Classify Waste Products Using Transfer Learning

Table of contents

Overview

EcoClean currently lacks an efficient and scalable method to automate the waste sorting process. The manual sorting of waste is not only labor-intensive but also prone to errors, leading to contamination of recyclable materials. The goal of this project is to leverage machine learning and computer vision to automate the classification of waste products, improving efficiency and reducing contamination rates. The project will use transfer learning with a pre-trained VGG16 model to classify images.

Aim of the Project

The aim of the project is to develop an automated waste classification model that can accurately differentiate between recyclable and organic waste based on images. By the end of this project, you will have trained, fine-tuned, and evaluated a model using transfer learning, which can then be applied to real-world waste management processes.

Final Output: A trained model that classifies waste images into recyclable and organic categories.

Tasks List

- Task 1: Print the version of tensorflow
- Task 2: Create a `test_generator` using the `test_datagen` object
- Task 3: Print the length of the `train_generator`
- Task 4: Print the summary of the model
- Task 5: Compile the model
- Task 6: Plot accuracy curves for training and validation sets (extract_feat_model)
- Task 7: Plot loss curves for training and validation sets (fine tune model)
- Task 8: Plot accuracy curves for training and validation sets (fine tune model)
- Task 9: Plot a test image using Extract Features Model (index_to_plot = 1)
- Task 10: Plot a test image using Fine-Tuned Model (index_to_plot = 1)

```
!pip install matplotlib
!pip install scikit-learn
!pip install tensorflow
!pip install keras
```

```
# Uninstall in a specific order to minimize temporary conflicts
!pip uninstall tensorflow -y
!pip uninstall tensorflow-gpu -y
!pip uninstall keras -y # Uninstall standalone Keras, if present
!pip uninstall h5py -y
!pip uninstall protobuf -y
!pip uninstall numpy -y # Uninstall numpy last of the major ones

print("Uninstall complete. Clearing pip cache...")
!pip cache purge
print("Installing compatible TensorFlow version (which will pull compatible numpy, h5py, protobuf)...")
!pip install tensorflow==2.16.1
```

Importing Required Libraries

```
import numpy as np
import os
import random, shutil
import glob
from matplotlib import pyplot as plt
from matplotlib import pyplot
from matplotlib.image import imread
from os import makedirs,listdir
from shutil import copyfile
from random import seed
from random import random
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator # This import should now work
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Conv2D, MaxPooling2D,GlobalAveragePooling2D, Input
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.applications import InceptionV3
from sklearn import metrics
import warnings
warnings.filterwarnings('ignore')
```

Print the version of tensorflow

```
tf.__version__
```

```
'2.16.1'
```

Background

Transfer learning uses the concept of keeping the early layers of a pre-trained network, and re-training the later layers on a specific dataset. You can leverage some state of that network on a related task.

A typical transfer learning workflow in Keras looks something like this:

1. Initialize base model, and load pre-trained weights (e.g. ImageNet)
2. "Freeze" layers in the base model by setting `training = False`

3. Define a new model that goes on top of the output of the base model's layers.
4. Train resulting model on your data set.

Create a model for distinguishing recyclable and organic waste images

Dataset

Waste Classification Dataset.

Importing Data

This will create a `o-vs-r-split` directory in your environment.

```
import requests
import zipfile
from tqdm import tqdm

url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/kd6057VPpABQ2FqCbgu9YQ/o-vs-r-split-reduced-1200.zip"
file_name = "o-vs-r-split-reduced-1200.zip"

print("Downloading file")
with requests.get(url, stream=True) as response:
    response.raise_for_status()
    with open(file_name, 'wb') as f:
        for chunk in response.iter_content(chunk_size=8192):
            f.write(chunk)


def extract_file_with_progress(file_name):
    print("Extracting file with progress")
    with zipfile.ZipFile(file_name, 'r') as zip_ref:
        members = zip_ref.infolist()
        with tqdm(total=len(members), unit='file') as progress_bar:
            for member in members:
                zip_ref.extract(member)
                progress_bar.update(1)
    print("Finished extracting file")


extract_file_with_progress(file_name)

print("Finished extracting file")
os.remove(file_name)
```

```
Downloading file
Extracting file with progress
100%|██████████| 1207/1207 [01:32<00:00, 12.99file/s]
Finished extracting file
Finished extracting file
```

Define configuration options

It's time to define some model configuration options.

