Open in Colab

Module Name: SE4050 - Deep Learning Assignment

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Group memebr details:

IT21251900 - Rajapaksha R.M.S.D

IT21302862 - Sri Samadhi L.A.S.S

IT21178054 - Kumari T.A.T.N

IT21360428 - Monali G.M.N.

Key Objective

The key objective of this project is to develop a machine learning model using Convolutional Neural Networks (CNN) to accurately identify diseases in potato leaves, specifically Early Blight and Late Blight, and determine if the plant is healthy.

Methodology

Supervised Learning

This project leverages supervised learning where the model is trained on labeled data—images of potato leaves categorized as Early Blight, Late Blight, or Healthy. This allows the model to learn features associated with each category and make predictions on new, unseen images.

Convolutional Neural Network (CNN)

CNNs are well-suited for image classification tasks due to their ability to automatically detect and learn hierarchical patterns like edges, textures, and shapes in images. The model architecture consists of multiple convolutional layers, pooling layers, and fully connected layers, which help in extracting features from the potato leaf images and classifying them into one of the three categories.

Dataset

The dataset used in this project is a Potato Leaf Disease Dataset, which consists of labeled images of potato leaves from three categories,

Early Blight: Leaves affected by the Early Blight disease caused by the fungus Alternaria solani.

Late Blight: Leaves affected by the Late Blight disease caused by the pathogen Phytophthora infestans.

Healthy: Leaves that are not affected by any disease and are classified as healthy.

Link: https://www.kaggle.com/datasets/arjuntejaswi/plant-village

Import all the Dependencies

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
```

Set all the Constants

```
BATCH_SIZE = 32
IMAGE_SIZE = 256
CHANNELS=3
EPOCHS=50
```

Import data into tensorflow dataset object

```
import os
import tensorflow as tf

# Step 1: Unzip the file from Google Drive to a local directory
zip_path = '/content/drive/MyDrive/DL_Project/PlantVillage'


dataset = tf.keras.preprocessing.image_dataset_from_directory(
    zip_path,
    seed=123,
    shuffle=True,
    image_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE
)
```

```
Found 2152 files belonging to 3 classes.
```

```
class_names = dataset.class_names
class_names
```

```
['Potato___Early_blight', 'Potato___Late_blight', 'Potato___healthy']
```

```
len(dataset)
```

```
68
```

```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

```
(32, 256, 256, 3)
[1 1 1 0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 0 1 1 2 0 0]
```

Basic Summary Statistics

Start by calculating summary statistics for the dataset. This includes measures like mean, median, standard deviation, and counts of unique values for categorical features.

```
import pandas as pd

df = pd.DataFrame(list(dataset.as_numpy_iterator()))

df.describe(include='all')  # Summary statistics
```

1 to 4 of 4 entries Filter

index	0	1
count	68	68
unique	68	68
top	[[[[129. 125. 139.] [118. 114. 128.] [136. 132. 146.] ... [142. 140. 153.] [86. 87. 92.] [137. 138. 142.]] [[116. 112. 126.] [127. 123. 137.] [150. 146. 160.] ... [134. 132. 145.] [103. 102. 108.] [91. 92. 96.]] [[116. 112. 126.] [142. 138. 152.] [116. 112. 126.] ... [102. 100. 113.] [117. 116. 122.] [111. 110. 115.]] ... [[188. 186. 199.] [192. 190. 203.] [178. 176. 189.] ... [166. 163. 180.] [161. 158. 175.] [165. 162. 179.]] [[177. 175. 188.] [194. 192. 205.] [180. 178. 191.] ... [168. 165. 182.] [163. 160. 177.] [174. 171. 188.]] [[176. 174. 187.] [203. 201. 214.] [191. 189. 202.] ... [174. 171. 188.] [163. 160. 177.] [174. 171. 188.]]] [[[134. 132. 137.] [152. 150. 155.] [173. 171. 176.] ... [179. 176. 183.] [183. 180. 187.] [186. 183. 190.]] [[130. 128. 133.] [132. 130. 135.] [146. 144. 149.] ... [204. 201. 208.] [193. 190. 197.] [174. 171. 178.]] [[148. 146. 151.] [135. 133. 138.] [138. 136. 141.] ... [182. 179. 186.] [189. 186. 193.] [194. 191. 198.]] ... [[146. 143. 152.] [153. 150. 159.] [155. 152. 161.] ... [162. 159. 168.] [161. 158. 167.] [154. 151. 160.]] [[140. 137. 146.] [142. 139. 148.] [147. 144. 153.] ... [150. 147. 156.] [187. 184. 193.] [167. 164. 173.]] [[136. 133. 142.] [129. 126. 135.] [134. 131. 140.] ... [177. 174. 183.] [167. 164. 173.] [171. 168. 177.]]] [[[149. 148. 156.] [148. 147. 155.] [150. 149. 157.] ... [179. 176. 183.] [178. 175. 182.] [175. 172. 179.]] [[154. 153. 161.] [136. 135. 143.] [134. 133. 141.] ... [202. 199. 206.] [202. 199. 206.] [199. 196. 203.]] [[120. 119. 127.] [155. 154. 162.] [129. 128. 136.] ... [188. 185. 192.] [190. 187. 194.] [188. 185. 192.]] ... [[154. 151. 162.] [140. 137. 148.] [142. 139. 150.] ... [176. 173. 184.] [178. 175. 186.] [178. 175. 186.]] [[156. 153. 164.] [139. 136. 147.] [140. 137. 148.] ... [168. 165. 176.] [164. 161. 172.] [160. 157. 168.]] [[157. 154. 165.] [138. 135. 146.] [135. 132. 143.] ... [186. 183. 194.] [180. 177. 188.] [172. 169. 180.]]] ... [[[160. 162. 175.] [162. 164. 177.] [168. 170. 183.] ... [168. 166. 177.] [148. 146. 157.] [132. 130. 141.]] [[158. 160. 173.] [164. 166. 179.] [175. 177. 190.] ... [142. 140. 151.] [139. 137. 148.] [141. 139. 150.]] [[156. 158. 171.] [162. 164. 177.] [174. 176. 189.] ... [150. 148. 159.] [147. 145. 156.] [132. 130. 141.]] ... [[184. 183. 191.] [185. 184. 192.] [186. 185. 193.] ... [144. 141. 148.] [189. 186. 193.] [169. 166. 173.]] [[179. 178. 186.] [180. 179. 187.] [182. 181. 189.] ... [170. 167. 174.] [157. 154. 161.] [156. 153. 160.]] [[181. 180. 188.] [184. 183. 191.] [187. 186. 194.] ... [140. 137. 144.] [166. 163. 170.] [150. 147. 154.]]] [[[130. 117. 126.] [131. 118. 127.] [133. 120. 129.] ... [157. 147. 156.] [160. 150. 159.] [164. 154. 163.]] [[132. 119. 128.] [132. 119. 128.] [134. 121. 130.] ... [157. 147. 156.] [159. 149. 158.] [161. 151. 160.]] [[135. 122. 131.] [135. 122. 131.] [135. 122. 131.] ... [158. 148. 157.] [158. 148. 157.] [157. 147. 156.]] ... [[143. 130. 137.] [171. 158. 165.] [169. 156. 163.] ... [168. 161. 168.] [171. 164. 171.] [184. 177. 184.]] [[155. 142. 149.] [155. 142. 149.] [158. 145. 152.] ... [193. 186. 193.] [183. 176. 183.] [168. 161. 168.]] [[160. 147. 154.] [145. 132. 139.] [166. 153. 160.] ... [163. 156. 163.] [156. 149. 156.] [168. 161. 168.]]] [[[161. 158. 169.] [172. 169. 180.] [179. 176. 187.] ... [194. 193. 201.] [191. 190. 198.] [188. 187. 195.]] [[169. 166. 177.] [177. 174. 185.] [181. 178. 189.] ... [191. 190. 198.] [190. 189. 197.] [190. 189. 197.]] [[171. 168. 179.] [174. 171. 182.] [175. 172. 183.] ... [188. 187.	[0 0 0 1 0 0 1 0]

View the DataFrame Columns

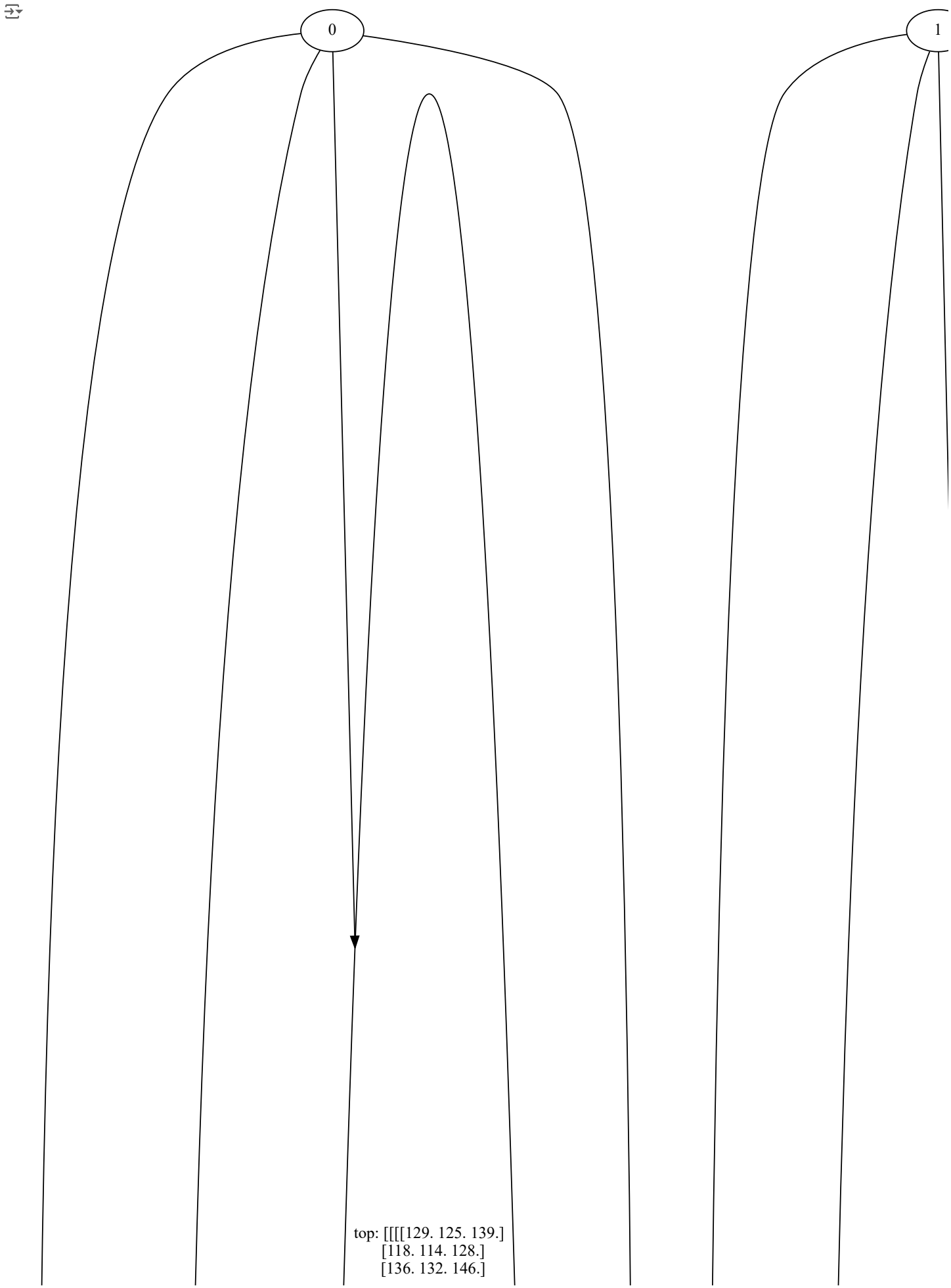
```
print(df.columns)
print("Shape of the DataFrame:", df.shape)
print("Columns in the DataFrame:", df.columns.tolist())
```

```
RangeIndex(start=0, stop=2, step=1)
Shape of the DataFrame: (68, 2)
Columns in the DataFrame: [0, 1]
```

```
pip install pandas matplotlib graphviz
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.2.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (0.20.3)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
```

```
import pandas as pd
from graphviz import Digraph

# Generate summary statistics
summary_stats = df.describe(include='all')

# Initialize a graph
dot = Digraph(comment='Summary Statistics')

# Add nodes for each statistic
for col in summary_stats.columns:
    # Convert column name to string and clean it
    col_name = str(col).replace(" ", "_")
    dot.node(col_name, col_name)
    for stat in summary_stats.index:
        stat_value = summary_stats.at[stat, col]
        # Convert stat to string for the node identifier
        stat_name = f"{col_name}_{str(stat).replace(' ', '_')}"
        dot.node(stat_name, f"{stat}: {stat_value}")
        dot.edge(col_name, stat_name)

# Render the flowchart
dot.render('summary_statistics_flowchart', format='png', cleanup=True)
dot.view()
display(dot)
```


