Project Title: Smart Crop Disease Detection System

Course: Artificial Intelligence

Team Members: Eshwar Kumar, Goutam, Mohammad Huzaifa

Instructor: Sir Ismail

Semester: VI Sec “C”

What is Deep Learning?

Deep Learning is a branch of Machine Learning that relies on neural networks to analyze and learn from vast datasets. These networks are capable of identifying patterns on their own and are particularly effective in handling tasks related to image recognition and natural language processing.

Why Deep Learning: Because deep networks are capable of modeling complex patterns and relationships in data, especially image-based tasks like plant disease detection.

Neural Networks

Neural networks are a key technology behind deep learning. They are inspired by how the human brain works and are made up of layers of units called neurons. These neurons are connected to each other and pass information from one layer to the next.

Input Layer – This is where data enters the network (e.g., an image or a sentence).

Hidden Layers – These layers process the data using mathematical operations. The network learns patterns in the data here.

Output Layer – This gives the final result (e.g., a predicted label like "cat" or "dog").

Each connection between neurons has a weight that adjusts as the network learns. The learning process involves adjusting these weights so the network improves at making predictions or classifications over time.

Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of neural network designed for analyzing visual data. Unlike traditional fully connected networks, CNNs use convolutional layers to learn spatial hierarchies of features from images. This makes them highly effective for tasks like image classification, object detection, and segmentation, as they can capture local patterns such as edges, textures, and shapes through convolutional filters.

Why CNN: Because they automatically capture local patterns (like disease spots or leaf texture) through filters, outperforming traditional machine learning on image recognition tasks.

Convolutional Layers: These apply filters to input images to detect features like edges, corners, and textures.

Activation Layers: Often using the ReLU (Rectified Linear Unit) function to introduce non-linearity.

Pooling Layers: These reduce the spatial dimensions of the data, making the network more computationally efficient.

Fully Connected Layers: These interpret the high-level features and output the final predictions, such as class labels.

Overview:

Plant Disease Detection is a task in the field of Artificial Intelligence that involves identifying diseases in plant leaves using images. It primarily relies on Computer Vision techniques to analyze visual patterns and detect abnormalities associated with various plant diseases.

It uses Deep Learning to classify whether a plant is healthy or diseased and, if diseased, identify the type of disease.

Our system uses deep learning models to automatically extract features from leaf images and classify them into predefined disease categories.

Tools & Technologies

Programming Language: Python

Deep Learning Frameworks: TensorFlow & Keras

Image Feature Extractor & Classifier: CNN

Model Components: CNN, Fully Connected Layers, Softmax Output

Dataset Source: Kaggle (New Plant Diseases Dataset)

Libraries Used

NumPy (np): A fundamental package for scientific computing in Python. It is used for handling large, multi-dimensional arrays and matrices, along with a wide collection of high-level mathematical functions to operate on these arrays. In this project, NumPy helps in image preprocessing and manipulation of numerical data.

TensorFlow (tf): An open-source deep learning framework developed by Google. TensorFlow is the backbone for creating, training, and deploying machine learning models, particularly deep neural networks. In this project, TensorFlow, in combination with Keras, is used to build and train the Convolutional Neural Network (CNN) for plant disease detection.

Seaborn (sns): A Python data visualization library based on Matplotlib, providing a high-level interface for drawing attractive and informative statistical graphics. Seaborn is used for visualizing the confusion matrix in our project.

OS: The OS library provides functions for interacting with the operating system, like reading directories or managing file paths. In your project, it is used for navigating the file system to load and save image files for training and testing.

Glob: A library used for finding files and directories matching a specified pattern. It helps in collecting images from directories for further processing and model training.

OpenCV (cv2): A library primarily used for computer vision tasks. OpenCV is used to load and preprocess images (such as resizing, color adjustments, and other transformations) before feeding them into the model.

Matplotlib (plt): A plotting library used for creating static, interactive, and animated visualizations. In this project, it is used to visualize the results, such as loss curves, accuracy plots, or sample images from the dataset.

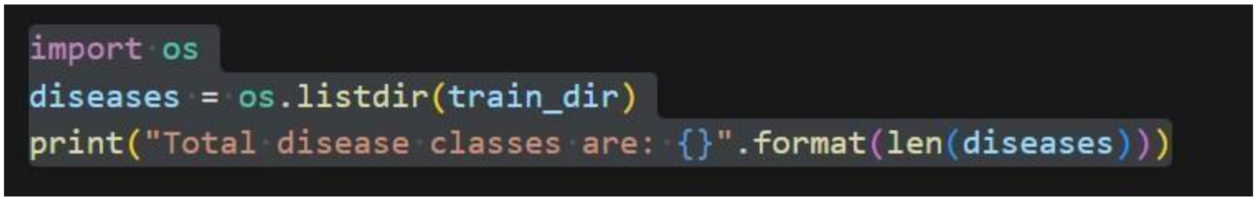
KaggleHub: A library that facilitates easy access to datasets directly from Kaggle. It is used to load the plant disease dataset for model training without manually downloading and extracting files.

Keras: A high-level neural networks API, running on top of TensorFlow. Keras simplifies the process of building and training deep learning models. In this project, Keras is used to define and compile the CNN architecture for plant disease classification.

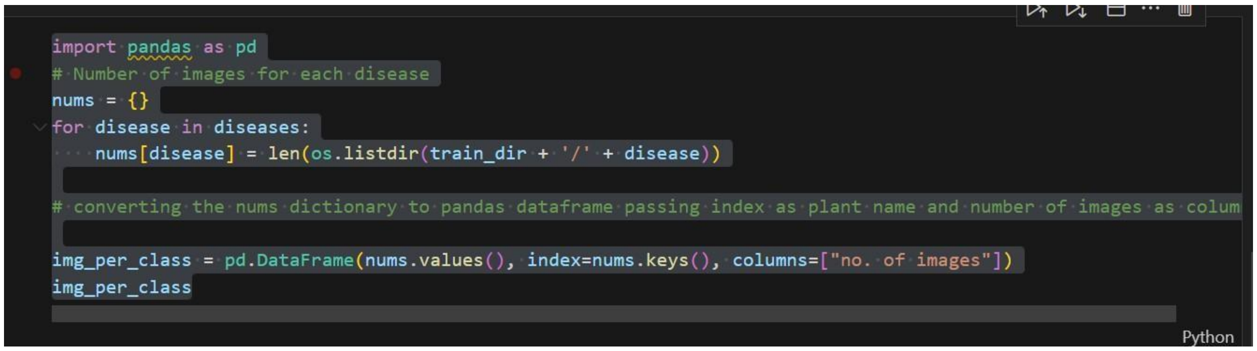
Dataset: **https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset/**

Why This Dataset?

It includes labeled, high-resolution images across 38 classes, offering diversity and depth necessary to train an accurate classification model. Also, it contained almost all balanced classes.

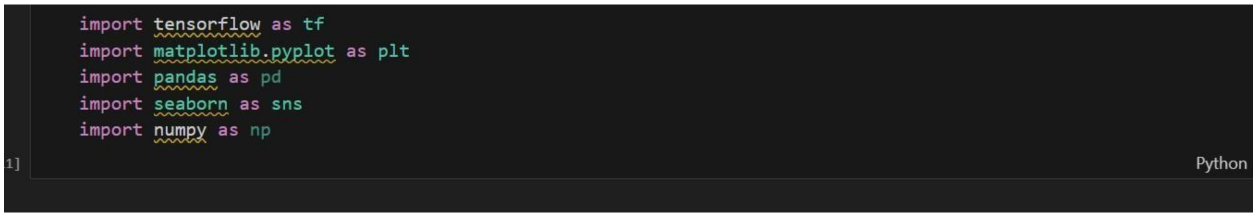


```python
import os
diseases = os.listdir(train_dir)
print("Total disease classes are: {}".format(len(diseases)))
```

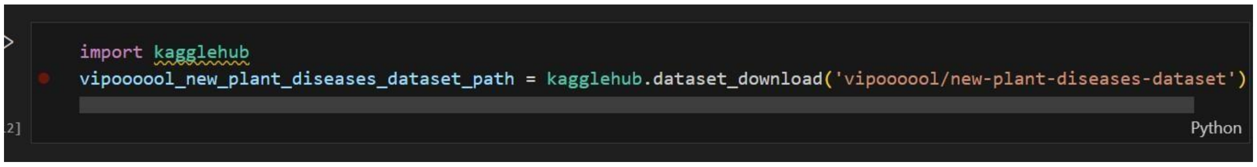


```python
import pandas as pd
# Number of images for each disease
nums = {}
for disease in diseases:
    nums[disease] = len(os.listdir(train_dir + '/' + disease))

# converting the nums dictionary to pandas dataframe passing index as plant name and number of images as colum

img_per_class = pd.DataFrame(nums.values(), index=nums.keys(), columns=["no. of images"])
img_per_class
```

To analyze the distribution of training images, we calculated the number of images available for each plant disease category. We used Python’s os module to count the number of image files in each folder (representing a specific disease). Then, we organized this information into a Pandas DataFrame to make it easier to visualize. This helped us understand whether our dataset was balanced or if some classes had significantly more or fewer images, which could affect the model's learning.

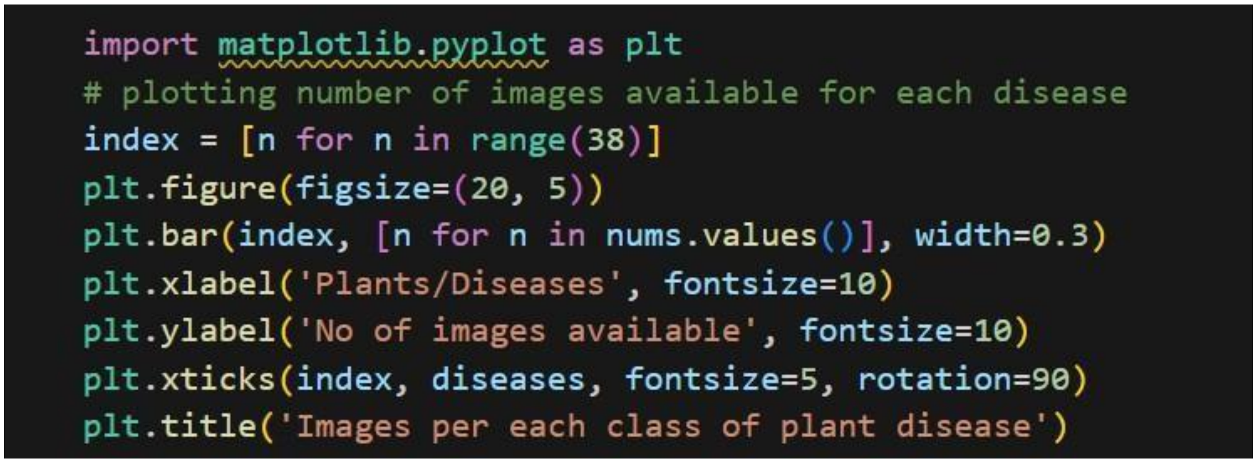


```python
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
```



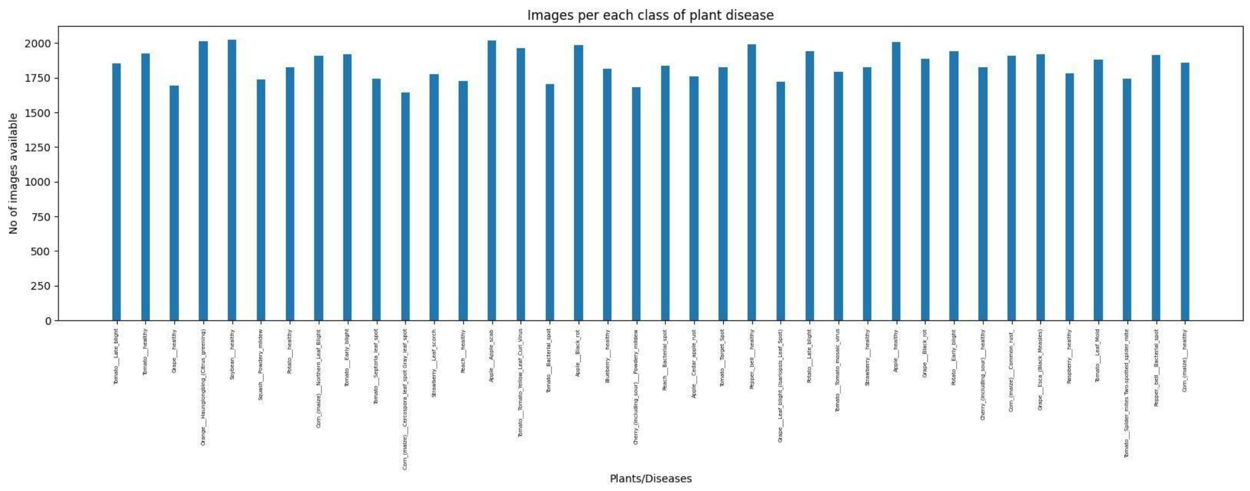
```python
import kagglehub
vipoooool_new_plant_diseases_dataset_path = kagglehub.dataset_download('vipoooool/new-plant-diseases-dataset')
```

Here we use the kagglehub helper to fetch the “vipoooool/new-plant-diseases-dataset” directly into our workspace and store its local path so we can load it later.



```
import matplotlib.pyplot as plt
# plotting number of images available for each disease
index = [n for n in range(38)]
plt.figure(figsize=(20, 5))
plt.bar(index, [n for n in nums.values()], width=0.3)
plt.xlabel('Plants/Diseases', fontsize=10)
plt.ylabel('No of images available', fontsize=10)
plt.xticks(index, diseases, fontsize=5, rotation=90)
plt.title('Images per each class of plant disease')
```

To better understand the distribution of our dataset, we plotted a bar chart showing the number of training images available for each plant disease class. We used Python's matplotlib library to visualize this data. The x-axis represents different plant diseases, while the y-axis shows the number of images available for each class. This helped us observe that data is almost balanced there is minor difference between number of images in each class.



