Personal Professional Project

4th year Industrial Computing and automation

Plant Disease Detection

Realised by:

Bilel Bekalti

Khalil El Amine

Moemen Ben Hamad

Yassine Fendi

Supervised by:

Ms. Fafa Ben Hatira

Academic year 2022/2023

Acknowledgement

We would like to give our sincere appreciation to our supervisor, Ms. Fafa Ben Hatira, for her invaluable guidance and support throughout the duration of this project. Her expertise, insights, and unwavering commitment have been instrumental in the successful completion of this endeavor.

We are grateful for the mentoring provided by Ms.Fafa Ben Hatira, contributed to our professional growth and the refinement of our skills.

We would also extend our acknowledgment the collaborative efforts of our team members, who have worked closely together to achieve the project's objectives. Their dedication, cooperation, and shared enthusiasm have made this project a truly rewarding and enriching experience.

Furthermore, we extend our gratitude to our dear educational institution INSAT for their provision of resources and the conducive learning environment that has facilitated our research and development. Their support has been pivotal in the seamless execution of this project.

Lastly, we would like to express our heartfelt appreciation to all individuals who have directly or indirectly contributed to this project. Your contributions, encouragement, and support have played a significant role in our journey, and we are truly thankful.

Contents

List of Figures

Chapter 1

Introduction

Late blight and early blight are plant diseases that pose significant threats to crops like tomatoes and potatoes, with profound implications for agricultural production and food security. These diseases exhibit rapid spread and persistence in the environment, making effective control measures challenging. late blight caused by Phytophthora infestans and early blight caused by Alternaria solani are characterized by their ability to produce abundant spores, which can be easily disseminated by wind, water, or insects, enabling long-distance transmission and accelerating the diseases' proliferation. Furthermore, their growth and reproduction are favored by specific environmental conditions, such as cool and moist weather, facilitating their rapid expansion. The contagious nature of these diseases allows for quick transmission between infected and neighboring plants, leading to widespread outbreaks within crop fields or adjacent areas. Additionally, late blight and early blight can persist in the environment, surviving in plant debris, soil, and infected tubers or seeds, resulting in recurring outbreaks when conducive conditions reoccur. Understanding these factors is crucial for implementing effective control and management strategies to mitigate the impact of late blight and early blight on crop production.

Detecting and managing plant diseases play a vital role in maintaining crop yield and ensuring global food security. Among the common and devastating diseases that affect various crops, Late blight and early blight have emerged as significant threats, particularly in the cultivation of potatoes and tomatoes. These diseases can lead to substantial losses in agricultural production, sometimes resulting in complete crop destruction. Late blight is caused by a pathogenic oomycete that manifests as water-soaked lesions with a fuzzy, white-to-gray mold-like growth on leaves, stems, and fruits. In contrast, early blight forms concentric rings of dark brown or black color with a lighter center. To manage these diseases effectively, different approaches are required. Late blight control involves the use of fungicides, resistant varieties, and sanitation practices, while early blight management focuses on cultural practices, fungicide application, and removal of infected debris. Consulting local experts for accurate diagnosis and tailored treatment recommendations is crucial in mitigating the impact of these diseases on crop health and productivity. By addressing these challenges and developing effective disease detection and management approaches, it is possible to mitigate the detrimental effects of late blight and early blight

on agricultural production, contributing to improved food security on a global scale. As shown in the next figure, Akbar Hossain displays the potato plants that died in his field at Teesta char in Gangachara upazila of Rangpur due to late blight attack

Figure 1.1: Crop destruction by Late blight.

The ability to identify and manage plant diseases effectively can mitigate yield losses and prevent the spread of infections, ultimately impacting crop production and the livelihoods of farmers. Traditional manual methods of disease detection are labor-intensive and time-consuming, requiring expertise in plant pathology. However, advancements in computer vision and machine learning techniques have opened up new possibilities for automating the process of plant disease detection. In this project, we aim to develop a deep learning model using a Convolutional Neural Network (CNN) to detect two common plant diseases, late blight and early blight, which pose significant threats to crops such as potatoes and tomatoes.

The detection of plant diseases using deep learning techniques, particularly CNNs, has gained considerable attention in recent years. CNNs have proven to be highly effective in various computer vision tasks, including object recognition and image classification, due to their ability to automatically extract meaningful features from visual data. By leveraging CNNs, we can analyze plant images and identify disease patterns that are often challenging to detect with the naked eye. The use of deep learning models for plant disease detection offers several advantages, including improved accuracy, scalability, and potential for real-time monitoring. However, it is essential to understand the existing research landscape and the limitations associated with CNN-based approaches for plant disease detection.

To develop our deep learning model, we will start by collecting and preprocessing a dataset of plant images infected with late blight and early blight. The dataset will be carefully curated to include a diverse range of plant species and disease severities. Preprocessing steps, such as image resizing, normalization, and augmentation, will be applied to ensure the model's robustness and generalization capabilities. Once the dataset is prepared, we will design a CNN architecture tailored specifically for plant disease detection. The model will undergo rigorous training and evaluation, where we will fine-tune hyperparameters, select an appropriate optimizer and loss function, and assess the model's performance using various evaluation metrics.

Chapter 2

Literature Review

In a study by S. Khirade et al. (2015) [**1**], the authors addressed the detection of plant diseases through the utilization of digital image processing techniques and a backpropagation neural network (BPNN). Their approach involved segmenting the infected regions in plant leaves by employing Otsu's thresholding, boundary detection, and spot detection algorithms. Subsequently, various features such as color, texture, morphology, and edges were extracted for disease classification. The BPNN was employed to successfully detect and classify plant diseases.

Shiroop Madiwalar and Medha Wyawahare (2017) [**2**] conducted a comparative analysis of different image processing approaches for plant disease detection. The researchers focused on analyzing color and texture features to identify plant diseases. Their experimentation involved a dataset of 110 RGB images, and features such as mean, standard deviation, grey level co-occurrence matrix (GLCM), and Gabor filter were extracted for classification. Support vector machine (SVM) was utilized as the classifier, achieving an accuracy of 83.34% using all the extracted features.

Hyperspectral imaging was explored by Peyman Moghadam et al. (2017) [**3**] for plant disease detection. The authors utilized visible and near-infrared (VNIR) as well as shortwave infrared (SWIR) spectrums in their research. The segmentation of the leaf regions was accomplished using a k-means clustering algorithm in the spectral domain. To address the challenge of grid interference in hyperspectral images, the authors proposed a novel grid removal algorithm. The study achieved an accuracy of 83% using vegetation indices in the VNIR spectral range and 93% accuracy using the full spectrum. However, the proposed method required a costly hyperspectral camera with 324 spectral bands.

In the detection of bacterial blight in pomegranate plants, Sharath D. M. et al. (2019) [**4**] developed a system that utilized various features such as color, mean, homogeneity, standard deviation, variance, correlation, entropy, and edges. Their methodology involved grab cut segmentation to isolate the region of interest in the images, and Canny edge detection was applied to extract the edges. The developed system successfully predicted the infection levels in the fruit.

Convolutional Neural Networks (CNNs) offer several advantages for plant disease detec-

tion. They can automatically learn discriminative features from images, eliminating the need for manual feature engineering. CNNs exhibit high accuracy in classifying various plant diseases, enabling effective disease identification and management. However, CNNs have certain limitations. They require large labeled datasets for training, which can be challenging to obtain. Additionally, CNNs often demand significant computational resources and training time, making them computationally expensive for deployment in resource-constrained environments.

Chapter 3

Data Collection and Preprocessing

3.1 Dataset Description

The dataset used in this project is the Plant Village dataset which is an open source dataset sourced from Kaggle.com [**5**]. Plant Village provides a comprehensive collection of images capturing various plant species affected by different diseases. The dataset covers a wide range of crops, including tomatoes, potatoes, and peppers. It serves as a valuable resource for training machine learning models to accurately detect plant diseases.

The dataset comprises high-resolution images captured under diverse environmental conditions and at different plant growth stages. Each image is labeled with the corresponding disease or healthy category, enabling supervised learning approaches. It offers a substantial number of samples for each class, ensuring a balanced representation of different diseases and healthy plants. This diversity is essential for building robust models capable of generalizing to real-world scenarios.

