TRANSPARENT DEEP LEARNING TO DETECT BLIGHT THREATS IN POTATO (Solanum Tuberosum L.) AND TOMATO (Solanum Lycopersicum L.) CROPS

Report submitted to SASTRA Deemed to be University
As per the requirement for the course

CSE300 : MINI PROJECT

Submitted by

DHARUNA S
(125003070, B. Tech Computer Science and Engineering)
SOWBARANI V S
(125003332, B. Tech Computer Science and Engineering)
HARINI M
(125015043, B. Tech Information Technology)

May 2024

SCHOOL OF COMPUTING
THANJAVUR, TAMIL NADU, INDIA – 613 401

SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

SCHOOL OF COMPUTING
THANJAVUR – 613 401

Bonafide Certificate

This is to certify that the report titled **"Transparent Deep Learning to Detect Blight Threats in Potato (Solanum Tuberosum L.) and Tomato (Solanum Lycopersicum L.) Crops"** submitted as a requirement for the course, **CSE300 : MINI PROJECT** for B.Tech. is a bonafide record of the work done by **Ms. Dharuna S (125003070, B. Tech Computer Science and Engineering**) , **Ms. Sowbarani V S (125003332, B. Tech Computer Science and Engineering) and Ms. Harini M (125015043, B. Tech Information Technology)** during the academic year 2023-24, in the School of Computing, under my supervision.

Signature of Project Supervisor :
Name with Affiliation :
Date :

Mini Project *Viva voice* held on _____

Examiner 1 **Examiner 2**

ACKNOWLEDGEMENTS

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli**, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr. V. S. Shankar Sriram**, Dean, School of Computing, **Dr. R. Muthaiah**, Associate Dean, Research, **Dr. K. Ramkumar**, Associate Dean, Academics, **Dr. D. Manivannan**, Associate Dean, Infrastructure, **Dr. R. Algeswaran**, Associate Dean, Student Welfare

Our guide **Dr. Devika R**, Assistant Professor, School of Computing was the driving force behind this whole idea from the start. Her deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing me an opportunity to showcase my skills through this project.

List of Figures

List of Tables

Abbreviations

RGB	Red Green Blue
VGG	Visual Geometry Group
ReLU	Rectified Linear Unit
TPE	Tree-structured Parzen Estimator
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

Notations

English Symbols (in alphabetical order)

L	Loss.
Log	Logarithmic.

Greek Symbols (in alphabetical order)

σ	Variance.
Σ	Summation.

ABSTRACT

Blight disease in potato and tomato are caused by fungus (Alternaria Solani) which is principally a disease of aging plant tissue, that causes collapse and decay. Early and Late blight are the two diseases that frequently affects potato (Solanum Tuberosum L.) and tomato (Solanum Lycopersicum L.) crops. Detecting this disease at the earliest will help the farmers to act quickly on crops by using fungicides (Quinone, Azoxystrobin), to prevent further spreading of blight. A deep learning ResNet-9 model has been proposed to identify the condition of blight on which farmers can rely. Resnet-9 model considers the shape of the leaf, affected areas present and healthy areas of the leaf for its prediction. Provision of saliency maps which provide insights into the reasoning behind Resnet-9 model predictions is also provided. We will apply a rigorous hyperparameter optimization technique and enhance the training dataset.

Resnet-9 may have limitations in capturing intricate and hierarchical features due to its shallower architecture. Whereas Inception models, with their multi scale feature extraction through different kernel sizes, are designed to address this issue more effectively. Hence we propose a pretrained model named VGG-16 which follows a 13 CONV layers for testing. The proposed modelling framework contributes to deploying Convolutional Neural Network (CNN) models for proximal sensing image classification, aiding in early-stage leaf infection detection for crop protection. By implementing the proposed model, we aim to compare the gained accuracy with the Resnet-9 model. This proposed 32-step Deep Learning technique
is to be trained and validated on potato (Solanum Tuberosum L.) and tomato (Solanum Lycopersicum L.) plant leaves dataset.

KEY WORDS: Deep Learning, Pretrained models, VGG-16, Resnet-9

Table of Contents

CHAPTER 1
SUMMARY OF THE BASE PAPER

Title	**:**	Automatic Blight Disease Detection in Potato (Solanum Tuberosum L.) and Tomato (Solanum Lycopersicum, L. 1753) plants using Deep Learning.
Publisher	**:**	ScienceDirect
Year	**:**	2023
Journal name	**:**	Smart Agricultural Technology.
DOI	**:**	10.1016/j.atech.2023.100178
Base paper URL	**:**	https://www.sciencedirect.com/science/article/pii/S2772375523000084

The base paper's primary additions are:

- Introducing a powerful ResNet-9 model for detecting stages of blight disease in potato and tomato leaf pictures.
- Adapting VGG-16 model with its hyperparameters to the same task and contrasting with the suggested Resnet-9 model.
- Several models are implemented, and the accuracy of each model is noted.

Two main steps make up our method:

1.1 DATASET

We received our project's dataset from the open Plant Village Dataset. The dataset covers plant types such as apples, blueberries, cherries, corn, grapes, oranges, peaches, bell peppers, potatoes, raspberries, soybeans, squash, strawberries, and tomatoes.Images illustrating 17 fungal illnesses, 4 bacterial diseases, 2 mold (oomycete) diseases, 2 viral diseases, and 1 mite-related sickness are included in this collection. But the datasets for potatoes and tomatoes are the main focus of this effort.

The dataset's images were primarily captured by technicians within experimental labs across the United States. Leaves were positioned against a paper backdrop, typically gray or black. Each leaf underwent multiple image captures, ranging from 4 to 7, using a standard digital camera set to automatic mode. To capture various angles and disease manifestations, leaves were rotated 360 degrees during photography. Every picture in the collection is saved in JPEG format and is scaled to 256 pixels in both height and width.

Crop Type	Class	Number of Images	Total
Potato	Early Blight	1000	
	Healthy	152	2152
	Late Blight	1000	
Tomato	Early Blight	1000	
	Healthy	1591	4500
	Late Blight	1909	
	-	-	6652

Table 1.1. Summary of Potato and Tomato Data in Plant Village Dataset.

Here we observe a clear case of an imbalanced dataset. The potato healthy class has 152 photos, but the other classes have 1,000 images or more. This data imbalance problem is taken care of by data augmentation.

1.1.1 Dataset splits and normalization

To divide the 6,652 photos per class into training, validation, and test sets in the ratio 60:20:20, a custom function is utilized.. The process of normalizing the photos involves dividing each pixel by 255, resulting in a range of pixel values between 0 and 1. Neural networks function better with normalized data, which is why this normalization is done.