Neural Network Components and Their Role

Input

Defines the shape of the input images (128x128x3). It tells the model what kind of input to expect and helps structure the CNN accordingly.

Resizing from 256×256 to 128×128 gave us faster training with minimal loss in classification performance.

Conv2D (Convolutional Layer)

Extracts important features from plant leaf images by applying filters that detect textures, colors, edges, and disease-specific patterns like spots or discoloration.

This is the foundational layer of the CNN that identifies visual symptoms of diseases.

MaxPooling2D (Pooling Layer)

Reduces the spatial dimensions of the feature maps while retaining the most important information.

It helps make the model efficient and robust to small changes in image position or lighting.

Activation (e.g., ReLU)

Introduces non-linearity into the model so it can learn complex features—such as the subtle differences between healthy and infected leaves.

Dropout

Randomly deactivates some neurons during training to prevent the model from overfitting on the training data.

This ensures better generalization to new, unseen images of plant leaves.

Flatten

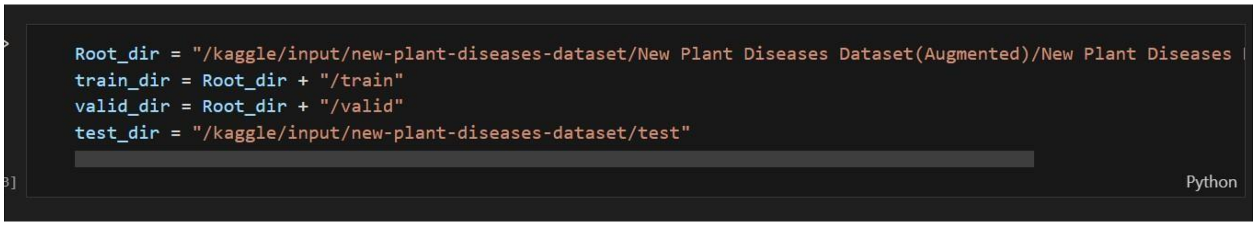
Transforms the 2D feature maps (from convolutional layers) into a 1D array so it can be fed into the Dense (fully connected) layers for classification.

Dense (Fully Connected Layer)

Performs the final decision-making. These layers interpret the extracted features and assign a probability score to each possible disease category.

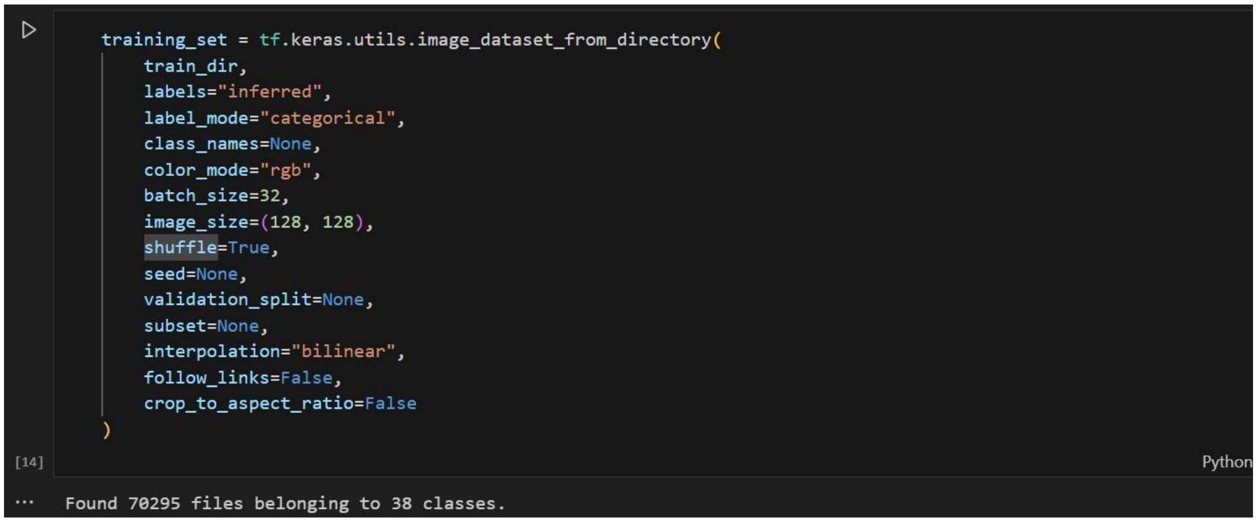
The final Dense layer often uses Softmax to output the predicted disease class.

Training:



```
Root_dir = "/kaggle/input/new-plant-diseases-dataset/New Plant Diseases Dataset(Augmented)/New Plant Diseases
train_dir = Root_dir + "/train"
valid_dir = Root_dir + "/valid"
test_dir = "/kaggle/input/new-plant-diseases-dataset/test"
```

This snippet shows how we define the main dataset folder (Root_dir) and then point train_dir, valid_dir, and test_dir to the training, validation, and test image subfolders, keeping our data organized for loading into TensorFlow.



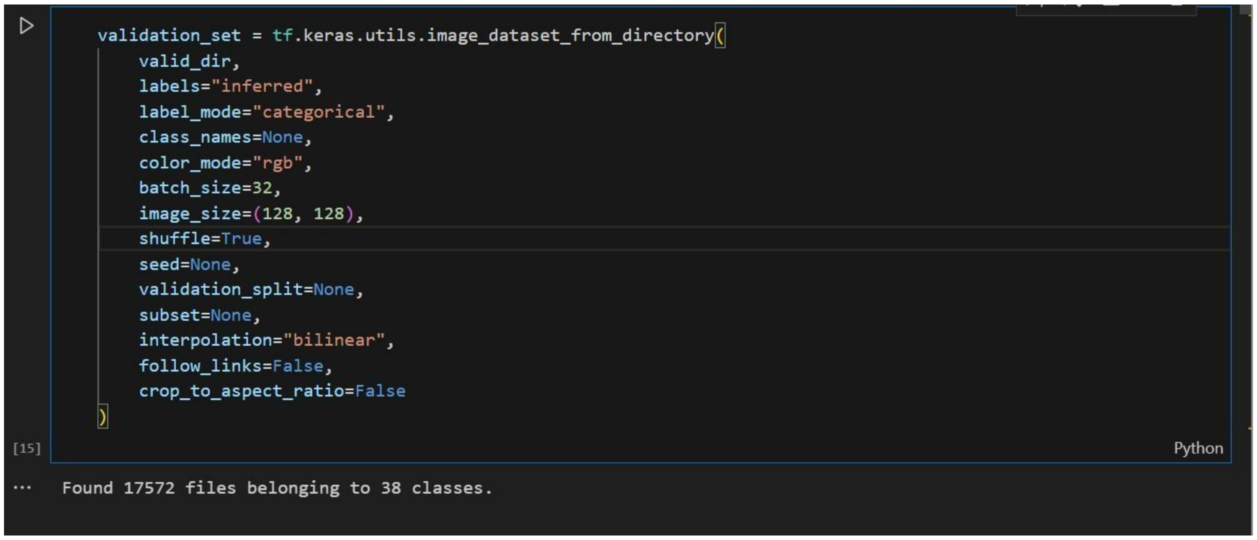
```
training_set = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    labels="inferred",
    label_mode="categorical",
    class_names=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(128, 128),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    crop_to_aspect_ratio=False
)
```

```
Found 70295 files belonging to 38 classes.
```

In this cell, we call image_dataset_from_directory on train_dir to automatically:

- Infer labels from subfolder names,
- One-hot encode them (label_mode="categorical"),
- Load and resize images to 128×128 RGB,
- Batch them into groups of 32 and shuffle each epoch,
- Use bilinear interpolation when resizing.

This gives us a tf.data.Dataset (training_set) ready to feed into our model.



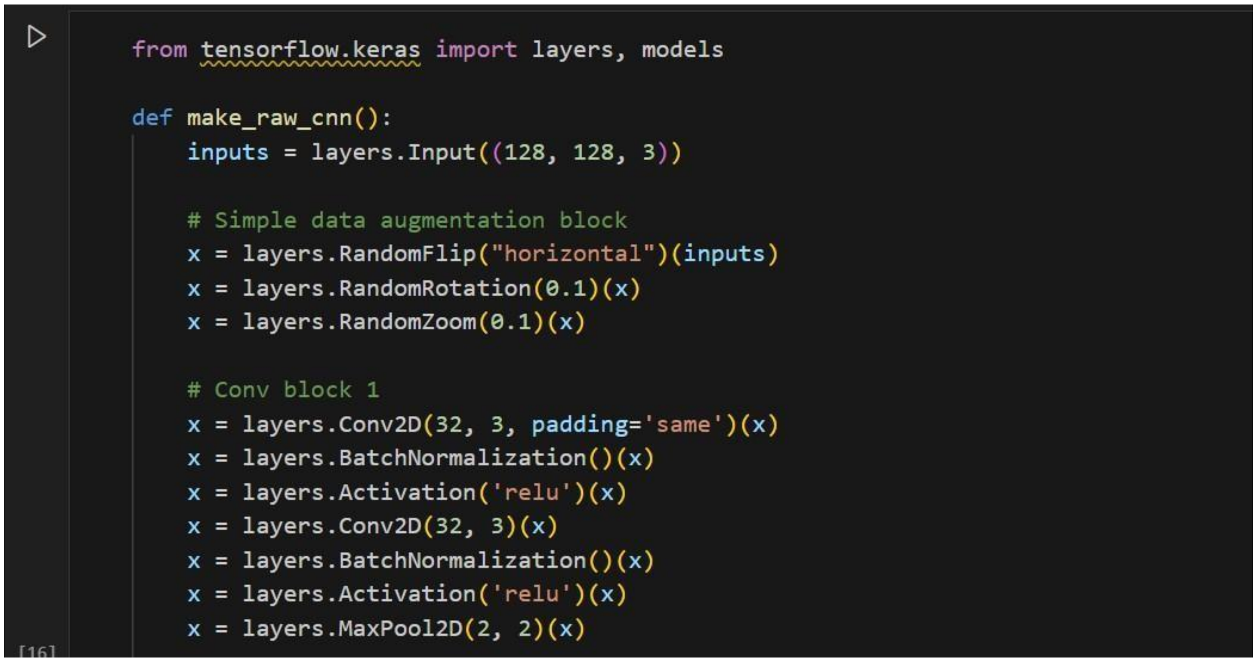
```python
validation_set = tf.keras.utils.image_dataset_from_directory(
    valid_dir,
    labels="inferred",
    label_mode="categorical",
    class_names=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(128, 128),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    crop_to_aspect_ratio=False
)
```

```
Found 17572 files belonging to 38 classes.
```

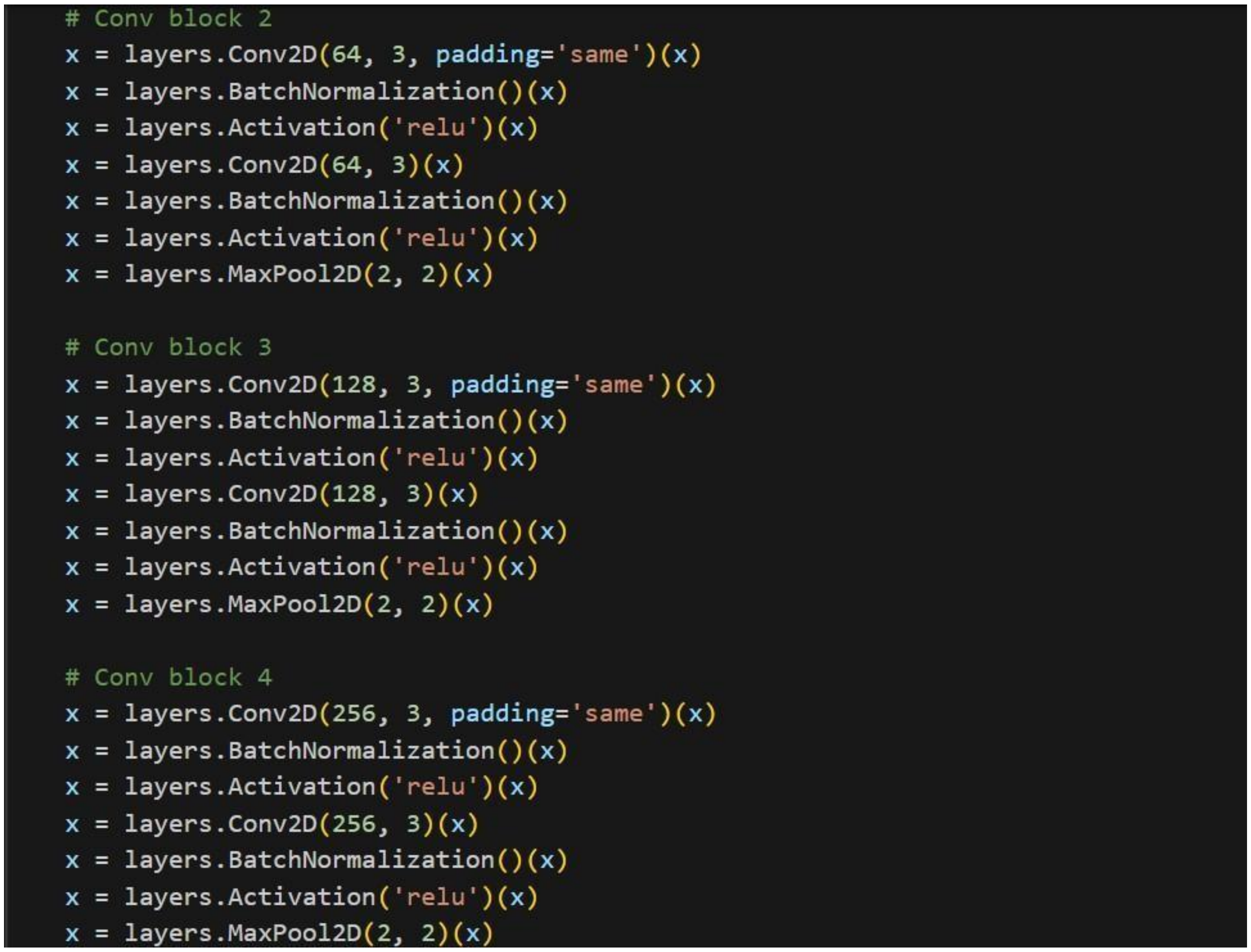
Same as training data loading but just directory is changed to validation data set

Why Use image_dataset_from_directory()?

