University of Pisa

Artificial Intelligence and Data Engineering
Computational Intelligence and Deep Learning

Precision in *Plant Disease Diagnosis*: A CNN Approach to Enhance Agricultural Practices

Francesco Londretti
Ricky Marinsalda

Contents

Chapter 1

Introduction

In the realm of agriculture, the ability to efficiently detect and diagnose diseases in plants holds the key to improving agricultural practices and ultimately increasing crop yields.

1.1 Project Objective

Our project is dedicated to addressing this critical need by harnessing the power of Convolutional Neural Networks (CNNs). These neural networks are adept at learning intricate patterns and features from images, equipping them with the capability to distinguish between healthy and infected plants with remarkable precision.

1.1.1 Practical Applications

The potential applications of this project extend beyond traditional farming, as it can also be seamlessly integrated into daily life to care for indoor plants.

1.1.2 Methodology Overview

To accomplish our goal, we will undertake a comprehensive evaluation of two CNN models: one built from scratch and another utilizing pre-trained weights. Through a thorough analysis of various metrics, we aim to determine which model exhibits superior accuracy in plant disease detection, thereby contributing to the advancement of agricultural practices.

1.2 Dataset Description

For our project, we utilized the New Plant Diseases dataset, which is accessible at the following link: `https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset`.

This dataset has been generated by applying offline augmentation techniques to the original dataset, which is accessible on this GitHub repository: `https://github.com/spMohanty/PlantVillage-Dataset`. The dataset comprises approximately 87,000 RGB images depicting both healthy and diseased crop leaves, categorized into 38 distinct classes. The complete dataset is partitioned into training and validation sets in an 80/20 ratio, while maintaining the original directory structure. Additionally, a separate directory containing 5438 test images has been established for prediction purposes.

1.2.1 Dataset Distribution

Classes	Num. of images
Strawberry__healthy	1824
Tomato__healthy	1926
Tomato__Septoria_leaf_spot	1745
Cherry_(including_sour)__healthy	1826
Potato__healthy	1824
Peach__Bacterial_spot	1838
Grape__Black_rot	1888
Tomato__Tomato_mosaic_virus	1790
Tomato__Leaf_Mold	1882
Strawberry__Leaf_scorch	1774
Tomato__Late_blight	1851
Corn_(maize)__healthy	1859
Squash__Powdery_mildew	1736
Tomato__Early_blight	1920
Grape__healthy	1692
Cherry_(including_sour)__Powdery_mildew	1683
Pepper,_bell__healthy	1988
Peach__healthy	1728
Tomato__Tomato_Yellow_Leaf_Curl_Virus	1961
Apple__healthy	2008
Potato__Late_blight	1939
Corn_(maize)__Northern_Leaf_Blight	1908
Pepper,_bell__Bacterial_spot	1913
Grape__Leaf_blight_(Isariopsis_Leaf_Spot)	1722
Raspberry__healthy	1781
Apple__Cedar_apple_rust	1760
Corn_(maize)__Common_rust_	1907
Soybean__healthy	2022
Tomato__Bacterial_spot	1702
Potato__Early_blight	1939
Grape__Esca_(Black_Measles)	1920
Tomato__Target_Spot	1827
Apple__Apple_scab	2016
Apple__Black_rot	1987
Corn_(maize)__Cercospora_leaf_spot_Gray_leaf_spot	1642
Tomato__Spider_mites_Two-spotted_spider_mite	1741
Orange__Haunglongbing_(Citrus_greening)	2010
Blueberry__healthy	1816