0
1
top: [[[[129. 125. 139.]
[118. 114. 128.]
[136. 132. 146.]

...
[142. 140. 153.]
[86. 87. 92.]
[137. 138. 142.]]

[[116. 112. 126.]
[127. 123. 137.]
[150. 146. 160.]
...
[134. 132. 145.]
[103. 102. 108.]
[91. 92. 96.]]

[[116. 112. 126.]
[142. 138. 152.]
[116. 112. 126.]
...
[102. 100. 113.]
[117. 116. 122.]
[111. 110. 115.]]

...

[[188. 186. 199.]
[192. 190. 203.]
[178. 176. 189.]
...
[166. 163. 180.]
[161. 158. 175.]
[165. 162. 179.]]

[[177. 175. 188.]
[194. 192. 205.]
[180. 178. 191.]
...
[168. 165. 182.]
[163. 160. 177.]
[174. 171. 188.]]

[[176. 174. 187.]
[203. 201. 214.]
[191. 189. 202.]
...
[174. 171. 188.]
[163. 160. 177.]
[174. 171. 188.]]]

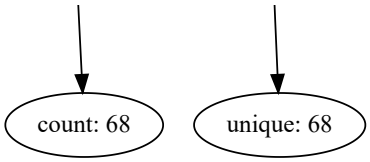
[[[134. 132. 137.]
[152. 150. 155.]
[173. 171. 176.]
...
[179. 176. 183.]
[183. 180. 187.]
[186. 183. 190.]]

[[130. 128. 133.]
[132. 130. 135.]
[146. 144. 149.]
...
[204. 201. 208.]
[193. 190. 197.]
[174. 171. 178.]]

[[148. 146. 151.]
[135. 133. 138.]
[138. 136. 141.]
...
[182. 179. 186.]
[189. 186. 193.]
[194. 191. 198.]]

...

```
[[146. 143. 152.]
 [153. 150. 159.]
 [155. 152. 161.]
 ...
 [162. 159. 168.]
 [161. 158. 167.]
 [154. 151. 160.]]

[[140. 137. 146.]
 [142. 139. 148.]
 [147. 144. 153.]
 ...
 [150. 147. 156.]
 [187. 184. 193.]
 [167. 164. 173.]]

[[136. 133. 142.]
 [129. 126. 135.]
 [134. 131. 140.]
 ...
 [177. 174. 183.]
 [167. 164. 173.]
 [171. 168. 177.]]]


[[[149. 148. 156.]
 [148. 147. 155.]
 [150. 149. 157.]
 ...
 [179. 176. 183.]
 [178. 175. 182.]
 [175. 172. 179.]]

[[154. 153. 161.]
 [136. 135. 143.]
 [134. 133. 141.]
 ...
 [202. 199. 206.]
 [202. 199. 206.]
 [199. 196. 203.]]

[[120. 119. 127.]
 [155. 154. 162.]
 [129. 128. 136.]
 ...
 [188. 185. 192.]
 [190. 187. 194.]
 [188. 185. 192.]]

 ...

[[154. 151. 162.]
 [140. 137. 148.]
 [142. 139. 150.]
 ...
 [176. 173. 184.]
 [178. 175. 186.]
 [178. 175. 186.]]

[[156. 153. 164.]
 [139. 136. 147.]
 [140. 137. 148.]
 ...
 [168. 165. 176.]
 [164. 161. 172.]
 [160. 157. 168.]]

[[157. 154. 165.]
 [138. 135. 146.]
 [135. 132. 143.]
 ...
 [186. 183. 194.]
```



[180. 177. 188.]
[172. 169. 180.]]]

...

[[[160. 162. 175.]
[162. 164. 177.]
[168. 170. 183.]
...
[168. 166. 177.]
[148. 146. 157.]
[132. 130. 141.]]

[[158. 160. 173.]
[164. 166. 179.]
[175. 177. 190.]
...
[142. 140. 151.]
[139. 137. 148.]
[141. 139. 150.]]

[[156. 158. 171.]
[162. 164. 177.]
[174. 176. 189.]
...
[150. 148. 159.]
[147. 145. 156.]
[132. 130. 141.]]

...

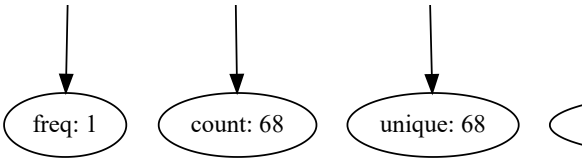
[[184. 183. 191.]
[185. 184. 192.]
[186. 185. 193.]
...
[144. 141. 148.]
[189. 186. 193.]
[169. 166. 173.]]

[[179. 178. 186.]
[180. 179. 187.]
[182. 181. 189.]
...
[170. 167. 174.]
[157. 154. 161.]
[156. 153. 160.]]

[[181. 180. 188.]
[184. 183. 191.]
[187. 186. 194.]
...
[140. 137. 144.]
[166. 163. 170.]
[150. 147. 154.]]]

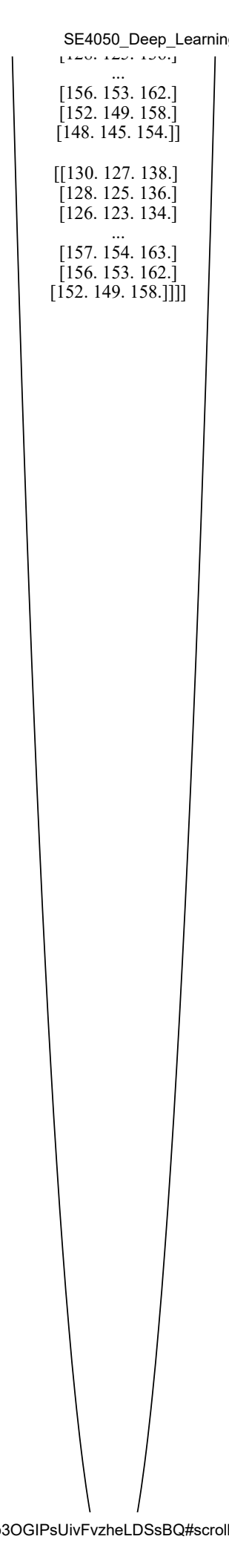
[[[130. 117. 126.]
[131. 118. 127.]
[133. 120. 129.]
...
[157. 147. 156.]
[160. 150. 159.]
[164. 154. 163.]]

[[132. 119. 128.]
[132. 119. 128.]
[134. 121. 130.]
...
[157. 147. 156.]
[159. 149. 158.]
[161. 151. 160.]]