- **batch size** is set to 32.
- The **number of classes** is 2.
- You will use 20% of the data for **validation** purposes.
- You have two **labels** in your dataset: organic (O), recyclable (R).

```
img_rows, img_cols = 150, 150
batch_size = 32
n_epochs = 10
n_classes = 2
val_split = 0.2
verbosity = 1
path = 'o-vs-r-split/train/'
path_test = 'o-vs-r-split/test/'
input_shape = (img_rows, img_cols, 3)
labels = ['O', 'R']
seed = 42
```

Loading Images using ImageGeneratorClass

Transfer learning works best when models are trained on smaller datasets.

ImageDataGenerators

creating ImageDataGenerators used for training, validation and testing.

Image data generators create batches of tensor image data with real-time data augmentation. The generators loop over the data in batches and are useful in feeding data to the training process.

```python
# Create ImageDataGenerators for training and validation and testing
train_datagen = ImageDataGenerator(
    validation_split = val_split,
    rescale=1.0/255.0,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

val_datagen = ImageDataGenerator(
    validation_split = val_split,
    rescale=1.0/255.0,
)

test_datagen = ImageDataGenerator(
    rescale=1.0/255.0
)
```

```python
train_generator = train_datagen.flow_from_directory(
    directory = path,
    seed = seed,
    batch_size = batch_size,
    class_mode='binary',
    shuffle = True,
    target_size=(img_rows, img_cols),
    subset = 'training'
)
```

```
Found 800 images belonging to 2 classes.
```

```python
val_generator = val_datagen.flow_from_directory(
    directory = path,
    seed = seed,
    batch_size = batch_size,
    class_mode='binary',
    shuffle = True,
    target_size=(img_rows, img_cols),
    subset = 'validation'
)
```

```
Found 200 images belonging to 2 classes.
```

Creating a `test_generator` using the `test_datagen` object

- **directory** should be set to `path_test`.
- **class_mode** should be set to `'binary'`.
- **seed** should be set to `seed`.
- **batch_size** should be set to `batch_size`.
- **shuffle** should be set to `False`.
- **target_size** should be set to `(img_rows, img_cols)`.

```python
# Task 2: Create a `test_generator` using the `test_datagen` object
test_generator = test_datagen.flow_from_directory(
    directory = path,
    seed = seed,
    batch_size = batch_size,
    class_mode='binary',
    shuffle = True,
    target_size=(img_rows, img_cols)
)
```

```
Found 1000 images belonging to 2 classes.
```

Print the length of the `train_generator`

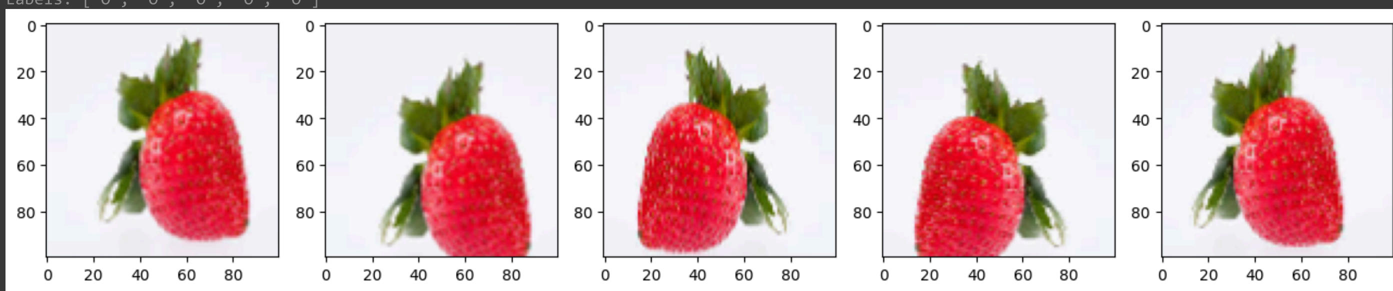
```python
# Task 3: print the length of the `train_generator`
test_generator = test_datagen.flow_from_directory(
    directory = path,
    seed = seed,
    batch_size = batch_size,
    class_mode='binary',
    shuffle = True,
    target_size=(img_rows, img_cols)
)
```

```
Found 1000 images belonging to 2 classes.
```

