

[Open in Colab](#)

## Module Name: SE4050 - Deep Learning Assignment

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

→ Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount)
```

### Group member 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

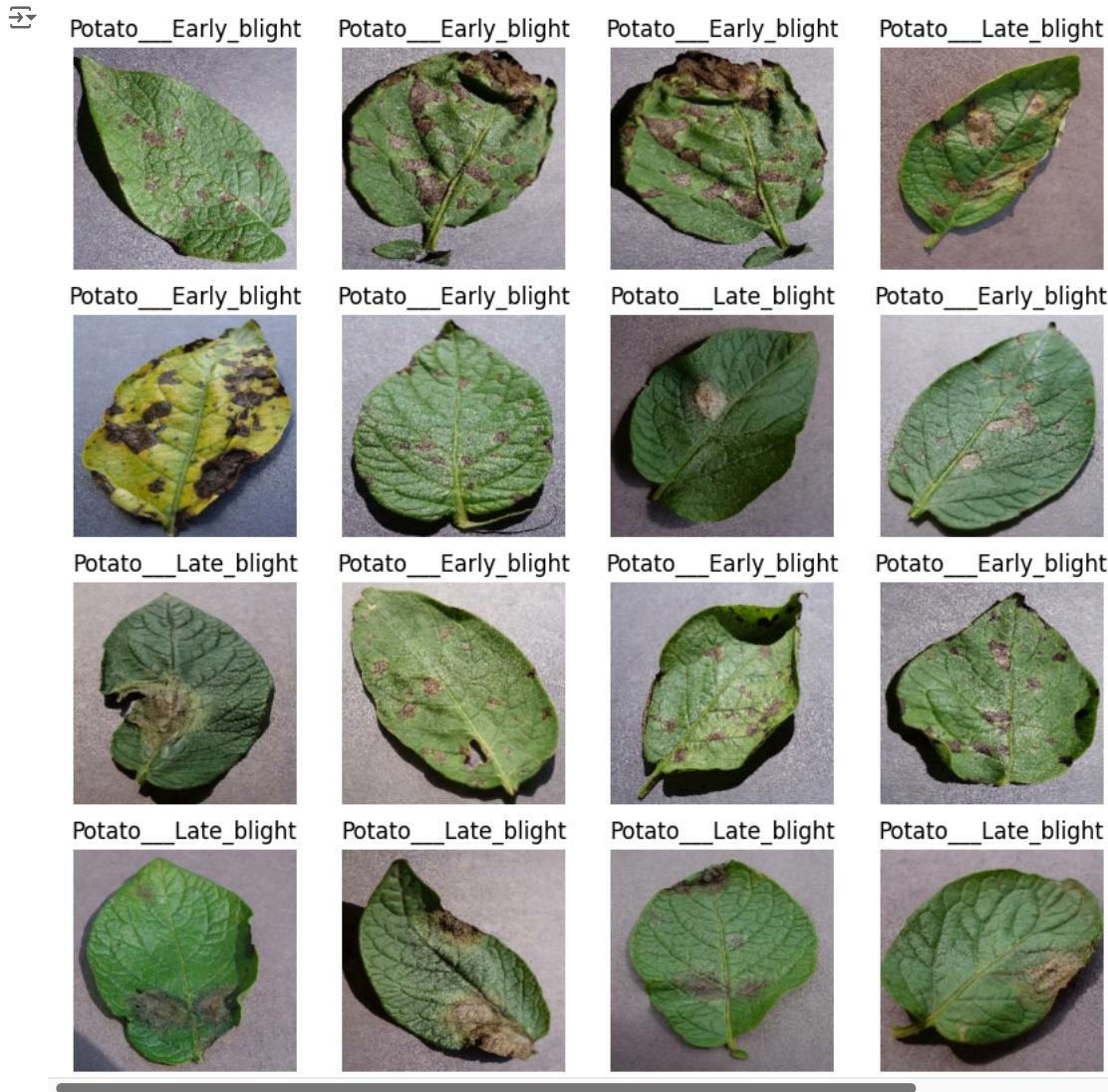
```
import os
import tensorflow as tf

# Step 1: Unzip the file from Google Drive to a local directory
zin_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 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 0 1 1 2 0 0]
```

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
→ 54.40000000000006

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
→ 6.80000000000001

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

```

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

→ 8

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

```

Cache, Shuffle, and Prefetch the Dataset

```

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

```

## ▼ IT21302862 - Model 01

**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.

```

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

```
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.

