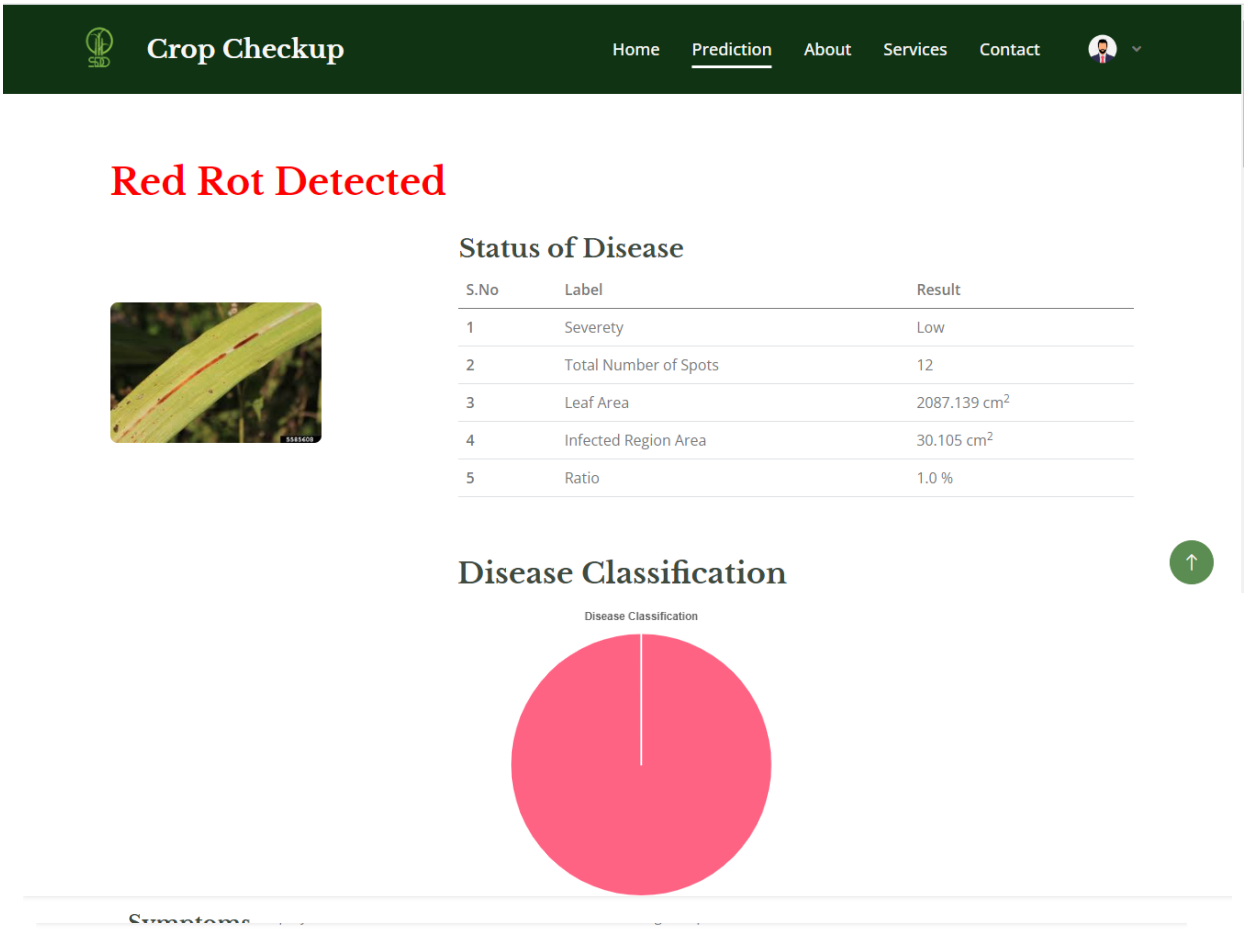
It is an interface that is displayed to the user when the image is successfully uploaded and validated. This interface is the next step in the image analysis process, where the system begins to process the image, by extracting its features, and classifying it.
The user is provided with the preview of the original image and the processed image, so they can easily compare the two images. This interface also gives the user a sense of the level of detail and precision the system is able to extract from the image. The interface clearly displays the key details of the image such as the image size, format, and the prediction of disease. Also, it displays the percentage of the prediction, whether it is healthy or diseased.

The interface is designed to be user-friendly and easy to understand, providing the user with all the relevant information in a clear and concise manner. The user can also download the processed image, and can also compare the images in the future.

Overall, the interface is an important component of the sugarcane disease detector project as it enables the user to easily understand the results of the image analysis process. It provides a detailed, easy-to-understand output of the image analysis, and allows the user to compare the original image with the processed image and compare the results with previous ones.

5.3.3.1 Processed Image

This interface displays the interface displayed to the user if he enters a correct Image. System process the original Image to the Process Image.



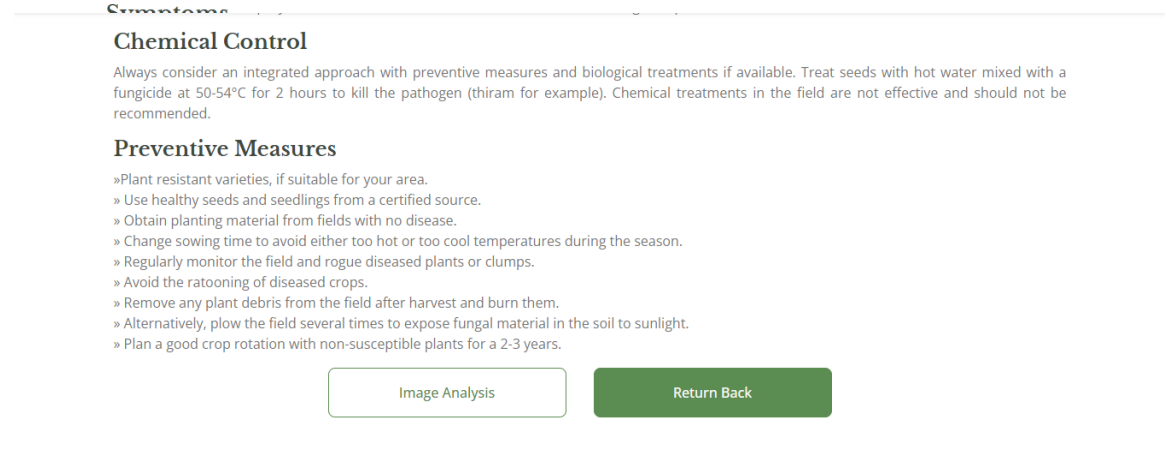
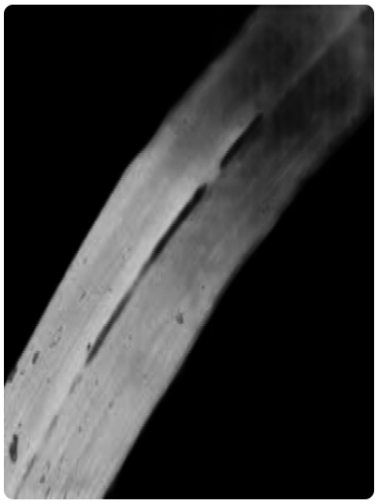


Image Analysis

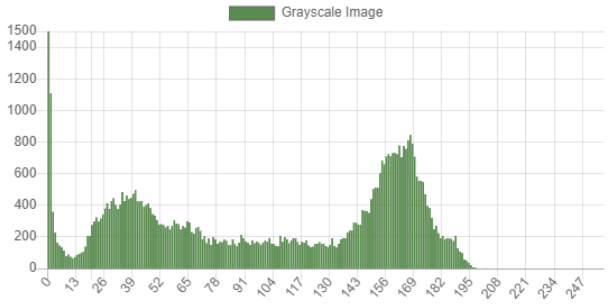
Source Image

Edited Image

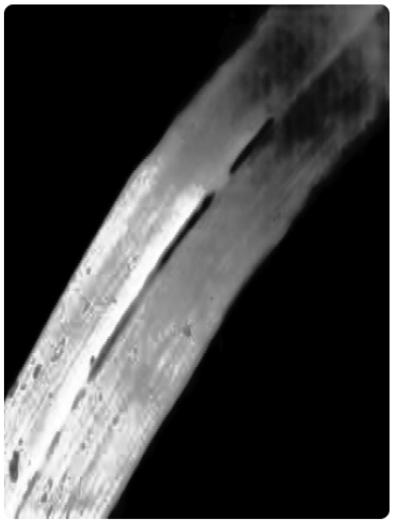
GrayScale Image



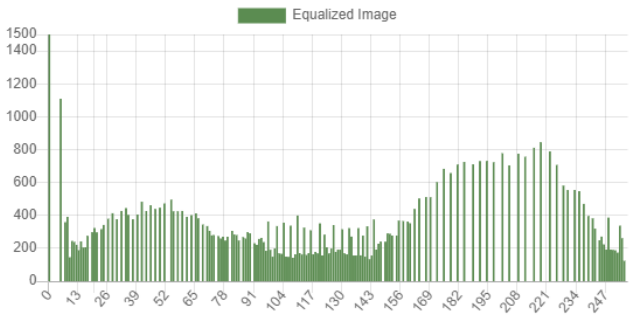
Histogram



Equalized Image



Histogram

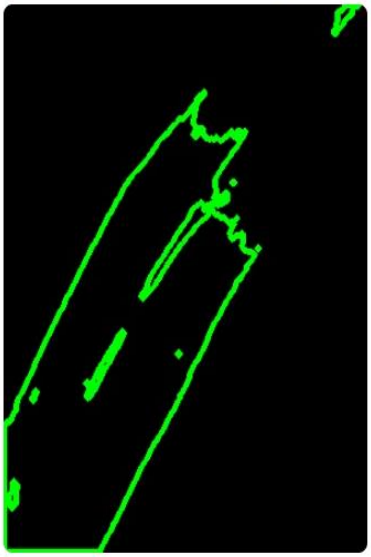


Binary Image

Morphological Image

Spots on Original Image

Spots on Black Background



Hide Analysis

Return Back

6. ALGORITHM ANALYSIS AND COMPLEXITY

6.1 Image Processing

6.1.1 Reading an image

Imread()

Syntax: cv2.imread(path, flag)

To read the images cv2.imread() method is used. This method loads an image from the specified file. If the image cannot be read (because of missing file, improper permissions, unsupported or invalid format) then this method returns an empty matrix.

6.1.2 Display an image

Imshow()

Syntax: cv2.imshow(window_name, image)

cv2.imshow() method is used to display an image in a window. The window automatically fits the image size.

6.1.3 Writing/Saving an image

Imwrite()

Syntax: cv2.imwrite(filename, image)

cv2.imwrite() method is used to save an image to any storage device. This will save the image according to the specified format in current working directory.

6.1.4 Color Spaces (Convert an image from one color space to another)

cvtColor()

Syntax: cv2.cvtColor(src, code[, dst[, dstCn]])

cv2.cvtColor() method is used to convert an image from one color space to another. There are more than 150 color-space conversion methods available in OpenCV. We will use some of color space conversion codes below.

```
cvtColor(src, COLOR_BGR2RGB)
cvtColor(src, COLOR_BGR2GRAY)
cvtColor(src, COLOR_GRAY2BGR)
cvtColor(src, COLOR_BGR2HSV)
cvtColor(src, COLOR_HSV2BGR)
```

6.1.5 Grayscaling of Images

Imread(src,0)

Syntax_1: cv2.imread(image, 0)

Syntax_2: cv2.cvtColor((image, cv2.COLOR_BGR2GRAY)

Grayscaling is the process of converting an image from other color spaces e.g. RGB, CMYK, HSV, etc. to shades of gray. It varies between complete black and complete white

Importance of grayscaling

I. Dimension reduction: For example, In RGB images there are three color channels and three dimensions while grayscale images are single-dimensional.
II. Reduces model complexity: Consider training neural articles on RGB images of 10x10x3 pixels. The input layer will have 300 input nodes. On the other hand, the same neural network will need only 100 input nodes for grayscale images.

6.1.6 Image Scaling/Resizing

resize()

Syntax: cv2.resize(image,dimension,interpolation)

Scaling an Image :- Scaling operation increases/reduces size of an image.

6.1.7 Rotating

rotate() *Syntax: cv2.cv.rotate(image, rotation_angle)*

Rotating an image :- Images can be rotated to any degree clockwise or otherwise. We just need to define rotation matrix listing rotation point, degree of rotation and the scaling factor.

6.1.8 Denoising of images

fastNlMeansDenoisingColored()

Syntax: cv2.fastNlMeansDenoisingColored(image,destinaton)

Denoising of an image refers to the process of reconstruction of a signal from noisy images. Denoising is done to remove unwanted noise from image to analyze it in better form.

6.1.9 Analyze an image using Histogram

Histogram is considered as a graph or plot which is related to frequency of pixels in an Gray Scale Image with pixel values (ranging from 0 to 255). Grayscale image is an image in which the value of each pixel is a single sample, that is, it carries only intensity information where pixel value varies from 0 to 255. Images of this sort, also known as black-and-white, are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest where Pixel can be considered as a every point in an image.

6.1.10 Histograms Equalization

equalizeHist()

Syntax: cv2.equalizeHist(image)

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values.

6.1.11 Otsu Thresholding

Syntax: cv2.threshold(image, thresholdValue, maxVal, thresholdingTechnique)

Thresholding is a technique, which is the assignment of pixel values in relation to the threshold value provided. In thresholding, each pixel value is compared with the threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value (generally 255).

In Otsu Thresholding, a value of the threshold isn't chosen but is determined automatically.

6.1.12 Edge Detection

Syntax: cv2.Canny(image, T_lower, T_upper, aperture_size, L2Gradient)

Canny Edge filter in OpenCV. Canny() Function in OpenCV is used to detect the edges in an image.

6.1.13 Segmentation using HSV color space

Image segmentation is an image processing task in which the image is segmented or partitioned into multiple regions such that the pixels in the same region share common characteristics.

- Eroding an Image
- Blurring an Image
- Create Border around Images
- Erosion and Dilation of images
- Filter Color with OpenCV
- Visualizing image in different color spaces
- Find Co-ordinates of Contours
- Bilateral Filtering
- Image Inpainting using OpenCV
- Intensity Transformation Operations on Images
- Image Registration
- Background subtraction
- Background Subtraction in an Image using Concept of Running Average
- Foreground Extraction in an Image using Grabcut Algorithm
- Morphological Operations in Image Processing (Opening)
- Morphological Operations in Image Processing (Closing)
- Morphological Operations in Image Processing (Gradient)
- Image segmentation using Morphological operations
- Image Translation

- Image Pyramid

7. IMPLEMENTATION

This chapter is about the background knowledge of our work, involving the basic concepts of deep learning and computer vision. Transfer learning 7.1 is used for pre-trained initialization in the experiment. Convolutional Neural Network (CNN) 7.2 Image Classification is applied to the Sugarcane Disease Dataset for our goals. Training Strategy 7.3 is the mathematical explanation of our experiment details.

7.1 Transfer Learning

Transfer learning is a powerful technique in deep learning that addresses the issues of model training and data dependency. With transfer learning, a model in a target domain can leverage pre-trained model parameters from another related task, serving as weight initialization for the target model. This process allows the model to transfer knowledge gained from one domain to another, reducing the need to train the target model from scratch.

After the pre-trained model parameters are initialized, the target model typically only needs to train its last layer or specific parts of the network to achieve convergence and improve performance. This fine-tuning step helps the model adapt to the specific requirements and tasks of the target domain. As a result, transfer learning significantly reduces training time and enhances the model's generality, making it applicable to various tasks.

In addition to mitigating the need for extensive training data, transfer learning also alleviates the dependency of deep learning models on large-scale data. Traditional deep learning models often require substantial amounts of data to understand underlying patterns. However, transfer learning enables the use of pre-trained models on related tasks, even if the training data and test data are not identically distributed. This capability makes transfer learning popular and widely used in deep learning, particularly in domains with limited training data availability.

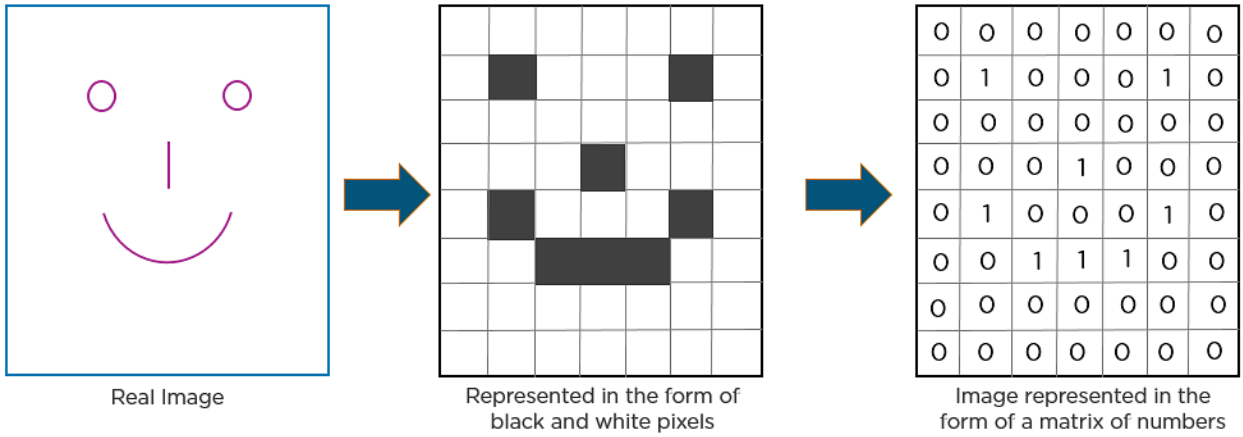
Moreover, transfer learning is particularly beneficial when dealing with resource limitations. Re-implementing state-of-the-art models trained with massive data and multiple GPUs can be challenging for individuals with limited computing resources. In contrast, transfer learning allows for the utilization of pre-trained models, offering better performance with faster training speed.

In this work, transfer learning is applied as a pre-trained initialization for the models, utilizing pre-trained weight parameters from the ImageNet dataset. This approach is a crucial aspect of the controlled experiments conducted, addressing the concerns of training from scratch and data dependency.

7.2 Convolutional Neural Network

CNN [12] is a classic Artificial Neural Network (ANN), specially designed for image tasks and understanding. Nowadays, CNN is the mainstream in the domain of computer vision, commonly used for many tasks such as image classification, object detection, image segmentation, and so on.

Here is an example to depict how CNN recognizes an image:



As you can see from the above diagram, only those values are lit that have a value of 1.

I) Architecture

An instance of CNN structure for classification is depicted in Figure 2.1.