Table 1.1: Number of images for each class of the training set

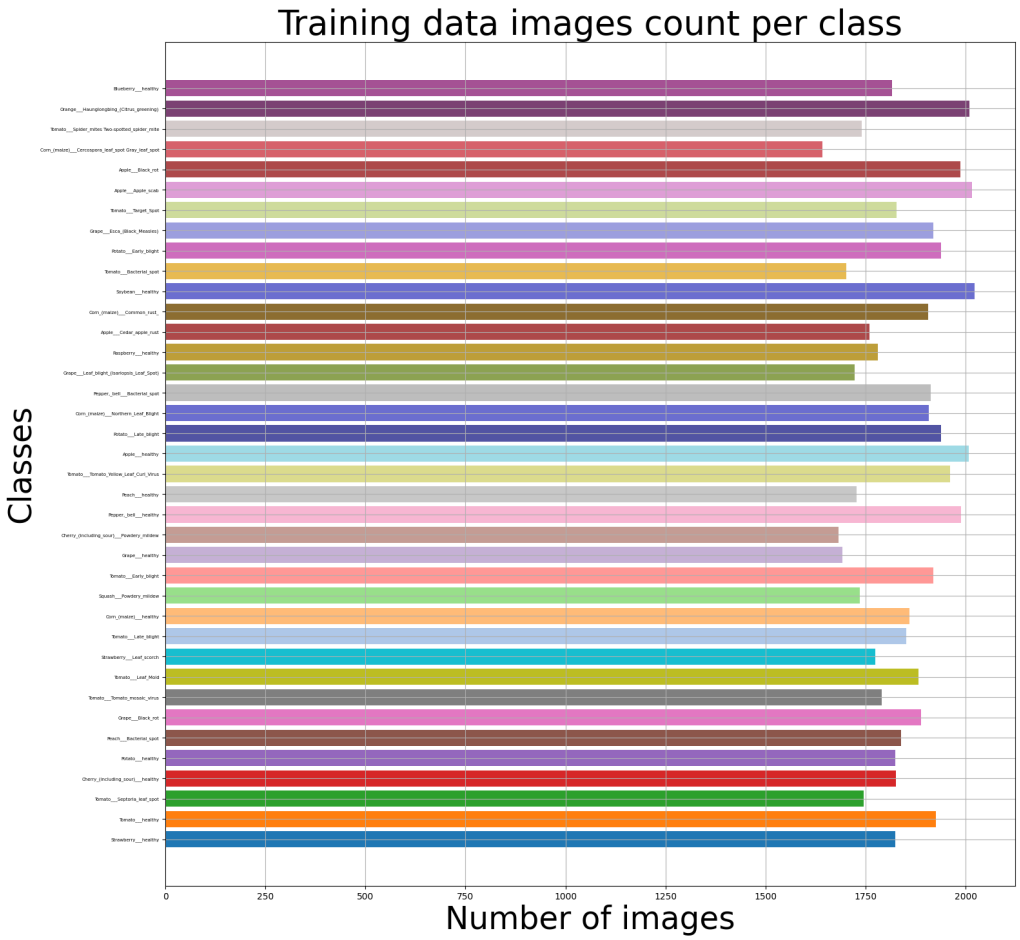

Figure 1.1: Class distribution

As we can see the class distribution of the training dataset is quite balanced.

Chapter 2

Related Works

In recent years, the application of Convolutional Neural Networks (CNNs) has revolutionized the field of image analysis, offering promising solutions to various problems, including the detection and diagnosis of plant diseases. Despite the success of CNNs, the challenges inherent in precise plant disease diagnosis persist and continue to drive research in this domain.

The recognition and classification of plant diseases from images is a complex task due to variations in plant species, environmental conditions, and the diverse manifestations of diseases. Researchers have made significant strides in addressing these challenges, and several noteworthy contributions in this area have advanced our understanding and capabilities.

F. Marzougui, M. Elleuch and M. Kherallah in their paper titled *A Deep CNN Approach for Plant Disease Detection* [2] presented an ensemble solution for plant disease diagnosis. They leveraged variants of classifiers designed with pretrained neural network architectures. Their approach achieved remarkable results, including a weighted F1-score of 97.2% on the test dataset.

Nishant Shelar, Suraj Shinde, Shubham Sawant, Shreyash Dhumal and Kausar Fakir in their publication *Plant Disease Detection Using Cnn* [3] Their paper aims to develop a Disease Recognition Model using leaf image classification and convolutional neural networks (CNNs) for precise plant disease detection. CNNs are specialized for image processing and recognition, making them suitable for this task. Their model demonstrated a weighted F1-score of 95.6% on the final test set.

Shi, T., Liu, Y., Zheng, explored *Recent advances in plant disease severity assessment using convolutional neural networks* [4]. This study reviews 16 CNN-based approaches for severity assessment, discusses dataset acquisition, evaluation metrics, and addresses challenges, offering research ideas and solutions for practical applications. Additionally, they incorporated a basic attention mechanism to enhance their model's performance. Their ensemble of neural networks achieved a weighted F1-score of 98.7% on the final test dataset.

Hassan, S.M.; Maji, A.K.; Jasi 'nski, M.; Leonowicz, Z.; Jasi 'nska introduced *Identification of Plant-Leaf Diseases Using CNN and Transfer-Learning Approach* [1], they optimized the models by using depth-separable convolution, reducing parameters and computation. With training on a diverse dataset, the models achieved high accuracy rates, surpassing traditional methods, and outperformed other deep-learning models, showing promise for efficient real-time disease detection in agriculture.

Our project, *Precision in Plant Disease Diagnosis: A CNN Approach to Enhance Agricultural Practices*, contributes to this body of work by evaluating the effectiveness of CNN models in plant disease detection. We aim to determine which approach, whether building a CNN from scratch or utilizing pre-trained weights, offers superior accuracy in plant disease diagnosis. Our efforts are geared towards advancing agricultural practices and promoting healthier crop yields.

Chapter 3

Methods and Experiments

In this chapter, we discuss the implementation of a Convolutional Neural Network (CNN) model using TensorFlow and Keras for the purpose of image classification. The model has been trained over 20 epochs. This chapter provides a detailed breakdown of the code used for building and training the CNN.

3.1 Neural Network From Scratch

3.1.1 Model Architecture

The model architecture is structured as follows:

Data Augmentation and Rescaling

These are the layers used to apply changes to the images.

Data Augmentation A data augmentation layer is added to enhance the training dataset by applying random transformations to the input images.

```
data_augmentation = keras.Sequential([
layers.RandomFlip("horizontal"), # Applies horizontal flipping to a random 50% of the
  ↪ images
layers.RandomRotation(0.1), # Rotates the input images by a random value in the range
  ↪ [-10\%, +10\%] (fraction of full circle [-36, 36])
layers.RandomZoom(0.1), # Zooms in or out of the image by a random factor in the range
  ↪  [-20%, +20%]
layers.RandomContrast(0.1),
],
name = "AugmentationLayer"
)
```