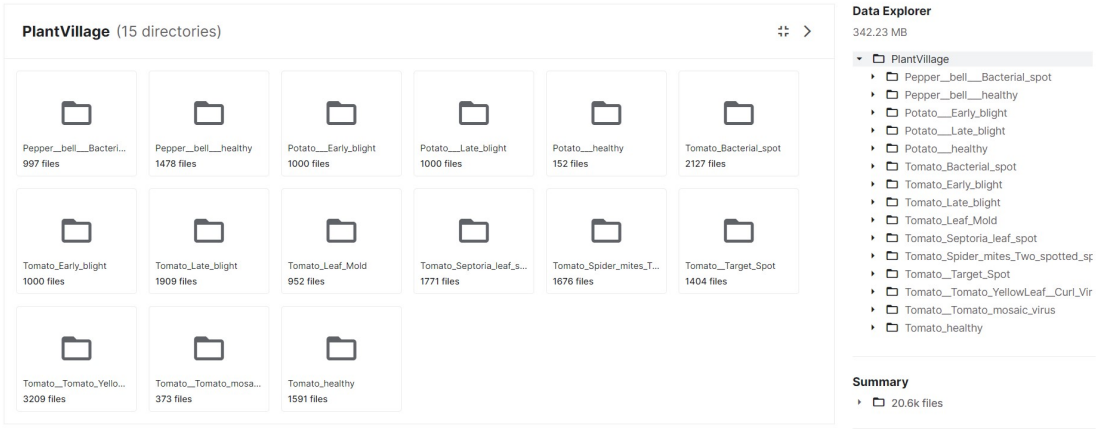


Figure 3.1: Plant Village Dataset

3.2 Dataset Extraction and Characteristics

We have extracted two subsets from the Plant Village Dataset: one consisting of potato images for initial training and the other containing tomato images for applying a transfer learning algorithm at a later stage. These extracted subsets have been used to create two new datasets, each with distinct characteristics. Here, we provide an overview of these new datasets:

Potato Dataset

- **Dataset size:** 2152 Samples.
- **Classes (Samples per class):** Potato early blight (1000), Potato late blight (1000), Potato healthy (152).

(a) Healthy (b) Early blight (c) Late blight

Figure 3.2: Samples from the Potato Dataset. (a) Represents the 'Potato healthy' class, (b) represents the 'Potato early blight' class, and (c) represents the 'Potato late blight' class

Tomato Dataset

- **Dataset size:** 4500 Samples.
- **Classes (Samples per class):** Tomato early blight (1000), Tomato late blight (1909), Tomato healthy (1591).

(a) Healthy (b) Early blight (c) Late blight

Figure 3.3: Samples from the Tomato Dataset. (a) Represents the 'Tomato healthy' class, (b) represents the 'Tomato early blight' class, and (c) represents the 'Tomato late blight' class

By creating these specialized datasets from the Plant Village Dataset, we can focus on training the model with images relevant to specific plant species, enabling more accurate and targeted disease detection for potatoes and tomatoes, respectively.

3.3 Dataset Preprocessing

To enhance the model's performance, several preprocessing steps were applied to the dataset. First, the images were resized to a consistent resolution, ensuring uniformity across the dataset. Resizing reduces computational complexity and helps the model focus on relevant features, regardless of the original image dimensions. The code snippet below demonstrates this step:

```
resize_and_rescale=tf.keras.Sequential([
    layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
    layers.experimental.preprocessing.Rescaling(1.0/255)
])
```

Normalization was then applied to standardize the pixel values of the images. This process brings the pixel intensities to a common scale, making the data more suitable for training neural networks. Normalization prevents any single feature from dominating the learning process and helps the model converge more effectively during training.

Additionally, data augmentation techniques were employed to increase the dataset's variability and mitigate overfitting. Augmentation operations, such as random rotations, flips, and zooms, were performed on the images. This strategy introduces artificial diversity, enabling the model to learn from a broader range of variations and improve its ability to generalize to unseen data. The code snippet below demonstrates this step:

```
data_augmentation=tf.keras.Sequential([
    layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
    layers.experimental.preprocessing.RandomRotation(0.2)
])
```

Further preprocessing steps were applied to optimize the reading time during training. Caching and shuffling the dataset were performed to improve the training process. The code snippet below demonstrates this step:

```
train_ds=train_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
val_ds=val_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds=test_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
```

By conducting these preprocessing steps, the dataset was prepared to ensure uniformity, improve model training, and enhance the robustness of the plant disease detection model.

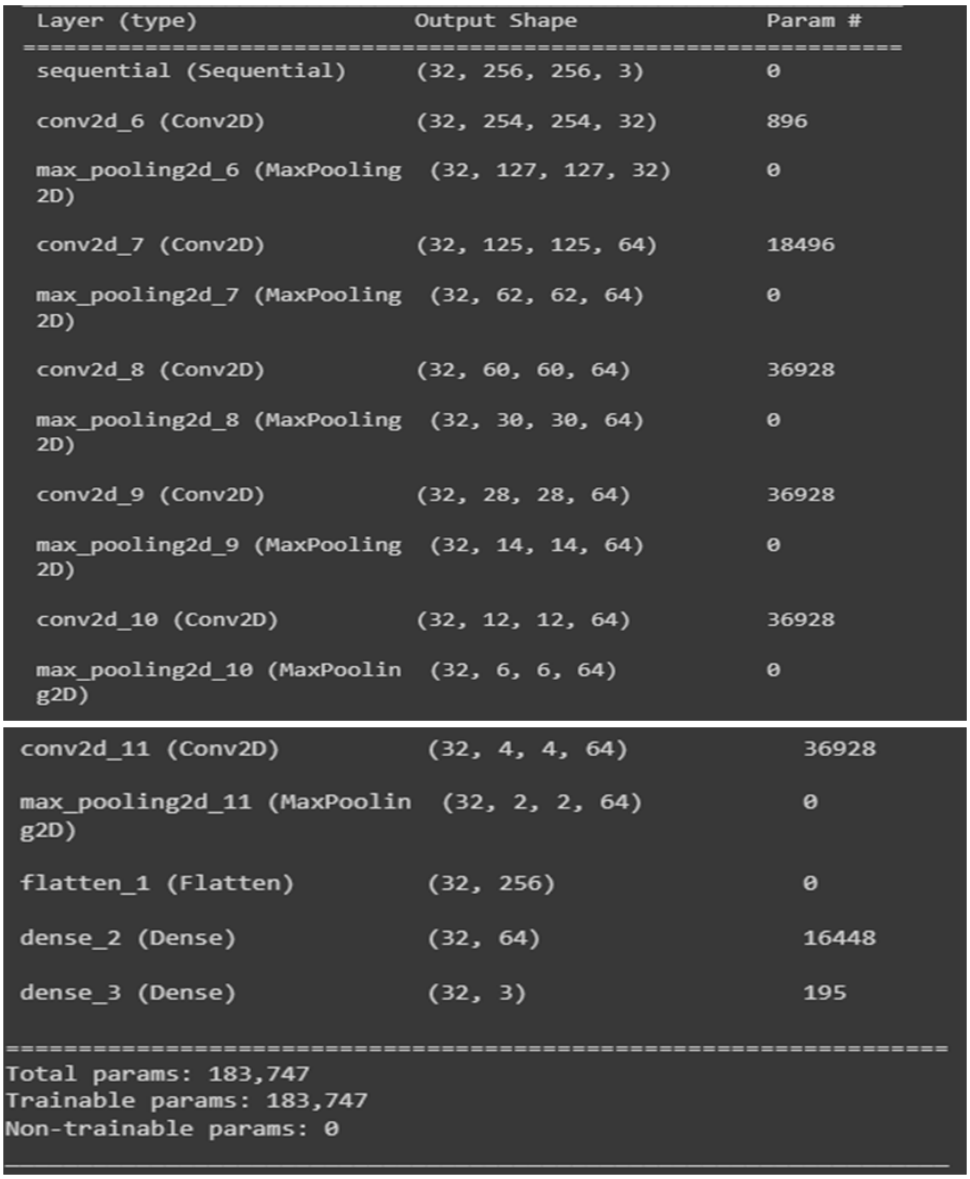
Chapter 4

Model Architecture

4.1 Detailed architecture of the Neural Network

During training, the input to our ConvNets is a fixed-size 224 × 224 RGB image. The only preprocessing we do is subtracting the mean RGB value, computed on the training set, from each pixel. The image is passed through a stack of convolutional (conv.) layers, where we use filters with a very small receptive field: 3 × 3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations we also utilize 1 × 1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1 pixel for 3 × 3 conv. layers. Spatial pooling is carried out by 5 max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is carried out over a 2 × 2 pixel window, with stride 2. A stack of convolutional layers (which has a different depth in different architectures) is followed by three Fully-Connected (FC) layers. All hidden layers are equipped with the rectification (ReLU (Krizhevsky et al., 2012)) non-linearity. **[6]**