```
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)

→ /usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `in 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 KB)
```

### 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']
)

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

```

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  
Epoch 30/50  
**54/54** 20s 344ms/step - accuracy: 0.9851 - loss: 0.0435 - val\_accuracy: 0.9948 - val\_loss: 0.0207  
Epoch 31/50  
**54/54** 18s 338ms/step - accuracy: 0.9842 - loss: 0.0400 - val\_accuracy: 0.9740 - val\_loss: 0.0658  
Epoch 32/50  
**54/54** 20s 365ms/step - accuracy: 0.9938 - loss: 0.0219 - val\_accuracy: 0.9635 - val\_loss: 0.0772  
Epoch 33/50  
**54/54** 19s 344ms/step - accuracy: 0.9759 - loss: 0.0719 - val\_accuracy: 0.9635 - val\_loss: 0.1009  
Epoch 34/50  
**54/54** 19s 344ms/step - accuracy: 0.9880 - loss: 0.0380 - val\_accuracy: 0.9531 - val\_loss: 0.1779  
Epoch 35/50  
**54/54** 20s 366ms/step - accuracy: 0.9855 - loss: 0.0411 - val\_accuracy: 0.9531 - val\_loss: 0.1234  
Epoch 36/50  
**54/54** 18s 342ms/step - accuracy: 0.9845 - loss: 0.0438 - val\_accuracy: 0.9427 - val\_loss: 0.1386  
Epoch 37/50  
**54/54** 19s 343ms/step - accuracy: 0.9917 - loss: 0.0215 - val\_accuracy: 0.9688 - val\_loss: 0.0929  
Epoch 38/50  
**54/54** 20s 365ms/step - accuracy: 0.9903 - loss: 0.0307 - val\_accuracy: 1.0000 - val\_loss: 0.0102  
Epoch 39/50  
**54/54** 19s 344ms/step - accuracy: 0.9900 - loss: 0.0307 - val\_accuracy: 0.9583 - val\_loss: 0.0923  
Epoch 40/50  
**54/54** 19s 345ms/step - accuracy: 0.9518 - loss: 0.1248 - val\_accuracy: 0.9375 - val\_loss: 0.1637  
Epoch 41/50  
**54/54** 20s 365ms/step - accuracy: 0.9826 - loss: 0.0415 - val\_accuracy: 0.9896 - val\_loss: 0.0251  
Epoch 42/50  
**54/54** 19s 348ms/step - accuracy: 0.9937 - loss: 0.0270 - val\_accuracy: 0.9844 - val\_loss: 0.0577  
Epoch 43/50  
**54/54** 20s 362ms/step - accuracy: 0.9874 - loss: 0.0327 - val\_accuracy: 0.9688 - val\_loss: 0.1245  
Epoch 44/50  
**54/54** 20s 352ms/step - accuracy: 0.9911 - loss: 0.0248 - val\_accuracy: 0.9948 - val\_loss: 0.0211  
Epoch 45/50  