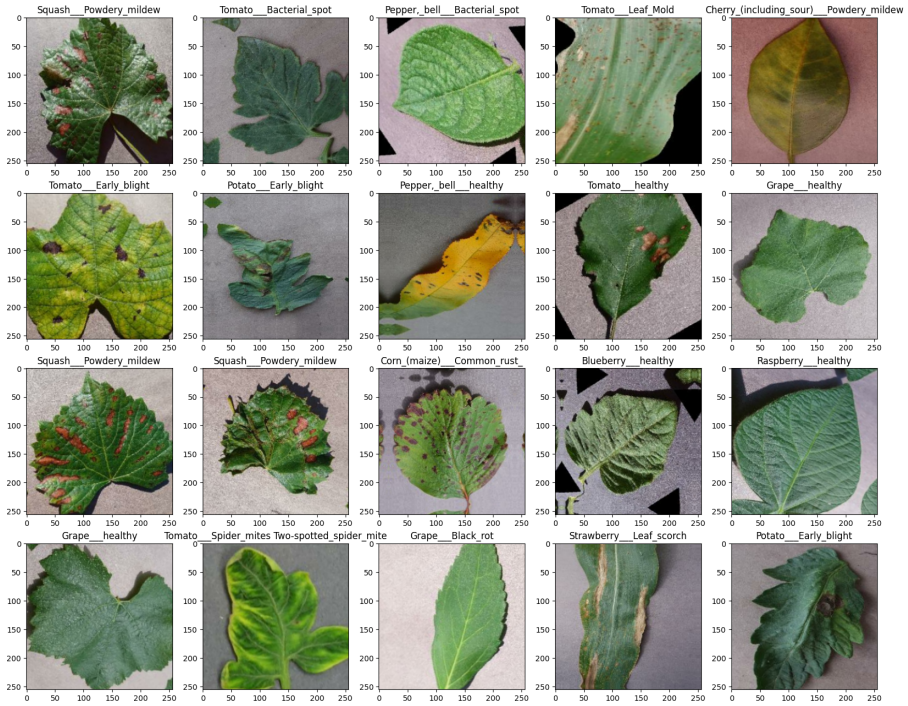

Figure 3.1: Some of the augmented images

Rescaling A rescaling layer is incorporated to normalize the pixel values to the range [0, 1].

```
model.add(layers.Rescaling(1./255))
```

Convolutional Layers

Multiple convolutional layers with varying numbers of filters, kernel sizes, and activation functions (ReLU) are introduced to extract features from the input images. Max-pooling and batch normalization layers are applied subsequently to downsample and normalize the feature maps.

```
model.add(layers.Conv2D(32, kernel_size = 3, activation = "relu6", padding = "same",
    ↪ input_shape = (256, 256,3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())

model.add(layers.Conv2D(64, kernel_size = 3, activation='relu', padding = "same"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())

model.add(layers.Conv2D(128, kernel_size = 3, activation='relu', padding = "same"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())

model.add(layers.Conv2D(256, kernel_size = 3, activation='relu', padding = "same"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())

model.add(layers.Conv2D(512, kernel_size = 3, activation='relu', padding = "same"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())

model.add(layers.Conv2D(512, kernel_size = 3, activation='relu', padding = "same"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
```

Of course, this represents the final and most optimized model in our architecture. The model's composition began with just three building blocks, each consisting of a *Conv2D* layer, a *MaxPooling2D* layer, and a *BatchNormalization* layer. In its initial form, the model included three chunks, each comprising 32, 64, and 128 feature maps, respectively. As our experimentation progressed, we systematically added more chunks to the architecture to enhance the training accuracy.

Top Layer

It is responsible for adapting the network's output to the specific problem.

Flatten Layer A flatten layer is employed to convert the output from the convolutional layers into a 1D vector.

```
model.add(layers.Flatten())
```

Dropout Layer Dropout with a rate of 0.5 is used to regularize the model by randomly deactivating a portion of input units. It's used against overfitting.

```
model.add(layers.Dropout(0.5))
```

Dense Layers One fully connected (dense) layer with *ReLU* activation function is added to facilitate the classification. The last layer is formed by the same number of neuron as the number of classes and it has the *Softmax* activation function.

```
model.add(layers.Dense(256,activation="relu"))
model.add(layers.Dense(38,activation="softmax"))
```

This is the resulting model:

Layer	Output Shape	Param #
AugmentationLayer	(None, 256, 256, 3)	0
rescaling_3	(None, 256, 256, 3)	0
conv2d_6	(None, 256, 256, 32)	896
max_pooling2d_6	(None, 128, 128, 32)	0
batch_normalization_9	(None, 128, 128, 32)	128
conv2d_7	(None, 128, 128, 64)	18496
max_pooling2d_7	(None, 64, 64, 64)	0
batch_normalization_10	(None, 64, 64, 64)	256
conv2d_8	(None, 64, 64, 128)	73856
max_pooling2d_8	(None, 32, 32, 128)	0
batch_normalization_11	(None, 32, 32, 128)	512
conv2d_9	(None, 32, 32, 256)	295168
max_pooling2d_9	(None, 16, 16, 256)	0
batch_normalization_12	(None, 16, 16, 256)	1024
conv2d_10	(None, 16, 16, 512)	1180160
max_pooling2d_10	(None, 8, 8, 512)	0
batch_normalization_13	(None, 8, 8, 512)	2048
conv2d_11	(None, 8, 8, 512)	2359808
max_pooling2d_11	(None, 4, 4, 512)	0
batch_normalization_14	(None, 4, 4, 512)	2048
flatten_4	(None, 8192)	0
dropout_4	(None, 8192)	0
dense_8	(None, 256)	2097408
dense_9	(None, 38)	9766

Table 3.1: From scratch model

3.1.2 Model Training

The model is trained through the following steps:

Compile Settings

In this section we explain the settings chosen for the compilation of the model:

- **loss**: we used *categorical_ crossentropy* because it quantifies the dissimilarity between predicted class probabilities and actual class labels. The formula for the categorical crossentropy is as follows: $-\sum_i y_i \cdot log(p_i)$ where $y_i$is the true probability distribution for class i and p_i is the predicted probability for class i as output by the model.
- **optimizer**: the algorithm used to update the model's weights during training is *Adam*. We chose this optimizer instead of *rmsprop* because of his popularity and adaptive learning rates.
- **metrics**: we chose to use *accuracy* as our metric due to the balanced dataset, its interpretability, and the absence of a need to favor either FP or FN.