Dataset	Number of Images
Training	3990
Validation	1331
Test	1331
Total	6652

Table 1.2 Dataset splits.

1.1.2 Data augmentation

The amount of dataset samples is increased by applying data augmentation techniques to the training photographs..The techniques are center crop, gaussian blur and random rotation.In the center crop augmentation, a photo is cropped at the center, based on the size passed into the center crop function. A random selection of sizes is made from [180, 200, 220, 240]. A size of 180 indicates that an original image measuring 256x256 would be centered and resized, resulting in a new image size of 180x180.In the gaussian blur augmentation, the kernel size and sigma values are used for this gaussian blur augmentation. The sigma value is the standard deviation used for creating the kernel to perform the blurring. The range of values from which the sigma values were selected was [1, 1.5, 2, 30 2.5].The angles for the random rotation were selected randomly from 50°, 70°, 100°, 130° and 150°. Because of the changes in image sizes, during the augmentation step with center cropping, images were scaled back to their original dimensions. To be more precise, only the photos in the training set that were augmented underwent resizing.

Dataset	Class	Number of Images	Total Number of Images
Training	Early blight Healthy Late blight Early blight	3996 4011 4002 442	12009
Validation	Healthy Late blight	443 446	1331
Test	-	-	1331

Table 1.3 Number of images per class after data augmentation.

Sample Images

Fig. 1.1. Sample images of potato early blight, healthy and potato late blight.

Fig. 1.2. Sample images of tomato early blight, healthy and late blight.

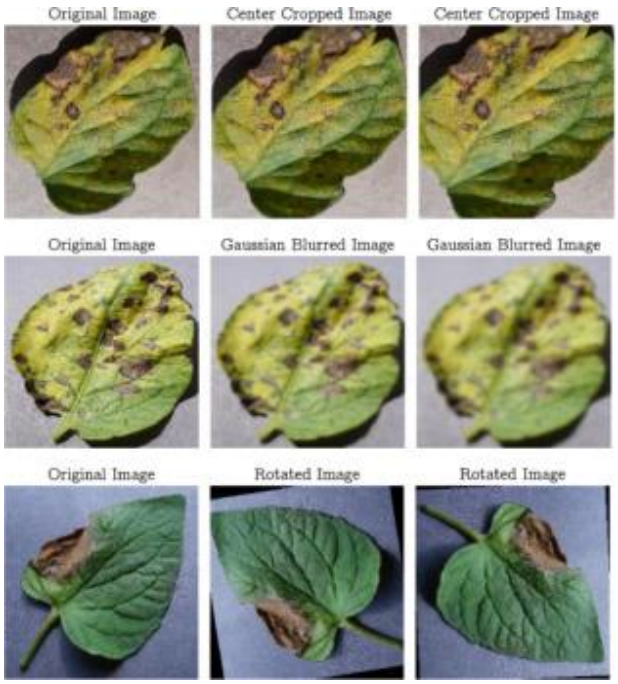
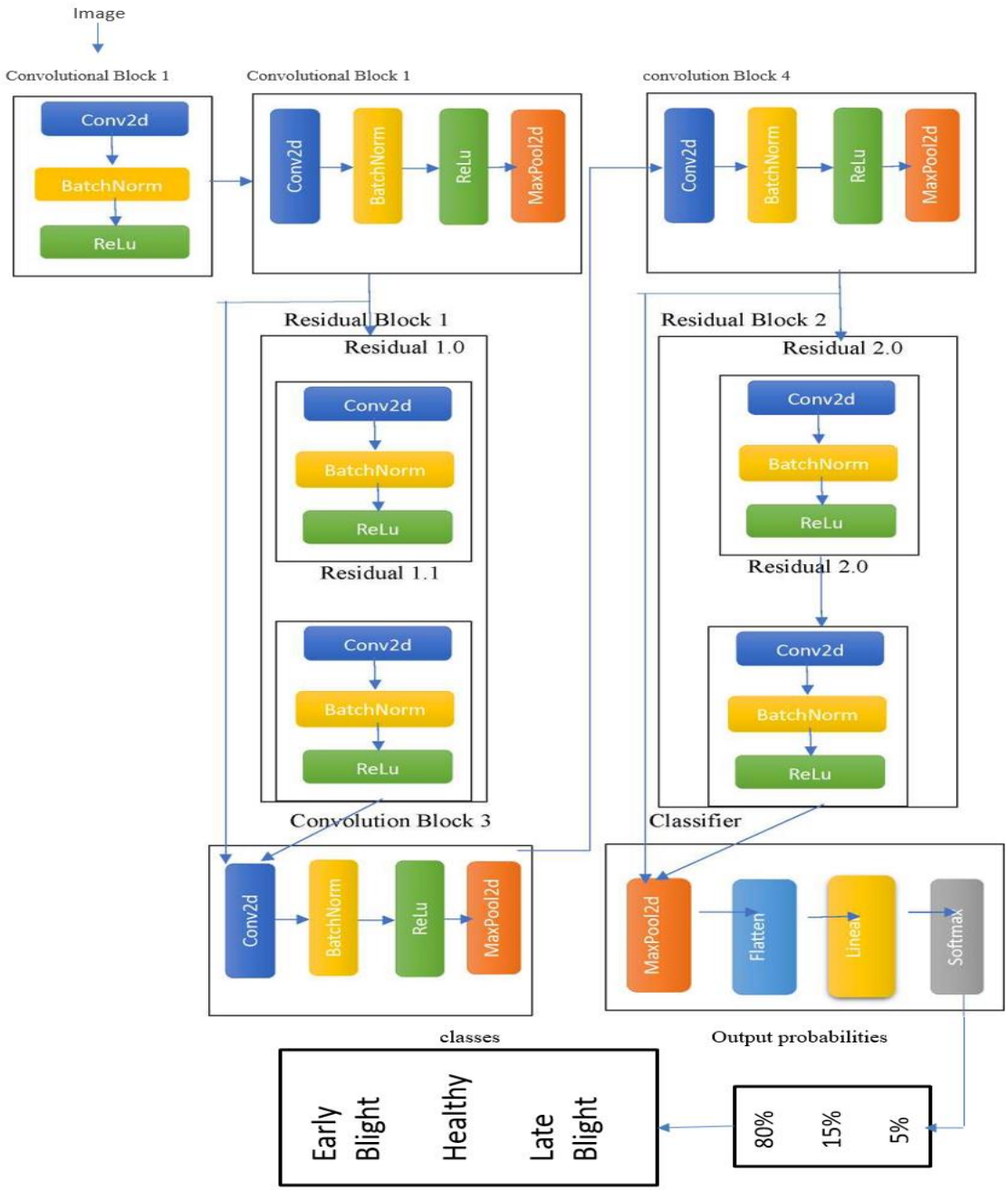


Fig. 1.3. Augmented images.

