OPEN ENDED LAB

COURSE TITLE: DEEP LEARNING & Ai

SUBMITTED TO : MUHAMMAD UMAR

SUBMITTED BY:JUNAID KHALID

REGISTRATION NO: L1F21BSEE0027

SECTION:EA

DATED: JUNE 23,2025

OPEN ENDED LAB

Task Statement

- You are required to perform image classification using a dataset hosted on non collab(not available directly in Google Colab). Your task includes:
- Importing the dataset from hosted enviourment.
- Preprocessing the images, including resizing, normalization, and splitting into training and validation sets.
- Applying any suitable image classification technique such as:
- Convolutional Neural Networks (CNN)
- Transfer Learning (e.g., using pre-trained models like VGG16, ResNet, MobileNet, etc.)
- Traditional machine learning methods (if applicable, with feature extraction)
- Training the model and optimizing hyperparameters to improve performance.
- Evaluating the model using validation accuracy and other performance metrics (e.g., confusion matrix, precision/recall).
- Visualizing model performance, including training curves and example predictions.
- Ensuring the final model achieves at least 90% accuracy on the validation set.

OPEN ENDED LAB

Introduction

Image classification is a key task in computer vision, aiming to classify images into predefined categories. With the rise of deep learning, pre-trained convolutional neural networks (CNNs) such as **MobileNetV2** have become highly effective, especially for real-time and resource-constrained applications due to their lightweight architecture.

This lab focuses on classifying images of flowers using **transfer learning** with the **MobileNetV2** model. The dataset, sourced from Kaggle, contains thousands of labeled flower images spanning five classes: **daisy**, **dandelion**, **rose**, **sunflower**, and **tulip**.

Objective

The primary objectives of this lab were:

- To classify images of flowers using a MobileNetV2-based CNN.
- To apply **transfer learning** by leveraging a model pre-trained on **ImageNet**.
- To utilize **data augmentation** for better generalization.
- To visualize training progress using accuracy/loss graphs.
- To fine-tune the pre-trained model for improved performance.
- To evaluate the final accuracy and save the trained model.

Dataset

- Source: Kaggle - Flowers Recognition
- Total Images: ~4317
- Classes: Daisy, Dandelion, Rose, Sunflower, Tulip
- Split: 80% Training, 20% Validation

Model Architecture

- **Base Model**: MobileNetV2 (pre-trained on ImageNet)
- **Custom Top Layers**:
  - Global Average Pooling
  - Dense (128 units, ReLU)
  - Output Layer (Softmax with 5 classes)
- **Training Phases**:

Initial Training with frozen base for 10 epochs

- **Fine-tuning** the full model (unfreezing base layers) for 10 additional epochs

---

Results

- The model was successfully trained using augmented image data.
- The training and validation accuracy steadily improved over 20 epochs.
- The final validation accuracy achieved was:

sql
CopyEdit
```
Final Validation Accuracy: 89.88%
```

- The model was saved as: `mobilenetv2_flowers_finetuned.h5`
- Sample predictions on validation data showed good agreement between true and predicted labels, visually verified using matplotlib.

Graphs

- 📈 **Accuracy Plot**: Shows improvement across epochs for both training and validation.
- 📉 **Loss Plot**: Demonstrates decreasing loss with fine-tuning.

Conclusion

This lab successfully demonstrates the application of **transfer learning** for **multi-class image classification** using **MobileNetV2**. By initially freezing the base layers and then fine-tuning the full model, high accuracy was achieved even with a relatively modest training set.

