CHAPTER 1

INTRODUCTION

Medicinal plants are gaining popularity due to the health benefits, and preventive properties they provide. Naturally, they are abundant and can grow in various environments, but many of those remain unidentified due to lack of knowledge. Proper identification is crucial to harness their medicinal properties, which are vital for human health and disease prevention [1]. Recognition of plant species from images is challenging due to a large variety of plants, difference in orientation, viewpoint, and background [2]. According to Botanical Survey of India (BSI), around 8000 medicinal plant species were identified in India and the Medicinal Pharmacopoeia of India records more than 350 plant species that are utilized in the preparation of herbal medicines, highlighting the importance of proper identification to tap their medicinal value.

Recent developments in artificial intelligence, combined with the worldwide spread of smartphones, have transformed solutions in agriculture, medicine, and education, making intelligent vision-based applications capable of solving real-world issues [12]. Identifying herbal plants can contribute to the development of Medicinal medicine and boost the Economy. People can also grow these herbal plants in their households, which can be beneficial as natural remedies instead of completely depending on medicines. Smart phone cameras, which are now ubiquitous, provide a valuable tool for the automation of Medicinal plant species recognition, raising public awareness, assisting sustainable cultivation, and facilitating pharmaceutical manufacture [18]. Such technologies enable users to investigate the pharmacological potential of plants found around them, going hand in hand with the emerging trend for cheap healthcare options.

Inspite of such opportunities, it is extremely challenging to classify Medicinal plants from photo-based images. The changes in illumination, complicated backgrounds, occlusions, out-of-focus images, intra-class variability within the same species, and inter-class similarity between different species tend to produce high misclassification rates [22]. The vastness of Medicinal plants with minute morphological differences also make automatic identification challenging, especially when the images are obtained from low-resolution smartphone images in uncontrolled environments. Although more general applications such as PlantSnap and PlantNet succeed with images from high-quality cameras, issues persist in more complex environments, medicinal varieties, low-resolution inputs, or more simplistic plant features [18]. Eliminating these issues is vital in order to create strong, user-friendly applications to suit Medicinal plant recognition.

The literature on the classification of plant species provides precious data, but there is a lacuna in classifying Medicinal plants in particular. For instance, one research carried out fused texture, run-length matrix, and multispectral features from leaves of six medicinal plants (e.g., Tulsi, Pepper, Mint) with a splendid 99.01% accuracy using a Multilayer Perceptron classifier [12]. Another study dealt with plants such as Papaya and Neem using color and texture attributes with Support Vector Machine (SVM) and Radial Basis Exact Fit Neural Network classifiers achieving 90% accuracy [18]. Although these studies show the effectiveness of machine learning in plant classification, they emphasize controlled datasets, non-medicinal plants, or small sets of plants, little tackling real-world issues such as inter-class similarities or poor-quality images common in Medicinal plant identification. This work proposes Deep-Medic, a new vision-based application to classify Medicinal plant species, encouraging their medicinal use and cultivation via smartphone technology. Building on the shortcomings of existing research, Deep-Medic seeks to provide high accuracy in classification under various conditions. The work makes the following main contributions:

- **Whole-Plant Dataset:**
  A collection of whole-plant images with 40 Medicinal plant species of high medicinal importance, including eight groups showing inter-class similarities. Taken with smartphones of different resolutions, this dataset represents real-world variability. Additionally, we created a dataset which consists of multi-plant images i.e., multiple plant species in each image.

- **Deep-MedicV1 model:**
  A light-weight deep convolutional neural network, Deep-Medic was built, trained, and validated to identify Medicinal single plant images. In order to deal with issues such as separating green target areas from intricate green backgrounds, a segmentation model is utilized to separate meaningful plant parts, improving model performance [1].

- **Deep-MedicV2 model:**
  YOLO (You Only Look Once) and RetinaNet are the object detection models widely used for multi-object tasks, in this case multiple plant species identification. YOLO predicts bounding boxes and class labels in a single forward pass, making it efficient for high speed and real-time performance, but struggle to detect small or overlapping plants. To overcome this, RetinaNet is introduced that addresses class imbalance and more accurate in detecting small or overlapping plants.

By working on actual real-time problems and utilizing deep learning, this research aims to improve the automatic identification of Medicinal plants, ensuring their conservation and incorporation into contemporary healthcare systems. The rest of the paper is structured as follows: Section 3 presents a review of the literature, Section 4 outlines the dataset, Section 5 introduces the Deep-Medic model, and Section 6 reports experimental results.

CHAPTER 2

LITERATURE SURVEY

Variety of methods have been taken as a contribution to the literature for plant species recognition, which primarily depends on entire image to be processed without removing complex backgrounds, and occlusions. A solution is to have direct access on plant leaf regions, that makes simple to extract the most distinguishable features based on their shape, color, texture, and vein patterns, on which further classifications were accomplished. This study's literature review categorizes the approaches based on machine learning and deep learning.

2.1. Machine Learning approaches

In work by [3] have done a leaf recognition of different medicinal plant species using a robust image processing algorithm and artificial neural networks classifier. Here the model struggles with inter-class similarities, which leads to misclassify closely related plant species. Next, [4] proposed techniques based on artificial intelligence to identify diseased leaves from the input of leaf images. Here the challenges associated with photo-based inputs are not addressed and focused on limited plant species.

A Weighted KNN model by [5] is proposed to extract morphological features from the plant. Limited to the Folio Leaf dataset, which may not generalize well to other datasets with different plant species. [6] proposed an intelligent medicinal leaf classification model using handcrafted and deep learning methods. This Model relies on only pretrained CNNs, limiting adaptability to new datasets without fine-tuning. The Classification of Medicinal Plants was done using Multispectral and Texture feature, a machine learning approach, by [12] worked on classification of medicinal plant leaves using various machine learning approaches. But this work is limited to only six varieties of medicinal plants leaves.

[18] has proposed classification of Indian medicinal plants using SVM, ANN, and RBENN classifiers and Color histogram for feature extraction. This work is limited to only 900 images across five species, it may not fully represent the diversity of medicinal plants in Ayurveda, limiting the model's applicability to a broader range of species. Next, plant species recognition by [22] used morphological features for feature extraction and multi-layer perceptron (MLP) with adaboost for classification and compared against KNN and decision tree classifiers. This work is limited to only Flavia dataset which may limit generalizability to other plant species or real-world conditions. The work done by [23] includes automatic classification of medicinal plant leaves for Chinese medicine by using SVM classifier for 12 different medicinal plant species.

2.2. Deep Learning approaches

In work by [2], Recognition of plant species was done on Folio, Swedish Leaf, Flavia, and Leaf12 datasets using deep learning VGG16, InceptionV3 architectures and categorized with different classifiers. The results had not clearly shown accurate recognition of leaf images. Next,

[7] have worked on segmentation and classification of leaf images with a complex background using deep learning VGG16 architecture. The dataset contains 1500 images of 15 different plant species, Mask R-CNN was used for segmentation and VGG16, VGG19, Inception ResnetV2 were compared based on metrics during classification. The results are accurate but the dataset has only limited plant species which does not contain inter-class similarities i.e., different plants appearing similar.

Subsequently, A deep ensemble learning model was used for automatic identification of medicinal leaves [15], in which a transfer learning approach used to extract the features from the ensemble of MobileNetV2, InceptionV3, ResNet50 and classified using softmax and dense layers. A higher accuracy was achieved, but it is only limited to dataset containing of high-resolution single leaf images with lack of complex backgrounds. Further, [11] have done olive leaf diseases classification using a deep hybrid learning model, in which EfficientNetB0 was combined with logistic regression classifier, limited to only one plant. A similar work [24] was carried out, where, DenseNet121 architecture was used to classify 29 different diseases on 7 plants, but limited to only non-medicinal variety of plant species.

Later, [1] A deep convolutional neural network named Ayur-PlantNet was used to classify Medicinal plants, the dataset contains 6000 images of 40 different plant species with inter-class similarities. Augmentation and segmentation were done during pre-processing and a lightweight CNN was trained on the dataset with fewer trainable parameters and less computational complexity compared to the existing pre-trained models VGG16, Resnet50, DenseNet121 etc., resulted in efficient classification on single plant images. Consequently, [10] A deep learning model to identify real-time medicinal plants, specifically limited to only six in number, was done using lightweight MobileNet deep learning model, uploaded to the cloud and an application was designed for real-time medicinal plant identification.

Another deep learning model which combines [8] Attention-based Enhanced Local and Global Features Network were used for medicinal leaf and plant classification. They used the dataset from Kaggle, consists of 80 classes of medicinal leaves and 40 classes of medicinal plants, merged the classes and done the classification, however the computational complexity is high. And [6] have worked on medicinal plants identification with the help of handcrafted feature descriptors using edge histograms, binary patterns to extract heterogenous leaf features. Deep Convolutional Neural Networks, a transfer learning approach, was also used for deep feature extraction resulted in accurate classification. Later, [19] A review was carried out for plant recognition and classification with the help of deep learning techniques namely Artificial, Probabilistic and Convolutional Neural Networks. Another [13] review was conducted on some medicinal herbs which are effective in the treatment of cancer. Although, patients are using synthetic medicines, they have an adverse effect on the human body and can cause side effects.

CHAPTER 3

DATA MATERIALS

3.1. DATA COLLECTION

The Indian Medicinal Leaves Dataset is an open-access repository available on Kaggle. It includes two folders i.e., Medicinal leaves of 80 different plant species and Medicinal plants of 40 different species. We utilized the Medicinal plants dataset of 40 different species which contains about 5200 images.

3.2. DATA AUGMENTATION

In the proposed method, Augmentation is performed to increase the size of dataset i.e., a balanced dataset with equal number of images in all classes. The image samples were considered with their variations concerning varying levels of contrast, blur, viewpoint, and orientations. This helps in reducing the underfitting issues during learning and prediction as the dataset acquired consists of imbalanced sample sizes of each class.

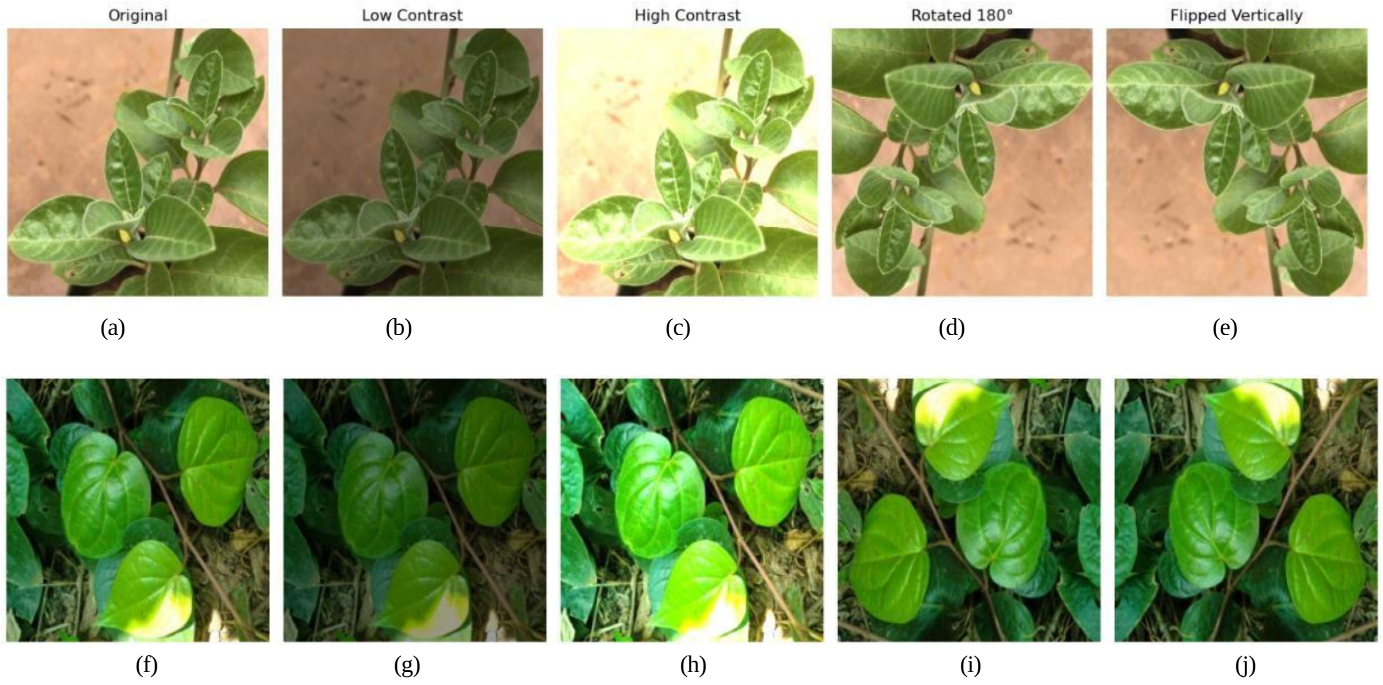


Fig. 1. Augmented samples Intensity and Geometric transformations

3.2.1. Intensity transformations: Let T be the intensity transformation applied on Image ‘I’ concerning r, g, and b channels, then the transformed image

$$I_t = \text{Concatenate}\ (T[I(r)], T[I(g)], T[I(b)])$$

$T[I(r)] = l_k \times \delta$ where $\delta < 1$ for low-contrast image
$T[I(g)] = l_k \times \delta$ where $\delta = 1$ for the original image

$$T[I(b)] = l_k \times \delta \text{ where } \delta > 1 \text{ for high-contrast image}$$

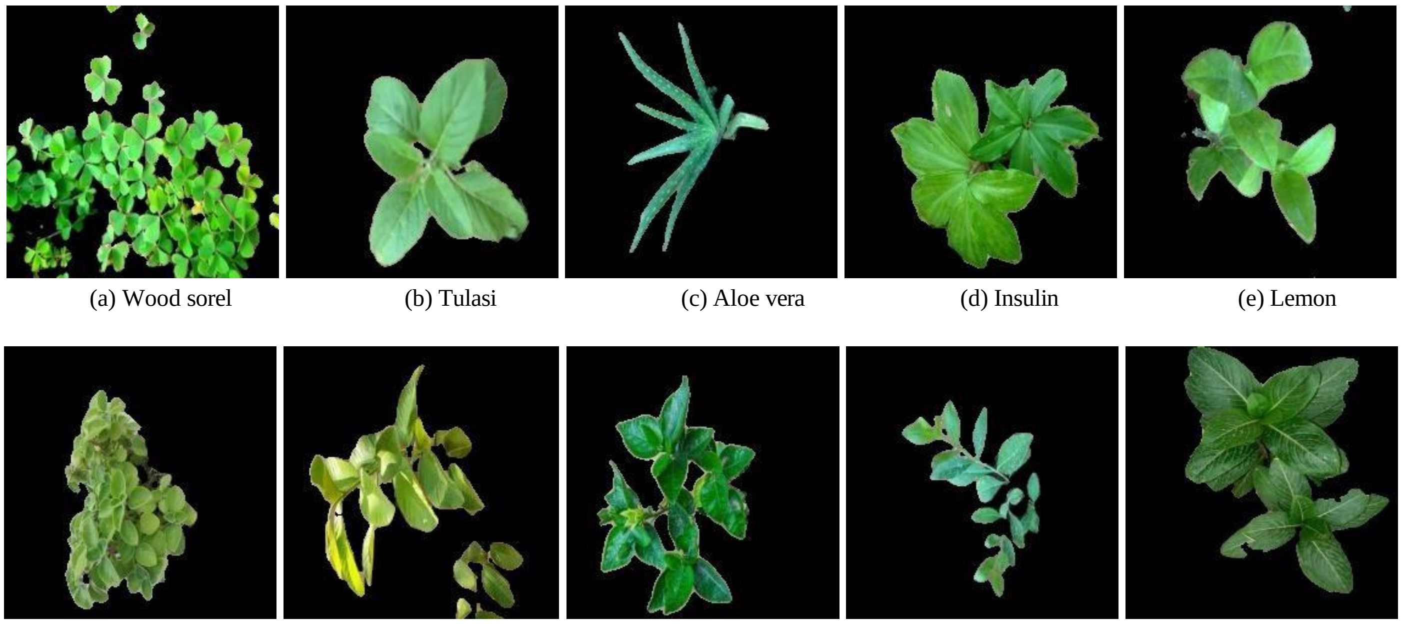
we use δ = 0.6 to produce low-contrast(dim) images and δ = 1.5 to produce high-contrast(bright) images.

3.2.2. Geometric transformations: This method uses two geometrical transformations, rotate and flip. Rotation produces an image rotated by 180 degrees, and another is images will flip in a vertical direction using OpenCV functions rotate() and flip() respectively.

$$I_r = R(I, 180^o)$$
$$I_f = F(I, d)$$

3.3. IMAGE SEGMENTATION

Segmentation is performed to extract only the green plant region while removing the background. This helps in reducing noise, occlusions, and unnecessary information, allowing the model to focus on the plant's structure for accurate classification. The segmentation is carried out using RGB-based color channel thresholding and morphological operations.



(a) Wood sorel (b) Tulasi (c) Aloe vera (d) Insulin (e) Lemon

(f) Doddapatre (g) Guava (h) Hibiscus (i) Henna (j) Nityapushpa

Fig. 2. Segmented sample images used for training

3.3.1. RGB-Based Color Channel Thresholding: Let S be the segmented image obtained from the input image by applying a color threshold in the HSV color space. The plant regions are identified using lower and upper HSV bounds:

$$S = \text{Mask}(I_{hsv}, L_{green}, U_{green})$$

Where: $L_{green} = [25,40,40]$ (Lower bound of green)

$U_{green} = [90,255,255]$ (Upper bound of green)

3.3.2. Morphological Operations: To refine the segmented plant region, morphological operations are applied to remove background noise and fill gaps in the plant region. The two main operations used are closing and opening:

3.3.2.1. Closing Operation (Mc) - Fills small holes in the segmented plant region

M_c = Close (M, K) where K is a 5×5 kernel.

3.3.2.2. Opening Operation (Mo) - Removes small background noise

M_o = Open (M_c, K)

3.3.3. Mask Application: Once the refined mask is obtained, it is applied to the original image to extract only the plant region, while the background is turned black:

$S_{final} = I_{rgb} \times M_o$

3.4. MULTI PLANT IMAGE FORMATION

We have taken images from different classes, combined for composite images creation in 2*2 grid layout of size 800*800 using OpenCV-python library. Generated a dataset which contains of1000 multi-plant images.

