Open in Colab

Module Name: IT4010 - Research Project

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

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

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 chili peppers, specifically athacnose peppers and red peppers, and determine if the pepper is healthy.

Methodology

Supervised Learning

This project leverages supervised learning where the model is trained on labeled data—images of chili pepper categorized as Red peppers, Anthacnose Peppers, 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 Chili Pepper Qulaity Idenitify Dataset, which consists of labeled images of chili peppers from three categories,

Red (Non-productive) Chili Peppers: that are not used for production level.

Anthracnose Disease Chili Peppers: that have disease in chili pepper which are not used for production.

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

Link: https://www.kaggle.com/datasets/prudhvi143413s/anthracnose-disease-in-chili-palnadu-ap

Import all the Dependencies

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

Set all the Constants

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

Import data into tensorflow dataset object

```
import os
import tensorflow as tf

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

```
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 1217 files belonging to 3 classes.
```

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

```
['chilli_anthacnose', 'chilli_healthy', 'chilli_red']
```

```
len(dataset)
```

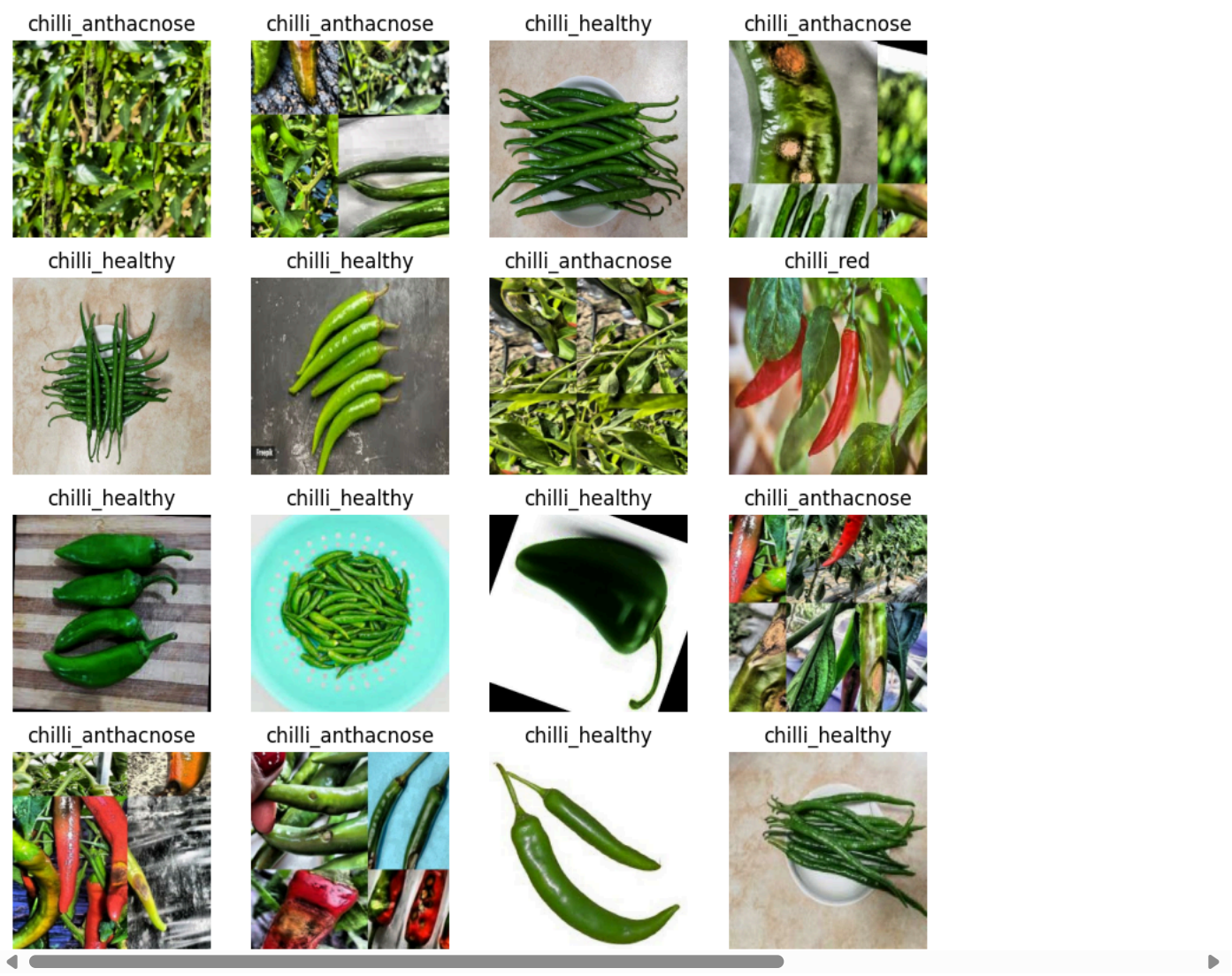
```
39
```

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

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