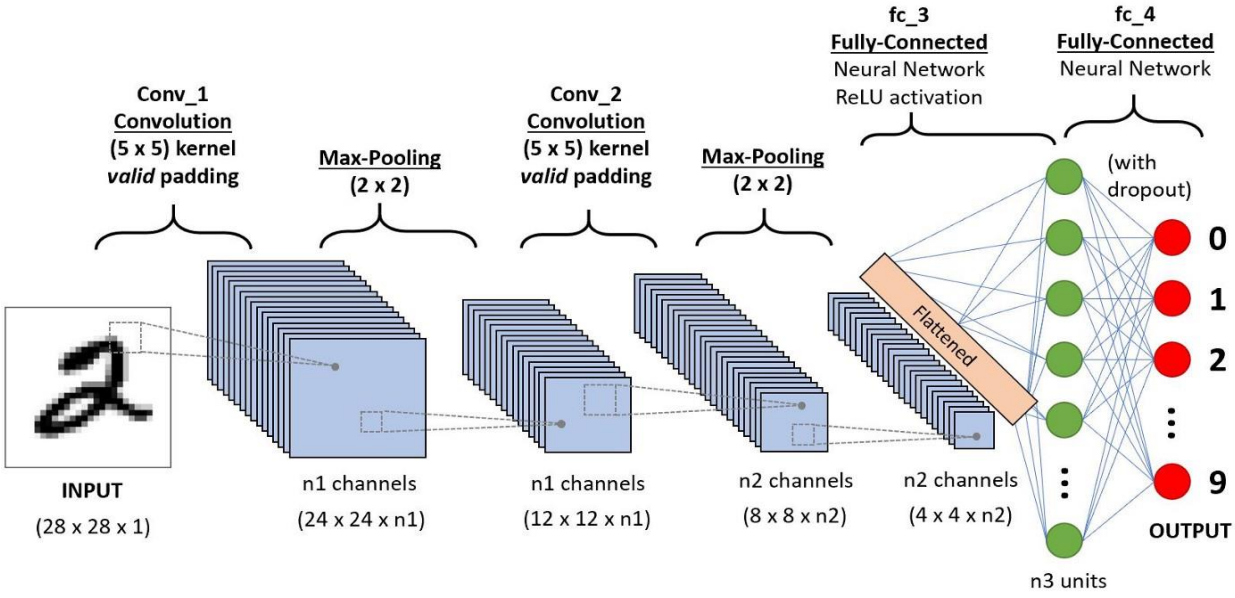
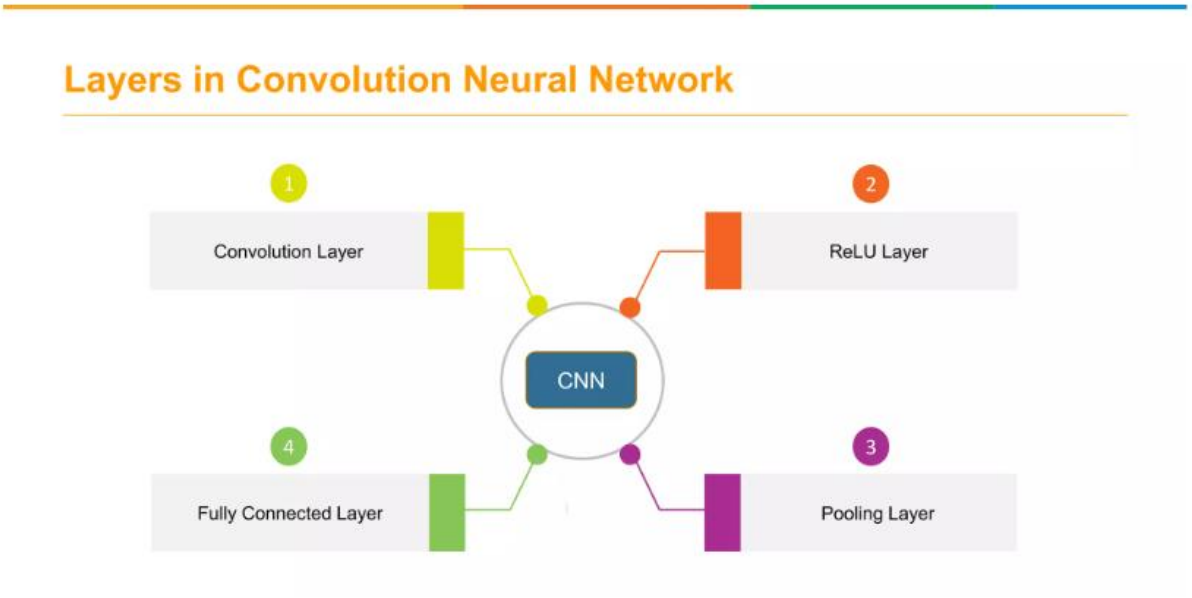


Figure 7.1: A CNN sequence for handwritten digits classification. [12]

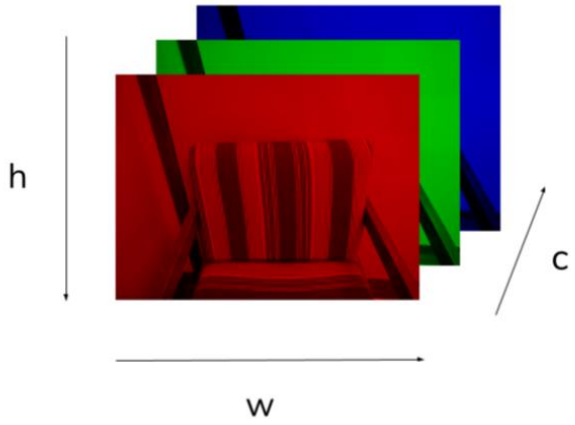
The core component of CNN includes three kinds of layers.

1. convolutional layers
2. pooling layers
3. fully connected layers



II) Forword propagation

- An image is of shape (h, w, c) where:
  - h: image height.
  - w: image width.
  - c: channel.
- Here is an RGB image (3 channels):



1) Convolutional Layers

The convolutional layer is a fundamental component in convolutional neural networks (CNNs) and plays a crucial role in image processing tasks. It is responsible for applying the convolutional operation using learnable kernels or filters, which simulate the receptive field of human visual mechanisms.
Typically, the input to a convolutional layer is an image matrix with RGB channels. The convolutional kernel "slides" spatially over the input from the previous layers, performing dot products between the kernel and specific regions of the input. These dot products are computed by stacking the values across the RGB channels, resulting in the generation of a feature map that is then passed to the next layer.
The beauty of convolutional layers lies in several key properties. Firstly, they achieve parameter sharing among various regions of the input, meaning the same kernel is applied to different parts of the input, enabling the network to learn patterns more efficiently. Secondly, convolutional layers exhibit translation invariance, where they can detect features regardless of their spatial position in the image. Thirdly, sparse interactions are facilitated between layers, allowing the network to process data more efficiently while maintaining useful information.
Through iterative optimization of a targeted loss function during training, convolutional layers can learn representative features from images. The initial layers tend to extract low-level features such as edges and corners, while the subsequent layers combine information from previous layers to capture more complex, high-level semantic features.
By effectively learning and combining these hierarchical features, convolutional neural networks can achieve remarkable performance in image-related tasks, such as object recognition, image classification, and image segmentation. The strength of CNNs lies in their ability to extract and process relevant information hierarchically, making them a powerful tool for a wide range of computer vision applications.

- A convolution operation is defined as follow:

$$Conv(Input, Kernel) = Output$$

Mathematically

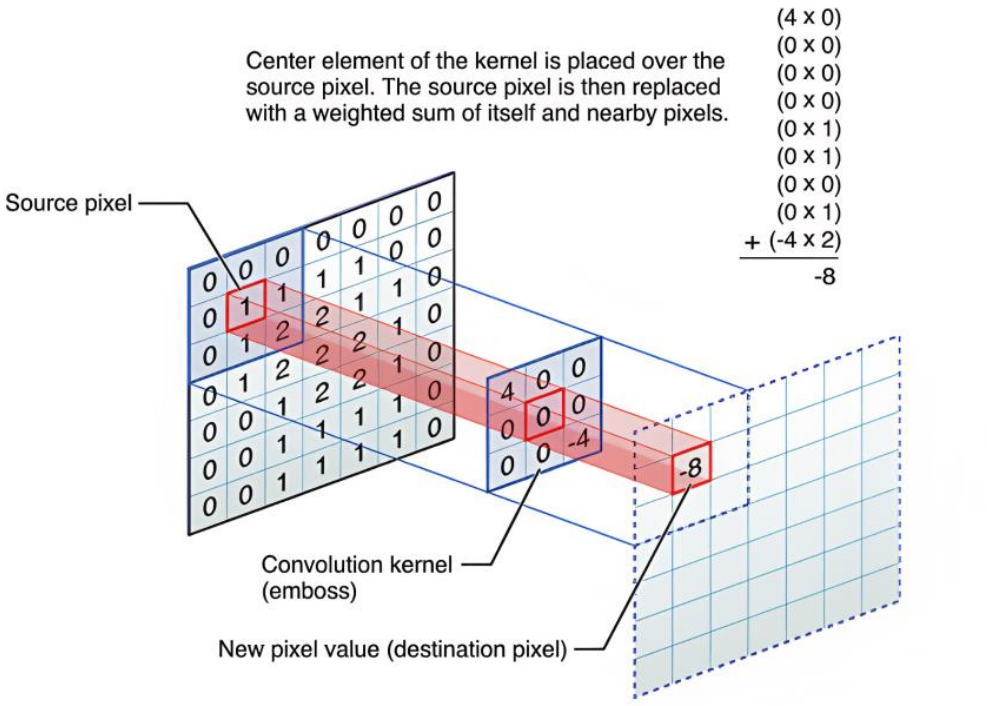
Given a signal $x[n]$, and a kernel (also called a filter) $h[n]$, then the convolution of $x[n]$ with $h[n]$ is written as $y[n] = (x \star h)[n]$, and is computed via a sliding dot-product, mathematically given by:

$$y[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m]$$

The above is for one-dimensional signals, but the same can be said for images, which are just two-dimensional signals. In that case, the equation becomes:

$$I_{new}[r,c] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} I_{old}[u,v]\ k[r-u, c-v]$$

Pictorially, this is what is happening:



considered has a single channel. But in reality, most images have 3 channels (RGB). Hence the kernel also needs to have 3 channels. Such a convolution of two rank 3 tensors results in a rank 2 tensor.

- In order to perform a convolution operation, both Input and Kernel must have the exact same number of channels.
- During the convolution operation, the following formula is used to get the output shape.

$$O = \left\lfloor \frac{I + 2p - K}{s} + 1 \right\rfloor$$

- O: Output shape
- I: Input shape
- p: padding
- K: Kernel shape
- s: stride
- ⌊. . .⌋: floor function.

Now that we know where the output shape comes from, let's see how the content of the output image is generated. During the convolution operation, the kernel is sliding over the whole input. In our following example, we perform a convolution between a (5,5,3) input and 1 kernel of size (3,3,3) to get an (3,3,1) image. At each slide, we perform element-wise multiplication and sum everything to get a single value.

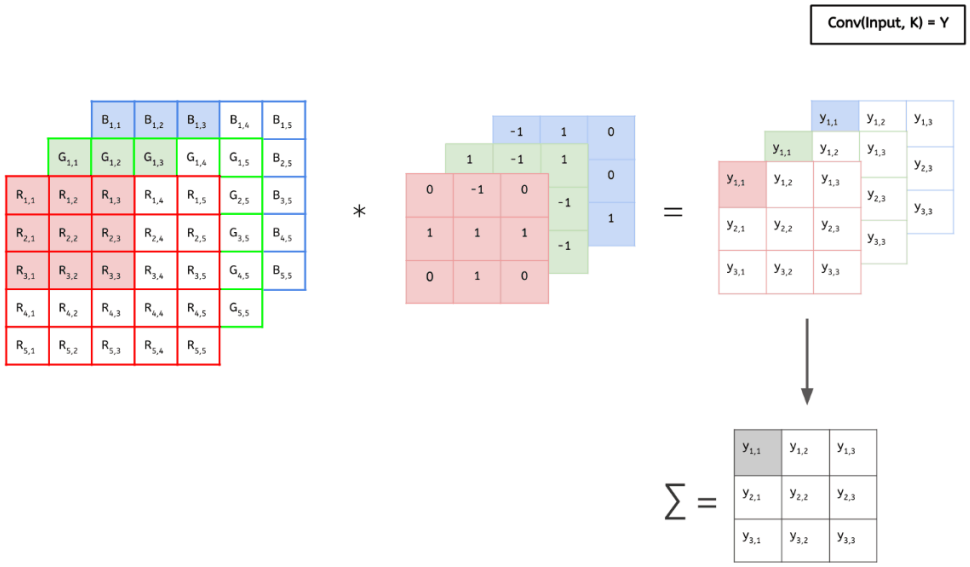
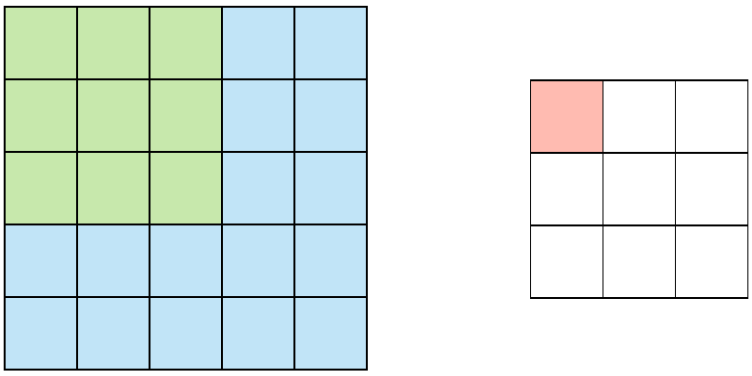


Figure 7.2: Convolution of a 3-channel image with a 3x3x3 kernel [11]

Stride

Stride denotes how many steps we are moving in each step in convolution. By default, it is one.

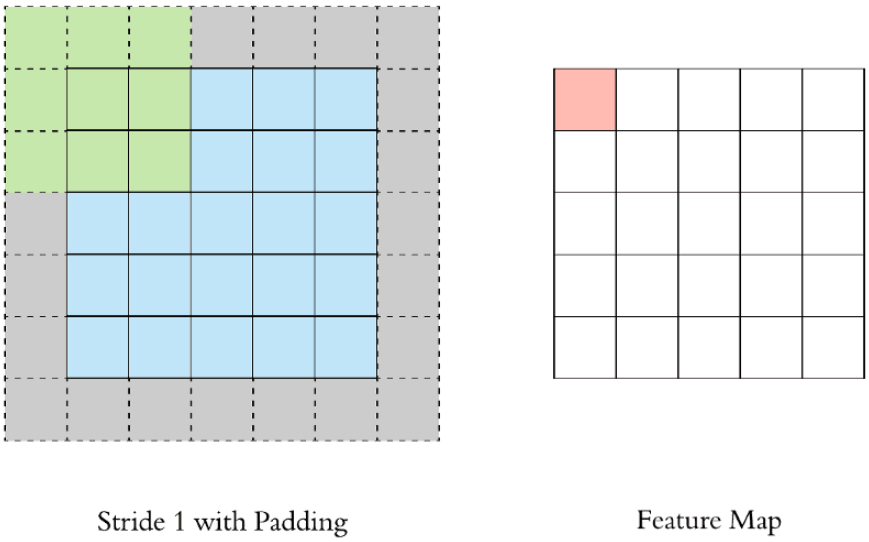


Stride 1 Feature Map

Convolution with Stride 1

Padding

We can observe that the size of output is smaller than input. To maintain the dimension of output as in input , we use padding. Padding is a process of adding zeros to the input matrix symmetrically. In the following example, the extra grey blocks denote the padding. It is used to make the dimensions of output same as input.



Stride 1 with Padding 1

Activation Function (ReLU)

Just like any other Neural Network, we use an activation function to make our output non-linear. In the case of a Convolutional Neural Network, the output of the convolution will be passed through the activation function. This could be the ReLU activation function.

Rectified Linear Units

Most of the deep learning networks use rectified linear units (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. ReLUs' machinery is more like a real neuron in your body.

$$f(x) = max(x, 0)$$

Description:

Figure 7.5 shows ReLU (and non-linear activation functions in general) will introduce non-linear properties in a neural network that enables it to learn more complex arbitrary structures in the inputs. Without activation functions **between the layers**, your neural network will simply be a linear function, regardless of the number of layers it has.

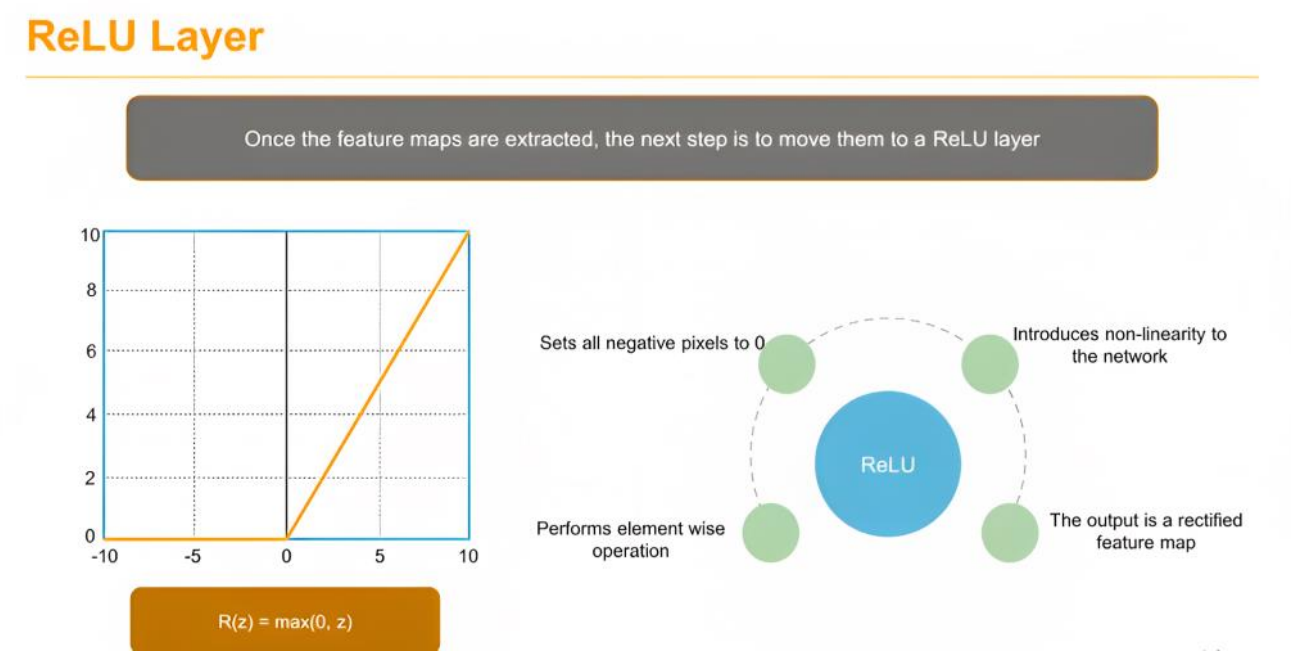


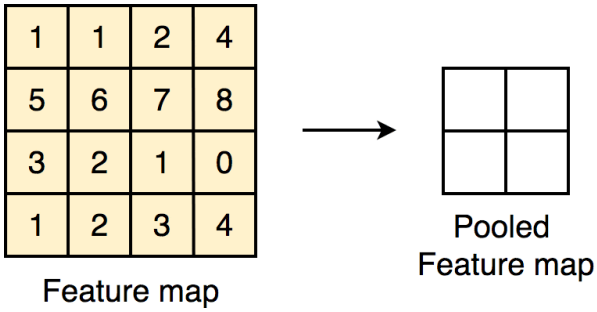
Figure 7.5: RELU Layer

- If we perform a convolution between an (224,224,3) input and 64 kernels of shape (3,3,3) with padding of 1 and stride of 1 using ReLU as activation function. We will have to repeat the convolution operation 64 times on each of the (3,3,3) kernel.
- This will result in an output of shape (224,224,64).
- Here is an implementation of what we have seen so far.

```
model = Sequential()
model.add(Conv2D(input_shape=(224,224,3),
                 filters=64,
                 kernel_size=(3,3),
                 padding="same",
                 activation="relu"))
```