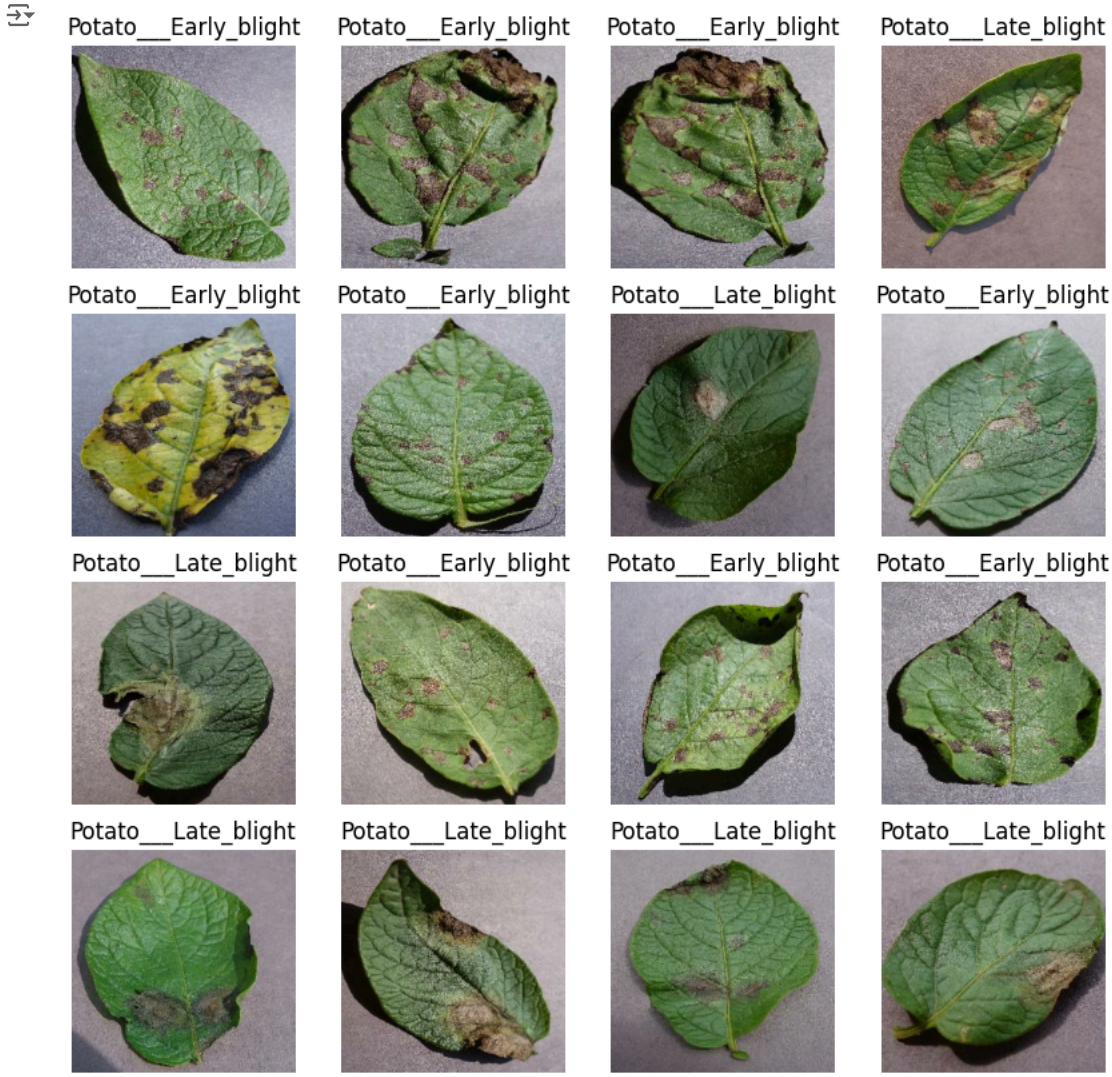
```
[[135. 122. 131.]
 [135. 122. 131.]
 [135. 122. 131.]
 ...
 [158. 148. 157.]
 [158. 148. 157.]
 [157. 147. 156.]]

...

[[143. 130. 137.]
 [171. 158. 165.]
 [169. 156. 163.]
 ...
 [168. 161. 168.]
 [171. 164. 171.]
 [184. 177. 184.]]

[[155. 142. 149.]
 [155. 142. 149.]
 [158. 145. 152.]
 ...
 [193. 186. 193.]
 [183. 176. 183.]
 [168. 161. 168.]]

[[160. 147. 154.]
 [145. 132. 139.]
 [166. 153. 160.]
 ...
 [163. 156. 163.]
 [156. 149. 156.]
 [168. 161. 168.]]]


[[[161. 158. 169.]
 [172. 169. 180.]
 [179. 176. 187.]
 ...
 [194. 193. 201.]
 [191. 190. 198.]
 [188. 187. 195.]]

[[169. 166. 177.]
 [177. 174. 185.]
 [181. 178. 189.]
 ...
 [191. 190. 198.]
 [190. 189. 197.]
 [190. 189. 197.]]

[[171. 168. 179.]
 [174. 171. 182.]
 [175. 172. 183.]
 ...
 [188. 187. 195.]
 [190. 189. 197.]
 [193. 192. 200.]]

...

[[126. 123. 134.]
 [121. 118. 129.]
 [119. 116. 127.]
 ...
 [153. 150. 159.]
 [150. 147. 156.]
 [146. 143. 152.]]

[[138. 135. 146.]
 [133. 130. 141.]
 [128. 125. 136.]
```

```
[128. 129. 130.]
...
[156. 153. 162.]
[152. 149. 158.]
[148. 145. 154.]]

[[130. 127. 138.]
[128. 125. 136.]
[126. 123. 134.]
...
[157. 154. 163.]
[156. 153. 162.]
[152. 149. 158.]]]]
```



Visualize some of the images from our dataset

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for image_batch, labels_batch in dataset.take(1):
    for i in range(16):
        ax = plt.subplot(4, 4, i + 1) # Changed the grid to 4x4 to accommodate 16 images
        plt.imshow(image_batch[i].numpy().astype("uint8"))
        plt.title(class_names[labels_batch[i]])
        plt.axis("off")
```



Function to Split Dataset Dataset should be bifurcated into 3 subsets, namely: **bold text**

Training: Dataset to be used while training Validation: Dataset to be tested against while training Test: Dataset to be tested against after we trained a model

```
len(dataset)
```

```
68
```

```
train_size = 0.8
len(dataset)*train_size
```

```
54.400000000000006
```

```
train_ds = dataset.take(54)
len(train_ds)
```

```
54
```

```
test_ds = dataset.skip(54)
len(test_ds)
```

```
14
```

```
val_size=0.1
len(dataset)*val_size
```

```
6.800000000000001
```

```
val_ds = test_ds.take(6)
len(val_ds)
```

```
6
```

```
test_ds = test_ds.skip(6)
len(test_ds)
```

```
8
```

The get_dataset_partitions_tf function partitions a TensorFlow dataset into training, validation, and test sets based on specified proportions. It accepts parameters for the split ratios (with defaults of 80% for training, 10% for validation, and 10% for testing), a shuffle flag to randomize the dataset before partitioning, and a shuffle size for the randomization buffer.

```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1, shuffle=True, shuffle_size=10000):
    assert (train_split + test_split + val_split) == 1

    ds_size = len(ds)

    if shuffle:
        ds = ds.shuffle(shuffle_size, seed=12)

    train_size = int(train_split * ds_size)
    val_size = int(val_split * ds_size)

    train_ds = ds.take(train_size)
    val_ds = ds.skip(train_size).take(val_size)
    test_ds = ds.skip(train_size).skip(val_size)

    return train_ds, val_ds, test_ds
```

```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

```
len(train_ds)
```

```
54
```

```
len(val_ds)
```

```
6
```

```
len(test_ds)
```

```
8
```

IT21302862 - Model 01

Dataset Caching and Prefetching

To optimize the performance and reduce latency during training, we use caching, shuffling, and prefetching. These techniques allow us to preprocess the data while the model is training. AUTOTUNE is used to adjust the prefetching buffer size automatically to improve efficiency.

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

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

IT21302862 - Building the Model

Creating a Layer for Resizing and Normalization

Before feed our images to network, we should be resizing it to the desired size. Moreover, to improve model performance, we should normalize the image pixel value (keeping them in range 0 and 1 by dividing by 256). This should happen while training as well as inference. Hence we can add that as a layer in our Sequential Model.

```
resize_and_rescale = tf.keras.Sequential([
  tf.keras.layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  tf.keras.layers.Rescaling(1./255),
])
```

Data Augmentation

This boosts the accuracy of our model by augmenting the data. It applies random transformations to the input images such as flipping them horizontally and vertically, and rotating them by a random factor (here up to 20% of the image). By augmenting the data, we reduce overfitting,improve the model's ability to generalize, and make it more robust to variations in the input data.

```
import tensorflow as tf

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

Applying Data Augmentation to Train Dataset

The data augmentation layer is applied only during training (not during validation or testing) using the training=True flag. The map function is used to apply augmentation to each image in the training dataset, maintaining their corresponding labels.

```
train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

IT21302862 - Model Architecture

We use a CNN coupled with a Softmax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

First Conv2D Layer

A 2D convolution layer with 32 filters, each of size 3x3. The ReLU activation function introduces non-linearity. This layer is responsible for learning basic image features like edges and textures. The input shape is specified as (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS), matching the size and channels of the input images.

Second Conv2D Layer

Another Conv2D layer with 64 filters and 3x3 kernels. This layer learns more complex features as it builds on the previous one. Using 64 filters allows the network to capture a higher variety of features. .

Third Conv2D Layer

A third Conv2D layer with 64 filters and 3x3 kernels. This deeper layer captures more abstract patterns in the input images.

Fourth Conv2D Layer

A fourth Conv2D layer with 64 filters, using the same kernel size (3x3). As the model deepens, it extracts higher-level features such as shapes and textures.

Fifth Conv2D Layer

A fifth Conv2D layer, still with 64 filters, which allows the model to capture increasingly complex features.

Sixth Conv2D Layer

A sixth Conv2D layer, again with 64 filters. This continues to deepen the model and capture advanced-level features.

Flatten Layer

The flattening layer converts the 2D feature maps into a 1D vector that can be fed into fully connected layers. This prepares the data for the Dense (fully connected) layers.

First Dense Layer

A fully connected layer with 64 units. The ReLU activation function introduces non-linearity. This layer helps in learning combinations of the features extracted by the Conv2D layers.

```
input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 3

model = models.Sequential([
    resize_and_rescale,
    layers.Conv2D(32, kernel_size = (3,3), activation='relu', input_shape=input_shape),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(n_classes, activation='softmax'),
])

model.build(input_shape=input_shape)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
sequential (Sequential)	(32, 256, 256, 3)	0
conv2d (Conv2D)	(32, 254, 254, 32)	896
max_pooling2d (MaxPooling2D)	(32, 127, 127, 32)	0
conv2d_1 (Conv2D)	(32, 125, 125, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(32, 62, 62, 64)	0
conv2d_2 (Conv2D)	(32, 60, 60, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(32, 30, 30, 64)	0
conv2d_3 (Conv2D)	(32, 28, 28, 64)	36,928
max_pooling2d_3 (MaxPooling2D)	(32, 14, 14, 64)	0
conv2d_4 (Conv2D)	(32, 12, 12, 64)	36,928
max_pooling2d_4 (MaxPooling2D)	(32, 6, 6, 64)	0
conv2d_5 (Conv2D)	(32, 4, 4, 64)	36,928
max_pooling2d_5 (MaxPooling2D)	(32, 2, 2, 64)	0
flatten (Flatten)	(32, 256)	0
dense (Dense)	(32, 64)	16,448
dense_1 (Dense)	(32, 3)	195

Total params: 183,747 (717.76 KB)
Trainable params: 183,747 (717.76 KB)
Non-trainable params: 0 (0.00 B)

IT21302862 - Compiling the Model

We use adam Optimizer, SparseCategoricalCrossentropy for losses, accuracy as a metric

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

Model Training

This section trains the CNN model on the training dataset (`train_ds`) with the specified batch size and number of epochs. The `validation_data` parameter allows the model to evaluate its performance on the validation set (`val_ds`) after each epoch. The `verbose=1` option provides detailed output about the training process. The training history (loss, accuracy, etc.) will be stored in the `history` object, which can be used for further analysis or visualization.