```
39
```

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

```
31.200000000000003
```

```
train_ds = dataset.take(31)
len(train_ds)
```

```
31
```

```
test_ds = dataset.skip(31)
len(test_ds)
```

```
8
```

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

```
3.9000000000000004
```

```
val_ds = test_ds.take(3)
len(val_ds)
```

```
3
```

```
test_ds = test_ds.skip(3)
len(test_ds)
```

```
0
```

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

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

    ds_size = len(ds)

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

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

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

    return train_ds, val_ds, test_ds
```

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

```
len(train_ds)
```

```
31
```

```
len(val_ds)
```

```
3
```

```
len(test_ds)
```

```
5
```

IT21249570 - Model 01

Dataset Caching and Prefetching

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

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

IT21249570 - Building the Model

Creating a Layer for Resizing and Normalization

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

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

Data Augmentation

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

```
import tensorflow as tf

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

Applying Data Augmentation to Train Dataset

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

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

IT21249570 - Model Architecture

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

First Conv2D Layer

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

Second Conv2D Layer

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

Third Conv2D Layer

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

Fourth Conv2D Layer

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

Fifth Conv2D Layer

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

Sixth Conv2D Layer

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

Flatten Layer

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

First Dense Layer

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

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

model = models.Sequential([
    layers.Lambda(lambda x: resize_and_rescale(x), input_shape=input_shape),
    layers.Conv2D(32, 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, 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'),
])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/lambda_layer.py:65: UserWarning: Do not pass an `input_shape`/`input_dim`
  super().__init__(**kwargs)
```

```
model.summary()
```

Model: "sequential_7"

Layer (type)	Output Shape	Param #
lambda (Lambda)	(None, 256, 256, 3)	0
conv2d_30 (Conv2D)	(None, 254, 254, 32)	896
max_pooling2d_30 (MaxPooling2D)	(None, 127, 127, 32)	0
conv2d_31 (Conv2D)	(None, 125, 125, 64)	18,496
max_pooling2d_31 (MaxPooling2D)	(None, 62, 62, 64)	0
conv2d_32 (Conv2D)	(None, 60, 60, 64)	36,928
max_pooling2d_32 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_33 (Conv2D)	(None, 28, 28, 64)	36,928
max_pooling2d_33 (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_34 (Conv2D)	(None, 12, 12, 64)	36,928
max_pooling2d_34 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_35 (Conv2D)	(None, 4, 4, 64)	36,928
max_pooling2d_35 (MaxPooling2D)	(None, 2, 2, 64)	0
flatten_5 (Flatten)	(None, 256)	0
dense_10 (Dense)	(None, 64)	16,448
dense_11 (Dense)	(None, 3)	195

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

IT21249570 - Compiling the Model

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

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

```
    metrics=['accuracy']
)
```