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

Callbacks

The defined callbacks are **early stopping** and **model checkpoint**, to monitor the training process.

```
save_best_model = tf.keras.callbacks.ModelCheckpoint(modelPath, verbose=True, monitor=
  ↪ 'val_loss', save_best_only=True, save_weights_only=True)

earlyStopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

ModelCheckpoint callback is saving the best model weights based on validation loss, while the *EarlyStopping* callback is stopping the training early if the validation loss doesn't improve over a specified number of epochs (in this case, 5 consecutive epochs).

Fitting

The model is trained using the `model.fit()` method with the training and validation datasets. Training spans 20 epochs.

```
history = model.fit(
  train_dataset,
  epochs=20,
  validation_data=valid_dataset,
  validation_steps=len(valid_dataset),
  callbacks=[earlyStopping, save_best_model]
)
```

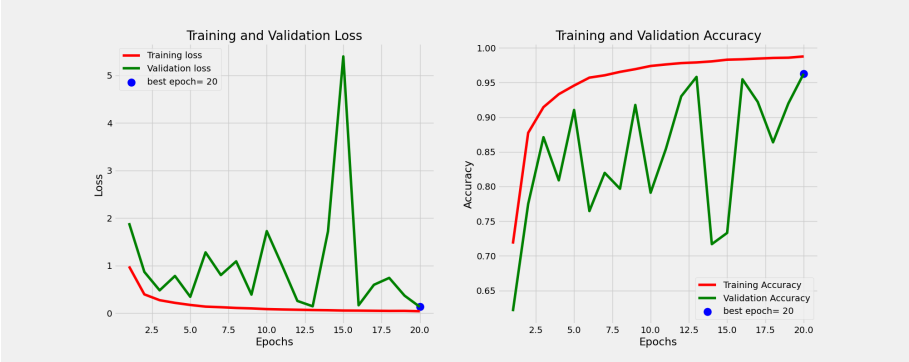

Figure 3.2: Training and Validation Accuracy/Loss of the model

3.2 Feature Extraction

In our study, we explored various pretrained Convolutional Neural Network (CNN) models, such as EfficientNetV2L, EfficientNetB5, and EfficientNetB7. However, these complex models demanded substantial computational resources and time for training, ultimately falling short of our performance expectations.

Recognizing these challenges, we redirected our efforts toward a simpler CNN architecture, VGG16. This decision yielded significantly improved accuracy, highlighting that, in specific scenarios, simplicity can outperform complexity. By adopting VGG16, we streamlined our experimentation, optimized resource allocation, and ultimately achieved more satisfactory outcomes for our research objectives.

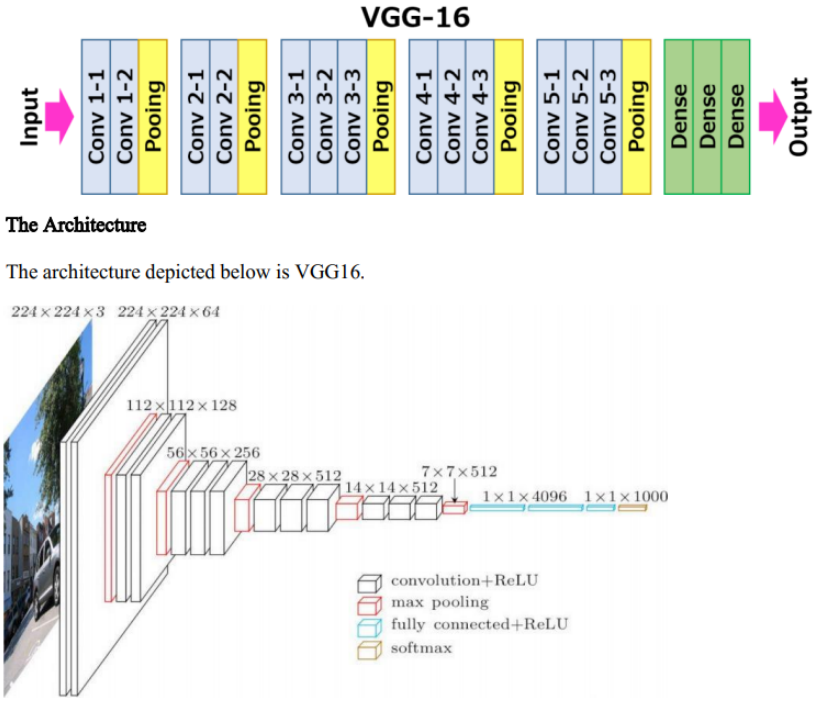

Figure 3.3: VGG16 architecture

3.2.1 Defining the CNN Model

Below is the code to define the CNN model based on VGG16:

```
# Define a CNN model based on pre-trained VGG16
cnn_base = tf.keras.applications.VGG16(
include_top=False,           # Do not include the fully connected layer on top of the
  ↪ model
weights="imagenet",          # Use pre-trained weights from ImageNet
input_shape=(IMAGE_HEIGHT, IMAGE_WIDTH, 3),  # Specify input shape (height, width,
  ↪ channels)
pooling='max',               # Use max-pooling as the pooling operation
)
```

In this implementation, we have used several key parameters to configure the VGG16 model according to our needs.

Parameter *include_top*

The *include_top* parameter is set to 'False', which means we will not include the fully connected layer at the top of the VGG16 model. This is useful when we intend to add our own layers on top to adapt the model for our specific tasks.

Parameter *weights*

The *weights* parameter is set to "imagenet," indicating that we are using pre-trained weights of the VGG16 model trained on ImageNet. This choice allows us to benefit from the knowledge learned by the network on a large dataset.

Parameter *input_shape*

We defined the input_shape parameter to specify the expected dimensions of the input that the model accepts. Both IMAGE_HEIGHT and IMAGE_WIDTH are set to 256. The value 3 signifies the number of channels for RGB images.

Parameter *pooling*

The *pooling* parameter is set to "max", indicating the use of max-pooling as the pooling operation after the convolutional layers. Max-pooling is used to reduce the size of feature maps.

3.2.2 Image Preprocessing and Convolutional Base

In this section, we'll discuss image preprocessing and the configuration of the convolutional base.

Image Preprocessing

We create a sequential model for resizing and rescaling images:

```
resize_and_rescale = tf.keras.Sequential([
layers.Resizing(IMAGE_HEIGHT, IMAGE_WIDTH), # Resize input images
layers.Rescaling(1./255)  # Rescale pixel values to [0, 1]
])
```

This sequential model consists of two layers. The first layer, **layers.Resizing**, resizes input images to the specified dimensions of `IMAGE_HEIGHT` and `IMAGE_WIDTH`. The second layer, **layers.Rescaling**, scales the pixel values of the images to the range [0, 1].

Layers Freezing

We set the convolutional base (`cnn_base`) to be non-trainable:

```
    cnn_base.trainable = False

```

This line ensures that the convolutional layers within the `cnn_base` model will not receive updates during training. Only the additional layers added on top of it will be trained.

3.2.3 Building and Compiling the Pretrained Model

In this section, we'll discuss the construction and compilation of the pretrained model.

Model Architecture

We construct a sequential model by stacking various layers:

```
pretrained_model = tf.keras.Sequential([
resize_and_rescale,  # Image preprocessing and rescaling
data_augmentation,   # Data augmentation for training
cnn_base,            # Pretrained convolutional base
layers.Flatten(),    # Flatten feature maps
layers.Dense(256, activation="relu"),  # Fully connected layer with ReLU activation
layers.BatchNormalization(),           # Batch normalization layer
layers.Dropout(0.4),                   # Dropout layer with 40% dropout rate
layers.Dense(38, activation="softmax") # Output layer with softmax activation
])
```

resize_and_rescale This is the image preprocessing and rescaling step that we discussed earlier.

data_augmentation Data augmentation is used for training, and it helps generate variations of the training data to improve model generalization.

cnn_base The pretrained convolutional base, which was frozen earlier, serves as the feature extractor.

layers.Flatten() This layer is used to flatten the feature maps from the convolutional base.

layers.Dense(256, activation="relu") A fully connected layer with 256 units and ReLU activation.

layers.BatchNormalization() A batch normalization layer to normalize activations.

layers.Dropout(0.4) A dropout layer with a dropout rate of 40% to prevent overfitting.

layers.Dense(38, activation="softmax") The output layer with softmax activation for multi-class classification. The number of units (38) should match the number of classes.

Layer (type)	Output Shape	Param #
sequential_4 (Sequential)	(None, 256, 256, 3)	0
AugmentationLayer (Sequential)	(None, 256, 256, 3)	0
vgg16 (Functional)	(None, 512)	14714688
flatten_3 (Flatten)	(None, 512)	0
dense_6 (Dense)	(None, 256)	131328
batch_normalization_8	(None, 256)	1024
dropout_3 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 38)	9766

Table 3.2: Feature Extraction model

We then proceeded following the same steps as the Neural Network from scratch.

Figure 3.4: Training and Validation Accuracy/Loss of the model

3.3 Fine-Tuning

In this section, we will explore the fine-tuning technique for building a Convolutional Neural Network. It's worth noting that some of the steps involved in constructing a fine-tuned CNN overlap with the steps used for feature extraction in the previously trained model. We will continue with the following steps:

3.3.1 Layers Unfreezing

Next, we unfreeze the last block of VGG16's convolutional layers:

```
# Allow training of the entire `cnn_base`
cnn_base.trainable = True

# Initialize a flag `set_trainable` to False
set_trainable = False

# Loop through the layers of the `cnn_base`
for layer in cnn_base.layers:
  # Check if the layer name is 'block5_conv1'
  if layer.name == 'block5_conv1':
    set_trainable = True

  # If `set_trainable` is True, make the layer trainable; otherwise, freeze it
  if set_trainable:
    layer.trainable = True
  else:
    layer.trainable = False
```

3.3.2 Model Compilation

As for the other models we compile it with an important difference:

```
pretrained_model.compile(
loss='categorical_crossentropy',  # Categorical cross-entropy loss
optimizer=tf.keras.optimizers.Adam(learning_rate=0.00001),  # Adam optimizer with a
  ↪ custom learning rate
metrics=["accuracy"]  # Accuracy metric for evaluation
)
```

The learning rate is equal to 0.00001, this is used to retain valuable features learned during initial training, prevent overfitting, ensure stability, and enable fine-grained adjustments.

3.3.3 Fine-Tuning Configuration

We proceed with fine-tuning and configure callbacks for the process:

```
# Define the model name for fine-tuning
modelName = "FT_base_256d"

# Construct the model path for saving
modelPath = os.path.join(fineTunedPath, modelName + ".keras")

# Configure ModelCheckpoint callback
save_best_model = tf.keras.callbacks.ModelCheckpoint(
modelPath,
monitor='val_loss', verbose=1,
save_best_only=True, save_weights_only=True
)