**54/54** 19s 349ms/step - accuracy: 0.9888 - loss: 0.0396 - val\_accuracy: 0.8906 - val\_loss: 0.4121  
Epoch 46/50  
**54/54** 21s 348ms/step - accuracy: 0.9771 - loss: 0.0712 - val\_accuracy: 0.9635 - val\_loss: 0.1299  
Epoch 47/50  
**54/54** 19s 343ms/step - accuracy: 0.9810 - loss: 0.0555 - val\_accuracy: 0.9792 - val\_loss: 0.0724  
Epoch 48/50  
**54/54** 20s 369ms/step - accuracy: 0.9898 - loss: 0.0221 - val\_accuracy: 0.9844 - val\_loss: 0.0366  
Epoch 49/50  
**54/54** 19s 344ms/step - accuracy: 0.9955 - loss: 0.0188 - val\_accuracy: 0.9688 - val\_loss: 0.0824  
Epoch 50/50  
**54/54** 19s 344ms/step - accuracy: 0.9822 - loss: 0.0391 - val\_accuracy: 0.9635 - val\_loss: 0.1113

```
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

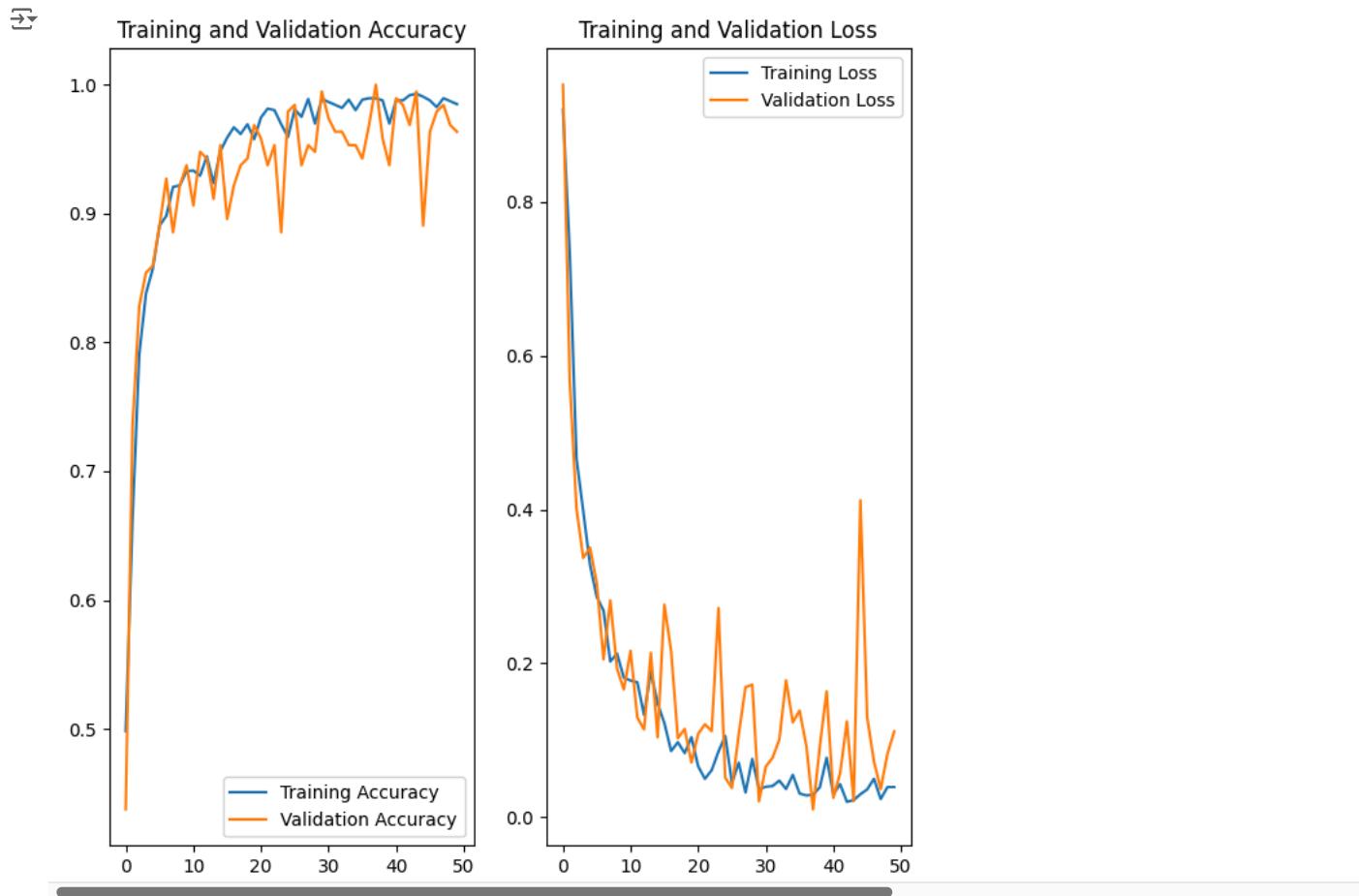
history.history['loss'][:5]
→ [0.9199034571647644,
  0.7345544099807739,
  0.4664019048213959,
  0.3981457054615021,
  0.3280259966850281]

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

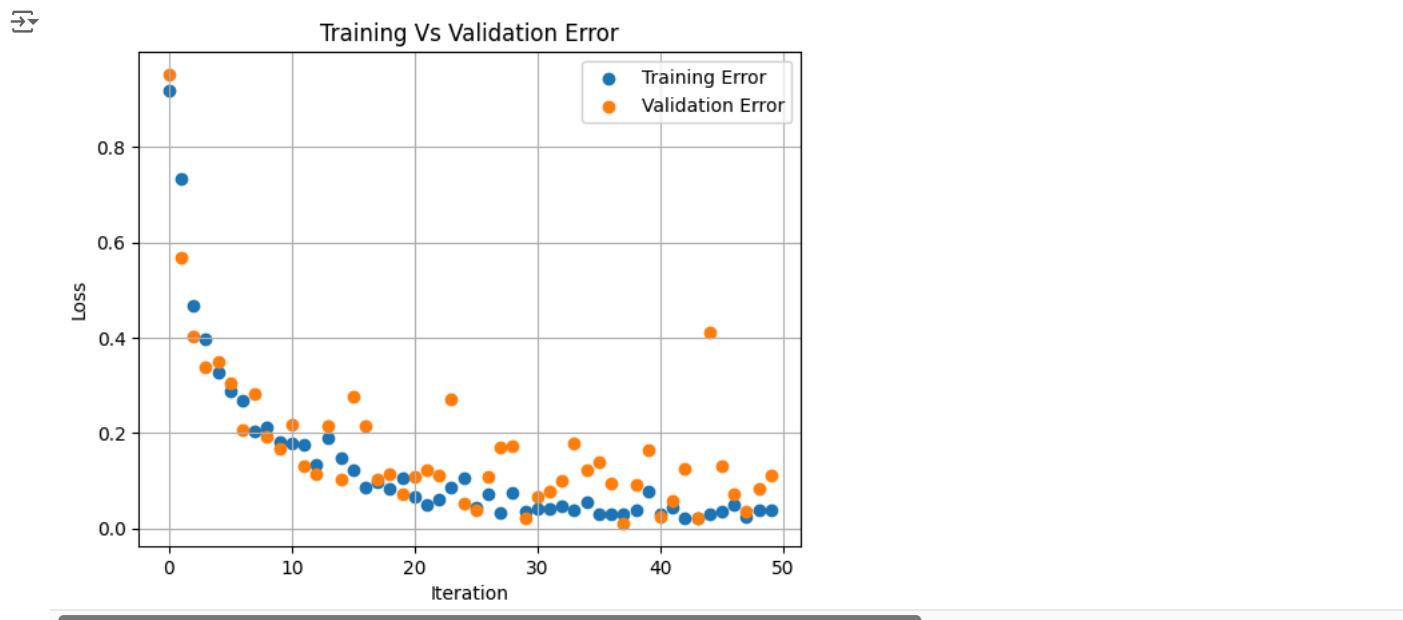
loss = history.history['loss']
val_loss = history.history['val_loss']

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()
```



```
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

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

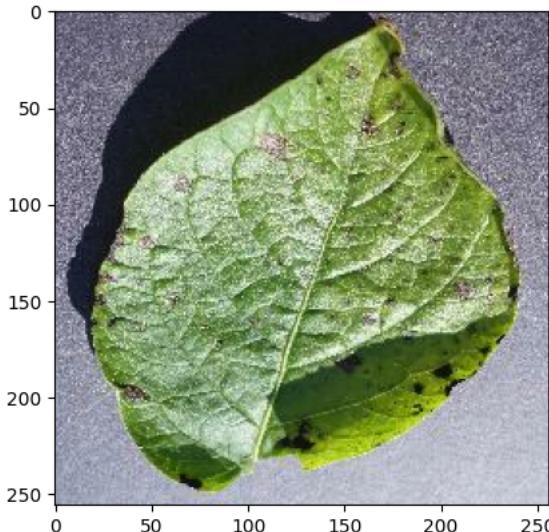
    first_image = images_batch[0].numpy().astype('uint8')
    first_label = labels_batch[0].numpy()

    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

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: 98.73%



Actual: Potato\_\_Late\_blight,  
Predicted: Potato\_\_Late\_blight.  
Confidence: 81.51%



Actual: Potato\_\_Late\_blight,  
Predicted: Potato\_\_Late\_blight.  
Confidence: 100.0%



Actual: Potato\_\_Late\_blight,  
Predicted: Potato\_\_Late\_blight.  
Confidence: 99.79%



Actual: Potato\_\_Early\_blight,  
Predicted: Potato\_\_Early\_blight.  
Confidence: 100.0%

Actual: Potato\_\_Early\_blight,  
Predicted: Potato\_\_Early\_blight.  
Confidence: 100.0%



Actual: Potato\_\_Early\_blight,  
Predicted: Potato\_\_Early\_blight.  
Confidence: 76.04%



Actual: Potato\_\_Late\_blight,  
Predicted: Potato\_\_Late\_blight.  
Confidence: 99.38%



## 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')
    ]
)
```