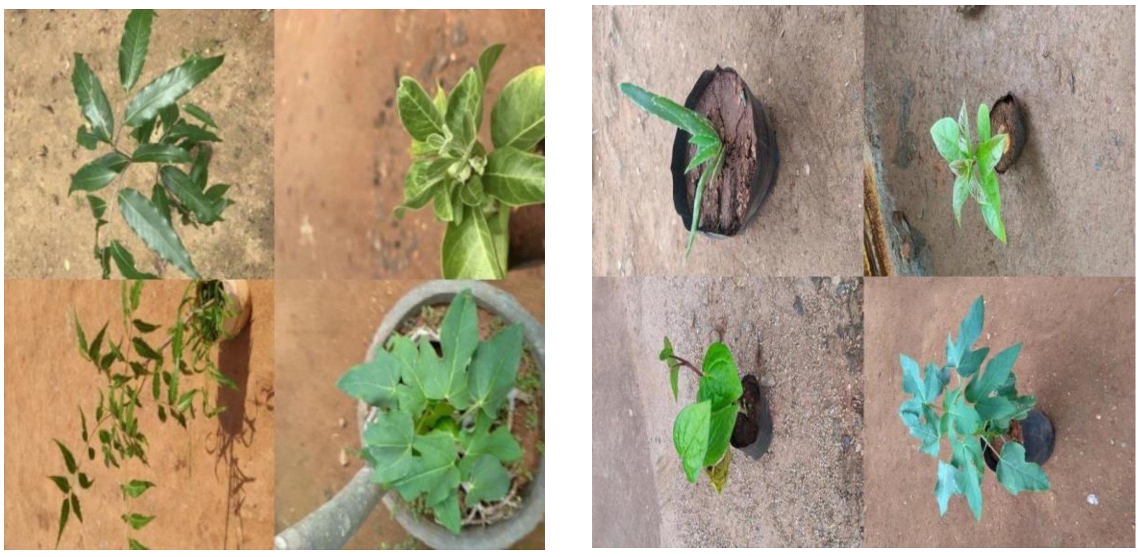

(a) (b)

Fig. 3. Samples of multi-plant images

CHAPTER 4

METHODOLOGY

4.1. DEEP-MEDIC ARCHITECTURE

Feature extraction is performed using deep feature learning blocks and convolutional layers. This step ensures that the model captures important patterns, textures, and structures from the segmented plant images, which are then used for classification. The feature extraction module consists of deep feature learning blocks, convolution layers, ReLU activation, and batch normalization.

4.1.1. Convolution Layer: The primary building block of a deep neural network is convolution. It is mainly used to extract features from an image. It applies a set of filters that slide over the input image to detect different patterns such as edges, textures, shapes, and structures.

4.1.2. Filter: Also known as kernel, filter is a small matrix of defined size which moves over the image to extract meaningful features. The size of filter should be defined e.g., 3*3, 5*5 and the depth is number of input channels used e.g., 3 for RGB. The more filters we use, more features will be extracted.

4.1.3. Stride: Responsible for controlling how much the filter should slide over the image, stride of 1 means, the filter is moving one pixel at a time and stride of 2 means, filter is moving two pixels at a time, which will reduce the output size. Larger strides can lead to smaller feature maps that will reduce computational cost, but loses details.

4.1.4. Feature Map: Result of the convolution operation performed on an image using a filter, that slides over the image i.e., multiply and sum up the values at each step will get stored in the feature map. In simple terms, feature map is the output of earlier convolution layer that highlights important patterns.

4.1.5. Batch Normalization and ReLU: Batch normalization is used to equalize the inputs (scales the values between -1 and 1) to each layer for each mini-batch. As a result, the learning process will get stabilized, and the amount of epochs needed for training deep neural networks will be decreased. ReLU stands for Rectified Linear Unit, helps in producing improved accuracy by mapping all negative values to 0 compared to other non-linear activation functions, thus preventing the vanishing gradient problem.
ReLU: $f(x) = \text{maximum}(0, x)$

4.1.6. Pooling layers: The pooling layer (max & average) aggregates the features in an image region generated by convolution layers results in dimensionality reduction.

4.1.6.1. Max pooling layer: It refers to a process that chooses the most significant element from the feature map area that the filter covers. The output of the max pooling layer contains the most noticeable and prominent features from the previous feature map.

4.1.6.2. Average pooling layer: It refers to a process that chooses the average of all the elements from the feature map area that the filter covers. In the proposed method, average pooling was performed globally (Global average pooling layer) across the entire feature map, reduces each feature map to a single value. At the end of CNN, instead of using flattening method, we chose this to convert the final matrix to one dimensional vector.

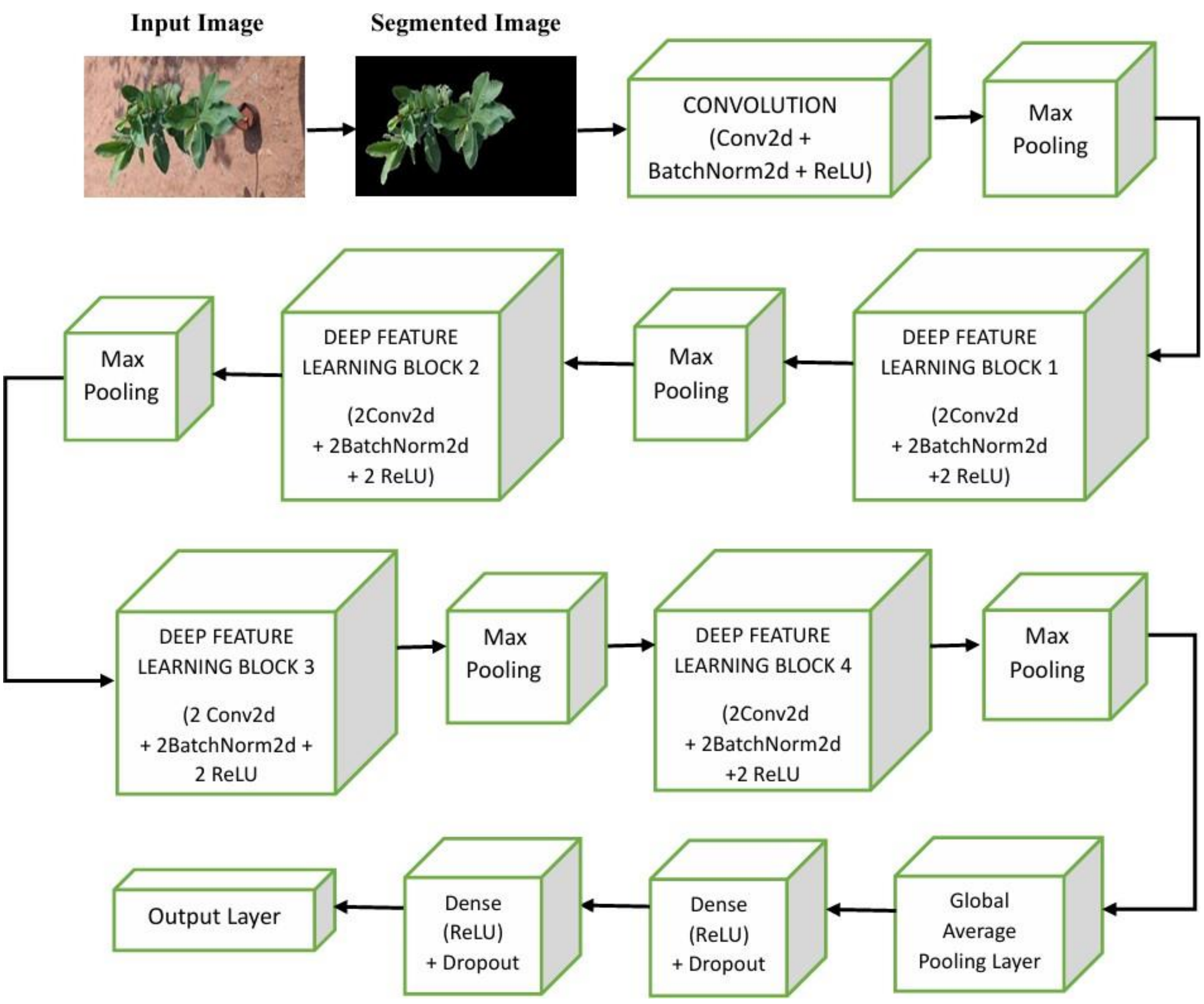

Input Image
Segmented Image
CONVOLUTION (Conv2d + BatchNorm2d + ReLU)
Max Pooling
DEEP FEATURE LEARNING BLOCK 1 (2Conv2d + 2BatchNorm2d +2 ReLU)
Max Pooling
DEEP FEATURE LEARNING BLOCK 2 (2Conv2d + 2BatchNorm2d + 2 ReLU)
Max Pooling
DEEP FEATURE LEARNING BLOCK 3 (2 Conv2d + 2BatchNorm2d + 2 ReLU
Max Pooling
DEEP FEATURE LEARNING BLOCK 4 (2Conv2d + 2BatchNorm2d +2 ReLU
Max Pooling
Global Average Pooling Layer
Dense (ReLU) + Dropout
Dense (ReLU) + Dropout
Output Layer

Fig.4. Deep-Medic Architecture

4.1.7. Feature learning blocks: The Feature learning blocks are considered the main building blocks of the proposed model. The idea of introducing this block is to concatenate layers before proceeding with down sampling using pooling layers. In the proposed model, there are four deep feature learning blocks.

4.1.8. Fully Connected Layer and Dropout: Also known as Dense layer, connects all the neurons present in the previous layer to all the neurons present in the next layer. High-level representations will be learned and helps in making predictions. Uses activation functions like reLU, softmax. Dropout means disabling some neurons, helped as a regularization technique by reducing over-fitting errors with the proposed model compared to other pre-trained models.

4.1.9. Classification layer: This layer is composed of global average pooling and a dense layer. Each feature map is averaged by the global average pooling layer, thus results to a one-dimensional vector. In a multi-class problem, softmax activation function gives decimal probability to each class between 0 to 1.

Table 1: Deep learning models vs number of trainable parameters

Deep Learning Model	Trainable parameters
ResNet50	23,589,992
ResNet34	21,305,192
EfficientNetB4	17,620,336
VGG16	15,262,528
DenseNet121	6,994,856
MobileNetV3	4,253,272
Deep-Medic (Proposed method)	5,142,184

4.1.10 Deep-Medic Model Training

Training the Deep-Medic model is a crucial step in Medicinal plant classification. The model is trained as a supervised classification algorithm using convolutional neural networks (CNNs). It consists of multiple convolutional layers, feature learning blocks, fully connected layers, and an output layer with a softmax activation function to classify plant species.

The input images are resized to 224×224 pixels, this ensures consistent input across all samples, improving training stability and convergence speed. And then augmented and segmented. The network took 224*224 RGB segmented images as input, followed by which the image is passed to a convolution layer which is of 64 filters each of 7*7 size, with a stride length of 2. And then the Batch Normalization will normalize the values of the output received from the previous convolution layer. And then reLU activation function is applied to map the negative values (considered as unimportant) to zero. Later, Max pooling layer with a 2*2 kernel of stride length 2 is used to extract prominent features i.e., choosing maximum value from each filter, thus performs the dimensionality reduction.

Then there are four feature learning blocks where each block consists of two convolution layers followed by Batch Normalization and reLU and a max pooling layer. The first feature learning block uses 64 filters, and the remaining blocks will use 128, 256, 512 filters respectively. By increasing the number of filters, the model is able to learn more features in depth. And the resulted feature map from the last feature learning block will be passed to Fully connected layers i.e., Global average pooling layer will average the values of each filter to a single value, thus the entire feature map will be converted to a single vector.

And then two dense layers are used, which learn more complex features followed by dropout to disable some neurons during training which will reduce overfitting issues. Finally, fully connected layer with softmax activation function will assign decimal probability for each of the 40 classes, producing the final feature map. The one with the highest probability among all is the predicted class of plant species. The number of trainable parameters learned are 5,142,184 in Deep-Medic model, close to MobileNetV3 but relatively small when compared to the standard deep learning architectures such as Resnet34, Resnet50, VGG16, DenseNet121, EfficientNetB4. Table 2 illustrates the details of trainable parameters used by the mentioned deep architectures.

The doubling of filters in feature learning blocks (64, 128, 256, 512) was adopted from VGG16 architecture, learn the features in depth. Global average pooling, a layer used instead of flattening was taken from MobileNet architecture, lightweight often used in mobile and embedded applications. Here, the parameters such as weights and biases were initialized randomly and get trained. Linearization using Global average pooling is carried out with around 50% dropout resulting in more robust features. The model uses batch size as 16 (mini-batches), trained with 60 epochs. Refer Fig.4.

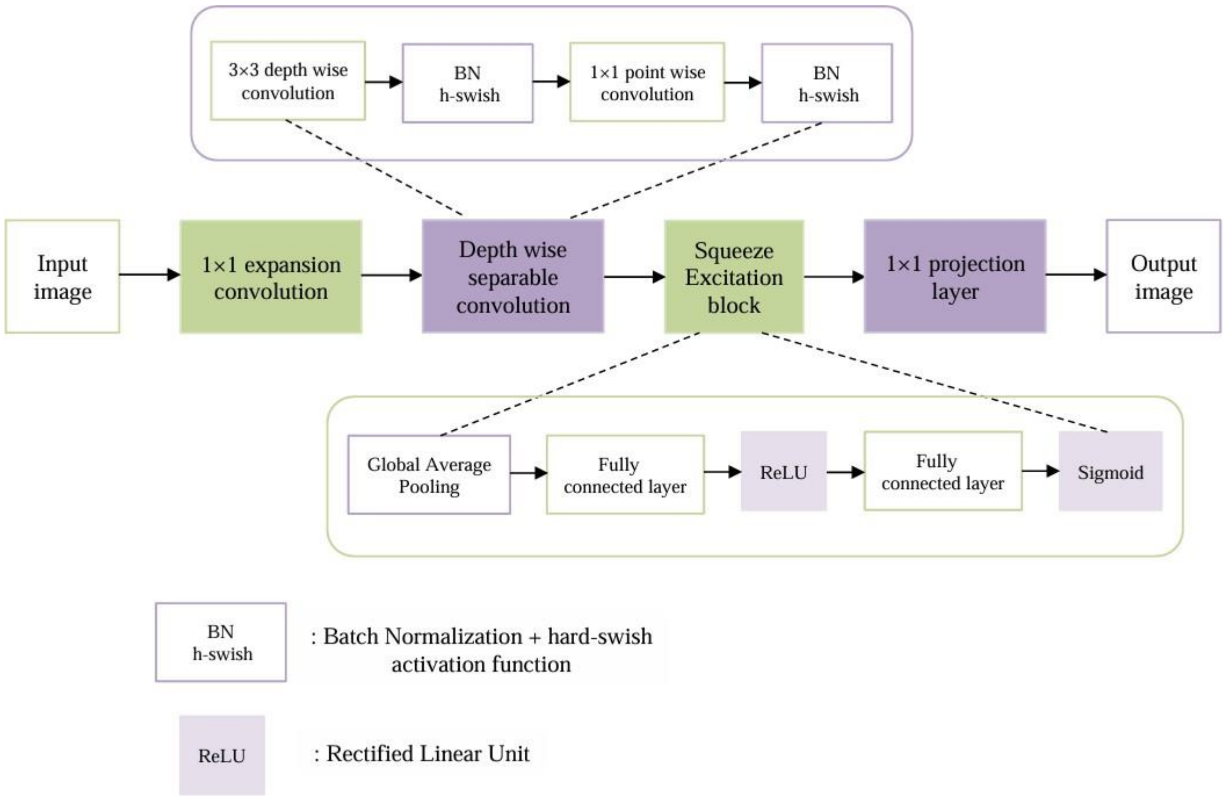


Fig.5. MobileNetV3 Architecture

4.2. MOBILENET V3

MobileNetV3 is implemented with a mix of depthwise separable convolutions, inverted residual blocks, squeeze-and-excitation (SE) modules, and new non-linear activation functions like ReLU and h-swish. The model exists in two variants namely MobileNetV3-Large and MobileNetV3-Small tuned for varying performance and efficiency. The network starts with an initial standard convolution layer followed by batch normalization and h-swish activation, paving the way for the lightweight and efficient operations that follow. The backbone of MobileNetV3 is a cascade of inverted residual blocks that each have a 1 × 1 convolution (expansion), a depthwise separable convolution with typical kernel sizes 3 × 3 or 5 × 5, and a 1 × 1 projection convolution which decreases the dimensionality. Some of these blocks can contain a squeeze-and-excitation (SE) module, which adjusts the channel-wise feature responses. The SE block applies global average pooling and a tiny fully connected network along with a sigmoid activation function to highlight informative features.

There are sixteen such bottleneck blocks in MobileNetV3-Large, which are organized to increasingly extract high-level features at low computational cost. MobileNetV3- Small, which is for still more limited environments, comprises thirteen bottleneck blocks with modified settings.

Residual connections are employed across the architecture wherever input and output shapes match so that gradients can pass through the network more freely. After the bottleneck sequence, there is a final expansion layer by a 1 × 1 convolution and then a global average pooling layer that minimizes spatial sizes. It is linked to a fully connected layer and a softmax for end-classification. The activation functions utilized by the model include both ReLU and h-swish, chosen strategically with empirical performance in layers. Fig.5. illustrates the MobileNetv3 architecture diagram.

Each convolutional unit in MobileNetV3 typically follows a sequence of batch normalization, non-linear activation, and convolution operation, often followed by dropout where necessary. This combination of efficient operations and attention mechanisms allows MobileNetV3 to achieve high accuracy while being suitable for real-time inference on mobile and embedded devices. Refer Fig.5.

4.3. YOLOv9 ARCHITECTURE

YOLOv9 is a cutting-edge real-time object detection (one-stage) model that pushes accuracy and efficiency through incorporating multiple novel architectural elements. CSPNet (Cross Stage Partial Network) enhances the ability to learn while decreasing the computational expense through dividing feature maps and re-combining them later, promoting gradient flow and eliminating redundancy. ELAN (Efficient Layer Aggregation Network) provides efficient gradient propagation and reuse of features by strategically constructing connections between layers to achieve optimization in learning.

Based on these, YOLOv9 employs the Information Bottleneck Principle to counteract information loss within deep networks. In an effort to combat this, it adopts reversible functions via PGI (Programmable Gradient Information), where there's a primary inference branch, a secondary reversible branch, and multi-level auxiliary information for effective and accurate training.

Lastly, GELAN (Generalized Efficient Layer Aggregation Network) provides a complement to PGI through a very dynamic and efficient architecture that improves information retention and processing. Combined, these elements result in YOLOv9 being a strong model that sets new standards in real-time detection performance in terms of speed, accuracy, and efficiency.