```
history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=EPOCHS,
)
```

```
Epoch 1/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 532s 526ms/step - accuracy: 0.4962 - loss: 0.9273 - val_accuracy: 0.4375 - val_loss: 0.9524
Epoch 2/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.5856 - loss: 0.8384 - val_accuracy: 0.7344 - val_loss: 0.5681
Epoch 3/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 355ms/step - accuracy: 0.7749 - loss: 0.4962 - val_accuracy: 0.8281 - val_loss: 0.4016
Epoch 4/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 349ms/step - accuracy: 0.8187 - loss: 0.4167 - val_accuracy: 0.8542 - val_loss: 0.3371
Epoch 5/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 348ms/step - accuracy: 0.8465 - loss: 0.3436 - val_accuracy: 0.8594 - val_loss: 0.3506
Epoch 6/50
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 341ms/step - accuracy: 0.8694 - loss: 0.3505 - val_accuracy: 0.8906 - val_loss: 0.3042
Epoch 7/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.8977 - loss: 0.2645 - val_accuracy: 0.9271 - val_loss: 0.2054
Epoch 8/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 364ms/step - accuracy: 0.9181 - loss: 0.2084 - val_accuracy: 0.8854 - val_loss: 0.2818
Epoch 9/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9146 - loss: 0.2282 - val_accuracy: 0.9219 - val_loss: 0.1931
Epoch 10/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9413 - loss: 0.1647 - val_accuracy: 0.9375 - val_loss: 0.1661
Epoch 11/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 371ms/step - accuracy: 0.9316 - loss: 0.1777 - val_accuracy: 0.9062 - val_loss: 0.2162
Epoch 12/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.9248 - loss: 0.1862 - val_accuracy: 0.9479 - val_loss: 0.1295
Epoch 13/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9455 - loss: 0.1391 - val_accuracy: 0.9427 - val_loss: 0.1141
Epoch 14/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 338ms/step - accuracy: 0.9470 - loss: 0.1309 - val_accuracy: 0.9115 - val_loss: 0.2137
Epoch 15/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 18s 339ms/step - accuracy: 0.9382 - loss: 0.1726 - val_accuracy: 0.9531 - val_loss: 0.1038
Epoch 16/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 22s 366ms/step - accuracy: 0.9620 - loss: 0.1110 - val_accuracy: 0.8958 - val_loss: 0.2763
Epoch 17/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 342ms/step - accuracy: 0.9638 - loss: 0.0961 - val_accuracy: 0.9219 - val_loss: 0.2160
Epoch 18/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9608 - loss: 0.1153 - val_accuracy: 0.9375 - val_loss: 0.1026
Epoch 19/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9686 - loss: 0.0827 - val_accuracy: 0.9427 - val_loss: 0.1148
Epoch 20/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9655 - loss: 0.0805 - val_accuracy: 0.9688 - val_loss: 0.0711
Epoch 21/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9736 - loss: 0.0712 - val_accuracy: 0.9583 - val_loss: 0.1084
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 354ms/step - accuracy: 0.9808 - loss: 0.0458 - val_accuracy: 0.9375 - val_loss: 0.1208
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.9823 - loss: 0.0587 - val_accuracy: 0.9531 - val_loss: 0.1119
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 358ms/step - accuracy: 0.9826 - loss: 0.0600 - val_accuracy: 0.8854 - val_loss: 0.2718
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 343ms/step - accuracy: 0.9467 - loss: 0.1361 - val_accuracy: 0.9792 - val_loss: 0.0515
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9846 - loss: 0.0337 - val_accuracy: 0.9844 - val_loss: 0.0379
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 368ms/step - accuracy: 0.9807 - loss: 0.0551 - val_accuracy: 0.9375 - val_loss: 0.1078
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 350ms/step - accuracy: 0.9865 - loss: 0.0360 - val_accuracy: 0.9531 - val_loss: 0.1691
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9779 - loss: 0.0628 - val_accuracy: 0.9479 - val_loss: 0.1723
```

```
scores = model.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - accuracy: 0.9763 - loss: 0.0931
```

```
scores
```

```
[0.092591293156147, 0.97265625]
```

Plotting the Accuracy and Loss Curves

```
history
```

```
<keras.src.callbacks.history.History at 0x7dfd700e3610>
```

```
history.params
```

```
{'verbose': 1, 'epochs': 50, 'steps': 54}
```

```
history.history.keys()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
type(history.history['loss'])
```

```
list
```

```
len(history.history['loss'])
```

```
50
```

```
history.history['loss'][:5]
```

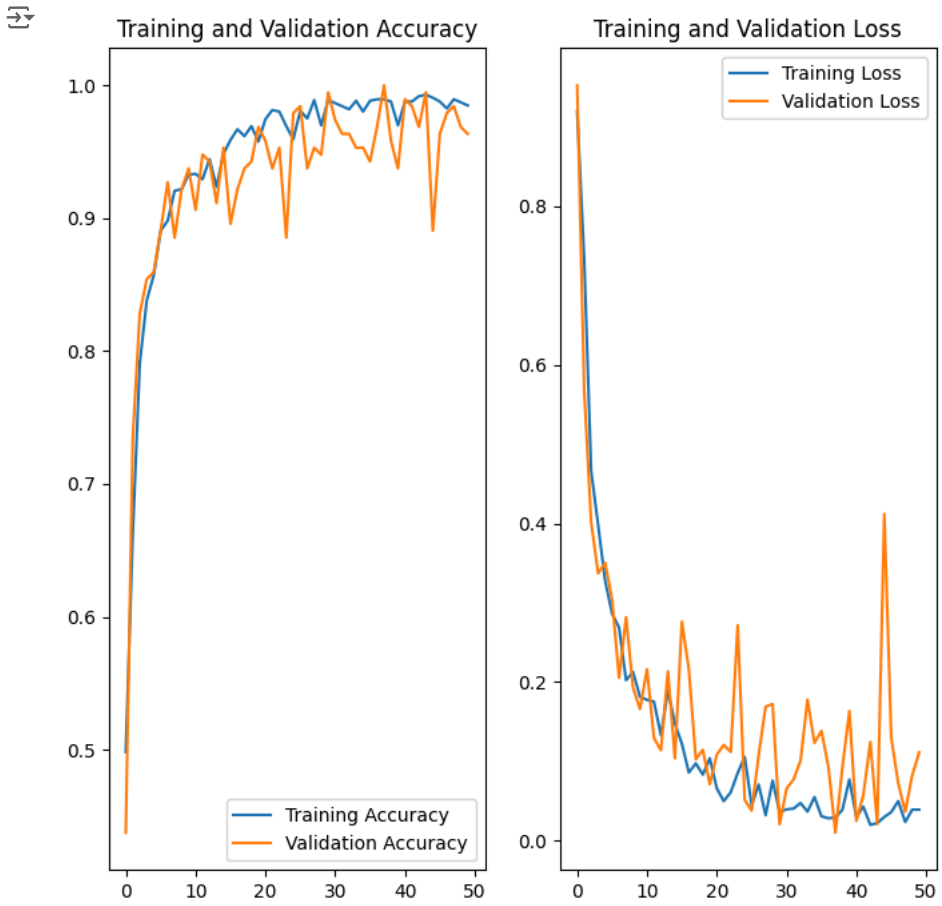
```
[0.9199034571647644,
 0.7345544099807739,
 0.4664019048213959,
 0.3981457054615021,
 0.3280259966850281]
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']
```

Plotting Training and Validation Accuracy

The first subplot displays the training accuracy and validation accuracy over epochs. The x-axis represents the number of epochs, while the y-axis represents accuracy values. `acc` refers to the list of training accuracy values, and `val_acc` refers to the validation accuracy values.

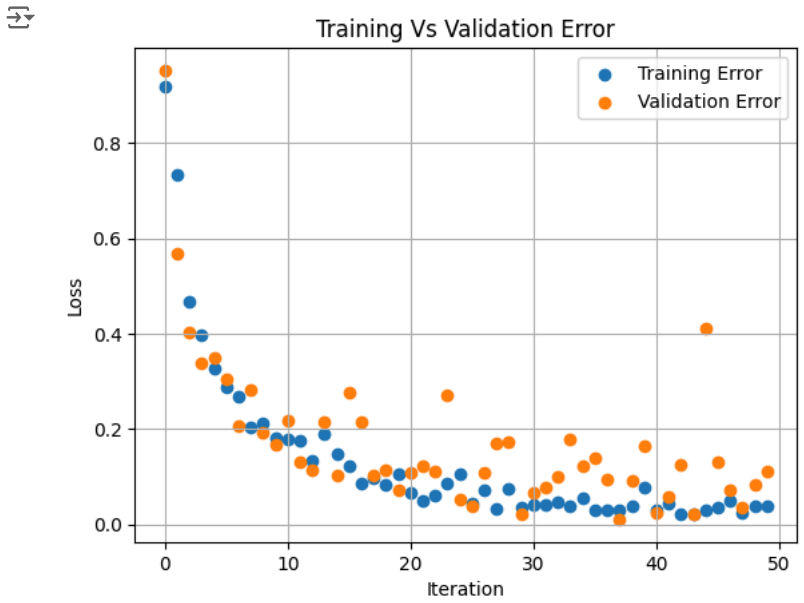
```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(len(loss)), loss, label='Training Loss')
plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



Scatter Plot: Training vs Validation Error

This plot visualizes the loss (error) during training and validation across different iterations (epochs). The x-axis represents the number of epochs, while the y-axis shows the loss values. `history.history['loss']` contains the training loss values, and `history.history['val_loss']` contains the validation loss. The scatter plot allows easy comparison between training and validation errors to assess how well the model is generalizing.

```
plt.scatter(x=history.epoch,y=history.history['loss'],label='Training Error')
plt.scatter(x=history.epoch,y=history.history['val_loss'],label='Validation Error')
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Vs Validation Error')
plt.legend()
plt.show()
```

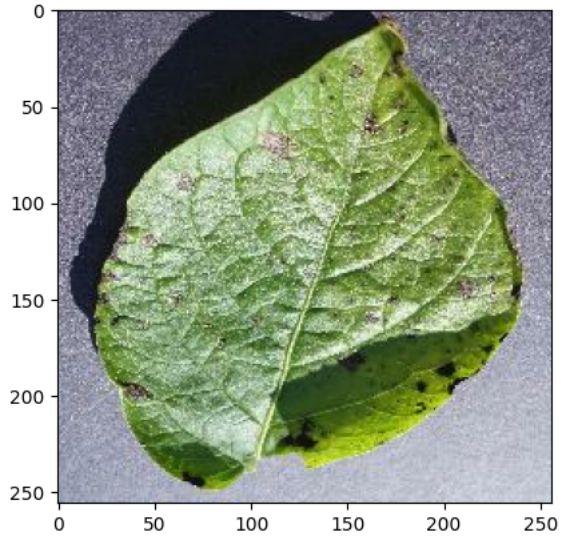


Run prediction on a sample image

This code takes a batch of images from the test dataset and selects the first image and its corresponding label. It displays the image using plt.imshow() and prints its actual label. Then, the model makes predictions on the batch, and the predicted label for the first image is printed. This helps compare the model's prediction with the actual label for that image.

```
import numpy as np
for images_batch, labels_batch in test_ds.take(1):

    first_image = images_batch[0].numpy().astype('uint8')
    first_label = labels_batch[0].numpy()

    print("first image to predict")
    plt.imshow(first_image)
    print("actual label:",class_names[first_label])

    batch_prediction = model.predict(images_batch)
    print("predicted label:",class_names[np.argmax(batch_prediction[0])])
```

```
first image to predict
actual label: Potato___Early_blight
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 393ms/step
predicted label: Potato___Early_blight
```