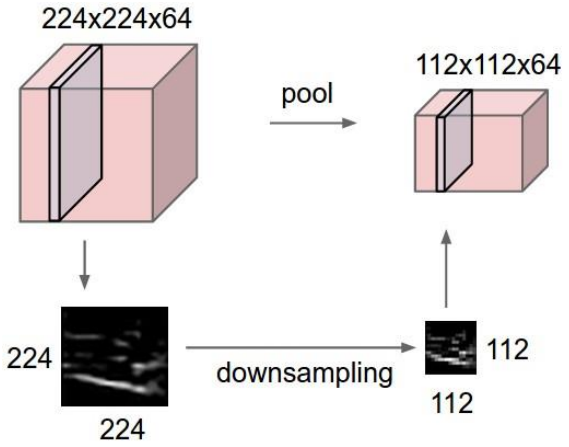
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792
```

2) Pooling Layer



The pooling layer is designed for down-sampling operation between successive convolutional layers, The common pooling strategies contain max pooling and average pooling, among which max pooling is to take the maximum value of the area covered by the kernel, while average pooling is to set the mean value of that.

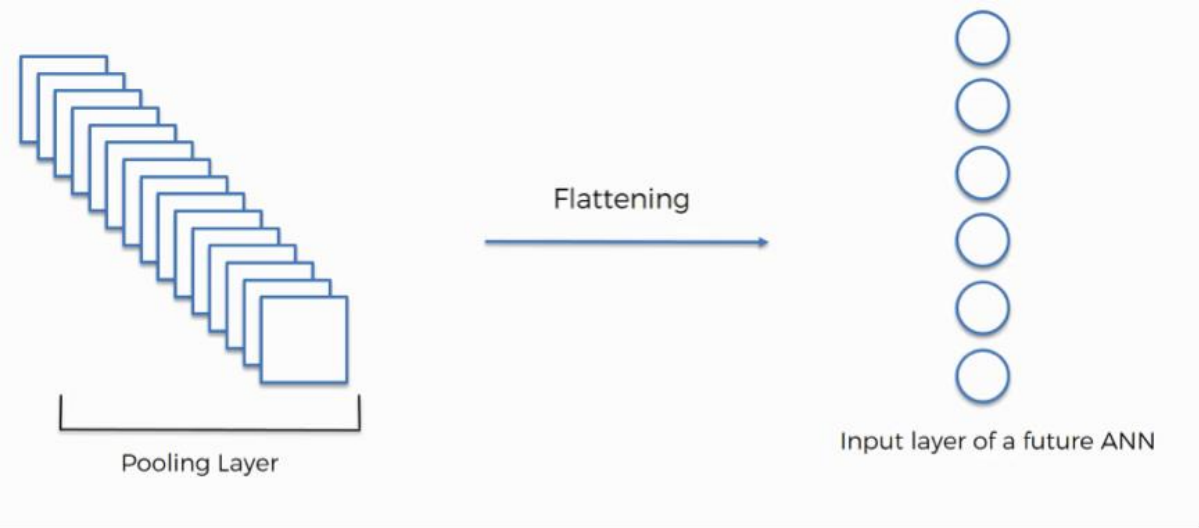
Pooling layer is used to reduce the spatial volume of input image after convolution. It is used between two convolution layers by decreasing the number of model's parameters and making the model training more feasible. Besides, it can learn dominant features from irrelevant positions, which can be considered the extension of the receptive field. If we apply FC after Convo layer without applying pooling or max pooling, then it will be computationally expensive, and we don't want it. So, max pooling is the only way to reduce the spatial volume of input image. In our model, we have applied max pooling in single depth slice (using 2 x 2 filter) with Stride of 2. You can observe the (224 x 224) dimension input is reduced to (112 x 112) dimension.

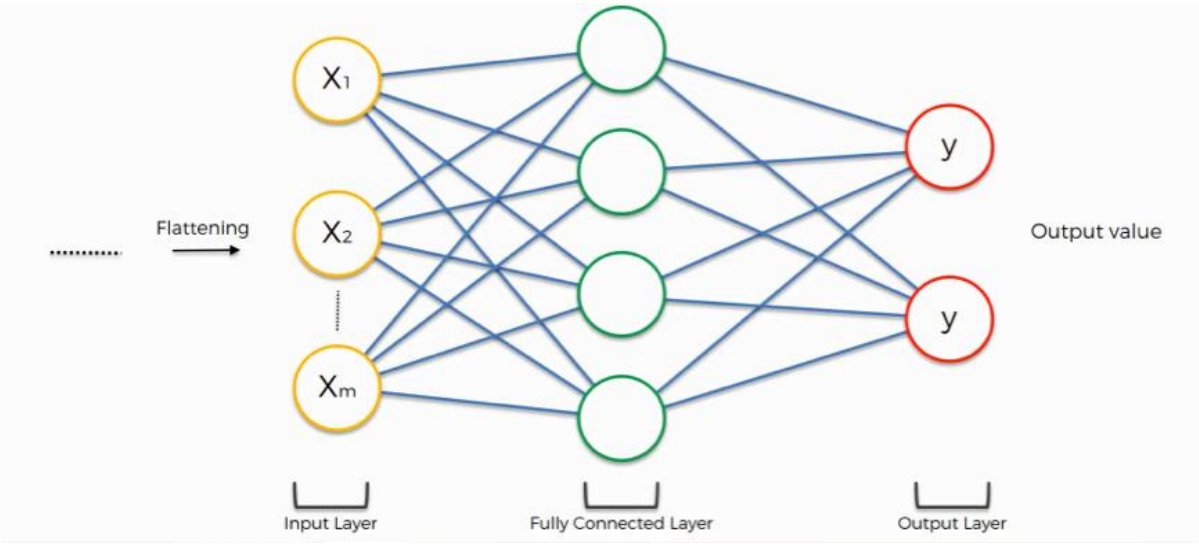