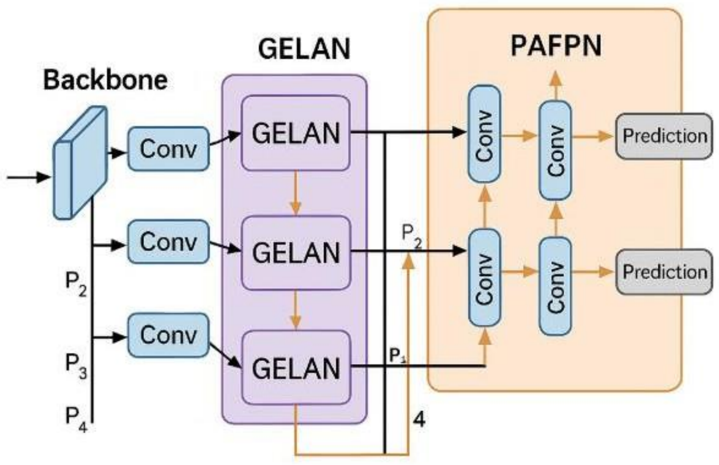


Fig.6. YOLOv9 Architecture

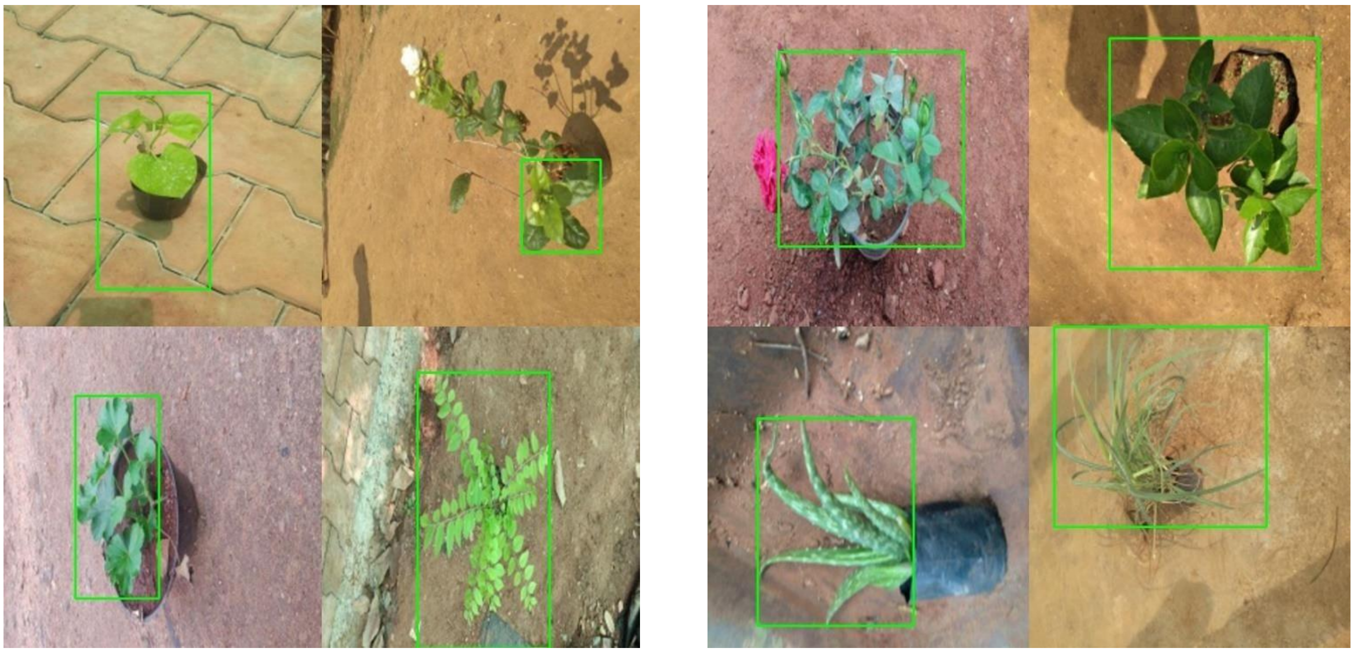
4.3.1. YOLOv9 model training

To facilitate multi-object detection of medicinal plants in real-world scenarios, a composite dataset was created by merging individual plant images into 2×2 grid structures. Each multi-plant image was annotated in YOLO format with normalized bounding boxes and respective class labels. In this format, each plant within an image is defined by one line in a.txt file with five space-delimited values: <class_label> <x_center> <y_center> <width> <height>. The class_label is an integer identifier of the class of the object and generally initializes from 0. The rest of the four values x_center, y_center, width, and height are all normalized within the context of the dimensions of the image such that their range is from 0 to 1.

Namely, the x_center and y_center value the coordinates in the middle of the bounding box, while the width and the height define the bounding box dimensions. These values are obtained by dividing pixel values of the bounding box with the width and height of the image respectively. This normalized format provides consistency and scalability across images of different sizes and is consumed by YOLO models for effective detection and localization of objects during training as well as inference. 1,000 such images were created using randomly chosen plants from 40 available classes. To create bounding box annotations for composite images of multiple plants, a contour-based approach was adopted specifically for the purpose. For every plant image positioned on the 2×2 grid canvas, a green-based segmentation mask was created with HSV color thresholding to separate the plant areas from the background. Precisely, the image was converted initially to the HSV color space, and a mask was used using a set range of green colors.

Contours were then detected with the mask using the OpenCV findContours() method. For the most significant object, only the largest contour by area was used. A bounding rectangle was created around the contour with boundingRect(), and the coordinates were normalized relative to the composite canvas size to match the YOLO annotation format. Refer Fig.7. for bounding boxes detection using. This was made possible with the ability to extract bounding box with high accuracy specific to the segmented region of interest, which improved annotation quality for training synthetic multi-plant object detection models. The respective YOLO annotation files were created and stored separately for training. After dataset creation, the whole dataset was split into training, validation, and test sets with an 8:1:1 ratio to guarantee proper generalization and model testing.

The well-organized dataset was formatted into subdirectories for images and labels within each split, as demanded by YOLO training. YOLO employs the data.yaml file to interpret the dataset configuration in a human-readable and standardized format. This enables the training engine to dynamically adjust to any special dataset without the use of hard-coded values. The YAML format promotes modularity and reusability so that users can train models on various datasets by just altering this one configuration file. Utilization of YAML (Yet Another Markup Language) offers simplicity, readability, and flexibility in defining the training parameters, matching YOLO's aim to simplify the pipeline for object detection.



(a) (b)

Fig.7. Samples of multi-plant images with bounding boxes drawn using contour

YAML file comprises necessary metadata i.e., relative directory paths to the training, validation, and testing image directories such that the model is able to find the image data properly. The file also indicates the number of classes (nc) within the dataset and gives a list of class names under the names field such that each index maps to a distinct label ID. In the current research, the data.yaml file contained 40 unique classes of medicinal plants. The YOLOv9 model was subsequently trained against this personal dataset utilizing the Ultralytics library, with training set to 50 epochs and an image resolution of 640×640 pixels.

After training, the best checkpoint model was used for testing against unseen test images. Inference on the test dataset was carried out using the trained model, and the prediction was visualized. The test images were processed individually, and the results of detection were stored, displaying the bounding boxes and predicted labels. The predicted outputs were inspected visually to determine model performance, such that the model was able to detect and locate several plant species in a single image.

4.4. RETINANET ARCHITECTURE

RetinaNet is a powerful one-stage object detection model that strikes an effective balance between accuracy and computational efficiency. It introduces Focal Loss, a modified version of cross-entropy loss, to address the extreme class imbalance typical in dense object detection by reducing the loss contribution from easy negatives and focusing training on hard, misclassified examples. This enables the model to perform better on rare or small objects.

Another major innovation is the Feature Pyramid Network (FPN), built on a ResNet backbone, which enhances multi-scale feature extraction. FPN constructs a top-down pathway with lateral connections to merge high-level semantic features with fine spatial details from earlier layers, enabling accurate detection of both small and large objects. At each level of the feature pyramid, RetinaNet employs two subnetworks: a classification subnetwork that predicts object classes and their confidence scores using sigmoid activations, and a bounding box regression subnetwork that refines anchor boxes to closely match ground-truth objects.

These subnetworks operate at multiple scales, making the model robust to varying object sizes and aspect ratios. Additionally, the use of upsampling and 1×1 convolutions in lateral connections helps preserve important spatial features while reducing channel dimensions for efficiency. Together, these architectural components make RetinaNet a high-performing, real-time detector well-suited for complex visual recognition tasks.

4.4.1. RetinaNet model for training:

In this paper, the RetinaNet model was used for object detection and classification of multiple medicinal plants in one image. Training annotations for the RetinaNet model were created in CSV format tailored for multi-object detection tasks related to medicinal plants. Every row in the CSV depicts a single instance of a plant in an image and contains necessary information needed for training. The annotations include the path to the image file (image_path), the coordinates of the bounding box (x1, y1, x2, y2), and the name of the corresponding plant class (class_name). The bounding box coordinates define the top-left and bottom-right corners of every detected plant in pixel coordinates, precisely locating them in the image. As there can be more than one plant in an image, the same image_path could be repeated across rows with different bounding box values.

In preprocessing, a column named label was created by encoding the class_name values into numerical labels. Alphabetical ordering of all unique class names and giving each a distinct integer index were done to have a consistent mapping of labels for model training. This formatted annotation format gave a clean and effective method to associate image data with object location and class, making it ideal for deep learning-based object detection models such as RetinaNet. A custom PyTorch dataset was used to load the multi-plant images and their respective bounding box annotations and class labels. Data processing involved converting images to tensors to make them compatible with the model.

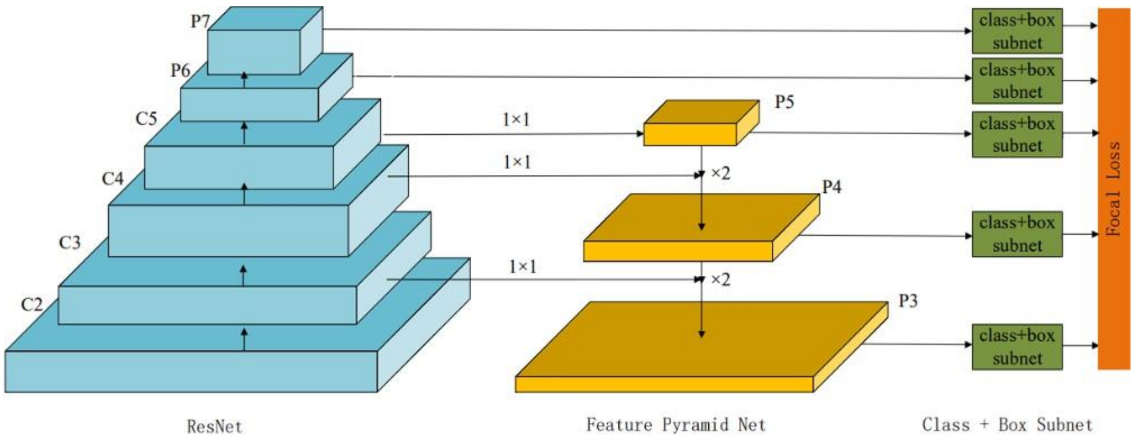


Fig.8. RetinaNet architecture

A pre-trained RetinaNet with a ResNet50 backbone was used as the initial architecture, and its classification head was modified to have the number of unique plant classes in the dataset. The model was trained for 50 epochs with the AdamW optimizer and a learning rate and weight decay fine-tuned to generalize better. Throughout training, the overall loss was tracked that included both classification loss and the bounding box regression loss to keep the model successfully converging. Following training, the model was tested under inference mode upon unseen composite images with multiple plant. This method provided a solid pipeline from training and annotation to visualization, allowing detection of multiple plant species per image with high accuracy.

CHAPTER 5

IMPLEMENTATION DETAILS

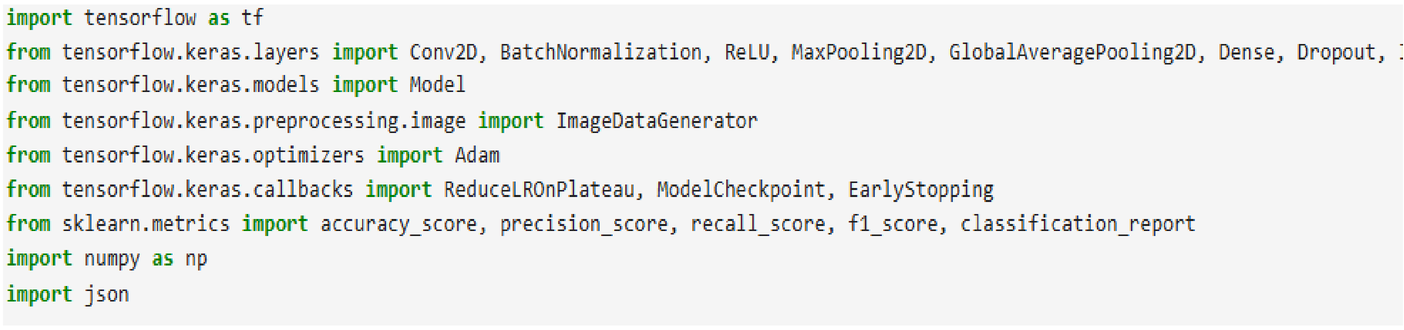
Augmented Dataset Generation:

```python
# Intensity transformation function
def adjust_contrast(image, delta):
    """ Adjusts image contrast using a scaling factor δ. """
    return np.clip(image * delta, 0, 255).astype(np.uint8)

# Augmentation function
def augment_image(image):
    """ Applies transformations (contrast, rotation, flip). """
    return [
        adjust_contrast(image, 0.6),  # Low contrast
        adjust_contrast(image, 1.5),  # High contrast
        cv2.rotate(image, cv2.ROTATE_180),  # Rotate 180
        cv2.flip(image, 0)  # Flip vertically
    ]
```

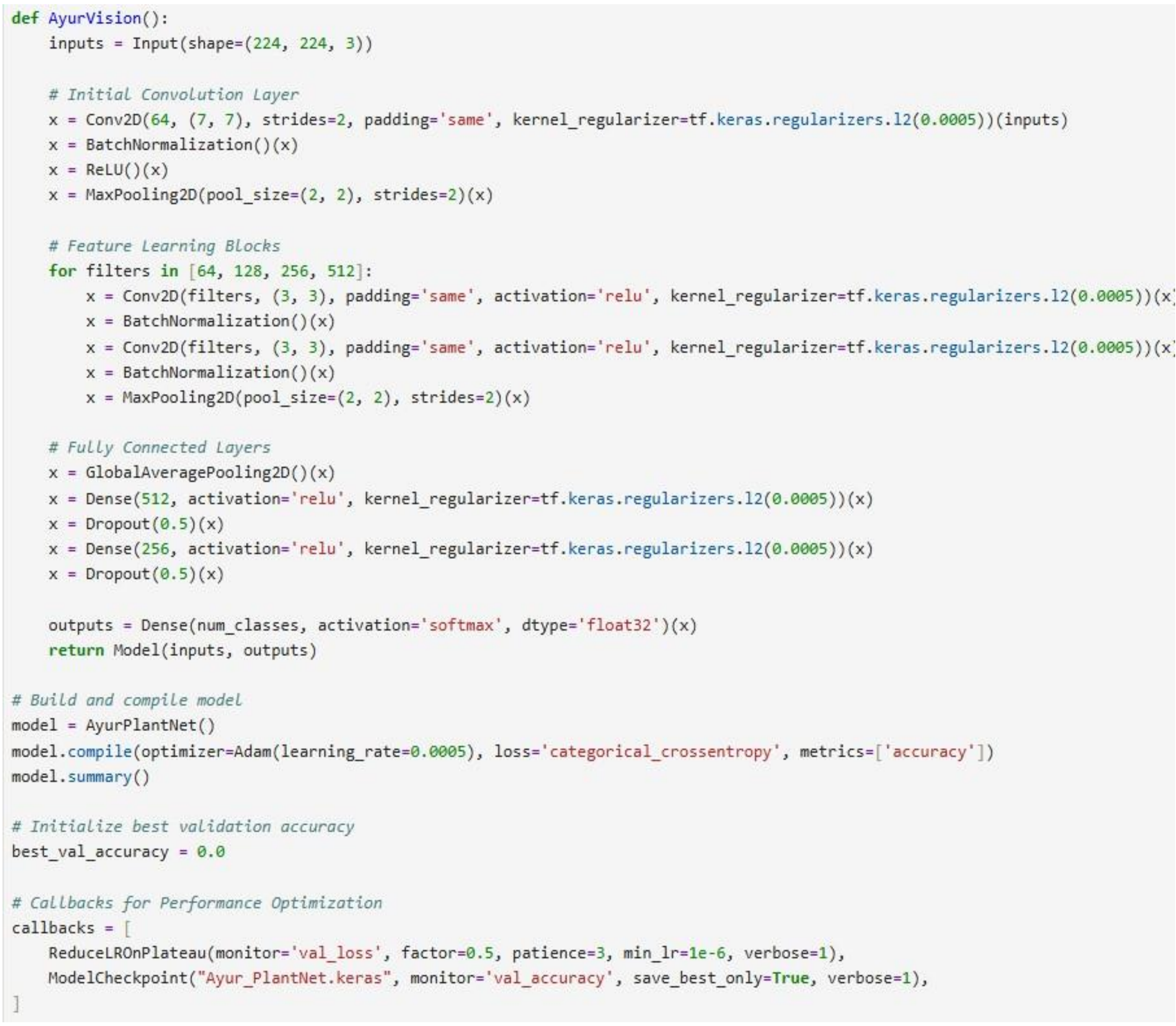
Segmentation:

```python
def segment_green_plant(image_path, save_path):
    """Extracts only the green plant and makes the background black."""

    # Read image
    image = cv2.imread(image_path)
    if image is None:
        return  # Skip unreadable images

    # Convert to RGB & HSV (for color filtering)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define Green Color Range in HSV
    lower_green = np.array([25, 40, 40])  # Lower bound of green
    upper_green = np.array([90, 255, 255])  # Upper bound of green

    # Create a binary mask where green regions are white (255) and others are black (0)
    mask = cv2.inRange(image_hsv, lower_green, upper_green)

    # Apply median filtering to remove noise
    mask = cv2.medianBlur(mask, 5)

    # Use mask to extract the plant from the original image
    result = cv2.bitwise_and(image_rgb, image_rgb, mask=mask)

    # Save the segmented image
    cv2.imwrite(save_path, cv2.cvtColor(result, cv2.COLOR_RGB2BGR))
```

1. **Deep -Medic:**
   Without K-Fold:



```python
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, MaxPooling2D, GlobalAveragePooling2D, Dense, Dropout,
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau, ModelCheckpoint, EarlyStopping
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report
import numpy as np
import json
```