Write a function for inference

```
def predict(model, img):
    img_array = tf.keras.preprocessing.image.img_to_array(images[i].numpy())
    img_array = tf.expand_dims(img_array, 0)

    predictions = model.predict(img_array)

    predicted_class = class_names[np.argmax(predictions[0])]
    confidence = round(100 * (np.max(predictions[0])), 2)
    return predicted_class, confidence
```

The following code snippet visualizes the predictions by taking a batch of images from the test dataset. It displays a 3x3 grid of images, showing the actual and predicted classes along with the confidence for each image. Each subplot presents the image, its actual label, the predicted label, and the prediction confidence percentage, providing a clear comparison of the model's performance.

```
plt.figure(figsize=(15, 15))
for images, labels in test_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))

        predicted_class, confidence = predict(model, images[i].numpy())
        actual_class = class_names[labels[i]]

        plt.title(f"Actual: {actual_class},\n Predicted: {predicted_class}.\n Confidence: {confidence}%")

        plt.axis("off")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 874ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
```

Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 100.0%

Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 100.0%

Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 100.0%

Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 98.73%

Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 99.79%

Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 76.04%

Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 81.51%

Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 100.0%

Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 99.38%

IT21302862 - Saving the Model

We append the model to the list of models as a new version

```
import os

# Create the directory if it doesn't exist
if not os.path.exists("../models"):
    os.makedirs("../models")

# Extract file names without extensions and convert them to integers
model_files = [f for f in os.listdir("../models") if f.endswith(".keras")]
model_versions = [int(f.split('.')[0]) for f in model_files]

# Get the maximum model version, or default to 0 if no models exist
model_version = max(model_versions + [0]) + 1

# Save the new model with the next version number
model.save(f"../models/{model_version}.keras")
```

```
model.save("../potatoes.keras")
```

```
import os

model_path = os.path.abspath(f"../models/{model_version}.keras")
print(f"Model saved at: {model_path}")
```

```
Model saved at: /models/1.keras
```

```
import os
print(os.getcwd())

print(os.path.exists("../models"))
!ls /models
```

```
/content
True
1.keras
```

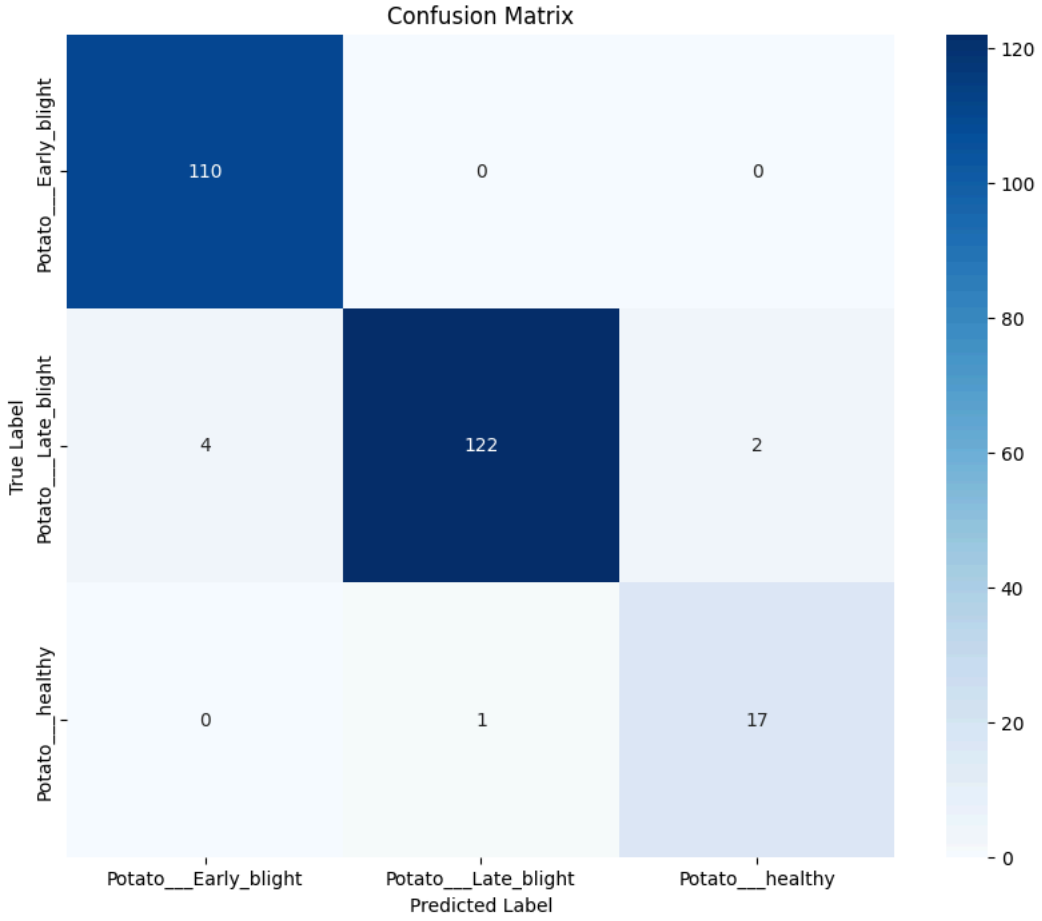
```
import tensorflow as tf
from tensorflow.keras import metrics

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        metrics.Precision(name='precision'),
        metrics.Recall(name='recall')
    ]
)
```

Calculates the confusion matrix based on the true and predicted labels.The code plots the confusion matrix using Seaborn's heatmap, displaying the counts of true positives, false positives, and false negatives for each class. The resulting heatmap provides a visual representation of the model's classification performance, with labeled axes for easy interpretation.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Step 1: Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)
```

```
    y_true.extend(labels.numpy())
    y_pred.extend(predicted_classes)

# Step 2: Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Step 3: Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 427ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
```



IT21251900 - Model 02

IT21251900 - Resizing and Normalization

In this section, we are preparing our dataset before feeding it into the neural network. We need to ensure that all images have a consistent size and their pixel values are normalized for better model performance.

Resizing and Rescaling Layer

We use the Sequential API to create a preprocessing pipeline that resizes images to 224x224 pixels (the input size expected by VGG16) and normalizes the pixel values to a range between 0 and 1 by dividing them by 255.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential # Import Sequential from tensorflow.keras.models
from tensorflow.keras import layers

resize_and_rescale = Sequential([
    layers.Resizing(224, 224),  # Resize to 224x224 (VGG16 input size)
    layers.Rescaling(1./255),   # Normalize pixel values between 0 and 1
])
```

Mapping Dataset

We apply the resize_and_rescale layer to the training, validation, and test datasets using the map function. This ensures that every image in each dataset is resized and normalized before feeding it into the model.

```
train_ds = train_ds.map(lambda x, y: (resize_and_rescale(x), y))
val_ds = val_ds.map(lambda x, y: (resize_and_rescale(x), y))
test_ds = test_ds.map(lambda x, y: (resize_and_rescale(x), y))
```

Dataset Caching and Prefetching

To optimize the performance and reduce latency during training, we use caching, shuffling, and prefetching. These techniques allow us to preprocess the data while the model is training. AUTOTUNE is used to adjust the prefetching buffer size automatically to improve efficiency.

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

IT21251900 - Data Augmentation

In this section, we implement data augmentation, which is a technique used to artificially increase the diversity of the training dataset by applying random transformations. This helps prevent overfitting and improves the generalization capability of the model.

Data Augmentation Layer

We define a Sequential model for data augmentation using TensorFlow's Keras layers. The augmentation consists of

RandomFlip: Randomly flipping the image both horizontally and vertically, which introduces variations in the image orientation.

RandomRotation: Applying random rotations (up to 20% of the image) to further augment the dataset.

Applying Augmentation to the Training Dataset:

The data augmentation layer is applied only during training (not during validation or testing) using the training=True flag. The map function is used to apply augmentation to each image in the training dataset, maintaining their corresponding labels.

```
import tensorflow as tf

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

train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

IT21251900 - Model Architecture

This section defines a custom Convolutional Neural Network (CNN) model using the Sequential API. The architecture is built for image classification tasks and includes multiple convolutional layers for feature extraction, followed by dense layers for classification.

Convolutional and MaxPooling Layers

First Conv2D Layer

A 2D convolution layer with 32 filters, each of size 3x3. The ReLU activation function is used to introduce non-linearity. The input shape is set to (224, 224, 3), corresponding to RGB images of size 224x224. A MaxPooling layer with a 2x2 pool size is added to downsample the image.

Second Conv2D Layer

Another Conv2D layer with 64 filters and 3x3 kernels, followed by MaxPooling for further downsampling.

Third Conv2D Layer

A Conv2D layer with 128 filters, followed by MaxPooling to capture more complex features.

Fourth Conv2D Layer

A Conv2D layer with 256 filters, with a MaxPooling layer to continue reducing the spatial dimensions.

Fifth Conv2D Layer

A final Conv2D layer with 512 filters and MaxPooling. This deeper layer captures even more complex features of the input images. python

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model_2 = Sequential()

model_2.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3))) # Changed input_shape to (224, 224, 3)
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(64, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(128, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

# Adjusted the architecture to reduce output volume before flattening
model_2.add(Conv2D(256, (3, 3), activation='relu'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))

model_2.add(Conv2D(512, (3, 3), activation='relu')) # Added another Conv2D layer
model_2.add(MaxPooling2D(pool_size=(2, 2))) # Added another MaxPooling2D layer


model_2.add(Flatten())
# This dense layer's input shape is adjusted automatically based on the preceding layer
model_2.add(Dense(256, activation='relu'))
model_2.add(Dropout(0.5))
num_classes = 10  # Replace 10 with the actual number of classes in your dataset
model_2.add(Dense(num_classes, activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
from tensorflow.keras.losses import SparseCategoricalCrossentropy

model_2.compile(optimizer='adam', loss=SparseCategoricalCrossentropy(), metrics=['accuracy'])


model_2.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
conv2d_3 (Conv2D)	(None, 24, 24, 256)	295,168
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 256)	0
conv2d_4 (Conv2D)	(None, 10, 10, 512)	1,180,160
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 512)	0
flatten (Flatten)	(None, 12800)	0
dense (Dense)	(None, 256)	3,277,056
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

Total params: 4,848,202 (18.49 MB)
Trainable params: 4,848,202 (18.49 MB)
Non-trainable params: 0 (0.00 B)

