Smart Sorting: Transfer Learning for Identifying Rotten Fruits and Vegetables

Smart Sorting is an innovative project focused on enhancing the precision and efficiency of detecting rotten fruits and vegetables using cutting-edge transfer learning techniques. By leveraging pre-trained deep learning models and adapting them to specific datasets of fruits and vegetables, this project aims to revolutionize the process of sorting and quality control in the agricultural and food industry.

Execution and Explanation:

Code Description: Image Classification Using MobileNetV2 and TensorFlow

This Python script sets up the foundational components for building an image classification model using TensorFlow and Keras. It leverages transfer learning by utilizing the MobileNetV2 architecture, a lightweight deep neural network optimized for mobile and embedded vision applications.

Key Components:

Importing Required Libraries:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
import numpy as np
import matplotlib.pyplot as plt
import os
```

- **TensorFlow/Keras**: Used for building and training the neural network.
- **ImageDataGenerator**: A Keras utility for loading and augmenting image data from directories.
- **MobileNetV2**: A pre-trained model used as a base for transfer learning.
- **preprocess_input**: Preprocessing function that prepares images in a format suitable for MobileNetV2.
- **NumPy**: For numerical operations.
- **Matplotlib**: For plotting results like training accuracy and loss.
- **os**: For file and path management.

Purpose and Next Steps:

This setup is typically followed by:

- Initializing ImageDataGenerator objects for training and validation datasets.
- Loading data from directories.
- Defining the model architecture by extending MobileNetV2.
- Compiling and training the model.
- Evaluating and visualizing performance.

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

loads the CIFAR-10 dataset, which is a standard benchmark dataset in computer vision. It consists of:

- **60,000 color images** (32x32 pixels)
- **10 classes** (e.g., airplane, car, bird, cat, etc.)
- Split into:
  - **50,000 training images** (x_train, y_train)

- **10,000 test images** (x_test, y_test)

Each label (y_train / y_test) is an integer from 0 to 9 representing the class.

```
def filter_classes(x, y, class_list):
    idx = np.isin(y, class_list).flatten()
    x_filtered = x[idx]
    y_filtered = y[idx]
    return x_filtered, y_filtered
```

filters a dataset to include only the samples whose labels are in class_list.

- x: Input data (e.g., images)
- y: Corresponding labels
- class_list: A list of class indices to keep

It returns only the **images and labels** that match the specified classes.

```
class_map = {0: 'fresh', 1: 'rotten'}
x_train, y_train = filter_classes(x_train, y_train, [0, 1])
x_test, y_test = filter_classes(x_test, y_test, [0, 1])
```

maps class labels 0 and 1 to human-readable names ('fresh' and 'rotten') and then:

- **Filters** both training and test datasets to only include samples labeled as class 0 or 1.

This is useful for binary classification tasks using a subset of a multi-class dataset.

```
x_train = tf.image.resize(x_train, [96, 96]) / 255.0
x_test = tf.image.resize(x_test, [96, 96]) / 255.0
```

resizes images in the training and test sets to 96x96 pixels and **normalizes pixel values** to the [0, 1] range by dividing by 255.0.

This prepares the data for models like MobileNetV2, which expect larger input sizes and normalized input.

```
y_train = tf.keras.utils.to_categorical(y_train, num_classes=2)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=2)
```

converts integer labels into **one-hot encoded vectors** for binary classification.

- Example: label 0 becomes [1, 0], and label 1 becomes [0, 1].
- This format is commonly used when training classification models with categorical_crossentropy.