# Configure EarlyStopping callback
earlyStopping = tf.keras.callbacks.EarlyStopping(
monitor='val_loss', patience=5
)
```

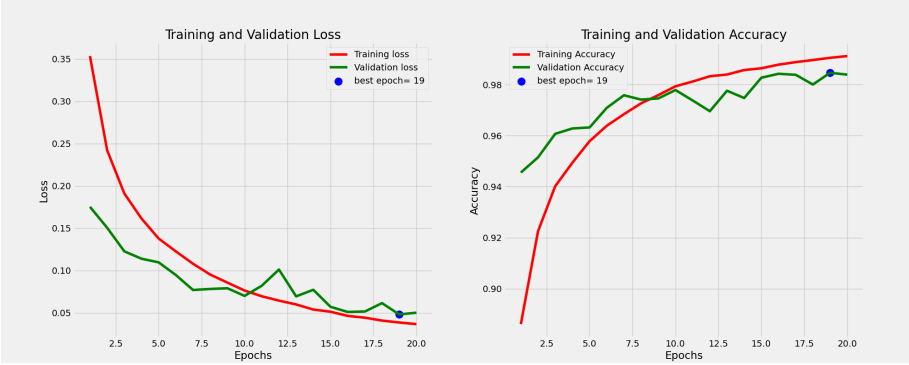

Figure 3.5: Training and Validation Accuracy/Loss of the model

Chapter 4

Results

In conclusion these are the results of our resulting models applied on the test set. Note that the tests are made only on the model from scratch and the fine-tuned one, because the feature extraction one underperforms in every way in comparison of the fine-tuned model.

4.1 From Scratch Model

From scratch	
Test Loss	0.08954
Test Accuracy	97.59%

Table 4.1: Numerical accuracy and loss

Class	Precision	Recall	F1-Score	Support
Strawberry_healthy	1.00	0.89	0.94	63
Tomato_healthy	0.93	1.00	0.96	62
Tomato_Septoria_leaf_spot	1.00	1.00	1.00	28
Cherry_healthy	0.98	0.98	0.98	165
Potato_healthy	0.96	1.00	0.98	150
Peach_Bacterial_spot	1.00	1.00	1.00	105
Grape_Black_rot	0.86	1.00	0.92	86
Tomato_Tomato_mosaic_virus	0.85	1.00	0.92	52
Tomato_Leaf_Mold	1.00	1.00	1.00	119
Strawberry_Leaf_scorch	0.99	0.92	0.95	99
Tomato_Late_blight	1.00	1.00	1.00	116
Corn_healthy	0.94	1.00	0.97	118
Squash_Powdery_mildew	1.00	0.99	1.00	139
Tomato_Early_blight	0.99	1.00	1.00	108
Grape_healthy	0.98	0.98	0.98	43
Cherry_Powdery_mildew	1.00	1.00	1.00	551
Pepper_bell_healthy	0.99	0.98	0.99	230
Peach_healthy	0.97	0.97	0.97	36
Tomato_Tomato_Yellow_Leaf_Curl_Virus	0.94	0.99	0.97	100
Apple_healthy	0.99	0.99	0.99	148
Potato_Late_blight	0.98	1.00	0.99	100
Corn_Northern_Leaf_Blight	1.00	0.95	0.97	100
Pepper_bell_Bacterial_spot	0.88	0.47	0.61	15
Grape_Leaf_blight	0.97	0.97	0.97	37
Raspberry_healthy	0.98	0.99	0.99	509
Apple_Cedar_apple_rust	1.00	0.99	0.99	184
Corn_Common_rust	0.99	1.00	1.00	111
Soybean_healthy	0.96	0.98	0.97	46
Tomato_Bacterial_spot	0.98	0.97	0.97	213
Potato_Early_blight	0.96	0.94	0.95	100
Grape_Esca	0.99	0.93	0.96	191
Tomato_Target_Spot	0.97	0.99	0.98	95
Apple_Apple_scab	0.99	0.88	0.93	177
Blueberry_healthy	0.93	1.00	0.96	159
Accuracy			0.98	5438
Macro Avg	0.97	0.96	0.96	5438
Weighted Avg	0.98	0.98	0.98	5438

Table 4.2: Classification Report

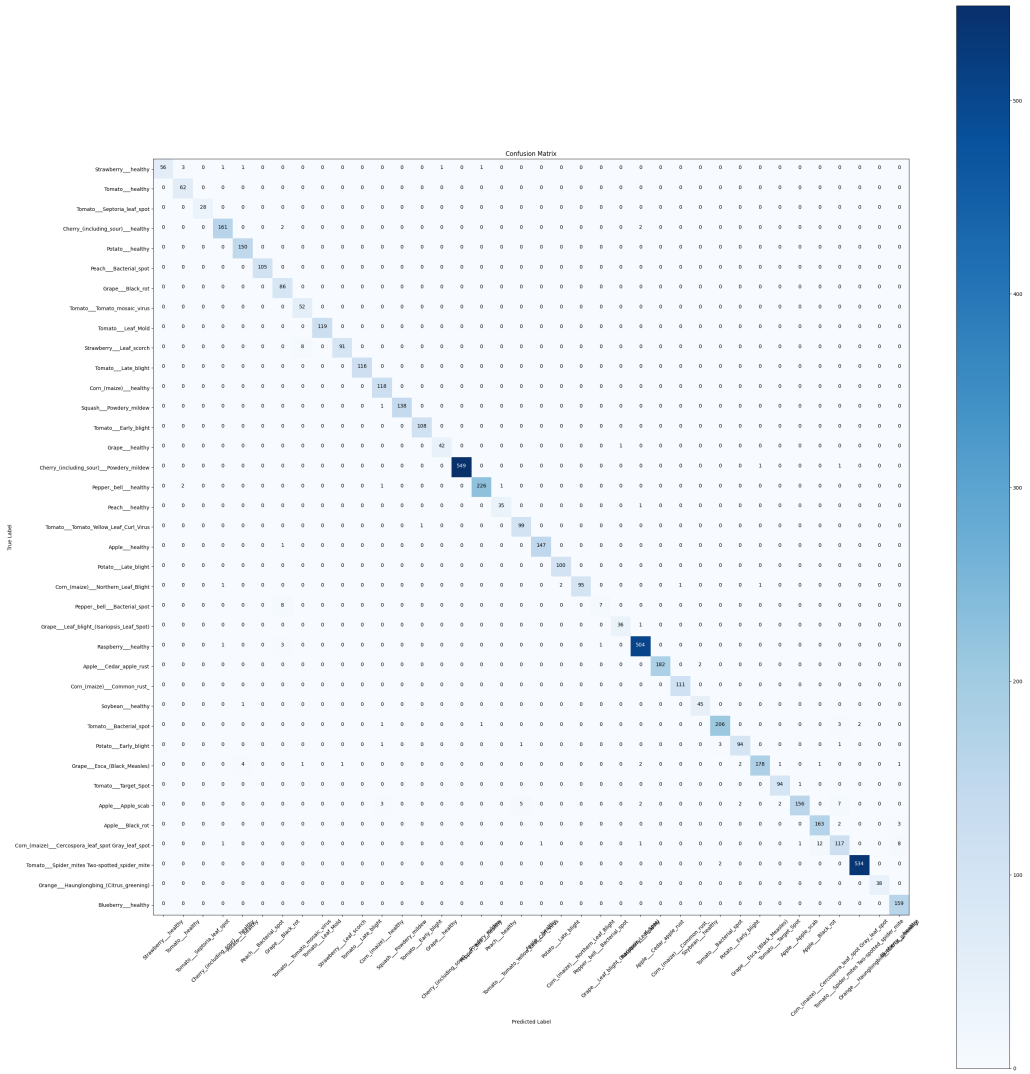

Figure 4.1: Confusion Matrix