Here is an overview of the architecture:



Layer (type)	Output Shape	Param #
sequential (Sequential)	(32, 256, 256, 3)	0
conv2d_6 (Conv2D)	(32, 254, 254, 32)	896
max_pooling2d_6 (MaxPooling 2D)	(32, 127, 127, 32)	0
conv2d_7 (Conv2D)	(32, 125, 125, 64)	18496
max_pooling2d_7 (MaxPooling 2D)	(32, 62, 62, 64)	0
conv2d_8 (Conv2D)	(32, 60, 60, 64)	36928
max_pooling2d_8 (MaxPooling 2D)	(32, 30, 30, 64)	0
conv2d_9 (Conv2D)	(32, 28, 28, 64)	36928
max_pooling2d_9 (MaxPooling 2D)	(32, 14, 14, 64)	0
conv2d_10 (Conv2D)	(32, 12, 12, 64)	36928
max_pooling2d_10 (MaxPoolin g2D)	(32, 6, 6, 64)	0
conv2d_11 (Conv2D)	(32, 4, 4, 64)	36928
max_pooling2d_11 (MaxPoolin g2D)	(32, 2, 2, 64)	0
flatten_1 (Flatten)	(32, 256)	0
dense_2 (Dense)	(32, 64)	16448
dense_3 (Dense)	(32, 3)	195

Total params: 183,747
Trainable params: 183,747
Non-trainable params: 0

Figure 4.1: VGG16 model resume

overall we have 183.747 trainable parameters.

CNNs are a valid choice for this task because of the following advantages :

- Detects disease in plants that show distinct visual symptoms by capturing spatial features: CNNs are excellent at evaluating photos and identifying spatial relationships.
- Discover important visual patterns linked to various diseases by automatically learning hierarchical representations of the input data using CNNs that can recognize hierarchical features.

- Handle variances: CNNs generalize well in real-world contexts and are robust to variations in color, shape, size, and lighting conditions frequently encountered in plant photos.
- Gain from transfer learning: CNN models that have been previously trained on extensive datasets can be adjusted for plant disease diagnosis, utilizing newly learnt features and enhancing performance with little training data.
- Scalability and effectiveness are enabled. CNNs are well suited for automated disease detection systems across huge plant populations or fields because they can process images efficiently in real-time.

Taking into account these benefits, CNNs present a potent and dependable method for studying plant photos, extracting useful features, and precisely classifying the presence of diseases.

4.2 Testing with another model:MobileNet

Convolutional neural network (CNN) architecture called MobileNet is created specifically for mobile and embedded devices' need for quick and easy picture classification. It was first mentioned in the study "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications" by Andrew G. Howard and colleagues.

By balancing model accuracy and computational efficiency, MobileNet's main goal is to enable real-time picture categorization jobs on devices with limited resources. Depthwise separable convolutions are used to do this, which lowers computational complexity while maintaining model performance.

Convolutional layers are the first step in the MobileNet architecture, which is then followed by batch normalization and ReLU activation functions. A stack of depthwise separable convolutional layers is added after the network's initial conventional convolutional layer.

The MobileNet convolutional neural network (CNN) architecture was developed primarily to meet the demands of embedded and mobile devices for quick and simple image classification. In the study "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications" by Andrew G. Howard and colleagues, it was first brought up.

The fundamental objective of MobileNet is to enable real-time picture categorization tasks on devices with constrained resources by balancing model accuracy and computational efficiency. This reduces the computational complexity while preserving model performance and uses depthwise separable convolutions.

The MobileNet design starts with convolutional layers, then moves on to batch normalization and ReLU activation procedures. The network's first standard convolutional layer is followed by a stack of depthwise separable convolutional layers. [**7**]

In order to use the model we uploaded the layers using tensorflow library. However, 2 more layers are to be added manually: A global average pooling layer and a softmax

layer. The GlobalAveragePooling2D layer should be added before the softmax layer for two reasons:

- Dimensionality reduction: The network becomes more efficient and compact in terms of memory use and computational complexity by decreasing the spatial dimensions to a set length. This is particularly advantageous for embedded and mobile devices with constrained resources.
- Bringing together global data: The global average pooling procedure brings together data from the whole feature map, bringing out the key elements of the image. By doing so, it is possible to gain a more complete representation of the image's information, which enables the following softmax layer to generate predictions that are more well-informed.

Here is a resume of the new model incorporating the mobilenet architecture:

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Funct  (None, 8, 8, 1280)       2257984
 ional)

 global_average_pooling2d (G  (None, 1280)             0
 lobalAveragePooling2D)

 dense_2 (Dense)             (None, 3)                 3843

=================================================================
Total params: 2,261,827
Trainable params: 3,843
Non-trainable params: 2,257,984
_________________________________________________________________
```

Figure 4.2: MobileNet model Resume

As we observe the number of parameters increased reaching 3.843 trainable parameters and 2,257.984 non trainable parameters.

Chapter 5

Training and Evaluation

5.1 Training Process

The following code snippet covers the training process :

```
[ ]  model.compile(
        optimizer='adam',
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
        metrics=['accuracy'])
```

```
[ ]  checkpoint_path = "/content/drive/MyDrive/PPP/cp.ckpt"
     checkpoint_dir = os.path.dirname(checkpoint_path)

     # Create a callback that saves the model's weights
     cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                      save_weights_only=True,
                                                      verbose=1)