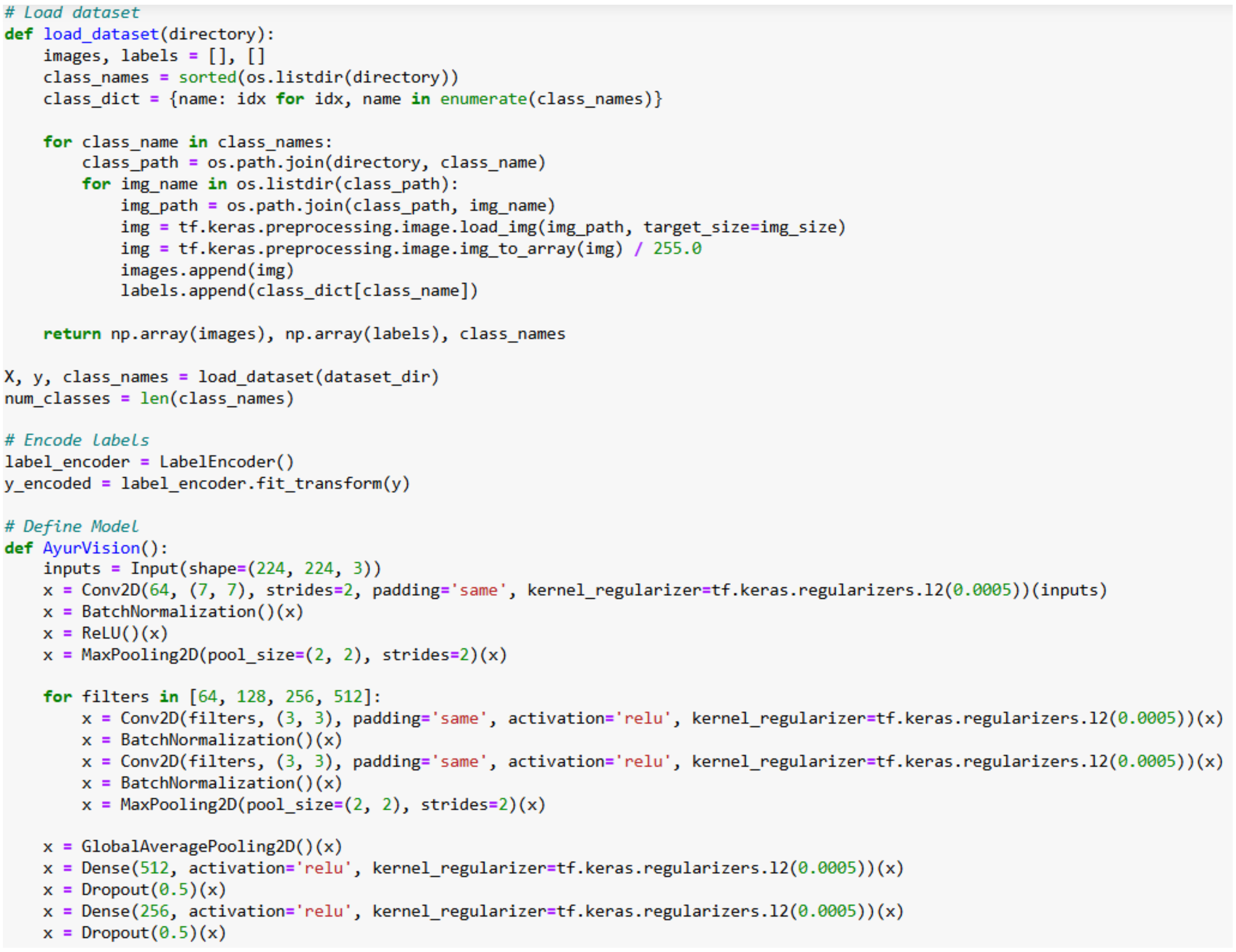


```python
def AyurVision():
    inputs = Input(shape=(224, 224, 3))

    # Initial Convolution Layer
    x = Conv2D(64, (7, 7), strides=2, padding='same', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = MaxPooling2D(pool_size=(2, 2), strides=2)(x)

    # Feature Learning Blocks
    for filters in [64, 128, 256, 512]:
        x = Conv2D(filters, (3, 3), padding='same', activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x
        x = BatchNormalization()(x)
        x = Conv2D(filters, (3, 3), padding='same', activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x
        x = BatchNormalization()(x)
        x = MaxPooling2D(pool_size=(2, 2), strides=2)(x)

    # Fully Connected Layers
    x = GlobalAveragePooling2D()(x)
    x = Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
    x = Dropout(0.5)(x)

    outputs = Dense(num_classes, activation='softmax', dtype='float32')(x)
    return Model(inputs, outputs)

# Build and compile model
model = AyurPlantNet()
model.compile(optimizer=Adam(learning_rate=0.0005), loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()

# Initialize best validation accuracy
best_val_accuracy = 0.0

# Callbacks for Performance Optimization
callbacks = [
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-6, verbose=1),
    ModelCheckpoint("Ayur_PlantNet.keras", monitor='val_accuracy', save_best_only=True, verbose=1),
]
```

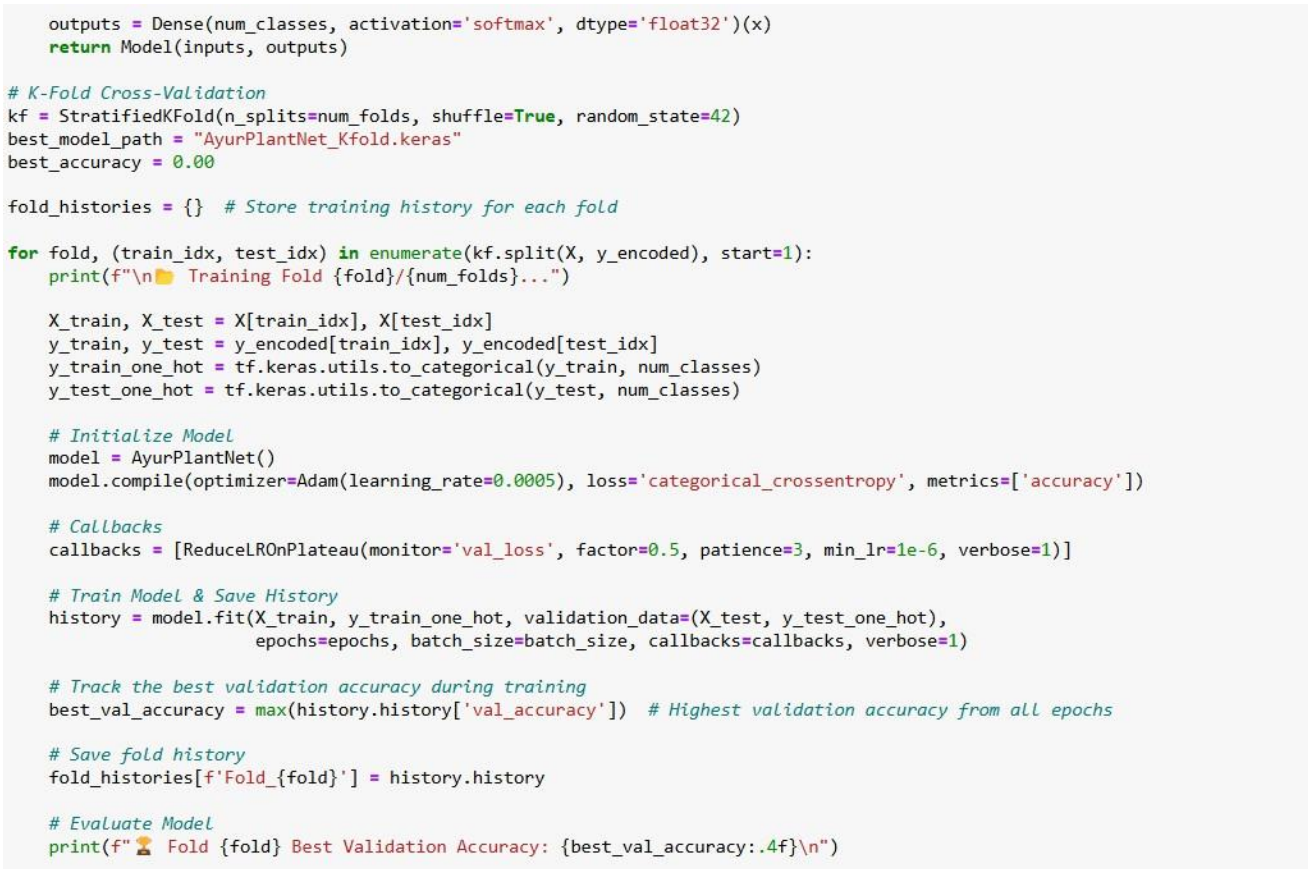
```
# Train model
epochs = 60
history = model.fit(train_generator, validation_data=test_generator, epochs=epochs, callbacks=callbacks)

# Save training history as JSON
with open("Training_history.json", "w") as f:
    json.dump(history.history, f)

# Update best validation accuracy
best_val_accuracy = max(history.history['val_accuracy'])
print(f"🏆 Best Validation Accuracy: {best_val_accuracy:.4f}")

# Load the Best Model
model = tf.keras.models.load_model("Ayur_PlantNet.keras")

# Evaluate model
y_pred = model.predict(test_generator)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = test_generator.classes

# Compute Metrics
accuracy = accuracy_score(y_true, y_pred_classes)
precision = precision_score(y_true, y_pred_classes, average='macro')
recall = recall_score(y_true, y_pred_classes, average='macro')
f1 = f1_score(y_true, y_pred_classes, average='macro')

# Print Evaluation Metrics
print(f"✅ Accuracy: {accuracy:.4f}")
print(f"🎯 Precision: {precision:.4f}")
print(f"🔁 Recall: {recall:.4f}")
print(f"⚖️  F1-score: {f1:.4f}")

# Classification Report
print("\nClassification Report:")
print(classification_report(y_true, y_pred_classes))
```

With K fold:

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Dropout, Input
from tensorflow.keras.layers import MaxPooling2D, GlobalAveragePooling2D, Dense,
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import numpy as np
import os
import gc
import json

# Enable Mixed Precision for Memory Optimization
tf.keras.mixed_precision.set_global_policy("mixed_float16")

# Paths
dataset_dir = "/kaggle/working/segmented_dataset"

# Image properties
img_size = (224, 224)
batch_size = 16
epochs = 60
num_folds = 10
```

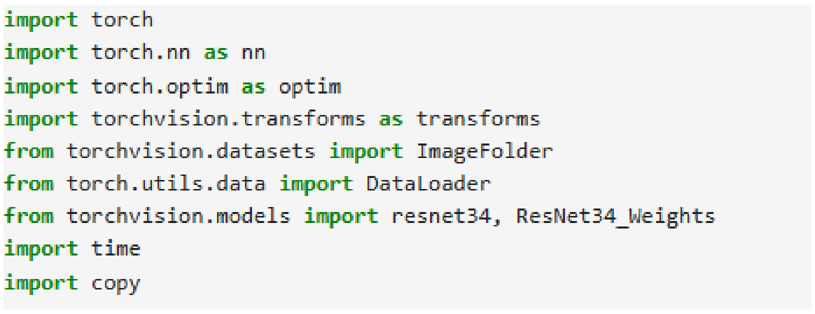


```python
# Load dataset
def load_dataset(directory):
    images, labels = [], []
    class_names = sorted(os.listdir(directory))
    class_dict = {name: idx for idx, name in enumerate(class_names)}

    for class_name in class_names:
        class_path = os.path.join(directory, class_name)
        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            img = tf.keras.preprocessing.image.load_img(img_path, target_size=img_size)
            img = tf.keras.preprocessing.image.img_to_array(img) / 255.0
            images.append(img)
            labels.append(class_dict[class_name])

    return np.array(images), np.array(labels), class_names

X, y, class_names = load_dataset(dataset_dir)
num_classes = len(class_names)

# Encode labels
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Define Model
def AyurVision():
    inputs = Input(shape=(224, 224, 3))
    x = Conv2D(64, (7, 7), strides=2, padding='same', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = MaxPooling2D(pool_size=(2, 2), strides=2)(x)

    for filters in [64, 128, 256, 512]:
        x = Conv2D(filters, (3, 3), padding='same', activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
        x = BatchNormalization()(x)
        x = Conv2D(filters, (3, 3), padding='same', activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
        x = BatchNormalization()(x)
        x = MaxPooling2D(pool_size=(2, 2), strides=2)(x)

    x = GlobalAveragePooling2D()(x)
    x = Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.0005))(x)
    x = Dropout(0.5)(x)
```

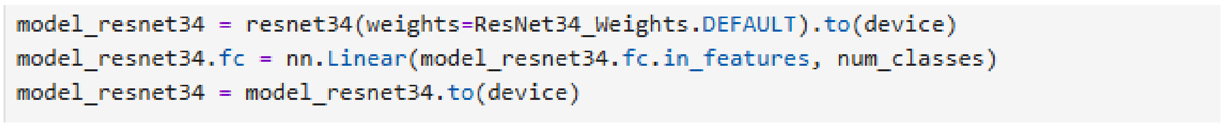


```python
    outputs = Dense(num_classes, activation='softmax', dtype='float32')(x)
    return Model(inputs, outputs)

# K-Fold Cross-Validation
kf = StratifiedKFold(n_splits=num_folds, shuffle=True, random_state=42)
best_model_path = "AyurPlantNet_Kfold.keras"
best_accuracy = 0.00

fold_histories = {}  # Store training history for each fold

for fold, (train_idx, test_idx) in enumerate(kf.split(X, y_encoded), start=1):
    print(f"\n📁 Training Fold {fold}/{num_folds}...")

    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y_encoded[train_idx], y_encoded[test_idx]
    y_train_one_hot = tf.keras.utils.to_categorical(y_train, num_classes)
    y_test_one_hot = tf.keras.utils.to_categorical(y_test, num_classes)

    # Initialize Model
    model = AyurPlantNet()
    model.compile(optimizer=Adam(learning_rate=0.0005), loss='categorical_crossentropy', metrics=['accuracy'])

    # Callbacks
    callbacks = [ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-6, verbose=1)]

    # Train Model & Save History
    history = model.fit(X_train, y_train_one_hot, validation_data=(X_test, y_test_one_hot),
                        epochs=epochs, batch_size=batch_size, callbacks=callbacks, verbose=1)

    # Track the best validation accuracy during training
    best_val_accuracy = max(history.history['val_accuracy'])  # Highest validation accuracy from all epochs

    # Save fold history
    fold_histories[f'Fold_{fold}'] = history.history

    # Evaluate Model
    print(f"⏳ Fold {fold} Best Validation Accuracy: {best_val_accuracy:.4f}\n")
```

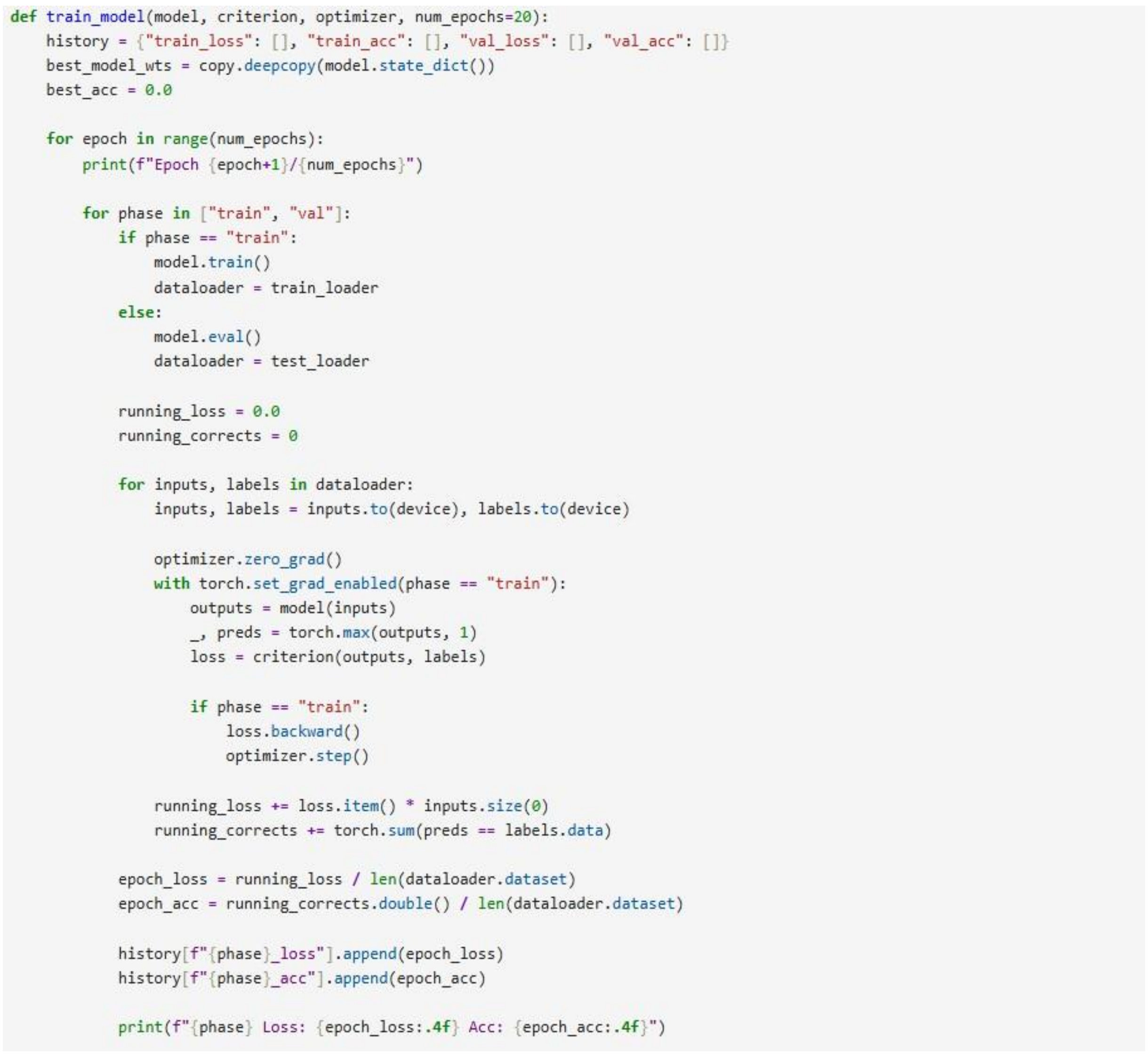
2. ResNet34:



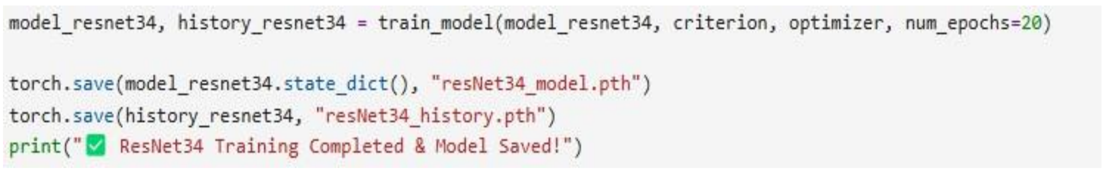
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision.models import resnet34, ResNet34_Weights
import time
import copy
```



```
model_resnet34 = resnet34(weights=ResNet34_Weights.DEFAULT).to(device)
model_resnet34.fc = nn.Linear(model_resnet34.fc.in_features, num_classes)
model_resnet34 = model_resnet34.to(device)
```

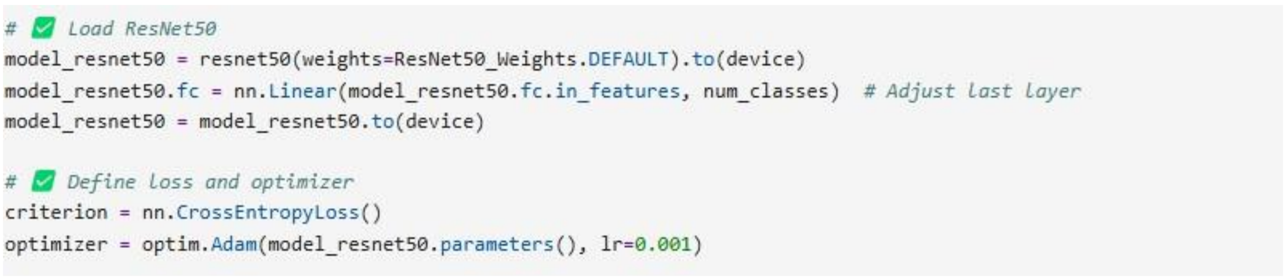


```
def train_model(model, criterion, optimizer, num_epochs=20):
    history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")

        for phase in ["train", "val"]:
            if phase == "train":
                model.train()
                dataloader = train_loader
            else:
                model.eval()
                dataloader = test_loader

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloader:
                inputs, labels = inputs.to(device), labels.to(device)

                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == "train"):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == "train":
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloader.dataset)
            epoch_acc = running_corrects.double() / len(dataloader.dataset)

            history[f"{phase}_loss"].append(epoch_loss)
            history[f"{phase}_acc"].append(epoch_acc)

            print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")
```



