Garbage_and_Waste_Classification

October 23, 2023

1 Garbage and Waste Classifier Using DL

1.1 Problem Statement:

The "Garbage and Waste Classifier Using Deep Learning" project aims to address the pressing issue of improper waste management and recycling by developing an accurate and efficient deep learning system that can classify various types of waste and garbage items.

1.2 Problem Description:

Improper disposal and mismanagement of waste and garbage have detrimental effects on the environment, public health, and sustainability. Inadequate waste sorting and recycling practices contribute to pollution, landfill overflow, and resource depletion. To mitigate these problems, there is a critical need for a highly accurate and automated system that can identify and classify different types of waste and garbage items.

The objective of this project is to create a robust deep learning-based classification system that can accurately categorize waste and garbage items into various classes such as plastics, glass, paper, organic waste, metals, and more. The system will be capable of distinguishing between recyclable and non-recyclable materials, as well as identifying hazardous waste items that require special disposal methods.

Key Challenges:

1. **Variability in Waste Types:** Garbage and waste items come in a wide range of shapes, sizes, colors, and materials. Developing a classifier that can handle this variability is a significant challenge.
2. **Data Collection and Labeling:** Gathering a diverse and extensive dataset of waste items with accurate labels is crucial for training the deep learning model.
3. **Real-time Classification:** The system should be capable of classifying waste items in real-time or with minimal latency to be practical for use in waste sorting facilities or mobile applications.
4. **Environmental Conditions:** The system should be robust to environmental conditions such as varying lighting, humidity, and occlusions that are typically encountered in waste sorting facilities.
5. **Cost-Effective Hardware:** Designing an affordable hardware setup, possibly utilizing cameras and sensors, to capture waste item images is vital for practical implementation.

Significance:

The successful development of a "Garbage and Waste Classifier Using Deep Learning" system has the potential to significantly improve waste management practices. It can enhance the efficiency of waste sorting and recycling facilities, promote environmentally responsible behaviors, and contribute to a cleaner and more sustainable environment.

This project will not only have immediate practical applications in waste management but also raise awareness about the importance of proper waste disposal and recycling, ultimately leading to a more sustainable and eco-friendly society.

2 Importing Libraries

[ ]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import cv2
import sklearn
import tensorflow as tf
from tensorflow import keras
from PIL import Image as im
from glob import glob
from sklearn.model_selection import train_test_split
import keras
#from tf.keras.utils import to_categorical
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.models import Model
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from PIL import Image as im
import cv2
import tensorflow as tf
```

The code below provided is a function called seed_everything(), which sets the seeds for the random number generators in Python, NumPy, TensorFlow, and the operating system. This is done to make the experiment more reproducible, meaning that if you run the code again with the same seed, we will get the same results

[ ]:
```python
# Set seeds to make the experiment more reproducible.
import random
```

```
def seed_everything(seed = 0):
    """Sets the seeds for the random number generators in Python, NumPy,␣
 ↪TensorFlow, and the operating system.

    Args:
        seed: The seed to use.
    """

    # Set the Python random seed.
    random.seed(seed)

    # Set the NumPy random seed.
    np.random.seed(seed)

    # Set the TensorFlow random seed.
    tf.random.set_seed(seed)

    # Set the operating system random seed.
    os.environ['PYTHONHASHSEED'] = str(seed)

# Set the seed to 0.
seed = 0

# Call the seed_everything() function with the seed.
seed_everything(seed)
```

3 Loading Datasets

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

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

[ ]:
```
import os

data_path="/content/drive/MyDrive/Colab Notebooks/DS_PROJECT/
 ↪Waste-Classifier-WebApp/Data/image"
os.listdir(data_path)
```

[ ]:
```
['train', 'test']
```

[ ]:
```
input_path=data_path
train_data_dir=input_path + '/' + 'train'
test_data_dir=input_path + '/' + 'test'
```

[ ]:
```
BATCH_SIZE=64
img_height=244
img_width=244
```

preparing for training datasets

[ ]:
```
train_ds=tf.keras.utils.image_dataset_from_directory(
    train_data_dir,validation_split=0.2,
    subset='training',label_mode='categorical',
    image_size=(img_height,img_width),
    batch_size=BATCH_SIZE,
    seed=123
)
```

```
Found 2187 files belonging to 7 classes.
Using 1750 files for training.
```

[ ]:
```
validation_ds = tf.keras.utils.image_dataset_from_directory(
    train_data_dir,
    validation_split = 0.2,
    subset = 'validation',
    label_mode = 'categorical',
    image_size = (img_height, img_width),
    batch_size = BATCH_SIZE,
    seed = 123
  )
```

```
Found 2187 files belonging to 7 classes.
Using 437 files for validation.
```

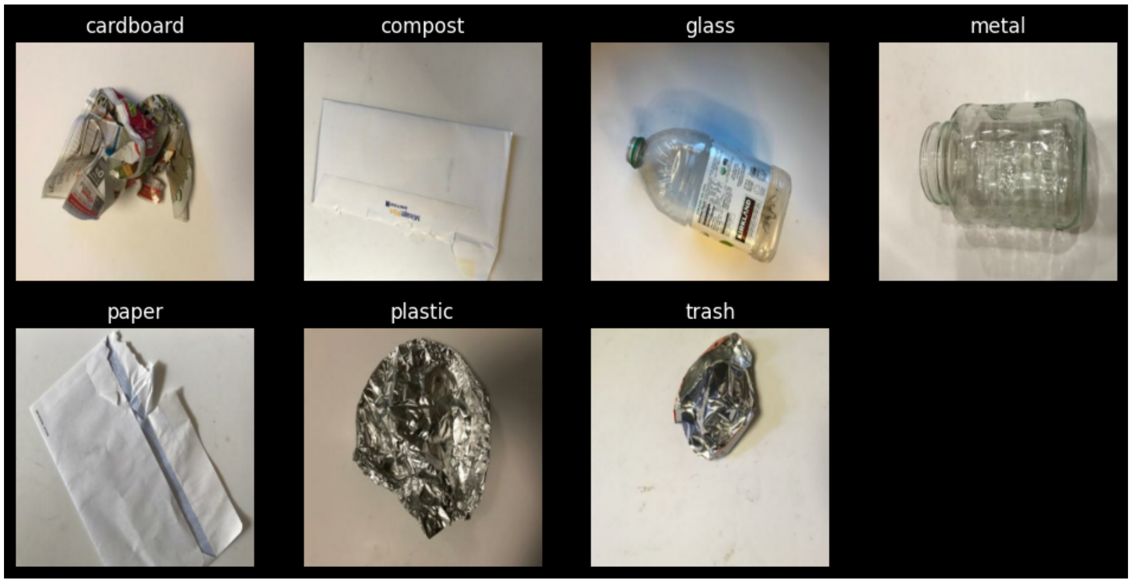
[ ]:
```
class_names=train_ds.class_names
print(class_names)
```