Let's look at a few augmented images:

```python
from pathlib import Path

IMG_DIM = (100, 100)

train_files = glob.glob('./o-vs-r-split/train/O/*')
train_files = train_files[:20]
train_imgs = [tf.keras.preprocessing.image.img_to_array(tf.keras.preprocessing.image.load_img(img, target_size=IMG_DIM)) for img in train_files]
train_imgs = np.array(train_imgs)
train_labels = [Path(fn).parent.name for fn in train_files]

img_id = 0
O_generator = train_datagen.flow(train_imgs[img_id:img_id+1], train_labels[img_id:img_id+1],
                                   batch_size=1)
O = [next(O_generator) for i in range(0,5)]
fig, ax = plt.subplots(1,5, figsize=(16, 6))
print('Labels:', [item[1][0] for item in O])
l = [ax[i].imshow(O[i][0][0]) for i in range(0,5)]

```

```
Labels: ['O', 'O', 'O', 'O', 'O']
```



Pre-trained Models

Pre-trained models are saved networks that have previously been trained on some large datasets. They are typically used for large-scale image-classification task. They can be used as they are or could be customized to a given task using transfer learning. These pre-trained models form the basis of transfer learning.

VGG-16

Let us load the VGG16 model.

```python
from tensorflow.keras.applications import vgg16

input_shape = (150, 150, 3)
vgg = vgg16.VGG16(include_top=False,
                        weights='imagenet',
                        input_shape=input_shape)


```

We flatten the output of a vgg model and assign it to the model `output`, we then use a Model object `basemodel` to group the layers into an object for training and inference . With the following inputs and outputs

inputs: `vgg.input`

outputs: `tf.keras.layers.Flatten()(output)`

```python
output = vgg.layers[-1].output
output = tf.keras.layers.Flatten()(output)
basemodel = Model(vgg.input, output)
```

Next, you freeze the basemodel.

```python
for layer in basemodel.layers:
    layer.trainable = False
```

Create a new model on top. You add a Dropout layer for regularization, only these layers will change as for the lower layers you set `training=False` when calling the base model.

```python
input_shape = basemodel.output_shape[1]

model = Sequential()
model.add(basemodel)
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
```

```
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))
```

Print the summary of the model

```
# Task: print the summary of the model
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
functional (Functional)	(None, 8192)	14,714,688
dense (Dense)	(None, 512)	4,194,816
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 512)	262,656
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 1)	513

Total params: 19,172,673 (73.14 MB)
Trainable params: 4,457,985 (17.01 MB)

Compile the model

- **loss**: `'binary_crossentropy'`.
- **optimizer**: `optimizers.RMSprop(learning_rate=1e-4)`.
- **metrics**: `['accuracy']`.

```
for layer in basemodel.layers:
    layer.trainable = False

# Task 5: Compile the model
model.compile(
    loss='binary_crossentropy',
    optimizer=optimizers.RMSprop(learning_rate=1e-4),
    metrics=['accuracy']
)
```

You will use early stopping to avoid over-training the model.

```
from tensorflow.keras.callbacks import LearningRateScheduler


checkpoint_path='O_R_tlearn_vgg16.keras'

# define step decay function
class LossHistory_(tf.keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
        self.lr = []

    def on_epoch_end(self, epoch, logs={}):
        self.losses.append(logs.get('loss'))
        self.lr.append(exp_decay(epoch))
        print('lr:', exp_decay(len(self.losses)))

def exp_decay(epoch):
    initial_lrate = 1e-4
    k = 0.1
    lrate = initial_lrate * np.exp(-k*epoch)
    return lrate