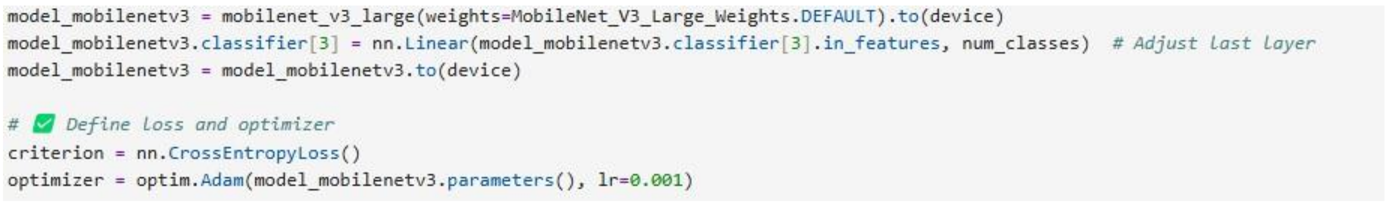
```
model_resnet34, history_resnet34 = train_model(model_resnet34, criterion, optimizer, num_epochs=20)

torch.save(model_resnet34.state_dict(), "resNet34_model.pth")
torch.save(history_resnet34, "resNet34_history.pth")
print("✅ ResNet34 Training Completed & Model Saved!")
```

3. **ResNet50:**



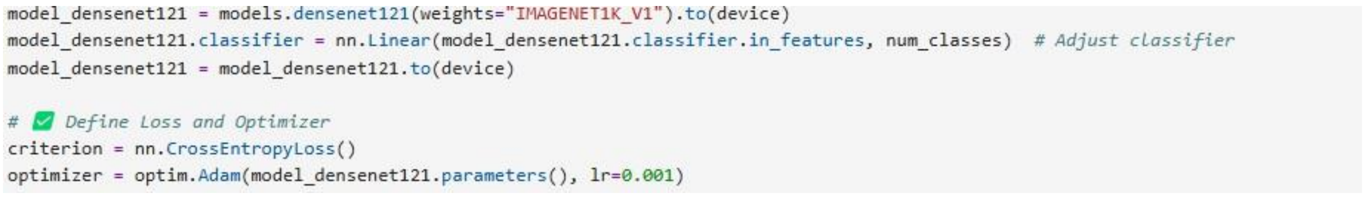
```
# ✅ Load ResNet50
model_resnet50 = resnet50(weights=ResNet50_Weights.DEFAULT).to(device)
model_resnet50.fc = nn.Linear(model_resnet50.fc.in_features, num_classes)  # Adjust last layer
model_resnet50 = model_resnet50.to(device)

# ✅ Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_resnet50.parameters(), lr=0.001)
```

4. **MobileNetV3 Large:**



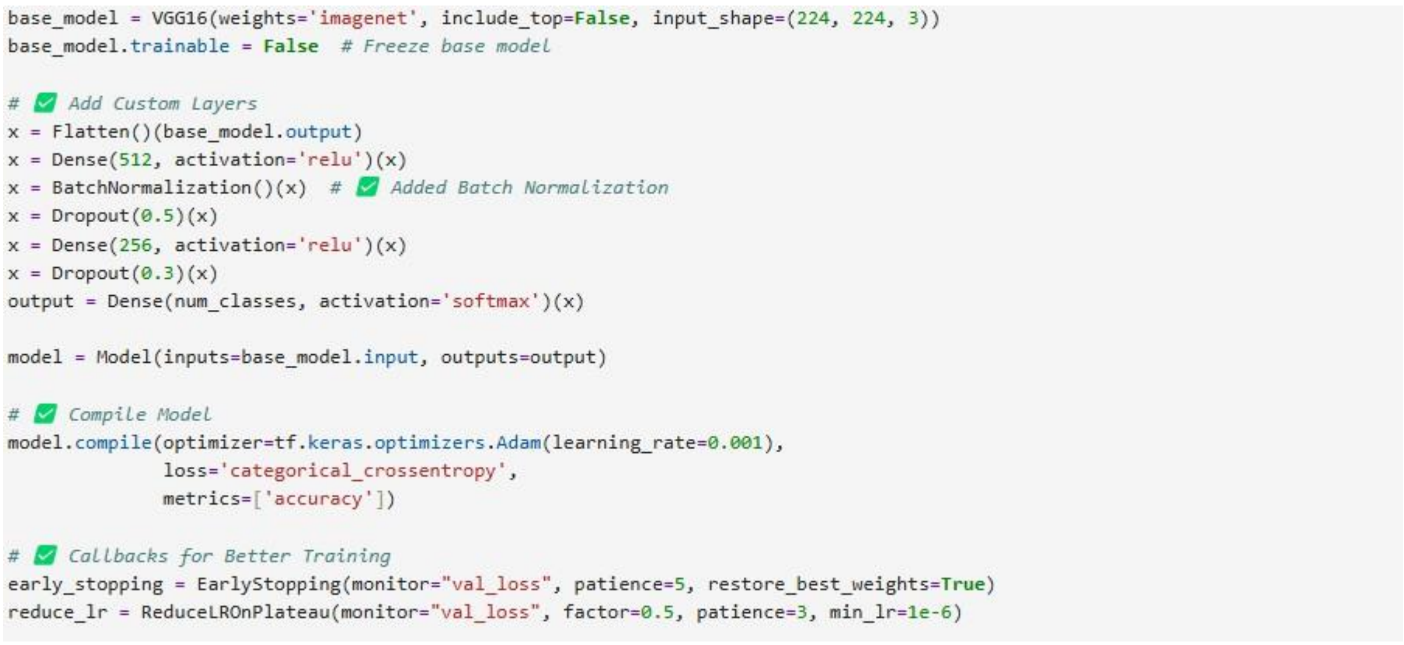
```
model_mobilenetv3 = mobilenet_v3_large(weights=MobileNet_V3_Large_Weights.DEFAULT).to(device)
model_mobilenetv3.classifier[3] = nn.Linear(model_mobilenetv3.classifier[3].in_features, num_classes)  # Adjust last layer
model_mobilenetv3 = model_mobilenetv3.to(device)

# ✅ Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_mobilenetv3.parameters(), lr=0.001)
```

5. **DenseNet 121:**



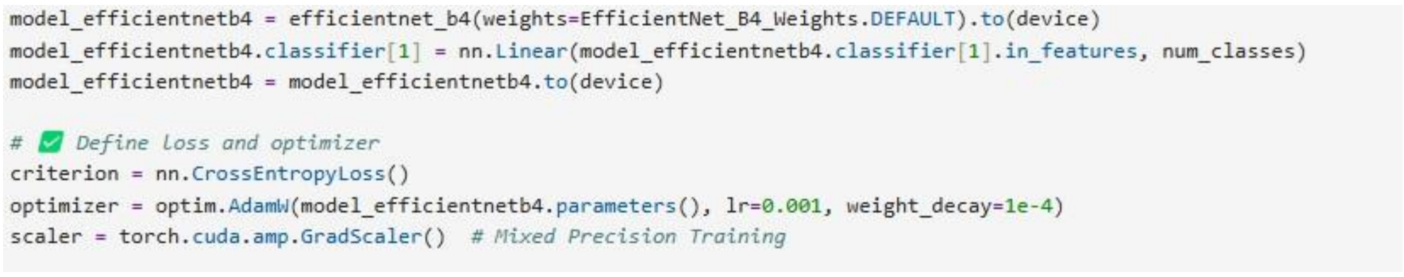
```
model_densenet121 = models.densenet121(weights="IMAGENET1K_V1").to(device)
model_densenet121.classifier = nn.Linear(model_densenet121.classifier.in_features, num_classes)  # Adjust classifier
model_densenet121 = model_densenet121.to(device)

# ✅ Define Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_densenet121.parameters(), lr=0.001)
```

6. **VGG16:**



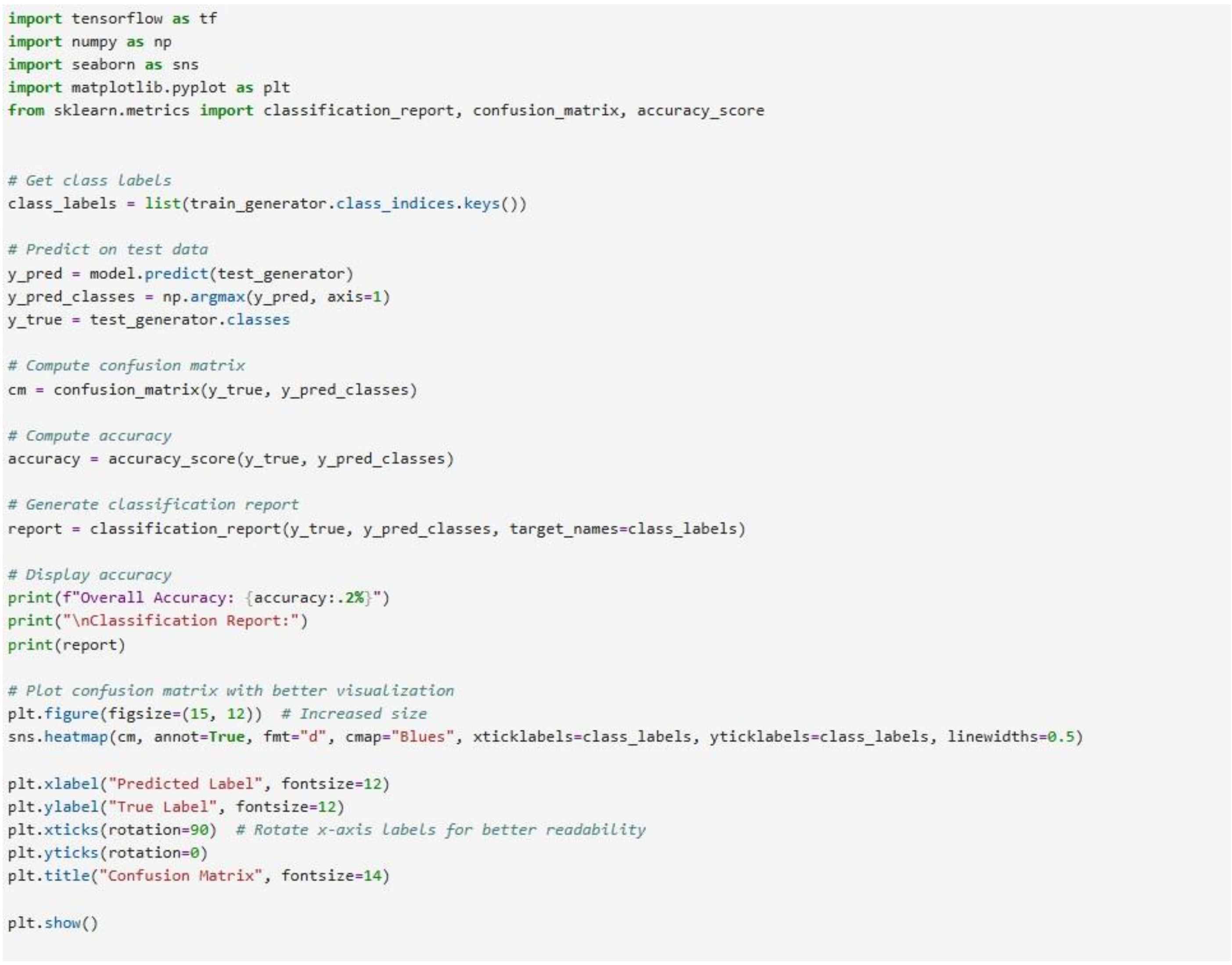
```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze base model

# ✅ Add Custom Layers
x = Flatten()(base_model.output)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)  # ✅ Added Batch Normalization
x = Dropout(0.5)(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.3)(x)
output = Dense(num_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=output)

# ✅ Compile Model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# ✅ Callbacks for Better Training
early_stopping = EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6)
```

7. EfficientNet B4:



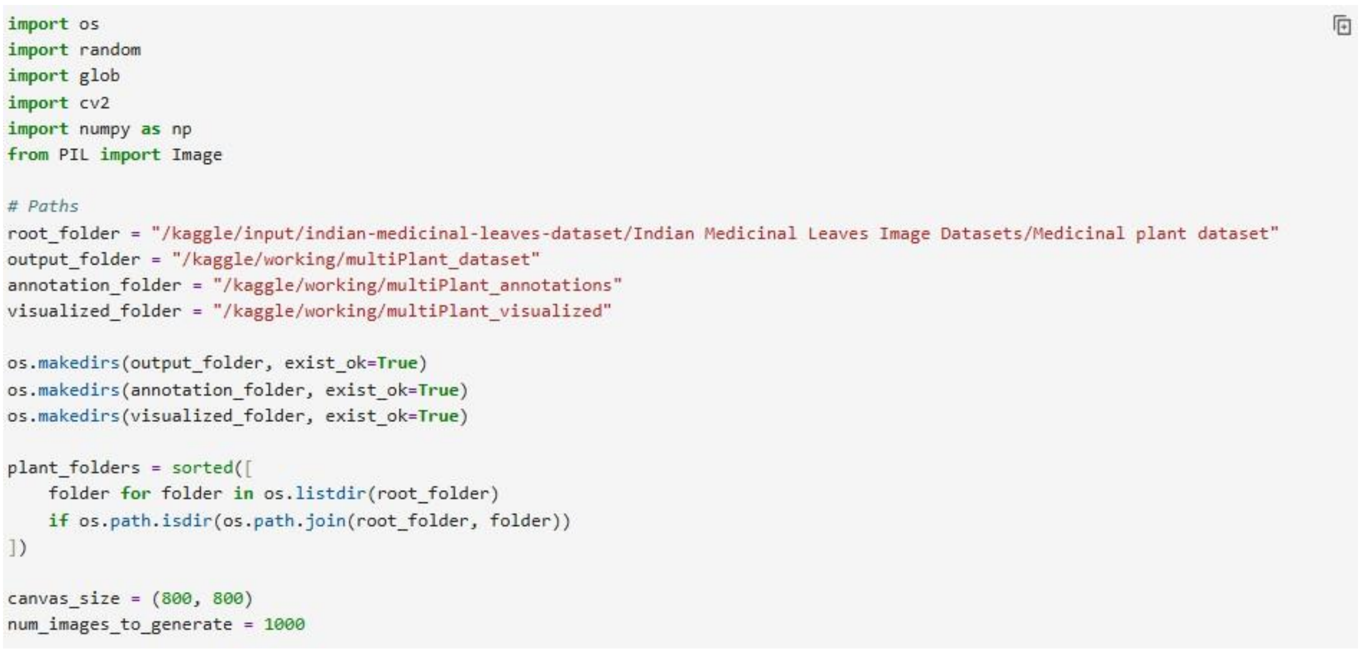
```
model_efficientnetb4 = efficientnet_b4(weights=EfficientNet_B4_Weights.DEFAULT).to(device)
model_efficientnetb4.classifier[1] = nn.Linear(model_efficientnetb4.classifier[1].in_features, num_classes)
model_efficientnetb4 = model_efficientnetb4.to(device)

# ✅ Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model_efficientnetb4.parameters(), lr=0.001, weight_decay=1e-4)
scaler = torch.cuda.amp.GradScaler()  # Mixed Precision Training
```

8. Proposed model Confusion Matrix :

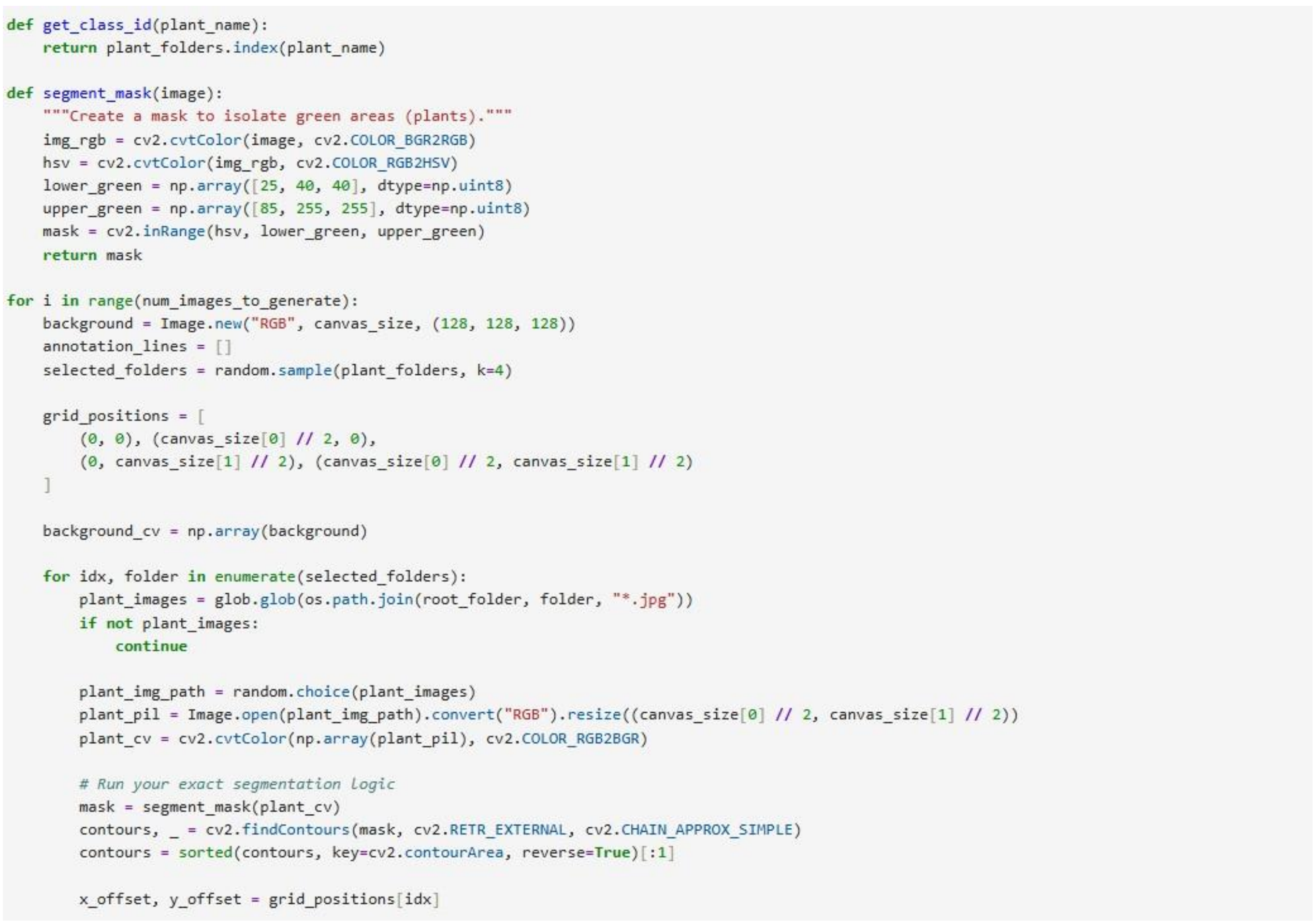


```
import tensorflow as tf
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score


# Get class labels
class_labels = list(train_generator.class_indices.keys())

# Predict on test data
y_pred = model.predict(test_generator)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = test_generator.classes

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# Compute accuracy
accuracy = accuracy_score(y_true, y_pred_classes)

# Generate classification report
report = classification_report(y_true, y_pred_classes, target_names=class_labels)

# Display accuracy
print(f"Overall Accuracy: {accuracy:.2%}")
print("\nClassification Report:")
print(report)

# Plot confusion matrix with better visualization
plt.figure(figsize=(15, 12))  # Increased size
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels, yticklabels=class_labels, linewidths=0.5)

plt.xlabel("Predicted Label", fontsize=12)
plt.ylabel("True Label", fontsize=12)
plt.xticks(rotation=90)  # Rotate x-axis labels for better readability
plt.yticks(rotation=0)
plt.title("Confusion Matrix", fontsize=14)

plt.show()
```