     history = model.fit(
         train_ds,
         batch_size=BATCH_SIZE,
         validation_data=val_ds,
         verbose=1,
         epochs=10,
         callbacks=[cp_callback]
     )
```

Figure 5.1: Training code

We are going to breakdown the training process step by step to fully grasp the operation.

- **Model Compilation:**The first step of training is compiling the model. The model.compile line prepares the model for training. The Adam optimizer is chosen as the optimizer. Adam is a popular optimization algorithm known for its efficiency in training deep learning models. It adjusts the learning rate adaptively

while training, According to Kingma et al., 2014, the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".[8]

- **Loss function :**The loss function, specified as

  `tf.keras.losses.SparseCategoricalCrossentropy`

  ,measures how well the model is performing during training. It measures the difference between the generated output and the actual target labels. The Sparse categorical Crossentropy is designed for multi-class classification tasks.

- **Hyperparameters:** The training process itself is defined by hyperparameters. Hyperparameters like batch size, number of epochs, and route to save model checkpoints are defined in the given code.
  - The batch size (BATCH-SIZE) controls how many training pictures are treated in an iteration.
  - The model's total number of iterations within the whole the training set is specified by the number of epochs. A full run of the dataset, updating the model's weights, is represented as an epoch.
  - The checkpoint path determines the location in which we save the training weights. This step is crucial in case the training was interrupted as it helps us to resume from the last epoch where it stopped or even reuse the model later.

- **Training**: The training is started using the fit method. It trains the model on the provided training set (train-ds) with the specified hyperparameters. The validation dataset (val-ds) can be used to evaluate the model's performance during training. During training, the model updates its weights by minimizing the specified loss function using the optimizer. The training data is processed in batches, with each batch contributing to weight updates. This process repeats for the specified number of epochs, allowing the model to learn from the data and improve its performance over time.

5.2 Training Results

5.2.1 Results of VGG16 architecture

In order to measure the model performance, we are going to count on the accuracy metric. After training the model for 10 epochs, we plot the resulting loss and accuracy functions.

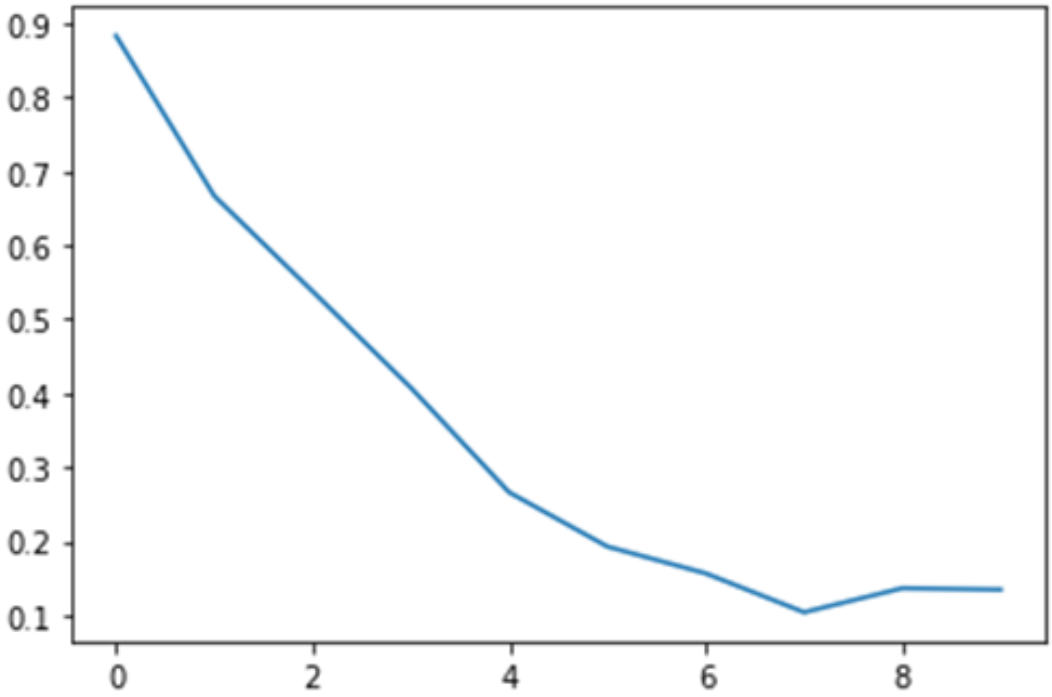


Figure 5.2: VGG16 Loss Function

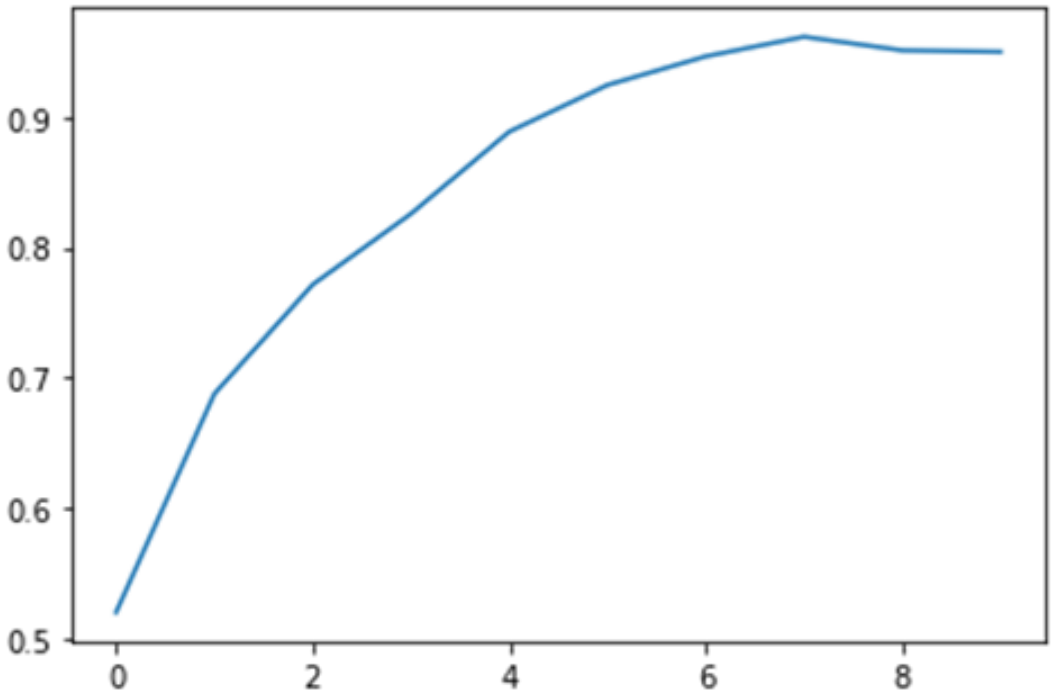


Figure 5.3: VGG16 Accuracy

Overall, the model achieved 0.96 accuracy.

To further evaluate the model, we took samples from our test data set and we printed the predicted label alongside the prediction accuracy. The following results are displayed:

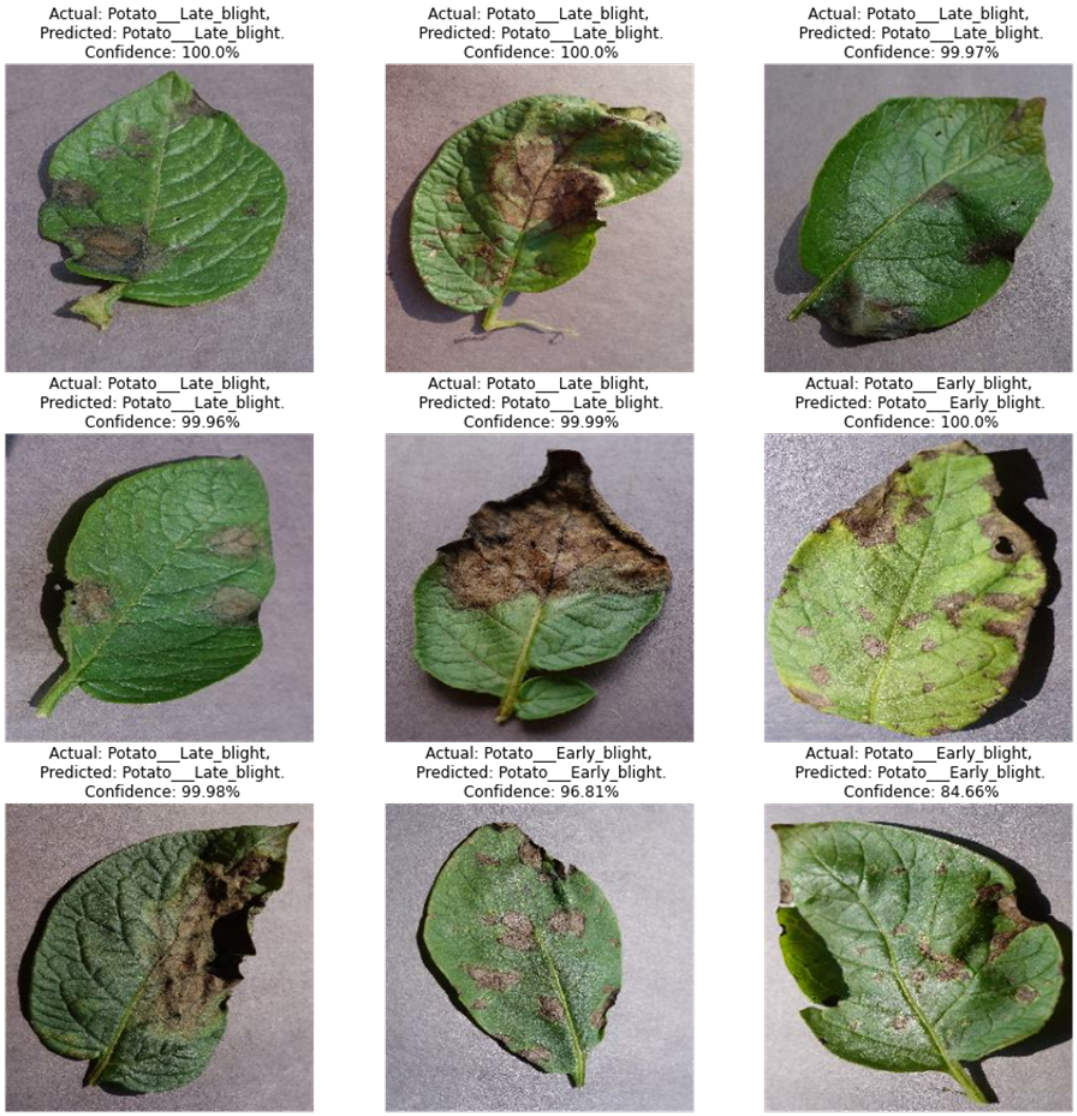


Figure 5.4: VGG16 Prediction Results

5.2.2 Results of the MobileNet architecture