Analysis of the confusion matrix and the F1-scores for individual classes reveals that the class posing the most significant challenge is *Pepper_ bell_ Bacterial_ spot*, with an F1-score of 61% (notably, the lowest recall at 47%). This issue may primarily stem from the limited availability of images for this particular class. This hypothesis gains support when considering that all other classes, benefiting from more extensive support, consistently perform at a significantly higher level, consistently exceeding a 90% F1-score.

4.2 Fine Tuned Model

Fine Tuned	
Test Loss	0.03156
Test Accuracy	99.01%

Table 4.3: Numerical accuracy and loss

Class	Precision	Recall	F1-Score	Support
Strawberry_healthy	1.00	0.98	0.99	63
Tomato_healthy	1.00	1.00	1.00	62
Tomato_Septoria_leaf_spot	1.00	1.00	1.00	28
Cherry_healthy	1.00	0.99	1.00	165
Potato_healthy	1.00	1.00	1.00	150
Peach_Bacterial_spot	1.00	1.00	1.00	105
Grape_Black_rot	1.00	1.00	1.00	86
Tomato_Tomato_mosaic_virus	0.98	0.94	0.96	52
Tomato_Leaf_Mold	1.00	1.00	1.00	119
Strawberry_Leaf_scorch	0.96	0.99	0.98	99
Tomato_Late_blight	1.00	1.00	1.00	116
Corn_healthy	1.00	1.00	1.00	118
Squash_Powdery_mildew	1.00	1.00	1.00	139
Tomato_Early_blight	1.00	1.00	1.00	108
Grape_healthy	1.00	1.00	1.00	43
Cherry_Powdery_mildew	1.00	1.00	1.00	551
Pepper_bell_healthy	0.99	0.98	0.99	230
Peach_healthy	0.88	1.00	0.94	36
Tomato_Tomato_Yellow_Leaf_Curl_Virus	1.00	0.99	0.99	100
Apple_healthy	0.99	1.00	1.00	148
Potato_Late_blight	0.99	1.00	1.00	100
Corn_Northern_Leaf_Blight	1.00	0.94	0.97	100
Pepper_bell_Bacterial_spot	0.71	1.00	0.83	15
Grape_Leaf_blight	1.00	1.00	1.00	37
Raspberry_healthy	1.00	1.00	1.00	509
Apple_Cedar_apple_rust	0.99	1.00	1.00	184
Corn_Common_rust	1.00	1.00	1.00	111
Soybean_healthy	1.00	1.00	1.00	46
Tomato_Bacterial_spot	1.00	1.00	1.00	213
Potato_Early_blight	1.00	0.94	0.97	100
Grape_Esca	0.98	0.98	0.98	191
Tomato_Target_Spot	1.00	0.88	0.94	95
Apple_Apple_scab	0.99	0.98	0.99	177
Blueberry_healthy	0.99	1.00	0.99	159
Accuracy			0.99	5438
Macro Avg	0.98	0.99	0.98	5438
Weighted Avg	0.99	0.99	0.99	5438

Table 4.4: Classification Report

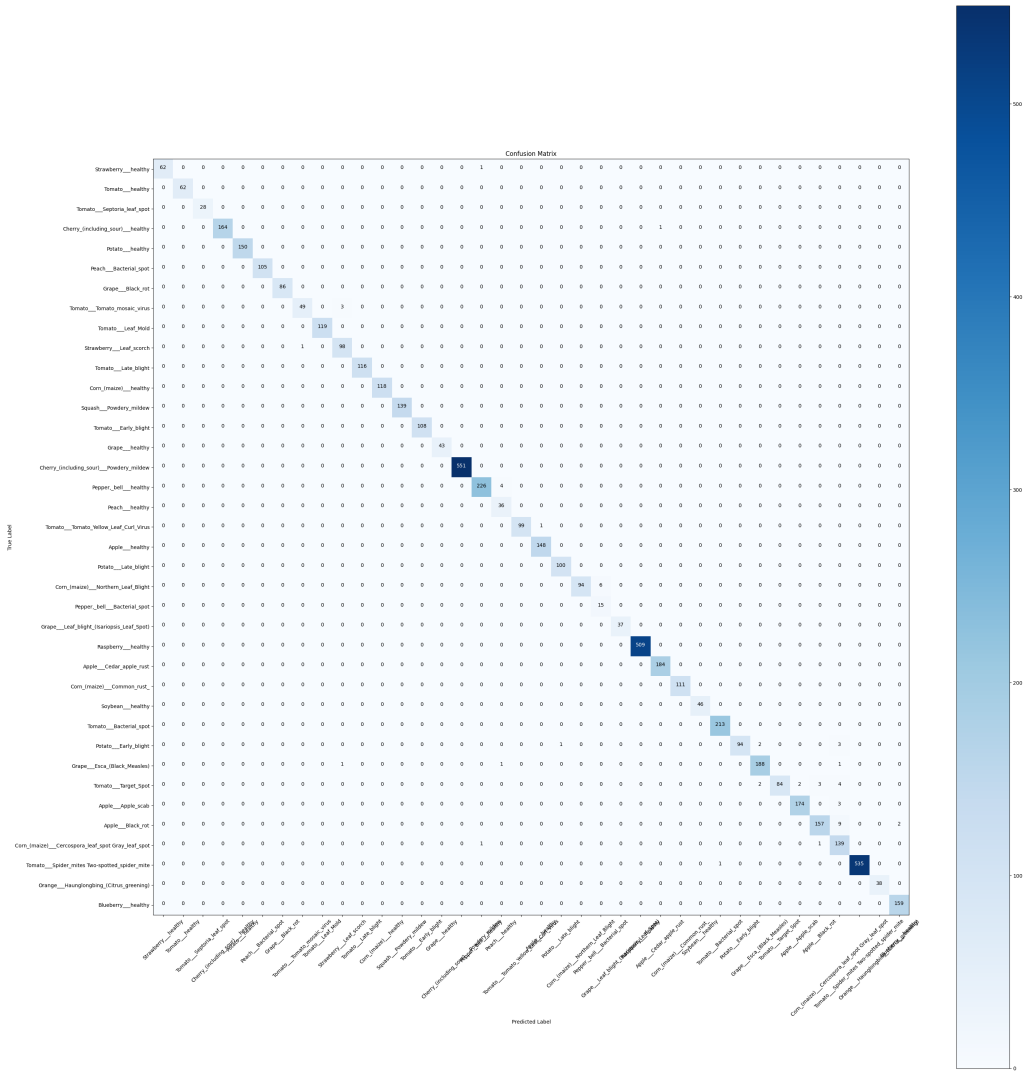

Figure 4.2: Confusion Matrix