```
['cardboard', 'compost', 'glass', 'metal', 'paper', 'plastic', 'trash']
```

4 Data Visualization

[ ]:
```
plt.style.use('dark_background')
```

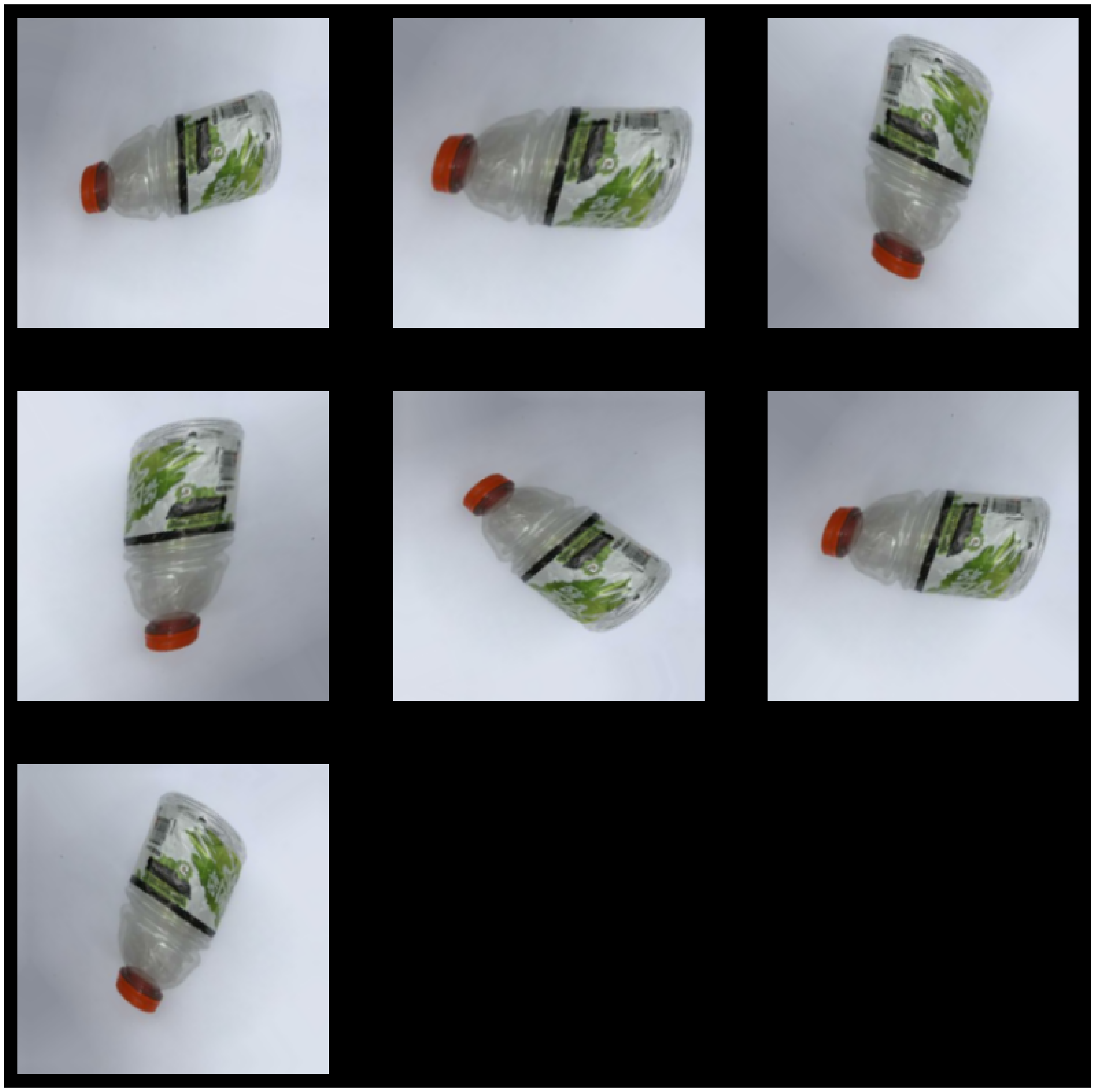
[ ]:
```
plt.figure(figsize=(12,12))
for img,label in train_ds.take(1):
  for i in range(len(class_names)):
    ax=plt.subplot(4,4,i+1)
    plt.imshow(img[i].numpy().astype('uint8'))
    plt.title(class_names[i])
    plt.axis('off')
```



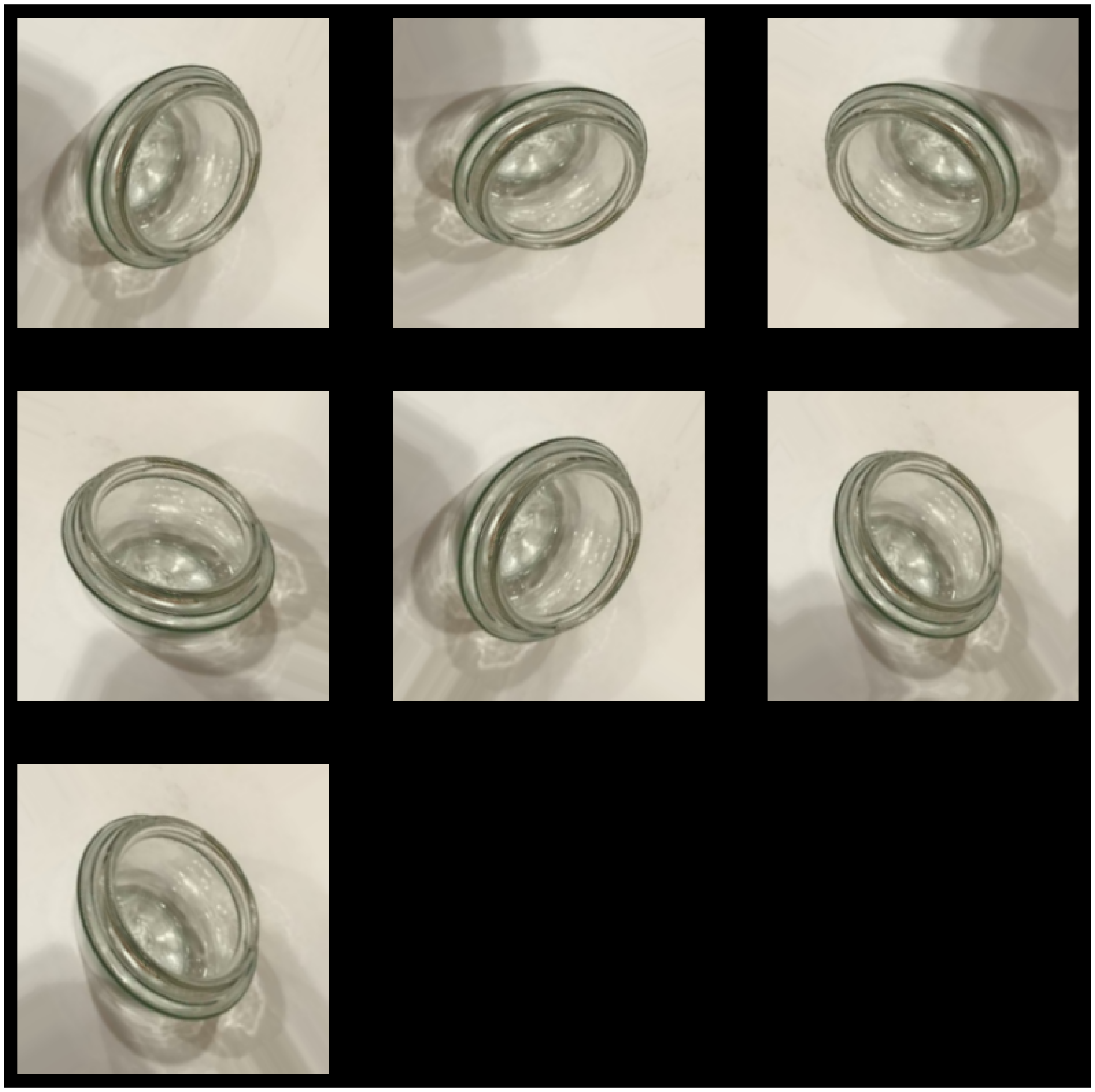
The code below provided is a sequence of Keras layers that can be used for image data augmentation. Data augmentation is a technique that can be used to increase the size and diversity of a training dataset by creating new images from existing images. This can help to improve the performance of machine learning models, especially when the training dataset is small.

```
[ ]: data_augmentation = keras.Sequential([
         #layers.CenterCrop(125, 125),
         tf.keras.layers.RandomFlip('horizontal', input_shape = (img_height,␣
      ↪img_width, 3)),
         tf.keras.layers.RandomRotation(0.2, fill_mode = 'nearest'),
         tf.keras.layers.RandomZoom(0.1),
     ])
```

```
[ ]: plt.figure(figsize=(10, 10))
     for images, _ in train_ds.take(1):
       for i in range(len(class_names)):
         augmented_images = data_augmentation(images)
         ax = plt.subplot(3, 3, i + 1)
         plt.imshow(augmented_images[0].numpy().astype("uint8"))
        # plt.title(class_names[2])
         plt.axis("off")
```



```
[ ]: plt.figure(figsize=(10, 10))
     for images, _ in train_ds.take(1):
       for i in range(len(class_names)):
         augmented_images = data_augmentation(images)
         ax = plt.subplot(3, 3, i + 1)
         plt.imshow(augmented_images[0].numpy().astype("uint8"))
        # plt.title(class_names[2])
         plt.axis("off")
```



The code below provided is used to prefetch the training and validation datasets. Prefetching is a technique that can be used to improve the performance of machine learning models by loading data into memory in advance. This can help to reduce the time it takes to train the model, especially when the dataset is large.

[ ]:
```
AUTOTUNE = tf.data.AUTOTUNE

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

5 Model Architecture

5.1 VGG-16 Base Model

VGG-16, short for "Visual Geometry Group 16-layer," is a convolutional neural network (CNN) architecture designed for image classification. It was developed by the Visual Geometry Group at the University of Oxford and is known for its simplicity and effectiveness. VGG-16 is part of the VGG family of models, which includes various configurations with different numbers of layers, but VGG-16 specifically consists of 16 weight layers, including 13 convolutional layers and 3 fully connected layers.

Key characteristics of the VGG-16 architecture:

1. **Architecture:** VGG-16 comprises a stack of convolutional layers, followed by max-pooling layers, and concludes with fully connected layers. The network architecture is relatively deep, and it uses 3x3 convolutional filters with a stride of 1 and 2x2 max-pooling with a stride of 2.
2. **Uniform Convolutional Layers:** One notable feature of VGG-16 is its uniform use of 3x3 convolutional filters. The repeated use of these small filters allows the network to learn a wide range of image features at different scales.
3. **Depth:** VGG-16 is considered deep for its time (2014), with 13 convolutional layers. The depth of the network aids in capturing complex hierarchical features.
4. **Fully Connected Layers:** The convolutional layers are followed by three fully connected layers, with the last fully connected layer producing the final classification output.
5. **ReLU Activation Function:** Rectified Linear Units (ReLU) are used as activation functions throughout the network, promoting faster convergence during training.