# learning schedule callback
loss_history_ = LossHistory_()
lrate_ = LearningRateScheduler(exp_decay)

keras_callbacks = [
      EarlyStopping(monitor = 'val_loss',
                    patience = 4,
                    mode = 'min',
                    min_delta=0.01),
      ModelCheckpoint(checkpoint_path, monitor='val_loss', save_best_only=True, mode='min')
]

callbacks_list_ = [loss_history_, lrate_] + keras_callbacks
```

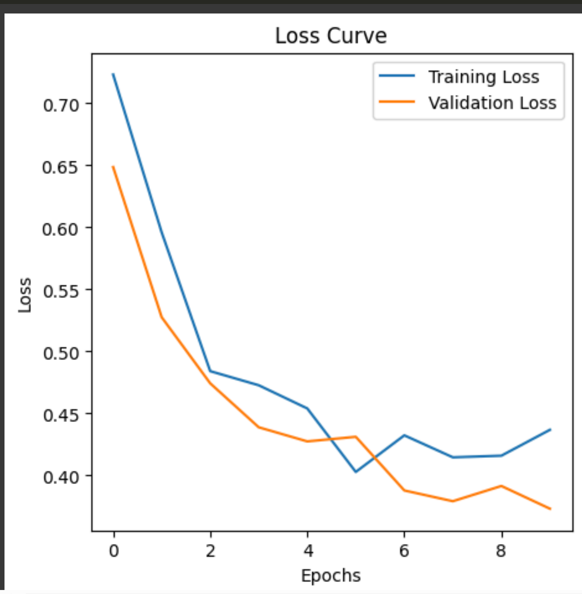
Fit and train the model

```
extract_feat_model = model.fit(train_generator,
                               steps_per_epoch=5,
                               epochs=10,
                               callbacks = callbacks_list_,
                               validation_data=val_generator,
                               validation_steps=val_generator.samples // batch_size,
                               verbose=1)
```

```
Epoch 1/10
lr: 9.048374180359596e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.4627 - loss: 0.7538
5/5 ━━━━━━━━━━━━━━━━━━━━ 56s 12s/step - accuracy: 0.4700 - loss: 0.7487 - val_accuracy: 0.5729 - val_loss: 0.6486 - learning_rate: 1.0000e-04
Epoch 2/10
lr: 8.187307530779819e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.6384 - loss: 0.6155
5/5 ━━━━━━━━━━━━━━━━━━━━ 52s 12s/step - accuracy: 0.6435 - loss: 0.6123 - val_accuracy: 0.7760 - val_loss: 0.5276 - learning_rate: 9.0484e-05
Epoch 3/10
lr: 7.408182206817179e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8142 - loss: 0.4771
5/5 ━━━━━━━━━━━━━━━━━━━━ 53s 12s/step - accuracy: 0.8118 - loss: 0.4783 - val_accuracy: 0.7812 - val_loss: 0.4743 - learning_rate: 8.1873e-05
Epoch 4/10
lr: 6.703200460356393e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7915 - loss: 0.4520
5/5 ━━━━━━━━━━━━━━━━━━━━ 52s 12s/step - accuracy: 0.7835 - loss: 0.4554 - val_accuracy: 0.8229 - val_loss: 0.4389 - learning_rate: 7.4082e-05
Epoch 5/10
lr: 6.065306597126335e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7755 - loss: 0.4868
5/5 ━━━━━━━━━━━━━━━━━━━━ 52s 12s/step - accuracy: 0.7817 - loss: 0.4814 - val_accuracy: 0.7969 - val_loss: 0.4275 - learning_rate: 6.7032e-05
Epoch 6/10
lr: 5.488116360940264e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8421 - loss: 0.3920
5/5 ━━━━━━━━━━━━━━━━━━━━ 51s 11s/step - accuracy: 0.8392 - loss: 0.3938 - val_accuracy: 0.8229 - val_loss: 0.4312 - learning_rate: 6.0653e-05
Epoch 7/10
lr: 4.9658530379140954e-05━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7651 - loss: 0.4577
5/5 ━━━━━━━━━━━━━━━━━━━━ 51s 12s/step - accuracy: 0.7678 - loss: 0.4535 - val_accuracy: 0.8229 - val_loss: 0.3880 - learning_rate: 5.4881e-05
Epoch 8/10
lr: 4.493289641172216e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8369 - loss: 0.4091
5/5 ━━━━━━━━━━━━━━━━━━━━ 51s 12s/step - accuracy: 0.8349 - loss: 0.4100 - val_accuracy: 0.8281 - val_loss: 0.3793 - learning_rate: 4.9659e-05
Epoch 9/10
lr: 4.0656965974059915e-05━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8465 - loss: 0.3891
5/5 ━━━━━━━━━━━━━━━━━━━━ 50s 11s/step - accuracy: 0.8439 - loss: 0.3936 - val_accuracy: 0.8073 - val_loss: 0.3916 - learning_rate: 4.4933e-05
Epoch 10/10
lr: 3.6787944117144224e-05━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8058 - loss: 0.4248
5/5 ━━━━━━━━━━━━━━━━━━━━ 51s 12s/step - accuracy: 0.8049 - loss: 0.4268 - val_accuracy: 0.8438 - val_loss: 0.3733 - learning_rate: 4.0657e-05
```

Plot loss curves for training and validation sets (extract_feat_model)

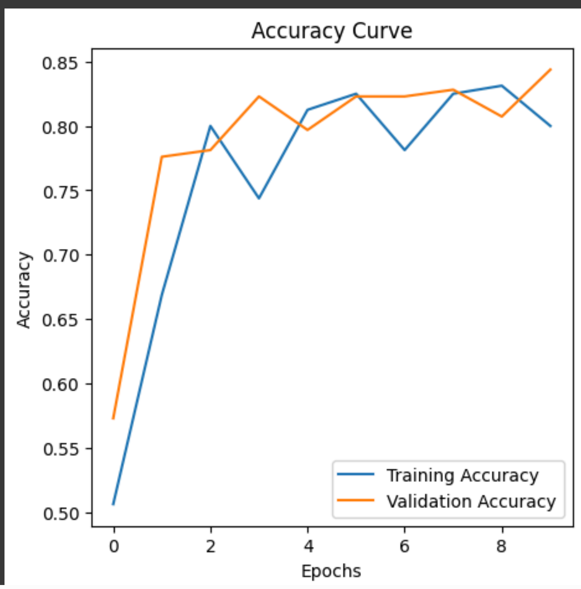
```
import matplotlib.pyplot as plt