```
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
```

```
block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0
```

Fully Connected Layer

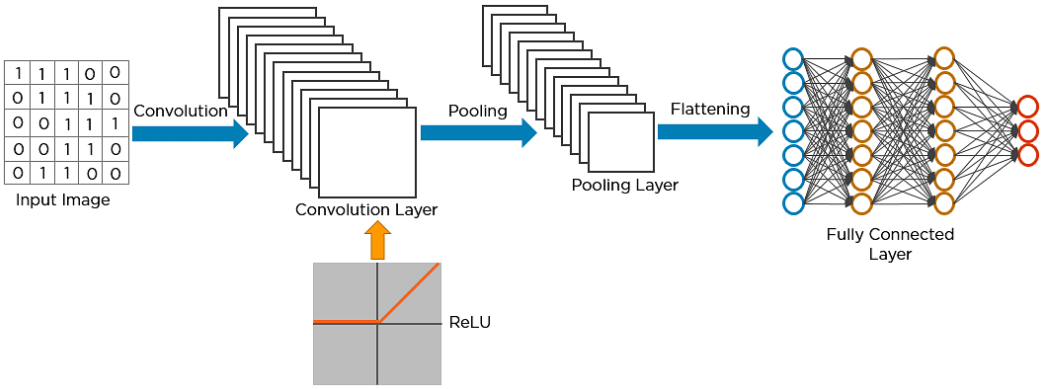




To go from an output of convolutional/pooling layers to a fully connected layer part, we must flatten the output. With the flattened feature map as input, fully connected layers play the role of classifier in the traditional classification, aiming to predict outputs by the feature map extracted by the previous layers. Indeed, the output form is up to the model architecture and targeted loss function.

- For example, the last convolutional layer gives an output of shape (5,5,16).
- After flattening it, we get a (5x5x16) = (400).
- Then, we just perform a weighted sum.

[8] [9] fully connected Layer



[10] complete architecture of CNN

Dropout (Regularization)

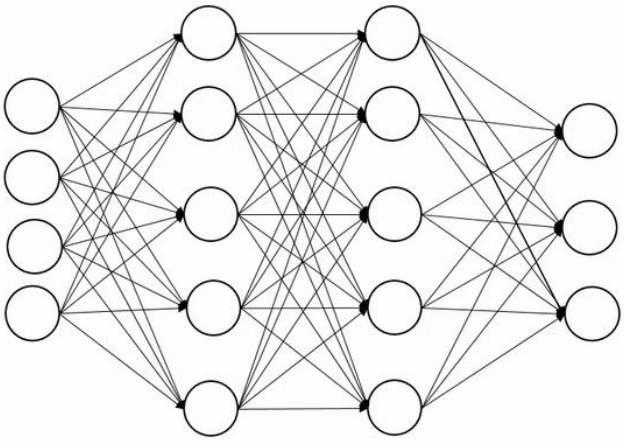
The Dropout layer serves as a mask that randomly deactivates some neurons' contributions to the next layer while preserving others. It is a valuable component in training Convolutional Neural Networks (CNNs) as it helps prevent overfitting on the training data.

When Dropout layers are not included, the initial batch of training samples can have a disproportionate impact on the learning process. This could hinder the learning of features that only appear in later samples or batches, resulting in a less generalized model.

By introducing a Dropout layer with a dropout rate of 0.20 (which implies 20% of input units being dropped), the model's accuracy has increased from 80% to 86.1%. This improvement indicates that the Dropout layer effectively helps reduce overfitting, leading to better generalization and performance on unseen data.

Why will dropouts help with overfitting?

- It can’t rely on one input as it might be randomly dropped out.
- Neurons will not learn redundant details of inputs.



Source: mohcinemadkour.github.io

Batch Normalization

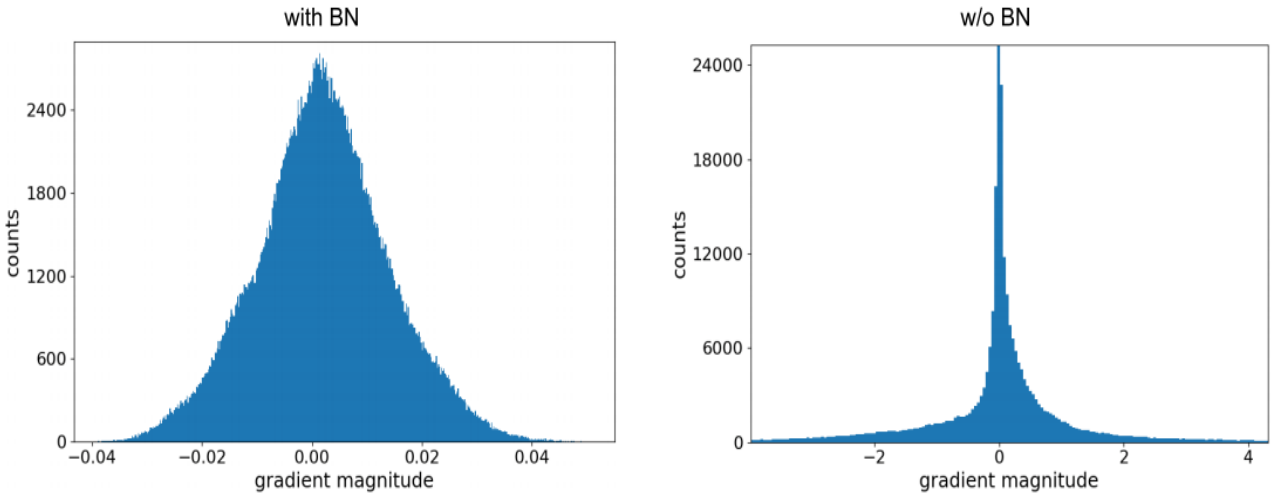
A recently developed technique by Ioffe and Szegedy called Batch Normalization alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. The core observation is that this is possible because normalization is a simple differentiable operation. In the implementation, applying this technique usually amounts to inserting the BatchNorm layer immediately after fully connected layers (or convolutional layers), and before activations.

In practice networks that use Batch Normalization are significantly more robust to bad

Task	Last layer activation function
Binary classification	Sigmoid
Multiclass single-class classification	Softmax
Multiclass multiclass classification	Sigmoid

initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network but integrated into the network itself in a differentiable manner.

The effect of BN is reflected clearly in the distribution of the gradients for the same set of parameters as shown below.



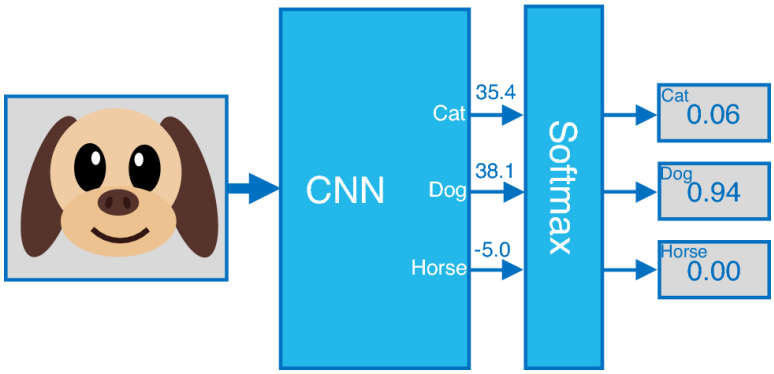
Histograms over the gradients at initialization for (midpoint) layer 55 of a network with BN (left) and without (right). For the unnormalized network, the gradients are distributed with heavy tails, whereas for the normalized networks the gradients are concentrated around the mean.[11]

Last layer activation function

The activation function applied to the last fully connected layer is usually different from the others. An appropriate activation function needs to be selected according to each task. An activation function applied to the multiclass classification task is a softmax function which normalizes output real values from the last fully connected layer to target class probabilities, where each value ranges between 0 and 1 and all values sum to 1. Typical choices of the last layer activation function for various types of tasks are summarized in Table 2.

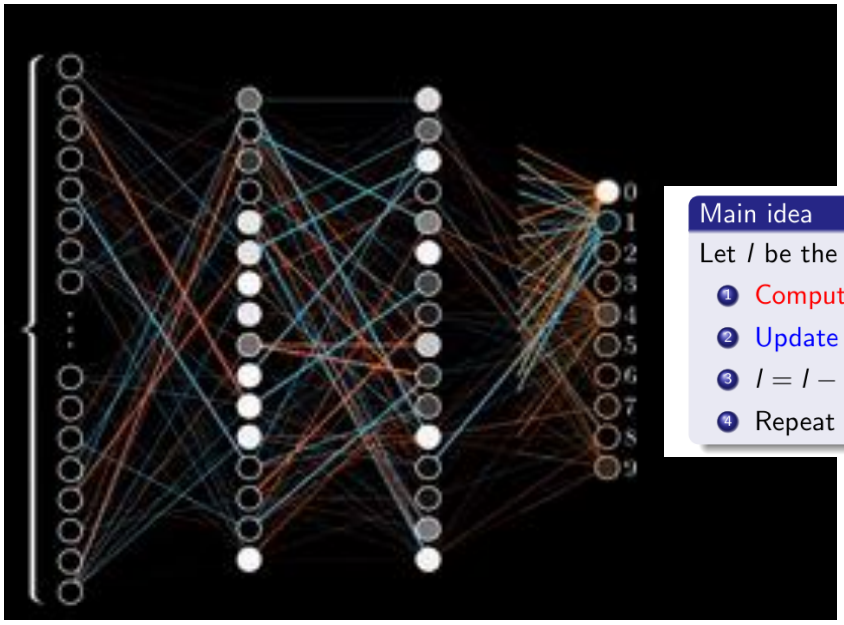
Softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



III) Backward Propagation

Here comes the tricky part. Most of the tutorials I have read so far only say that the backward propagation is the same as in a MLP (which is true). However, we will see that it's not that straightforward to implement, especially at the convolution layer.



Main idea

Let l be the last layer.

1. Compute error at layer l.
2. Update weights at layer l. (if layer $\neq$ pooling layer)
3. $l = l - 1$
4. Repeat

1) Fully Connected Layer

- To compute the loss gradient, we first need to compute the errors δ.
- The error δ is computed differently when you are at:
  - The last layer L of the network (softmax level).
  - Every other layer l.

- At the last layer L, the formula to compute the error is:

$$\delta(L) = (a(L) - y)$$

  - δ(L): error at last layer.
  - a(L): activation function output at last layer.
  - y: ground truth label.

- At every other layer I, the formula to compute the error is:

$$\boxed{\delta^{(l)} = ((\Theta^{(l+1)})^T \delta^{(l+1)}) \ .* \ a'(z^{(l)})}$$

  - $\delta(l)$: error at layer l.
  - $\Theta(l+1)$: Weight matrix at layer $l+1$.
  - $\delta(l+1)$: error at layer $l+1$.
  - $a'(z(l))$: derivative of activation function at layer l.

- We can then compute the loss gradient at each layer with the following formula:

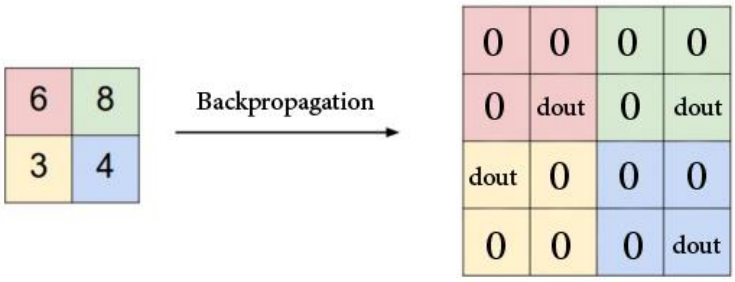
$$\boxed{\frac{\partial \mathcal{L}}{\partial \Theta_{j,k}^{(l)}} = \frac{1}{m} \sum_{t=1}^{m} a_j^{(t)(l)} \delta_k^{(t)(l+1)}}$$

- This will then be used to update your weight.

$$\boxed{\Theta^{(l)} \leftarrow \Theta^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial \Theta^{(l)}}}$$

2) Pooling Layer

- During forward propagation, we were selecting the max value from the input within the pooling window size.
- During backward propagation, we need to proportionally back-propagate the error to the input.
- Remember, no weights gradient is computed here! We only compute the layer gradient.



[5]

3) Convolutional Layer

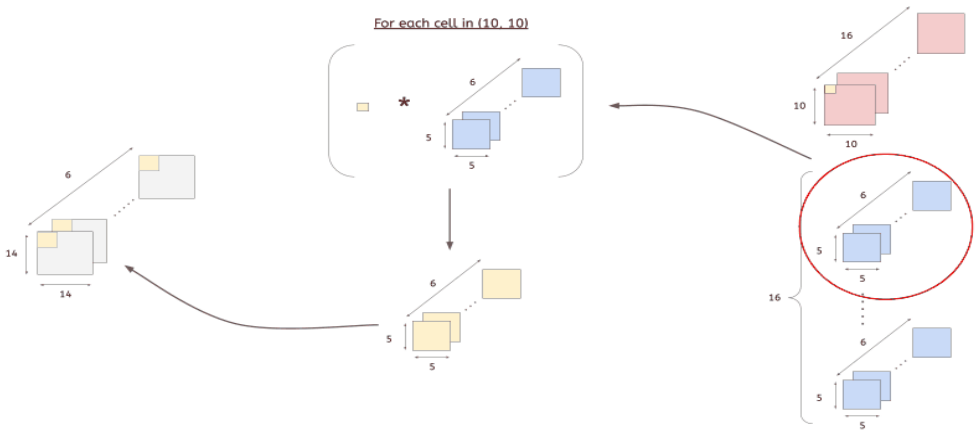
- The idea of backward propagation is to back-propagate the gradient from lower layers to upper layers.
- In order to perform backward propagation in the example above, we have to do 2 things:
  - a) Compute the layer gradients $\frac{\partial L}{\partial I}$ at layer (H, W, K)
  - b) Compute the kernel gradients $\frac{\partial L}{\partial K}$ at layer (H, W, K)
- The optimizer (SGD, Adam, RMSprop) will then update the kernels value.

a) Compute Layer gradient

$$\boxed{\frac{\partial L}{\partial I} = Conv\,(K, \frac{\partial L}{\partial O})}$$

- $\frac{\partial L}{\partial I}$: Input gradient.
- K: Kernels.
- $\frac{\partial L}{\partial O}$:Output gradient.

In the previous problem, we encountered a channels mismatch issue during convolution, where the number of channels in the input (6) did not match the number of kernels (16). The forward propagation process involved performing convolutions between the input tensor (14x14x6) and 16 kernels of size (5x5x6), resulting in an output tensor of size (10x10x16).
Each feature map in the output was generated by convolving the input with each respective kernel. During backward propagation, the gradients in each feature map of the output tensor need to be back-propagated to each corresponding filter.
To achieve this, the gradients from each feature map in the (10x10) output tensor need to be "broadcasted" to the corresponding "slides" made during forward propagation over the input. In other words, the gradients are propagated back to each filter that was used during the forward pass.
By adding the gradients from each feature map to their respective filters, the backpropagation process ensures that the gradients are correctly updated for each kernel, allowing the network to learn and adjust the kernels during training to minimize the loss and improve performance. This iterative process of forward and backward propagation, along with the kernel updates, continues throughout the training process until the network converges to a set of optimized kernels and achieves the desired performance on the training data.

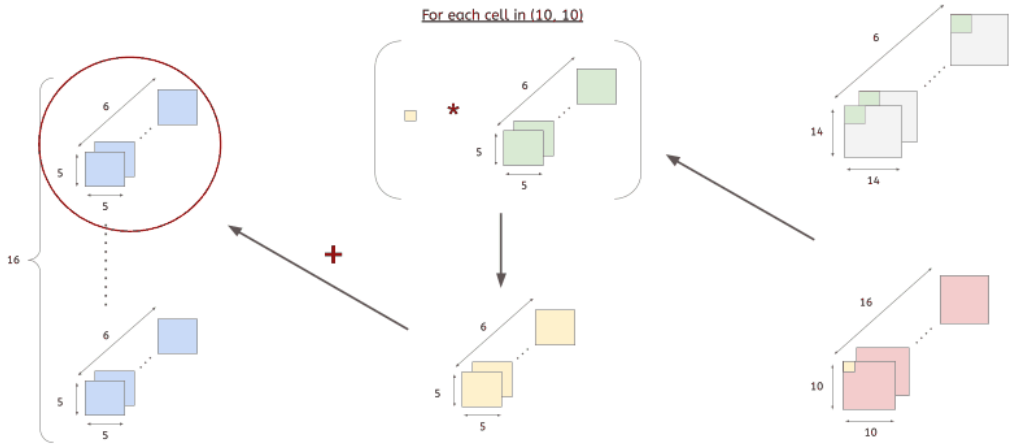


b) Compute Kernel gradient

$$\frac{\partial L}{\partial K} = Conv\left(I, \frac{\partial L}{\partial O}\right)$$

- $\frac{\partial L}{\partial K}$: Kernels gradient.
- I: Input image.
- $\frac{\partial L}{\partial O}$: Output gradient.

In the previous problem, we encountered a channels mismatch issue during convolution, where the number of channels in the input (6) did not match the number of kernels (16). The forward propagation process involved performing convolutions between the input tensor (14x14x6) and 16



kernels of size (5x5x6), resulting in an output tensor of size (10x10x16).
Each feature map in the output was generated by convolving the input with each respective kernel. During backward propagation, the gradients in each feature map of the output tensor need to be back-propagated to each corresponding filter.
To achieve this, the gradients from each feature map in the (10x10) output tensor need to be "broadcasted" to the corresponding "slides" made during forward propagation over the input. In other words, the gradients are propagated back to each filter that was used during the forward pass.
By adding the gradients from each feature map to their respective filters, the backpropagation process ensures that the gradients are correctly updated for each kernel, allowing the network to learn and adjust the kernels during training to minimize the loss and improve performance. This iterative process of forward and backward propagation, along with the kernel updates, continues throughout the training process until the network converges to a set of optimized kernels and achieves the desired performance on the training data.

An overview of a convolutional neural network (CNN) architecture and the training process. A CNN is composed of a stacking of several building blocks: convolution layers, pooling layers (e.g., max pooling), and fully connected (FC) layers. A model's performance under kernels and weights is calculated with a loss function through forward propagation on a training dataset, and learnable parameters, i.e., kernels and weights, are updated according to the loss value through backpropagation with gradient descent optimization algorithm. ReLU, rectified linear unit.

As far now we have learned all the basic and important concept of a Convolutional Neural

Network (CNN)

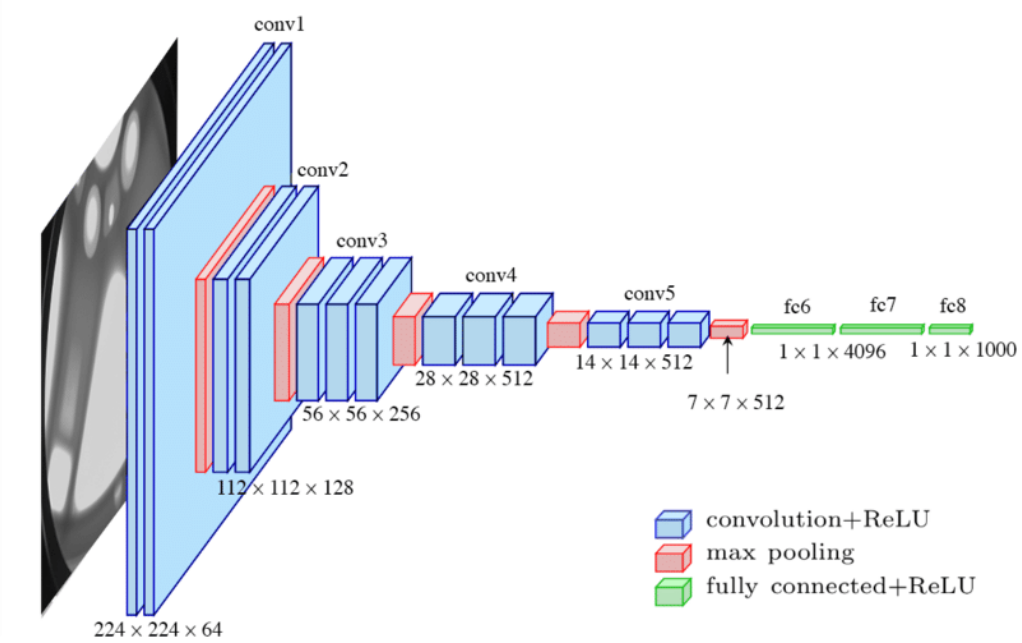
Now take a dive in our final Year Project (Sugarcane disease Detection) that classify the 8 different classes of diseases of Sugarcane through a pretrained CNN Model called VGG16.

VGG16

The VGG16 (Visual Geometry Group 16) [20] is a widely used convolutional neural network (CNN) architecture for image classification tasks. It was developed by the Visual Geometry Group at the University of Oxford. In this report, we leverage the power of VGG16 to detect diseases in sugarcane plants and classify them in 8 different classes.

7.3 Model Architecture and Overview:

The VGG16 architecture is quite straightforward and very similar to the original convolutional networks. The main idea behind VGG was to make the network deeper by stacking more convolutional layers. And this was made possible by restricting the size of the convolutional windows or kernel to only 3x3 pixels.
The VGG16 architecture consists of a total of 16 layers, including 13 convolutional layers and 3 fully connected layers. The network follows a sequential structure, with each layer performing specific operations to extract and classify features from the input image.



VGG16 Block Diagram (source: neurohive.io)

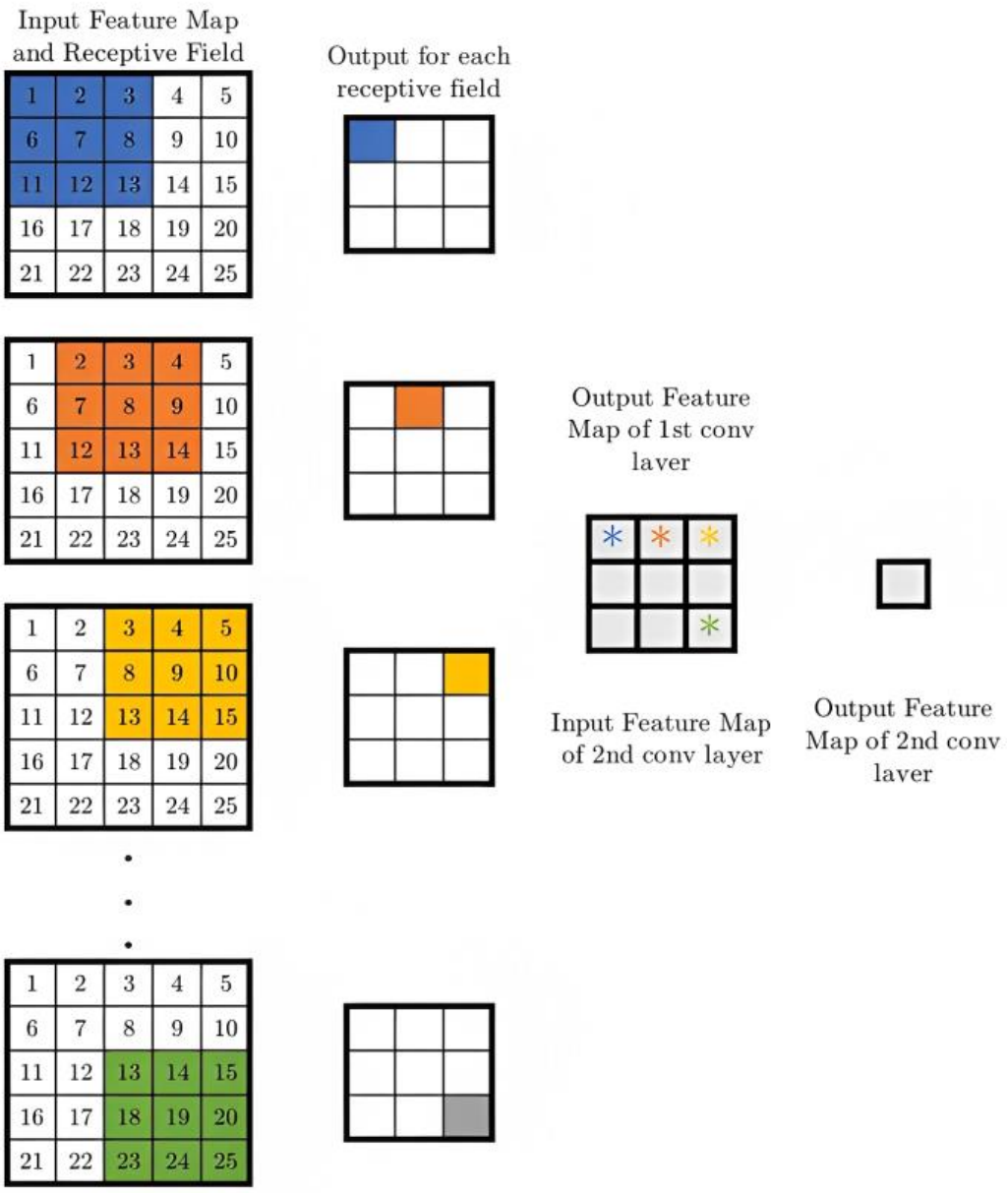
VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

VGG16 has a total of 138 million parameters. The important point to note here is that all the conv kernels are of size 3x3 and maxpool kernels are of size 2x2 with a stride of two.

VGG16 has a total of 138 million parameters. The important point to note here is that all the conv kernels are of size 3x3 and maxpool kernels are of size 2x2 with a stride of two.

How? The idea behind having fixed size kernels is that all the variable size convolutional kernels used in Alexnet (11x11, 5x5, 3x3) can be replicated by making use of multiple 3x3 kernels as building blocks. The replication is in terms of the receptive field covered by the kernels.

Let's consider the following example. Say we have an input layer of size 5x5x1. Implementing a conv layer with a kernel size of 5x5 and stride one will result in an output feature map of 1x1. The same output feature map can be obtained by implementing two 3x3 conv layers with a stride of 1 as shown below.



Now let's look at the number of variables that needed to be trained. For a 5x5 conv layer filter, the number of variables is 25. On the other hand, two conv layers of kernel size 3x3 have a total of 3x3x2=18 variables (a reduction of 28%).

Similarly, the effect of one 7x7 (11x11) conv layer can be achieved by implementing three (five) 3x3 conv layers with a stride of one. This reduces the number of trainable variables by 44.9% (62.8%). A reduced number of trainable variables means faster learning and more robust to over-fitting.

2. Input Layer:

The input to the VGG16 model is an RGB image of fixed size (typically 224x224 pixels). The image is fed into the network as a tensor.

```
model = Sequential()
model.add(Conv2D(input_shape=(224,224,3),
                 filters=64,
                 kernel_size=(3,3),
                 padding="same",
                 activation="relu"))
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0
```

3. Convolutional Layers:

The convolutional layers in VGG16 are responsible for learning various image features at different scales. These layers use small receptive fields (3x3 filters) with a stride of 1 pixel and employ a rectified linear unit (ReLU) activation function.

```
model.add(Conv2D(filters=64,
                 kernel_size=(3,3),
                 padding="same",
                 activation="relu"))
```

```
 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792
```

4. Max Pooling Layers:

After every two convolutional layers, a max pooling layer is inserted. Max pooling reduces the spatial dimensions of the feature maps, helping to capture translation invariance and reduce computational complexity.

```
model.add(MaxPool2D(pool_size=(2,2),
                    strides=(2,2)))
```

```
 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0
```

5. Fully Connected Layers:

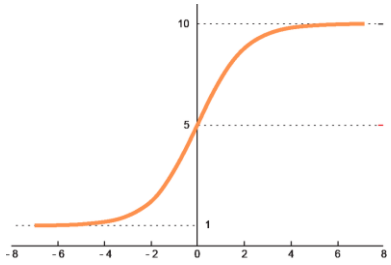
The last three layers of VGG16 are fully connected layers that perform classification based on the learned features. These layers have a large number of parameters, allowing the model to capture complex relationships between features.

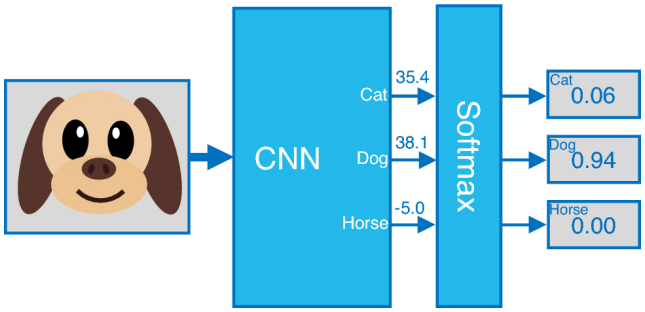
```
model.add(Flatten())
model.add(Dense(units=4096,activation="relu"))
```

6. Softmax Activation:

The final layer of the VGG16 model employs the softmax activation function, which produces a probability distribution over the classes. It assigns a probability to each class, indicating the likelihood of the input image belonging to that class.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



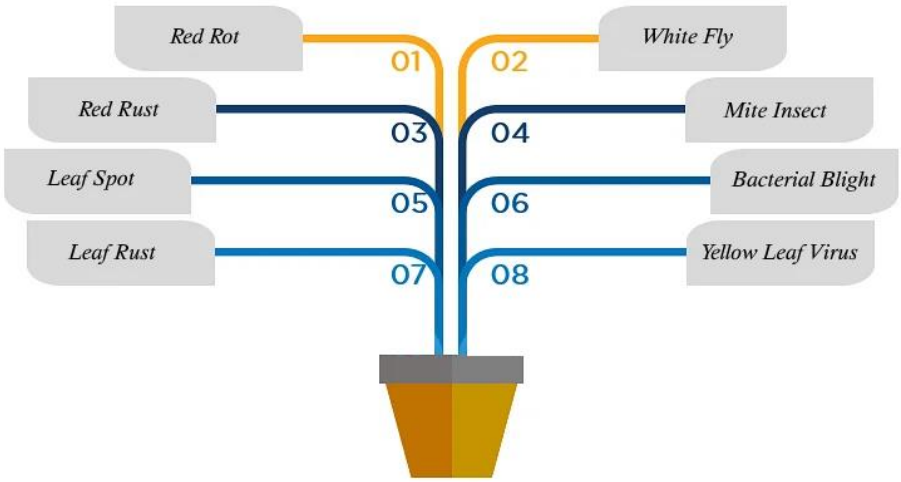


7.4 Usecase implementation using CNN (VGG16)

The images Dataset

We'll be using the Sugarcane Diseases dataset that we collected under the supervision of PARC's expert team at the fields of Sugarcane from different areas of Pakistan For Advanced Research for classifying Sugarcane Diseases across 8 categories using fine-tuned pretrained VVG16 convolutional Neural Network (CNN) model.

Initially the dataset contains 1000 images of disease then we augment the data to 2800 images that each class contain the 350 images of infected sugarcane leaf.



7.5 Preparing the Training and Testing Data

7.5.1.1.1 Load the Dataset

```
import cv2
import glob
import os
import numpy as np

SIZE = 224  #Resize images

#Capture training data and labels into respective lists
dataset = []
labels = []

#iterating over the directory using glob
for directory_path in glob.glob("/content/drive/MyDrive/FYP Datasets/Balance Dataset/*"):
    label =  os.path.basename(directory_path)
    print(label)
    for img_path in glob.glob(os.path.join(directory_path, "*.jpg")):
        # print(img_path)

        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (SIZE, SIZE),interpolation=cv2.INTER_AREA)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        dataset.append(img)
        labels.append(label)

#Convert lists to arrays
dataset = np.array(dataset)
labels = np.array(labels)

print('\nThe shape of images dataset : ', dataset.shape)
print('The shape of images labels : ', labels.shape)
```

```
Bacterial Blight
Yellow Leaf Virus
Mite Insect
Red Rust
Leaf Spot
White Fly
Red Rot
Leaf Rust

The shape of images dataset :  (2800, 224, 224, 3)
The shape of images labels :  (2800,)
```

7.5.1.1.2 Label and One Hot Encoding

```
#Encode labels from text to integers.
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
le.fit(labels)
labels_encoded = le.transform(labels)

#One hot encode integer labels for neural network.
from keras.utils import to_categorical
labels_one_hot = to_categorical(labels_encoded)
labels_one_hot
```

```
array([[0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

7.5.1.1.3 Split the Dataset

```
#====== Split the dataset ==================
#
# I split the dataset into training and testing dataset.
# 1. Training data: 80%
# 2. Testing data: 20%

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(dataset,labels_one_hot, test_size = 0.20, random_state = 42)

print(f'{len(X_train)} images for training.')
print(f'{len(X_test)} images for testng.')
```

```
2240 images for training.
560 images for testng.
```

7.5.1.1.4 Rescaling the Data

```
# Normalize the values of image from 0-255 to 0-1
X train, X test = X train/255.0, X test/255.0
```

```
[[[0.69803922, 0.72156863, 0.67843137],
  [0.69019608, 0.71372549, 0.67058824],
  [0.67058824, 0.68627451, 0.64313725],
  ...,
  [0.44705882, 0.59607843, 0.57254902],
  [0.44705882, 0.59607843, 0.57254902],
  [0.44705882, 0.59607843, 0.57254902]]],
```

7.5.2 The VGG16 Model

7.5.2.1.1 Load VGG16b Model without classifier/fully connected layers.

- Import only the convolutional part of VGG16, by setting the include_top parameter to False because we are adding our own classifier in the last fully connected layer and the parameter with respect to our task.