VGG-16 was trained on the ImageNet dataset, a large dataset with millions of labeled images from thousands of categories. As a result, it achieved state-of-the-art performance on various image classification tasks at the time of its introduction.

While VGG-16 is a powerful architecture for image classification, it has a relatively high number of parameters, making it computationally expensive to train and deploy. More recent CNN architectures, like those in the ResNet and Inception families, have addressed some of the limitations of VGG by introducing skip connections and more efficient architectures. However, VGG-16 remains a valuable reference point and is often used as a base model for transfer learning in computer vision tasks. Researchers and practitioners may take the pre-trained VGG-16 model and fine-tune it on their specific image classification tasks to leverage its learned features.

[ ]:
```
# from PIL import Image
# from IPython.display import display
# img1=Image.open("Data/VGG_16_Architecture.png")
# img1=img1.convert('RGB')
# display(img1)
```

5.2 Architecture and Working..

The VGG-16 architecture is a widely recognized convolutional neural network (CNN) designed for image classification. It was introduced by the Visual Geometry Group (VGG) at the University of

Oxford and is known for its simplicity and effectiveness. Here's an overview of its architecture and how it works:

Architecture: The VGG-16 architecture consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. It is characterized by its uniform use of small 3x3 convolutional filters and 2x2 max-pooling layers. The network progressively reduces spatial dimensions and increases the depth as you move through the layers. Below is a breakdown of the architecture:

1. **Input Layer:** VGG-16 takes an RGB image as input with a fixed size, typically 224x224 pixels.
2. **Convolutional Layers:** The initial layers consist of two or more stacked 3x3 convolutional layers, followed by a 2x2 max-pooling layer to reduce the spatial dimensions. These layers are designed to capture low-level features such as edges and simple textures.
3. **Intermediate Convolutional Layers:** VGG-16 repeats this pattern of 3x3 convolutional layers and max-pooling layers multiple times, increasing the depth and allowing the network to learn more complex and abstract features.
4. **Fully Connected Layers:** After the convolutional layers, there are three fully connected layers. These layers perform high-level feature extraction and produce the final classification output. The last fully connected layer typically has as many neurons as there are classes in the classification task, and it is activated using the softmax function to produce class probabilities.
5. **Softmax Layer:** The final layer of the network is a softmax layer that assigns class probabilities to the input image based on the output of the previous fully connected layer.

Working: The VGG-16 network works by passing the input image through the layers sequentially, extracting and transforming features at each layer. Here's how it operates:

1. **Forward Propagation:** When an image is fed into the VGG-16 network, it undergoes a series of convolution operations, followed by activation functions (typically ReLU), and max-pooling operations. This process continues through the convolutional layers, gradually reducing the spatial dimensions while increasing the depth.
2. **Feature Extraction:** The convolutional layers learn to recognize various features, starting with simple shapes and edges in the early layers and progressing to more complex patterns and textures in the deeper layers.
3. **Flattening:** After the convolutional layers, the feature maps are flattened into a 1D vector, which is then passed through the fully connected layers.
4. **Classification:** The fully connected layers perform high-level feature extraction and classification. The output of the last fully connected layer is used to produce class probabilities through the softmax activation function.
5. **Prediction:** The class with the highest probability in the softmax output is considered the predicted class for the input image.

Training a VGG-16 model typically involves using a large labeled dataset, such as ImageNet, and backpropagation to adjust the model's weights and biases to minimize the classification error. Once trained, the model can be used for image classification tasks. Additionally, it's often employed as a base model for transfer learning, where the pre-trained VGG-16 model is fine-tuned on a specific dataset for a particular image classification task, leveraging the learned features from the earlier layers.

```
[ ]: IMG_SHAPE=(img_height,img_width,3)

     base_model=tf.keras.applications.VGG16(
         input_shape=IMG_SHAPE,
         include_top=False,
         weights='imagenet'
     )

     base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 244, 244, 3)]     0

 block1_conv1 (Conv2D)       (None, 244, 244, 64)      1792

 block1_conv2 (Conv2D)       (None, 244, 244, 64)      36928

 block1_pool (MaxPooling2D)  (None, 122, 122, 64)      0

 block2_conv1 (Conv2D)       (None, 122, 122, 128)     73856

 block2_conv2 (Conv2D)       (None, 122, 122, 128)     147584

 block2_pool (MaxPooling2D)  (None, 61, 61, 128)       0

 block3_conv1 (Conv2D)       (None, 61, 61, 256)       295168

 block3_conv2 (Conv2D)       (None, 61, 61, 256)       590080

 block3_conv3 (Conv2D)       (None, 61, 61, 256)       590080

 block3_pool (MaxPooling2D)  (None, 30, 30, 256)       0

 block4_conv1 (Conv2D)       (None, 30, 30, 512)       1180160

 block4_conv2 (Conv2D)       (None, 30, 30, 512)       2359808

 block4_conv3 (Conv2D)       (None, 30, 30, 512)       2359808
 block4_pool (MaxPooling2D)  (None, 15, 15, 512)       0

 block5_conv1 (Conv2D)       (None, 15, 15, 512)       2359808

 block5_conv2 (Conv2D)       (None, 15, 15, 512)       2359808

 block5_conv3 (Conv2D)       (None, 15, 15, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 14714688 (56.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[ ]:
```
def print_layer_trainable():
  for l in base_model.layers:
    print("{0}:\t==>\t{1}".format(l.trainable,l.name))
```

[ ]:
```
print_layer_trainable()
```

```
True:   ==>     input_1
True:   ==>     block1_conv1
True:   ==>     block1_conv2
True:   ==>     block1_pool
True:   ==>     block2_conv1
True:   ==>     block2_conv2
True:   ==>     block2_pool
True:   ==>     block3_conv1
True:   ==>     block3_conv2
True:   ==>     block3_conv3
True:   ==>     block3_pool
True:   ==>     block4_conv1
True:   ==>     block4_conv2
True:   ==>     block4_conv3
True:   ==>     block4_pool
True:   ==>     block5_conv1
True:   ==>     block5_conv2
True:   ==>     block5_conv3
True:   ==>     block5_pool
```

5.2.1 Fix the weights

[ ]:
```
base_model.trainable=False
```

[ ]:
```
for layer in base_model.layers:
  layer.trainable=False
```

[ ]:
```
print_layer_trainable()
```

```
False:  ==>     input_1
False:  ==>     block1_conv1
False:  ==>     block1_conv2
False:  ==>     block1_pool
False:  ==>     block2_conv1
False:  ==>     block2_conv2
False:  ==>     block2_pool
False:  ==>     block3_conv1
False:  ==>     block3_conv2
False:  ==>     block3_conv3
False:  ==>     block3_pool
False:  ==>     block4_conv1
False:  ==>     block4_conv2
False:  ==>     block4_conv3
False:  ==>     block4_pool
False:  ==>     block5_conv1
False:  ==>     block5_conv2
False:  ==>     block5_conv3
False:  ==>     block5_pool
```

[ ]:
```
# Number of classes in the dataset
n_classes = len(class_names)