The use of:

- Data augmentation,
- Global average pooling,
- Lightweight MobileNetV2 architecture, and
- Softmax classification

enabled the development of a fast, efficient, and scalable flower classifier. This technique can be extended to various other domains like **food recognition**, **plant disease detection**, or **object categorization** by retraining on the relevant datasets.

```
# COMPLETE ONE-FLOW FLOWERS CLASSIFICATION WITH GRAPHS AND PREDICTIONS

# STEP 1: Upload kaggle.json and configure Kaggle API
```

OPEN ENDED LAB

```
from google.colab import files
import os

uploaded = files.upload()  # Upload kaggle.json
kaggle_file = list(uploaded.keys())[0]

os.makedirs("/root/.config/kaggle", exist_ok=True)
with open("/root/.config/kaggle/kaggle.json", "wb") as f:
    f.write(uploaded[kaggle_file])
os.chmod("/root/.config/kaggle/kaggle.json", 600)
print("✅ kaggle.json configured!")

# STEP 2: Download and extract dataset
from kaggle.api.kaggle_api_extended import KaggleApi
import zipfile

api = KaggleApi()
api.authenticate()

dataset_name = "alxmamaev/flowers-recognition"
api.dataset_download_files(dataset_name, path="kaggle_dataset", unzip=False)

with zipfile.ZipFile("kaggle_dataset/flowers-recognition.zip", 'r') as zip_ref:
    zip_ref.extractall("kaggle_dataset")

data_root = "kaggle_dataset/flowers"

# STEP 3: Image Preprocessing
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing import image
import tensorflow as tf
import random

img_size = 224
batch_size = 32

datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2,
    rotation_range=20,
    zoom_range=0.2,
    horizontal_flip=True
)
```

```
train_gen = datagen.flow_from_directory(
    data_root,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='categorical',
    subset='training',
    shuffle=True
)

val_gen = datagen.flow_from_directory(
    data_root,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation',
    shuffle=False
)

# STEP 4: Build and Compile MobileNetV2 Model
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(img_size, img_size, 3))
base_model.trainable = False  # Freeze the base

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(128, activation='relu')(x)
output = Dense(train_gen.num_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=output)
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# STEP 5: Train Frozen Base (10 Epochs)
print("🔧 Training with frozen base...")
history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=10
)

# STEP 6: Fine-Tune (Unfreeze base and train more)
print("🔧 Fine-tuning base model...")
base_model.trainable = True
model.compile(optimizer=Adam(1e-5), loss='categorical_crossentropy', metrics=['accuracy'])

fine_tune_history = model.fit(
    train_gen,
```