Model Training

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

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

```
Epoch 1/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 184s 653ms/step - accuracy: 0.4641 - loss: 1.0056 - val_accuracy: 0.8229 - val_loss: 0.4399
Epoch 2/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 359ms/step - accuracy: 0.8344 - loss: 0.4241 - val_accuracy: 0.8333 - val_loss: 0.4239
Epoch 3/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 357ms/step - accuracy: 0.8750 - loss: 0.3409 - val_accuracy: 0.9375 - val_loss: 0.1779
Epoch 4/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 20s 342ms/step - accuracy: 0.9403 - loss: 0.1796 - val_accuracy: 0.9167 - val_loss: 0.1876
Epoch 5/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 360ms/step - accuracy: 0.9474 - loss: 0.1737 - val_accuracy: 0.9688 - val_loss: 0.1114
Epoch 6/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 355ms/step - accuracy: 0.9705 - loss: 0.0920 - val_accuracy: 0.9375 - val_loss: 0.1331
Epoch 7/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 355ms/step - accuracy: 0.9688 - loss: 0.0777 - val_accuracy: 0.9688 - val_loss: 0.1195
Epoch 8/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 21s 358ms/step - accuracy: 0.9686 - loss: 0.0751 - val_accuracy: 0.9583 - val_loss: 0.1071
Epoch 9/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 357ms/step - accuracy: 0.9713 - loss: 0.0859 - val_accuracy: 0.9479 - val_loss: 0.1372
Epoch 10/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 349ms/step - accuracy: 0.9784 - loss: 0.0502 - val_accuracy: 0.9479 - val_loss: 0.0702
Epoch 11/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 342ms/step - accuracy: 0.9757 - loss: 0.0789 - val_accuracy: 0.9583 - val_loss: 0.0833
Epoch 12/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 21s 354ms/step - accuracy: 0.9721 - loss: 0.0831 - val_accuracy: 0.9792 - val_loss: 0.0471
Epoch 13/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 355ms/step - accuracy: 0.9777 - loss: 0.0653 - val_accuracy: 0.9583 - val_loss: 0.0789
Epoch 14/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 357ms/step - accuracy: 0.9752 - loss: 0.0644 - val_accuracy: 0.9688 - val_loss: 0.1567
Epoch 15/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 10s 334ms/step - accuracy: 0.9797 - loss: 0.0845 - val_accuracy: 0.9896 - val_loss: 0.0543
Epoch 16/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 21s 354ms/step - accuracy: 0.9752 - loss: 0.0528 - val_accuracy: 0.9688 - val_loss: 0.0568
Epoch 17/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 12s 401ms/step - accuracy: 0.9833 - loss: 0.0484 - val_accuracy: 0.9792 - val_loss: 0.0329
Epoch 18/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 353ms/step - accuracy: 0.9831 - loss: 0.0555 - val_accuracy: 1.0000 - val_loss: 0.0298
Epoch 19/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 10s 312ms/step - accuracy: 0.9889 - loss: 0.0303 - val_accuracy: 1.0000 - val_loss: 0.0084
Epoch 20/20
31/31 ━━━━━━━━━━━━━━━━━━━━ 11s 352ms/step - accuracy: 0.9769 - loss: 0.0718 - val_accuracy: 0.9688 - val_loss: 0.0548
```

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

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 4s 22ms/step - accuracy: 0.9610 - loss: 0.1109
```

```
scores
```

```
[0.0880059078335762, 0.9624999761581421]
```

Plotting the Accuracy and Loss Curves

```
history
```

```
<keras.src.callbacks.history.History at 0x7977902823b0>
```

```
history.params
```

```
{'verbose': 1, 'epochs': 20, 'steps': 31}
```

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

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

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