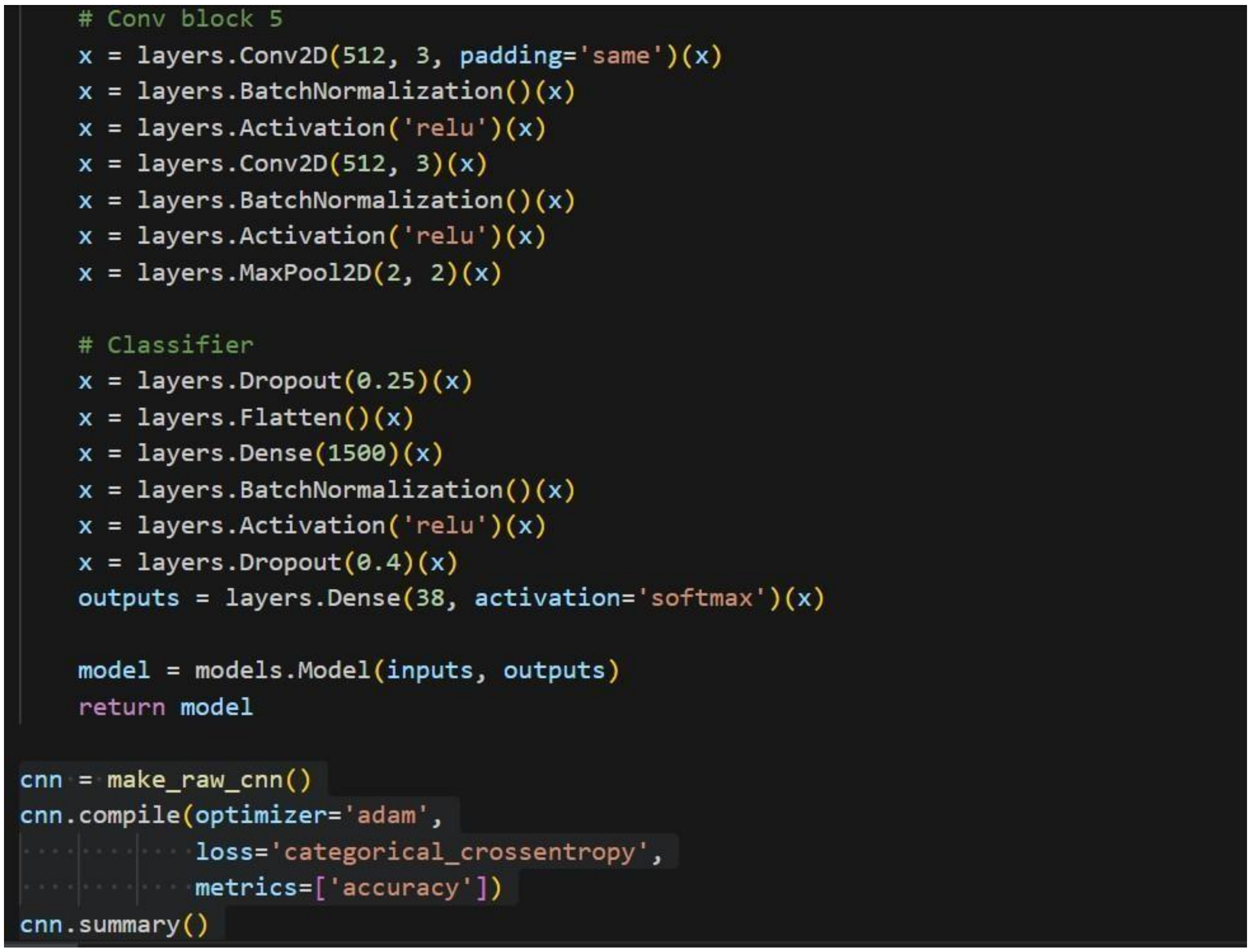
This function handles loading, resizing, batching, and labeling.
Why? It simplifies the pipeline and ensures consistent preprocessing across all data splits.



```python
from tensorflow.keras import layers, models

def make_raw_cnn():
    inputs = layers.Input((128, 128, 3))

    # Simple data augmentation block
    x = layers.RandomFlip("horizontal")(inputs)
    x = layers.RandomRotation(0.1)(x)
    x = layers.RandomZoom(0.1)(x)

    # Conv block 1
    x = layers.Conv2D(32, 3, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(32, 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(2, 2)(x)
```



```python
    # Conv block 2
    x = layers.Conv2D(64, 3, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(64, 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(2, 2)(x)

    # Conv block 3
    x = layers.Conv2D(128, 3, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(128, 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(2, 2)(x)

    # Conv block 4
    x = layers.Conv2D(256, 3, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(256, 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(2, 2)(x)
```



```
    # Conv block 5
    x = layers.Conv2D(512, 3, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(512, 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(2, 2)(x)

    # Classifier
    x = layers.Dropout(0.25)(x)
    x = layers.Flatten()(x)
    x = layers.Dense(1500)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Dropout(0.4)(x)
    outputs = layers.Dense(38, activation='softmax')(x)

    model = models.Model(inputs, outputs)
    return model

cnn = make_raw_cnn()
cnn.compile(optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy'])
cnn.summary()
```

What we’re doing overall:

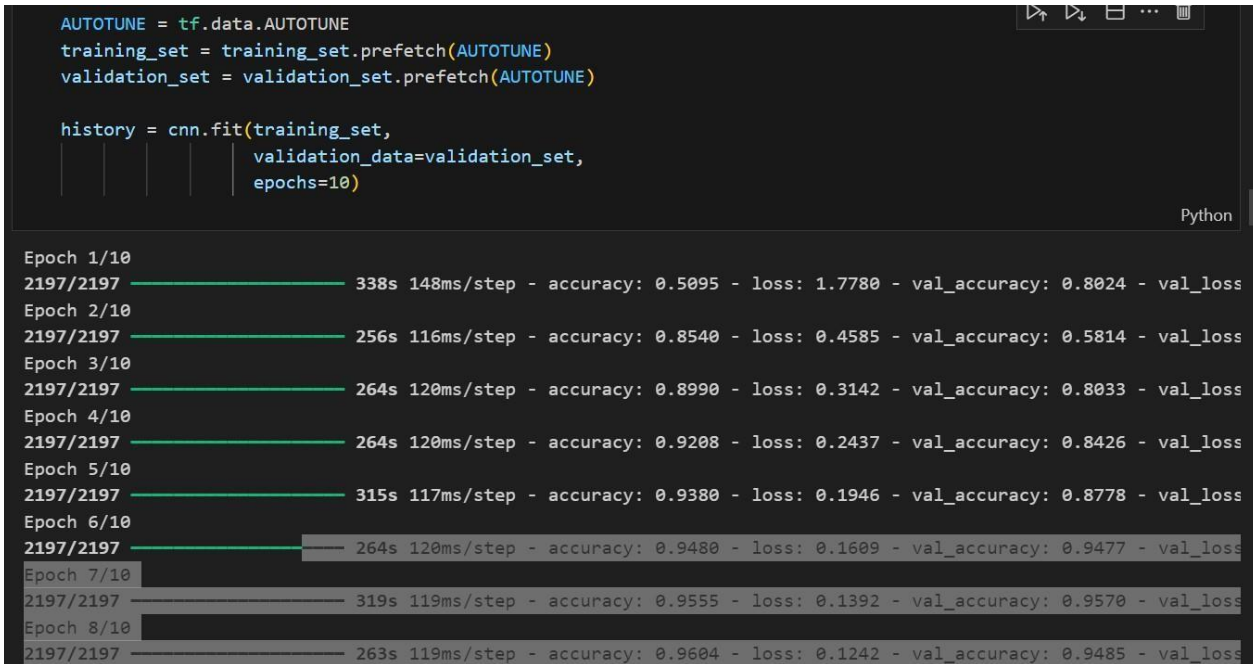
We are building a custom CNN model (function name make_raw_cnn) that:

- Takes 128×128 colored images as input,
- Applies data augmentation to improve generalization,
- Passes images through multiple convolutional layers to extract features,
- Finally classifies them into 38 classes using a softmax output layer.

Concepts used in this model:

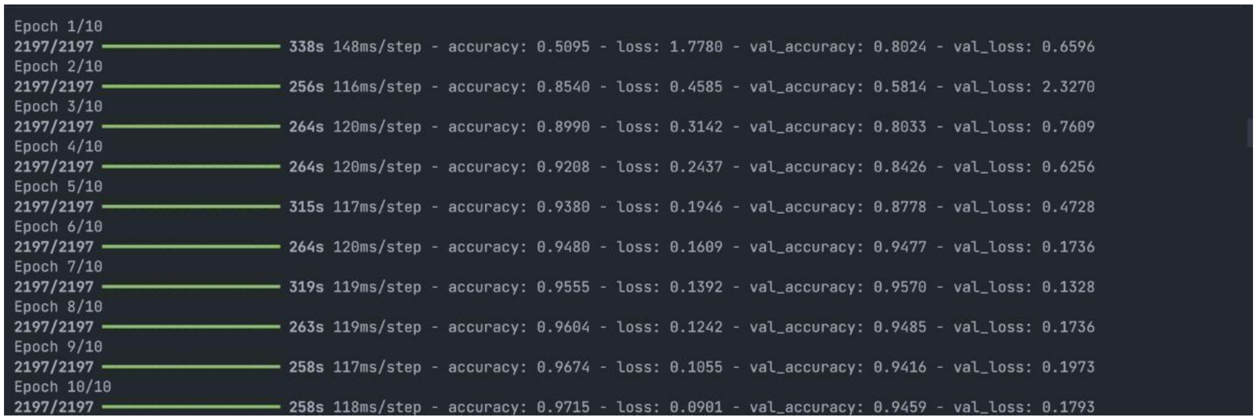
1. **Data Augmentation** – Randomly flips, rotates, and zooms images during training to make the model more robust.
2. **Convolutional Layers** – Extract important features like edges, shapes, and patterns from images.