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

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

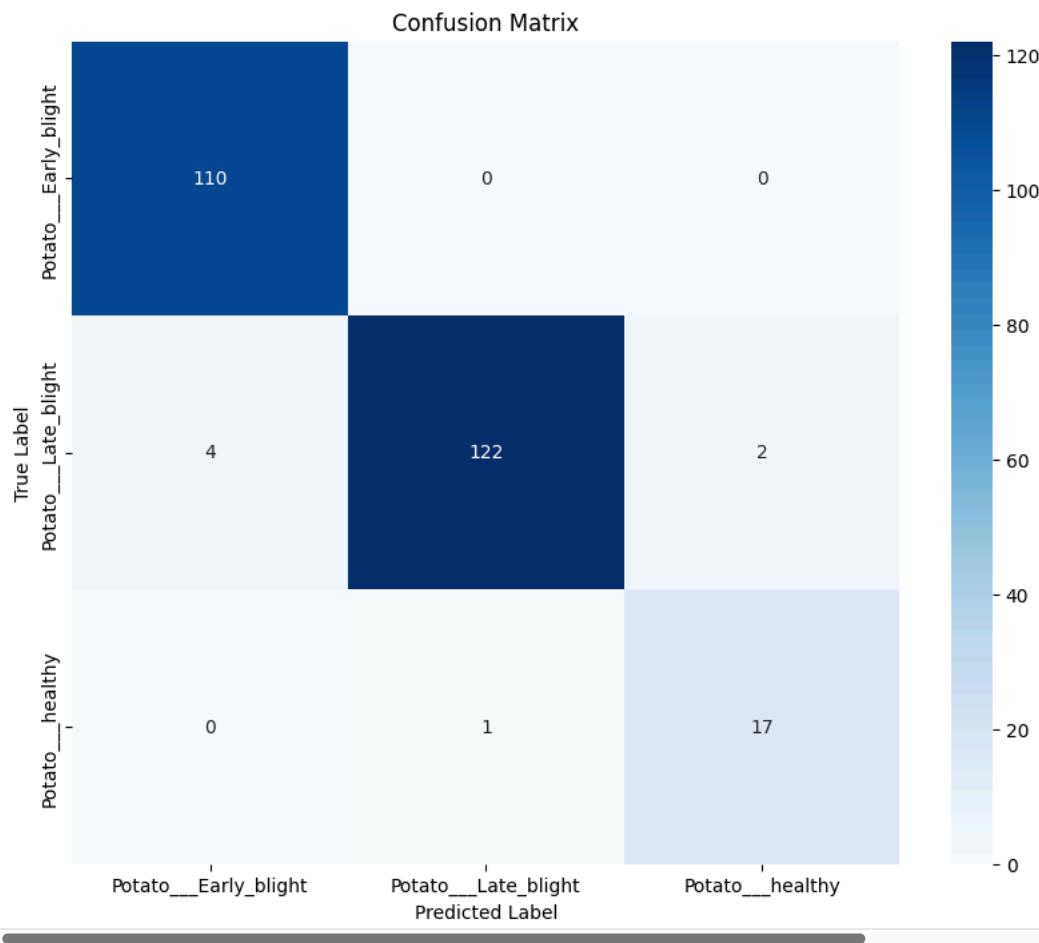
for images, labels in test_ds:
    predictions = model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)