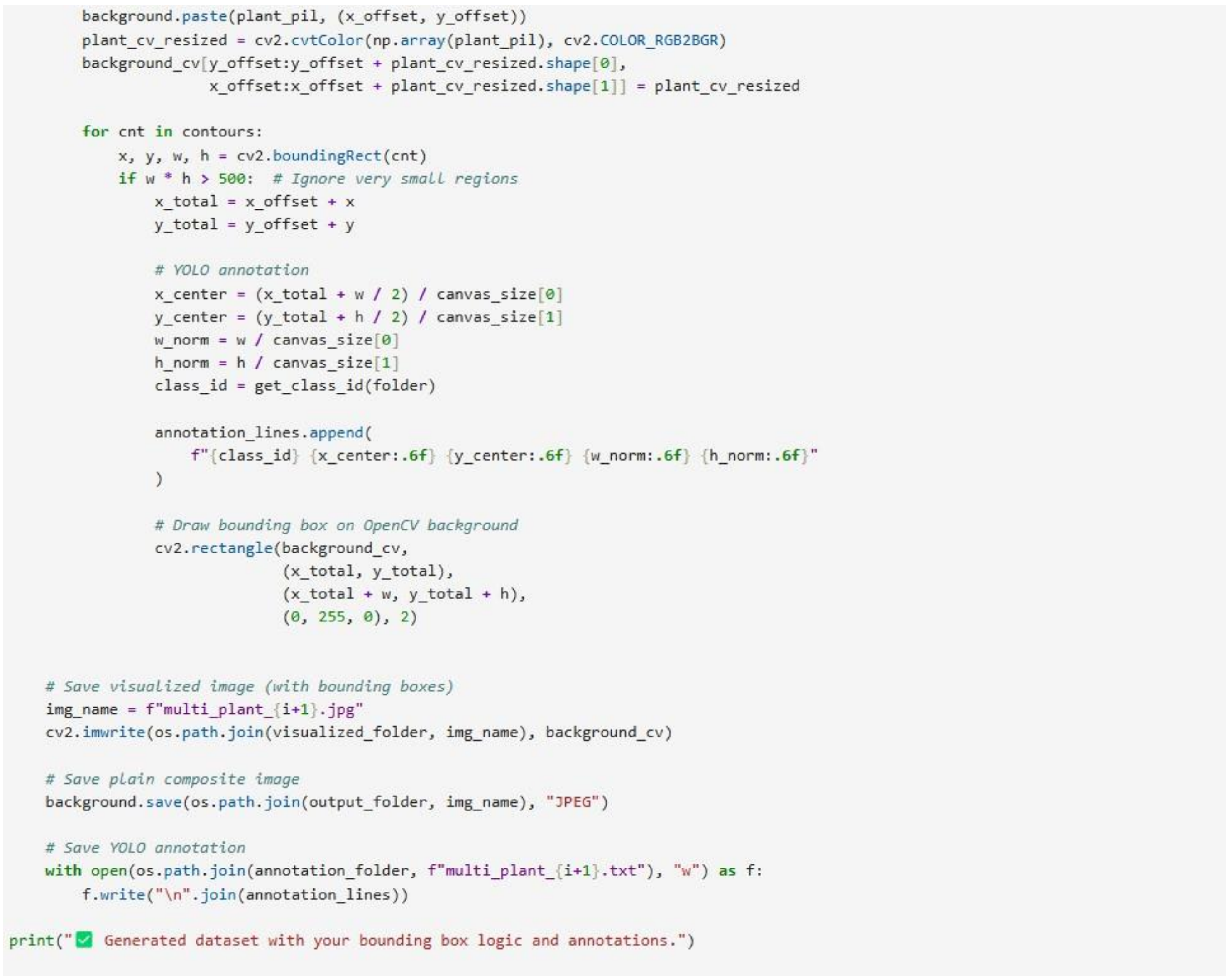
9. Multi Plant Dataset Generation:



```
import os
import random
import glob
import cv2
import numpy as np
from PIL import Image

# Paths
root_folder = "/kaggle/input/indian-medicinal-leaves-dataset/Indian Medicinal Leaves Image Datasets/Medicinal plant dataset"
output_folder = "/kaggle/working/multiPlant_dataset"
annotation_folder = "/kaggle/working/multiPlant_annotations"
visualized_folder = "/kaggle/working/multiPlant_visualized"

os.makedirs(output_folder, exist_ok=True)
os.makedirs(annotation_folder, exist_ok=True)
os.makedirs(visualized_folder, exist_ok=True)

plant_folders = sorted([
    folder for folder in os.listdir(root_folder)
    if os.path.isdir(os.path.join(root_folder, folder))
])

canvas_size = (800, 800)
num_images_to_generate = 1000
```



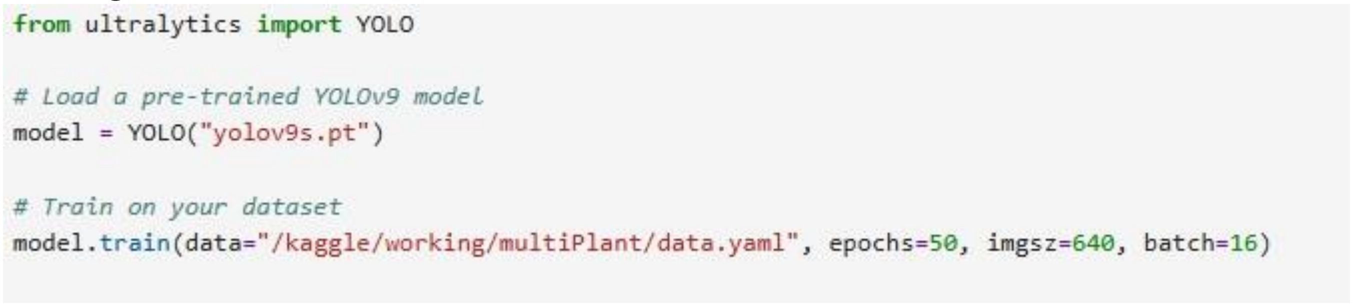
```
def get_class_id(plant_name):
    return plant_folders.index(plant_name)

def segment_mask(image):
    """Create a mask to isolate green areas (plants)."""
    img_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    hsv = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2HSV)
    lower_green = np.array([25, 40, 40], dtype=np.uint8)
    upper_green = np.array([85, 255, 255], dtype=np.uint8)
    mask = cv2.inRange(hsv, lower_green, upper_green)
    return mask

for i in range(num_images_to_generate):
    background = Image.new("RGB", canvas_size, (128, 128, 128))
    annotation_lines = []
    selected_folders = random.sample(plant_folders, k=4)

    grid_positions = [
        (0, 0), (canvas_size[0] // 2, 0),
        (0, canvas_size[1] // 2), (canvas_size[0] // 2, canvas_size[1] // 2)
    ]

    background_cv = np.array(background)

    for idx, folder in enumerate(selected_folders):
        plant_images = glob.glob(os.path.join(root_folder, folder, "*.jpg"))
        if not plant_images:
            continue

        plant_img_path = random.choice(plant_images)
        plant_pil = Image.open(plant_img_path).convert("RGB").resize((canvas_size[0] // 2, canvas_size[1] // 2))
        plant_cv = cv2.cvtColor(np.array(plant_pil), cv2.COLOR_RGB2BGR)

        # Run your exact segmentation logic
        mask = segment_mask(plant_cv)
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        contours = sorted(contours, key=cv2.contourArea, reverse=True)[:1]

        x_offset, y_offset = grid_positions[idx]
```



```
        background.paste(plant_pil, (x_offset, y_offset))
        plant_cv_resized = cv2.cvtColor(np.array(plant_pil), cv2.COLOR_RGB2BGR)
        background_cv[y_offset:y_offset + plant_cv_resized.shape[0],
                      x_offset:x_offset + plant_cv_resized.shape[1]] = plant_cv_resized

        for cnt in contours:
            x, y, w, h = cv2.boundingRect(cnt)
            if w * h > 500:  # Ignore very small regions
                x_total = x_offset + x
                y_total = y_offset + y

                # YOLO annotation
                x_center = (x_total + w / 2) / canvas_size[0]
                y_center = (y_total + h / 2) / canvas_size[1]
                w_norm = w / canvas_size[0]
                h_norm = h / canvas_size[1]
                class_id = get_class_id(folder)

                annotation_lines.append(
                    f"{class_id} {x_center:.6f} {y_center:.6f} {w_norm:.6f} {h_norm:.6f}"
                )

                # Draw bounding box on OpenCV background
                cv2.rectangle(background_cv,
                              (x_total, y_total),
                              (x_total + w, y_total + h),
                              (0, 255, 0), 2)


    # Save visualized image (with bounding boxes)
    img_name = f"multi_plant_{i+1}.jpg"
    cv2.imwrite(os.path.join(visualized_folder, img_name), background_cv)

    # Save plain composite image
    background.save(os.path.join(output_folder, img_name), "JPEG")

    # Save YOLO annotation
    with open(os.path.join(annotation_folder, f"multi_plant_{i+1}.txt"), "w") as f:
        f.write("\n".join(annotation_lines))

print("✅ Generated dataset with your bounding box logic and annotations.")
```

10. YOLOv9s :

Training the model:



```
from ultralytics import YOLO

# Load a pre-trained YOLOv9 model
model = YOLO("yolov9s.pt")

# Train on your dataset
model.train(data="/kaggle/working/multiPlant/data.yaml", epochs=50, imgsz=640, batch=16)
```

Detecting image classes:

```python
import matplotlib.pyplot as plt
from ultralytics import YOLO
import cv2
import os
import glob

# Load trained model
model = YOLO("runs/detect/train/weights/best.pt")

# Perform inference on test images
results = model.predict(source="/kaggle/working/multiPlant/images/test", save=True)

# Get saved images path
predicted_images = sorted(glob.glob("runs/detect/predict/*.jpg"))  # Adjust path if needed

# Display results with spacing
num_images = len(predicted_images)
cols = 2  # Number of columns in the grid
rows = (num_images // cols) + (num_images % cols > 0)  # Calculate rows

fig, axes = plt.subplots(rows, cols, figsize=(25, 10 * rows))  # Adjust figure size dynamically

for i, img_path in enumerate(predicted_images[:20]):
    img = cv2.imread(img_path)  # Read image
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert to RGB
    img_name = os.path.basename(img_path)  # Get filename

    ax = axes[i // cols, i % cols] if rows > 1 else axes[i % cols]  # Adjust for single row
    ax.imshow(img)
    ax.axis("off")  # Hide axis
    ax.set_title(img_name, fontsize=30)  # Set title as image filename

# Hide unused subplots if any
for j in range(i + 1, rows * cols):
    fig.delaxes(axes[j // cols, j % cols] if rows > 1 else axes[j % cols])

plt.tight_layout()
plt.show()
```

11. Retina Net:

```python
import os
import time
import torch
import pandas as pd
from PIL import Image
from torchvision.models.detection import retinanet_resnet50_fpn
from torchvision.models.detection.retinanet import RetinaNetClassificationHead
from torchvision.transforms import functional as F
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import LabelEncoder
import torchvision
import numpy as np
```

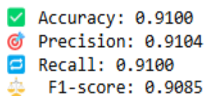
```python
def get_transform():
    return torchvision.transforms.Compose([
        torchvision.transforms.ToTensor()
    ])

# Dataset and DataLoader
dataset = PlantDataset(df, transforms=get_transform())
data_loader = DataLoader(dataset, batch_size=4, shuffle=True, collate_fn=lambda x: tuple(zip(*x)))

# RetinaNet model
num_classes = len(sorted_class_names)
model = retinanet_resnet50_fpn(pretrained=True)
in_features = model.head.classification_head.conv[0][0].in_channels
num_anchors = model.head.classification_head.num_anchors
model.head.classification_head = RetinaNetClassificationHead(in_features, num_anchors, num_classes)

# Move model to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

# Optimizer and learning rate
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.AdamW(params, lr=5e-5, weight_decay=1e-4)
```

```python
num_epochs = 50
model.train()
for epoch in range(num_epochs):
    epoch_start = time.time()
    epoch_loss = 0.0
    classification_loss_total = 0.0
    regression_loss_total = 0.0

    for images, targets in data_loader:
        images = [img.to(device) for img in images]
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

        epoch_loss += losses.item()
        classification_loss_total += loss_dict['classification'].item()
        regression_loss_total += loss_dict['bbox_regression'].item()

    epoch_time = time.time() - epoch_start
    print(f"Epoch {epoch + 1}/{num_epochs} - Total Loss: {epoch_loss:.4f} | "
          f"Cls Loss: {classification_loss_total:.4f} | Reg Loss: {regression_loss_total:.4f} | "
          f"Time: {epoch_time:.2f}s")
```

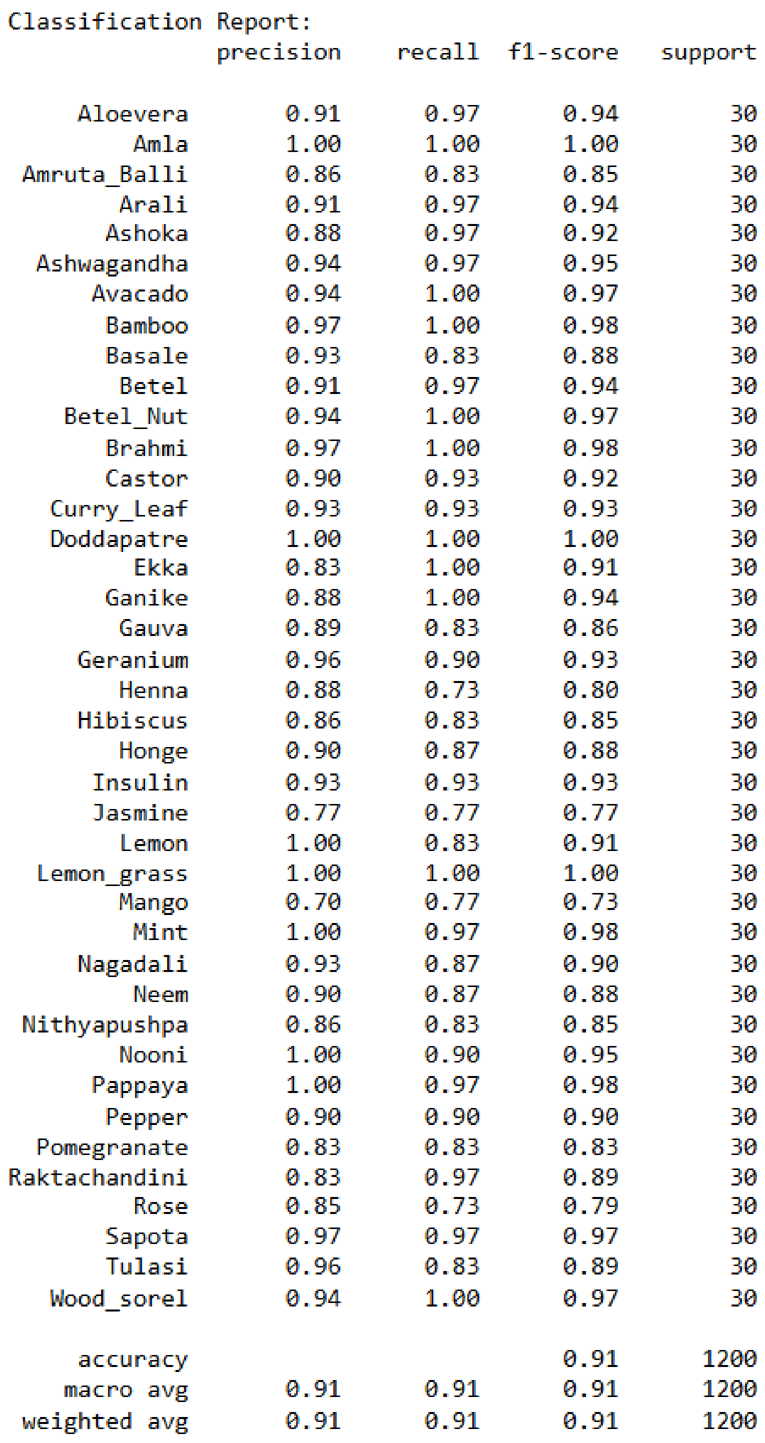
CHAPTER 6

PERFORMANCE EVALUATION

Deep-Medic



```
✅ Accuracy: 0.9100
🎯 Precision: 0.9104
🔁 Recall: 0.9100
⚖️  F1-score: 0.9085
```



```
Classification Report:
               precision    recall  f1-score   support

     Aloevera       0.91      0.97      0.94        30
         Amla       1.00      1.00      1.00        30
  Amruta_Balli       0.86      0.83      0.85        30
        Arali       0.91      0.97      0.94        30
       Ashoka       0.88      0.97      0.92        30
   Ashwagandha       0.94      0.97      0.95        30
      Avacado       0.94      1.00      0.97        30
       Bamboo       0.97      1.00      0.98        30
       Basale       0.93      0.83      0.88        30
        Betel       0.91      0.97      0.94        30
    Betel_Nut       0.94      1.00      0.97        30
       Brahmi       0.97      1.00      0.98        30
       Castor       0.90      0.93      0.92        30
    Curry_Leaf       0.93      0.93      0.93        30
    Doddapatre       1.00      1.00      1.00        30
         Ekka       0.83      1.00      0.91        30
       Ganike       0.88      1.00      0.94        30
        Gauva       0.89      0.83      0.86        30
     Geranium       0.96      0.90      0.93        30
        Henna       0.88      0.73      0.80        30
     Hibiscus       0.86      0.83      0.85        30
        Honge       0.90      0.87      0.88        30
      Insulin       0.93      0.93      0.93        30
      Jasmine       0.77      0.77      0.77        30
        Lemon       1.00      0.83      0.91        30
   Lemon_grass       1.00      1.00      1.00        30
        Mango       0.70      0.77      0.73        30
         Mint       1.00      0.97      0.98        30
     Nagadali       0.93      0.87      0.90        30
         Neem       0.90      0.87      0.88        30
  Nithyapushpa       0.86      0.83      0.85        30
        Nooni       1.00      0.90      0.95        30
      Pappaya       1.00      0.97      0.98        30
       Pepper       0.90      0.90      0.90        30
   Pomegranate       0.83      0.83      0.83        30
Raktachandini       0.83      0.97      0.89        30
         Rose       0.85      0.73      0.79        30
       Sapota       0.97      0.97      0.97        30
       Tulasi       0.96      0.83      0.89        30
    Wood_sorel       0.94      1.00      0.97        30

     accuracy                           0.91      1200
    macro avg       0.91      0.91      0.91      1200
  weighted avg       0.91      0.91      0.91      1200
```

Fig. 9. Performance metrics Deep-Medic with test sets with cross-validation.