3. **Batch Normalization** – Speeds up training and makes it more stable.
4. **Activation Function (ReLU)** – Helps the model learn complex patterns.
5. **Max Pooling** – Reduces image size to make computation faster while keeping key features.
6. **Dropout** – Prevents overfitting by randomly turning off some neurons during training.
7. **Dense Layer** – Fully connected layer that acts like the model's brain for decisionmaking.
8. **Softmax Activation** – Outputs probabilities for each class (used for multi-class classification).
9. **Compilation** – We use Adam optimizer and categorical crossentropy loss function, which is standard for multi-class problems.



```python
AUTOTUNE = tf.data.AUTOTUNE
training_set = training_set.prefetch(AUTOTUNE)
validation_set = validation_set.prefetch(AUTOTUNE)

history = cnn.fit(training_set,
                  validation_data=validation_set,
                  epochs=10)
```

```
Epoch 1/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 338s 148ms/step - accuracy: 0.5095 - loss: 1.7780 - val_accuracy: 0.8024 - val_loss
Epoch 2/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 256s 116ms/step - accuracy: 0.8540 - loss: 0.4585 - val_accuracy: 0.5814 - val_loss
Epoch 3/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.8990 - loss: 0.3142 - val_accuracy: 0.8033 - val_loss
Epoch 4/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.9208 - loss: 0.2437 - val_accuracy: 0.8426 - val_loss
Epoch 5/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 315s 117ms/step - accuracy: 0.9380 - loss: 0.1946 - val_accuracy: 0.8778 - val_loss
Epoch 6/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.9480 - loss: 0.1609 - val_accuracy: 0.9477 - val_loss
Epoch 7/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 319s 119ms/step - accuracy: 0.9555 - loss: 0.1392 - val_accuracy: 0.9570 - val_loss
Epoch 8/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 263s 119ms/step - accuracy: 0.9604 - loss: 0.1242 - val_accuracy: 0.9485 - val_loss
```



```
Epoch 1/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 338s 148ms/step - accuracy: 0.5095 - loss: 1.7780 - val_accuracy: 0.8024 - val_loss: 0.6596
Epoch 2/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 256s 116ms/step - accuracy: 0.8540 - loss: 0.4585 - val_accuracy: 0.5814 - val_loss: 2.3270
Epoch 3/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.8990 - loss: 0.3142 - val_accuracy: 0.8033 - val_loss: 0.7609
Epoch 4/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.9208 - loss: 0.2437 - val_accuracy: 0.8426 - val_loss: 0.6256
Epoch 5/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 315s 117ms/step - accuracy: 0.9380 - loss: 0.1946 - val_accuracy: 0.8778 - val_loss: 0.4728
Epoch 6/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 264s 120ms/step - accuracy: 0.9480 - loss: 0.1609 - val_accuracy: 0.9477 - val_loss: 0.1736
Epoch 7/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 319s 119ms/step - accuracy: 0.9555 - loss: 0.1392 - val_accuracy: 0.9570 - val_loss: 0.1328
Epoch 8/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 263s 119ms/step - accuracy: 0.9604 - loss: 0.1242 - val_accuracy: 0.9485 - val_loss: 0.1736
Epoch 9/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 258s 117ms/step - accuracy: 0.9674 - loss: 0.1055 - val_accuracy: 0.9416 - val_loss: 0.1973
Epoch 10/10
2197/2197 ━━━━━━━━━━━━━━━━━━━━ 258s 118ms/step - accuracy: 0.9715 - loss: 0.0901 - val_accuracy: 0.9459 - val_loss: 0.1793
```

We are telling TensorFlow to **prepare the next batch of data in the background** while the model is training on the current batch.

This is done using .prefetch(AUTOTUNE) which helps **speed up training** and **reduce waiting time**.

AUTOTUNE lets TensorFlow automatically choose the best prefetch buffer size.

Why Use Prefetching with AUTOTUNE?

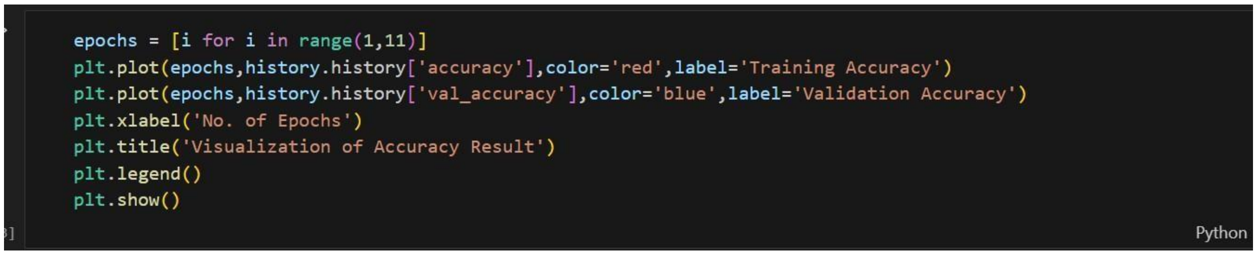
Speeds up training by loading future data in the background.
Why? Reduces latency and increases efficiency of training loops.

We start training our CNN model using the **training dataset**.

At the same time, we check the model's performance on the **validation dataset** after each epoch.

We train for **10 epochs**, meaning the model sees the entire training data 10 times.

Why 10 epochs? 10 epochs provided a good trade-off between accuracy, training time, and overfitting prevention.



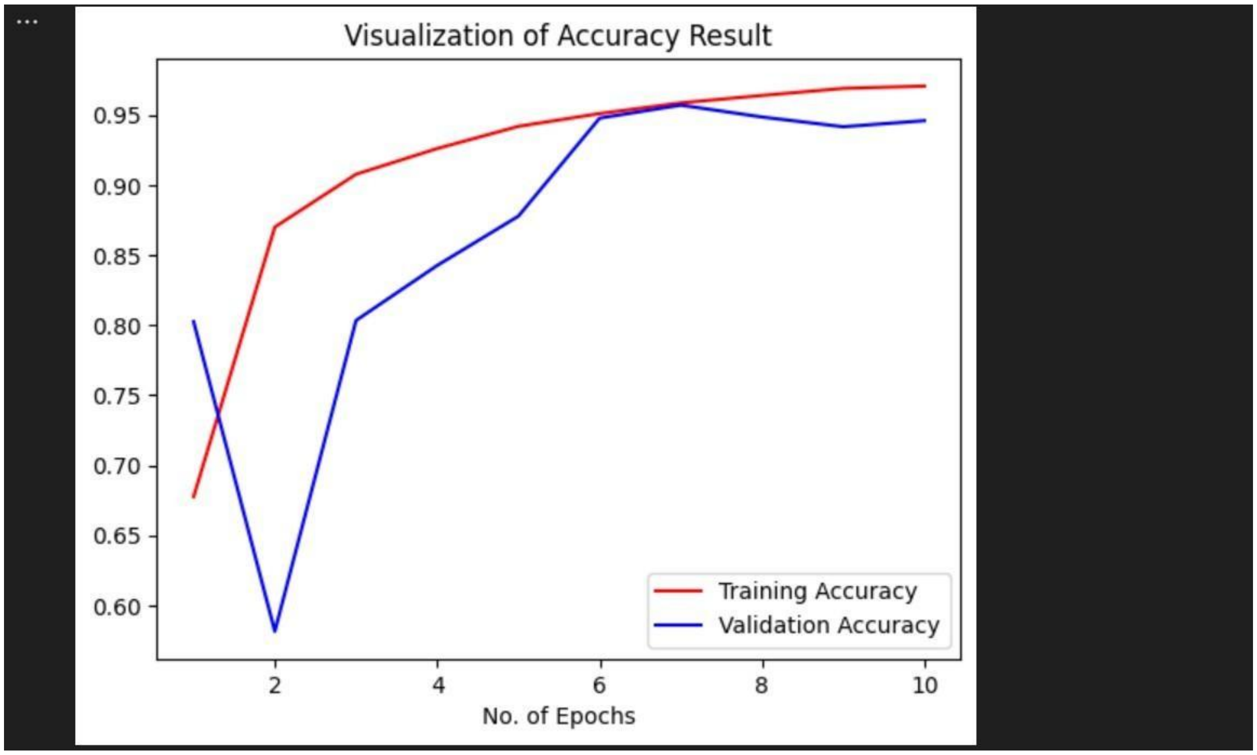
```python
epochs = [i for i in range(1,11)]
plt.plot(epochs,history.history['accuracy'],color='red',label='Training Accuracy')
plt.plot(epochs,history.history['val_accuracy'],color='blue',label='Validation Accuracy')
plt.xlabel('No. of Epochs')
plt.title('Visualization of Accuracy Result')
plt.legend()
plt.show()
```

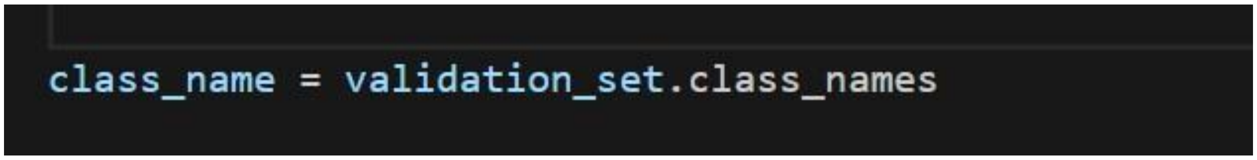
To analyze the performance of our Convolutional Neural Network (CNN) model, we plotted the training and validation accuracy for each epoch. We trained the model for 10 epochs and recorded accuracy values at the end of each one.

Using Matplotlib, we created a line plot where the red line represents the training accuracy and the blue line shows the validation accuracy. This graph helps us observe how well the model is learning during training and how it generalizes to unseen data.

If the validation accuracy closely follows the training accuracy, it indicates that the model is not overfitting and is generalizing well. Any large gap between these lines would indicate overfitting or underfitting.

This visualization provides a clear understanding of how the model improved over time.



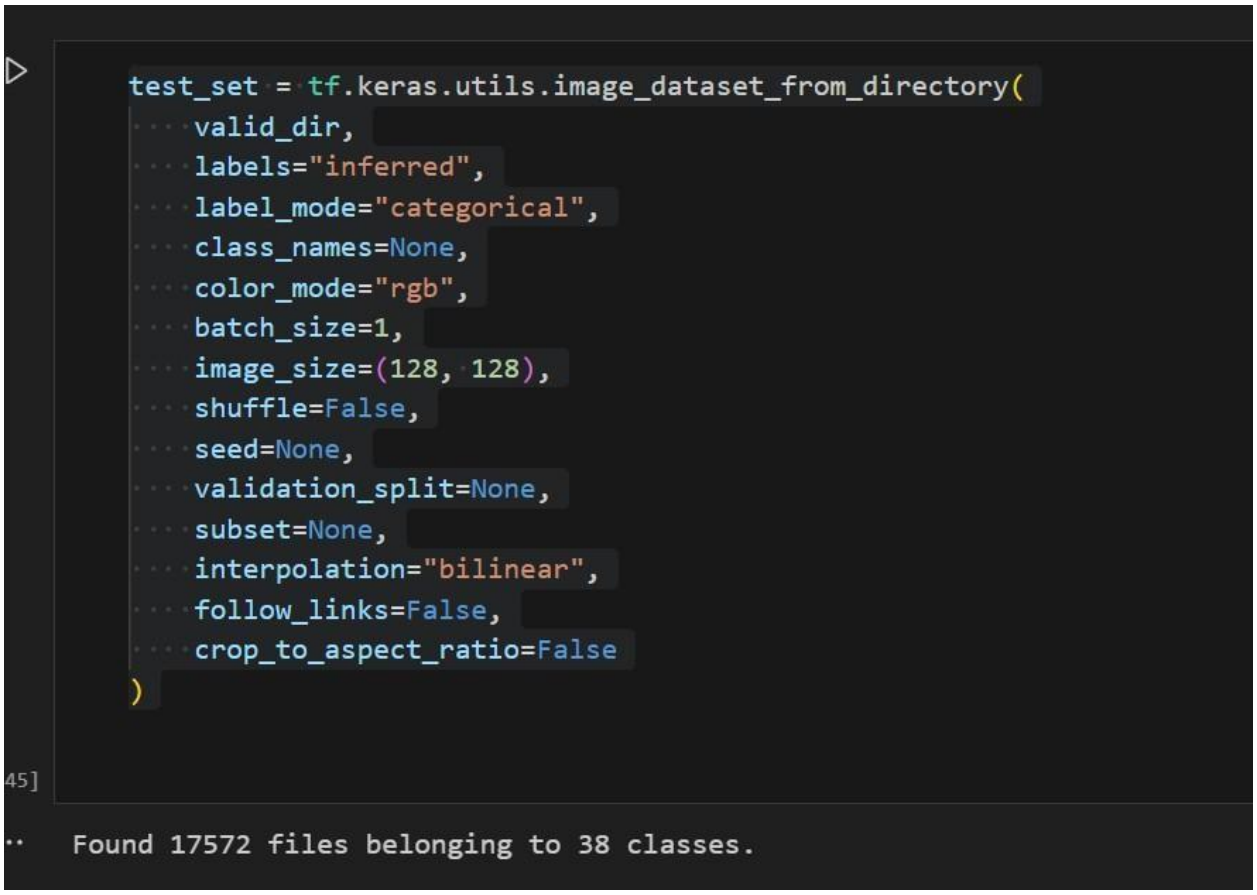


```
class_name = validation_set.class_names
```

We are extracting the **list of class labels** (like ['Potato', 'Tomato', 'Apple']) from the validation_set.
This helps us later to convert predicted class **indexes (0, 1, 2, etc.)** back into actual names (like 'cat').

Why? Because the model gives output in numeric form, and we need class names to interpret the results.

Testing:



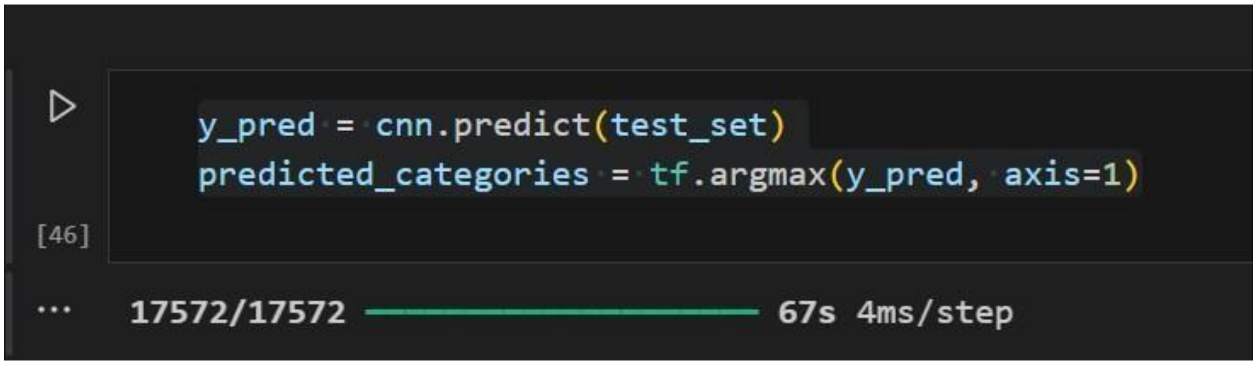
```
test_set = tf.keras.utils.image_dataset_from_directory(
    valid_dir,
    labels="inferred",
    label_mode="categorical",
    class_names=None,
    color_mode="rgb",
    batch_size=1,
    image_size=(128, 128),
    shuffle=False,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    crop_to_aspect_ratio=False
)
```

```
Found 17572 files belonging to 38 classes.
```

Here we are **reloading the validation set again**, but with batch_size=1 and shuffle=False.
We do this because:

- batch_size=1 allows us to predict **one image at a time**.
- shuffle=False ensures the order of images is preserved, which is important when comparing predictions with true labels.