    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 `in 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
)

```

```

→ Epoch 1/50
54/54 ━━━━━━━━━━ 182s 442ms/step - accuracy: 0.4721 - loss: 1.2800 - val_accuracy: 0.4375 - val_loss: 0.9095
Epoch 2/50
54/54 ━━━━━━━━ 37s 286ms/step - accuracy: 0.4667 - loss: 0.9569 - val_accuracy: 0.8125 - val_loss: 0.4936
Epoch 3/50
54/54 ━━━━━━ 16s 286ms/step - accuracy: 0.7741 - loss: 0.5839 - val_accuracy: 0.8750 - val_loss: 0.3095
Epoch 4/50
54/54 ━━━━ 16s 290ms/step - accuracy: 0.8485 - loss: 0.3782 - val_accuracy: 0.8958 - val_loss: 0.2681
Epoch 5/50
54/54 ━━━━ 15s 284ms/step - accuracy: 0.8804 - loss: 0.3265 - val_accuracy: 0.9583 - val_loss: 0.1284
Epoch 6/50

```

```
54/54 ━━━━━━━━━━ 15s 282ms/step - accuracy: 0.9342 - loss: 0.1707 - val_accuracy: 0.9010 - val_loss: 0.2397
Epoch 7/50
54/54 ━━━━━━━━━━ 15s 282ms/step - accuracy: 0.9349 - loss: 0.1730 - val_accuracy: 0.9792 - val_loss: 0.0761
Epoch 8/50
54/54 ━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9541 - loss: 0.1423 - val_accuracy: 0.9427 - val_loss: 0.1551
Epoch 9/50
54/54 ━━━━━━━━━━ 16s 287ms/step - accuracy: 0.9542 - loss: 0.1310 - val_accuracy: 0.8958 - val_loss: 0.3265
Epoch 10/50
54/54 ━━━━━━━━━━ 15s 282ms/step - accuracy: 0.9464 - loss: 0.1652 - val_accuracy: 0.9479 - val_loss: 0.1059
Epoch 11/50
54/54 ━━━━━━━━━━ 15s 282ms/step - accuracy: 0.9446 - loss: 0.1451 - val_accuracy: 0.9062 - val_loss: 0.2318
Epoch 12/50
54/54 ━━━━━━━━━━ 16s 289ms/step - accuracy: 0.9318 - loss: 0.1916 - val_accuracy: 0.9375 - val_loss: 0.2062
Epoch 13/50
54/54 ━━━━━━━━━━ 20s 275ms/step - accuracy: 0.9250 - loss: 0.2033 - val_accuracy: 0.9688 - val_loss: 0.1138
Epoch 14/50
54/54 ━━━━━━━━━━ 15s 279ms/step - accuracy: 0.9652 - loss: 0.1065 - val_accuracy: 0.9688 - val_loss: 0.0528
Epoch 15/50
54/54 ━━━━━━━━━━ 15s 282ms/step - accuracy: 0.9636 - loss: 0.1027 - val_accuracy: 0.8802 - val_loss: 0.3858
Epoch 16/50
54/54 ━━━━━━━━━━ 15s 281ms/step - accuracy: 0.9733 - loss: 0.0751 - val_accuracy: 0.9844 - val_loss: 0.0421
Epoch 17/50
54/54 ━━━━━━━━━━ 15s 286ms/step - accuracy: 0.9649 - loss: 0.1136 - val_accuracy: 0.9792 - val_loss: 0.0500
Epoch 18/50
54/54 ━━━━━━━━━━ 15s 279ms/step - accuracy: 0.9871 - loss: 0.0361 - val_accuracy: 0.9115 - val_loss: 0.2430
Epoch 19/50
54/54 ━━━━━━━━━━ 21s 286ms/step - accuracy: 0.9664 - loss: 0.0945 - val_accuracy: 0.9792 - val_loss: 0.0477
Epoch 20/50
54/54 ━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9801 - loss: 0.0554 - val_accuracy: 0.9740 - val_loss: 0.0581
Epoch 21/50
54/54 ━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9874 - loss: 0.0437 - val_accuracy: 0.9948 - val_loss: 0.0296
Epoch 22/50
54/54 ━━━━━━━━━━ 16s 286ms/step - accuracy: 0.9799 - loss: 0.0597 - val_accuracy: 0.9740 - val_loss: 0.0461
Epoch 23/50
54/54 ━━━━━━━━━━ 20s 279ms/step - accuracy: 0.9808 - loss: 0.0470 - val_accuracy: 0.9792 - val_loss: 0.0567
Epoch 24/50
54/54 ━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9682 - loss: 0.0766 - val_accuracy: 0.9740 - val_loss: 0.0756
Epoch 25/50
54/54 ━━━━━━━━━━ 21s 292ms/step - accuracy: 0.9828 - loss: 0.0490 - val_accuracy: 0.9635 - val_loss: 0.1160
Epoch 26/50
54/54 ━━━━━━━━━━ 20s 288ms/step - accuracy: 0.9860 - loss: 0.0492 - val_accuracy: 0.9583 - val_loss: 0.1124
Epoch 27/50
54/54 ━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9864 - loss: 0.0384 - val_accuracy: 0.9896 - val_loss: 0.0387
Epoch 28/50
54/54 ━━━━━━━━━━ 15s 284ms/step - accuracy: 0.9848 - loss: 0.0433 - val_accuracy: 0.9792 - val_loss: 0.0473
Epoch 29/50
54/54 ━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9790 - loss: 0.0726 - val_accuracy: 0.9806 - val_loss: 0.0206
54/54 ━━━━━━━━━━ 15s 285ms/step - accuracy: 0.9790 - loss: 0.0726 - val_accuracy: 0.9806 - val_loss: 0.0206
```