history = extract_feat_model

# plot loss curve
plt.figure(figsize=(5, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



** Plot accuracy curves for training and validation sets (extract_feat_model)**

NOTE: As training is a stochastic process, the loss and accuracy graphs may differ across runs. As long as the general trend shows decreasing loss and increasing accuracy, the model is performing as expected and full marks will be awarded for the task.

```
import matplotlib.pyplot as plt

history = extract_feat_model
## Task 6: Plot accuracy curves for training and validation sets
plt.figure(figsize=(5, 5))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



Fine-Tuning model

Fine-tuning is an optional step in transfer learning, it usually ends up improving the performance of the model.

You will **unfreeze** one layer from the base model and train the model again.

```
from tensorflow.keras.applications import vgg16

input_shape = (150, 150, 3)
vgg = vgg16.VGG16(include_top=False,
                        weights='imagenet',
                        input_shape=input_shape)
output = vgg.layers[-1].output
output = tf.keras.layers.Flatten()(output)
basemodel = Model(vgg.input, output)
for layer in basemodel.layers:
    layer.trainable = False
display([layer.name for layer in basemodel.layers])
set_trainable = False
for layer in basemodel.layers:
    if layer.name in ['block5_conv3']:
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
for layer in basemodel.layers:
    print(f"{layer.name}: {layer.trainable}")
```

```
['input_layer_2',
 'block1_conv1',
 'block1_conv2',
 'block1_pool',
 'block2_conv1',
 'block2_conv2',
 'block2_pool',
 'block3_conv1',
 'block3_conv2',
 'block3_conv3',
 'block3_pool',
 'block4_conv1',
 'block4_conv2',
 'block4_conv3',
 'block4_pool',
 'block5_conv1',
 'block5_conv2',
 'block5_conv3',
 'block5_pool',
 'flatten_1']
input_layer_2: 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: True
block5_pool: True
flatten_1: True
```

Similar to what you did before, you will create a new model on top, and add a Dropout layer for regularization.