1.2 POTATO AND TOMATO LEAF CLASSIFICATION WITH RESNET-9 MODEL:

1.2.1 ResNet-9 network architecture and implementation

There are four convolutional blocks, two residual blocks (made of two residual layers each) and one Classifier totaling nine layers, hence the name ResNet-9. The Convolutional Block 1 is the input layer, which obtains input data (plant leaf image) and sends it to the hidden layer for further computations. The hidden layer comprises Convolutional Layer 2 to the Linear Layer in the Classifier. The output layer for this ResNet-9 model is a Softmax layer, which returns class probabilities with the model prediction being the highest probability class.



`

Fig. 1.4. Resnet-9 Architecture.

1.2.2 THE NETWORK'S FUNCTIONS AND OPERATIONS

Input images have shapes of [3, 256, 256] where 3 is the number of channels - red, green and blue, seen in all coloured images. The image's width is represented by the first 256 number, while its height is represented by the second 256 value. As the data moves from layer to layer in the ResNet-9 model, the shapes of the outputs change and are illustrated in table 1.1. An input first passes through the Convolutional Block 1 where a 2d convolutional layer operation is applied on it to extract features such as colour, texture, edges and shape from the image, then normalized by the batch normalization layer which normalizes the inputs from the previous layer before it enters the next layer. By enabling gradients to exhibit a consistent behavior, batch normalization speeds up network training.

Layer	Block Type	Operation	Output Shape
Input	Convolutional Block 1	Conv2d	[32, 64, 256, 256]
		BatchNorm	[32, 64, 256, 256]
		ReLU	[32, 64, 256, 256]
Hidden	Convolutional Block 2	Conv2d	[32, 128, 256, 256]
		BatchNorm	[32, 128, 256, 256]
		ReLU	[32, 128, 256, 256]
		MaxPool2d	[32, 128, 64, 64]
	Residual Block 1.0	Conv2d	[32, 128, 64, 64]
		BatchNorm	[32, 128, 64, 64]
		ReLU	[32, 128, 64, 64]
	Residual Block 1.1	Conv2d	[32, 128, 64, 64]
		BatchNorm	[32, 128, 64, 64]
		ReLU	[32, 128, 64, 64]
	Convolutional Block 3	Conv2d	[32, 256, 64, 64]
		BatchNorm	[32, 256, 64, 64]
		ReLU	[32, 256, 64, 64]
		MaxPool2d	[32, 256, 16, 16]
	Convolutional Block 4	Conv2d	[32, 512, 16, 16]
		BatchNorm	[32, 512, 16, 16]
		ReLU	[32, 512, 16, 16]
		MaxPool2d	[32, 512, 4, 4]
	Residual Block 2.0	Conv2d	[32, 512, 4, 4]
		BatchNorm	[32, 512, 4, 4]
		ReLU	[32, 512, 4, 4]
	Residual Block 2.1	Conv2d	[32, 512, 4, 4]
		BatchNorm	[32, 512, 4, 4]
		ReLU	[32, 512, 4, 4]
	Classifier	MaxPool2d	[32, 512, 1, 1]
		Flatten	[32, 512]
		Linear	[32, 3]
Output	Classifier	Softmax	[32, 3]

Table 1.4 Output Shapes of all Layers in Resnet-9 model.

The ReLU activation function, which is a non-linear activation function, introduces non-linearity for the data to be linearly separable, for classification.In the hidden layer, the input continuously goes through the computations of 2d convolutional layer, Batch normalization and ReLU are combined with a MaxPool2d operation with a kernel size of 4. This lowers spatial measurements (height and width) and selects the most significant picture attributes. From the current layer to the subsequent layer, as seen in Table 1.1, as the input picture 4 passes through the network's blocks, its height, width decrease.In the final residual block of the hidden layer (i.e., Residual Block 2.1), the number of channels value grows to 512, the height value reduces to 4 and width value also reduces to 4. The final 2d max pooling operation of the hidden layer decreases the height, width dimensions from 4 to 1 each. In order to apply the softmax function to input tensors, the flatten layer reshapes them to a single dimension.The linear function, which is the fully linked layer, then sends the data forward into the softmax layer. The softmax layer found in the output layer scales the input tensors in the range [0,1], makes the sum of all tensors equal to one, and returns the probabilities of the input belonging to three classes. The highest probability class would be the predicted class for the image

$$\sigma(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \quad (1.1)$$

Softmax activation function

The loss function of the ResNet-9 model was the Cross Entropy (CE) loss whose equation is shown in Eq (1.2). The network's backpropagation and training phases resulted in an optimization (reduction) of this loss function.

$$\mathcal{L} = -\sum_{j=1}^{n} t_j \log(p_j) \quad (1.2)$$

CE loss equation

- n represents total number of classes,

- t_j is the truth label, and

- p_j serves as the j-th class's softmax output.

1.2.3. Hyperparameter Optimization

During the training process of the ResNet-9 model, several hyperparameters were taken into consideration, including the optimizer, gradient clipping, weight decay, learning rate, momentum, and number of times. With the help of Optuna , a python library, the Tree4 structured Parzen Estimator (TPE) algorithm greatly assisted in the process of determining the optimal set of hyperparameter values,a bayesian optimization method, and this sampler was integrated with a pruning mechanism. The pruning mechanism saved time during the optimization process by terminating trials with unpromising results An objective function

with 100 trials was constructed and executed for the hyperparameter optimization. The objective function was run with the intention of minimizing the CE loss function. To adjust the learning rate dependent on the quantity, we use a one cycle learning rate scheduler,many epochs to ascertain the ideal learning rate figure. This scheduler ensures that the learning rate values drop over the course of the remaining epochs in order to obtain the lowest CE loss value.. We define search spaces for all other hyperparameters, as shown in table 1.2. The optimal set of hyperparameter settings that produced the lowest possible CE loss was determined by the optimization process, and we display these values in table 1.3.

Hyperparameter	**Values Tested**
Optimizer	Adam, SGD
Initial Learning Rate	[0.01, 0.06] (log-uniformly sampled)
Weight Decay	[0.001, 0.1] (log-uniformly sampled)
Momentum	[0.0, 1.0] (uniformly sampled)
Gradient Clipping	[0.1, 0.5] (uniformly sampled)
Number of epochs	[8, 25] (uniformly sampled)

Table 1.5 Hyperparameters explored during training.