# Define the sequential model
model = Sequential([

    # Apply random transformations to the input images
    data_augmentation,

    # Normalize the input images to the range [0, 1]
    tf.keras.layers.Rescaling(1./255),

    # Use a pre-trained model as a starting point
    base_model,

    # Convert the input images to a single vector of features
    tf.keras.layers.GlobalAveragePooling2D(),

    # Learn a non-linear transformation of the input features
    tf.keras.layers.Dense(128, activation = 'relu'),

    # Prevent overfitting by randomly dropping out some of the units during␣
↪training
    tf.keras.layers.Dropout(0.5),

    # Classify the input images into one of n_classes classes
    tf.keras.layers.Dense(n_classes, activation = 'softmax')
])
```

This code snippet defines a Keras model for image classification. The model is defined using the Sequential API, which allows us to stack layers in a sequential order.

The first layer in the model is the data_augmentation layer. This layer applies random transformations to the input images, such as cropping, flipping, and rotating. This helps to improve the model's generalization performance.

The next layer is the Rescaling layer. This layer normalizes the input images to the range [0, 1]. This is a common practice in image classification, as it helps to improve the performance of many models.

The next layer is the base_model. This is a pre-trained model that has been trained on a large dataset of images. We can use a pre-trained model as a starting point for our own model, which can save us a lot of time and effort.

The next layer is the GlobalAveragePooling2D layer. This layer converts the input images to a single vector of features. This is a common practice in image classification, as it allows us to use a fully connected layer to classify the images.

The next layer is a Dense layer with 128 units and the relu activation function. This layer learns a non-linear transformation of the input features.

The next layer is a Dropout layer with a dropout rate of 0.5. This layer helps to prevent overfitting by randomly dropping out some of the units during training.

The last layer in the model is a Dense layer with n_classes units and the softmax activation function. This layer classifies the input images into one of n_classes classes.

6 Transfer Learning:

Transfer learning is a machine learning technique where a model trained on one task is adapted for use on a second related task. It leverages the knowledge gained from the source task to improve learning and performance on the target task. Transfer learning is widely used in various fields, including computer vision, natural language processing, and speech recognition, to save time and resources while achieving better results.

Here's how transfer learning typically works:

1. **Pre-training**: A neural network model, often a deep learning model like a convolutional neural network (CNN) for image tasks or a recurrent neural network (RNN) for text tasks, is trained on a large dataset for a related task. This initial training, known as pre-training, helps the model learn useful features and representations of the data.

2. **Fine-tuning**: After pre-training, the model is fine-tuned on the target task. The earlier layers of the model, which have learned general features, are often kept fixed, while the later layers are adjusted or retrained for the specific target task. This allows the model to adapt its knowledge to the new task, making the training process faster and more effective.

Transfer learning offers several advantages:

1. **Reduced Data Requirements**: Since the model has already learned valuable features from the source task, it may require less data for training on the target task.
2. **Faster Training**: Transfer learning typically accelerates the training process, as the model starts with a foundation of knowledge.
3. **Improved Generalization**: Models trained with transfer learning often generalize better to new tasks and domains because they capture more general patterns.
4. **Resource Efficiency**: It saves computational resources and time compared to training a model from scratch.

Transfer learning is used in various applications, such as image classification, object detection, sentiment analysis, machine translation, and more. Popular pre-trained models like BERT for natural language processing and ResNet for computer vision have demonstrated the effectiveness of this technique in different domains.

[ ]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, 244, 244, 3)       0

 rescaling (Rescaling)       (None, 244, 244, 3)       0

 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 global_average_pooling2d (  (None, 512)               0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 128)               65664

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 7)                 903

=================================================================
Total params: 14781255 (56.39 MB)
Trainable params: 66567 (260.03 KB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Why do we need to compile the model?

We need to compile the model before we can train or evaluate it. The compiler configures the learning process and ensures that all of the necessary components are in place.

```
[ ]: model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001),
                   loss = 'categorical_crossentropy',
                   metrics = ['accuracy'])
```

```
[ ]: from keras.callbacks import ReduceLROnPlateau

     # Model Chackpoint
     tl_checkpoint_1 = ModelCheckpoint(filepath = 'vgg16_best_weights.hdf5',␣
      ↪save_best_only = True, verbose = 0)

     # EarlyStopping
     early_stop = EarlyStopping(monitor = 'val_loss', patience = 5,␣
      ↪restore_best_weights = True, mode = 'min')

     # ReduceLROnPlateau to stabilize the training process of the model
     rop_callback = ReduceLROnPlateau(monitor = 'val_loss', patience = 3, verbose =␣
      ↪1, factor = 0.5, min_lr = 0.000001)
```

Model Chackpoint

The ModelCheckpoint callback saves the model's weights to a file at the end of each epoch. If the model is performing better at the end of an epoch than it was at the end of the previous epoch, the weights are overwritten. This ensures that we always have the best performing model weights saved.

EarlyStopping

The EarlyStopping callback stops training the model if the monitored metric does not improve for a specified number of epochs. This helps to prevent overfitting and save training time.

ReduceLROnPlateau

The ReduceLROnPlateau callback reduces the learning rate of the optimizer if the monitored metric does not improve for a specified number of epochs. This can help to improve the model's performance and prevent overfitting.

```
[ ]: %%time

     history=model.
      ↪fit(train_ds,epochs=51,validation_data=validation_ds,callbacks=[tl_checkpoint_1,early_stop,
```

```
Epoch 1/51
28/28 [==============================] - 402s 6s/step - loss: 1.9179 - accuracy:
0.2223 - val_loss: 1.6663 - val_accuracy: 0.4485 - lr: 0.0010
Epoch 2/51
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3000:
UserWarning: You are saving your model as an HDF5 file via `model.save()`. This
file format is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(

28/28 [==============================] - 14s 488ms/step - loss: 1.6292 -
accuracy: 0.3663 - val_loss: 1.4900 - val_accuracy: 0.4531 - lr: 0.0010
Epoch 3/51
28/28 [==============================] - 14s 508ms/step - loss: 1.5052 -
accuracy: 0.4257 - val_loss: 1.4020 - val_accuracy: 0.4943 - lr: 0.0010
Epoch 4/51
28/28 [==============================] - 13s 482ms/step - loss: 1.3992 -
accuracy: 0.4509 - val_loss: 1.2944 - val_accuracy: 0.5561 - lr: 0.0010
Epoch 5/51
28/28 [==============================] - 13s 463ms/step - loss: 1.3341 -
accuracy: 0.5074 - val_loss: 1.2339 - val_accuracy: 0.5744 - lr: 0.0010
Epoch 6/51
28/28 [==============================] - 13s 457ms/step - loss: 1.2753 -
accuracy: 0.5109 - val_loss: 1.1801 - val_accuracy: 0.5973 - lr: 0.0010
Epoch 7/51
28/28 [==============================] - 13s 459ms/step - loss: 1.2311 -
accuracy: 0.5514 - val_loss: 1.1441 - val_accuracy: 0.6064 - lr: 0.0010
Epoch 8/51
28/28 [==============================] - 13s 468ms/step - loss: 1.1929 -
accuracy: 0.5531 - val_loss: 1.0910 - val_accuracy: 0.6224 - lr: 0.0010
Epoch 9/51
28/28 [==============================] - 13s 469ms/step - loss: 1.1769 -
accuracy: 0.5537 - val_loss: 1.0963 - val_accuracy: 0.6247 - lr: 0.0010
Epoch 10/51
28/28 [==============================] - 16s 572ms/step - loss: 1.1297 -
accuracy: 0.5869 - val_loss: 1.0374 - val_accuracy: 0.6636 - lr: 0.0010
Epoch 11/51
28/28 [==============================] - 13s 470ms/step - loss: 1.1168 -
accuracy: 0.5914 - val_loss: 1.0126 - val_accuracy: 0.6705 - lr: 0.0010
Epoch 12/51
28/28 [==============================] - 13s 472ms/step - loss: 1.0955 -
accuracy: 0.5966 - val_loss: 0.9792 - val_accuracy: 0.6865 - lr: 0.0010
Epoch 13/51
28/28 [==============================] - 13s 470ms/step - loss: 1.0570 -
accuracy: 0.6137 - val_loss: 0.9701 - val_accuracy: 0.6728 - lr: 0.0010
Epoch 14/51
28/28 [==============================] - 16s 565ms/step - loss: 1.0528 -
accuracy: 0.6189 - val_loss: 0.9487 - val_accuracy: 0.7048 - lr: 0.0010
Epoch 15/51
28/28 [==============================] - 13s 465ms/step - loss: 1.0212 -
accuracy: 0.6286 - val_loss: 0.9400 - val_accuracy: 0.6934 - lr: 0.0010
Epoch 16/51
```

```
28/28 [==============================] - 13s 470ms/step - loss: 1.0112 -
accuracy: 0.6331 - val_loss: 0.9167 - val_accuracy: 0.7071 - lr: 0.0010
Epoch 17/51
28/28 [==============================] - 13s 472ms/step - loss: 0.9903 -
accuracy: 0.6429 - val_loss: 0.9129 - val_accuracy: 0.6911 - lr: 0.0010
Epoch 18/51
28/28 [==============================] - 13s 472ms/step - loss: 0.9944 -
accuracy: 0.6440 - val_loss: 0.9088 - val_accuracy: 0.7071 - lr: 0.0010
Epoch 19/51
28/28 [==============================] - 16s 566ms/step - loss: 0.9654 -
accuracy: 0.6417 - val_loss: 0.8825 - val_accuracy: 0.7254 - lr: 0.0010
Epoch 20/51
28/28 [==============================] - 13s 469ms/step - loss: 0.9597 -
accuracy: 0.6469 - val_loss: 0.8854 - val_accuracy: 0.6934 - lr: 0.0010
Epoch 21/51
28/28 [==============================] - 13s 469ms/step - loss: 0.9418 -
accuracy: 0.6680 - val_loss: 0.8656 - val_accuracy: 0.7231 - lr: 0.0010
Epoch 22/51
28/28 [==============================] - 13s 464ms/step - loss: 0.9165 -
accuracy: 0.6646 - val_loss: 0.8715 - val_accuracy: 0.7162 - lr: 0.0010
Epoch 23/51
28/28 [==============================] - 13s 472ms/step - loss: 0.9235 -
accuracy: 0.6674 - val_loss: 0.8457 - val_accuracy: 0.7231 - lr: 0.0010
Epoch 24/51
28/28 [==============================] - 15s 560ms/step - loss: 0.9123 -
accuracy: 0.6829 - val_loss: 0.8475 - val_accuracy: 0.7231 - lr: 0.0010
Epoch 25/51
28/28 [==============================] - 13s 466ms/step - loss: 0.8905 -
accuracy: 0.6737 - val_loss: 0.8158 - val_accuracy: 0.7277 - lr: 0.0010
Epoch 26/51
28/28 [==============================] - 13s 461ms/step - loss: 0.9143 -
accuracy: 0.6783 - val_loss: 0.8268 - val_accuracy: 0.7368 - lr: 0.0010
Epoch 27/51
28/28 [==============================] - 13s 462ms/step - loss: 0.8942 -
accuracy: 0.6834 - val_loss: 0.8262 - val_accuracy: 0.7300 - lr: 0.0010
Epoch 28/51
28/28 [==============================] - 13s 470ms/step - loss: 0.8593 -
accuracy: 0.6880 - val_loss: 0.7974 - val_accuracy: 0.7414 - lr: 0.0010
Epoch 29/51
28/28 [==============================] - 13s 458ms/step - loss: 0.8715 -
accuracy: 0.6857 - val_loss: 0.7992 - val_accuracy: 0.7185 - lr: 0.0010
Epoch 30/51
28/28 [==============================] - 13s 480ms/step - loss: 0.8722 -
accuracy: 0.6949 - val_loss: 0.7944 - val_accuracy: 0.7346 - lr: 0.0010
Epoch 31/51
28/28 [==============================] - 13s 473ms/step - loss: 0.8375 -
accuracy: 0.7051 - val_loss: 0.8039 - val_accuracy: 0.7277 - lr: 0.0010
Epoch 32/51
```