- We are using the pretrained weights of ImageNet dataset.

- We are freezing the pre-trained layer of VGG16 that the weights and bias don't get updated during the training.

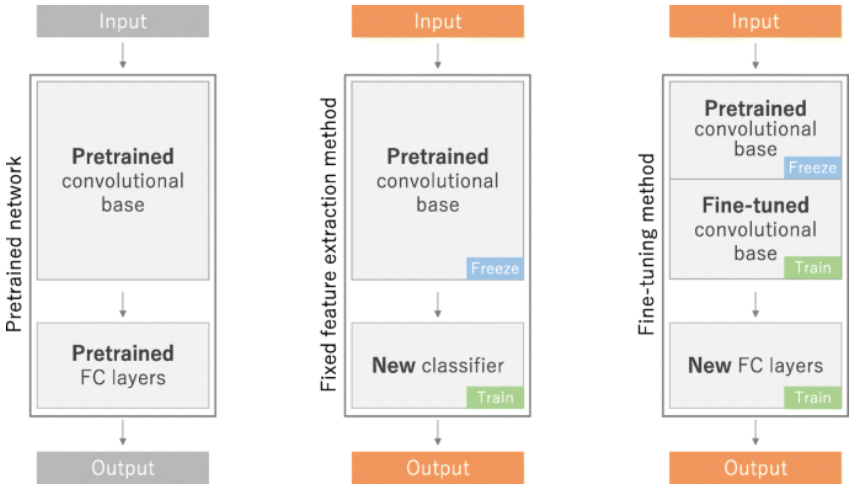
```
#Load model without classifier/fully connected layers

from keras.applications.vgg16 import VGG16

VGG_model = VGG16(weights='imagenet', include_top=False, input_shape=(SIZE, SIZE, 3))

#Make loaded layers as non-trainable. This is important as we want to work with pre-trained weights

for layer in VGG_model.layers:
    layer.trainable = False

VGG_model.summary()
```

```
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

- Note that the Trainable Parameters are 0 due to the freezing of layers.

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

7.5.3 Fine-tuning of VGG16 Model:



In this approach, we employ a strategy called Fine-Tuning. The goal of fine-tuning is to allow a portion of the pre-trained layers to retrain.
In the previous approach, we used the pre-trained layers of VGG16 to extract features. We passed our image dataset through the convolutional layers and weights, outputting the transformed visual features. There was no actual training on these pre-trained layers.
Fine-tuning a Pre-trained Model entails:

1. Bootstrapping a new "top" portion of the model (i.e., Fully Connected and Output layers)
2. Freezing pre-trained convolutional layers
3. Un-freezing the last few pre-trained layers training.

```
#Load model without classifier/fully connected layers

from keras.applications.vgg16 import VGG16

VGG_model = VGG16(weights='imagenet', include_top=False, input_shape=(SIZE, SIZE, 3))

#Make loaded layers as non-trainable. This is important as we want to work with pre-trained weights

fine_tune = 2 # How many layers you want to unfreez from the last

if fine_tune > 0:
        for layer in VGG_model.layers[:-fine_tune]: #The parameters of last 2 layer will be trained.
            layer.trainable = False
else:
    for layer in VGG_model.layers:
        layer.trainable = False

VGG_model.summary()
```

```
block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 2,359,808
Non-trainable params: 12,354,880
_________________________________________________________________
```

After unfreezing the last 2 layers of the model, there are 2,359,808 Trainable Parameters in the convolution part of the model for training. Freezing certain layers of the network and training only the last few layers or specific parts of the network to adapt it to the target task of sugarcane disease detection.

7.5.4 The Top Model

7.5.4.1.1 Flattened the Feature Maps .

```
top_model = VGG_model.output
top_model = Flatten(name="flatten")(top_model)
```

```
flatten (Flatten)           (None, 25088)             0
```

7.5.4.1.2 Fully Connected Layer

```
top_model = Dense(100, activation='relu')(top_model)
```

```
dense (Dense)               (None, 100)               2508900
```

7.5.4.1.3 Batch Normalization Layer

```
top_model = BatchNormalization(axis = -1)(top_model)
```

```
batch_normalization_1 (Batc  (None, 100)              400
hNormalization)
```

7.5.4.1.4 Dropout Layer

```
top_model = Dropout(0.2)(top_model)
```

```
dropout (Dropout)           (None, 100)               0
```

7.5.4.1.5 Output Layer

The output layer contains only 8 neurons because we have to classify 8 classes of Sugarcane Diseases.

```
output_layer = Dense(n_classes, activation='softmax')(top_model)
```

```
 dense_3 (Dense)             (None, 8)                 408

=================================================================
Total params: 17,240,146
Trainable params: 4,884,766
Non-trainable params: 12,355,380
_________________________________________________________________
```

7.5.4.1.6 3 fully connected Layer:

```
# Create a new 'top' of the model (i.e. fully-connected layers).

# This is 'bootstrapping' a new top_model onto the pretrained layers.

n_classes = 8

top_model = VGG_model.output
top_model = Flatten(name="flatten")(top_model)

top_model = Dense(100, activation='relu')(top_model)
top_model = BatchNormalization(axis = -1)(top_model)
top_model = Dropout(0.2)(top_model)

top_model = Dense(100, activation='relu')(top_model)
top_model = BatchNormalization(axis = -1)(top_model)
top_model = Dropout(0.2)(top_model)

top_model = Dense(50, activation='relu')(top_model)
top_model = BatchNormalization(axis = -1)(top_model)
top_model = Dropout(0.2)(top_model)

output_layer = Dense(n_classes, activation='softmax')(top_model)

# Group the convolutional base and new fully-connected layers into a Model object.
model = Model(inputs=VGG_model.input, outputs=output_layer)
```

```
flatten (Flatten)           (None, 25088)             0

dense (Dense)               (None, 100)               2508900

batch_normalization (BatchN  (None, 100)              400
ormalization)

dropout (Dropout)           (None, 100)               0

dense_1 (Dense)             (None, 100)               10100

batch_normalization_1 (Batc  (None, 100)              400
hNormalization)

dropout_1 (Dropout)         (None, 100)               0

dense_2 (Dense)             (None, 50)                5050

batch_normalization_2 (Batc  (None, 50)               200
hNormalization)

dropout_2 (Dropout)         (None, 50)                0

dense_3 (Dense)             (None, 8)                 408
```

7.5.5 Compilation of Model

7.5.5.1.1 Optimizer

```
from keras.optimizers import Adam

optimizer = Adam(learning_rate=0.0001)
```

7.5.5.1.2 Compile Model

```
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

7.5.6 Training of Model

After calculating the loss, the backpropagation algorithm takes over. It propagates the error backward through the network, computing the gradients for each learnable parameter based on their contribution to the overall loss. The gradients provide information on how much and in what direction each parameter should be adjusted to minimize the loss.
With the gradients in hand, an optimization algorithm, such as gradient descent, is employed to update the learnable parameters iteratively. The gradients guide the optimization algorithm to adjust the kernels and weights, moving them towards values that reduce the loss on the training dataset.
This iterative process of forward propagation, loss calculation, backpropagation, and parameter updates continues until the network converges to a set of kernels and weights that result in minimal loss and satisfactory performance on the training data. This trained model can then be used for making accurate predictions on new, unseen data.

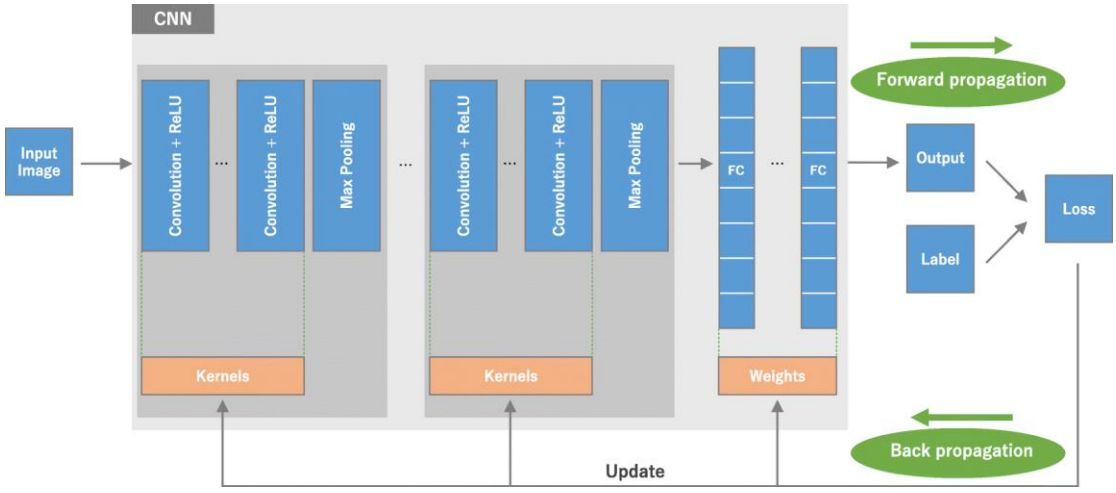


Fig. 1

7.5.6.1.1 Loss function

A loss function, also referred to as a cost function, measures the compatibility between output predictions of the network through forward propagation and given ground truth labels. Commonly used loss function for multiclass classification is cross entropy, in our model we have used categorical_crossentropy. A type of loss function is one of the hyperparameters and needs to be determined according to the given tasks.

```
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

7.5.6.1.2 Gradient descent

Gradient descent is a commonly employed optimization algorithm in machine learning, particularly for updating the learnable parameters such as kernels and weights in a network to minimize the loss function. The gradient of the loss function provides us with the direction in which the function increases most rapidly. By updating the learnable parameters in the opposite direction of the gradient, the algorithm attempts to move towards the optimal parameter values that result in reduced loss.

The step size for each update is controlled by a hyperparameter known as the learning rate. The learning rate influences the size of each step taken during parameter updates, and it plays a critical role in determining the convergence and stability of the optimization process. Properly setting the learning rate is crucial, as a large learning rate may lead to overshooting the optimal values, while a small learning rate may result in slow convergence or getting stuck in local minima. Finding an appropriate learning rate is an important aspect of training a machine learning model effectively. (as shown in Figure 7).

Mathematically, the gradient represents the partial derivative of the loss function concerning each learnable parameter. To update a single parameter, the process can be formulated as follows:

parameter = parameter - learning_rate * gradient

This iterative process continues until the loss function is minimized or reaches a predefined stopping criterion. By following the negative direction of the gradient and adjusting the learnable parameters accordingly, gradient descent helps the neural network converge to the optimal values that result in reduced loss and improved model performance.

$$\boldsymbol{w} := \boldsymbol{w} - \boldsymbol{\alpha} * \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}$$

In the context of updating the learnable parameters (denoted as "w") using gradient descent:
w represents each learnable parameter.

α (alpha) is the learning rate, a crucial hyperparameter that needs to be set before training begins. It controls the step size in each parameter update and affects the convergence and stability of the training process.
L is the loss function, which measures the difference between the predicted output of the neural network and the actual target output.
In practical implementations, due to memory limitations and efficiency concerns, the gradients of the loss function with respect to the parameters are often computed using a subset of the training dataset known as a mini-batch. The mini-batch gradient descent, or stochastic gradient descent (SGD), involves updating the parameters based on the gradients computed from these mini-batches. The size of the mini-batch is also a hyperparameter that needs to be chosen.
Furthermore, there have been various improvements proposed for the basic gradient descent algorithm, and one popular optimizer used in training models is Adam (Adaptive Moment Estimation). Adam optimizer effectively combines the benefits of both the Adagrad and RMSprop optimizers. It adjusts the learning rates of each parameter based on the historical gradients, resulting in faster convergence and better performance during training.
Overall, selecting the appropriate learning rate, mini-batch size, and optimizer is crucial in achieving efficient and effective training of neural network models.

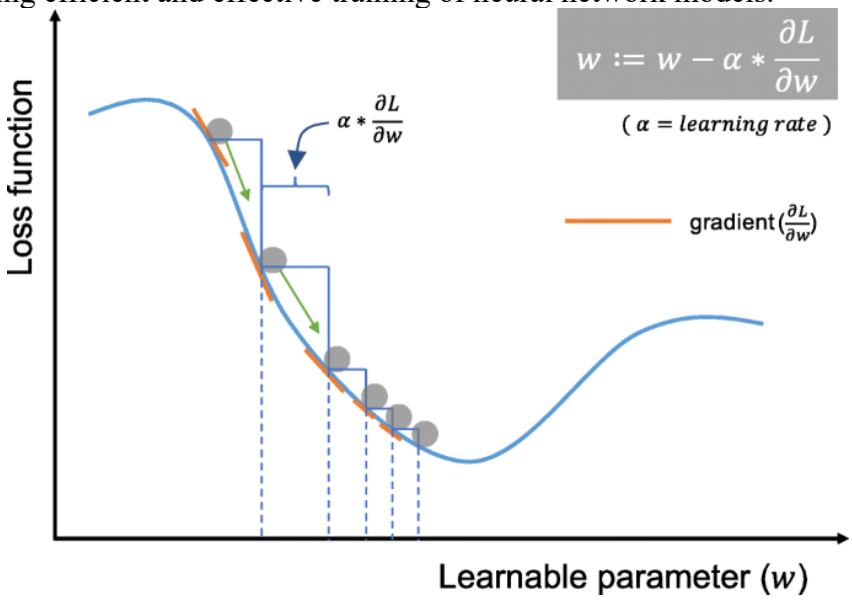


Fig. 7

7.5.6.1.3 Validation Data

The data available for the project is split into three distinct sets: the training set, the validation set, and the test set (as shown in Figure 8). However, there are other variations, such as cross-validation.
The training set is used to train the neural network. During this process, loss values are calculated through forward propagation, and the learnable parameters of the network are updated using backpropagation to optimize the model.
The validation set plays a crucial role in monitoring the model's performance during the training phase. It helps fine-tune hyperparameters and aids in the process of model selection, ensuring the best-performing model is chosen.
The test set has a unique purpose: it is used ideally only once, at the very end of the project. The final model, which has been fine-tuned and selected based on its performance on the training and validation sets, is evaluated on this test set to assess its overall performance and generalization ability. This separation of training, validation, and test sets ensures a reliable evaluation of the model's performance without introducing bias.

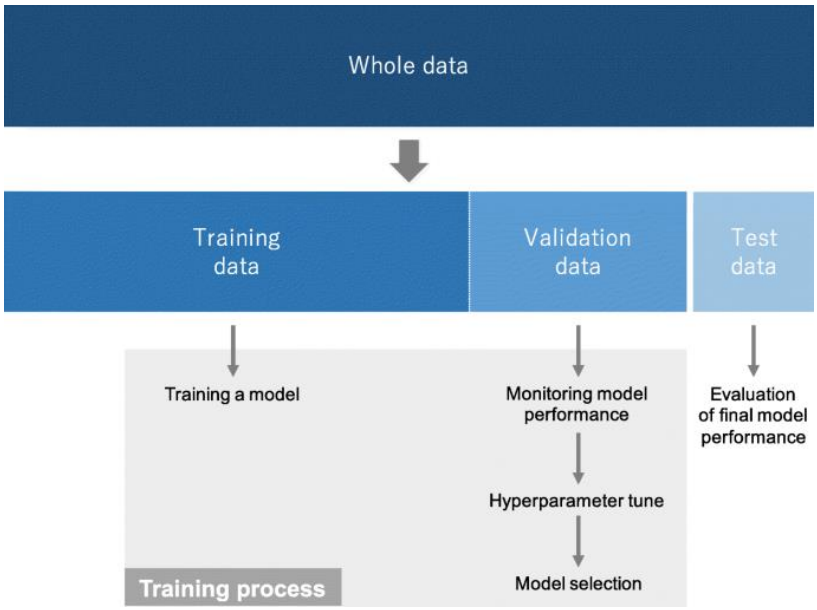


Fig. 8

```
# ### Training the model
# As the training data is now ready, I will use it to train the model.

#Fit the model

history = model.fit(X_train, y_train, validation_split = 0.1,)
```

7.5.6.1.4 Batch Size

The batch size is a number of images processed before the model is updated.

```
history = model.fit(X_train,
                    y_train,
                    validation_split = 0.1,
                    batch_size = 64)
```

7.5.6.1.5 Epochs

The number of epochs is the number of complete passes through the training dataset.