Number of Epochs	**Optimizer**	**Initial Learning Rate**	**Weight Decay**	**Momentum**	**Gradient Clipping**
22	SGD	0.011	0.00018	0.42	0.13

Table 1.6 Hyperparameters chosen after optimization.

1.3 POTATO AND TOMATO LEAF CLASSIFICATION WITH VGG-16 MODEL:

1.3.1 VGG-16 network and architecture and implementation

Although VGG16 has twenty-one layers total—13 convolutional layers, 5 Max Pooling layers, 3 Dense levels—it only contains sixteen weight layers, or learnable parameters layers. The max pool and convolution layers are arranged in the same sequence throughout the whole architecture. There are 64 filters in Conv-1 Layer, 128 filters in Conv-2, 256 filters in Conv-3, 512 filters in Conv-4, and 512 filters in Conv-5. Soft-max layer is the last layer.

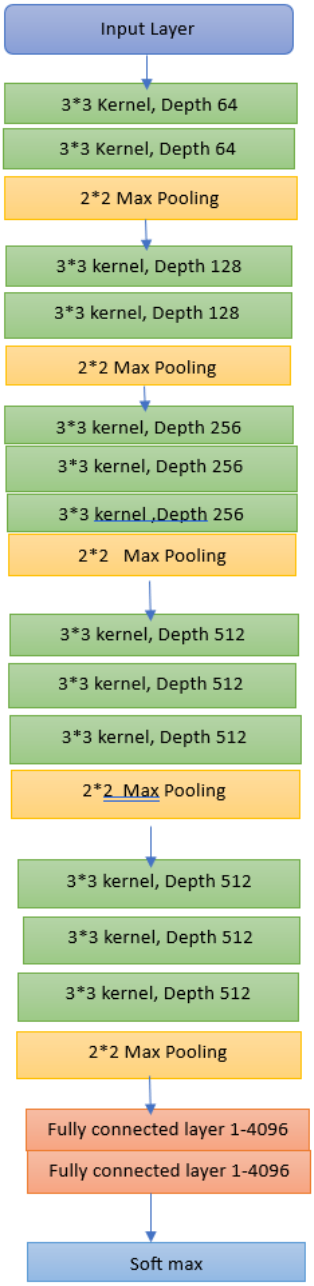


Fig. 1.5. VGG-16 Architecture.

1.3.2 OPERATIONS AND FUNCTIONS OF THE NETWORK

In all network configurations, input is viewed as a fixed 224 × 224 image with three channels (R, G, and B).The width of the picture is shown by the first 256 number, and its height is indicated by the second 256 value. With a receptive size of just 3 × 3, the image goes through the first stack of two convolution layers. After then, ReLU activations occur. Each of these two layers contains 64 filters. Both the padding and the fixed convolution stride are one pixel in size. The spatial resolution is preserved and the size of the input image and the output activation map are the same. The input image's dimensions correspond to the output activation map's size. Next, using spatial max pooling, the

activation maps are passed through a 2 x 2 pixel window with a stride of 2 pixels. The activations pass via a second stack that is identical to the first except that it has 128 filters rather than 64.

Layer	Operation	Output Shape
Input		[224,224,3]
Convolutional Block 1	Conv2d-1	[224,224,64]
	Conv2d-2	[224,224,64]
	MaxPool2d	[112,112,64]
Convolutional Block 2	Conv2d-3	[112,112,128]
	Conv2d-4	[112,112,128]
	Maxpool2d	[56,56,128]
Convolutional Block 3	Conv2d-5	[56,56,256]
	Conv2d-6	[56,56,256]
	Conv2d-7	[56,56,256]
	MaxPool2d	[28,28,256]
Convolutional Block 4	Conv2d-8	[28,28,512]
	Conv2d-9	[28,28,512]
	Conv2d-10	[28,28,512]
	MaxPool2d	[14,14,512]
Convolutional Block 5	Conv2d-11	[14,14,512]
	Conv2d-12	[14,14,512]
	Conv2d-13	[14,14,512]
	MaxPool2d	[7,7,512]
Fully Connected 1	ReLu	[1,4096]
Fully Connected 2	ReLu	[1,4096]
	SoftMax	[1,1000]
Output		

Table 1.7 Output Shapes of all Layers in VGG-16 model.

The size that comes after the second stack is therefore 56 x 56 x 128. The next stack consists of three convolutional layers and a max pool layer. Because there were 256 filters used in this case, the stack's output size was 28 x 28 x 256. This is followed by the placement of two stacks of three convolutional layers, each with 512 filters. The stacks of convolutional layers are followed by three completely linked layers 11 and a flattening layer. There are

1,000 neurons in the last fully 1 10 connected layer, which serves as the output layer. This number of neurons corresponds to the 1,000 potential classes in the ImageNet dataset. There are 4,096 neurons in each of the preceding two levels. The output layer's softmax layer scales the input tensors in the range [0,1], outputs the probability that the input picture belongs to each of the three classes, and sets the total tensors to equal one.The class with the highest probability would be the one predicted for the image.

$$\sigma(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \quad (1.3)$$

Softmax activation function

The loss function of VGG-16 model was the Cross Entropy (CE) loss whose equation is shown in Eq (1.3). During the network's training and backpropagation phases, the loss curve was optimized (decreased).

$$\mathcal{L} = -\sum_{j=1}^{n} t_j \log(p_j) \quad (1.4)$$

CE loss equation

Where n is the total number of classes,

t_j is the truth label, and

p_j is the output of the softmax for the j-th class

1.3.3 HYPERPARAMETER OPTIMIZATION

Number of epochs	Batch size	Optimizer	Learning Rate	Weight Decay	Momentum
30	32	SGD	0.0005	0.0005	0.9

Table 1.8 Hyperparameters of VGG-16 after optimization.

1.4 EVALUATION OF MODELS

Accuracy, precision, recall and f1-score metrics are used on the test set, to evaluate both ResNet-9 and VGG-16 models and also use loss curves to understand training process for both models.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \quad (1.5)$$

$$\text{Precision} = \frac{TP}{TP+FP} \quad (1.6)$$

$$\text{Recall} = \frac{TP}{TP+FN} \quad (1.7)$$

$$\text{F1 – Score} = 2\,\frac{Precision\,.\,Recall}{Precision+Recall} \quad (1.8)$$

Evaluation metrics

where, considering one class at the time,

- True Positive (TP) is when the model correctly predicted to be the considered class.
- True Negative (TN) is when the model correctly predicted to not be the considered class.
- False Positive (FP) is when the model incorrectly predicted to be the considered class.
- False Negative (FN) is when the model incorrectly predicted to not be the considered class. For Precision, Recall and F1-score, the final scores are computed by averaging over the classes.