```
28/28 [==============================] - 13s 465ms/step - loss: 0.8592 -
accuracy: 0.6960 - val_loss: 0.8187 - val_accuracy: 0.7231 - lr: 0.0010
Epoch 33/51
28/28 [==============================] - 13s 466ms/step - loss: 0.8563 -
accuracy: 0.7086 - val_loss: 0.7826 - val_accuracy: 0.7437 - lr: 0.0010
Epoch 34/51
28/28 [==============================] - 13s 466ms/step - loss: 0.8566 -
accuracy: 0.6949 - val_loss: 0.7800 - val_accuracy: 0.7391 - lr: 0.0010
Epoch 35/51
28/28 [==============================] - 13s 463ms/step - loss: 0.8215 -
accuracy: 0.7023 - val_loss: 0.7873 - val_accuracy: 0.7391 - lr: 0.0010
Epoch 36/51
28/28 [==============================] - 13s 469ms/step - loss: 0.8129 -
accuracy: 0.7229 - val_loss: 0.7744 - val_accuracy: 0.7323 - lr: 0.0010
Epoch 37/51
28/28 [==============================] - 13s 464ms/step - loss: 0.8139 -
accuracy: 0.7171 - val_loss: 0.7814 - val_accuracy: 0.7300 - lr: 0.0010
Epoch 38/51
28/28 [==============================] - 13s 464ms/step - loss: 0.8295 -
accuracy: 0.7034 - val_loss: 0.7931 - val_accuracy: 0.7437 - lr: 0.0010
Epoch 39/51
28/28 [==============================] - 16s 566ms/step - loss: 0.8202 -
accuracy: 0.7183 - val_loss: 0.7584 - val_accuracy: 0.7368 - lr: 0.0010
Epoch 40/51
28/28 [==============================] - 13s 462ms/step - loss: 0.7878 -
accuracy: 0.7160 - val_loss: 0.7588 - val_accuracy: 0.7414 - lr: 0.0010
Epoch 41/51
28/28 [==============================] - 13s 466ms/step - loss: 0.8114 -
accuracy: 0.7017 - val_loss: 0.7623 - val_accuracy: 0.7460 - lr: 0.0010
Epoch 42/51
28/28 [==============================] - 13s 472ms/step - loss: 0.8006 -
accuracy: 0.7143 - val_loss: 0.7346 - val_accuracy: 0.7368 - lr: 0.0010
Epoch 43/51
28/28 [==============================] - 13s 464ms/step - loss: 0.7919 -
accuracy: 0.7126 - val_loss: 0.7590 - val_accuracy: 0.7506 - lr: 0.0010
Epoch 44/51
28/28 [==============================] - 13s 462ms/step - loss: 0.7858 -
accuracy: 0.7337 - val_loss: 0.7429 - val_accuracy: 0.7529 - lr: 0.0010
Epoch 45/51
28/28 [==============================] - ETA: 0s - loss: 0.7919 - accuracy:
0.7200
Epoch 45: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
28/28 [==============================] - 13s 454ms/step - loss: 0.7919 -
accuracy: 0.7200 - val_loss: 0.7387 - val_accuracy: 0.7300 - lr: 0.0010
Epoch 46/51
28/28 [==============================] - 13s 477ms/step - loss: 0.7676 -
accuracy: 0.7120 - val_loss: 0.7312 - val_accuracy: 0.7483 - lr: 5.0000e-04
Epoch 47/51
```

```
28/28 [==============================] - 13s 477ms/step - loss: 0.7616 -
accuracy: 0.7343 - val_loss: 0.7487 - val_accuracy: 0.7414 - lr: 5.0000e-04
Epoch 48/51
28/28 [==============================] - 13s 473ms/step - loss: 0.7400 -
accuracy: 0.7343 - val_loss: 0.7251 - val_accuracy: 0.7437 - lr: 5.0000e-04
Epoch 49/51
28/28 [==============================] - 13s 460ms/step - loss: 0.7543 -
accuracy: 0.7320 - val_loss: 0.7300 - val_accuracy: 0.7437 - lr: 5.0000e-04
Epoch 50/51
28/28 [==============================] - 13s 456ms/step - loss: 0.7521 -
accuracy: 0.7377 - val_loss: 0.7255 - val_accuracy: 0.7529 - lr: 5.0000e-04
Epoch 51/51
28/28 [==============================] - 13s 473ms/step - loss: 0.7404 -
accuracy: 0.7394 - val_loss: 0.7176 - val_accuracy: 0.7529 - lr: 5.0000e-04
CPU times: user 1min 39s, sys: 32.4 s, total: 2min 11s
Wall time: 18min
```

[ ]:
```
acc = history.history['accuracy']
acc
```

[ ]:
```
[0.22228571772575378,
 0.3662857115268707,
 0.42571428418159485,
 0.45085713267326355,
 0.5074285864830017,
 0.5108571648597717,
 0.5514285564422607,
 0.5531428456306458,
 0.5537142753601074,
 0.5868571400642395,
 0.5914285778999329,
 0.5965714454650879,
 0.6137142777442932,
 0.6188571453094482,
 0.6285714507102966,
 0.6331428289413452,
 0.6428571343421936,
 0.6439999938011169,
 0.6417142748832703,
 0.6468571424484253,
 0.6679999828338623,
 0.6645714044570923,
 0.6674285531044006,
 0.6828571557998657,
 0.673714280128479,
 0.6782857179641724,
 0.6834285855293274,
 0.6880000233650208,
 0.6857143044471741,
 0.694857120513916,
 0.7051428556442261,
 0.6959999799728394,
 0.7085714340209961,
 0.694857120513916,
 0.7022857069969177,
 0.7228571176528931,
 0.7171428799629211,
 0.7034285664558411,
 0.7182857394218445,
 0.7160000205039978,
 0.701714277267456,
 0.7142857313156128,
 0.7125714421272278,
 0.7337142825126648,
 0.7200000286102295,
 0.7120000123977661,
 0.7342857122421265,
 0.7342857122421265,
 0.7319999933242798,
 0.7377142906188965,
 0.7394285798072815]
```

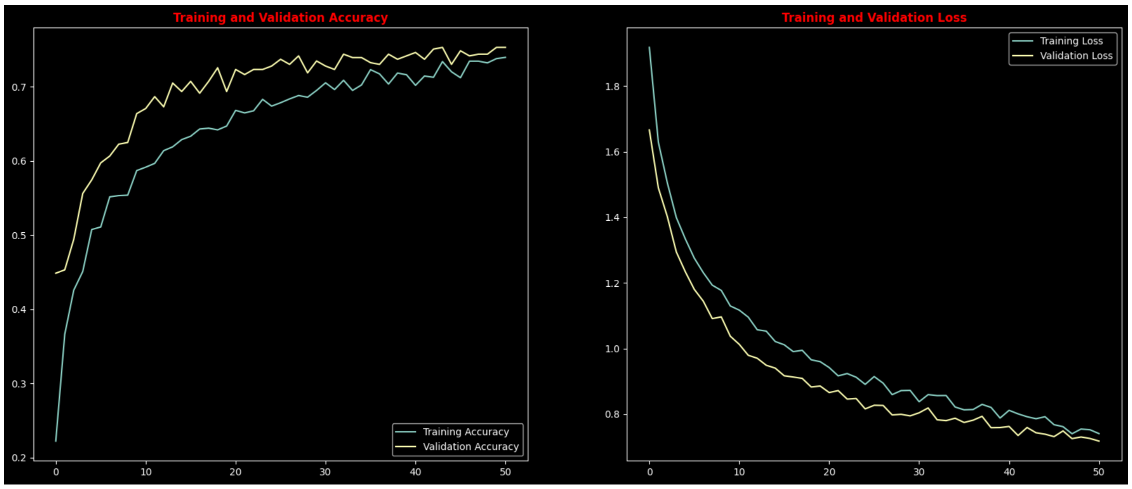
[ ]:
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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

epochs_range = range(51)

plt.figure(figsize = (20, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label = 'Training Accuracy')
plt.plot(epochs_range, val_acc, label = 'Validation Accuracy')
plt.legend(loc = 'lower right')
plt.title('Training and Validation Accuracy',weight='bold',color='red')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label = 'Training Loss')
plt.plot(epochs_range, val_loss, label = 'Validation Loss')
plt.legend(loc = 'upper right')
plt.title('Training and Validation Loss',weight='bold',color='red')
plt.show()
```



6.1 Tips For Reading above Graph...

Reading and interpreting training and validation accuracy, as well as training and validation loss, graphs is a common practice in machine learning to assess the performance of a model during training. These graphs help you understand how well your model is learning and whether it's overfitting or underfitting. Here's how to read and interpret these graphs:

1. **Training Accuracy and Validation Accuracy**:

   - **Training Accuracy**: This line represents the accuracy of your model on the training data as the number of training epochs (iterations) increases. It shows how well your model is fitting the training data.