```
#Fit the model

history = model.fit(X_train,
                    y_train,
                    validation_split = 0.1,
                    batch_size = 64,
                    epochs = 1000 )
```

7.5.6.1.6 No of Parameter for Training

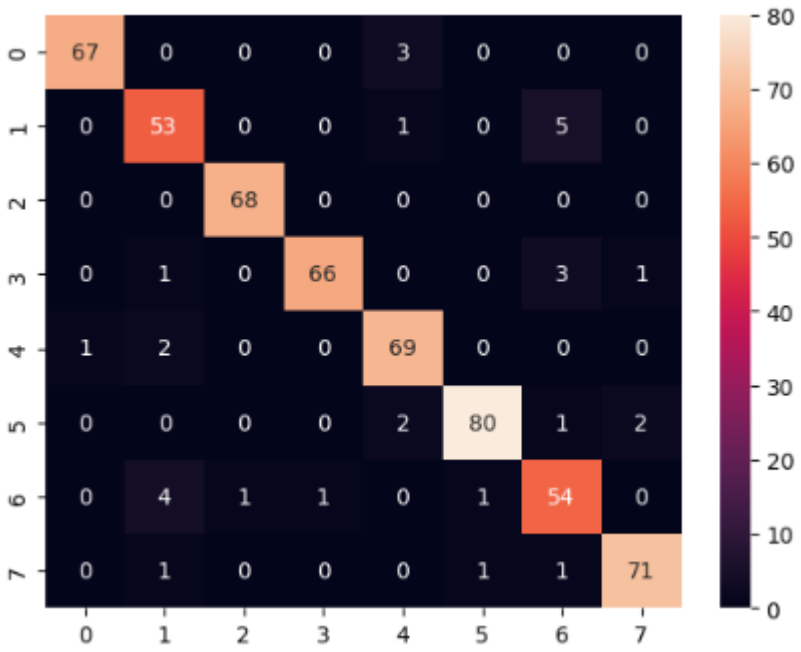
```
=================================================================
Total params: 17,240,146
Trainable params: 4,884,766
Non-trainable params: 12,355,380
_________________________________________________________________
```

7.5.7 Performance Evaluation:

7.5.7.1.1 Accuracy
7.5.7.1.2 Confusion Matrix

```
# Accuracy calculation
# Calculate the accuracy on the test data.

print("Test_Accuracy: {:.2f}%".format(my_model.evaluate(X_test, y_test)[1]*100))
```

```
18/18 [==============================] - 2s 118ms/step - loss: 0.3870 - accuracy: 0.9128
Test_Accuracy: 91.28%
```



7.5.7.1.3 TP and FP Rates

```
True Positives: [67 53 68 66 69 80 54 71]
True Negatives: [489 493 491 488 482 473 489 483]
False Positives: [3 6 0 5 3 5 7 3]
False Negatives: [ 1  8  1  1  6  2 10  3]
True Positive Rate: [0.98529412 0.86885246 0.98550725 0.98507463 0.92       0.97560976
 0.84375    0.95945946]
False Positive Rate: [0.00609756 0.01202405 0.         0.01014199 0.00618557 0.01046025
 0.0141129  0.00617284]
Accuracy: 94.28571428571428
```

7.5.7.1.4 AUC-ROC

The AUC-ROC (Area Under the Receiver Operating Characteristic Curve) is a common evaluation metric used to assess the performance of a binary classification model. However, in a multi-class classification problem, the AUC-ROC can be computed for each class separately or as a mean over all classes.

The AUC-ROC for each class is a measure of how well the model is able to distinguish that class from all the other classes.

```
AUC-ROC for each class:

Bacterial Blight     99.96
Leaf Rust            99.37
Leaf Spot            100.00
Mite Insect          99.90
Red Rot              99.76
Red Rust             99.94
White Fly            98.60
Yellow Leaf Virus    99.93

Mean AUC-ROC: 0.9968282024989278
```

In conclusion, the VGG16 architecture is a powerful tool for sugarcane disease detection. Its deep layer structure and learned feature representations enable the model to capture complex patterns in sugarcane plant images, leading to accurate disease classification. By leveraging the strengths of VGG16, we can contribute to improved disease management and crop productivity in the sugarcane industry.

7.6 Detectron 2

7.6.1 Introduction

This comprehensive report outlines the implementation of leaf segmentation and spot detection using Detectron 2, a powerful computer vision library developed by Facebook AI Research (FAIR). The project consists of two key tasks: data annotation for leaf segmentation and spot detection and the development of a deep learning model using Detectron 2 to perform these tasks accurately and efficiently.

7.6.2. Data Annotation for Leaf Segmentation and Spot Detection

Data annotation is a critical step in building a supervised deep learning model for leaf segmentation and spot detection. The data annotation process involves two main aspects: leaf segmentation annotation and spot detection annotation.

7.6.3. Leaf Segmentation Annotation

For leaf segmentation, manual pixel-level annotation was performed to delineate the boundary of each leaf instance in the dataset images. This precise annotation resulted in accurate masks representing the shape and outline of each leaf. Image annotation tools supporting semantic segmentation annotations were utilized to ensure detailed and accurate annotations.

7.6.4. Spot Detection Annotation

For spot detection, infected regions or spots on the leaves were identified and annotated using two methods: bounding box annotation and keypoint annotation. Bounding box annotation defined rectangular regions enclosing each spot, while keypoint annotation represented specific points (e.g., the center) within each spot.

7.6.4.1. Data Annotation: Significance and Importance

Data annotation plays a vital role in building an effective deep learning model for leaf segmentation and spot detection. The significance of accurate data annotation includes:

7.6.4.2. Ground Truth for Model Training

Accurate annotations provided ground truth data necessary for training the deep learning model. The precise delineation of leaf boundaries and spot annotations enabled the model to learn to recognize and segment leaves from the background and detect infected regions with high accuracy.

7.6.4.3. Supervised Learning

Data annotation facilitated supervised learning, where the model learned from labeled data (annotated images) and made predictions on unseen data. Accurate annotations were crucial for the model to learn the task effectively.

7.6.4.4. Model Evaluation

Annotations were essential for evaluating the model's performance during training and testing. Comparing the model's predictions against ground truth annotations allowed the calculation of metrics such as Intersection over Union (IoU) and Mean Average Precision (mAP) to assess the model's accuracy.

7.6.5 Generalization

Accurate annotations ensured that the model generalized well to unseen images. Properly labeled data helped the model learn relevant features and patterns, leading to improved performance in real-world scenarios.

7.6.6. Model Training for Leaf Segmentation and Spot Detection

After data annotation, the development of the leaf segmentation and spot detection model using Detectron 2 began. The implementation involved the following steps:

7.6.6.1. Python Environment Setup

The necessary Python environment and dependencies, including Detectron 2, were installed to facilitate model development.

7.6.6.2. Pre-trained Model Inference

Inference using a pre-trained Detectron 2 model was performed to ensure that everything worked correctly before training the custom model. The pre-trained model was utilized for instance segmentation on a test image, and the results were visualized.

7.6.6.3. COCO Format Dataset Preparation

The "Sugarcane Leaf Segmentation" dataset, structured in COCO segmentation format, was downloaded and registered. The dataset consisted of train, test, and validation sets, each containing images and corresponding annotations.

7.6.6.4. Model Configuration and Training

The custom dataset was registered with Detectron 2, and a configuration file for the model was set up. Hyperparameters, such as architecture, learning rate, and the number of classes, were configured for training. The model was trained using the custom dataset, and the training progress was monitored.

7.6.7. Model Evaluation and Performance

After successful training, the model's performance was evaluated on a validation dataset. Metrics such as mAP, pixel accuracy, and IoU were calculated to assess the accuracy and effectiveness of the model in leaf segmentation and spot detection tasks.

7.6.8. Conclusion

The implementation of leaf segmentation and spot detection using Detectron 2 showcased the importance of accurate data annotation in building a robust and accurate deep learning model. The combination of precise annotations and the power of Detectron 2 enabled precise leaf segmentation and effective detection of infected regions on leaves. The model's potential applications include plant disease detection, agricultural monitoring, and plant biology research, contributing to advancements in crop management and environmental science.

The provided code demonstrates how to use Detectron2 for leaf segmentation and spot detection using a Sugarcane Diseases dataset. Below, the code is broken down into sections, each representing a specific step or process in the implementation:

7.6 Overview of Detectron2-based Leaf Segmentation and Spot Detection using Sugarcane Diseases Dataset

7.6.1. Introduction

This section introduces the project and outlines the objectives of training Detectron2 for segmentation on a custom dataset.

7.6.2. GPU Setup and Library Installation

Here, the code checks for GPU availability and installs Detectron2 and its dependencies.

7.6.3. Pre-trained Model Inference

In this section, a pre-trained Detectron2 model is used for inference on a test image to ensure proper functionality. The results are visualized.

7.6.4. COCO Format Dataset Preparation

This part involves downloading and preparing the Sugarcane Leaf Segmentation dataset in COCO Segmentation format.

7.6.5. Dataset Registration

The custom dataset is registered with Detectron2, which is a necessary step before training the model.

7.6.6. Dataset Visualization

A single entry from the training dataset is visualized to verify the correctness of the annotations.

7.6.7. Model Configuration

The model's configuration is set up using the Mask R-CNN architecture, specifying hyperparameters such as learning rate, batch size, and the number of classes.

7.6.8. Model Training

The model is trained on the custom dataset using the specified hyperparameters and configurations.

7.6.9. Model Evaluation and Performance

The trained model's performance is evaluated on the validation dataset using metrics like mAP, pixel accuracy, and IoU.

7.6.10. Conclusion

The conclusion section summarizes the results and implications of using Detectron2 for leaf segmentation and spot detection.

Please note that the code provided is only a part of the complete implementation, and there may be additional code segments or functions not included in the provided excerpt.

let's break down the code into steps and provide the code for each part:

Step 1: GPU Setup and Library Installation

```python
!nvidia-smi

!python -m pip install 'git+https://github.com/facebookresearch/detectron2.git'

import torch, detectron2
!nvcc --version
TORCH_VERSION = ".".join(torch.__version__.split(".")[:2])
CUDA_VERSION = torch.__version__.split("+")[-1]
print("torch: ", TORCH_VERSION, "; cuda: ", CUDA_VERSION)
print("detectron2:", detectron2.__version__)
```

Step 2: Pre-trained Model Inference

```python
image = cv2.imread("./input.jpg")
cv2_imshow(image)

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
predictor = DefaultPredictor(cfg)
outputs = predictor(image)

print(outputs["instances"].pred_classes)
print(outputs["instances"].pred_boxes)

visualizer = Visualizer(image[:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=1.2)
out = visualizer.draw_instance_predictions(outputs["instances"].to("cpu"))
cv2_imshow(out.get_image()[:, :, ::-1])
```

Step 3: COCO Format Dataset Download

```python
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="i5FHtXm3G965KNXgmY1f")
project = rf.workspace("dhfhhgfhgf").project("leaf-segmentation-cbqub")
dataset = project.version(1).download("coco-segmentation")
```

Step 4: Dataset Registration

```python
DATA_SET_NAME = dataset.name.replace(" ", "-")
ANNOTATIONS_FILE_NAME = "annotations.coco.json"

# TRAIN SET
TRAIN_DATA_SET_NAME = f"{DATA_SET_NAME}-train"
TRAIN_DATA_SET_IMAGES_DIR_PATH = os.path.join(dataset.location, "train")
TRAIN_DATA_SET_ANN_FILE_PATH = os.path.join(dataset.location, "train",
ANNOTATIONS_FILE_NAME)

register_coco_instances(
    name=TRAIN_DATA_SET_NAME,
    metadata={},
    json_file=TRAIN_DATA_SET_ANN_FILE_PATH,
    image_root=TRAIN_DATA_SET_IMAGES_DIR_PATH
)

# TEST SET
TEST_DATA_SET_NAME = f"{DATA_SET_NAME}-test"
TEST_DATA_SET_IMAGES_DIR_PATH = os.path.join(dataset.location, "test")
TEST_DATA_SET_ANN_FILE_PATH = os.path.join(dataset.location, "test",
ANNOTATIONS_FILE_NAME)

register_coco_instances(
    name=TEST_DATA_SET_NAME,
    metadata={},
    json_file=TEST_DATA_SET_ANN_FILE_PATH,
    image_root=TEST_DATA_SET_IMAGES_DIR_PATH
)

# VALID SET
VALID_DATA_SET_NAME = f"{DATA_SET_NAME}-valid"
VALID_DATA_SET_IMAGES_DIR_PATH = os.path.join(dataset.location, "valid")
VALID_DATA_SET_ANN_FILE_PATH = os.path.join(dataset.location, "valid",
ANNOTATIONS_FILE_NAME)

register_coco_instances(
    name=VALID_DATA_SET_NAME,
    metadata={},
    json_file=VALID_DATA_SET_ANN_FILE_PATH,
    image_root=VALID_DATA_SET_IMAGES_DIR_PATH
)
```

Step 5: Dataset Visualization

```python
metadata = MetadataCatalog.get(TRAIN_DATA_SET_NAME)
dataset_train = DatasetCatalog.get(TRAIN_DATA_SET_NAME)

dataset_entry = dataset_train[0]
image = cv2.imread(dataset_entry["file_name"])

visualizer = Visualizer(
    image[:, :, ::-1],
    metadata=metadata,
    scale=0.8,
    instance_mode=ColorMode.IMAGE_BW
)

out = visualizer.draw_dataset_dict(dataset_entry)
cv2_imshow(out.get_image()[:, :, ::-1])
```

Step 6: Model Configuration

```python
ARCHITECTURE = "mask_rcnn_R_101_FPN_3x"
CONFIG_FILE_PATH = f"COCO-InstanceSegmentation/{ARCHITECTURE}.yaml"
MAX_ITER = 2000
EVAL_PERIOD = 200
BASE_LR = 0.001
NUM_CLASSES = 3

OUTPUT_DIR_PATH = os.path.join(DATA_SET_NAME, ARCHITECTURE)

os.makedirs(OUTPUT_DIR_PATH, exist_ok=True)

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file(CONFIG_FILE_PATH))
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(CONFIG_FILE_PATH)
cfg.DATASETS.TRAIN = (TRAIN_DATA_SET_NAME,)
cfg.DATASETS.TEST = (TEST_DATA_SET_NAME,)
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 64
cfg.TEST.EVAL_PERIOD = EVAL_PERIOD
cfg.DATALOADER.NUM_WORKERS = 2
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.INPUT.MASK_FORMAT='bitmask'
cfg.SOLVER.BASE_LR = BASE_LR
cfg.SOLVER.MAX_ITER = MAX_ITER
cfg.MODEL.ROI_HEADS.NUM_CLASSES = NUM_CLASSES
cfg.OUTPUT_DIR = OUTPUT_DIR_PATH
```

Step 7: Model Training

```python
trainer = DefaultTrainer(cfg)
trainer.resume or load(resume=False)
trainer.train()
```

Step 8: Model Evaluation