```
model = Sequential()
model.add(basemodel)
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

checkpoint_path='O_R_tlearn_fine_tune_vgg16.keras'

# learning schedule callback
loss_history_ = LossHistory_()
lrate_ = LearningRateScheduler(exp_decay)
keras_callbacks = [
      EarlyStopping(monitor = 'val_loss',
                    patience = 4,
                    mode = 'min',
                    min_delta=0.01),
      ModelCheckpoint(checkpoint_path, monitor='val_loss', save_best_only=True, mode='min')
]
callbacks_list_ = [loss_history_, lrate_] + keras_callbacks
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(learning_rate=1e-4),
              metrics=['accuracy'])
fine_tune_model = model.fit(train_generator,
                    steps_per_epoch=5,
                    epochs=10,
                    callbacks = callbacks_list_,
                    validation_data=val_generator,
                    validation_steps=val_generator.samples // batch_size,
                    verbose=1)
```

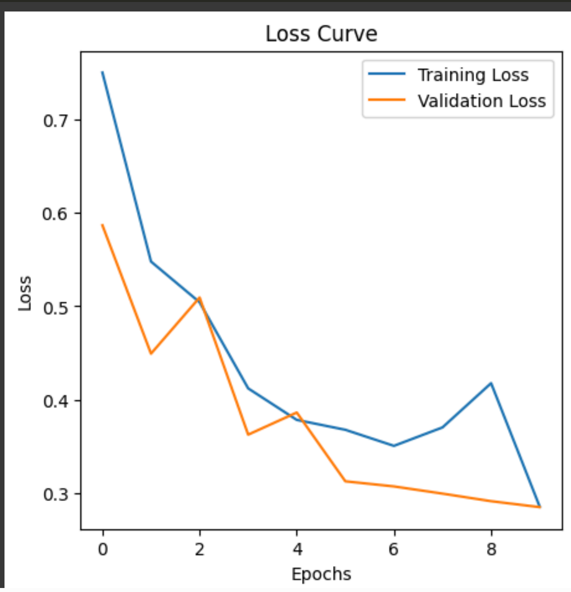
```
Epoch 1/10
lr: 9.048374180359596e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.6036 - loss: 0.7620
5/5 ━━━━━━━━━━━━━━━━━━━━ 60s 13s/step - accuracy: 0.6020 - loss: 0.7601 - val_accuracy: 0.6615 - val_loss: 0.5868 - learning_rate: 1.0000e-04
Epoch 2/10
lr: 8.187307530779819e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.6868 - loss: 0.5760
5/5 ━━━━━━━━━━━━━━━━━━━━ 56s 13s/step - accuracy: 0.6859 - loss: 0.5714 - val_accuracy: 0.8490 - val_loss: 0.4493 - learning_rate: 9.0484e-05
Epoch 3/10
lr: 7.408182206817179e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7725 - loss: 0.4945
5/5 ━━━━━━━━━━━━━━━━━━━━ 53s 12s/step - accuracy: 0.7667 - loss: 0.4961 - val_accuracy: 0.7240 - val_loss: 0.5094 - learning_rate: 8.1873e-05
Epoch 4/10
lr: 6.703200460356393e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8190 - loss: 0.4782
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 57s 13s/step - accuracy: 0.8241 - loss: 0.4672 - val_accuracy: 0.8490 - val_loss: 0.3626 - learning_rate: 7.4082e-05
Epoch 5/10
lr: 6.065306597126335e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8631 - loss: 0.3537
5/5 ━━━━━━━━━━━━━━━━━━━━ 53s 12s/step - accuracy: 0.8578 - loss: 0.3578 - val_accuracy: 0.7969 - val_loss: 0.3862 - learning_rate: 6.7032e-05
Epoch 6/10
lr: 5.488116360940264e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7985 - loss: 0.4742
5/5 ━━━━━━━━━━━━━━━━━━━━ 56s 13s/step - accuracy: 0.8092 - loss: 0.4565 - val_accuracy: 0.8542 - val_loss: 0.3126 - learning_rate: 6.0653e-05
Epoch 7/10
lr: 4.9658530379140954e-05━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8462 - loss: 0.3587
5/5 ━━━━━━━━━━━━━━━━━━━━ 54s 12s/step - accuracy: 0.8479 - loss: 0.3574 - val_accuracy: 0.8750 - val_loss: 0.3071 - learning_rate: 5.4881e-05
Epoch 8/10
lr: 4.493289641172216e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8672 - loss: 0.3206
5/5 ━━━━━━━━━━━━━━━━━━━━ 54s 12s/step - accuracy: 0.8633 - loss: 0.3289 - val_accuracy: 0.8802 - val_loss: 0.2994 - learning_rate: 4.9659e-05
Epoch 9/10
lr: 4.0656965974059915e-05━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8066 - loss: 0.3703
5/5 ━━━━━━━━━━━━━━━━━━━━ 55s 12s/step - accuracy: 0.8065 - loss: 0.3782 - val_accuracy: 0.8750 - val_loss: 0.2913 - learning_rate: 4.4933e-05
Epoch 10/10
lr: 3.678794411714424e-05━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.8839 - loss: 0.2748
5/5 ━━━━━━━━━━━━━━━━━━━━ 55s 13s/step - accuracy: 0.8824 - loss: 0.2766 - val_accuracy: 0.8646 - val_loss: 0.2849 - learning_rate: 4.0657e-05
```

** Plot loss curves for training and validation sets (fine tune model)**

NOTE: As training is a stochastic process, the loss and accuracy graphs may differ across runs. As long as the general trend shows decreasing loss and increasing accuracy, the model is performing as expected and full marks will be awarded for the task.