Upon examination, it becomes evident that this model has surpassed its predecessor, achieving higher F1-scores and accuracy rates for the majority of classes. Impressively, even in the face of limited support for the *Pepper_ bell_ Bacterial_ spot* class, the model successfully classifies it with a commendable accuracy score of 83%.

4.3 Conclusion

In summary, our study highlights the significant advantages of fine-tuning a pre-trained model over building one from scratch in the leaf classification task. The fine-tuned model demonstrated exceptional performance, achieving an accuracy of 99.01%, which effectively categorizes leaves into multiple classes. While the from-scratch model, while lighter in complexity, still achieved a respectable accuracy of 97.59%, the superior performance of the fine-tuned model underscores the value of leveraging pre-trained networks for image classification tasks.

It's crucial to emphasize that the 99.01% accuracy achieved by the fine-tuned model meets the specific classification requirements of our study, ensuring reliable leaf categorization.

Moreover, it's worth noting that both models could have benefited from better support in the test set.

Additionally, we acknowledge that ensemble methods, although not implemented due to resource and time constraints in our academic setting, could have potentially further improved our

classification results.

In conclusion, our findings underscore the effectiveness of fine-tuning pre-trained models and highlight the practicality of achieving impressive accuracy levels in leaf classification

Bibliography

[1] S.M. Hassan, A.K. Maji, M. Jasi 'nski, Z. Leonowicz, and E. Jasi 'nska. Identification of plant-leaf diseases using cnn and transfer-learning approach. *Electronics*, 10:1388, 2021.

[2] F. Marzougui, M. Elleuch, and M. Kherallah. A deep cnn approach for plant disease detection. *2020 21st International Arab Conference on Information Technology (ACIT)*, pages 1–6, 2020.

[3] Nishant Shelar, Suraj Shinde, Shubham Sawant, Shreyash Dhumal, and Kausar Fakir. Plant disease detection using cnn. *Department of Electronics and Telecommunication, Ramrao Adik Institute of Technology, Navi Mumbai, India*, 2022.

[4] T. Shi, Y. Liu, X. Zheng, and altri. Recent advances in plant disease severity assessment using convolutional neural networks. *Sci Rep*, 13:2336, 2023.