   - **Validation Accuracy**: This line represents the accuracy of your model on a separate validation dataset, which the model has not seen during training. It's used to evaluate the model's generalization to new, unseen data.

   - **Interpretation**:

     – If both training and validation accuracy increase and stay close to each other, your model is likely learning well without overfitting.
     – If training accuracy is significantly higher than validation accuracy, it may indicate overfitting, where the model is fitting the training data too closely and not generalizing well to new data.
     – If both training and validation accuracy are low and don't improve, it might indicate that your model is underfitting the data.

2. **Training Loss and Validation Loss**:

   - **Training Loss**: This line represents the loss on the training data as the model is trained. The loss measures how well the model's predictions match the actual target values. The goal is to minimize this loss.

   - **Validation Loss**: Similar to validation accuracy, validation loss represents the loss on the validation dataset, which is used to evaluate how well the model generalizes.

- **Interpretation**:
  - A decreasing training loss is a good sign, as it indicates that your model is learning and fitting the training data.
  - A decreasing validation loss is also desirable, as it suggests that the model generalizes well to unseen data.
  - If training loss continues to decrease while validation loss starts to increase, it may indicate overfitting.
  - If both training and validation losses remain high or decrease very slowly, it may indicate underfitting.

Here are some key takeaways when reading these graphs:

- Overfitting: If validation accuracy/loss starts to worsen (e.g., increase in loss or decrease in accuracy) while training accuracy/loss continues to improve, your model is likely overfitting the training data.
- Underfitting: If both training and validation metrics are not improving, or they plateau at a low value, your model may be underfitting the data.
- Ideally, you want training and validation metrics to show improvement and be relatively close, indicating that your model is learning effectively without overfitting or underfitting.

It's common to monitor these graphs during model training and potentially adjust hyperparameters or employ regularization techniques to achieve the best model performance.

```
[ ]: test_ds = tf.keras.utils.image_dataset_from_directory(
         test_data_dir,
         label_mode = 'categorical',
         image_size = (img_height, img_width),
         batch_size = 1,
         seed = 123)

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

```
Found 564 files belonging to 7 classes.
```

6.2 Prediction using above Tarined Model..

```
[ ]: model.load_weights('/content/vgg16_best_weights.hdf5')
     preds=model.predict(test_ds)
     pred_classes=np.argmax(preds,axis=1)
```

```
564/564 [==============================] - 88s 152ms/step
```

```
[ ]: model.evaluate(test_ds,verbose=1)
```

```
564/564 [==============================] - 7s 12ms/step - loss: 0.8043 -
accuracy: 0.7234
```

```
[ ]: [0.8043269515037537, 0.7234042286872864]
```

7 Fine-Tuning

Fine-tuning is a process in machine learning, particularly in the context of deep learning, where a pre-trained model is further trained on a new or specific task. This involves making small adjustments to the model's parameters to adapt it to the target task while retaining the knowledge gained from the original training. Fine-tuning is a crucial step in transfer learning, where a pre-trained model is utilized as a starting point for a new, related task.

Here's how fine-tuning typically works:

1. **Pre-trained Model**: A pre-trained model, often a deep neural network trained on a large dataset for a related task, is selected as a starting point. This model has already learned valuable features and representations from the source task.
2. **Modification**: The architecture of the pre-trained model may be modified to suit the requirements of the target task. This might involve adding or removing layers, adjusting the number of neurons in the layers, or making other architectural changes.
3. **Partial Freezing**: Typically, the earlier layers of the pre-trained model, known as the feature extraction layers, are kept frozen. This means their weights are not updated during fine-tuning. The later layers, often referred to as the classification or prediction layers, are modified and trained on the new task.
4. **Training on Target Task Data**: The fine-tuning process involves training the modified model using the target task's dataset. The model is exposed to the new data, and its weights are updated to learn the specific patterns and information relevant to the target task.
5. **Hyperparameter Tuning**: Hyperparameters like learning rates, batch sizes, and regularization methods may need to be adjusted for fine-tuning to achieve the best performance on the new task.

The primary goal of fine-tuning is to leverage the knowledge and representations learned during the pre-training phase and adapt them to the nuances of the target task. This process often leads to faster convergence and better results compared to training a model from scratch for the new task, especially when there's a limited amount of task-specific data available.

Fine-tuning is commonly used in various applications, such as image classification, text sentiment analysis, language translation, and more. It is a key technique in transfer learning, where the knowledge transfer from a pre-trained model to a new task is a fundamental part of the approach.

[ ]:
```
fine_tune=base_model
```

[ ]:
```
fine_tune.trainable=True
```

[ ]:
```
for l in fine_tune.layers:
  trainable=('block5' in l.name or 'block4' in l.name)

  l.trainable=trainable
```

[ ]:
```
print_layer_trainable()
```

```
False:  ==>     input_1
False:  ==>     block1_conv1
False:  ==>     block1_conv2
False:  ==>     block1_pool
False:  ==>     block2_conv1
False:  ==>     block2_conv2
False:  ==>     block2_pool
False:  ==>     block3_conv1
False:  ==>     block3_conv2
False:  ==>     block3_conv3
False:  ==>     block3_pool
True:   ==>     block4_conv1
True:   ==>     block4_conv2
True:   ==>     block4_conv3
True:   ==>     block4_pool
True:   ==>     block5_conv1
True:   ==>     block5_conv2
True:   ==>     block5_conv3
True:   ==>     block5_pool
```

[ ]:
```
fine_tune.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 244, 244, 3)]     0

 block1_conv1 (Conv2D)       (None, 244, 244, 64)      1792

 block1_conv2 (Conv2D)       (None, 244, 244, 64)      36928

 block1_pool (MaxPooling2D)  (None, 122, 122, 64)      0

 block2_conv1 (Conv2D)       (None, 122, 122, 128)     73856

 block2_conv2 (Conv2D)       (None, 122, 122, 128)     147584

 block2_pool (MaxPooling2D)  (None, 61, 61, 128)       0

 block3_conv1 (Conv2D)       (None, 61, 61, 256)       295168

 block3_conv2 (Conv2D)       (None, 61, 61, 256)       590080

 block3_conv3 (Conv2D)       (None, 61, 61, 256)       590080

 block3_pool (MaxPooling2D)  (None, 30, 30, 256)       0
 block4_conv1 (Conv2D)       (None, 30, 30, 512)       1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, 30, 30, 512)       2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, 30, 30, 512)       2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, 15, 15, 512)       0         
                                                                 
 block5_conv1 (Conv2D)       (None, 15, 15, 512)       2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, 15, 15, 512)       2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, 15, 15, 512)       2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0         
                                                                 
=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 12979200 (49.51 MB)
Non-trainable params: 1735488 (6.62 MB)
_________________________________________________________________
```

```
[ ]: n_classes = len(class_names)

     model2 = Sequential([
         data_augmentation,
         tf.keras.layers.Rescaling(1./255),
         fine_tune,
         tf.keras.layers.GlobalAveragePooling2D(),
         tf.keras.layers.Dense(128, activation = 'relu'),
         tf.keras.layers.Dropout(0.5),
         tf.keras.layers.Dense(n_classes, activation = 'softmax')
     ])
```

```
[ ]: model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 sequential (Sequential)     (None, 244, 244, 3)       0         
                                                                 
 rescaling_1 (Rescaling)     (None, 244, 244, 3)       0         
                                                                 
 vgg16 (Functional)          (None, 7, 7, 512)         14714688  
                                                                 
 global_average_pooling2d_1  (None, 512)               0         
  (GlobalAveragePooling2D)                                       
 dense_2 (Dense)             (None, 128)               65664

 dropout_1 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 7)                 903

=================================================================
Total params: 14781255 (56.39 MB)
Trainable params: 13045767 (49.77 MB)
Non-trainable params: 1735488 (6.62 MB)
_________________________________________________________________
```

[ ]:
```
model2.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001),
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
```

[ ]:
```
# Model Chackpoint
tl_checkpoint_1 = ModelCheckpoint(filepath = 'vgg16_best_weights_fine_tuning.
 ↪hdf5', save_best_only = True, verbose = 0)