```
list
```

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

```
20
```

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

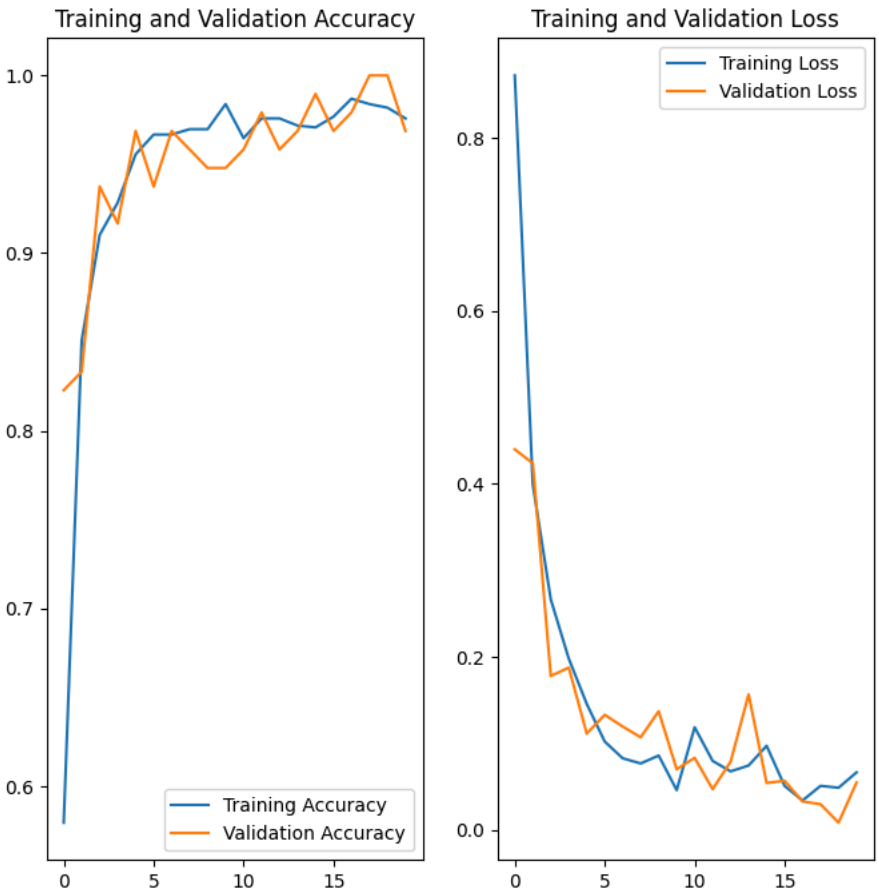
```
[0.8721910119056702,
 0.39965662360191345,
 0.2665603756904602,
 0.19808222353458405,
 0.145667165517807]
```

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

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

Plotting Training and Validation Accuracy

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

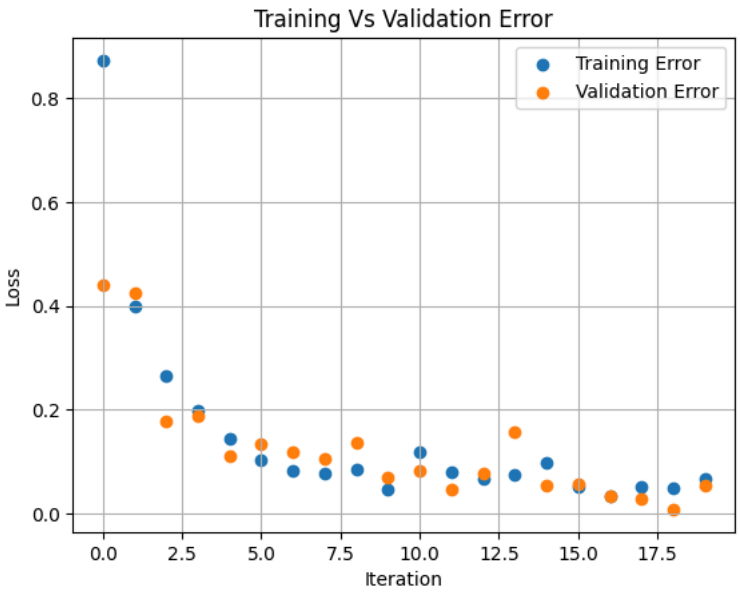
```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