```
history = model_2.fit(
    train_ds,
    validation_data=val_ds,
    batch_size=BATCH_SIZE,
    verbose=1,
    epochs=EPOCHS
)
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 287ms/step - accuracy: 0.9943 - loss: 0.0219 - val_accuracy: 1.0000 - val_loss: 0.0032
Epoch 42/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 289ms/step - accuracy: 0.9934 - loss: 0.0248 - val_accuracy: 0.9948 - val_loss: 0.0105
Epoch 43/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 306ms/step - accuracy: 0.9859 - loss: 0.0413 - val_accuracy: 0.9896 - val_loss: 0.0235
Epoch 44/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 307ms/step - accuracy: 0.9812 - loss: 0.0545 - val_accuracy: 0.9948 - val_loss: 0.0124
Epoch 45/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 17s 304ms/step - accuracy: 0.9904 - loss: 0.0263 - val_accuracy: 0.9896 - val_loss: 0.0250
Epoch 46/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 296ms/step - accuracy: 0.9870 - loss: 0.0259 - val_accuracy: 0.9896 - val_loss: 0.0175
Epoch 47/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 302ms/step - accuracy: 0.9970 - loss: 0.0095 - val_accuracy: 0.9844 - val_loss: 0.0425
Epoch 48/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 306ms/step - accuracy: 0.9923 - loss: 0.0229 - val_accuracy: 0.9948 - val_loss: 0.0212
Epoch 49/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 298ms/step - accuracy: 0.9925 - loss: 0.0142 - val_accuracy: 1.0000 - val_loss: 0.0043
Epoch 50/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 16s 298ms/step - accuracy: 0.9991 - loss: 0.0042 - val_accuracy: 0.9948 - val_loss: 0.0070
```

```
scores_2 = model_2.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 7s 28ms/step - accuracy: 0.9936 - loss: 0.0515
```

```
scores_2
```

```
[0.03048420511186123, 0.99609375]
```

```
history
```

```
<keras.src.callbacks.history.History at 0x7f5e00b7df60>
```

```
history.history.keys()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
type(history.history['loss'])
```

```
list
```

```
len(history.history['loss'])
```

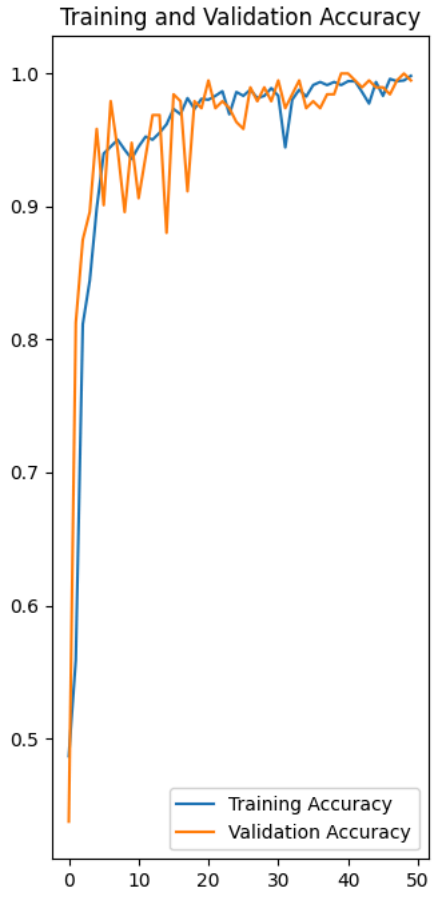
```
50
```

```
history.history['loss'][:5]
```

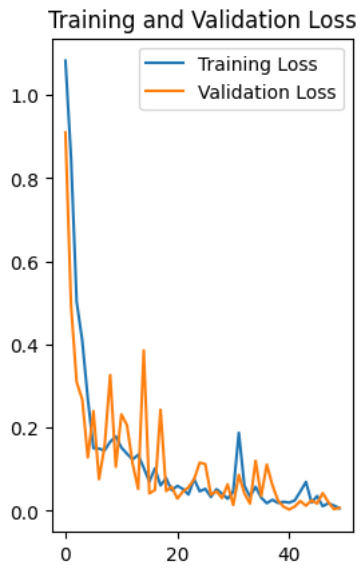
```
[1.0821800231933594,
 0.8482544422149658,
 0.5029866695404053,
 0.4097033143043518,
 0.2691083550453186]
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']


plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
```

```
Text(0.5, 1.0, 'Training and Validation Accuracy')
```

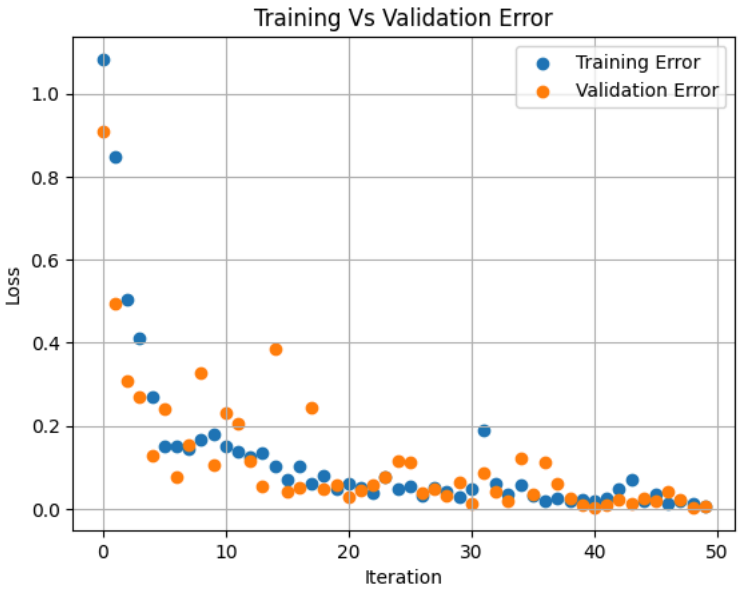



```
plt.subplot(1, 2, 2)
plt.plot(range(len(loss)), loss, label='Training Loss')
plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```




```
plt.scatter(x=history.epoch,y=history.history['loss'],label='Training Error')
plt.scatter(x=history.epoch,y=history.history['val_loss'],label='Validation Error')
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Vs Validation Error')
plt.legend()
```

```
plt.show()
```

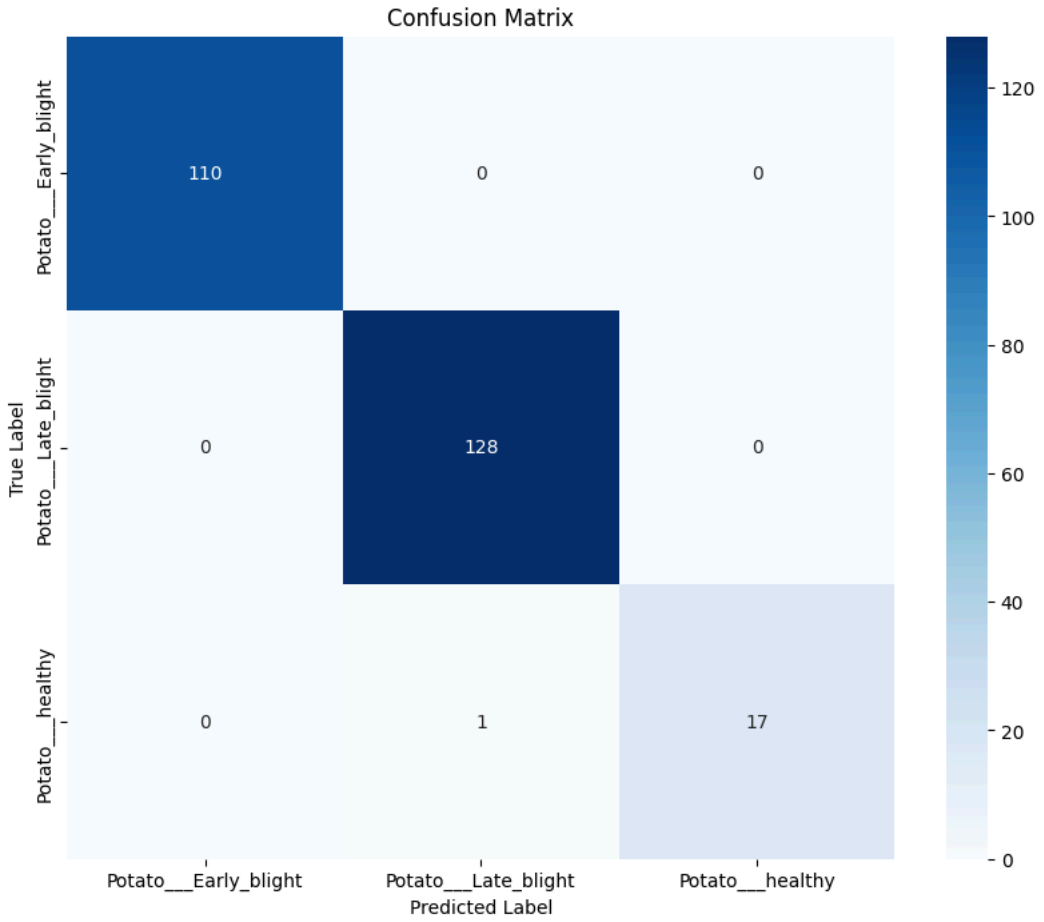



```
import tensorflow as tf
from tensorflow.keras import metrics

model_2.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        metrics.Precision(name='precision'),
        metrics.Recall(name='recall')
    ]
)
```

Calculates the confusion matrix based on the true and predicted labels.The code plots the confusion matrix using Seaborn's heatmap, displaying the counts of true positives, false positives, and false negatives for each class. The resulting heatmap provides a visual representation of the model's classification performance, with labeled axes for easy interpretation.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np

# Step 1: Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model_2.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)

    y_true.extend(labels.numpy())
    y_pred.extend(predicted_classes)

# Step 2: Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Step 3: Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
```



IT21178054 - Model 03

In this section, we are preparing our dataset before feeding it into the neural network. We need to ensure that all images have a consistent size and their pixel values are normalized for better model performance.

Resizing and Rescaling Layer

We use the Sequential API to create a preprocessing pipeline that resizes images to 224x224 pixels (the input size expected by VGG16) and normalizes the pixel values to a range between 0 and 1 by dividing them by 255.

```
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential # Import the Sequential object

resize_and_rescale = Sequential([
    layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),  # Resize to 224x224
    layers.Rescaling(1./255),   # Normalize pixel values between 0 and 1
])
```

```
train_ds = train_ds.map(lambda x, y: (resize_and_rescale(x), y))
val_ds = val_ds.map(lambda x, y: (resize_and_rescale(x), y))
test_ds = test_ds.map(lambda x, y: (resize_and_rescale(x), y))
```

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

```
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
for image_batch, label_batch in train_ds.take(1):
    print(f"Image batch shape: {image_batch.shape}")
    print(f"Label batch shape: {label_batch.shape}")
```

```
Image batch shape: (32, 256, 256, 3)
Label batch shape: (32,)
```

IT21178054 - Data Augmentation

In this section, we implement data augmentation, which is a technique used to artificially increase the diversity of the training dataset by applying random transformations. This helps prevent overfitting and improves the generalization capability of the model.

Data Augmentation Layer

We define a Sequential model for data augmentation using TensorFlow's Keras layers. The augmentation consists of

RandomFlip: Randomly flipping the image both horizontally and vertically, which introduces variations in the image orientation.

RandomRotation: Applying random rotations (up to 20% of the image) to further augment the dataset.

Applying Augmentation to the Training Dataset:

The data augmentation layer is applied only during training (not during validation or testing) using the training=True flag. The map function is used to apply augmentation to each image in the training dataset, maintaining their corresponding labels.

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

train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

IT21178054 - Model Architecture

Model Initialization

Initialize a Sequential model to stack layers for the CNN architecture.

First Convolutional Layer

Add a 2D convolution layer with 32 filters of size 3x3. The ReLU activation function introduces non-linearity. The input shape is set to (256, 256, 3), corresponding to RGB images of size 256x256.

Batch Normalization

Normalize the output of the previous layer to stabilize and accelerate training.

Second Convolutional Layer

Add another convolutional layer with 64 filters, followed by batch normalization and max pooling.

Third Convolutional Layer

Add a convolutional layer with 128 filters, followed by batch normalization and max pooling.

Fourth Convolutional Layer

Add a convolutional layer with 256 filters, followed by batch normalization and max pooling.