CHAPTER 2

MERITS AND DEMERITS OF THE BASE PAPER

LITERATURE SURVEY:

For the purpose of identifying tomato and potato leaf blight disease, various algorithms exist. A few are given below, showing the advantages and disadvantages of the suggested approach compared to each of the current approaches.

- **"Potato plant leaves disease detection and classification using machine learning methodologies"** by Singh & Kaur. The multi-class Support Vector Machine (SVM) and the k-means clustering technique serve as the foundation for this paper.Feature extraction was done using gray level co-occurrence matrix. But this method's drawback is it takes more time and computationally expensive.

- **"Potato leaf diseases detection using deep learning"** by Tiwari et al. This paper focuses the pretrained model VGG-19 which performs feature extraction. The selected features then passed into four different classification models which are SVM, Artificial Neural Networks, KNN and LR. The disadvantage of this paper is the lack of Hyperparameter optimization before training and evaluation.

- **"Tomato disease detection and classification by deep learning"** by Hong et al. Additionally, this work used five different deep learning architectures: DenseNet, Xception, ResNet7 7 50, MobileNet, and SuffleNet. This work's heavy reliance on the existence of a tiny collection of comparable color photographs may not always be feasible or useful in real-world situations, which is one potential drawback.

- **"Artificial intelligence in potato leaf disease classification: a deep learning approach"** by Khalifa et al. Two convolutional layers are part of the 14-layer deep learning architecture used in this work, which is used for feature extraction and picture classification. The absence of a pretrained architecture, which reduces calculation times, is a disadvantage, though.

- **"Image Based Tomato Leaf Disease Detection"** by Kumar and Vani. This study builds on thepretrained models such as LeNet, VGG-16, Xception and ResNet50 But the difference between this and the proposed paper is that sometimes incorrect recognition leads to results being brownish. It produces visually pleasing pictures but deviates from the ground truth.

MERITS AND DEMERITS

Merits:

- The paper proposes a strong deep learning model that determines the blight disease state of potato and tomato. This highly reduces the convergence time of the model and improves its accuracy.
- Furthermore, the dataset attributed to rigorous data augmentation, normalization of images and tuning of different hyperparameters before model training and evaluation which enhances the model's perfomance.
- More precise classification results reflecting the actual state of the leaves are obtained using this method.
- The proposed method is faster than the state-of-art methods, and the computation time increases only linearly. Hence applications requiring near real time can benefit from the suggested approach.

Demerits:

- As the methodology was limited to building a strong deep learning model using publicly available images of tomato and potato leaves, different variables like temperature of farms, soil characteristics, leaf area characteristics, and other factors taken into account by this approach.
- This method requires a wide variety of images on which the model should be trained, which is not possible

CHAPTER 3
SOURCE CODE

3.1. DATA PREPARATION FOR TRAINING

```
# datasets for validation and training
train = ImageFolder(train_dir, transform=transforms.Compose(
                                [transforms.Resize([256, 256]),
                                 transforms.ToTensor()]))

valid = ImageFolder(valid_dir, transform=transforms.Compose(
                                [transforms.Resize([256, 256]),
                                 transforms.ToTensor()]))
```

3.2. FUNCTIONS USED TO BUILD THE MODEL

```
# for moving data into GPU (if available)
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available:
        return torch.device("cuda")
    else:
        return torch.device("cpu")

# for moving data to device (CPU or GPU)
def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

# for loading in the device (GPU if available else CPU)
class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)
```

```
    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

3.3 CALCULATING ACCURACY

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationBase(nn.Module):

    def training_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        train_acc = accuracy(out, labels)
        return loss

    def training_step_4_acc(self, batch):
        images, labels = batch
        out = self(images)
        train_acc = accuracy(out, labels)
        return train_acc


    def validation_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {"val_loss": loss.detach(), "val_accuracy": acc}


    def validation_epoch_end(self, outputs):
        batch_losses = [x["val_loss"] for x in outputs]
        batch_accuracy = [x["val_accuracy"] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        epoch_accuracy = torch.stack(batch_accuracy).mean()
        return {"val_loss": epoch_loss, "val_accuracy": epoch_accuracy}

    def epoch_end(self, epoch, result):
```

```
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, train_accuracy: {:.4f}, val_loss:
{:.4f}, val_acc: {:.4f}".format(epoch, result['lrs'][-1], result['train_loss'],
result['train_accuracy'], result['val_loss'], result['val_accuracy']))
```

3.4 RESNET-9 MODEL IMPLEMENTATION

```
class SimpleResidualBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1,
padding=1)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1,
padding=1)
        self.relu2 = nn.ReLU()

    def forward(self, x):
        out = self.conv1(x)
        out = self.relu1(out)
        out = self.conv2(out)
        return self.relu2(out) + x


def ConvBlock(in_channels, out_channels, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True)]
    if pool:
        layers.append(nn.MaxPool2d(4))
    return nn.Sequential(*layers)


# resnet architecture
class ResNet9(ImageClassificationBase):
    def __init__(self, in_channels, num_diseases):
        super().__init__()

        self.conv1 = ConvBlock(in_channels, 64)
        self.conv2 = ConvBlock(64, 128, pool=True) # out_dim : 128 x 64 x 64
        self.res1 = nn.Sequential(ConvBlock(128, 128), ConvBlock(128, 128))

        self.conv3 = ConvBlock(128, 256, pool=True) # out_dim : 256 x 16 x 16
        self.conv4 = ConvBlock(256, 512, pool=True) # out_dim : 512 x 4 x 44
```

```
        self.res2 = nn.Sequential(ConvBlock(512, 512), ConvBlock(512, 512))

        self.classifier = nn.Sequential(nn.MaxPool2d(4),
                                        nn.Flatten(),
                                        nn.Linear(512, num_diseases),
                                        nn.Softmax(dim=1))

    def forward(self, xb): # xb is the loaded batch
        out = self.conv1(xb)
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.res2(out) + out
        out = self.classifier(out)
        return out
```