Why? To perform clean and accurate prediction comparison.



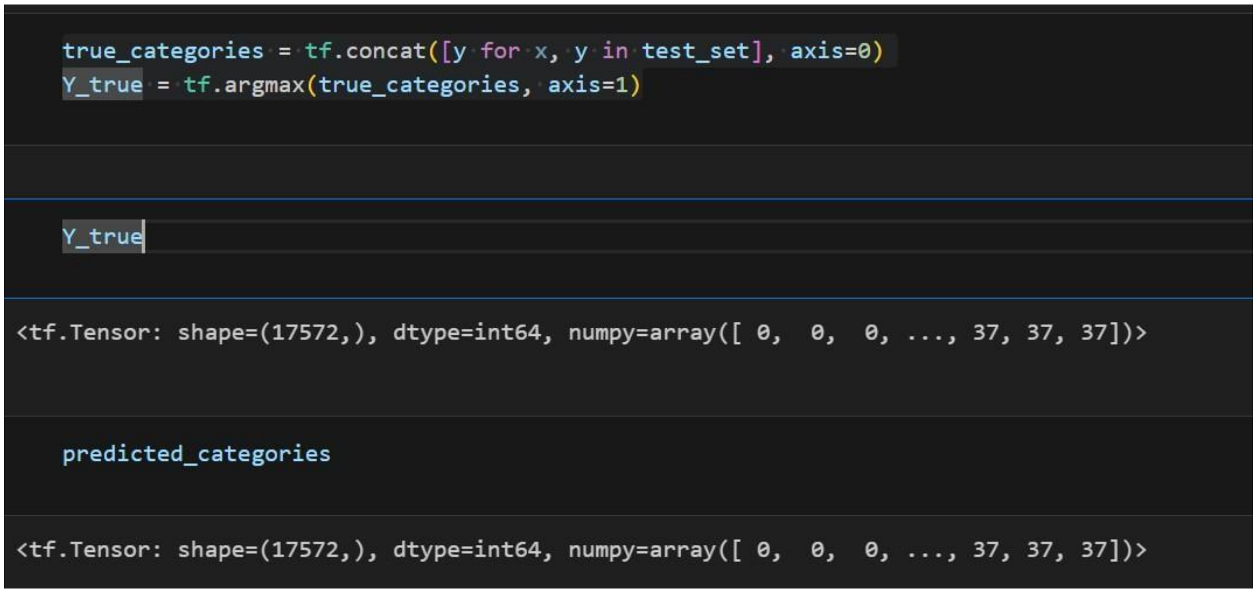
```
[46]
y_pred = cnn.predict(test_set)
predicted_categories = tf.argmax(y_pred, axis=1)

17572/17572 ━━━━━━━━━━━━━━━━━━━━ 67s 4ms/step
```

We are using our trained CNN model to make predictions on each image in the test_set.

The result y_pred contains probabilities for each class for every image.

This takes the **index of the highest value** from each prediction.
So if a prediction is [0.01, 0.97, 0.02], the highest value is at index 1, so the predicted class is 1.

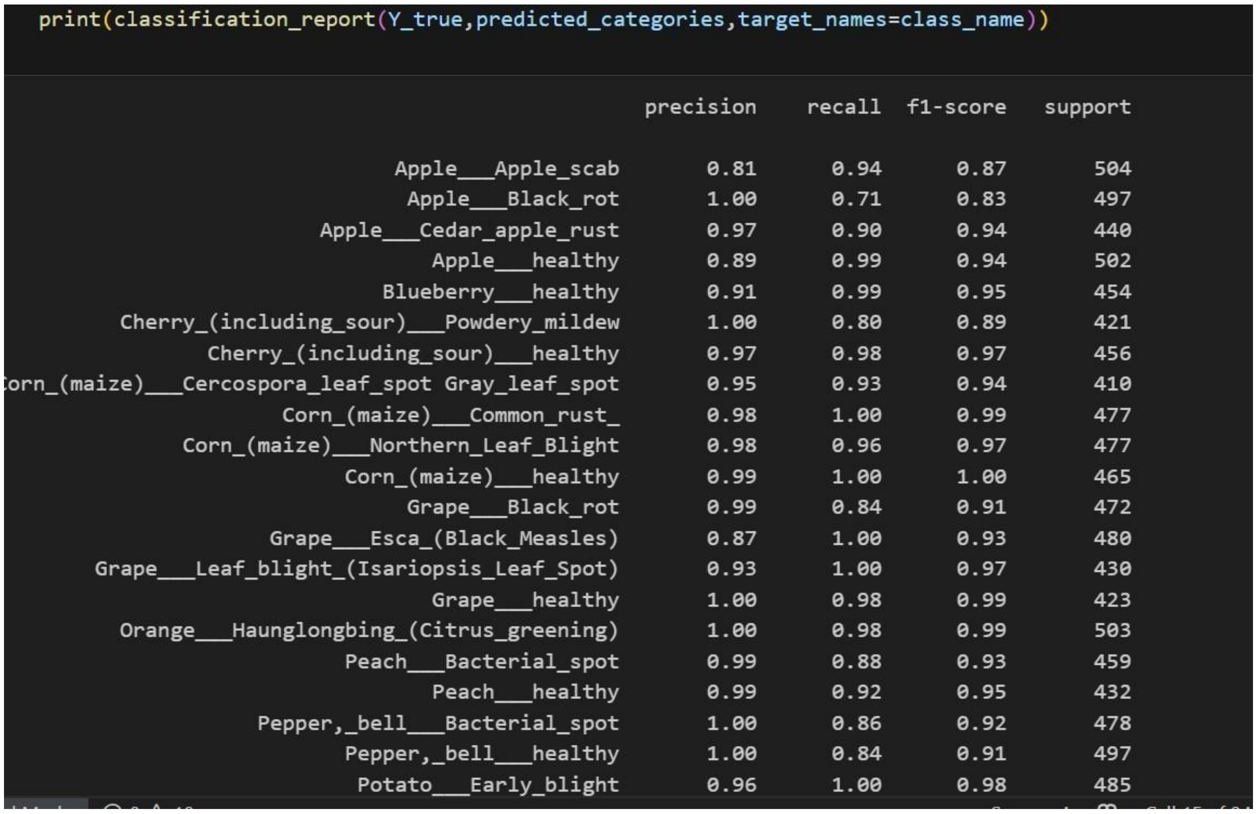
Why? Because the class with the highest probability is the final predicted class.



```
true_categories = tf.concat([y for x, y in test_set], axis=0)
Y_true = tf.argmax(true_categories, axis=1)

Y_true

<tf.Tensor: shape=(17572,), dtype=int64, numpy=array([ 0,  0,  0, ..., 37, 37, 37])>

predicted_categories

<tf.Tensor: shape=(17572,), dtype=int64, numpy=array([ 0,  0,  0, ..., 37, 37, 37])>
```

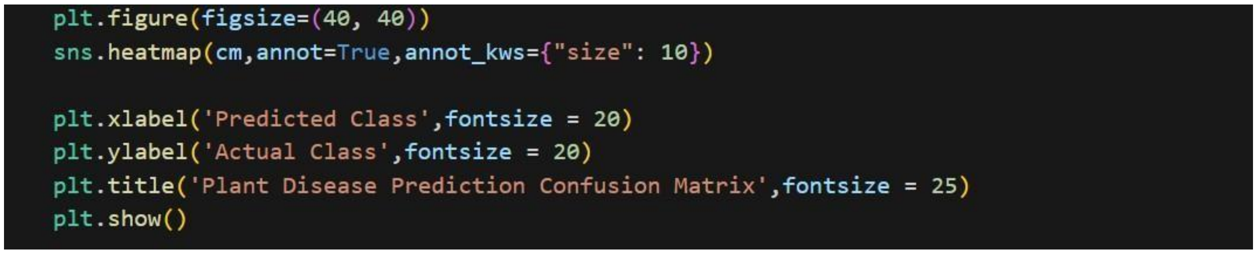
- First line combines all true labels from the dataset into one big tensor.
- Second line converts one-hot encoded true labels (like [0, 1, 0]) into class indexes (like 1).

Why? To get the actual correct classes so we can compare with predicted ones.

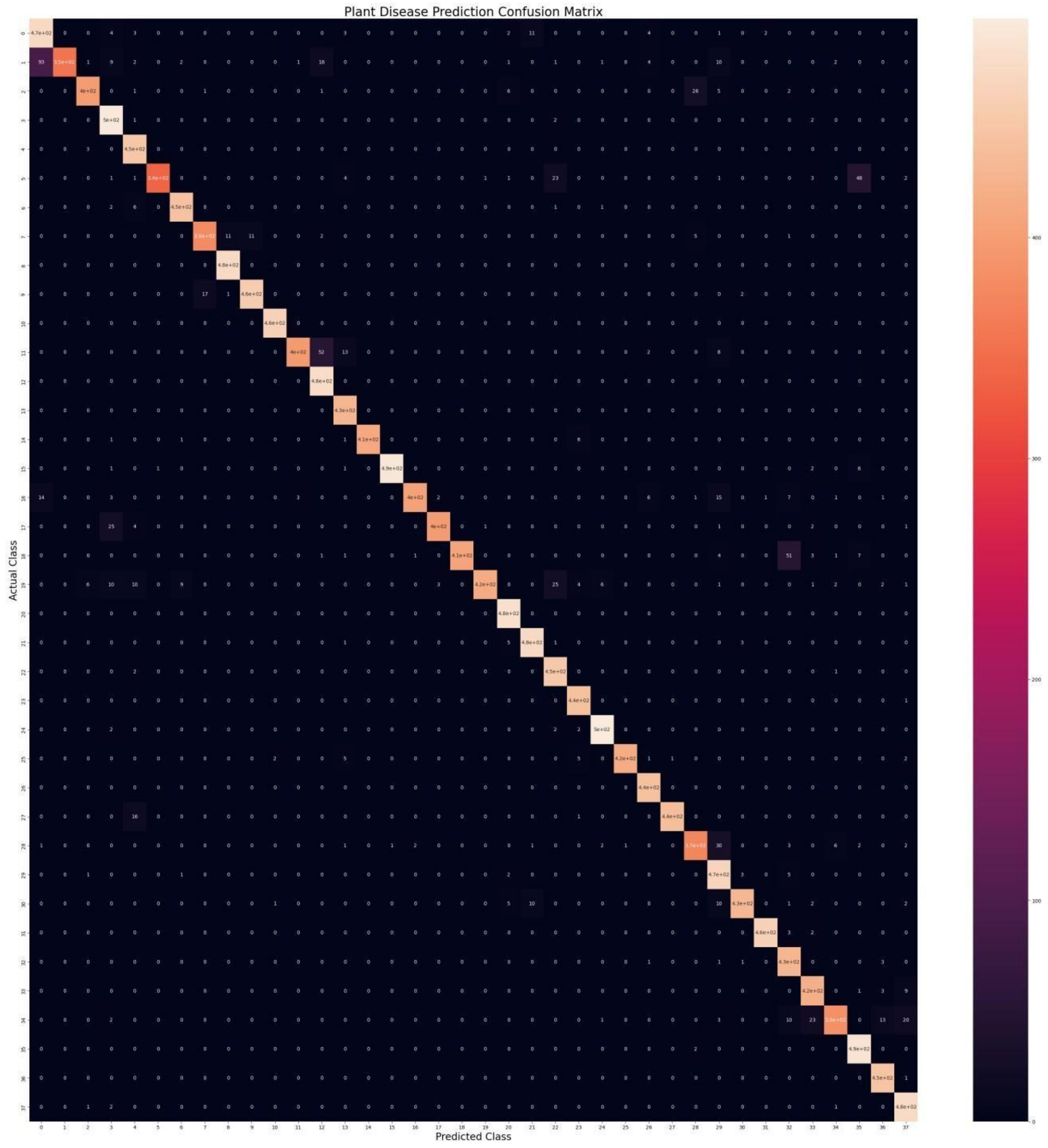


```
print(classification_report(Y_true,predicted_categories,target_names=class_name))
```