```
from tensorflow.keras import models, layers
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Dense, Dropout, BatchNormalization, Flatten
from tensorflow.keras.models import Sequential

model_3 = Sequential()

model_3.add(Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)))
```

```
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(64, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(128, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Conv2D(256, (3, 3), activation='relu'))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D(pool_size=(2, 2)))

model_3.add(Flatten())
# Fully Connected Layer with Dropout for regularization
model_3.add(Dense(128, activation='relu'))
model_3.add(Dropout(0.5))

# Output Layer with Softmax Activation
model_3.add(Dense(len(class_names), activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model_3.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                metrics=['accuracy'])
```

```
model_3.summary()  # Display the model architecture
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 254, 254, 32)	896
batch_normalization (BatchNormalization)	(None, 254, 254, 32)	128
max_pooling2d (MaxPooling2D)	(None, 127, 127, 32)	0
conv2d_1 (Conv2D)	(None, 125, 125, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 125, 125, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 62, 62, 64)	0
conv2d_2 (Conv2D)	(None, 60, 60, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 60, 60, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 30, 30, 128)	0
conv2d_3 (Conv2D)	(None, 28, 28, 256)	295,168
batch_normalization_3 (BatchNormalization)	(None, 28, 28, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 256)	0
flatten (Flatten)	(None, 50176)	0
dense (Dense)	(None, 128)	6,422,656
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 3)	387

Total params: 6,813,379 (25.99 MB)
Trainable params: 6,812,419 (25.99 MB)
Non-trainable params: 960 (3.75 KB)

```
history = model_3.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=EPOCHS,
)
```

```
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 352ms/step - accuracy: 0.9514 - loss: 0.1782 - val_accuracy: 0.6510 - val_loss: 3.9134
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 359ms/step - accuracy: 0.9634 - loss: 0.1230 - val_accuracy: 0.6146 - val_loss: 5.8954
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 361ms/step - accuracy: 0.9430 - loss: 0.2192 - val_accuracy: 0.8906 - val_loss: 0.6860
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 347ms/step - accuracy: 0.9420 - loss: 0.1821 - val_accuracy: 0.4792 - val_loss: 30.7310
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 358ms/step - accuracy: 0.9606 - loss: 0.1245 - val_accuracy: 0.4896 - val_loss: 27.4846
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 347ms/step - accuracy: 0.9555 - loss: 0.1299 - val_accuracy: 0.9271 - val_loss: 0.2916
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 348ms/step - accuracy: 0.9656 - loss: 0.1744 - val_accuracy: 0.8385 - val_loss: 2.2625
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 367ms/step - accuracy: 0.9691 - loss: 0.1232 - val_accuracy: 0.4844 - val_loss: 52.6334
Epoch 30/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 348ms/step - accuracy: 0.9553 - loss: 0.1797 - val_accuracy: 0.6042 - val_loss: 12.3287
Epoch 31/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 343ms/step - accuracy: 0.9421 - loss: 0.1761 - val_accuracy: 0.9219 - val_loss: 0.6259
Epoch 32/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 356ms/step - accuracy: 0.9466 - loss: 0.1998 - val_accuracy: 0.5990 - val_loss: 19.0535
Epoch 33/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 18s 341ms/step - accuracy: 0.9455 - loss: 0.1889 - val_accuracy: 0.5312 - val_loss: 22.6120
Epoch 34/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 345ms/step - accuracy: 0.9552 - loss: 0.1578 - val_accuracy: 0.4792 - val_loss: 89.0650
Epoch 35/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 375ms/step - accuracy: 0.9629 - loss: 0.1119 - val_accuracy: 0.7448 - val_loss: 3.2486
Epoch 36/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 358ms/step - accuracy: 0.9611 - loss: 0.1259 - val_accuracy: 0.9167 - val_loss: 0.3033
Epoch 37/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 363ms/step - accuracy: 0.9761 - loss: 0.1003 - val_accuracy: 0.9479 - val_loss: 0.2291
Epoch 38/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 18s 340ms/step - accuracy: 0.9621 - loss: 0.1245 - val_accuracy: 0.8333 - val_loss: 1.2297
Epoch 39/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 346ms/step - accuracy: 0.9863 - loss: 0.0475 - val_accuracy: 0.9740 - val_loss: 0.0644
Epoch 40/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 363ms/step - accuracy: 0.9676 - loss: 0.1410 - val_accuracy: 0.8594 - val_loss: 1.2971
Epoch 41/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 343ms/step - accuracy: 0.9727 - loss: 0.1070 - val_accuracy: 0.8333 - val_loss: 1.3208
Epoch 42/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 351ms/step - accuracy: 0.9635 - loss: 0.1511 - val_accuracy: 0.5469 - val_loss: 9.1315
Epoch 43/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 370ms/step - accuracy: 0.9715 - loss: 0.1025 - val_accuracy: 0.6771 - val_loss: 2.6289
Epoch 44/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 18s 341ms/step - accuracy: 0.9738 - loss: 0.0909 - val_accuracy: 0.8385 - val_loss: 1.0473
Epoch 45/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 381ms/step - accuracy: 0.9712 - loss: 0.1054 - val_accuracy: 0.9583 - val_loss: 0.1759
Epoch 46/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 40s 366ms/step - accuracy: 0.9735 - loss: 0.0886 - val_accuracy: 0.8802 - val_loss: 0.7358
Epoch 47/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.9692 - loss: 0.1572 - val_accuracy: 0.9167 - val_loss: 0.5542
Epoch 48/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 21s 351ms/step - accuracy: 0.9739 - loss: 0.0996 - val_accuracy: 0.7292 - val_loss: 2.9455
Epoch 49/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 20s 348ms/step - accuracy: 0.9712 - loss: 0.0953 - val_accuracy: 0.8594 - val_loss: 0.6228
Epoch 50/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 19s 345ms/step - accuracy: 0.9694 - loss: 0.1180 - val_accuracy: 0.6354 - val_loss: 12.6011
```

```
test_loss, test_acc = model_3.evaluate(test_ds)
print(f"Test accuracy: {test_acc}")
```

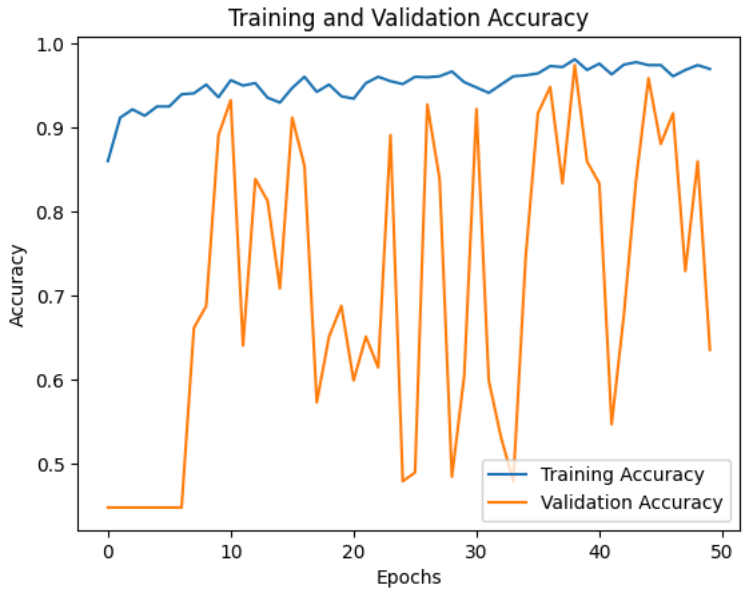
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.7642 - loss: 9.8739
Test accuracy: 0.73828125
```

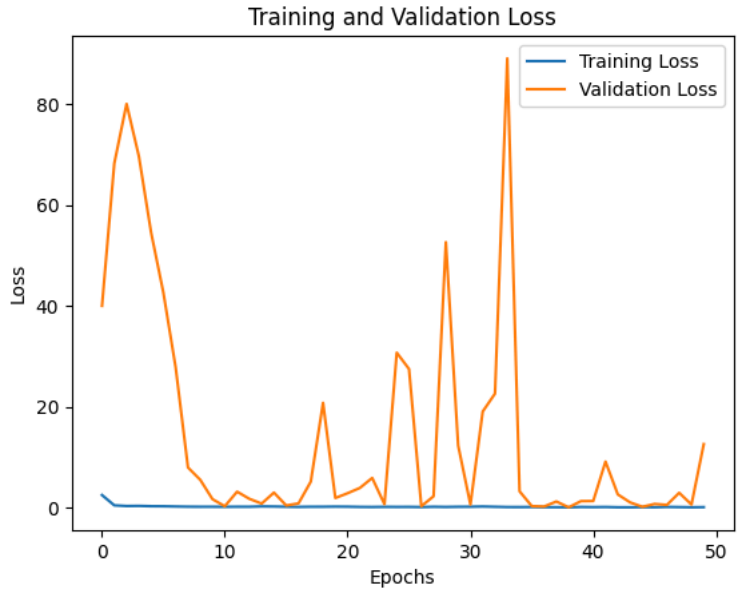
Plotting Training and Validation Accuracy

This code visualizes the training process of the model by plotting the training and validation accuracy as well as the training and validation loss over the epochs. The first plot displays how the training and validation accuracy change, providing insights into the model's performance and generalization ability during training. The second plot illustrates the training and validation loss, which helps in understanding the convergence

of the model and identifying potential overfitting or underfitting. Both plots are essential for assessing the effectiveness of the model training and guiding further adjustments.