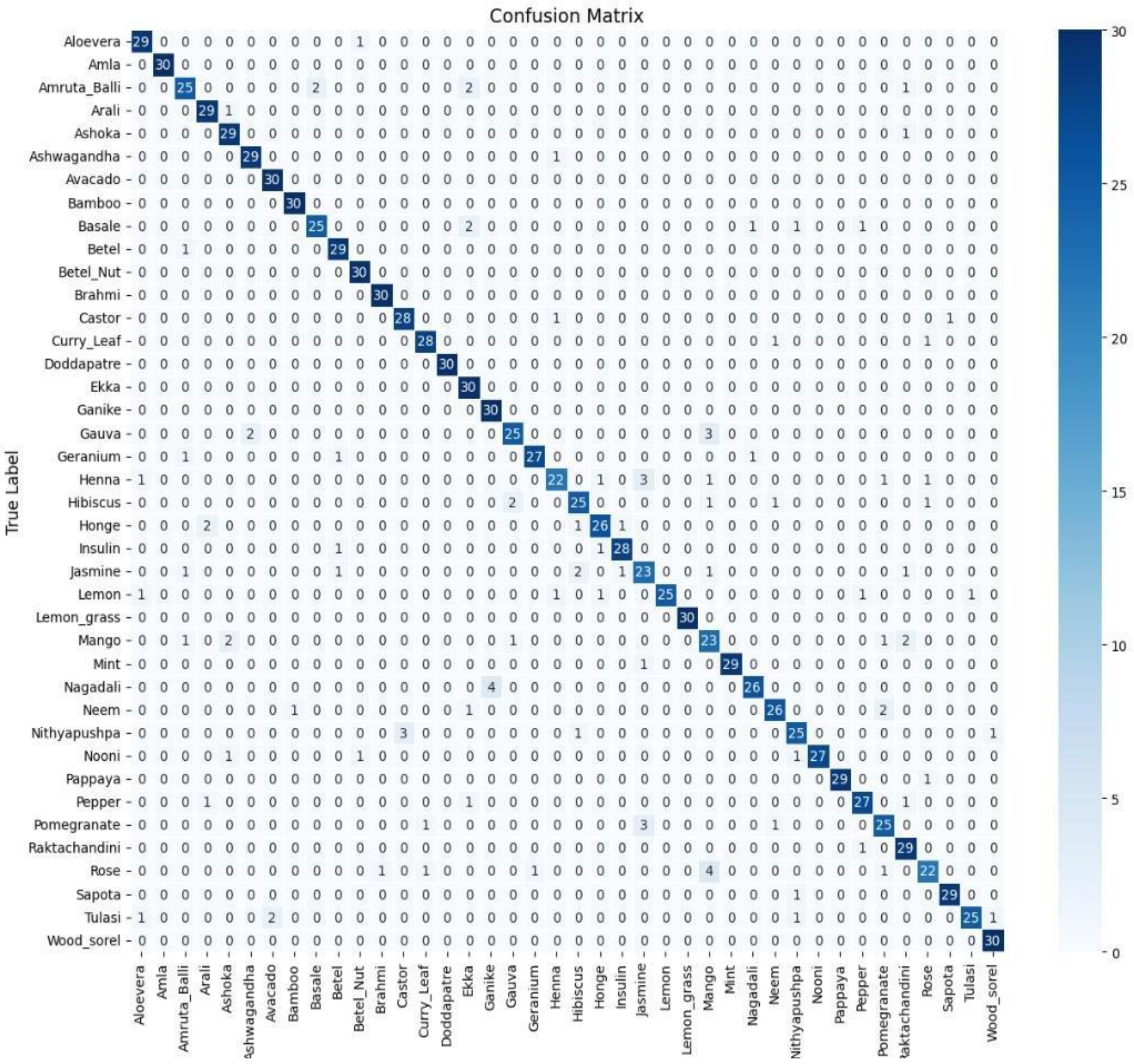


Fig. 10. Confusion matrix of random samples in test set.

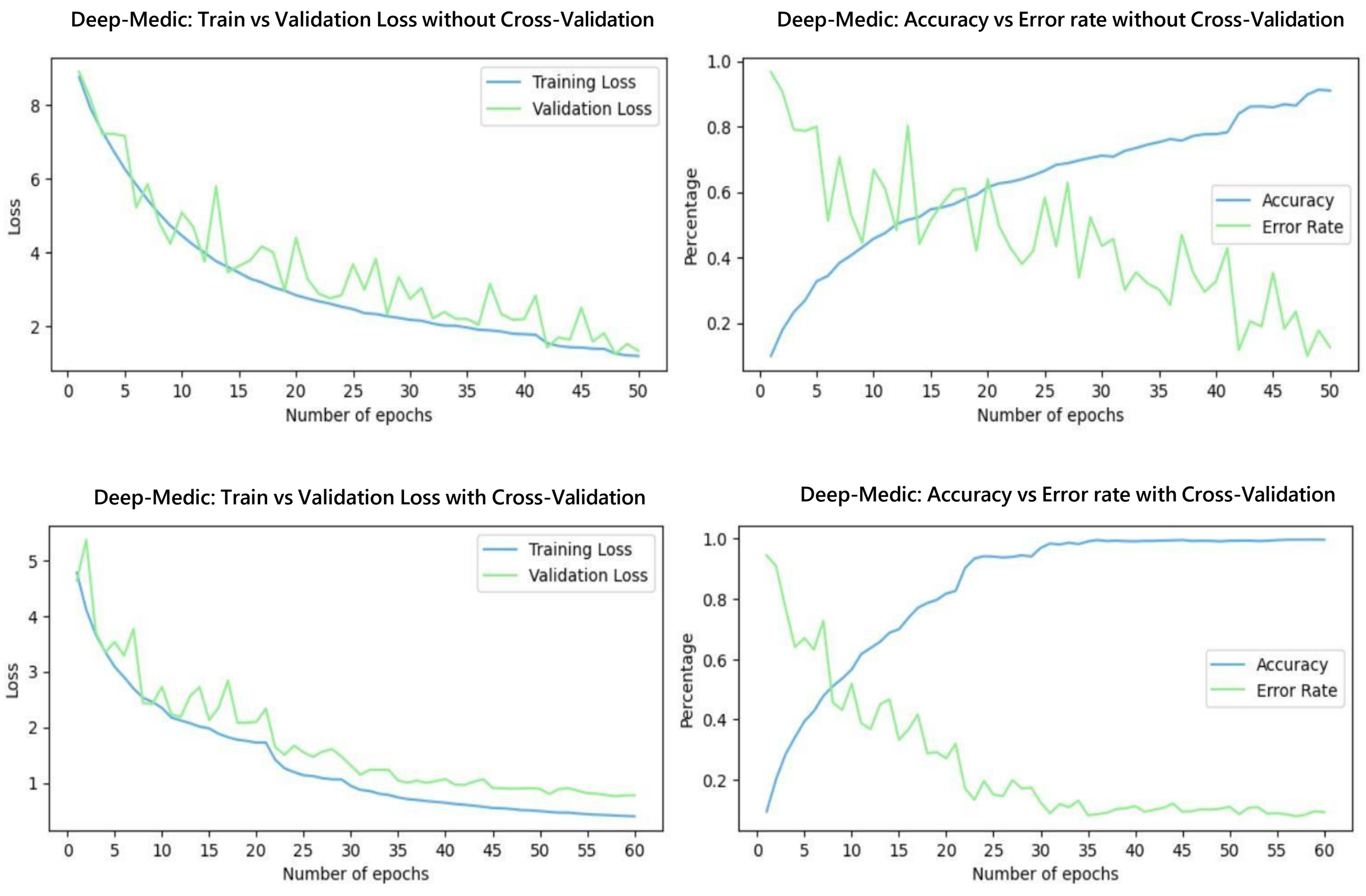


Fig. 11. Training dataset performance of Deep-Medic

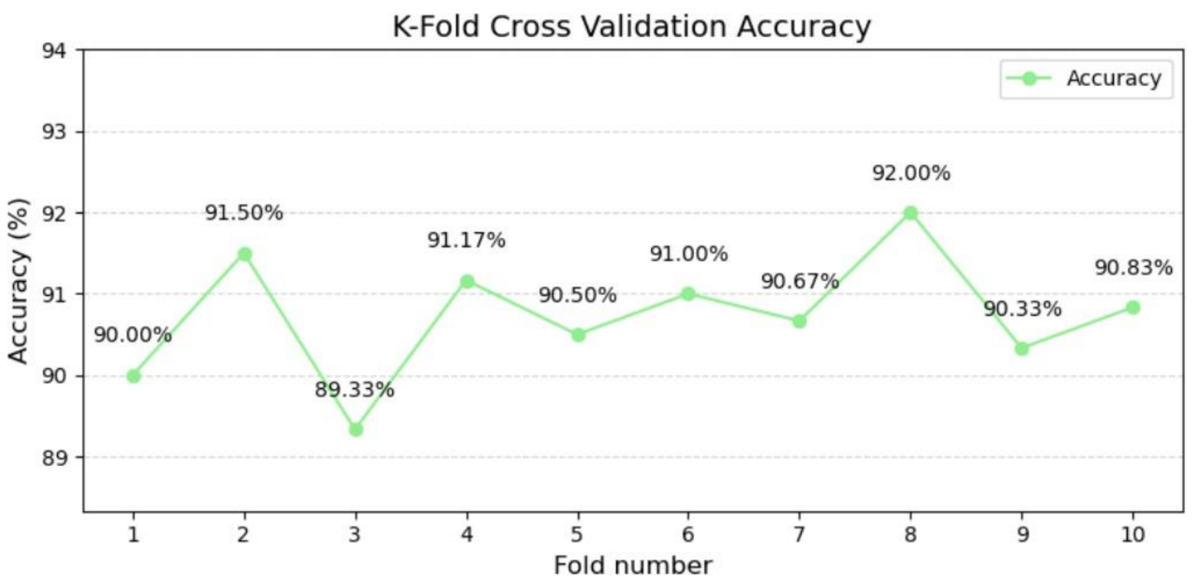


Fig. 12. K-fold cross validation with proposed Deep-Medic model with K=10

Table 2: Medicinal plant species with inter-class similarities in leaf shapes and structure

Group	Plant species 1	Plant species 2	Plant species 3	Plant species 4
Group 1	Ashoka	Guava	Mango	Nooni
Group 2	Amla	Curry Leaf	Neem	-
Group 3	Pepper	Betel	-	-
Group 4	Ashwagandha	Ekka	-	-
Group 5	Castor	Papaya	-	-
Group 6	Brahmi	Geranium	-	-
Group 7	Lemon	Avocado	Henna	-
Group 8	Amruta Balli	Basale	-	-

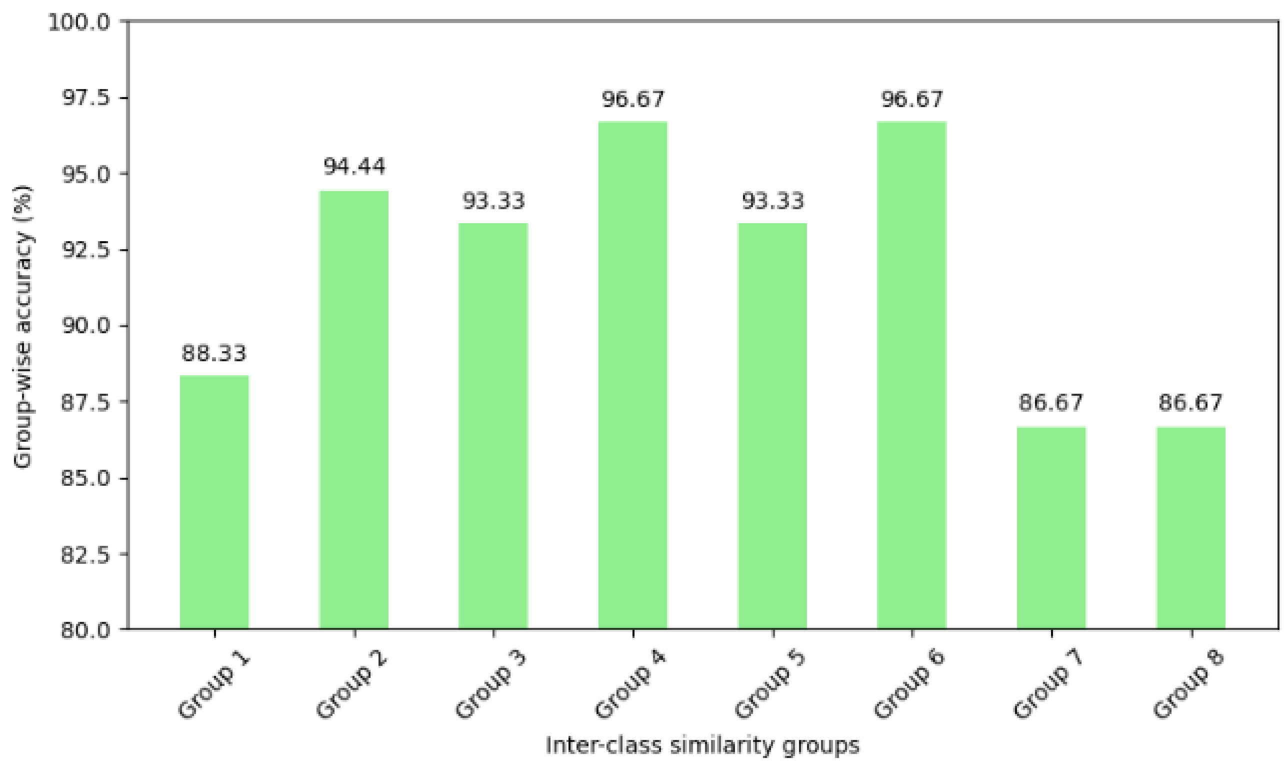


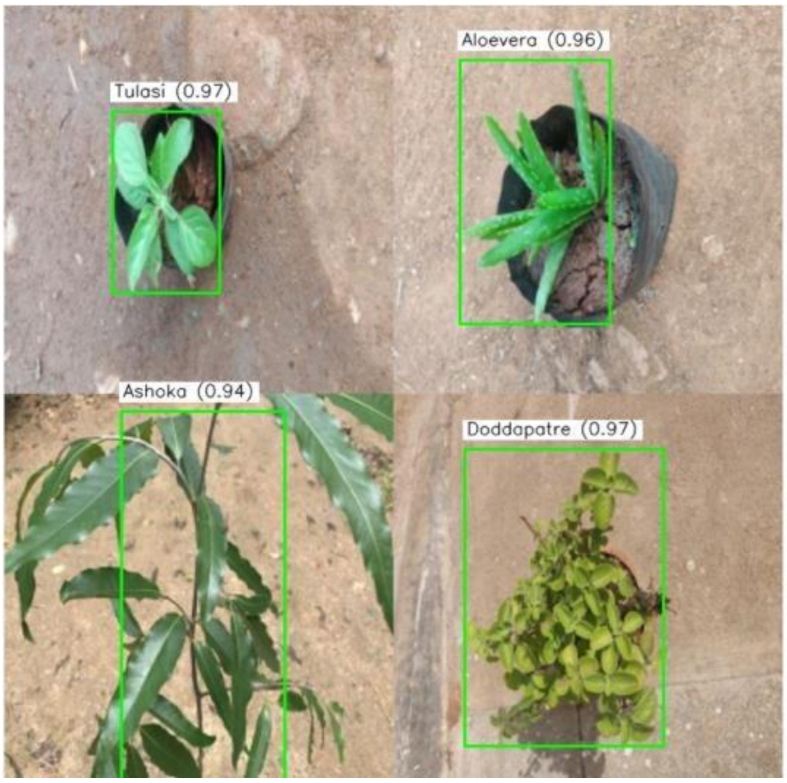
Fig. 13. Accuracy obtained over test set for inter-class similarity group using Deep-Medic

Table 3: Comparison of proposed method with other pre-trained models without K-fold

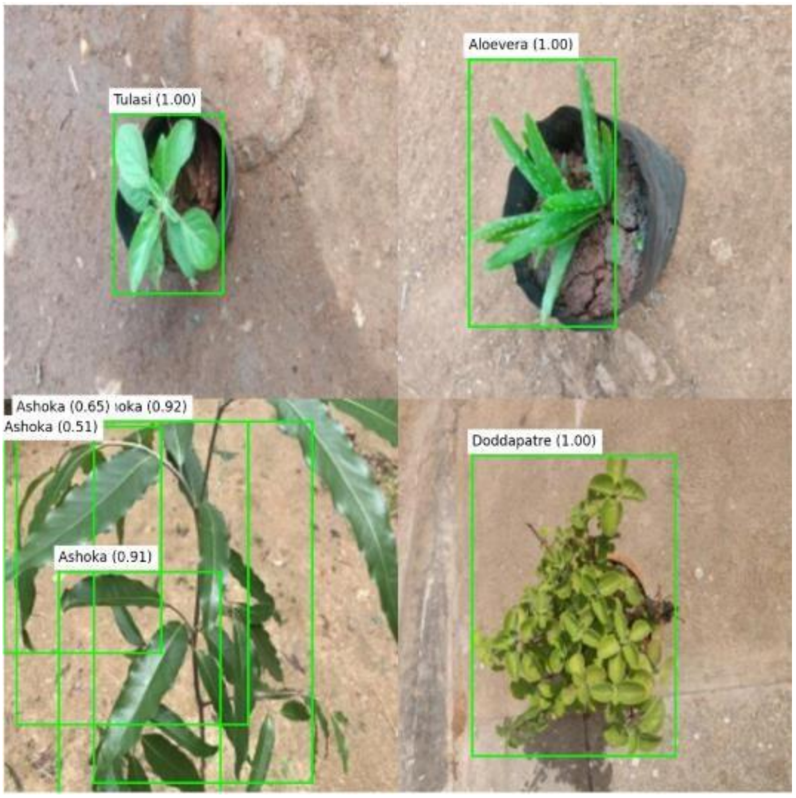
Model	Accuracy (%)	Precision	Recall	F1-Score
Resnet34	89.08	90.59	89.08	89.26
Resnet50	92.83	93.68	92.83	92.79
MobileNetV3-Large	92.83	94.04	92.83	92.84
DesneNet121	94.25	95.0	94.25	94.04
VGG16	78.0	82.0	78.0	78.0
EfficientNet-B4	86.0	89.0	86.0	86.0
Deep-Medic (proposed method)	91.0	91.04	91.0	90.85

Table 4: Comparison of proposed method with other pre-trained models with K-fold

Model	Accuracy (%)	Precision	Recall	F1-Score
Resnet34	89.18	90.22	89.18	89.1
Resnet50	94.48	94.86	94.48	94.4
MobileNetV3-Large	93.95	94.5	93.95	93.93
DesneNet121	94.25	95.0	94.25	94.04
VGG16	71.67	72.67	71.6	78.0
EfficientNet-B4	88.72	88.83	88.72	88.43
Deep-Medic (proposed method)	95.5	96.05	95.5	95.51



(a)



(b)

Fig. 14. (a) YOLO predicted image (b) RetinaNet predicted image

CHAPTER 7

RESULTS AND DISCUSSION

7.1. Single plant classification:

To analyze the efficiency of Deep-Medic model, a test dataset was created with 1200 random samples considering 30 images from each class as forty plant species are available in the dataset. Dataset was taken from kaggle, where each class contains of 100 to 150 images. To balance the dataset, Augmentation was done to make each class with 150 samples with a total of 6000. Training was done with 4800 samples and testing with 1200 samples (80:20 ratio). K-fold cross validation was carried out to validate the proposed model with K=10. Torch and tensorflow libraries, machine learning computational frameworks were used in this. Training of the models was performed on a dual NVIDIA T4 GPU setup (2 * T4, 16 GB each) with RAM of 29 GB and disk space of 57.6 GB per session. The computational environment was managed via Kaggle notebooks, which offer cloud-based Jupyter Notebook support.