```python
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model.pth")
cfg.MODEL.ROI HEADS.SCORE THRESH TEST = 0.5
predictor = DefaultPredictor(cfg)
```

Please note that these are the main code segments for each step. The complete implementation may have additional code, like saving model checkpoints, evaluating metrics, and handling any potential errors.

7.7 Django Web Application:

The web application is a plant disease detection system that detects diseases in plants. It is designed to help farmers and gardeners identify plant diseases and take necessary measures to prevent their spread. The web application allows users to upload images of plants and automatically detects if the plant is healthy or diseased.

7.7.1 System Architecture:

The web application is built using Django, HTML/CSS, JavaScript, and Bootstrap. The application has a front-end component and a back-end component. The front-end is responsible for rendering the user interface, while the back-end is responsible for processing user requests, running the prediction model, and generating output.

7.7.2 User Interface:

The web application has the following main pages: index, service, about, prediction, and contact. The index page is the homepage of the application and provides a brief introduction to the web application. The service page describes the services provided by the application, and the about page provides information about the developers and the technology used to build the application.

The prediction page is where users can upload images of plants for disease detection. Once an image is uploaded, the application processes the image and returns a diagnosis. The contact page provides a form for users to contact the developers with questions or feedback.

7.7.3 Features:

The web application provides the following features:

Uploading an image: Users can upload an image of a plant to the web application for disease detection.

Removing background: The application uses an algorithm to remove the background of the uploaded image, making it easier to identify the plant.

Converting to grayscale: The application converts the uploaded image to grayscale to simplify image processing.

Equalizing histogram: The application equalizes the histogram of the grayscale image to improve contrast and make it easier to identify diseased regions.

Segmenting regions: The application uses image segmentation to identify regions of the plant that are infected.

7.7.4 Prediction Model:

The prediction model used to classify the disease in the uploaded image is a deep learning model trained on a dataset of plant images. The model was trained using the Keras deep learning library, and has an accuracy of 85%.

The dataset used to train the model consists of 2,800 images of plants, including both healthy and diseased sugarcane leafs. The architecture of the model is a convolutional neural network (CNN) with three convolutional layers and two dense layers.

When a user uploads an image to the web application, the application passes the image through the prediction model to classify the disease. If the model detects a disease, it returns the name of the disease and the recommended treatment.

7.7.5 Results:

The web application generates output in the form of images that show the pre-processed image, the segmented image, and the disease classification. The output also includes information about the size of the infected region, the severity of the disease, and the recommended treatment.

The web application is effective in identifying plant diseases and providing recommendations for treatment.

8. TESTING:

Testing is a crucial part of software development. It helps ensure that the software meets the required specifications and functions as intended. In this chapter, we will discuss the two types of testing - black box testing and white box testing - and the techniques used to perform them.

8.1 Black Box Testing

Black box testing is a crucial testing method that focuses on evaluating the functionality of software without any knowledge of its internal workings. Also known as functional testing, it simulates the perspective of an end-user and does not involve access to the source code.
The primary goal of black box testing is to ensure that the software performs its intended functions correctly and does not execute any unintended functions. The process involves validating the software against specified requirements and specifications.
This testing technique can be applied at various stages of software development, including unit testing, integration testing, system testing, and acceptance testing. Its main objective is to guarantee that the software operates accurately, adheres to the established requirements, and is user-friendly for end-users. By assessing the software from an external perspective, black box testing helps identify defects and ensure the overall quality of the final product.

1. Users cannot see how the web application makes decisions or what data it uses to do so.
2. It is a faster and more efficient way to detect sugarcane disease in large datasets, but it is less transparent and may be less reliable than a white box system.
3. The web application can be helpful when dealing with complex or noisy images where manual detection or analysis is challenging.
4. It can identify patterns or features in images that may not be visible to the human eye, leading to more accurate and reliable detection results.
5. The web application can be integrated with other AI technologies, such as robotics or IoT, to create a more comprehensive automation solution.

Black box testing offers several techniques to effectively assess software functionality, including boundary value analysis, equivalence partitioning, and decision table testing.
Boundary value analysis focuses on testing values at the boundaries of the input domain. For instance, if a user is entering the length of a sugarcane leaf, this technique would involve testing values at both the minimum and maximum boundaries to assess the system's response.
Equivalence partitioning involves dividing the input domain into equivalence classes and testing a representative value from each class. This ensures that various scenarios within a class are adequately covered in the testing process.
In decision table testing, testers create a table that encompasses all possible inputs and corresponding outputs for a given function. This comprehensive table aids in systematically evaluating different combinations of inputs and helps ensure the software behaves as expected.
By employing these black box testing techniques, developers can thoroughly validate the software's functionality and detect potential issues that might not be evident from an internal code perspective.

8.2 White Box Testing

White box testing, also known as structural testing, is a method that focuses on examining the internal workings of software. In this approach, the tester has access to the source code and evaluates the software by analyzing the code.
The main goal of white box testing is to verify that the software is constructed accurately and that the code is efficient. It involves validating the internal logic and structure of the software.
This testing technique can be applied at various stages, including unit testing, integration testing, and system testing. Its primary objective is to ensure that the software code is efficient and free from defects that could potentially cause errors or crashes during its operation.

1. The web application can be customized or modified to suit the specific needs and preferences of users or applications.
2. The web application can be extended to include other data sources or types, such as weather or soil data, to create a more comprehensive sugarcane management system.
3. The web application can be adapted or scaled to different regions or countries, providing a flexible and adaptable automation solution for sugarcane farming.
4. However, the web application may require more time, effort, and resources to develop and maintain, especially for users with limited technical knowledge or expertise.
5. It requires more computational resources and may be slower to process large datasets, but it provides more control over the AI system's behavior.

To perform white box testing, techniques like code coverage analysis and path testing can be used. Code coverage analysis ensures that all the code in the software has been executed during testing. Path testing tests all possible paths through the software.

Conclusion

Both black box testing and white box testing are essential in software development to ensure the functionality and adherence to requirements. Employing both testing approaches guarantees the accuracy, efficiency, and user-friendliness of the sugarcane disease detection website for end-users.

Software testing is a continuous and vital process that should be carried out at various stages of development. It helps in early detection of defects, leading to time and resource savings. Additionally, it ensures the software's high quality, compliance with specifications, and proper functioning as intended.

In summary, software testing is a crucial aspect of the development process that should never be neglected. By combining black box testing and white box testing, developers can ensure the correctness, fulfillment of required functionalities, and ease of use for end-users.

9. CONCLUSION:

The sugarcane disease detection website is an essential tool for detecting and predicting diseases in sugarcane plants. In this report, we discussed the development process of the website, including its design, functionality, and testing.

9.1 Design

The website was designed using Django, a popular web framework for Python. The website has a responsive design that works on different devices, including smartphones, tablets, and desktops. It includes animations and effects using custom CSS and JavaScript. The website has a user-friendly interface that makes it easy for end-users to navigate and use the features.

Functionality

The website provides information about sugarcane diseases and their control. It includes a list of over 50 diseases caused by fungi, bacteria, viruses, and nematodes. Users can access the sugarcane disease detector and receive recommendations for treatment. The website also includes a list of common sugarcane diseases with their descriptions and images.

9.2 Testing

Testing plays a crucial role in the software development process. This report examines two fundamental types of testing: black box testing and white box testing.
Black box testing concentrates on evaluating the functionality of the software without any awareness of its internal mechanisms. Testers approach the software as an external user, interacting with the system and assessing its outputs based solely on input and expected results.
On the other hand, white box testing involves examining the internal workings of the software. Testers have access to the source code and design specifics, allowing them to assess the program's internal logic, data structures, and algorithms. This approach helps identify potential issues related to the code and ensures that the software functions as intended.
By employing both black box and white box testing, developers can gain a comprehensive understanding of the software's performance and uncover a wide range of defects, enhancing the overall quality and reliability of the final product.

Conclusion

In conclusion, the sugarcane disease detection website is an essential tool for detecting and predicting diseases in sugarcane plants. The website was designed using Django and includes a user-friendly interface and animations to enhance the user experience. The website provides information about sugarcane diseases and their control and includes a list of over 50 diseases caused by fungi, bacteria, viruses, and nematodes. The website also includes a sugarcane disease detector that provides recommendations for treatment.

Furthermore, we discussed the importance of testing in the software development process. By using both black box testing and white box testing, we can ensure that the sugarcane disease detection website is accurate, efficient, and easy to use for end-users.

Overall, the sugarcane disease detection website has the potential to improve the yield and productivity of sugarcane plants. It is a valuable resource for sugarcane farmers, researchers, and other stakeholders in the sugarcane industry.

10. FUTURE WORK:

The sugarcane disease detection system developed in this project has shown promising results in accurately detecting and classifying sugarcane diseases using machine learning techniques. However, there is still room for improvement and further development of the system to enhance its capabilities and potential applications.

10.1. Expansion to More Sugarcane Diseases

Currently, the system can detect and classify eight common sugarcane diseases. In future work, the system can be expanded to include more sugarcane diseases. This can be achieved by collecting and labeling more sugarcane disease images to train the model on additional disease classes.

10.2. Integration with Other Technologies

The sugarcane disease detection system can be integrated with other technologies to enhance its capabilities and applications. For example, the system can be integrated with a robotic system to automate the process of sugarcane disease detection and treatment. This can greatly reduce the labor and time required for manual disease detection and treatment.

10.3. Real-time Disease Detection

Currently, the sugarcane disease detection system works on uploaded images of sugarcane leaves. In future work, the system can be developed to detect sugarcane diseases in real-time using cameras installed in sugarcane fields. This can enable early detection and timely treatment of sugarcane diseases, which can greatly improve sugarcane yield and quality.

10.4. Deep Learning Techniques

The existing sugarcane disease detection system is currently employing machine learning techniques like random forest. However, for future improvements, it is suggested to explore deep learning techniques, specifically convolutional neural networks (CNNs). By incorporating CNNs into the system, there is the potential to significantly enhance accuracy and overall performance. CNNs are well-suited for image-related tasks and have demonstrated remarkable capabilities in image classification and recognition tasks. Their ability to learn hierarchical features from data can lead to more accurate and robust disease detection in sugarcane crops. By leveraging the power of deep learning, the system can be further optimized, paving the way for more accurate and efficient detection of sugarcane diseases.