# EarlyStopping
early_stop = EarlyStopping(monitor = 'val_loss', patience = 10,␣
 ↪restore_best_weights = True, mode = 'min')

#ReduceLROnPlateau to stabilize the training process of the model
rop_callback = ReduceLROnPlateau(monitor = 'val_loss', patience = 3, verbose =␣
 ↪1, factor = 0.5, min_lr = 0.000001)
```

[ ]:
```
%%time
history2 = model2.fit(train_ds,
                    epochs = 51,
                    validation_data = validation_ds,
                    callbacks = [tl_checkpoint_1, early_stop, rop_callback])
```

```
Epoch 1/51
28/28 [==============================] - 28s 715ms/step - loss: 2.4448 -
accuracy: 0.1989 - val_loss: 1.8293 - val_accuracy: 0.2380 - lr: 0.0010
Epoch 2/51
28/28 [==============================] - 17s 606ms/step - loss: 1.8517 -
accuracy: 0.2097 - val_loss: 1.7522 - val_accuracy: 0.2586 - lr: 0.0010
Epoch 3/51
28/28 [==============================] - 17s 599ms/step - loss: 1.7902 -
accuracy: 0.2651 - val_loss: 1.7313 - val_accuracy: 0.2540 - lr: 0.0010
Epoch 4/51
28/28 [==============================] - 17s 622ms/step - loss: 1.7640 -
accuracy: 0.2743 - val_loss: 1.6652 - val_accuracy: 0.3135 - lr: 0.0010
Epoch 5/51
```

```
28/28 [==============================] - 17s 622ms/step - loss: 1.7529 -
accuracy: 0.2840 - val_loss: 1.7156 - val_accuracy: 0.2998 - lr: 0.0010
Epoch 6/51
28/28 [==============================] - 17s 624ms/step - loss: 1.6864 -
accuracy: 0.3034 - val_loss: 1.5903 - val_accuracy: 0.3730 - lr: 0.0010
Epoch 7/51
28/28 [==============================] - 17s 613ms/step - loss: 1.5359 -
accuracy: 0.3537 - val_loss: 1.3628 - val_accuracy: 0.4737 - lr: 0.0010
Epoch 8/51
28/28 [==============================] - 17s 602ms/step - loss: 1.4467 -
accuracy: 0.4166 - val_loss: 1.3944 - val_accuracy: 0.4279 - lr: 0.0010
Epoch 9/51
28/28 [==============================] - 17s 604ms/step - loss: 1.4421 -
accuracy: 0.3966 - val_loss: 1.4012 - val_accuracy: 0.4920 - lr: 0.0010
Epoch 10/51
28/28 [==============================] - ETA: 0s - loss: 1.2826 - accuracy:
0.4909
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
28/28 [==============================] - 17s 614ms/step - loss: 1.2826 -
accuracy: 0.4909 - val_loss: 1.8391 - val_accuracy: 0.4783 - lr: 0.0010
Epoch 11/51
28/28 [==============================] - 18s 632ms/step - loss: 1.1816 -
accuracy: 0.5366 - val_loss: 1.1476 - val_accuracy: 0.5423 - lr: 5.0000e-04
Epoch 12/51
28/28 [==============================] - 17s 609ms/step - loss: 1.0650 -
accuracy: 0.5880 - val_loss: 1.1186 - val_accuracy: 0.5721 - lr: 5.0000e-04
Epoch 13/51
28/28 [==============================] - 17s 618ms/step - loss: 1.0238 -
accuracy: 0.6034 - val_loss: 1.1182 - val_accuracy: 0.6041 - lr: 5.0000e-04
Epoch 14/51
28/28 [==============================] - 17s 612ms/step - loss: 1.0408 -
accuracy: 0.5971 - val_loss: 1.1389 - val_accuracy: 0.5675 - lr: 5.0000e-04
Epoch 15/51
28/28 [==============================] - 17s 621ms/step - loss: 0.9488 -
accuracy: 0.6474 - val_loss: 1.0207 - val_accuracy: 0.6499 - lr: 5.0000e-04
Epoch 16/51
28/28 [==============================] - 17s 607ms/step - loss: 0.9112 -
accuracy: 0.6674 - val_loss: 1.1291 - val_accuracy: 0.6087 - lr: 5.0000e-04
Epoch 17/51
28/28 [==============================] - 17s 621ms/step - loss: 0.8995 -
accuracy: 0.6474 - val_loss: 0.9683 - val_accuracy: 0.6545 - lr: 5.0000e-04
Epoch 18/51
28/28 [==============================] - 17s 607ms/step - loss: 0.8886 -
accuracy: 0.6789 - val_loss: 1.0061 - val_accuracy: 0.6568 - lr: 5.0000e-04
Epoch 19/51
28/28 [==============================] - 17s 619ms/step - loss: 0.8115 -
accuracy: 0.7034 - val_loss: 0.9124 - val_accuracy: 0.6911 - lr: 5.0000e-04
Epoch 20/51
```

```
28/28 [==============================] - 17s 605ms/step - loss: 0.8179 -
accuracy: 0.7063 - val_loss: 0.9902 - val_accuracy: 0.6888 - lr: 5.0000e-04
Epoch 21/51
28/28 [==============================] - 17s 603ms/step - loss: 0.7815 -
accuracy: 0.7200 - val_loss: 0.9693 - val_accuracy: 0.6407 - lr: 5.0000e-04
Epoch 22/51
28/28 [==============================] - ETA: 0s - loss: 0.7460 - accuracy:
0.7131
Epoch 22: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
28/28 [==============================] - 17s 604ms/step - loss: 0.7460 -
accuracy: 0.7131 - val_loss: 0.9284 - val_accuracy: 0.6911 - lr: 5.0000e-04
Epoch 23/51
28/28 [==============================] - 17s 606ms/step - loss: 0.6735 -
accuracy: 0.7703 - val_loss: 0.9208 - val_accuracy: 0.7025 - lr: 2.5000e-04
Epoch 24/51
28/28 [==============================] - 17s 607ms/step - loss: 0.6807 -
accuracy: 0.7663 - val_loss: 1.0090 - val_accuracy: 0.6682 - lr: 2.5000e-04
Epoch 25/51
28/28 [==============================] - 17s 626ms/step - loss: 0.6209 -
accuracy: 0.7857 - val_loss: 0.8785 - val_accuracy: 0.7140 - lr: 2.5000e-04
Epoch 26/51
28/28 [==============================] - 17s 628ms/step - loss: 0.6108 -
accuracy: 0.7863 - val_loss: 0.8486 - val_accuracy: 0.7437 - lr: 2.5000e-04
Epoch 27/51
28/28 [==============================] - 17s 621ms/step - loss: 0.6283 -
accuracy: 0.7777 - val_loss: 0.8480 - val_accuracy: 0.7277 - lr: 2.5000e-04
Epoch 28/51
28/28 [==============================] - 17s 626ms/step - loss: 0.5832 -
accuracy: 0.7891 - val_loss: 0.8365 - val_accuracy: 0.7117 - lr: 2.5000e-04
Epoch 29/51
28/28 [==============================] - 17s 605ms/step - loss: 0.5376 -
accuracy: 0.8223 - val_loss: 0.9404 - val_accuracy: 0.7368 - lr: 2.5000e-04
Epoch 30/51
28/28 [==============================] - 17s 611ms/step - loss: 0.5529 -
accuracy: 0.8063 - val_loss: 1.0239 - val_accuracy: 0.7140 - lr: 2.5000e-04
Epoch 31/51
28/28 [==============================] - ETA: 0s - loss: 0.5399 - accuracy:
0.8211
Epoch 31: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
28/28 [==============================] - 17s 609ms/step - loss: 0.5399 -
accuracy: 0.8211 - val_loss: 0.8687 - val_accuracy: 0.7231 - lr: 2.5000e-04
Epoch 32/51
28/28 [==============================] - 17s 620ms/step - loss: 0.5119 -
accuracy: 0.8343 - val_loss: 0.9155 - val_accuracy: 0.6979 - lr: 1.2500e-04
Epoch 33/51
28/28 [==============================] - 17s 607ms/step - loss: 0.4619 -
accuracy: 0.8394 - val_loss: 0.8960 - val_accuracy: 0.7208 - lr: 1.2500e-04
Epoch 34/51
```