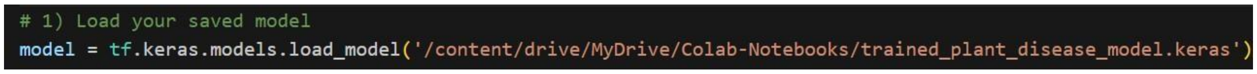
```
                                                     precision    recall  f1-score   support

                                 Apple___Apple_scab       0.81      0.94      0.87       504
                                  Apple___Black_rot       1.00      0.71      0.83       497
                           Apple___Cedar_apple_rust       0.97      0.90      0.94       440
                                    Apple___healthy       0.89      0.99      0.94       502
                                Blueberry___healthy       0.91      0.99      0.95       454
           Cherry_(including_sour)___Powdery_mildew       1.00      0.80      0.89       421
                  Cherry_(including_sour)___healthy       0.97      0.98      0.97       456
orn_(maize)___Cercospora_leaf_spot Gray_leaf_spot       0.95      0.93      0.94       410
                       Corn_(maize)___Common_rust_       0.98      1.00      0.99       477
               Corn_(maize)___Northern_Leaf_Blight       0.98      0.96      0.97       477
                            Corn_(maize)___healthy       0.99      1.00      1.00       465
                                 Grape___Black_rot       0.99      0.84      0.91       472
                      Grape___Esca_(Black_Measles)       0.87      1.00      0.93       480
        Grape___Leaf_blight_(Isariopsis_Leaf_Spot)       0.93      1.00      0.97       430
                                   Grape___healthy       1.00      0.98      0.99       423
          Orange___Haunglongbing_(Citrus_greening)       1.00      0.98      0.99       503
                            Peach___Bacterial_spot       0.99      0.88      0.93       459
                                   Peach___healthy       0.99      0.92      0.95       432
                     Pepper,_bell___Bacterial_spot       1.00      0.86      0.92       478
                            Pepper,_bell___healthy       1.00      0.84      0.91       497
                             Potato___Early_blight       0.96      1.00      0.98       485
```



```
plt.figure(figsize=(40, 40))
sns.heatmap(cm,annot=True,annot_kws={"size": 10})

plt.xlabel('Predicted Class',fontsize = 20)
plt.ylabel('Actual Class',fontsize = 20)
plt.title('Plant Disease Prediction Confusion Matrix',fontsize = 25)
plt.show()
```



To evaluate how well our model performed across all classes, we plotted a confusion matrix using the test set predictions. We used a heatmap to visualize the matrix, where each row represents the actual class and each column represents the predicted class. A larger figure size (40x40 inches) was chosen due to the high number of plant disease categories, which made the matrix easier to read. The matrix clearly shows where the model made correct predictions (diagonal values) and where it made mistakes (offdiagonal values). This helped us analyze which classes were often confused by the model.

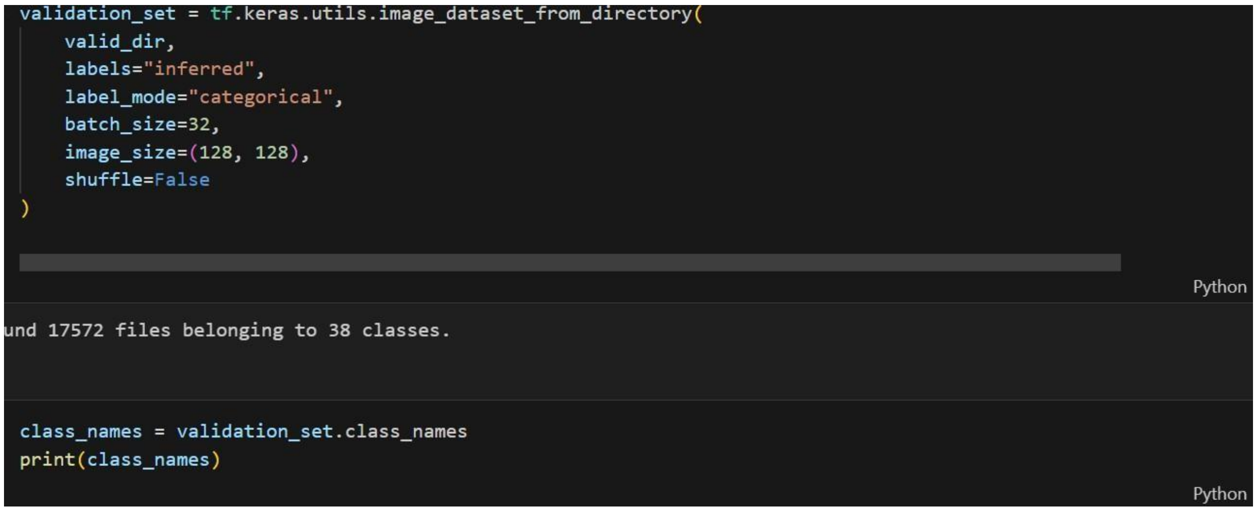


```python
# 1) Load your saved model
model = tf.keras.models.load_model('/content/drive/MyDrive/Colab-Notebooks/trained_plant_disease_model.keras')
```

Loads the saved Keras model file from your Drive.

Rebuilds its network architecture and restores all learned weights (and optimizer state, if saved).

Returns a ready-to-use tf.keras.Model instance for inference or further training.



```python
validation_set = tf.keras.utils.image_dataset_from_directory(
    valid_dir,
    labels="inferred",
    label_mode="categorical",
    batch_size=32,
    image_size=(128, 128),
    shuffle=False
)
```

Python

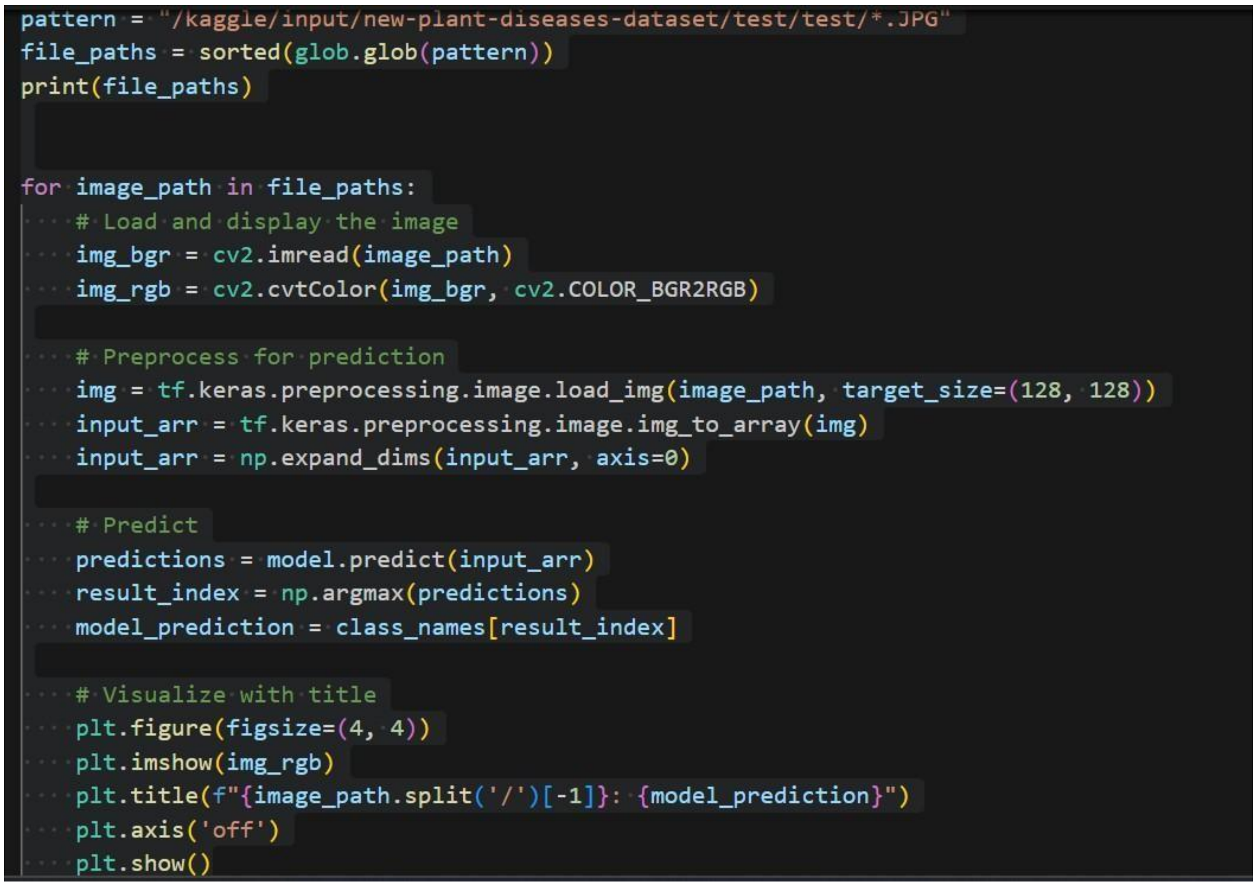
```
und 17572 files belonging to 38 classes.
```

```python
class_names = validation_set.class_names
print(class_names)
```

Python

We organized our held-out images in a directory (valid_dir) where each subfolder name corresponds to one of the target classes. To load and preprocess these images for evaluation, we used TensorFlow’s high-level utility:

Class_name ensures a consistent mapping between the one-hot label indices and the original class names during evaluation and interpretation.



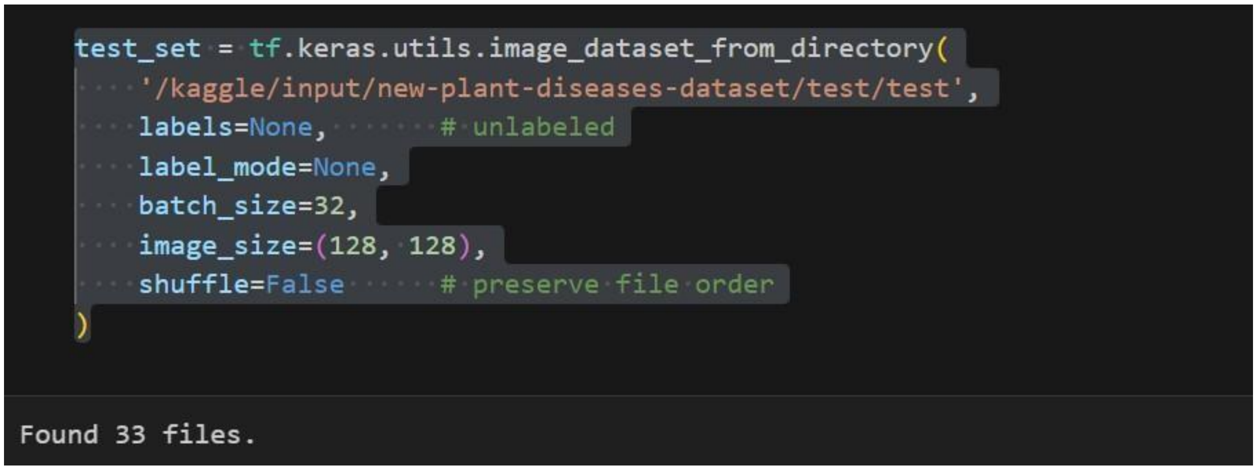
```python
pattern = "/kaggle/input/new-plant-diseases-dataset/test/test/*.JPG"
file_paths = sorted(glob.glob(pattern))
print(file_paths)


for image_path in file_paths:
    # Load and display the image
    img_bgr = cv2.imread(image_path)
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

    # Preprocess for prediction
    img = tf.keras.preprocessing.image.load_img(image_path, target_size=(128, 128))
    input_arr = tf.keras.preprocessing.image.img_to_array(img)
    input_arr = np.expand_dims(input_arr, axis=0)

    # Predict
    predictions = model.predict(input_arr)
    result_index = np.argmax(predictions)
    model_prediction = class_names[result_index]

    # Visualize with title
    plt.figure(figsize=(4, 4))
    plt.imshow(img_rgb)
    plt.title(f"{image_path.split('/')[-1]}: {model_prediction}")
    plt.axis('off')
    plt.show()
```

We use glob to retrieve and sort all .JPG test-set filenames.

OpenCV (cv2) reads in BGR order; we convert to RGB so matplotlib shows correct colors.

tf.keras.preprocessing.image.load_img + img_to_array handles resizing and array conversion, then we add a batch dimension. model.predict() returns a vector of probabilities; np.argmax picks the most likely class.

Finally, each image is plotted with its filename and predicted label, which allows for quick visual verification of model performance on diverse test samples.