3.5 VGG-16 MODEL IMPLEMENTATION

```
# convolution block with BatchNormalization
def ConvBlock(in_channels, out_channels, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True)]
    if pool:
        layers.append(nn.MaxPool2d(4))
    return nn.Sequential(*layers)


class VGG16(ImageClassificationBase):
    def __init__(self, num_classes):
        super(VGG16, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU())
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer3 = nn.Sequential(
```

```
        nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU())
    self.layer4 = nn.Sequential(
        nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = 2, stride = 2))
    self.layer5 = nn.Sequential(
        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU())
    self.layer6 = nn.Sequential(
        nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU())
    self.layer7 = nn.Sequential(
        nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = 2, stride = 2))
    self.layer8 = nn.Sequential(
        nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU())
    self.layer9 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU())
    self.layer10 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = 2, stride = 2))
    self.layer11 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU())
    self.layer12 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU())
    self.layer13 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(25088, 4096),  #or 25088, #(32x32768 and 25088x4096)
            nn.ReLU())
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU())
        self.fc2= nn.Sequential(
            nn.Linear(4096, num_classes))

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = self.layer6(out)
        out = self.layer7(out)
        out = self.layer8(out)
        out = self.layer9(out)
        out = self.layer10(out)
        out = self.layer11(out)
        out = self.layer12(out)
        out = self.layer13(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

3.6 FUNCTION FOR TRAINING MODEL

```
@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)
```

```
def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']


def fit_OneCycle(epochs, max_lr, model, train_loader, val_loader, momentum=0,
weight_decay=0,
                 grad_clip=None, opt_func=torch.optim.SGD):
    torch.cuda.empty_cache()
    history = []

    optimizer = opt_func(model.parameters(), max_lr, weight_decay, momentum)


    sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs,
steps_per_epoch=len(train_loader))


    for epoch in range(epochs):
        # Training
        model.train()
        train_losses = []
        train_accuracies = []
        lrs = []
        for batch in train_loader:
            loss = model.training_step(batch)
            t_acc = model.training_step_4_acc(batch)
            train_losses.append(loss)
            train_accuracies.append(t_acc)
            loss.backward()


            # gradient clipping
            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

            # recording and updating learning rates
            lrs.append(get_lr(optimizer))
            sched.step()
```

```
        # validation
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['train_accuracy'] = torch.stack(train_accuracies).mean().item()
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)

    return history
```

3.7 HYPERPARAMETER TUNING WITH OPTUNA FOR ResNet-9 MODEL

```
def train_and_evaluate(param, model, train_loader, val_loader, trial):
    torch.cuda.empty_cache()
    history = []

    #optimizer = opt_func(model.parameters(), max_lr, weight_decay=weight_decay)
    #opt_func = getattr(torch.optim, param['optimizer'])
    optimizer = getattr(torch.optim, param['optimizer'])(model.parameters(),
lr=param['initial_lr'], weight_decay=param['weight_decay'],
momentum=param['momentum'])


    sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, param['initial_lr'],
epochs=param['epochs'], steps_per_epoch=len(train_loader))

    for epoch in range(param['epochs']):
        # Training
        model.train()
        train_losses = []
        train_accuracies = []
        lrs = []
        for batch in train_loader:
            loss = model.training_step(batch)
            t_acc = model.training_step_4_acc(batch)
            train_losses.append(loss)
            train_accuracies.append(t_acc)
            loss.backward()

            # gradient clipping
            if param['grad_clip']:
                nn.utils.clip_grad_value_(model.parameters(), param['grad_clip'])

            optimizer.step()
```

```
            optimizer.zero_grad()

            # recording and updating learning rates
            lrs.append(get_lr(optimizer))
            sched.step()

        # validation
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['train_accuracy'] = torch.stack(train_accuracies).mean().item()
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)
        val_acc_last_epoch = history[-1]['val_accuracy'].item()

        trial.report(val_acc_last_epoch, epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()


        return val_acc_last_epoch



def objective(trial):
    params = {'initial_lr': trial.suggest_loguniform('initial_lr', 0.01, 0.06),
              'optimizer': trial.suggest_categorical("optimizer", ["SGD"]),
              'weight_decay': trial.suggest_loguniform('weight_decay', 1e-4, 1e-1),
              'grad_clip': trial.suggest_float('grad_clip', 0.1, 0.5),
              'epochs' : trial.suggest_int('epochs', 8, 25),
              'momentum' : trial.suggest_float('momentum', 0, 1),
              }

    val_accuracy = train_and_evaluate(params, model, train_dl, valid_dl, trial)
    return val_accuracy


optuna.visualization.matplotlib.plot_optimization_history(study)
plt.rcParams['figure.figsize']=[6,6]
plt.rcParams['figure.facecolor'] = 'white'
plt.tight_layout()
plt.show()
In [ ]:
optuna.visualization.plot_param_importances(study)
```

3.8 ResNet-9 MODEL TRAINING WITH OPTIMIZED HYPERPARAMETERS

```
%%time
epochs = 22
momentum = 0.42
grad_clip = 0.13
initial_lr = 0.011
weight_decay = 0.00018
opt_func = torch.optim.SGD

history = fit_OneCycle(epochs, initial_lr, model, train_dl, valid_dl,
                       momentum=0.42,
                       grad_clip=0.13,
                       weight_decay=0.00018,
                       opt_func=opt_func)
```

3.9 VGG-16 MODEL TRAINING WITH OPTIMIZED HYPERPARAMETERS

```
%%time
epochs = 30
max_lr = 0.0005
grad_clip = 0.1
weight_decay = 0.0005
opt_func = torch.optim.SGD

history += fit_OneCycle(epochs, max_lr, model, train_dl, valid_dl,
                        grad_clip=0.1, momentum=0.9,
                        weight_decay=0.0005,
                        opt_func=opt_func)
```

3.10 PLOTTING TRAINING ACCURACIES AND LOSSES FOR ResNet-9

```
def plot_accuracies(history, epochs):
    """This function plots both training and validation accuracies of ResNet-9 model"""
    plt.rcParams['figure.figsize'] = [8, 8]
    epochs = [i for i in range(1, epochs+1)]
    val_accuracies = [x['val_accuracy'] for x in history]
    train_accuracies = [x['train_accuracy'] for x in history]
    plt.plot(epochs, train_accuracies, '-o', color='blue', label='train_loss')
```

```
    plt.plot(epochs, val_accuracies, '-o', color='green', label='validation_loss')
    plt.xticks(np.arange(min(epochs), max(epochs)+1, 1.0))
    plt.xlabel('Epochs', size=13)
    plt.ylabel('Accuracies', size=13)
    plt.grid(color='#EAE4E3')
    #plt.xticks(rotation=90)
    plt.title('Training and validation accuracies of ResNet-9', size=13)
    plt.legend()
    plt.savefig('../working/resnet9-tv-accuracies.png', dpi=600,  bbox_inches="tight")
    plt.show()

def plot_losses(history, epochs):
    plt.rcParams['figure.figsize'] = [8, 8]
    epochs = [i for i in range(1, epochs+1)]
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x.get('val_loss').cpu().numpy() for x in history]
    plt.plot(epochs, train_losses, '-o', color='blue', label='train_loss')
    plt.plot(epochs, val_losses, '-o', color='green', label='validation_loss')
    plt.xticks(np.arange(min(epochs), max(epochs)+1, 1.0))
    plt.xlabel('Epochs', size=13)
    plt.ylabel('Losses', size=13)
    plt.grid(color='#EAE4E3')
    #plt.xticks(rotation=90)
    plt.title('Training and validation losses of ResNet-9', size=13)
    plt.legend()
    plt.savefig('../working/resnet9-tv-losses.png', dpi=600,  bbox_inches="tight")
    plt.show()
```

3.11 PLOTTING TRAINING ACCURACIES AND LOSSES FOR VGG-16