```
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]
```

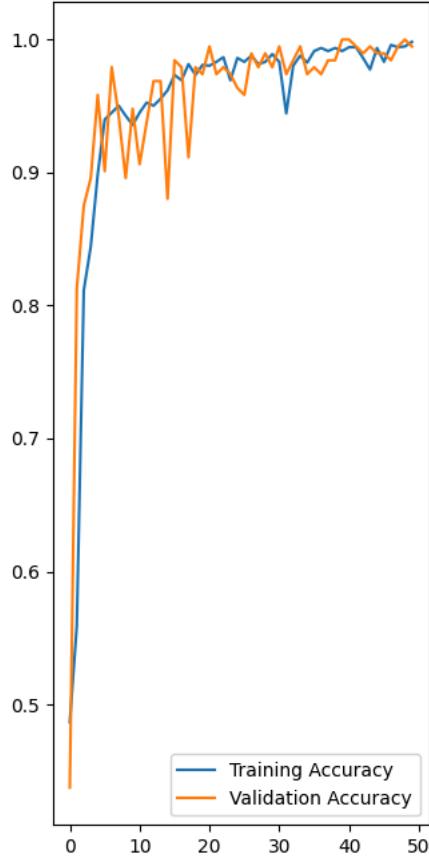
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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

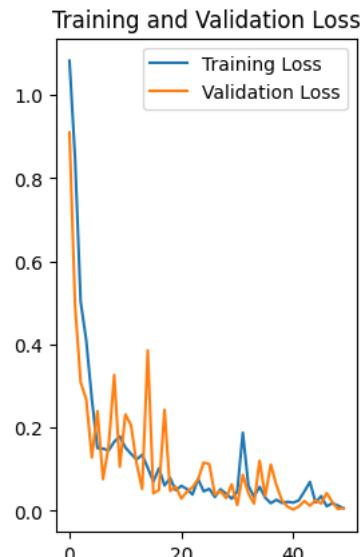
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
```