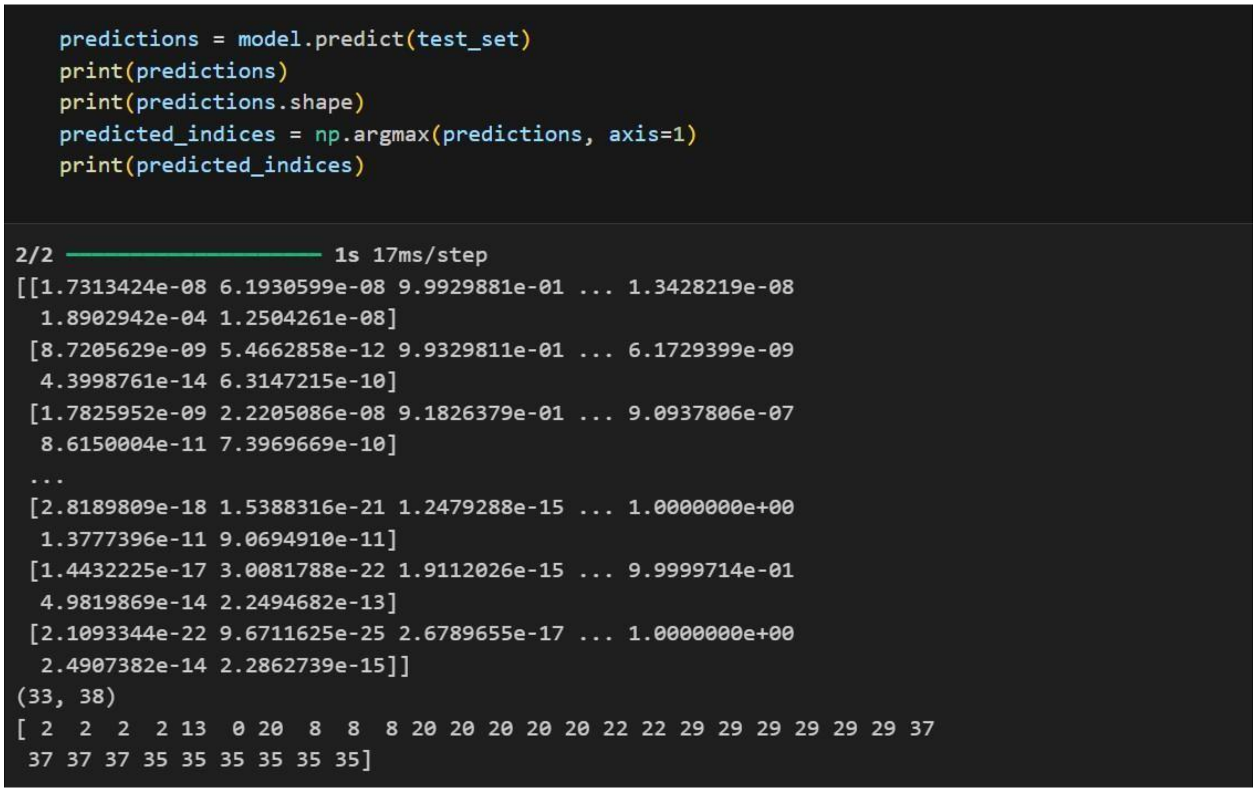


```
test_set = tf.keras.utils.image_dataset_from_directory(
    '/kaggle/input/new-plant-diseases-dataset/test/test',
    labels=None,       # unlabeled
    label_mode=None,
    batch_size=32,
    image_size=(128, 128),
    shuffle=False      # preserve file order
)
```

```
Found 33 files.
```

We load all test images (without labels) into a tf.data.Dataset for batched inference using TensorFlow's high-level utility

- By specifying labels=None and label_mode=None, the dataset yields only (image_batch,) tuples, which is ideal when you want to run model.predict() rather than model.evaluate().
- The fixed image_size ensures consistency with the input shape used during training.
- Disabling shuffling (shuffle=False) preserves the file order so that predictions can be easily mapped back to filenames if needed.



```
predictions = model.predict(test_set)
print(predictions)
print(predictions.shape)
predicted_indices = np.argmax(predictions, axis=1)
print(predicted_indices)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step
[[1.7313424e-08 6.1930599e-08 9.9929881e-01 ... 1.3428219e-08
  1.8902942e-04 1.2504261e-08]
 [8.7205629e-09 5.4662858e-12 9.9329811e-01 ... 6.1729399e-09
  4.3998761e-14 6.3147215e-10]
 [1.7825952e-09 2.2205086e-08 9.1826379e-01 ... 9.0937806e-07
  8.6150004e-11 7.3969669e-10]
 ...
 [2.8189809e-18 1.5388316e-21 1.2479288e-15 ... 1.0000000e+00
  1.3777396e-11 9.0694910e-11]
 [1.4432225e-17 3.0081788e-22 1.9112026e-15 ... 9.9999714e-01
  4.9819869e-14 2.2494682e-13]
 [2.1093344e-22 9.6711625e-25 2.6789655e-17 ... 1.0000000e+00
  2.4907382e-14 2.2862739e-15]]
(33, 38)
[ 2  2  2  2 13  0 20  8  8  8 20 20 20 20 20 22 22 29 29 29 29 29 29 37
 37 37 37 35 35 35 35 35 35]
```

Feeds every image in test_set through the trained CNN.

Returns a NumPy array of shape (N, 38), where N is the total number of test samples (here 33) and 38 is your number of classes.

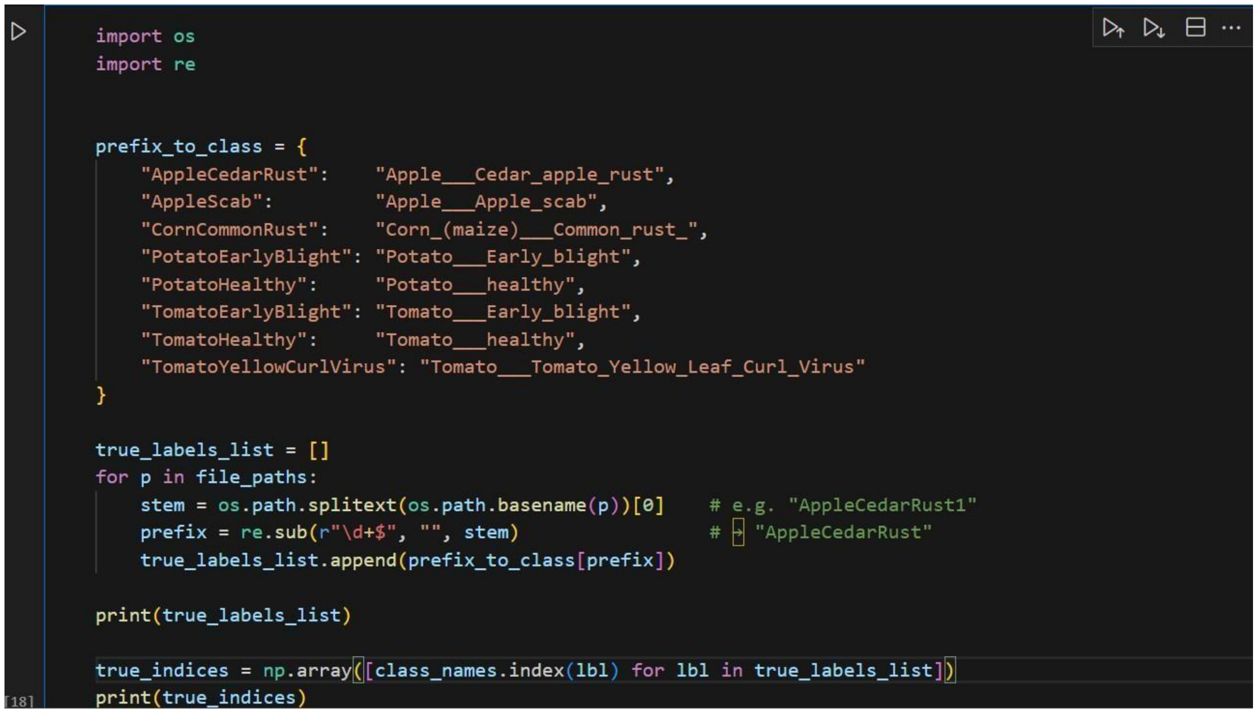
Each row predictions[i] is a 38-element probability vector for sample *i*.

np.argmax(..., axis=1) finds, for each row, the index of the highest probability Each

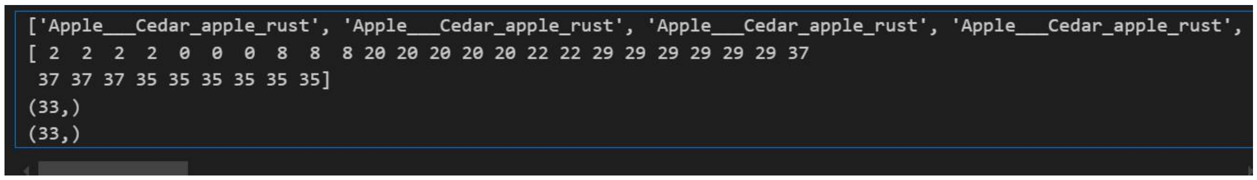
number is the predicted class index for that test image.

Why convert one-hot labels to integer class indices?

To make comparison with predicted labels possible (as predictions are class indices).



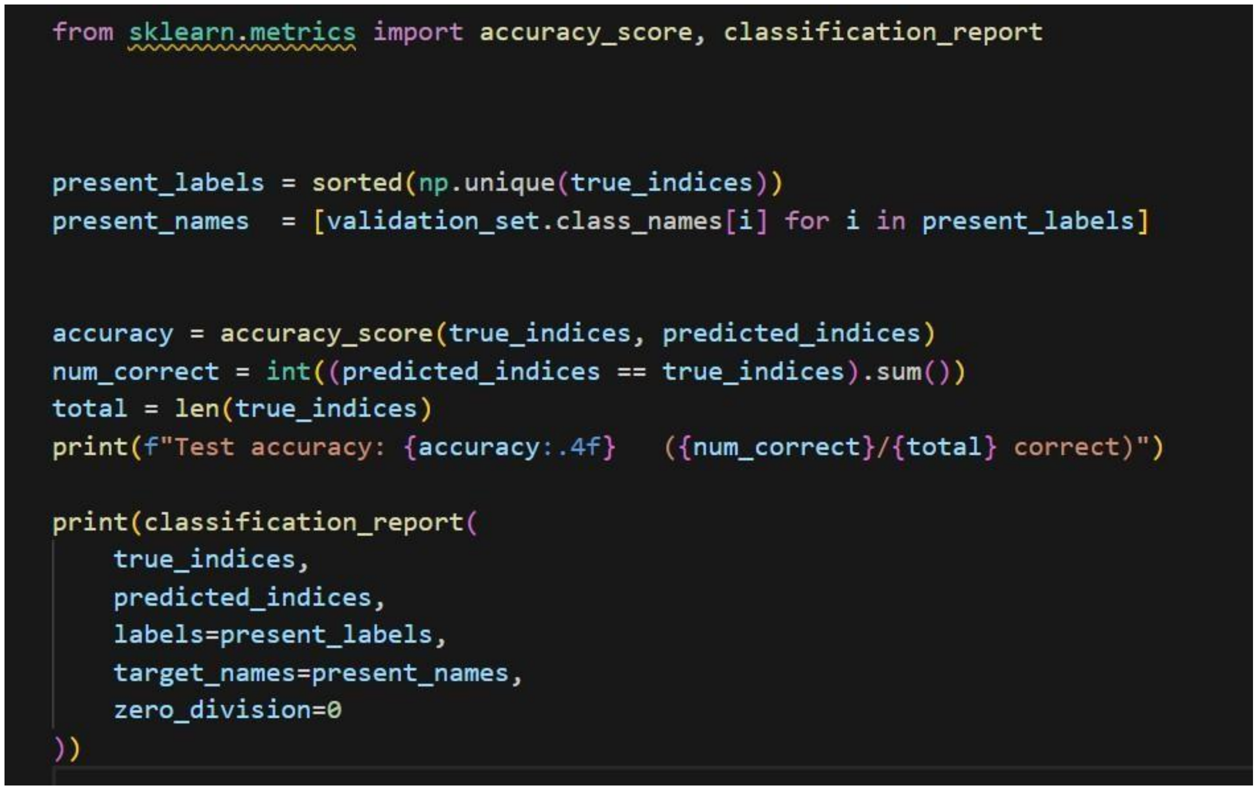
```python
import os
import re


prefix_to_class = {
    "AppleCedarRust":    "Apple___Cedar_apple_rust",
    "AppleScab":         "Apple___Apple_scab",
    "CornCommonRust":    "Corn_(maize)___Common_rust_",
    "PotatoEarlyBlight": "Potato___Early_blight",
    "PotatoHealthy":     "Potato___healthy",
    "TomatoEarlyBlight": "Tomato___Early_blight",
    "TomatoHealthy":     "Tomato___healthy",
    "TomatoYellowCurlVirus": "Tomato___Tomato_Yellow_Leaf_Curl_Virus"
}

true_labels_list = []
for p in file_paths:
    stem = os.path.splitext(os.path.basename(p))[0]    # e.g. "AppleCedarRust1"
    prefix = re.sub(r"\d+$", "", stem)                 # → "AppleCedarRust"
    true_labels_list.append(prefix_to_class[prefix])

print(true_labels_list)

true_indices = np.array([class_names.index(lbl) for lbl in true_labels_list])
print(true_indices)
```



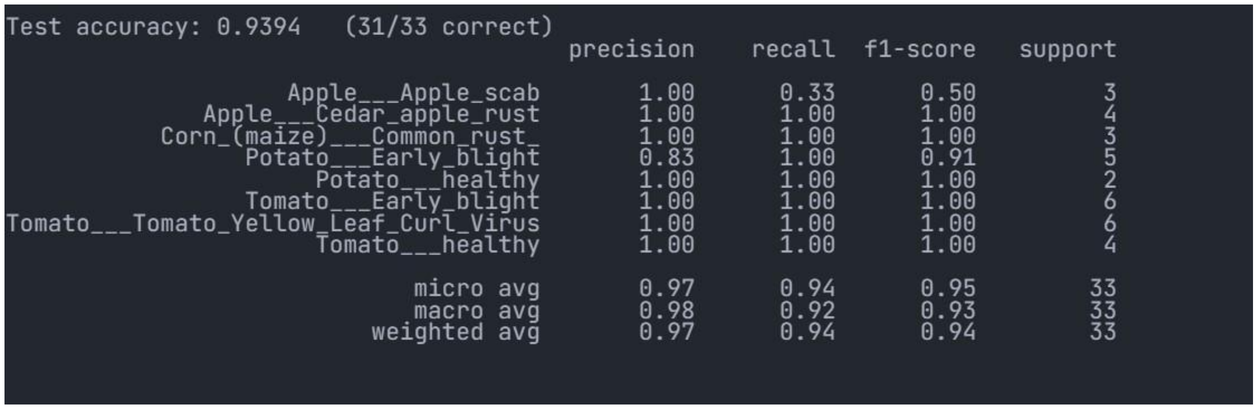
```
['Apple___Cedar_apple_rust', 'Apple___Cedar_apple_rust', 'Apple___Cedar_apple_rust', 'Apple___Cedar_apple_rust',
[ 2  2  2  2  0  0  0  8  8  8 20 20 20 20 20 22 22 29 29 29 29 29 29 37
 37 37 37 35 35 35 35 35 35]
(33,)
(33,)
```