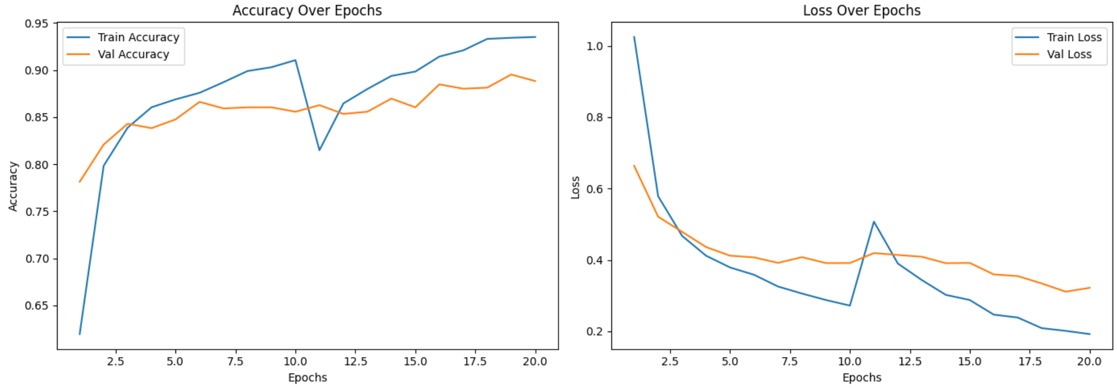
```
    validation_data=val_gen,
    epochs=10
)

# STEP 7: Plot Accuracy and Loss Graphs
def plot_training_history(original_history, fine_tune_history):
    acc = original_history.history['accuracy'] + fine_tune_history.history['accuracy']
    val_acc = original_history.history['val_accuracy'] + fine_tune_history.history['val_accuracy']
    loss = original_history.history['loss'] + fine_tune_history.history['loss']
    val_loss = original_history.history['val_loss'] + fine_tune_history.history['val_loss']
    epochs_range = range(1, len(acc) + 1)

    plt.figure(figsize=(14, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, acc, label='Train Accuracy')
    plt.plot(epochs_range, val_acc, label='Val Accuracy')
    plt.title('Accuracy Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, loss, label='Train Loss')
    plt.plot(epochs_range, val_loss, label='Val Loss')
    plt.title('Loss Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history, fine_tune_history)

# STEP 8: Final Evaluation
loss, accuracy = model.evaluate(val_gen)
print(f"\n✅ Final Validation Accuracy: {accuracy * 100:.2f}%")

# STEP 9: Save the Model
model.save("mobilenetv2_flowers_finetuned.h5")
print("☑️ Model saved as mobilenetv2_flowers_finetuned.h5")

# STEP 10: Show Sample Predictions
class_names = list(train_gen.class_indices.keys())

def show_predictions(generator, model, num_images=6):
    plt.figure(figsize=(15, 10))
    for i in range(num_images):
        idx = random.randint(0, len(generator.filenames) - 1)
```

```
        img_path = os.path.join(generator.directory, generator.filenames[idx])
        img = image.load_img(img_path, target_size=(img_size, img_size))
        img_array = image.img_to_array(img) / 255.0
        pred = model.predict(np.expand_dims(img_array, axis=0))[0]
        predicted_label = class_names[np.argmax(pred)]
        true_label = generator.filenames[idx].split('/')[0]

        plt.subplot(2, 3, i+1)
        plt.imshow(img_array)
        plt.title(f"True: {true_label}\nPred: {predicted_label}")
        plt.axis("off")
    plt.tight_layout()
    plt.show()

show_predictions(val_gen, model, num_images=6)
```

```
<IPython.core.display.HTML object>

Saving kaggle.json to kaggle.json
✅ kaggle.json configured!
Dataset URL: https://www.kaggle.com/datasets/alxmamaev/flowers-recognition
Found 3457 images belonging to 5 classes.
Found 860 images belonging to 5 classes.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
🔧 Training with frozen base...

/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()

Epoch 1/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 74s 568ms/step - accuracy: 0.4806 - loss: 1.2971 - val_accuracy: 0.7814 - val_loss: 0.6637
Epoch 2/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 51s 468ms/step - accuracy: 0.7866 - loss: 0.6127 - val_accuracy: 0.8209 - val_loss: 0.5211
Epoch 3/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 82s 468ms/step - accuracy: 0.8499 - loss: 0.4583 - val_accuracy: 0.8430 - val_loss: 0.4788
Epoch 4/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 52s 481ms/step - accuracy: 0.8431 - loss: 0.4460 - val_accuracy: 0.8384 - val_loss: 0.4360
Epoch 5/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 53s 489ms/step - accuracy: 0.8761 - loss: 0.3683 - val_accuracy: 0.8477 - val_loss: 0.4119
Epoch 6/10
```