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

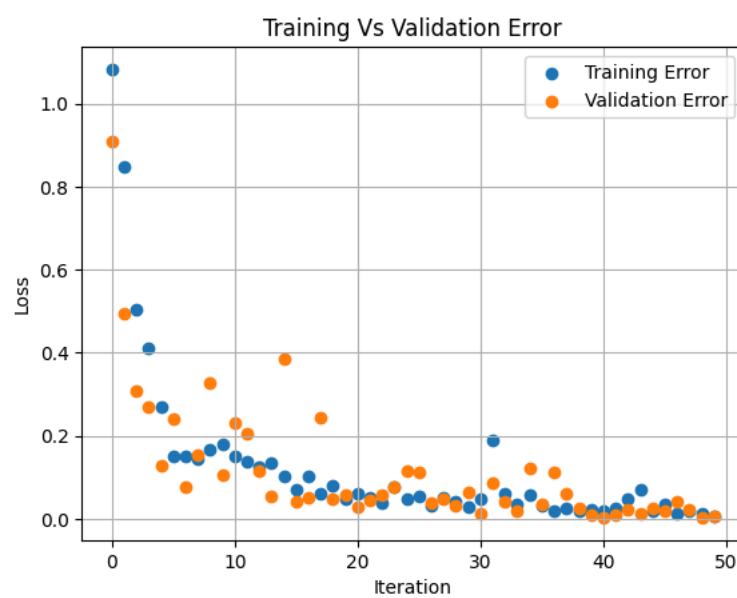
Training and Validation Accuracy



```
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')
    ]
)

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)

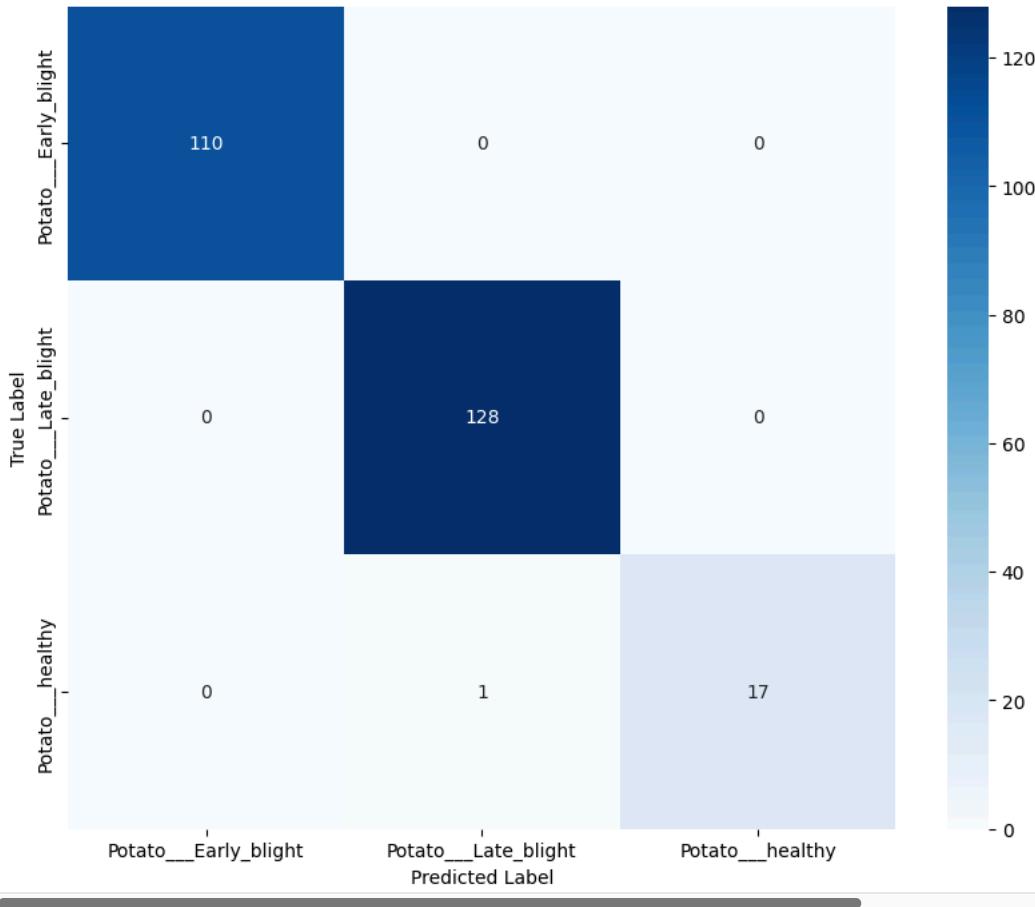
# Step 2: Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# 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

Confusion Matrix



## IT21178054 - Model 03

```

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,)

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)

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 `in
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,
)
```



```
54/54 ━━━━━━ 19s 345ms/step - accuracy: 0.9711 - loss: 0.1074 - val_accuracy: 0.6555 - val_loss: 1.5280
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
```

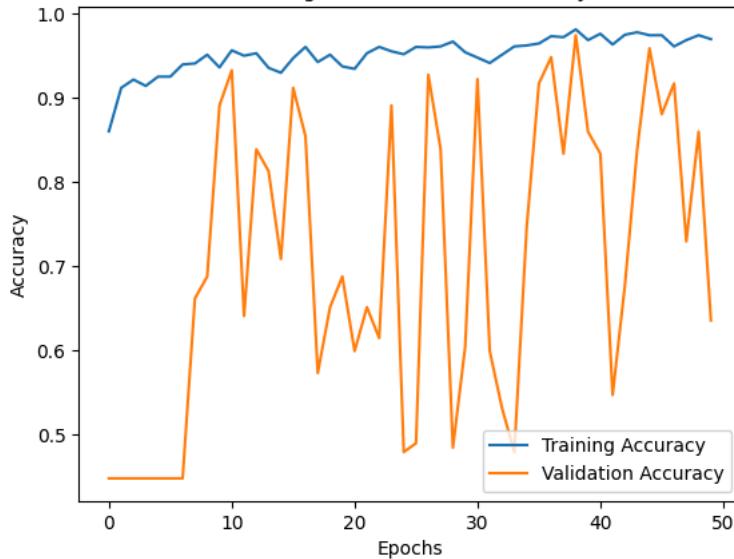
```
import matplotlib.pyplot as plt