Each test image file is named with a disease prefix (e.g. AppleCedarRust1.JPG). We defined a Python dictionary that maps these prefixes to the corresponding categorical labels used during training:

Convert class names to integer indices
Using the class_names list (sorted alphabetically) from our validation dataset, we mapped each true label to its integer index:



```python
from sklearn.metrics import accuracy_score, classification_report



present_labels = sorted(np.unique(true_indices))
present_names  = [validation_set.class_names[i] for i in present_labels]


accuracy = accuracy_score(true_indices, predicted_indices)
num_correct = int((predicted_indices == true_indices).sum())
total = len(true_indices)
print(f"Test accuracy: {accuracy:.4f}   ({num_correct}/{total} correct)")

print(classification_report(
    true_indices,
    predicted_indices,
    labels=present_labels,
    target_names=present_names,
    zero_division=0
))
```



```
Test accuracy: 0.9394   (31/33 correct)
                                        precision    recall  f1-score   support

                  Apple___Apple_scab       1.00      0.33      0.50         3
            Apple___Cedar_apple_rust       1.00      1.00      1.00         4
         Corn_(maize)___Common_rust_       1.00      1.00      1.00         3
               Potato___Early_blight       0.83      1.00      0.91         5
                    Potato___healthy       1.00      1.00      1.00         2
               Tomato___Early_blight       1.00      1.00      1.00         6
Tomato___Tomato_Yellow_Leaf_Curl_Virus       1.00      1.00      1.00         6
                    Tomato___healthy       1.00      1.00      1.00         4

                           micro avg       0.97      0.94      0.95        33
                           macro avg       0.98      0.92      0.93        33
                        weighted avg       0.97      0.94      0.94        33
```

Quantitative Evaluation on Test Set

To measure the model’s performance, we compared its predicted class indices against the ground-truth indices using standard classification metrics from scikit-learn:

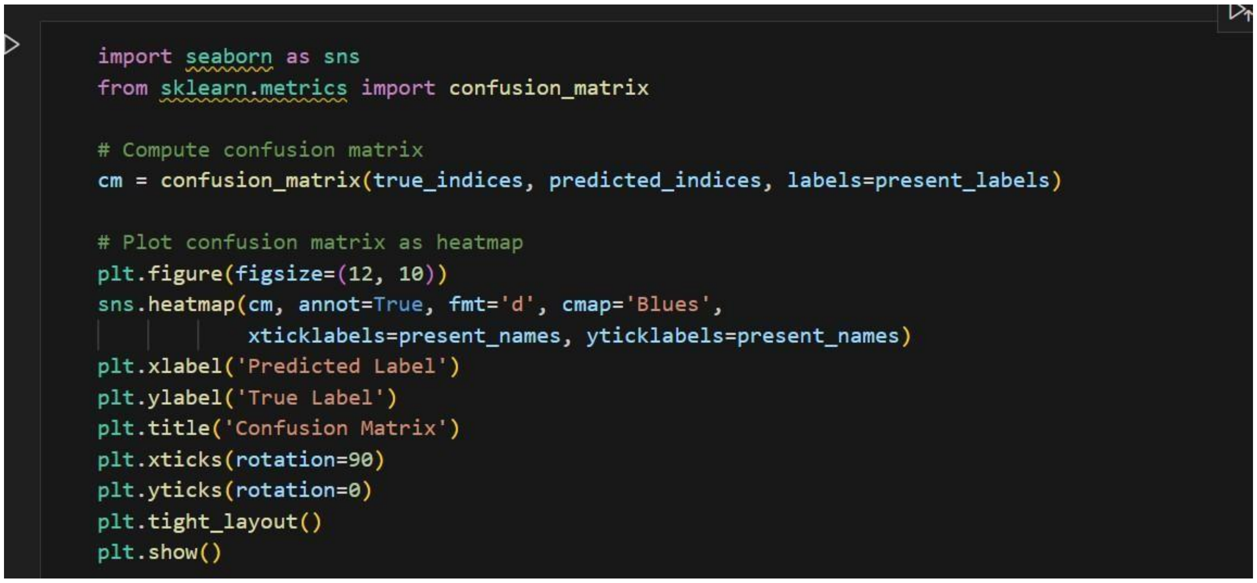
Overall accuracy: the fraction of test samples the model classified correctly, reported both as a percentage and in “correct/total” form.

Per-class metrics:

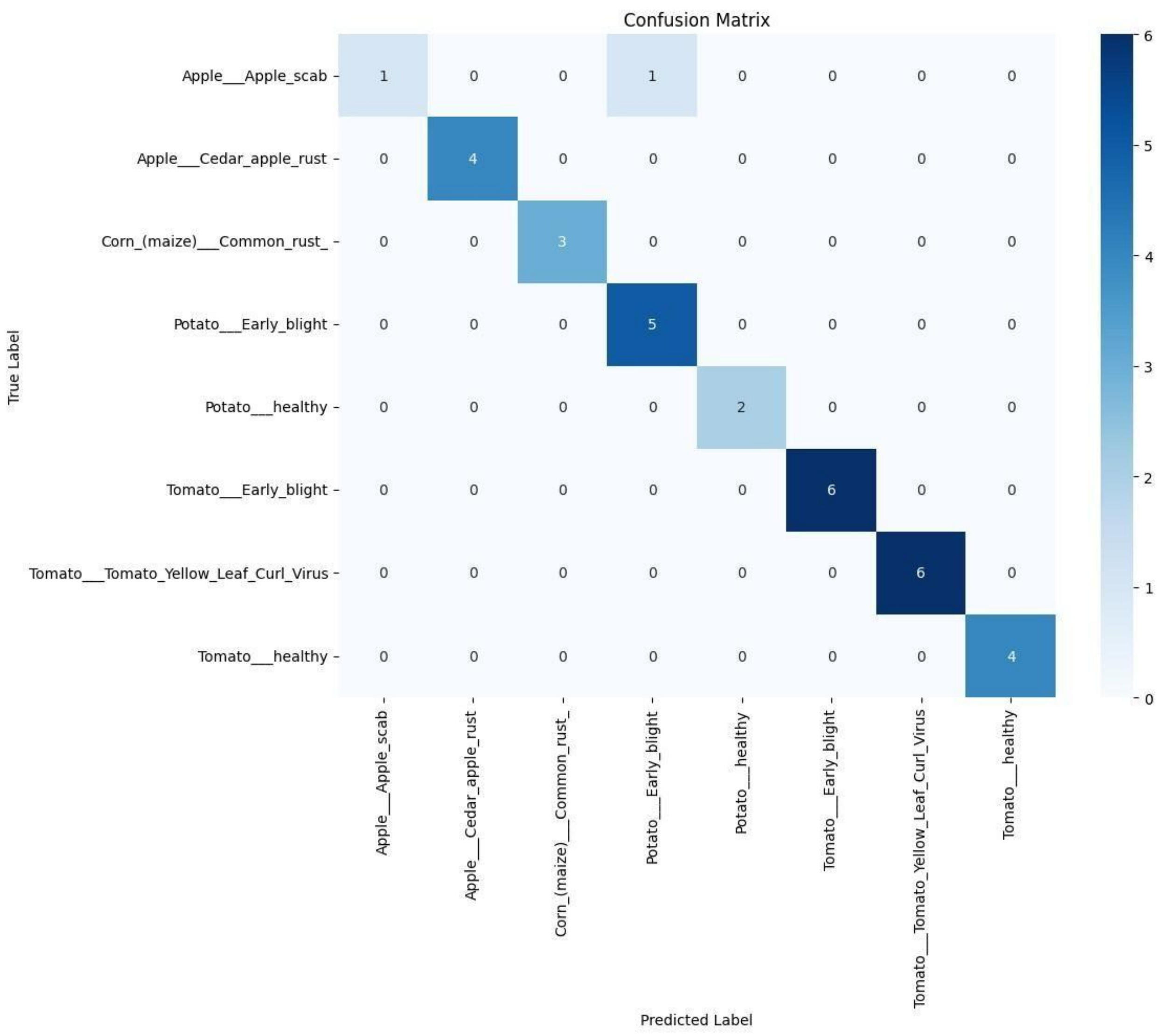
- **Precision**: of all images predicted as class *C*, the fraction that truly belong to *C*.

- **Recall**: of all images that truly are class *C*, the fraction the model correctly identified.
- **F1-score**: the harmonic mean of precision and recall, summarizing the model's accuracy on each category.
- **Support**: the number of test images available for each class.

Confusion Matrix:



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

# Compute confusion matrix
cm = confusion_matrix(true_indices, predicted_indices, labels=present_labels)

# Plot confusion matrix as heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=present_names, yticklabels=present_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



True \ Predicted	Apple___Apple_scab	Apple___Cedar_apple_rust	Corn_(maize)___Common_rust_	Potato___Early_blight	Potato___healthy	Tomato___Early_blight	Tomato___Tomato_Yellow_Leaf_Curl_Virus	Tomato___healthy
Apple___Apple_scab	1	0	0	1	0	0	0	0
Apple___Cedar_apple_rust	0	4	0	0	0	0	0	0
Corn_(maize)___Common_rust_	0	0	3	0	0	0	0	0
Potato___Early_blight	0	0	0	5	0	0	0	0
Potato___healthy	0	0	0	0	2	0	0	0
Tomato___Early_blight	0	0	0	0	0	6	0	0
Tomato___Tomato_Yellow_Leaf_Curl_Virus	0	0	0	0	0	0	6	0
Tomato___healthy	0	0	0	0	0	0	0	4

In this, we just calculate the confusion matrix using true and predicted labels, and plot it as a heatmap using Seaborn to clearly see where the model made correct or wrong predictions.

CNN Model Architecture

1. Input and Data-Augmentation Block

InputLayer: Receives 128×128 RGB images (shape (None, 128, 128, 3)).
Data-Augmentation Layers: RandomFlip, RandomRotation, RandomZoom. Introduce variability at training time (flips, rotations, zoom) to improve generalization and reduce overfitting.

Why Data Augmentation?

To create synthetic diversity by flipping, rotating, and zooming images.
Why? Increases robustness and prevents the model from overfitting to specific patterns in training data.

2. Convolutional Feature Extractor

Core of the network: five convolutional "blocks", each doubling filters and followed by spatial downsampling.

Block	Conv Layers	Filters	Output Spatial Size	Notes
1	Conv2D→BN→ReLU → Conv2D→BN→ReLU → MaxPool2D	32	128×128 → 63×63	Capture edges
2	Conv2D→BN→ReLU → Conv2D→BN→ReLU → MaxPool2D	64	63×63 → 30×30	Detect simple textures
3	Conv2D→BN→ReLU → Conv2D→BN→ReLU → MaxPool2D	128	30×30 → 14×14	Learn mid-level patterns
4	Conv2D→BN→ReLU → Conv2D→BN→ReLU → MaxPool2D	256	14×14 → 6×6	Capture complex shapes
5	Conv2D→BN→ReLU → Conv2D→BN→ReLU → MaxPool2D	512	6×6 → 2×2	High-level feature abstraction

3. Regularization

-Dropout: Applies after final pooling (rate as configured) to randomly zero-out activations, reducing overfitting by discouraging co-adaptation.

4. Classification Head

1. Flatten: Converts 2×2×512 tensor into a 2048-d vector.
2. Dense (1500 units) → BN → ReLU → Dropout: High-capacity layer to mix extracted features.
3. Dense (38 units): Outputs logits for each class; softmax applied in loss computation.

Future Improvements:

Expand Dataset Diversity: Incorporate more field-captured images across different geographies, lighting conditions, and leaf orientations.

Lightweight Architectures: Experiment with MobileNetV3 or EfficientNet-Lite for ondevice/mobile deployment.

Real-Time Deployment: Build a lightweight mobile/web app with a user-friendly interface for in-field diagnosis.

Continuous Learning: Implement online learning or periodic retraining pipelines to incorporate newly labeled data and adapt to evolving disease patterns.

Model Design: Why These Components?

- **Conv2D Layers:** Extract hierarchical features.
- **MaxPooling:** Reduces dimensionality and computation.
  Why? Keeps key information while lowering model complexity.
- **ReLU Activation:** Introduces non-linearity. **Why?** Enables learning complex patterns.
- **Dropout:** Prevents overfitting.
  Why? Helps the model generalize better to new data.
- **Softmax Output:** Converts logits to probability scores. **Why?** Required for multi-class classification.

Why This Architecture?

A 5-block CNN progressively extracts more complex features.
Why? Deeper layers allow the model to detect intricate details of leaf diseases, improving classification accuracy.