```
train_datagen = ImageDataGenerator(
    horizontal_flip=True,
    rotation_range=20,
    zoom_range=0.2,
    preprocessing_function=preprocess_input
)
```

Creates a data generator for training images with real-time **data augmentation**, including:

- Random horizontal flips
- Small rotations (up to 20 degrees)
- Zoom transformations (up to 20%)
- MobileNetV2-compatible **preprocessing** via preprocess_input

This helps improve model generalization by simulating a wider variety of input images.

test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

Creates a test data generator that applies only **MobileNetV2-compatible preprocessing** (e.g., scaling and normalization), without any data augmentation.

This ensures consistent input formatting during model evaluation or prediction.

train_gen = train_datagen.flow(x_train, y_train, batch_size=32)

test_gen = test_datagen.flow(x_test, y_test, batch_size=32)

Creates data generators that yield batches of augmented (training) and preprocessed (testing) image data and labels:

- train_gen: Generates batches of 32 augmented training samples.
- test_gen: Generates batches of 32 preprocessed test samples.

These generators can be used directly with model.fit() or model.evaluate().

base_model = MobileNetV2(input_shape=(96, 96, 3), include_top=False, weights='imagenet')

base_model.trainable = False

Loads the MobileNetV2 model with:

- input_shape=(96, 96, 3): Accepts 96×96 RGB images.
- include_top=False: Excludes the fully connected classification layers.
- weights='imagenet': Uses pre-trained weights from ImageNet.

```
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(2, activation='softmax')
])
```

Builds a transfer learning model using MobileNetV2 as the base, followed by custom layers:

- GlobalAveragePooling2D: Reduces feature maps to a vector.
- Dense(128, activation='relu'): Learns high-level features.
- Dropout(0.3): Prevents overfitting.
- Dense(2, activation='softmax'): Outputs class probabilities for 2 classes.

This structure is suitable for **binary image classification** tasks.

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

Compiles the model with:

- optimizer='adam': An adaptive optimizer for efficient training.
- loss='categorical_crossentropy': Suitable for multi-class or one-hot encoded binary classification.
- metrics=['accuracy']: Tracks accuracy during training and evaluation.

model.summary() then **prints a detailed summary** of the model architecture, layer shapes, and parameter counts.

history = model.fit(train_gen, validation_data=test_gen, epochs=5)

Trains the model using the training data generator train_gen for 5 epochs and evaluates performance on the validation data (test_gen) after each epoch.

- The training process returns a history object that contains metrics like loss and accuracy over epochs.

loss, acc = model.evaluate(test_gen)

print(f"\nTest Accuracy: {acc:.2f}")

Evaluates the trained model on the test dataset using test_gen and prints the test accuracy formatted to two decimal places.

This step provides a final performance check of your model on unseen data.

class_names = ['fresh', 'rotten']

x_sample, y_sample = next(test_gen)

preds = model.predict(x_sample)

Performs prediction on a batch of test images:

- class_names: Maps class indices to readable labels.
- next(test_gen): Retrieves one batch of images and labels from the test generator.
- model.predict(x_sample): Produces predicted probabilities for each image in the batch.

This setup is typically used for visualizing predictions or evaluating individual examples.

```
x_sample, y_sample = next(test_gen)
preds = model.predict(x_sample)
```

etches one batch of images (x_sample) and labels (y_sample) from the test generator, then uses the trained model to **predict class probabilities** for that batch (preds).

This is typically done to analyze or visualize model predictions on unseen data.

```
plt.figure(figsize=(12, 8))
for i in range(9):
    plt.subplot(3, 3, i+1)
    plt.imshow((x_sample[i] * 0.5 + 0.5))  # Undo preprocessing a bit for visibility
    pred_label = class_names[np.argmax(preds[i])]
    true_label = class_names[np.argmax(y_sample[i])]
    color = 'green' if pred_label == true_label else 'red'
    plt.title(f"Pred: {pred_label}\nTrue: {true_label}", color=color)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Visualizes predictions on 9 sample test images in a 3×3 grid:

- Applies inverse preprocessing for better image display.
- Shows both predicted and true class labels.
- Colors the title **green if correct**, **red if incorrect**—helping to quickly spot classification errors.

Final Result:

So the final result will be the identification for fresh and rotten items which are accurately checked and observed very well.

The main role is played by tensorflow packages to identify the item and compare and separate it.

So, finally our required AIML model project is created with good accuracy.