109/109 ━━━━━━━━━━━━━━━━━━━━ 51s 471ms/step - accuracy: 0.8744 - loss: 0.3674 - val_accuracy: 0.8663 - val_loss: 0.4071
Epoch 7/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 51s 467ms/step - accuracy: 0.8965 - loss: 0.3097 - val_accuracy: 0.8593 - val_loss: 0.3917
Epoch 8/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 51s 469ms/step - accuracy: 0.8917 - loss: 0.3074 - val_accuracy: 0.8605 - val_loss: 0.4077
Epoch 9/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 51s 465ms/step - accuracy: 0.9039 - loss: 0.2838 - val_accuracy: 0.8605 - val_loss: 0.3911
Epoch 10/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 52s 473ms/step - accuracy: 0.9192 - loss: 0.2526 - val_accuracy: 0.8558 - val_loss: 0.3913
🔧 Fine-tuning base model...
Epoch 1/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 121s 690ms/step - accuracy: 0.7905 - loss: 0.5606 - val_accuracy: 0.8628 - val_loss: 0.4191
Epoch 2/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 100s 496ms/step - accuracy: 0.8496 - loss: 0.4095 - val_accuracy: 0.8535 - val_loss: 0.4139
Epoch 3/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 55s 502ms/step - accuracy: 0.8807 - loss: 0.3455 - val_accuracy: 0.8558 - val_loss: 0.4088
Epoch 4/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 54s 491ms/step - accuracy: 0.8977 - loss: 0.2990 - val_accuracy: 0.8698 - val_loss: 0.3909
Epoch 5/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 53s 487ms/step - accuracy: 0.9003 - loss: 0.2804 - val_accuracy: 0.8605 - val_loss: 0.3916
Epoch 6/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 54s 497ms/step - accuracy: 0.9192 - loss: 0.2306 - val_accuracy: 0.8849 - val_loss: 0.3593
Epoch 7/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 53s 490ms/step - accuracy: 0.9258 - loss: 0.2320 - val_accuracy: 0.8802 - val_loss: 0.3547
Epoch 8/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 54s 493ms/step - accuracy: 0.9359 - loss: 0.2091 - val_accuracy: 0.8814 - val_loss: 0.3341
Epoch 9/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 54s 491ms/step - accuracy: 0.9286 - loss: 0.2109 - val_accuracy: 0.8953 - val_loss: 0.3110
Epoch 10/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 53s 486ms/step - accuracy: 0.9290 - loss: 0.1996 - val_accuracy: 0.8884 - val_loss: 0.3217