We trained the model for 10 epochs; the model reached an accuracy of 0.974. To further evaluate the model, we took samples from our test data set and we printed the predicted label alongside the prediction accuracy. The following results are displayed:

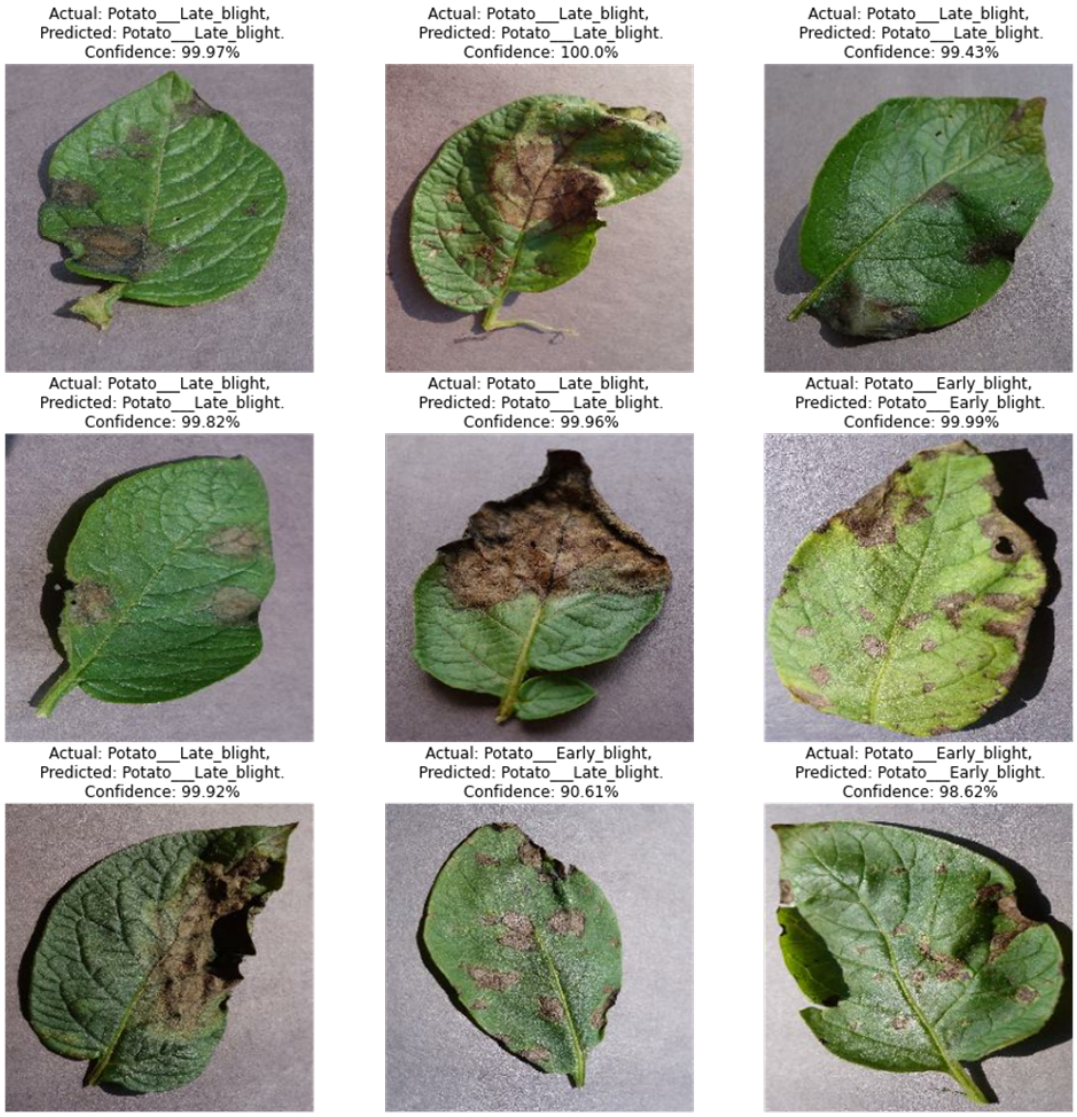


Figure 5.5: MobileNet Prediction Results

5.2.3 Comparison between the two models

We notice that the mobile net architecture performs slightly better than VGG16.

MobileNet performs better than VGG16 in certain scenarios due to its lightweight. MobileNet is specifically designed for mobile and embedded devices with restrained resources. As a result, MobileNet has faster inference times, lower latency, and lower memory requirements compared to VGG16. While MobileNet excels in efficiency and speed, VGG16 provides better results when complex feature representations and deeper abstraction are needed.

Overall, our problem is considered a relatively simple problem and the choice between VGG16 and mobileNet doesn't impact drastically the accuracy.

Chapter 6

Transfer learning

6.1 Explaining the principle behind transfer learning

Transfer learning aims at improving the performance of target learners on target domains by transferring the knowledge contained in different but related source domains. In this way, the dependence on a large number of target domain data can be reduced for constructing target learners. Due to the wide application prospects, transfer learning has become a popular and promising area in machine learning.[**11**]
The large scale applications of transfer learning has contributed to its growing popularity and recognition as a promising area of research in machine learning. Its potential impacts various domains, including computer vision, natural language processing, and speech recognition, where data scarcity or the high cost of data acquisition hinder the development of accurate models. Transfer learning opens up opportunities to make use pf existing knowledge to overcome these challenges and achieve superior performance in target domains.
To evaluate the model's capability to identify late blight and early blight in various plant species, we applied a transfer learning approach. By utilizing the pre-trained weights from our previous neural network specifically designed to detect these diseases in potatoes, we successfully extended its applicability to encompass the detection of early blight and late blight in tomatoes.

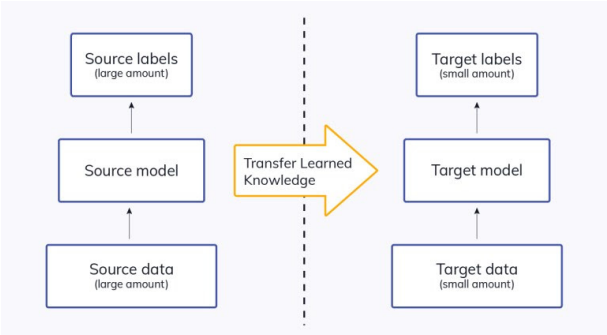


Figure 6.1: Transfer Learning

6.2 Testing the Transferability of the Model using a Tomato Database

Transfer learning is applied to evaluate the transferability of the pre-trained potato model to a tomato dataset. The following steps are performed:

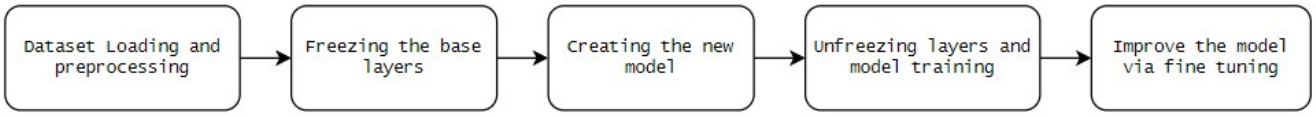


Figure 6.2: Transfer-learning steps

6.2.1 Dataset Loading and Preprocessing

The tomato dataset is loaded using the `image_dataset_from_directory` function. This function reads the images from the specified path, shuffles the data, resizes the images to a specified size, and batches the data for efficient processing.