```
28/28 [==============================] - ETA: 0s - loss: 0.4736 - accuracy:
0.8360
Epoch 34: ReduceLROnPlateau reducing learning rate to 6.25000029685907e-05.
28/28 [==============================] - 17s 606ms/step - loss: 0.4736 -
accuracy: 0.8360 - val_loss: 0.8759 - val_accuracy: 0.7300 - lr: 1.2500e-04
Epoch 35/51
28/28 [==============================] - 18s 629ms/step - loss: 0.4249 -
accuracy: 0.8491 - val_loss: 0.8211 - val_accuracy: 0.7323 - lr: 6.2500e-05
Epoch 36/51
28/28 [==============================] - 17s 610ms/step - loss: 0.4270 -
accuracy: 0.8549 - val_loss: 0.8284 - val_accuracy: 0.7391 - lr: 6.2500e-05
Epoch 37/51
28/28 [==============================] - 17s 626ms/step - loss: 0.4261 -
accuracy: 0.8577 - val_loss: 0.7683 - val_accuracy: 0.7551 - lr: 6.2500e-05
Epoch 38/51
28/28 [==============================] - 17s 608ms/step - loss: 0.3982 -
accuracy: 0.8600 - val_loss: 0.8408 - val_accuracy: 0.7506 - lr: 6.2500e-05
Epoch 39/51
28/28 [==============================] - 17s 606ms/step - loss: 0.4011 -
accuracy: 0.8583 - val_loss: 0.8052 - val_accuracy: 0.7483 - lr: 6.2500e-05
Epoch 40/51
28/28 [==============================] - 17s 620ms/step - loss: 0.3919 -
accuracy: 0.8686 - val_loss: 0.7674 - val_accuracy: 0.7574 - lr: 6.2500e-05
Epoch 41/51
28/28 [==============================] - 17s 613ms/step - loss: 0.3737 -
accuracy: 0.8714 - val_loss: 0.7236 - val_accuracy: 0.7666 - lr: 6.2500e-05
Epoch 42/51
28/28 [==============================] - 17s 612ms/step - loss: 0.3836 -
accuracy: 0.8749 - val_loss: 0.7991 - val_accuracy: 0.7551 - lr: 6.2500e-05
Epoch 43/51
28/28 [==============================] - 17s 611ms/step - loss: 0.3806 -
accuracy: 0.8766 - val_loss: 0.8208 - val_accuracy: 0.7597 - lr: 6.2500e-05
Epoch 44/51
28/28 [==============================] - ETA: 0s - loss: 0.3841 - accuracy:
0.8651
Epoch 44: ReduceLROnPlateau reducing learning rate to 3.125000148429535e-05.
28/28 [==============================] - 17s 611ms/step - loss: 0.3841 -
accuracy: 0.8651 - val_loss: 0.7671 - val_accuracy: 0.7689 - lr: 6.2500e-05
Epoch 45/51
28/28 [==============================] - 17s 610ms/step - loss: 0.3510 -
accuracy: 0.8754 - val_loss: 0.8323 - val_accuracy: 0.7437 - lr: 3.1250e-05
Epoch 46/51
28/28 [==============================] - 17s 605ms/step - loss: 0.3618 -
accuracy: 0.8783 - val_loss: 0.7889 - val_accuracy: 0.7735 - lr: 3.1250e-05
Epoch 47/51
28/28 [==============================] - ETA: 0s - loss: 0.3459 - accuracy:
0.8743
Epoch 47: ReduceLROnPlateau reducing learning rate to 1.5625000742147677e-05.
```

```
28/28 [==============================] - 17s 604ms/step - loss: 0.3459 -
accuracy: 0.8743 - val_loss: 0.7606 - val_accuracy: 0.7735 - lr: 3.1250e-05
Epoch 48/51
28/28 [==============================] - 17s 618ms/step - loss: 0.3302 -
accuracy: 0.8794 - val_loss: 0.7987 - val_accuracy: 0.7689 - lr: 1.5625e-05
Epoch 49/51
28/28 [==============================] - 17s 610ms/step - loss: 0.3352 -
accuracy: 0.8863 - val_loss: 0.7603 - val_accuracy: 0.7643 - lr: 1.5625e-05
Epoch 50/51
28/28 [==============================] - ETA: 0s - loss: 0.3336 - accuracy:
0.8920
Epoch 50: ReduceLROnPlateau reducing learning rate to 7.812500371073838e-06.
28/28 [==============================] - 17s 609ms/step - loss: 0.3336 -
accuracy: 0.8920 - val_loss: 0.7475 - val_accuracy: 0.7689 - lr: 1.5625e-05
Epoch 51/51
28/28 [==============================] - 17s 605ms/step - loss: 0.3178 -
accuracy: 0.8949 - val_loss: 0.7650 - val_accuracy: 0.7712 - lr: 7.8125e-06
CPU times: user 4min 56s, sys: 25.9 s, total: 5min 22s
Wall time: 15min 3s
```

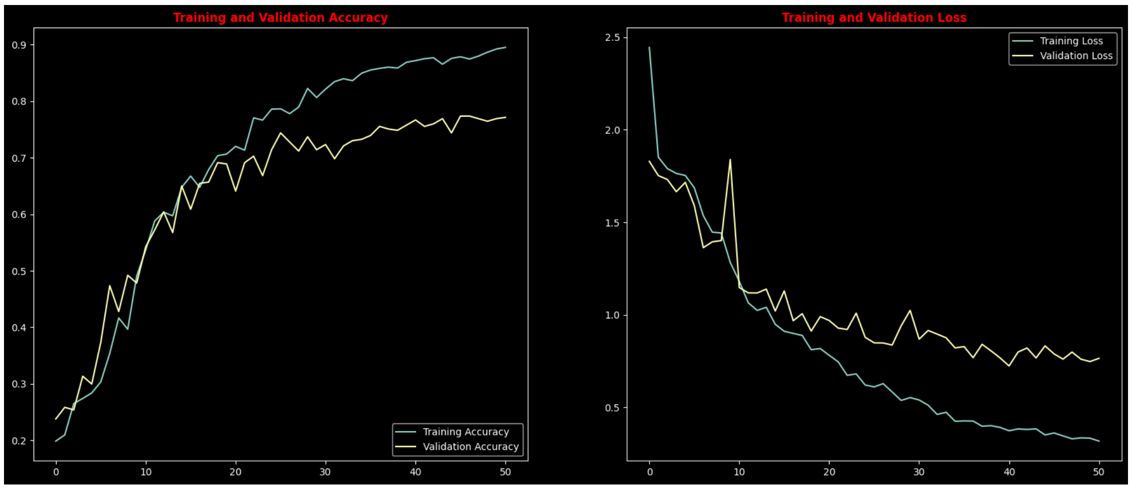
[ ]:
```python
acc2 = history2.history['accuracy']
val_acc2 = history2.history['val_accuracy']

loss2 = history2.history['loss']
val_loss2 = history2.history['val_loss']

epochs_range = range(51)

plt.figure(figsize = (20, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc2, label = 'Training Accuracy')
plt.plot(epochs_range, val_acc2, label = 'Validation Accuracy')
plt.legend(loc = 'lower right')
plt.title('Training and Validation Accuracy',weight='bold',color='red')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss2, label = 'Training Loss')
plt.plot(epochs_range, val_loss2, label = 'Validation Loss')
plt.legend(loc = 'upper right')
plt.title('Training and Validation Loss',weight='bold',color='red')
plt.show()
```



```
[ ]: acc2 = history2.history['accuracy']
     acc2
```

```
[ ]: [0.1988571435213089,
      0.20971427857875824,
      0.26514285802841187,
      0.2742857038974762,
      0.2840000092983246,
      0.3034285604953766,
      0.3537142872810364,
      0.4165714383125305,
      0.39657142758369446,
      0.49085715413093567,
      0.5365714430809021,
      0.5879999995231628,
      0.6034285426139832,
      0.5971428751945496,
      0.647428572177887,
      0.6674285531044006,
      0.647428572177887,
      0.678857147693634,
      0.7034285664558411,
      0.7062857151031494,
      0.7200000286102295,
      0.7131428718566895,
      0.7702857255935669,
      0.7662857174873352,
      0.7857142686843872,
      0.7862856984138489,
      0.7777143120765686,
      0.7891428470611572,
      0.8222857117652893,
      0.8062857389450073,
      0.821142852306366,
      0.8342857360839844,
      0.8394285440444946,
      0.8360000252723694,
      0.849142849445343,
      0.8548571467399597,
      0.8577142953872681,
      0.8600000143051147,
      0.8582857251167297,
      0.868571400642395,
      0.8714285492897034,
      0.8748571276664734,
      0.8765714168548584,
      0.8651428818702698,
      0.8754285573959351,
      0.8782857060432434,
      0.8742856979370117,
      0.8794285655021667,
      0.8862857222557068,
      0.8920000195503235,
      0.8948571681976318]
```

[ ]:
```
test_ds = tf.keras.utils.image_dataset_from_directory(
    test_data_dir,
    label_mode = 'categorical',
    image_size = (img_height, img_width),
    batch_size = 1,
    seed = 123)

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

```
Found 564 files belonging to 7 classes.
```

[ ]:
```
# Generate predictions
model2.load_weights('/content/vgg16_best_weights_fine_tuning.hdf5') #␣
 ↪initialize the best trained weights
preds = model2.predict(test_ds)
pred_classes = np.argmax(preds, axis = 1)
```

```
564/564 [==============================] - 6s 11ms/step
```

[ ]:
```
model2.evaluate(test_ds, verbose = 1)
```

```
564/564 [==============================] - 7s 12ms/step - loss: 0.8615 -
accuracy: 0.7730
```

```
[ ]: [0.8615399599075317, 0.7730496525764465]
```

8 Conclusion

Accuracy In Pre-Train Model => 72%

Accuracy Using Fine-Tuning Model => 77%

9 Thank You!