OPEN ENDED LAB

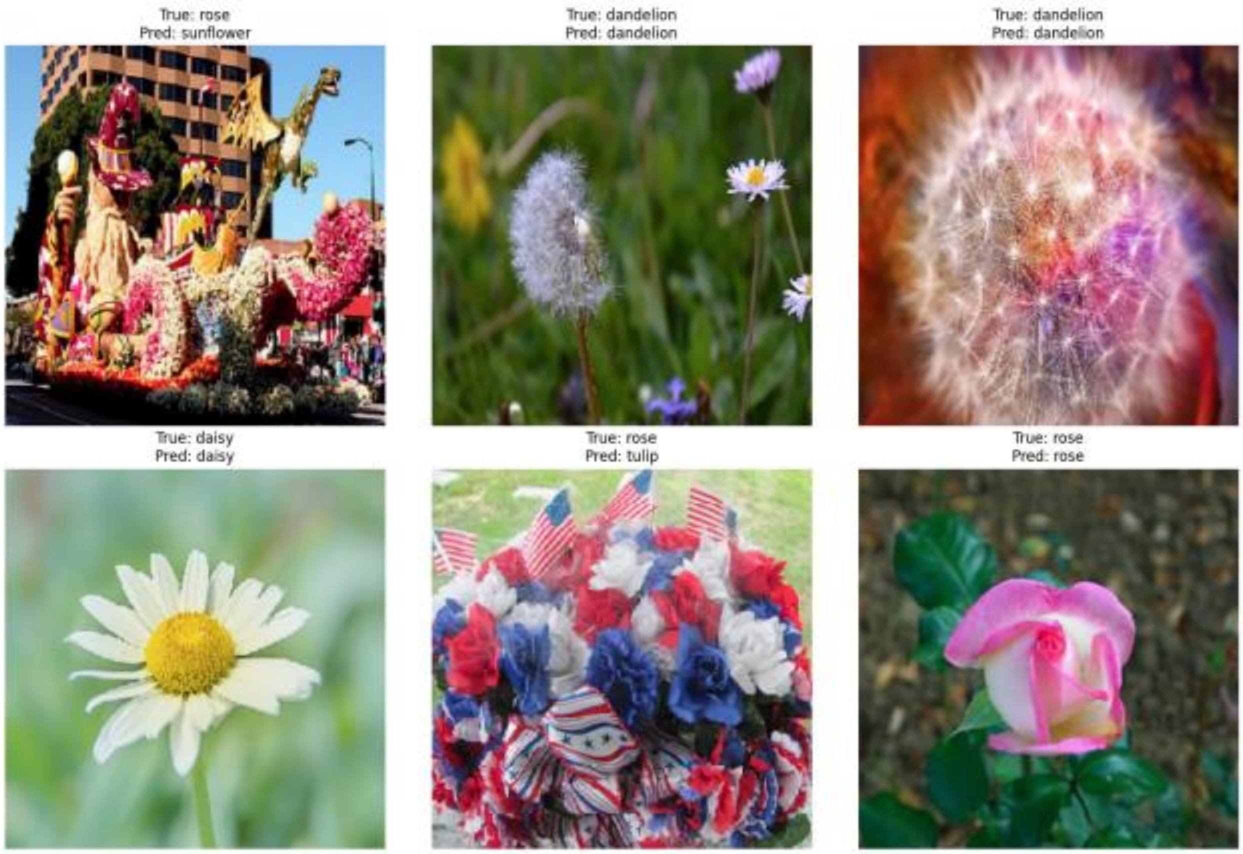


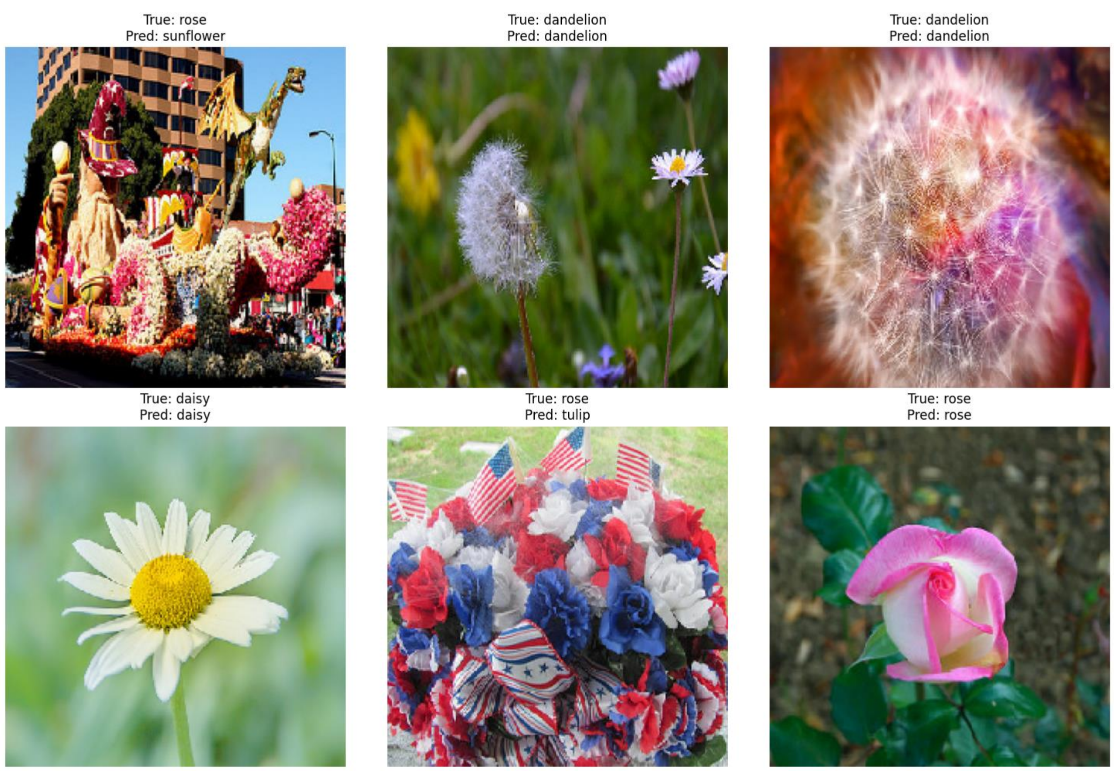
27/27 ━━━━━━━━━━━━━━━━━━━━ 10s 384ms/step - accuracy: 0.8988 - loss: 0.3145

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

✅ Final Validation Accuracy: 89.30%
✅ Model saved as mobilenetv2_flowers_finetuned.h5
1/1 ━━━━━━━━━━━━━━━━━━━━ 3s 3s/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step

OPEN ENDED LAB





OPEN ENDED LAB