```
def plot_accuracies(history, epochs):
    val_accuracies = [x['val_accuracy'] for x in history[1:]]
    train_accuracies = [x['train_accuracy'] for x in history[1:]]
    plt.grid(color='#EAE4E3')
    plt.plot(train_accuracies, '-o', color='blue', label='train_accuracy')
    plt.plot(val_accuracies, '-o', color='green', label = 'validation_accuracy')
    plt.xticks(np.arange(0, epochs+1, 1))
    plt.xticks(rotation=90)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracies')
    plt.legend()
    plt.title('Training and Validation accuracies of VGG-16')
    plt.savefig('../working/vgg16-tv-accuracies.png', dpi=600,  bbox_inches="tight")
```

```
def plot_losses(history, epochs):
    train_losses = [x.get('train_loss') for x in history[1:]]
    val_losses = [x.get('val_loss').cpu().numpy() for x in history[1:]] #[x['val_loss'] for x
in history]
    plt.grid(color='#EAE4E3')
    plt.plot(train_losses, '-bo', label='train_loss')
    plt.plot(val_losses, '-go', label='validation_loss')
    plt.xticks(np.arange(0, epochs+1, 1))
    plt.xticks(rotation=90)
    plt.xlabel('Epochs')
    plt.ylabel('Losses')
    plt.legend()#['Training', 'Validation'])
    plt.title('Training and Validation losses of VGG-16')
    plt.savefig('../working/vgg16-tv-losses.png', dpi=600,  bbox_inches="tight")


def plot_lrs(history):
    lrs = np.concatenate([x.get('lrs', []) for x in history])
    plt.grid(color='#EAE4E3')
    plt.plot(lrs)
    plt.xlabel('Batch no.')
    plt.ylabel('Learning rate')
    plt.title('Learning Rate vs. Batch no.');
```

3.12 PREDICTING IMAGES

```
predictions =[]
targets = []
for images, labels in test_loader_r:
    images, labels = images.cuda(), labels.cuda()
    logps = model(images)
    output = torch.exp(logps)
    pred = torch.argmax(output, 1)