```
import matplotlib.pyplot as plt

history = fine_tune_model

## Task 7: Plot loss curves for training and validation sets (fine tune model)

plt.figure(figsize=(5, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

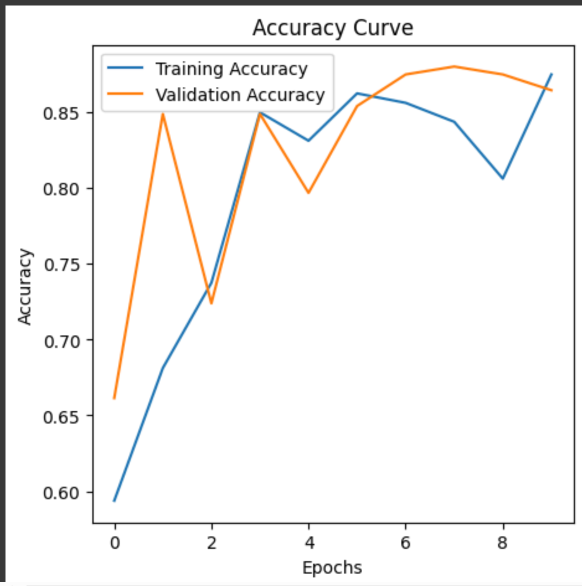


** Plot accuracy curves for training and validation sets (fine tune model)**

NOTE: As training is a stochastic process, the loss and accuracy graphs may differ across runs. As long as the general trend shows decreasing loss and increasing accuracy, the model is performing as expected and full marks will be awarded for the task.

```
import matplotlib.pyplot as plt
history = fine_tune_model

# Task 8: Plot accuracy curves for training and validation sets  (fine tune model)
plt.figure(figsize=(5, 5))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

```
plt.legend()

plt.show()
```



Evaluate both models on test data

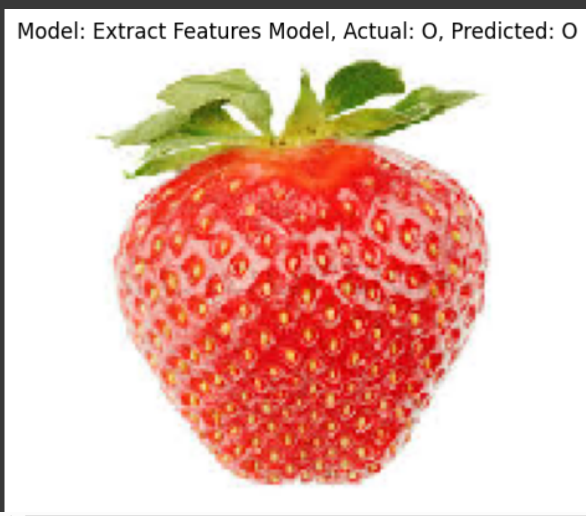
- Load saved models
- Load test images
- Make predictions for both models
- Convert predictions to class labels
- Print classification report for both models

```
from pathlib import Path

# Load saved models
extract_feat_model = tf.keras.models.load_model('O_R_tlearn_vgg16.keras')
fine_tune_model = tf.keras.models.load_model('O_R_tlearn_fine_tune_vgg16.keras')