```
dataset=tf.keras.preprocessing.image_dataset_from_directory(
    path,
    shuffle=True,
    image_size=(IMAGE_SIZE,IMAGE_SIZE),
    batch_size=BATCH_SIZE
)
```

The loaded dataset is partitioned into training, validation, and testing sets using the `get_dataset_partitions_tf` function. This function supports customizable split ratios and can shuffle the data before partitioning.

```
get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1,
                              test_split=0.1, shuffle=True,
                              shuffle_size=10000)
```

The training, validation, and testing datasets are preprocessed and cached for efficient training.

```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
val_ds = val_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds = test_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
```

6.2.2 Freezing the base layers

The pre-trained potato model is used as the base model for transfer learning. Its layers are frozen by setting `trainable=False`.

```
base_model=model
base_model.trainable=False
```

6.2.3 Creating the new model

A new model is created by adding a global average pooling layer and a softmax prediction layer on top of the base model.

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
prediction_layer = tf.keras.layers.Dense(3,activation='softmax')

new_model=models.Sequential([
    base_model,
    global_average_layer,
    prediction_layer
])

input_shape=(32,256,256,3)
new_model.build(input_shape=input_shape)
new_model.summary()
```

The new model is compiled with the appropriate optimizer, loss function, and metrics.

```
base_learning_rate = 0.001
new_model.compile(
  optimizer=tf.keras.optimizers.RMSprop(learning_rate=base_learning_rate),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
  metrics=['accuracy']
)
```

6.2.4 Unfreezing layers and model training

The softmax layer in the new model is unfrozen, and the model is trained for a specified number of epochs.

```
my_layer = new_model.get_layer('sequential_2')#unfreezing the softmax layer
my_layer.trainable = True
```

By following these steps, the transferability of the pre-trained potato model to the tomato dataset is evaluated. The model is trained and fine-tuned using transfer learning techniques, leveraging the knowledge learned from the potato model to improve performance on tomato images.

After training the model for 5 epochs, we achieve an accuracy of 0.96. We took several pictures for testing, we got the following results:

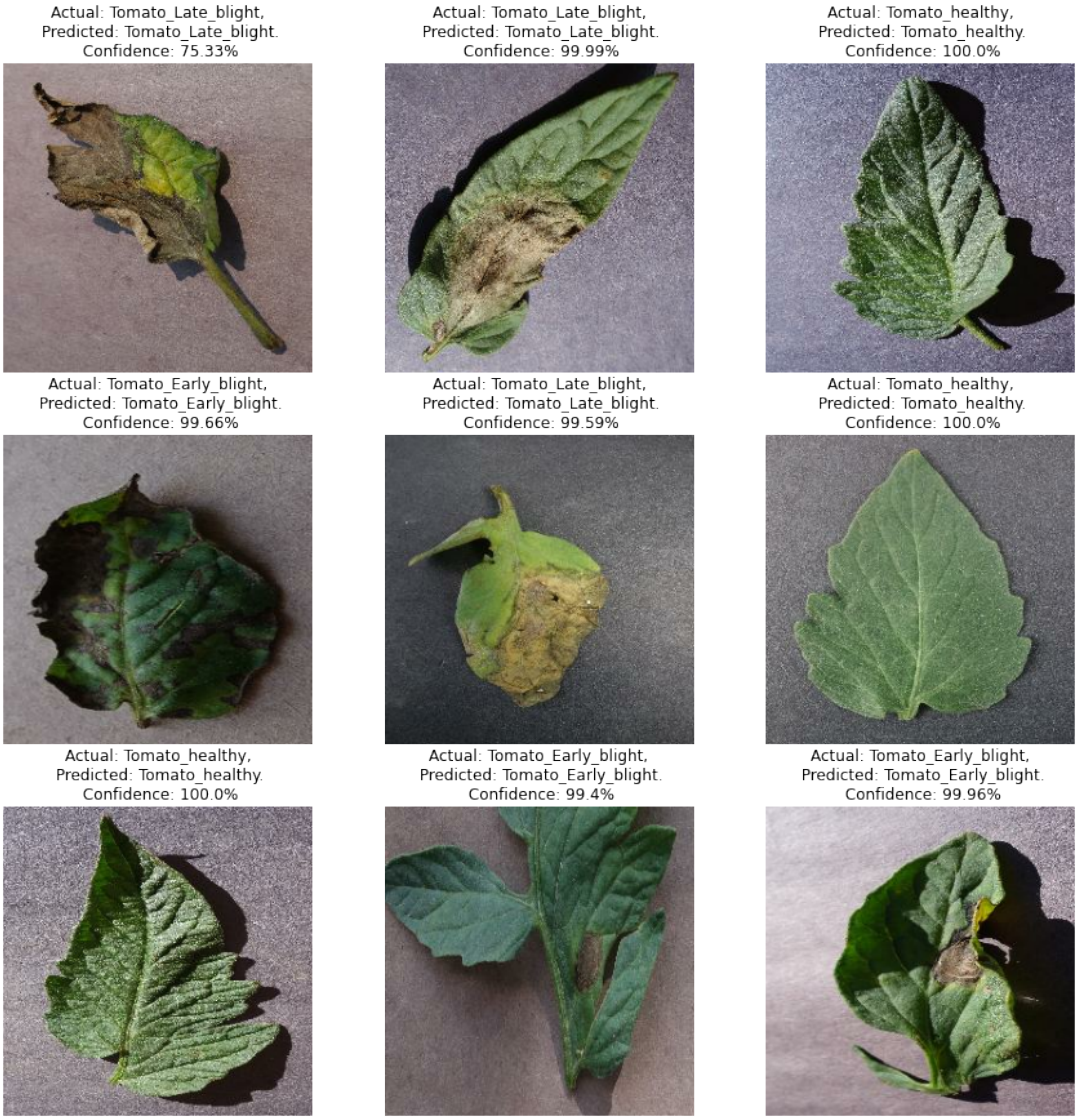


Figure 6.3: Tomato results

Chapter 7

User Interface

7.1 Implementation of FastAPI

7.1.1 What is API?

An application programming interface (API) is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software. A document or standard that describes how to build or use such a connection or interface is called an API specification. A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.

In contrast to a user interface, which connects a computer to a person, an application programming interface connects computers or pieces of software to each other. It is not intended to be used directly by a person (the end user) other than a computer programmer who is incorporating it into the software. An API is often made up of different parts which act as tools or services that are available to the programmer. A program or a programmer that uses one of these parts is said to call that portion of the API. The calls that make up the API are also known as subroutines, methods, requests, or endpoints. An API specification defines these calls, meaning that it explains how to use or implement them.[**9**]

7.1.2 General Presentation of FastAPI

FastAPI is a modern Python web framework, very efficient in building APIs. It is based on Python's type hints feature that has been added since Python 3.6 onwards. It is one of the fastest web frameworks of Python.

- As it works on the functionality of Starlette and Pydantic libraries, its performance is amongst the best and on par with that of NodeJS and Go.
- In addition to offering high performance, FastAPI offers significant speed for development, reduces human-induced errors in the code, is easy to learn and is completely

production-ready.

- FastAPI is fully compatible with well-known standards of APIs, namely OpenAPI and JSON schema.**[10]**

7.1.3 Implementation

The organigram presented in this project report illustrates the function of the FastAPI script :

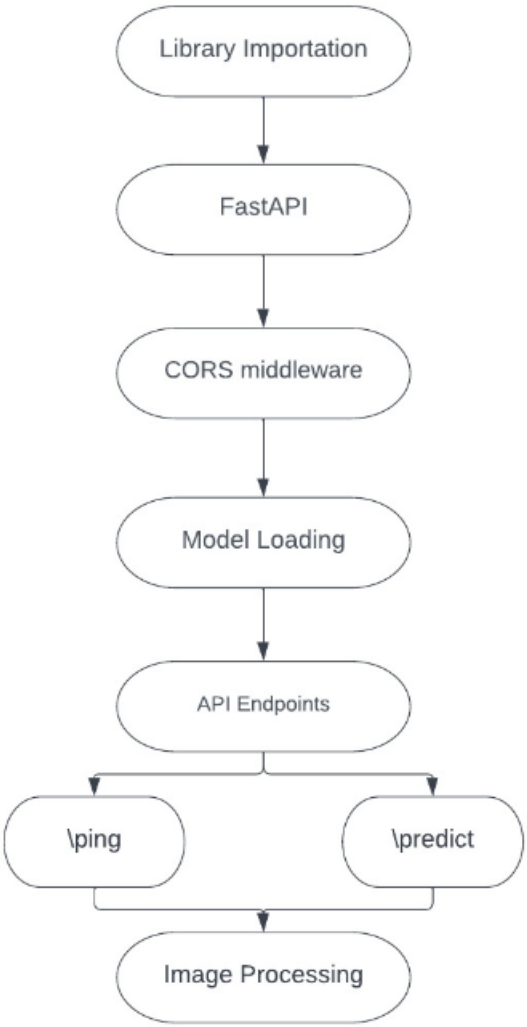


Figure 7.1: Organigram of FastAPI script

The major elements and connections among them in the code are represented by the organigram:

- FastAPI is the primary framework used to create the API.