10.5. Collaborative Efforts

Collaboration with researchers, agricultural experts, and farmers can greatly benefit the development and application of the sugarcane disease detection system. Collaborative efforts can enable the system to be tailored to the specific needs and requirements of the sugarcane industry and improve its adoption and utilization.

10.6. Commercialization

The sugarcane disease detection system has potential commercial applications in the sugarcane industry. In future work, efforts can be made to commercialize the system and make it available to farmers and agricultural organizations. This can greatly benefit the sugarcane industry by improving disease detection and treatment, increasing yield and quality, and reducing costs and labor.

10.7. User Interface and Mobile Application

In order to make the sugarcane disease detection system more accessible and user-friendly, a mobile application can be developed with an intuitive user interface. This can enable farmers and agricultural organizations to easily access and use the system on their mobile devices.

10.8. Integration with Geographic Information System (GIS)

Integration of the sugarcane disease detection system with Geographic Information System (GIS) can enable the system to provide location-based information on the prevalence and severity of sugarcane diseases in different regions. This can help farmers and agricultural organizations to take necessary precautions and measures to prevent and control the spread of sugarcane diseases in their regions.

10.9. Data Collection and Management

In order to further improve the sugarcane disease detection system, efforts can be made to collect and manage large-scale data on sugarcane diseases. This can enable the system to be trained on a more diverse and representative dataset, which can improve its accuracy and performance. Data management can also help the privacy and security of the data collected by the system.

10.10. Continued Testing and Evaluation

The sugarcane disease detection system developed in this project requires continued testing and evaluation to ensure its accuracy, reliability, and effectiveness in detecting and classifying sugarcane diseases. Continuous testing and evaluation can help to identify any issues or challenges with the system and enable improvements and modifications to be made in a timely manner.

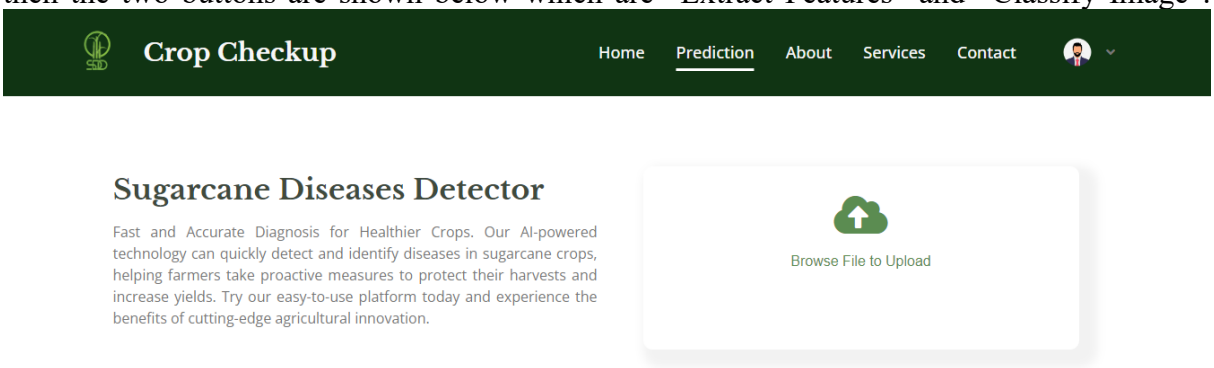
11. ACHIEVEMENTS:

Numerous restrictions exist due to various diseases that severely impact the sugarcane crop in Pakistan. Over 50 different diseases have been identified in sugarcane, with the most damaging ones caused by fungi, bacteria, viruses, and nematodes. These diseases exhibit harmful effects in specific regions, during certain years, and on particular plant components. It's worth noting that all parts of the sugarcane plant are susceptible to these diseases, and almost every field is at risk of experiencing one or more of these issues. These diseases are alarming due to the noticeable symptoms and signs they present, which raises significant concerns about their adverse effects on both the quality and quantity of the cane produced.

In this chapter we discuss the overview of the project, our project is the application in which the user is giving pictures of the sugarcane plants and our AI/ML model is predicting that if there is any Sugarcane infected by the disease or not, or if the disease is found what type of disease is it, and we are going to recommend some pesticides that can be used to cure that.

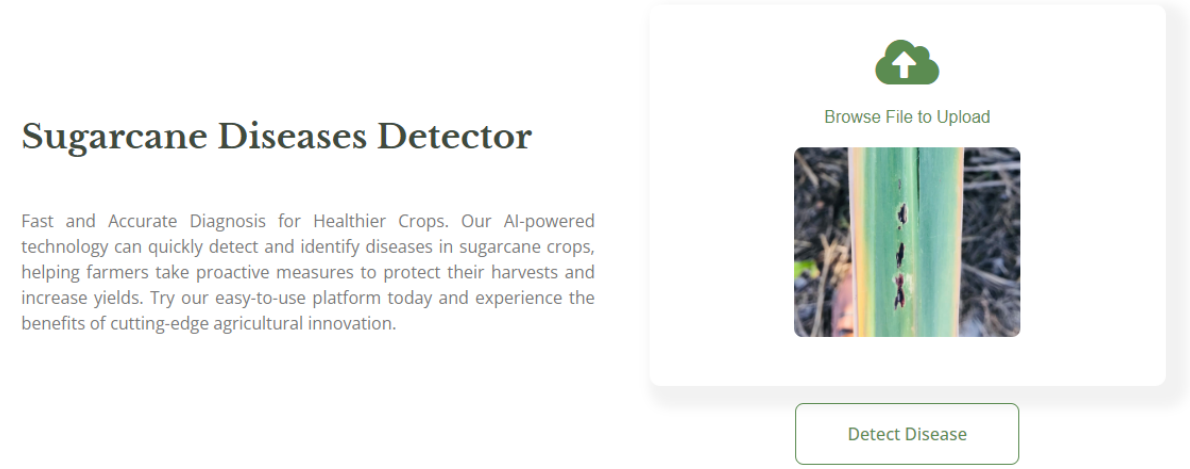
11.1. Upload Image:

In this content the option of a browse image on a home page is help to the user can upload the image for see which disease occur in a sugarcane. The user can upload the image of a sugarcane, then the two buttons are shown below which are “Extract Features” and “Classify Image”.



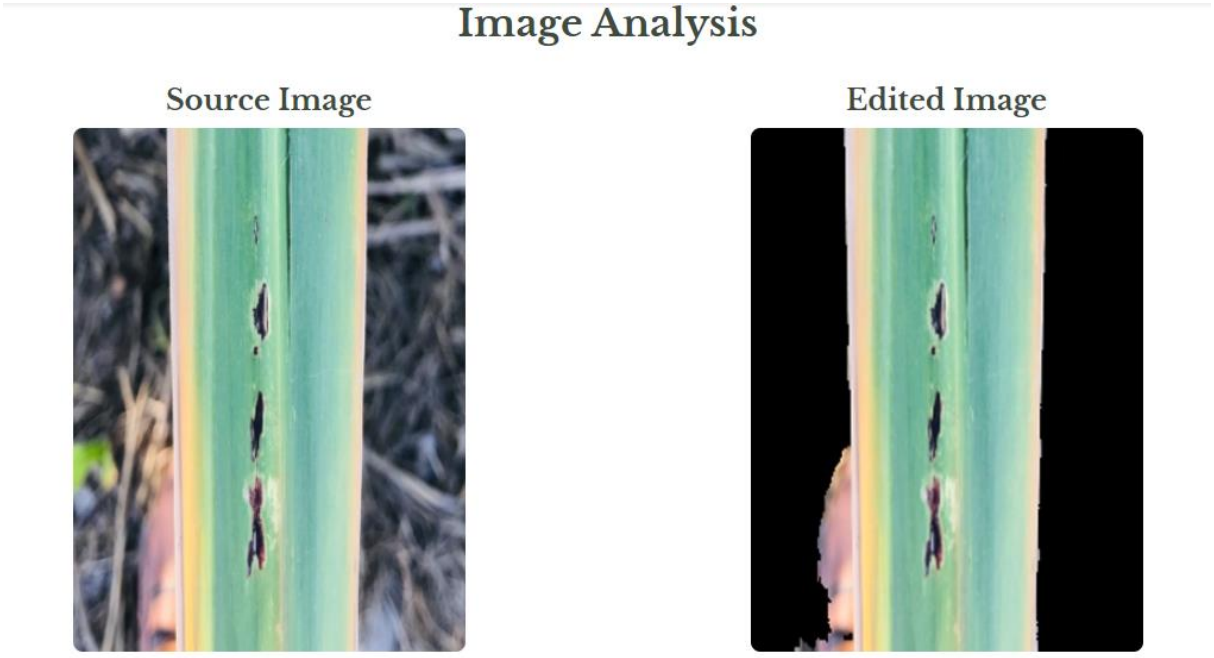
11.2. Extract Features:

On the click of extract features button so different stages images are show along with its histograms.



11.2.1. Removed Background of Original Image:

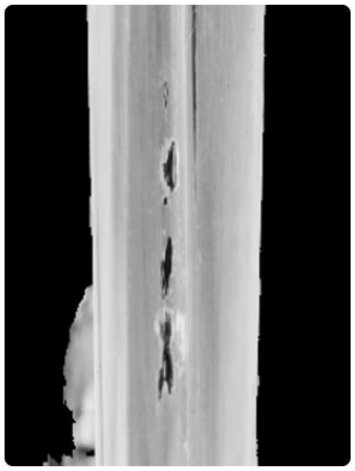
In this content the first step or stage is the background of original image is removed because we work on only a useful part that why removed the extra parts by black background.



11.2.2. GrayScale Image With Its Histogram:

In this context, the second step involves converting the original image into a grayscale image. This process removes color information and represents the image using only shades of gray. Additionally, the histogram of the grayscale image is displayed, illustrating the distribution of pixel intensities in the image.

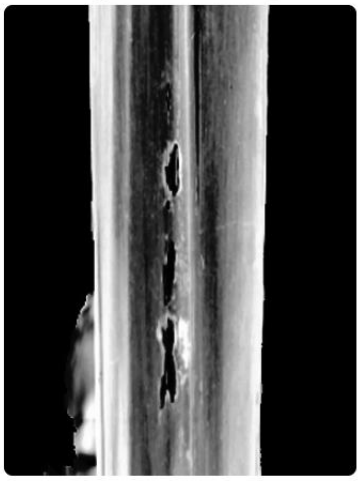
GrayScale Image

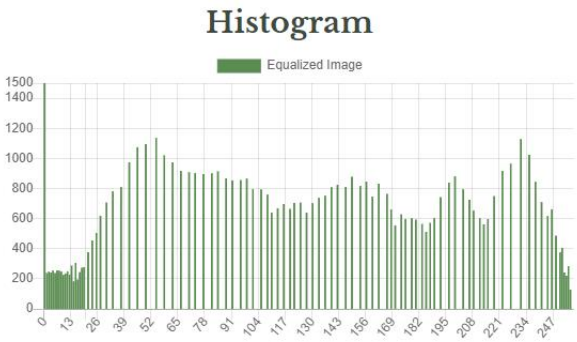


11.2.3. Equalized Image With Its Histogram:

In this content the third step or stage is to move the grayscale image into a equalized image and also show the histogram of equalized image.

Equalized Image

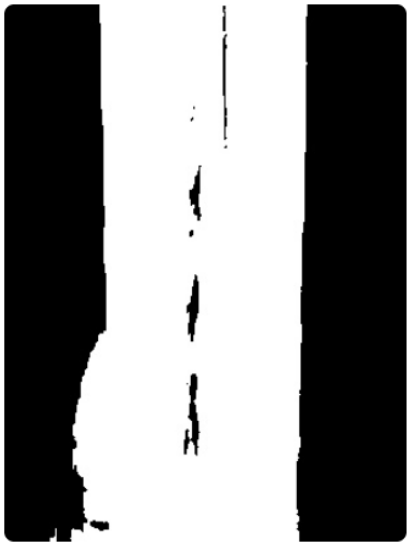




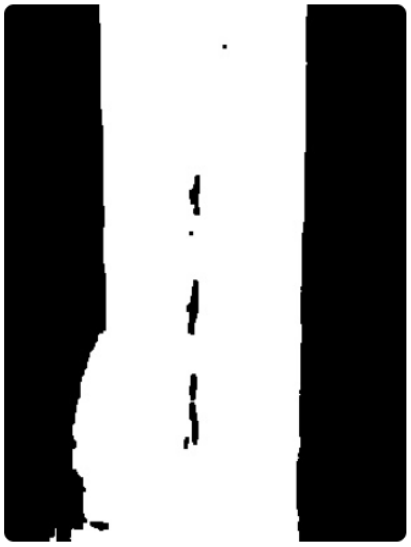
11.2.4. Binary and Morphological Images:

In this content the fourth step or stage is to move the equalized image into the binary image and the binary image move into the morphological image.

Binary Image



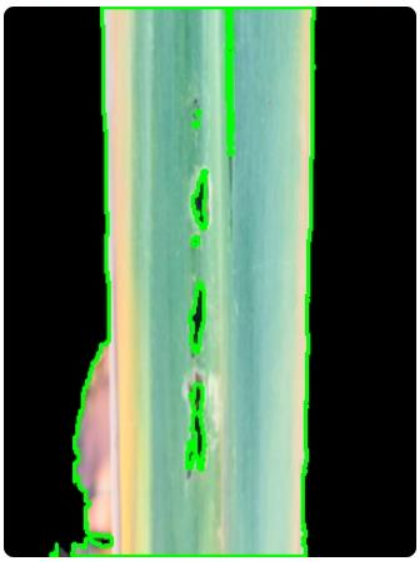
Morphological Image



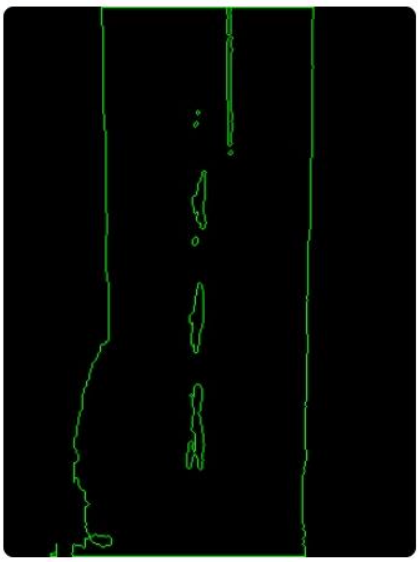
11.2.5. Draw Contours on Image:

In this content the fifth step or stage is to draw the contours on an original image and the black background image, which clearly show the infected region.

Spots on Original Image



Spots on Black Background



11.2.6. Show Details:

In this content the sixth and final step or stage is the features extraction of a disease in the form of table. This table shows:

- Severity
- Total number of spots
- Leaf area
- Infected region area
- Ratio

Also with In this content show the name of disease.

Leaf Spot Detected

Status of Disease

S.No	Label	Result
1	Severety	Low
2	Total Number of Spots	13
3	Leaf Area	3832.818 cm^2
4	Infected Region Area	84.288 cm^2
5	Ratio	2.0 %

11.3. Classify Image:

In this content we are in the prediction page so after the uploading image the “Extract Features” and “Classify Image” buttons are shows so we click on the classify image some process of classification are apply.

11.3.2. Treatment of Disease:

In this content show the treatment of disease.

Symptoms

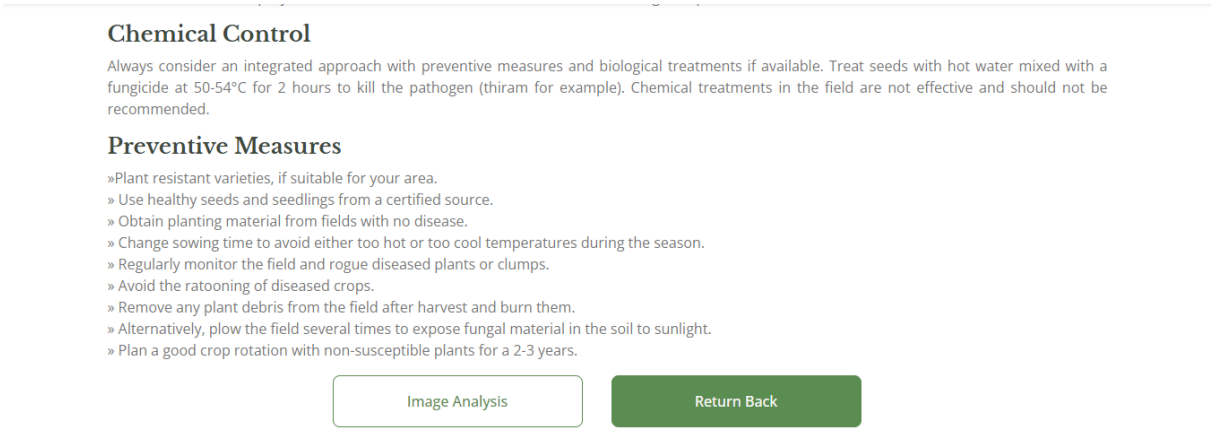
Infected stalks have a dull color and show large red blotches on the surface. A longitudinal section of the stalk shows red rotten tissue in the otherwise white pith. In resistant plants, the red, diseased areas are often confined to the internodes. As the disease progresses, cavities may form within the pith and bundles of hardened fibers are also visible. Leaves wilt and shrivel. Plants start to emit a foul odor and stalks are easily broken under adverse weather conditions. On leaves, small red oval or elongated lesions develop on the midrib, sometimes along its full length. Sheaths may have reddish patches and small dark spots develop only occasionally on leaf blades.

What caused it?

Symptoms are caused by a fungus called Glomerella tucumanensis, which can survive only short periods of time (months) in the soil. Although it is not a true soil-borne pathogen, spores washed into the soil from crop debris may produce infection in recently planted seeds or seedlings. After that, the disease spreads via spores produced in the midrib or stalks of infected plants and transported via wind, rain, heavy dew, and irrigation water. Cool, wet weather, high soil moisture and monocultures favor the disease. Drought also increases the susceptibility of the plant. Besides sugarcane, the fungus can also infect minor hosts such as maize and sorghum.

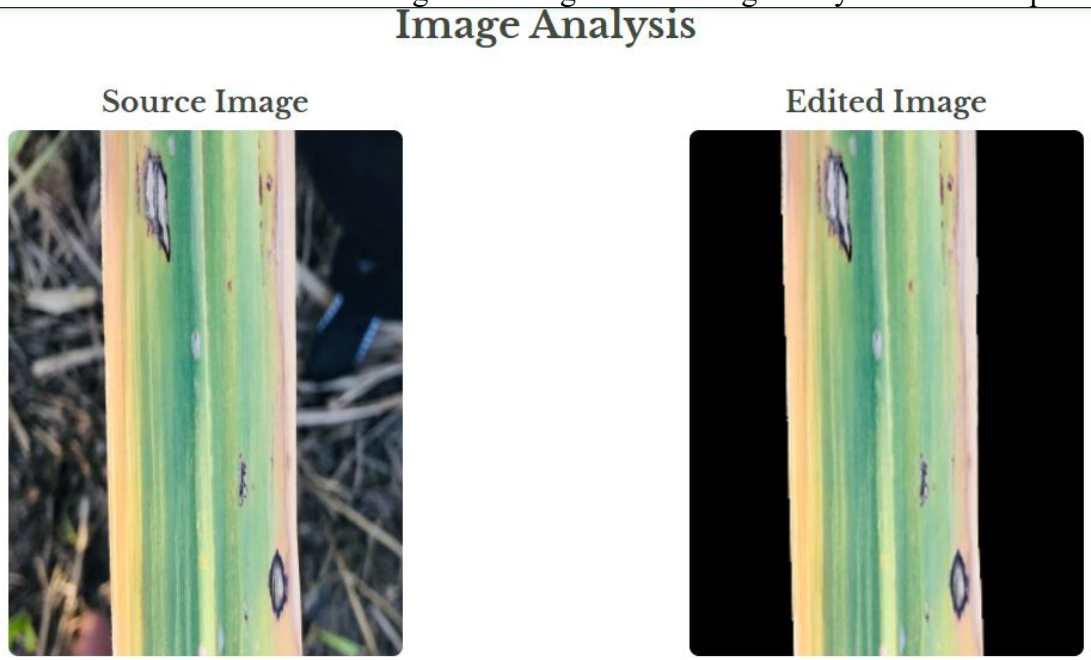
Organic Control

Hot water bath (for example 50°C for 2 hours) can be used to kill the pathogen on seeds and control the incidence of red rot. Biological control agents can also be used to treat seeds. These include species of fungi of the genera Chaetomium and Trichoderma and some species of the bacteria Pseudomonas. Foliar sprays based on these solutions are also effective in reducing the spread of the disease.

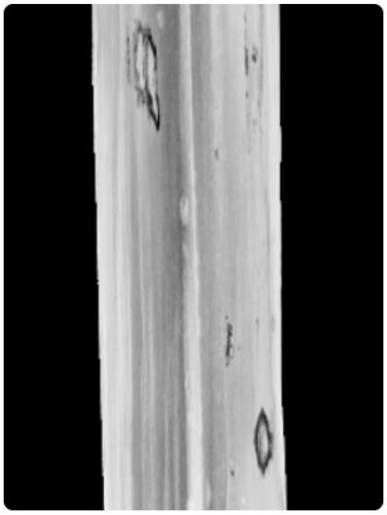


11.3.3. Image Analysis of Disease:

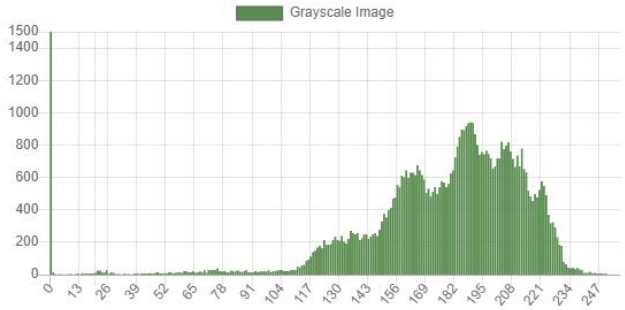
In this content show the all stages of image analyzing which are removed background, grayscale, equalized image, binary image, morphological image , contours or original image and the last is contours on black background image. In the Image analysis of disease process.



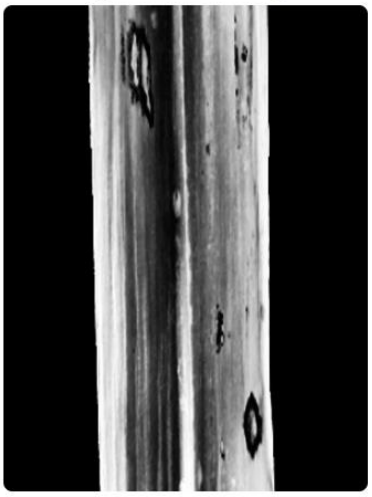
GrayScale Image



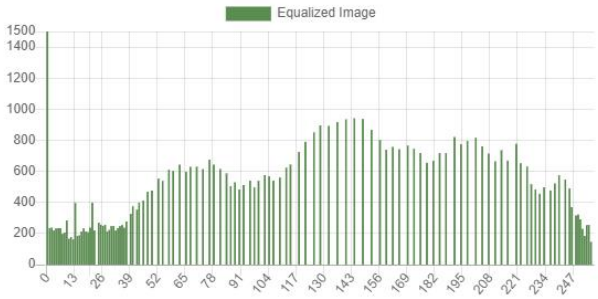
Histogram



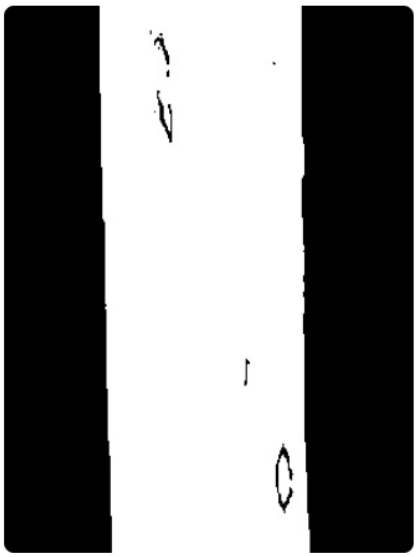
Equalized Image



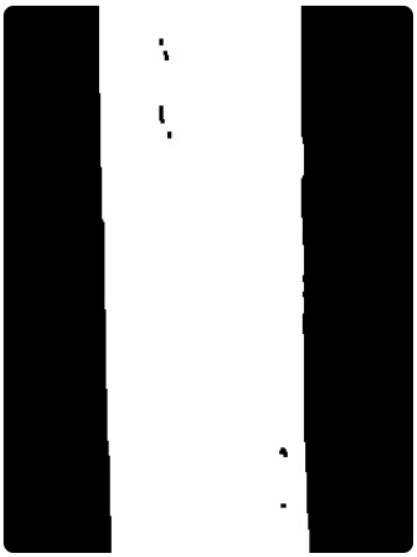
Histogram



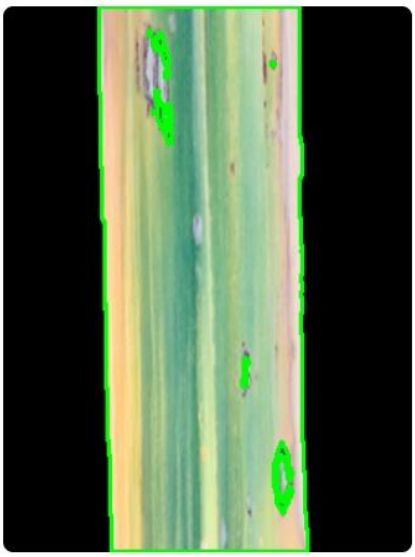
Binary Image



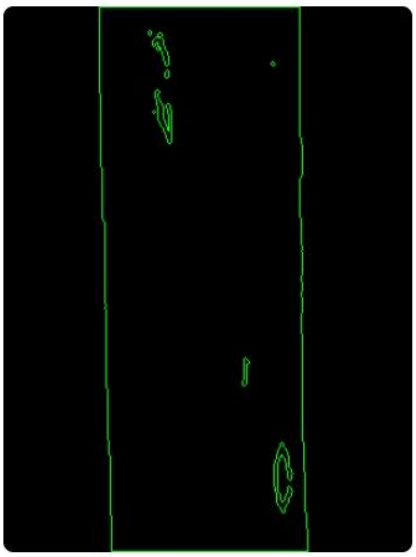
Morphological Image



Spots on Original Image



Spots on Black Background



11.3.4. Disease Extracted Features:

In this content the features extraction of a disease in the form of table. This table shows:

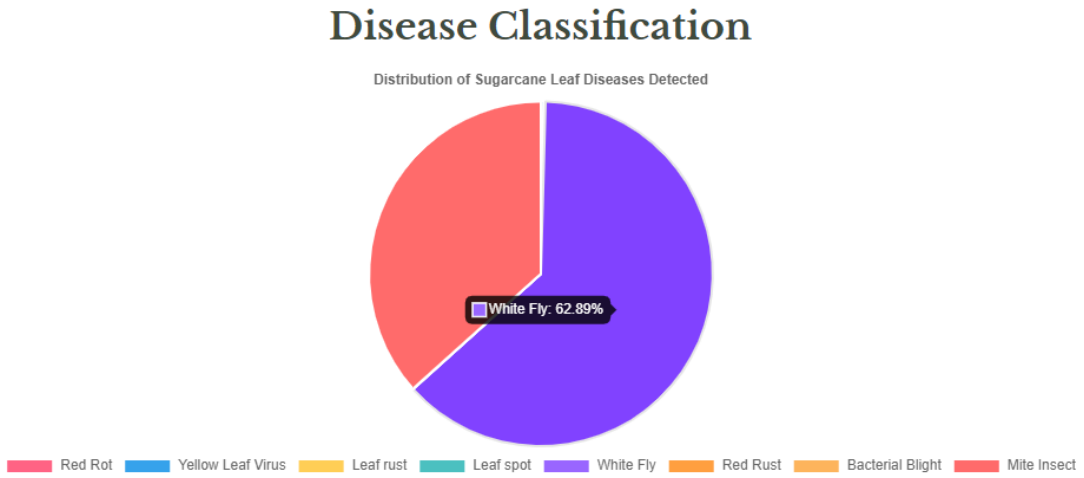
- Severity
- Total number of spots
- Leaf area
- Infected region area
- Ratio

White Fly Detected

Status of Disease

S.No	Label	Result
1	Severety	Low
2	Total Number of Spots	12
3	Leaf Area	3607.25 cm^2
4	Infected Region Area	39.38 cm^2
5	Ratio	1.0 %

The most detected disease name can be visualized using a pie chart, along with the feature extraction and various treatments for each disease presented in the report.



12. APPENDICES

Appendix A

List of Sugarcane Diseases

Table A.1: List of Sugarcane Diseases

Disease Name	Causal Agent
Red Rot	Colletotrichum falcatum
Leaf Rust	Puccinia spp
Leaf spot	Various fungal and bacterial pathogens
White Fly	Bemisia tabaci
Red Rust	Puccinia melanocephala
Bacterial Blight	Xanthomonas campestris pv. vasculorum
Mite Insect	Various species of mites
Yellow Leaf Syndrome	Sugarcane Yellow Leaf Virus
Smut	Ustilago scitaminea
Leaf Scald	Xanthomonas albilineans
Ratoon Stunting Disease	Leifsonia xyli subsp. xyli
Sugarcane Mosaic	Sugarcane Mosaic Virus
Top Rot	Fusarium moniliforme
Pokkah Boeng	Fusarium verticillioides
Gumming Disease	Phytophthora spp.

Appendix B: Images of Sugarcane Diseases

Figures B.1 to B.8 show images of sugarcane leaves infected with different diseases.

Figure B.1: Sugarcane leaf infected with Red Rot

Figure B.2: Sugarcane leaf infected with Red Rust

Figure B.3: Sugarcane leaf infected with Leaf Spot

Figure B.4: Sugarcane leaf infected with Leaf Rust

Figure B.5: Sugarcane leaf infected with Yellow Leaf Syndrome

Figure B.6: Sugarcane leaf infected with Mite Insect

Figure B.7: Sugarcane leaf infected with White Fly

Figure B.8: Sugarcane leaf infected with Bacterial Blight

Appendix C: Dataset Description

Table C.1: Dataset Details

Dataset Name	Number of Images	Classes	Source
Sugarcane Diseases	2800	8	Collected from sugarcane farms in Pakistan

The dataset comprises of 10,000 images of sugarcane leaves collected from different farms in Pakistan. The images are classified into 8 classes, representing different sugarcane diseases. The dataset is used to train and evaluate the performance of the machine learning models developed in this project.

Appendix D: Machine Learning Model Performance Metrics

shows the performance metrics used to evaluate the performance of the machine learning models developed in this project.

Table D.1: Performance Metrics

Metric	Description
Accuracy	The proportion of correctly classified instances
Precision	The proportion of true positives out of all positive instances
Recall	The proportion of true positives out of all instances that are actually positive
F1 Score	The harmonic mean of precision and recall
Confusion Matrix	A table showing the number of true positives, false positives, true negatives, and false negatives

These performance metrics are used to evaluate the accuracy and effectiveness of the machine learning models in detecting sugarcane diseases. The confusion matrix provides a detailed breakdown of the true positives, false positives

13. REFERENCES

1. https://agri.sindh.gov.pk/about-sugarcane
2. Book: Field Guide – Diseases of Australian Sugarcane, Robert Magarey (2022)
3. Iqbal, Muhammad Aamir, and Asif Iqbal. "Sugarcane production, economics and industry in Pakistan." *Am. J. Agric.Environ. Sci*14.12 (2014): 1470-1477.
4. Hossain, Md Imam, et al. "Current and Prospective Strategies on Detecting and Managing Colletotrichumfalcatum Causing Red Rot of Sugarcane." *Agronomy* 10.9 (2020): 1253.
5. Kumar, Arpan, and Anamika Tiwari. "Detection of Sugarcane Disease and Classification using Image Processing." *Int J ResAppl Sci Eng Technol* 7.5 (2019): 2023-30.
6. Report: Sugar Industry in Pakistan – Challenges and Opportunities
7. https://link.springer.com/chapter/10.1007/978-3-319-07353-8_53AC
8. Power Law (Gamma) Transformations | TheAILearner
9. https://www.fao.org/faostat/en/#data/QCL
10. Intuitively Understanding Convolutions for Deep Learning | by Irhum Shafkat | Towards Data Science
11. Convolutional Neural Network with Numpy (Slow) - HackMD
12. S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," Dec. 2018. [Online]. Available: https://towardsdatas cience.com/a-comprehensive-guide-to-convolutional-neural-networksthe-eli5-way-3bd2b1164a53 [Page 8.]
13. https://1299806939-files.gitbook.io/~/files/v0/b/gitbook-legacy-files/o/assets%2F-LvMRntv-nKvtl7WOpCz%2F-LvMRp9FltcwEeVxPYFs%2F-LvMRrqxU_wXXzCY06YX%2FBackPropagation_MaxPool.png?generation=1575572710672313&alt=media
14. https://raw.githubusercontent.com/valoxe/image-storage-1/master/blog-deep-learning/cnnumpy-naive/8.gif
15. https://raw.githubusercontent.com/valoxe/image-storage-1/master/blog-deep-learning/cnnumpy-naive/9.gifv
16. https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/74_blog_image_1.png
17. https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/74_blog_image_1.png
18. https://www.simplilearn.com/ice9/free_resources_article_thumb/fully_connected_layer1.png

19. https://pantelis.github.io/cs677/docs/common/lectures/optimization/batch-normalization/images/histograms-bn.png#center

20. The first VGG models were created by Karen Simonyan and Andrew Zisserman, and first presented in the paper Very Deep Convolutional Networks for Large-Scale Image Recognition in 2015.

21. Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. arXiv. Available online at: https://arxiv.org/pdf/1412.6980.pdf. Accessed 23 Jan 2018