IMG_DIM = (150, 150)

# Load test images
test_files_O = glob.glob('./o-vs-r-split/test/O/*')
test_files_R = glob.glob('./o-vs-r-split/test/R/*')
test_files = test_files_O[:50] + test_files_R[:50]

test_imgs = [tf.keras.preprocessing.image.img_to_array(tf.keras.preprocessing.image.load_img(img, target_size=IMG_DIM)) for img in test_files]
test_imgs = np.array(test_imgs)
test_labels = [Path(fn).parent.name for fn in test_files]

# Standardize
test_imgs_scaled = test_imgs.astype('float32')
test_imgs_scaled /= 255

class2num_lt = lambda l: [0 if x == 'O' else 1 for x in l]
num2class_lt = lambda l: ['O' if x < 0.5 else 'R' for x in l]

test_labels_enc = class2num_lt(test_labels)

# Make predictions for both models
predictions_extract_feat_model = extract_feat_model.predict(test_imgs_scaled, verbose=0)
predictions_fine_tune_model = fine_tune_model.predict(test_imgs_scaled, verbose=0)

# Convert predictions to class labels
predictions_extract_feat_model = num2class_lt(predictions_extract_feat_model)
predictions_fine_tune_model = num2class_lt(predictions_fine_tune_model)

# Print classification report for both models
print('Extract Features Model')
print(metrics.classification_report(test_labels, predictions_extract_feat_model))
print('Fine-Tuned Model')
print(metrics.classification_report(test_labels, predictions_fine_tune_model))

```

```
Extract Features Model
              precision    recall  f1-score   support

           O       0.70      0.90      0.79        50
           R       0.86      0.62      0.72        50
```

```
    accuracy                           0.76       100
   macro avg       0.78      0.76      0.76       100
weighted avg       0.78      0.76      0.76       100

Fine-Tuned Model
              precision    recall  f1-score   support

           O       0.78      0.90      0.83        50
           R       0.88      0.74      0.80        50

    accuracy                           0.82       100
   macro avg       0.83      0.82      0.82       100
weighted avg       0.83      0.82      0.82       100
```

```python
# Plot one of the images with actual label and predicted label as title
def plot_image_with_title(image, model_name, actual_label, predicted_label):
    plt.imshow(image)
    plt.title(f"Model: {model_name}, Actual: {actual_label}, Predicted: {predicted_label}")
    plt.axis('off')
    plt.show()

# Specify index of image to plot, for example index 0
index_to_plot = 0
plot_image_with_title(
    image=test_imgs[index_to_plot].astype('uint8'),
    model_name='Extract Features Model',
    actual_label=test_labels[index_to_plot],
    predicted_label=predictions_extract_feat_model[index_to_plot],
    )
```



** Plot a test image using Extract Features Model (index_to_plot = 1)**

NOTE: Due to the inherent nature of neural networks, predictions may vary from the actual labels. For instance, if the actual label is 'O', the prediction could be either 'O' or 'R', both of which are possible outcomes, and full marks will be awarded for the task.

```python
# Task 9: Plot a test image using Extract Features Model (index_to_plot = 1)

# Specify index of image to plot
index_to_plot = 1
plot_image_with_title(
    image=test_imgs[index_to_plot].astype('uint8'),
    model_name='Extract Features Model',
    actual_label=test_labels[index_to_plot],
    predicted_label=predictions_extract_feat_model[index_to_plot],
    )
```

Plot a test image using Fine-Tuned Model (index_to_plot = 1)

NOTE: Due to the inherent nature of neural networks, predictions may vary from the actual labels. For instance, if the actual label is 'O', the prediction could be either 'O' or 'R', both of which are possible outcomes, and full marks will be awarded for the task.

```
# Task 10: Plot a test image using Fine-Tuned Model (index_to_plot = 1)

# Specify index of image to plot
index_to_plot = 1
plot_image_with_title(
    image=test_imgs[index_to_plot].astype('uint8'),
    model_name='Fine-Tuned Model',
    actual_label=test_labels[index_to_plot],
    predicted_label=predictions_fine_tune_model[index_to_plot],
    )
```