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



Scatter Plot: Training vs Validation Error

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

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

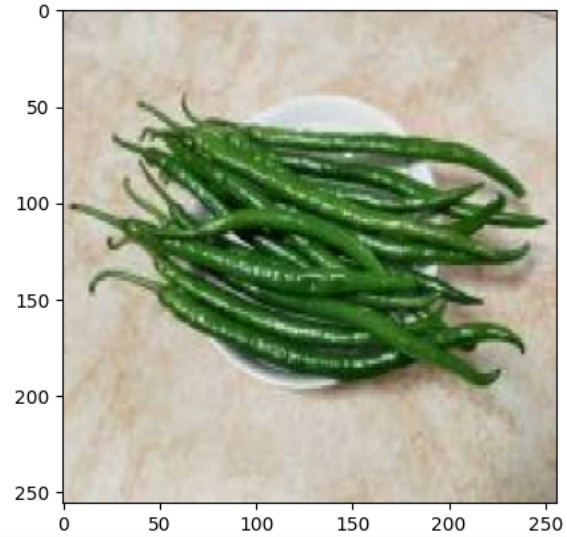


Run prediction on a sample image

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

```
import numpy as np
import matplotlib.pyplot as plt

# Define the suitability mapping
suitability_mapping = {
    "chilli_healthy": "Healthy chili peppers.These peppers are in optimal condition and meet the standards for agricultural production or pr
    "chilli_red": "Overripe or non-productive red chili peppers.These peppers are unsuitable for production due to their overripe state, whi
    "chilli_anthracnose": "Chili peppers affected by anthracnose disease.These are unsuitable due to fungal infection (anthracnose), which a
}

for images_batch, labels_batch in test_ds.take(1):
    first_image = images_batch[1].numpy().astype('uint8')
    first_label = labels_batch[1].numpy()

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

    batch_prediction = model.predict(images_batch)
    predicted_label = class_names[np.argmax(batch_prediction[1])]

    production_status = suitability_mapping.get(predicted_label, "Unknown")


    print("Predicted label:", predicted_label)
    print("Production suitability:", production_status)
```

```
First image to predict
Actual label: chilli_healthy
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
Predicted label: chilli_healthy
Production suitability: Healthy chili peppers.These peppers are in optimal condition and meet the standards for agricultural production
```



Write a function for inference

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

    predictions = model.predict(img_array)

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

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

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

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

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

        plt.axis("off")
```

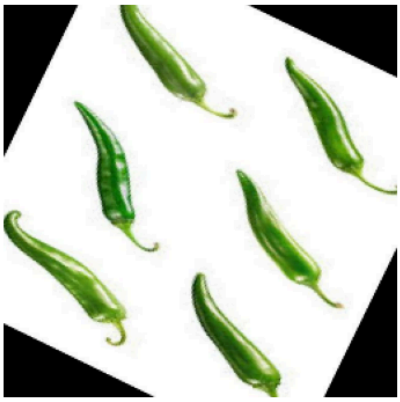
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

Actual: chilli_healthy,
Predicted: chilli_anthacnose.
Confidence: 99.12%

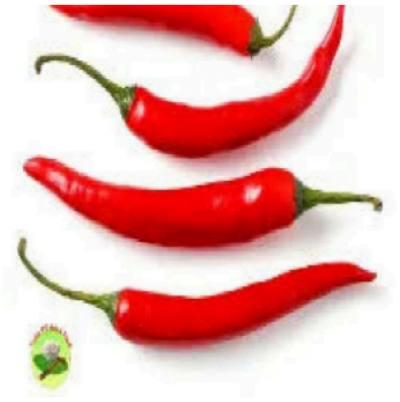
Actual: chilli_anthacnose,
Predicted: chilli_anthacnose.
Confidence: 99.87%

Actual: chilli_red,
Predicted: chilli_red.
Confidence: 99.99%

Actual: chilli_healthy,
Predicted: chilli_healthy.
Confidence: 99.41%



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



Actual: chilli_healthy,
Predicted: chilli_healthy.
Confidence: 99.54%

Actual: chilli_anthacnose,
Predicted: chilli_anthacnose.
Confidence: 99.99%

Actual: chilli_anthacnose,
Predicted: chilli_anthacnose.
Confidence: 99.95%

Actual: chilli_red,
Predicted: chilli_red.
Confidence: 99.99%

IT21249570 - Saving the Model

```
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("../chili_pepper_trained.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())
```