    # convert to numpy arrays
    pred = pred.detach().cpu().numpy()
    labels = labels.detach().cpu().numpy()
    for i in range(len(pred)):
        predictions.append(pred[i])
        targets.append(labels[i])
```

CHAPTER 4
OUTPUT SNAPSHOTS

4.1 Results of ResNet-9 Model:

Class	Precision	Recall	F1-Score
Early Blight	100.00%	98.00%	99.00%
Healthy	100.00%	100.00%	100.00%
Late Blight	99.00%	100.00%	99.00%
Overall	99.67%	99.33%	99.33%

Table 4.1 Classification Report for Resnet-9.

4.1.1 Loss curves during training

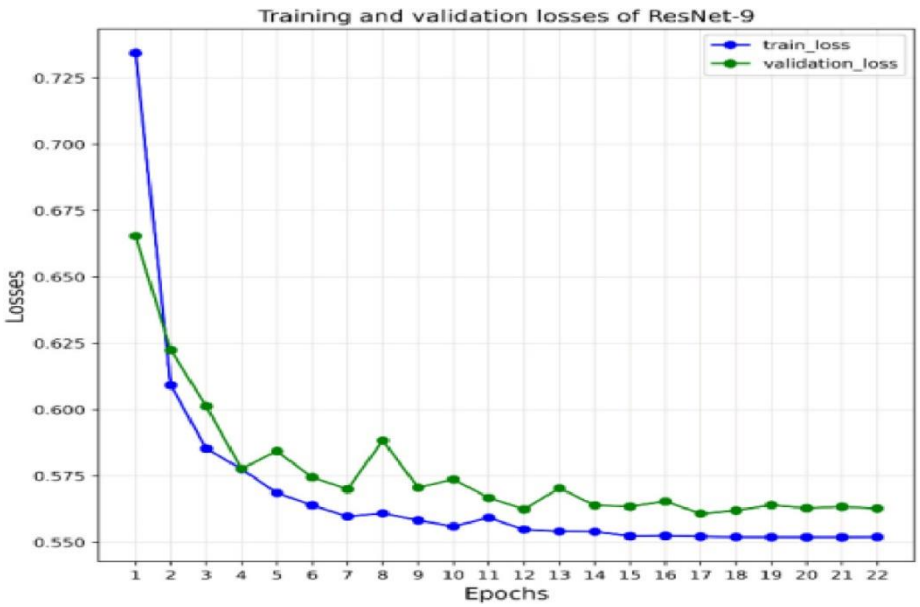


Fig. 4.1. Resnet-9 training and validation losses.

Fig 4.1 shows training and validation losses of the ResNet-9 model during training over 22 epochs. At epoch 1, training loss is 0.736, while validation loss is 0.663. At epoch 22, training loss is 0.550, while validation loss is 0.563.

4.1.2 Confusion matrix

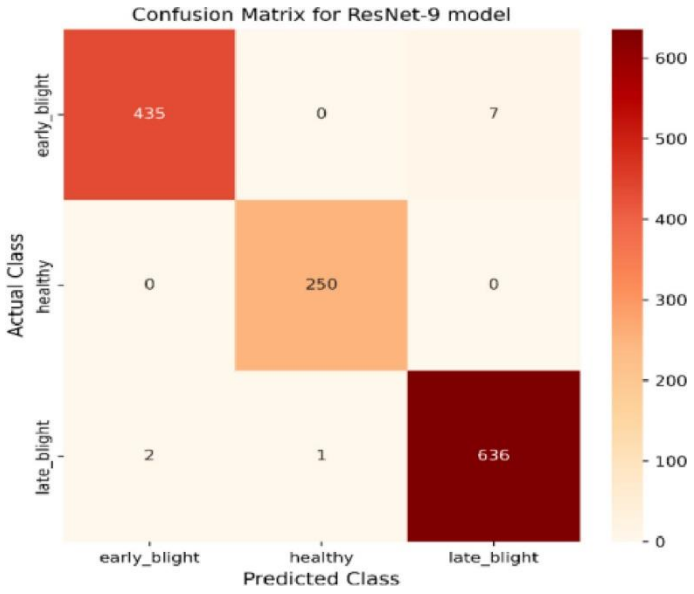


Fig. 4.2. Confusion Matrix for ResNet-9.

4.2 Results of VGG-16 model

Class	Precision	Recall	F1-Score
Early Blight	100.00%	98.00%	99.00%
Healthy	99.00%	99.00%	99.00%
Late Blight	98.00%	99.00%	99.00%
Overall	99.00%	98.67%	99.00%

Table 4.2 Classification Report for VGG-16.

4.2.1 Loss curves during Training

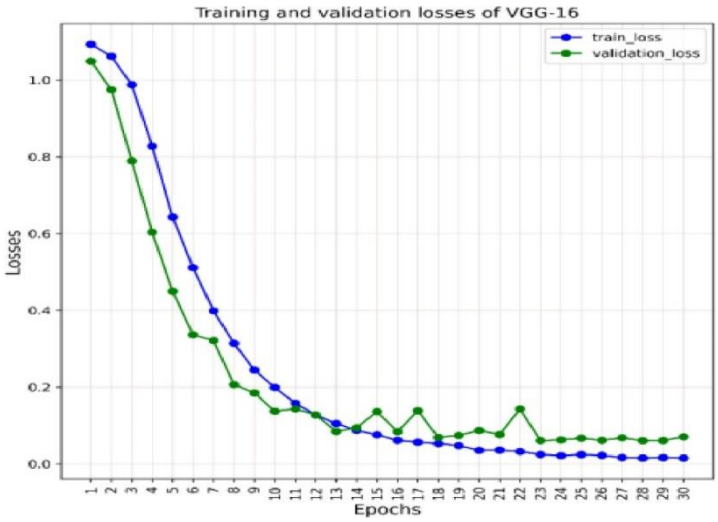


Fig. 4.3. VGG-16 training and validation losses.

Fig 4.3 shows the training and validation losses of the VGG-16 model during training over 30 epochs. At epoch 1, training loss is 1.12, while validation loss is 1.09. At epoch 30, training loss is 0.0, while validation loss is 0.9.

4.2.3 Confusion Matrix:

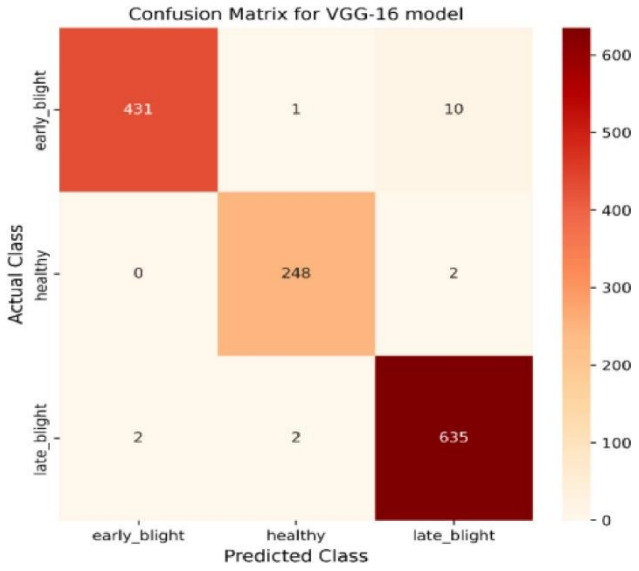


Fig. 4.4. Confusion matrix for VGG-16.

CHAPTER 5
CONCLUSION AND FUTURE PLANS

The findings of this study suggests a novel method for identifying potato and tomato plant blight disease in their leaves. This method is predicated on pretrained models like the VGG-16 and Resnet-9 architecture. This project uses VGG-16 as a baseline model to create Resnet-9 model. Then the model is trained for the training set using the pretrained models. Rigorous data preprocessing and hyperparameter optimization provided the set of best hyperparameters to achieve a good test accuracy (99.25%). When comparing the two models' accuracy, precision, recall, and F1-scores, the ResNet-9 model outperforms the VGG-16 baseline model.

Our model currently performs well on certain categories because it was solely trained on such images. However, to enhance its performance,Training and implementation of the ResNet-9 model can be further explored by adding more agricultural management and environmental data in the future. We could improve this approach by making it consider other plants like corn and pepper as well as potato and tomato diseases like common scab, bacterial wilt, leaf curl and bacteria spot. The deployment of this model with an interface, where farmers can take photos of their affected crops in real-time, would be beneficial. This is a future consideration and expansion upon this research.

CHAPTER 6
REFERENCES

[1] Anim-Ayeko, A.O., Schillaci, C. and Lipani, A., 2023. Automatic blight disease detection in potato (Solanum tuberosum L.) and tomato (Solanum lycopersicum, L. 1753) plants using deep learning. Smart Agricultural Technology, 4, p.100178.

[2] Kumar, A., & Vani, M. (2019, July). Image based tomato leaf disease detection. In 2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT) (pp. 1-6). IEEE.

[3] Singh, A., & Kaur, H. (2021). Potato plant leaves disease detection and classification using machine learning methodologies. In IOP Conference Series: Materials Science and Engineering (Vol. 1022, No. 1, p. 012121). IOP Publishing.

[4] Khalifa, N. E. M., Taha, M. H. N., Abou El-Maged, L. M., & Hassanien, A. E. (2021). Artificial intelligence in potato leaf disease classification: a deep learning approach. Machine learning and big data analytics paradigms: analysis, applications and challenges, 63-79.

[5] Tiwari, D., Ashish, M., Gangwar, N., Sharma, A., Patel, S., & Bhardwaj, S. (2020, May). Potato leaf diseases detection using deep learning. In 2020 4th international conference on intelligent computing and control systems (ICICCS) (pp. 461-466). IEEE.

[6] Hong, H., Lin, J., & Huang, F. (2020, June). Tomato disease detection and classification by deep learning. In 2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE) (pp. 25-29). IEEE.

CHAPTER 7
APPENDIX

BASE PAPER

Anim-Ayeko, A.O., Schillaci, C. and Lipani, A., 2023. Automatic blight disease detection in potato (Solanum tuberosum L.) and tomato (Solanum lycopersicum, L. 1753) plants using deep learning. *Smart Agricultural Technology, 4, p.100178.*

doi: 10.1016/j.atech.2023.100178

keywords: {Deep Learning, Pretrained models, VGG-16, Resnet-9, Blight disease; RGB; Training; Neural networks; Optimization}

URL:
https://www.sciencedirect.com/science/article/pii/S2772375523000084