Fig.11. shows the plots of training vs validation loss and accuracy vs error rate was done with and without K-fold. To reduce the overfitting issues, the model was trained until 60 epochs. With K-fold and without K-fold cross validations, the classification accuracies are 95% and 94.6% respectively. In the initial stage of training, Deep-Medic model's behavior was unpredictable from first epoch until twenty epochs. The learning is unstable, and the training, validation losses were fluctuating (sudden increase and sudden decrease) in both with and without K-fold. However, from the 25th epoch, loss has been decreasing and accuracy started increasing. The model became stable in between 45 to 60 epochs. Confusion matrix i.e., heat map depicts the performance of the proposed model with cross-validation for all 40 Medicinal plant species. Here, out of 40 plant species, 21 were classified with 100%, 9 were classified with greater than 95%, 5 were classified in between 90 to 94%, and 5 were classified in between 80 to 90%, keeping the number of misclassifications up to 1 or 2.

The performance of the model shows that the augmentation and segmentation steps have shown a greater impact during training. The test set was evaluated using the metrics such as accuracy, precision, recall, f1-score, presented in Fig.9. From that, the precision, F1-score, and accuracy provide a better intuition of predicted values with respect to each class. The precision is very important for examining the classification accuracy since it assists in interpreting the ratio of the true positives samples from out of all positively predicted samples. The precision value ranges between 0 to 1; the greater the precision more reliable the accuracy achieved. On the other hand, recall is the ratio of predicted positive out of total positive samples. In our work, the precision and recall achieved are consistently over 85% for 37 classes out of 40, which shows a high true positive rate in predicted results. The other three classes, namely Jasmine, Raktachandini, and rose, have a precision equal to an average of 0.77. The recall rate for the said three classes is 0.96, 1.00, and 0.96. This indicates that although the precision is low, the recall rate is high with respect to these classes leading to a lower false negative rate. Consequently, the suggested model predicts true positive results with over 0.85 precision and recall. Moreover, the F1-score, the harmonic mean of precision and recall, further establishes that precision and recall are always greater than 0.80 for all 40 classes in case of the predicted output.

As illustrated in Fig.13., the performance of Deep-Medic on inter class similarity groups, group 4 with ashwagandha and ekka species, group 5 having Brahmi and Geranium were classified with an overall accuracy of 0.96. Consequently, the accuracy of classifying all other groups stands above 88% except for group 7 and group 8, which is 86.67%. Results from experimentations conducted using K-fold cross validation with K=10 shown in Fig.12. reveals that the accuracies achieved ranges between 89.33% and 92 % from K=1 to K=10. For K=1, it is observed that the model is 90% accurate; subsequently, at K=8, it achieves a maximum gain accuracy of as high as 92%. Subsequently the accuracy continued decreasing to the lowest point of 90.83%, as observed at K=10. The performance is compared to six CNN architectures to identify 40 plant species. The pre trained models were tested to compare the efficiency of proposed Deep-Medic. Table 3 illustrates the performance evaluation metrics of the six pre-trained models without cross-validation, and Table 4 shows with cross-validation. Though numerous works reported using deep convolutional neural networks, the works that assume the image samples obtained with changing camera configurations and varying resolutions are taken into consideration.

It is noticed that, without cross validation, the proposed method gives predicted outcomes close to the pre-trained models, DenseNet121, ResNet50, and MobileNetV3. The least-performing model is VGG16, obtaining an accuracy of 82%. Though DenseNet121 produces an accuracy of 94.25%, the number of plant species with misclassifications per class more significant than 5 is 3. In the case of Deep-Medic, the number of misclassifications per class is less than 5 for all thirty-nine plant species except one. A similar performance is also noticed in MobileNetV3 and Resnet50, with misclassification per class is kept at less. Therefore, the experimentations are also performed with K-fold cross-validation with test sets involving proposed vs. pre-trained models. As per Table 4, accuracy of 95.5% is obtained by cross-validation. The ablation study demonstrates that the proposed model is superior over the pre-trained models. The results indicate that the ResNet50, DenseNet121 and MobileNetV3 achieve an accuracy of 92.83%, 94.25%, and 92.83 which is lower than predicted results.

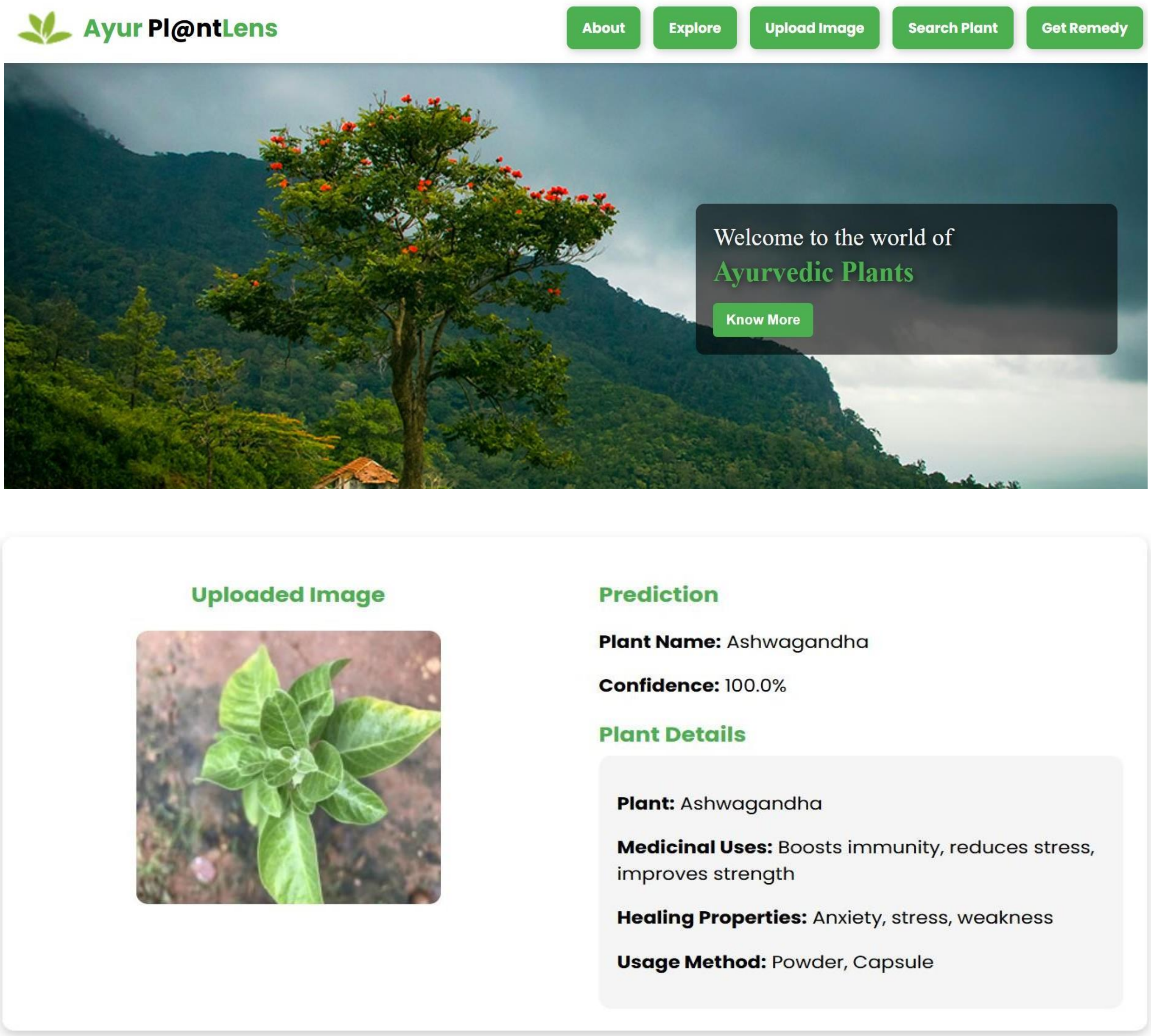
Additionally, the number of trainable parameters learnt by the model is 5,142,184 which is significantly less than all other pre-trained models except MobileNetV3 but gaining more accuracy than all the pre-trained models. Refer Table 1. Therefore, the proposed Deep-Medic is computationally efficient i.e., reduced trainable parameters and time for training compared to all other models.

7.2. Multi plant classification

YOLOv9s was used initially for training the multi-plant dataset, this model was build using 197 layers and learnt 7,182,568 trainable parameters. The evaluated metrics are precision of 93.36%, recall of 91.14% and mean average precision (mAP) at IoU with 0.5 is 95.03%. See Fig.10. And another object detection model named RetinaNet is used for multi-plant classification, it achieved better results when compared to YOLOv9. Detected and classified with 98% accuracy and precision.

7.3. User Interface

A user-friendly web application for the identification and exploration of medicinal plants was created with the Flask framework as part of this project. The application enables users to upload plant images and obtain real-time predictions of the plant species based on a trained deep learning model. Apart from identification, the system also gives comprehensive details of different medicinal plants, such as their pictures, medical properties, usage methods, and the diseases they cure. Users can even search for a particular medicinal plant to fetch its therapeutic details or can give the name of a disease to get recommendations of appropriate medicinal plants which can be used as remedies. This combination of deep learning and traditional medicinal wisdom illustrates the value of combining AI-based classification with traditional practice in a real-world and accessible format, specific to this project-based application.



Plant Name: Ashwagandha

Confidence: 100.0%

Plant: Ashwagandha

Medicinal Uses: Boosts immunity, reduces stress, improves strength

Healing Properties: Anxiety, stress, weakness

Usage Method: Powder, Capsule

Fig. 15. User Interface

CHAPTER 8

CONCLUSION AND FUTURE PLANS

In this research work, an Medicinal plant species classification system based on deep learning, AyurVision, was proposed for both single-plant and multi-plant situations. For single-plant classification, the dataset was obtained from publicly available collections, whereas a specially created multi-plant dataset was used because no public resources were found. The images were taken with smartphone cameras under diverse environmental conditions, such as varying resolutions, occlusions, and varying illumination. To improve model generalization and minimize bias and variance, aggressive data augmentation methods were utilized, such as geometric and intensity-based augmentations. Furthermore, color channel segmentation was utilized to separate and classify meaningful regions in intricate images, thereby minimizing computational overhead and the number of learnable parameters.

In summary, Deep-Medic was an effective and dependable solution for Medicinal plant identification. Its performance was superior to standard architectures such as ResNet50 and was on par with DenseNet121 and MobileNetV3 with lower computational complexity. For multi-plant classification, two recent object detection models YOLOv9 and RetinaNet were utilized and compared. The experimental results indicate that RetinaNet always performed better than YOLOv9 in terms of accuracy and detection consistency and thus is more appropriate for cases with overlapping and densely clustered plants. Here, the dataset of multi-plants was synthesized by putting individual plant images together. For future work, actual real-time multi-plant images taken from nature can be applied to strengthen the model and equip it with a capability to encounter real-life obstacles like background noise, occlusion, overlapping leaves, and light variations.

CHAPTER 9

REFERENCES

[1] Pushpa BR, Rani NS. Ayur-PlantNet: An unbiased light weight deep convolutional neural network for Indian Medicinal plant species classification. Journal of Applied Research on Medicinal and Aromatic Plants. 2023 Apr 1;34:100459.
http://dx.doi.org/10.1016/j.jarmap.2023.100459

[2] Anubha Pearline S, Sathiesh Kumar V, Harini S. A study on plant recognition using conventional image processing and deep learning approaches. Journal of Intelligent & Fuzzy Systems. 2019 Jan 1;36(3):1997-2004.
http://dx.doi.org/10.3233/JIFS-169911

[3] Azadnia R, Kheiralipour K. Recognition of leaves of different medicinal plant species using a robust image processing algorithm and artificial neural networks classifier. Journal of Applied Research on Medicinal and Aromatic Plants. 2021 Dec 1;25:100327.
http://dx.doi.org/10.1016/j.jarmap.2021.100327

[4] Yigit E, Sabanci K, Toktas A, Kayabasi A. A study on visual features of leaves in plant identification using artificial intelligence techniques. Computers and electronics in agriculture. 2019 Jan 1;156:369-77.
http://dx.doi.org/10.1016/j.compag.2018.11.036

[5] Patil S, Sasikala M. Segmentation and identification of medicinal plant through weighted KNN. Multimedia Tools and Applications. 2023 Jan;82(2):2805-19.
http://dx.doi.org/10.1007/s11042-022-13201-7

[6] Sharma M, Kumar N, Sharma S, Kumar S, Singh S, Mehandia S. Medicinal plants recognition using heterogeneous leaf features: an intelligent approach. Multimedia Tools and Applications. 2024 May;83(17):51513-40.
http://dx.doi.org/10.1007/s11042-023-17639-1

[7] Yang K, Zhong W, Li F. Leaf segmentation and classification with a complicated background using deep learning. Agronomy. 2020 Nov 6;10(11):1721.
http://dx.doi.org/10.3390/agronomy10111721

[8] Sharma S, Vardhan M. AELGNet: Attention-based Enhanced Local and Global Features Network for medicinal leaf and plant classification. Computers in Biology and Medicine. 2025 Jan 1;184:109447.
http://dx.doi.org/10.1016/j.compbiomed.2024.109447

[9] Wei Tan J, Chang SW, Abdul-Kareem S, Yap HJ, Yong KT. Deep learning for plant species classification using leaf vein morphometric. IEEE/ACM transactions on computational biology and bioinformatics. 2018 Jun 19;17(1):82-90.

http://dx.doi.org/10.1109/TCBB.2018.2848653

[10] Kavitha S, Kumar TS, Naresh E, Kalmani VH, Bamane KD, Pareek PK. Medicinal plant identification in real-time using deep learning model. SN Computer Science. 2023 Dec 7;5(1):73. http://dx.doi.org/10.1007/s42979-023-02398-5

[11] El Akhal H, Yahya AB, Moussa N, El Alaoui AE. A novel approach for image-based olive leaf diseases classification using a deep hybrid model. Ecological Informatics. 2023 Nov 1;77:102276. http://dx.doi.org/10.1016/j.ecoinf.2023.102276

[12] Naeem S, Ali A, Chesneau C, Tahir MH, Jamal F, Sherwani RA, Ul Hassan M. The classification of medicinal plant leaves based on multispectral and texture feature using machine learning approach. Agronomy. 2021 Jan 30;11(2):263.
http://dx.doi.org/10.3390/agronomy11020263

[13] Javid A, Haghirosadat BF. A review of medicinal plants effective in the treatment or apoptosis of cancer cells. Cancer Press Journal. 2017 Mar 29;3(1):22-6.
http://dx.doi.org/10.15562/tcp.41

[14] Dileep MR, Pournami PN. AyurLeaf: a deep learning approach for classification of medicinal plants. InTENCON 2019-2019 IEEE Region 10 Conference (TENCON) 2019 Oct 17 (pp. 321-325). IEEE.
http://dx.doi.org/10.1109/TENCON.2019.8929394

[15] Sachar S, Kumar A. Deep ensemble learning for automatic medicinal leaf identification. International Journal of Information Technology. 2022 Oct;14(6):3089-97.
http://dx.doi.org/10.1007/s41870-022-01055-z

[16] Manoharan JS. Flawless detection of herbal plant leaf by machine learning classifier through two stage authentication procedure. Journal of Artificial Intelligence and Capsule Networks. 2021 Jun 22;3(2):125-39.
http://dx.doi.org/10.36548/jaicn.2021.2.005

[17] Zin IA, Ibrahim Z, Isa D, Aliman S, Sabri N, Mangshor NN. Herbal plant recognition using deep convolutional neural network. Bulletin of Electrical Engineering and Informatics. 2020 Oct 1;9(5):2198-205.
http://dx.doi.org/10.11591/eei.v9i5.2250

[18] Anami BS, Nandyal SS, Govardhan A. A combined color, texture and edge features based approach for identification and classification of indian medicinal plants. International Journal of Computer Applications. 2010 Sep;6(12):45-51.
http://dx.doi.org/10.5120/1122-1471

[19] Azlah MA, Chua LS, Rahmad FR, Abdullah FI, Wan Alwi SR. Review on techniques for plant leaf classification and recognition. Computers. 2019 Oct 21;8(4):77.

http://dx.doi.org/10.3390/computers8040077

[20] Sivaranjani C, Kalinathan L, Amutha R, Kathavarayan RS, Kumar KJ. Real-time identification of medicinal plants using machine learning techniques. In2019 International Conference on Computational Intelligence in Data Science (ICCIDS) 2019 Feb 21 (pp. 1-4). IEEE. http://dx.doi.org/10.1109/ICCIDS.2019.8862126

[21] Chaudhury A, Barron JL. Plant species identification from occluded leaf images. IEEE/ACM transactions on computational biology and bioinformatics. 2018 Oct 4;17(3):1042-55. http://dx.doi.org/10.1109/TCBB.2018.2873611

[22] Kumar M, Gupta S, Gao XZ, Singh A. Plant species recognition using morphological features and adaptive boosting methodology. IEEE Access. 2019 Nov 7;7:163912-8. http://dx.doi.org/10.1109/ACCESS.2019.2952176

[23] Kan HX, Jin L, Zhou FL. Classification of medicinal plant leaf image based on multi-feature extraction. Pattern recognition and image analysis. 2017 Jul;27:581-7. http://dx.doi.org/10.1134/S105466181703018X

[24] Swaminathan A, Varun C, Kalaivani S. Multiple plant leaf disease classification using densenet-121 architecture. Int. J. Electr. Eng. Technol. 2021 May;12(5):38-57. http://dx.doi.org/10.34218/IJEET.12.5.2021.005