# 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()

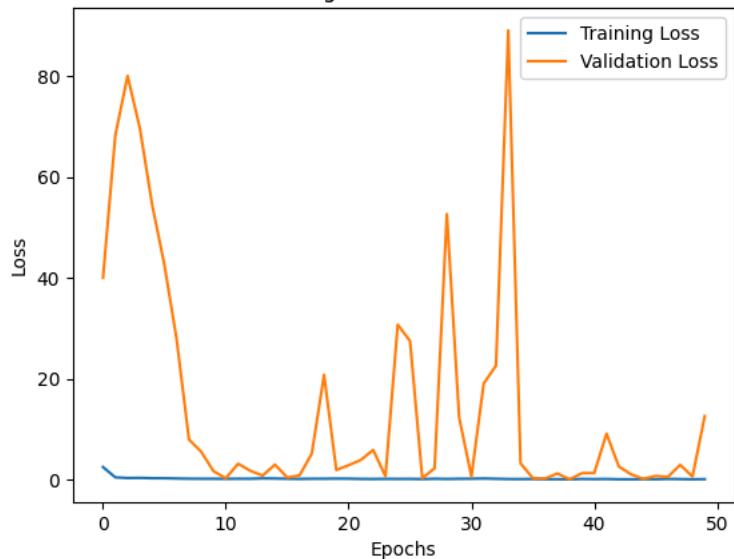
# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



Training and Validation Accuracy



Training and Validation Loss



```

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)

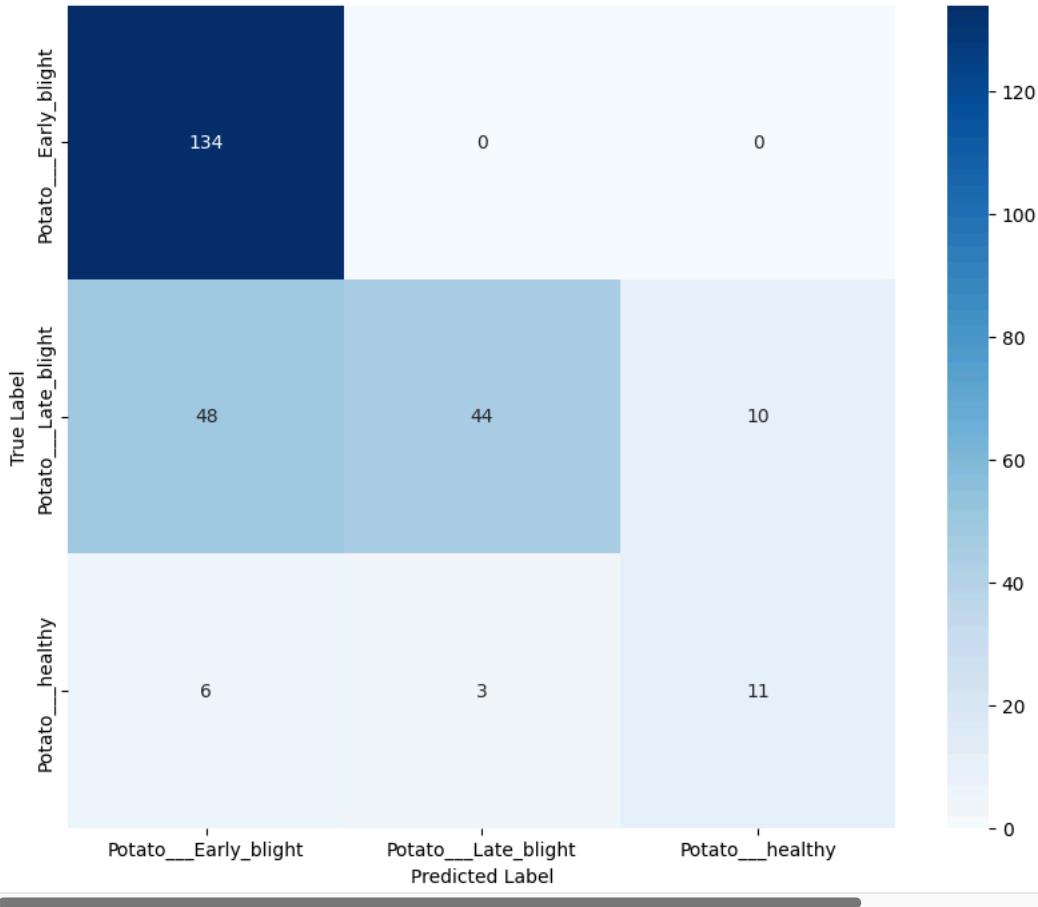
# Compute the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# 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
```

Confusion Matrix



## IT21360428 - Model 04

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

# 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),
])

# 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 `in 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
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+0
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-0
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']
```

```
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()
```

```
→
```

Training and Validation Accuracy

Training and Validation Loss