```
import matplotlib.pyplot as plt

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.show()

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



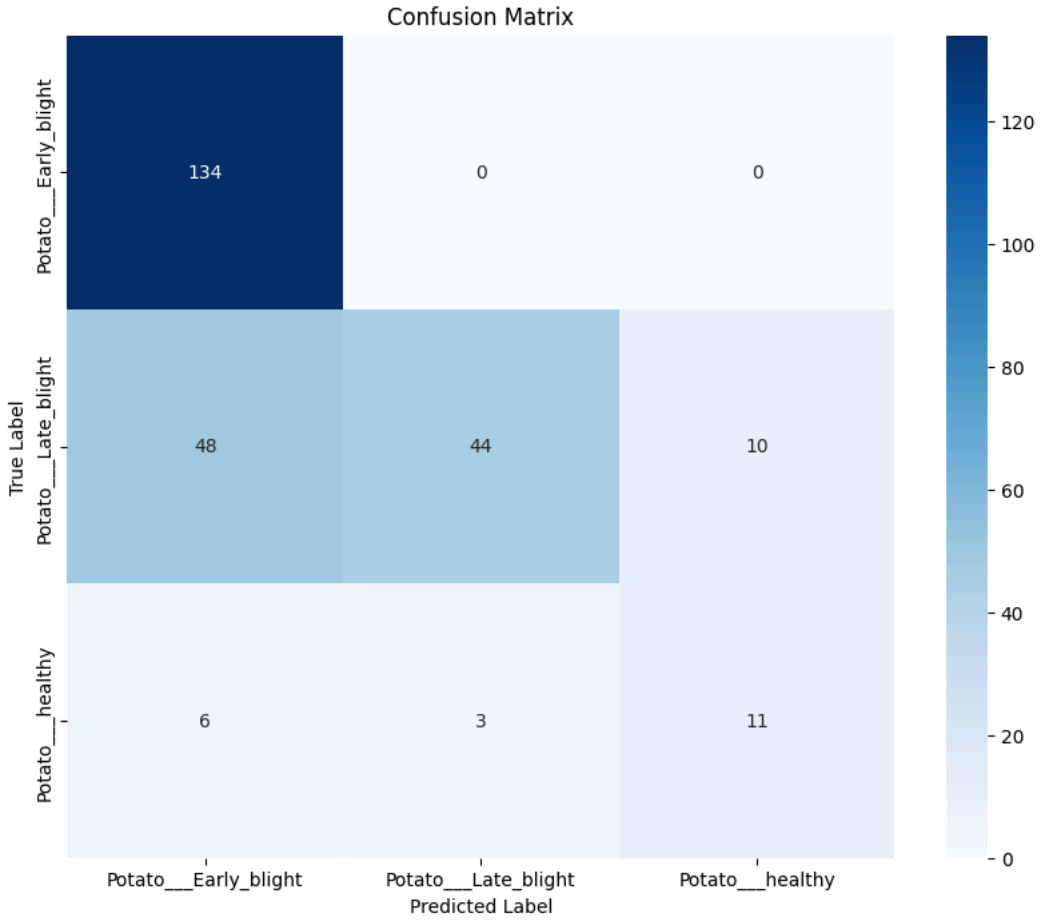


Calculates the confusion matrix based on the true and predicted labels.The code plots the confusion matrix using Seaborn's heatmap, displaying the counts of true positives, false positives, and false negatives for each class. The resulting heatmap provides a visual

representation of the model's classification performance, with labeled axes for easy interpretation.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np

# Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model_3.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)
    # Convert labels to multiclass format if they are in multilabel-indicator format
    true_classes = np.argmax(labels.numpy(), axis=1) if labels.ndim > 1 else labels.numpy()
    y_true.extend(true_classes)
    y_pred.extend(predicted_classes)

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 605ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
```



IT21360428 - Model 04

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
```

IT21360428 - Data Augmentation

In this section, we implement data augmentation, which is a technique used to artificially increase the diversity of the training dataset by applying random transformations. This helps prevent overfitting and improves the generalization capability of the model.

Data Augmentation Layer

We define a Sequential model for data augmentation using TensorFlow's Keras layers. The augmentation consists of

RandomFlip: Randomly flipping the image both horizontally and vertically, which introduces variations in the image orientation.

RandomRotation: Applying random rotations (up to 20% of the image) to further augment the dataset.

Applying Augmentation to the Training Dataset:

The data augmentation layer is applied only during training (not during validation or testing) using the training=True flag. The map function is used to apply augmentation to each image in the training dataset, maintaining their corresponding labels.

```
# Data Augmentation (same as Member 1, you can also modify it if necessary)
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2),
])
```

IT21360428 - Model Architecture

This code defines a convolutional neural network (CNN) using Keras for image classification.

The model consists of four convolutional blocks, each with convolutional, batch normalization, and max pooling layers to extract features from images.

After flattening the feature maps, a fully connected layer with dropout is used for regularization, followed by an output layer with softmax activation for multi-class classification.

The model is designed to preprocess images by resizing and rescaling pixel values to the range [0, 1].

Finally, the model's architecture is summarized to display the layers and parameters.

```
# Define the image size and number of channels
IMAGE_SIZE = 256
CHANNELS = 3

# You need to manually define class names based on your dataset directory
class_names = ['healthy', 'diseased_class1', 'diseased_class2', 'diseased_class3']  # Modify this according to your dataset

# Set the number of classes
n_classes = len(class_names)

# Define input shape
input_shape = (IMAGE_SIZE, IMAGE_SIZE, CHANNELS)

# Define the resize and rescale layers
resize_and_rescale = tf.keras.Sequential([
    tf.keras.layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),
    tf.keras.layers.Rescaling(1./255),
])

# Define the model
model_4 = models.Sequential([
    resize_and_rescale,  # Use the resize and rescale layer

    # First Convolutional Block
```

```
    layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=input_shape),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Second Convolutional Block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Third Convolutional Block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Fourth Convolutional Block
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Flatten the data
    layers.Flatten(),

    # Dense layers with Dropout for regularization
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(n_classes, activation='softmax'),  # Output layer with the correct number of classes
])

# Build and summarize the model
model_4.build(input_shape=(None, IMAGE_SIZE, IMAGE_SIZE, CHANNELS))  # Build the model with the correct input shape
model_4.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
sequential_1 (Sequential)	(None, 256, 256, 3)	0
conv2d (Conv2D)	(None, 256, 256, 32)	896
batch_normalization (BatchNormalization)	(None, 256, 256, 32)	128
max_pooling2d (MaxPooling2D)	(None, 128, 128, 32)	0
conv2d_1 (Conv2D)	(None, 128, 128, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 128, 128, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 64)	0
conv2d_2 (Conv2D)	(None, 64, 64, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 64, 64, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 128)	0
conv2d_3 (Conv2D)	(None, 32, 32, 256)	295,168
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 256)	0
flatten (Flatten)	(None, 65536)	0
dense (Dense)	(None, 512)	33,554,944
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 4)	2,052

Total params: 33,947,332 (129.50 MB)
Trainable params: 33,946,372 (129.50 MB)
Non-trainable params: 960 (3.75 KB)

```
# Compile the model
```

```
# Compile the model
model_4.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

```
# Train the model
history_4 = model_4.fit(
    train_ds,
    epochs=EPOCHS,
    validation_data=val_ds,
    verbose=1
)
```

```
Epoch 22/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9899 - loss: 0.2314 - val_accuracy: 0.5573 - val_loss: 48.1991
Epoch 23/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9954 - loss: 0.0331 - val_accuracy: 0.9167 - val_loss: 2.4063
Epoch 24/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9867 - loss: 0.2074 - val_accuracy: 0.9792 - val_loss: 0.7489
Epoch 25/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9805 - loss: 0.3205 - val_accuracy: 0.9844 - val_loss: 0.0877
Epoch 26/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9942 - loss: 0.0594 - val_accuracy: 0.8802 - val_loss: 3.7508
Epoch 27/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9823 - loss: 0.2125 - val_accuracy: 0.8594 - val_loss: 17.5817
Epoch 28/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9878 - loss: 0.2666 - val_accuracy: 0.9115 - val_loss: 2.6445
Epoch 29/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9722 - loss: 0.5200 - val_accuracy: 0.5365 - val_loss: 38.9405
Epoch 30/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9820 - loss: 0.3599 - val_accuracy: 0.9792 - val_loss: 0.1090
Epoch 31/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9754 - loss: 0.6550 - val_accuracy: 0.9219 - val_loss: 2.5198
Epoch 32/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9866 - loss: 0.1283 - val_accuracy: 0.9688 - val_loss: 0.9144
Epoch 33/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9942 - loss: 0.1021 - val_accuracy: 0.9740 - val_loss: 0.3248
Epoch 34/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9938 - loss: 0.1171 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 35/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9943 - loss: 0.1914 - val_accuracy: 0.9323 - val_loss: 3.3029
Epoch 36/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9912 - loss: 0.0796 - val_accuracy: 0.8333 - val_loss: 12.0867
Epoch 37/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9983 - loss: 0.0222 - val_accuracy: 1.0000 - val_loss: 6.6104e-05
Epoch 38/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9960 - loss: 0.1178 - val_accuracy: 0.9948 - val_loss: 0.0295
Epoch 39/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9939 - loss: 0.0916 - val_accuracy: 0.9740 - val_loss: 0.5488
Epoch 40/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9882 - loss: 0.1567 - val_accuracy: 0.9115 - val_loss: 5.0787
Epoch 41/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9870 - loss: 0.1437 - val_accuracy: 0.9844 - val_loss: 0.2586
Epoch 42/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 83ms/step - accuracy: 0.9902 - loss: 0.0906 - val_accuracy: 0.9896 - val_loss: 0.2727
Epoch 43/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9924 - loss: 0.0569 - val_accuracy: 0.9792 - val_loss: 0.0972
Epoch 44/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.9966 - loss: 0.0221 - val_accuracy: 0.9167 - val_loss: 4.1174
Epoch 45/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9945 - loss: 0.0476 - val_accuracy: 0.9688 - val_loss: 2.1822
Epoch 46/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9977 - loss: 0.0240 - val_accuracy: 0.7865 - val_loss: 31.3187
Epoch 47/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9930 - loss: 0.0556 - val_accuracy: 0.9219 - val_loss: 9.1026
Epoch 48/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.9925 - loss: 0.4436 - val_accuracy: 0.9271 - val_loss: 6.5865
Epoch 49/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9922 - loss: 0.1184 - val_accuracy: 0.9062 - val_loss: 8.3281
Epoch 50/50
54/54 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.9806 - loss: 0.4913 - val_accuracy: 0.9583 - val_loss: 2.0986
```

```
scores = model_4.evaluate(test_ds)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 6s 28ms/step - accuracy: 0.9733 - loss: 0.9552
```

```
scores
```

```
[1.4976621866226196, 0.96875]
```

```
scores
```

```
[1.4976621866226196, 0.96875]
```

```
history_4.params
```

```
{'verbose': 1, 'epochs': 50, 'steps': 54}
```

```
history_4.history.keys()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
type(history_4.history['loss'])
```

```
list
```

```
len(history_4.history['loss'])
```

```
50
```

```
history_4.history['loss'][:5]
```

```
[7.999459743499756,
 2.1771011352539062,
 0.8500080108642578,
 0.548255443572998,
 0.88930743932724]
```

```
acc = history_4.history['accuracy']
val_acc = history_4.history['val_accuracy']

loss = history_4.history['loss']
val_loss = history_4.history['val_loss']
```

Plotting Training and Validation Accuracy

The first subplot displays the training accuracy and validation accuracy over epochs. The x-axis represents the number of epochs, while the y-