- Handles cross-origin resource sharing with CORS middleware.
- Model Loading : Loads the tensorFlow model.
- /ping and /predict endpoints are included in the list of API endpoints.
  - /ping Endpoint: Manages HTTP GET and HTTP POST requests for the /ping path.
  - /predict Endpoint: Manages HTTP GET and HTTP POST requests for the /predict path.
- Image Processing: Consists of operations to read and modify the uploaded image.

7.2 Web page implementation

7.2.1 Introduction

This web page simplifies the use of the Deep Learning model for farmers, all they have to do is import an image and the communication is done implicitly finally a result is displayed in the web page.

The local web page is developed using a combination of HTML,CSS and JavaScript with react as platform, establishes a seamless line of communication with Deep Learning model via FastAPI.

7.2.2 Structure and Layout

The web page has a well-organized and visually appealing layout, with clear sections that are thoughtfully arranged. To better understand its structure and design, refer to the accompanying image that provides an overview of the webpage's layout.

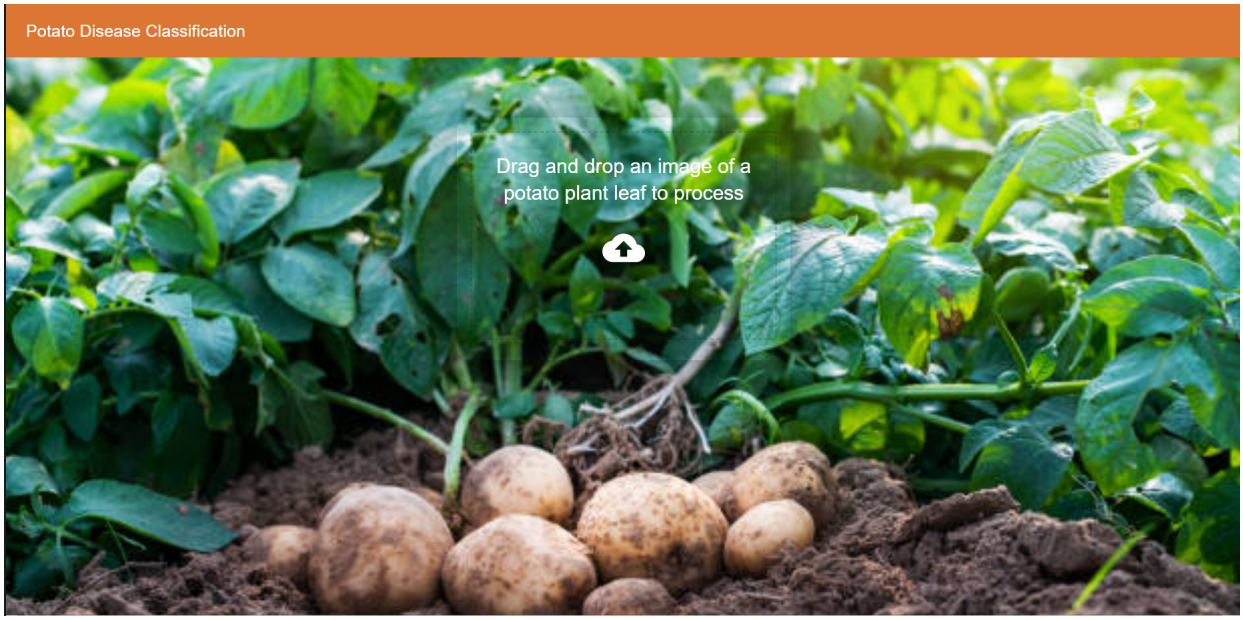


Figure 7.2: Frontend page(user Interface)

The front end page features includes a drag and drop area specifically designed for importing images.This functionality allows users to upload images by simply dragging and dropping them into the specific area.

7.2.3 Communication between frontend and model

The communication between the local web page and the Deep Learning model is facilitated by FastAPI which is a python web framework. FastAPI serves as the intemidiary between the frontend and the backend.Here is a description of the communication process:

- User Interaction :the web page allows the users to upload the picture of potato leaf for prediction.
- API Request : when the users makes an action like dropping image , the local web page sends an HTTP request to the FastAPI. The request contains the uploaded image file.
- FastAPI Routing: FastAPI receives the API request and routes it to the appropriate endpoint which is "/predict".
- Data processing: when the request is received, FastAPI perform operations like reading the uploaded image file and converting the image into a suitable format for the Deep Learning model.
- Deep Learning Model Prediction : FastAPI interacts with DL model to perform predictions on the precessed image data , this includes passing the data to the model and executing the model and obtaining the predicted class and confidence scores.

- API Response: FastAPI constructs a response in json format containing the prediction results such as the predicted class and confidence scores.The response is sended to the local web page as an HTTP response
- Displaying Results : The local web page receives the API response from FastAPI and display the predicted class and confidence scores to the user.

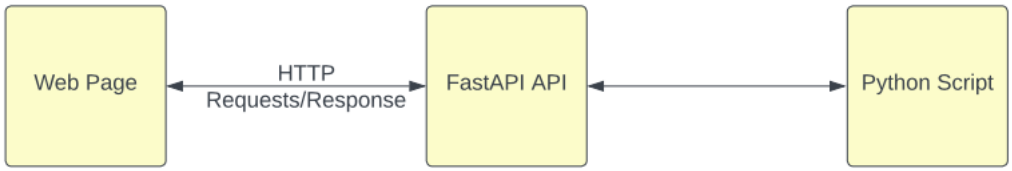


Figure 7.3: Communication Diagram

Chapter 8

Results and discussion

8.1 Results of the Web application

After executing the FastAPI code that deploys the model into the web page we are able to extract testing pictures to evaluate the system. This are some examples of the executions:

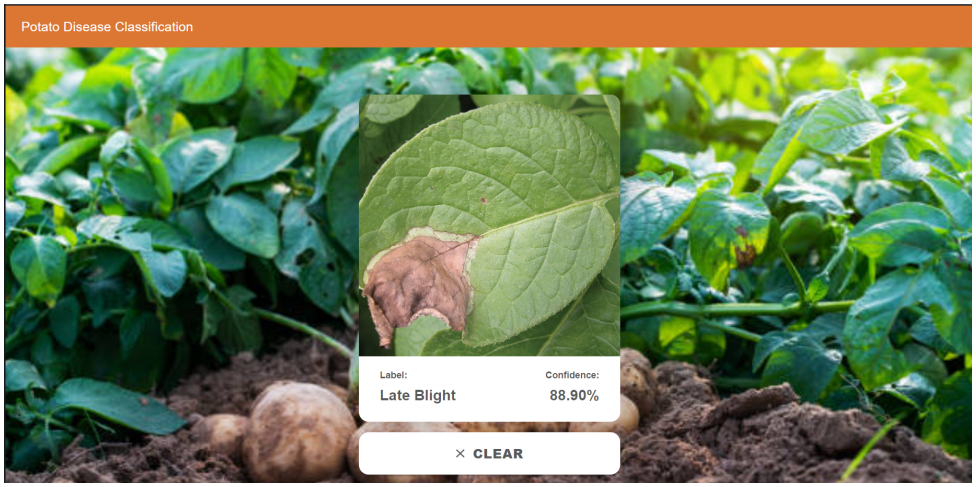


Figure 8.1: Late Blight result

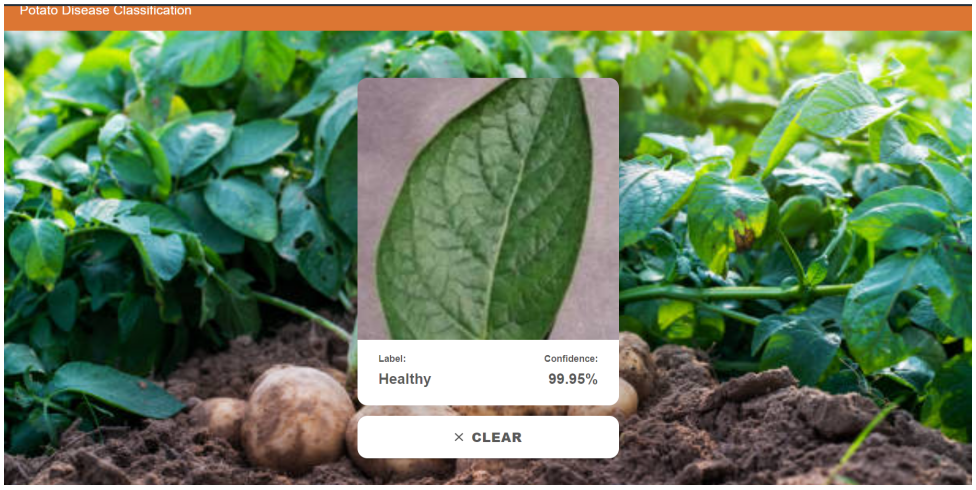


Figure 8.2: Healthy result

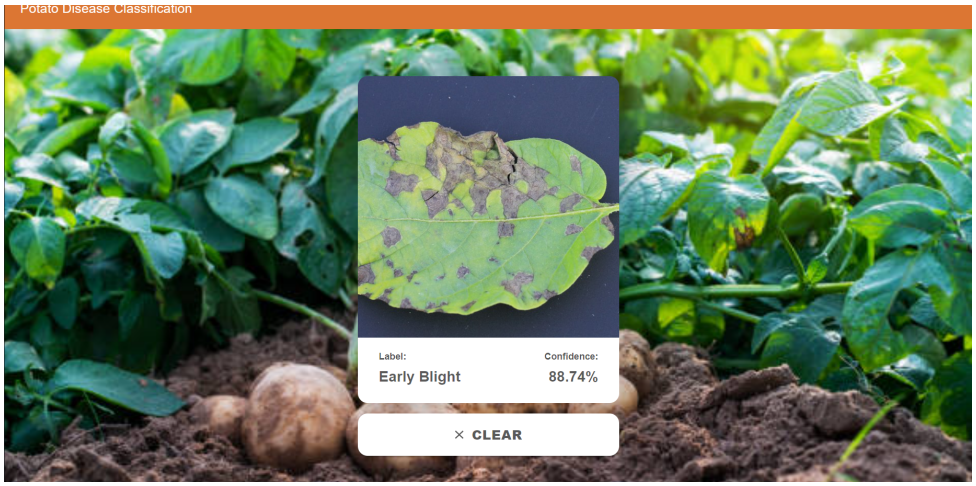


Figure 8.3: Early Blight result

8.2 Analyzing the strengths, weaknesses, and potential areas for improvement of the system.

Our system that uses Fastapi to deploy a model on a website has multiple strengths, weaknesses, and potential areas for improvement. Their analysis gives us the following results :

- Advantages:
  - Precision: the model achieved high level accuracy when identifying early blight and late blight. This accuracy enhances the system's dependability.
  - FastAPI Integration: FastAPI has high performances capabilities. This ensures efficient handling of HTTP requests and responses, resulting in a smooth user experience.

  - Scalability: Both FastAPI and Node.js frameworks offer scalability, enabling the system to handle an extensive volume of concurrent requests and cater to a growing user base effectively.

- Limitations:
  - Training Data Quality: The accuracy and reliability of the model heavily rely on the quality and diversity of the training data. Incomplete, biased, or unrepresentative training data may adversely affect the model's performance.
  - Potential for False Positives or Negatives: False positives (identifying healthy plants as diseased) or false negatives (failing to detect actual disease symptoms) can occur, impacting the effectiveness of the system and users' trust in its results.
- Areas for Enhancement:
  - Diverse Training Data: Enhancing the model's performance and generalization can be accomplished by incorporating a wider range of plant species and variations into the training data. This enriches the model's knowledge of robust and representative features.
  - Continuous Model Updating: Regular updates to the model using new data and advancements in disease detection techniques ensure improved accuracy and keep pace with evolving patterns and variations in blight diseases.
  - Enhancing the interactivity of the web page involves incorporating additional features while leveraging TF Serving to integrate multiple models simultaneously. This integration empowers the web page with the capability to upload models for different plant species, enabling the selection of the appropriate model for making predictions.

Chapter 9

Conclusion and Future Work

In this project, our main objective was to develop a deep learning model based on Convolutional Neural Network (CNN) architecture for the accurate detection of late blight and early blight in plants. The significance of plant disease detection in relation to crop yield and food security has been extensively discussed, emphasizing the critical need for reliable detection methods.

A comprehensive review of existing literature on plant disease detection using deep learning techniques was conducted. This review shed light on the benefits and limitations associated with the utilization of CNNs in this particular domain. We carefully curated a dataset for our project, consisting of diverse plant images from various sources. The dataset was subjected to meticulous preprocessing steps, including image resizing, normalization, and augmentation, to enhance the model's performance and robustness.

The CNN model architecture used in our project was meticulously designed, with each layer serving a specific purpose in capturing meaningful features from the input images. We made specific adaptations to the architecture to suit the unique challenges of plant disease detection. The model was trained using carefully selected hyperparameters, an appropriate optimizer, and a suitable loss function. Rigorous evaluation and testing were conducted, resulting in insightful performance metrics.

To evaluate the transferability of the model, we conducted experiments using a tomato database, which allowed us to assess its effectiveness in detecting diseases beyond the trained categories. The results of these experiments were thoroughly analyzed, providing valuable insights into the model's generalization capabilities.

In addition to model development, we implemented FastAPI to create a robust web API, enabling seamless deployment and integration of the trained model. Furthermore, a user-friendly Node.js website was developed to facilitate easy upload of plant images and obtain disease predictions in real-time.

In terms of future directions, there are several potential avenues for expanding the scope and enhancing the overall effectiveness of the plant disease detection system. These include:

- **Expand disease coverage:** Include more plant diseases to provide comprehensive support for farmers and agricultural professionals.
- **Optimize model architecture:** Refine the model by exploring advanced CNN architectures and fine-tuning hyperparameters.
- **Improve user interface:** Enhance the interface for easy usage and accessibility, with intuitive features and real-time feedback.
- **Develop mobile app:** Create a mobile application for convenient plant image capture, diagnosis, and recommendations on-the-go.

This project contributes to the overall goal of promoting sustainable agriculture, maximizing crop yield, and ensuring food security in the face of plant diseases.

Bibliography

[1] document "Plant Disease Detection Using Image Processing," BY S. D. Khirade and A. B. Patil(2015)https://ieeexplore.ieee.org/document/7155951

[2] document "Plant disease identification: A comparative study," by Shriroop. C. Madiwalar & M. V. Wyawahare, https://ieeexplore.ieee.org/document/8073478

[3] document "Plant Disease Detection Using Hyperspectral Imaging," by Peyman. Moghadam, Daniel. Ward, Ethan. Goan, Srimal. Jayawardena, P. Sikka and E. Hernandez,https://ieeexplore.ieee.org/document/8227476

[4] document "Image based Plant Disease Detection in Pomegranate Plant for Bacterial Blight," by Sharath D.M.& Akhilesh & S. Arun Kumar Rohan M.G.and Prathap C. https://ieeexplore.ieee.org/document/8698007

[5] Plant Village https://www.kaggle.com/datasets/arjuntejaswi/plant-village

[6] VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION Karen Simonyan & Andrew Zisserman https://arxiv.org/pdf/1409.1556.pdf

[7] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications , google inc.https://arxiv.org/pdf/1704.04861.pdf

[8] Kingma et al., 2014https://arxiv.org/pdf/1406.5298.pdf

[9] API https://en.wikipedia.org/wiki/API

[10] FastAPI documentation https://www.tutorialspoint.com/fastapi/fastapi_quick_guide.html

[11] Comprehensive Survey on Transfer Learning https://arxiv.org/pdf/1911.02685.pdf