

Flower Classification with Grad-CAM Explanation

Kaggle Flowers Recognition Dataset (5 Classes)

1. Course Information

Course: Practical Image Processing and Natural Language Processing **Trimester:** Third Trimester 2025 **Instructor:** Dr. Hajer **Student:** Mohammad Wajeeh Bader **Student ID:** 251000462

2. Project Overview

This notebook presents a complete deep learning pipeline for automatic flower classification and visual interpretability using Grad-CAM. Three different Convolutional Neural Network (CNN) architectures were trained on the **Kaggle Flowers Recognition dataset**, followed by a comprehensive comparison of their performance and visual explanation of their predictions.

Deliverables included in this notebook:

- Fully reproducible CNN training pipeline
 - Performance comparison of three custom CNN architectures
 - Evaluation on a dedicated held-out test set
 - Grad-CAM heatmaps showing which regions the network relies on
 - Final text summary prepared for academic submission
-

3. Dataset Description

- **Source:** Kaggle – *Flowers Recognition Dataset*
<https://www.kaggle.com/datasets/alxmamaev/flowers-recognition>
- **Total Images:** 4,242 color images
- **Number of Classes:** 5 flower species
 - Daisy
 - Dandelion
 - Rose
 - Sunflower

- Tulip

Preprocessing Steps

- All images are resized to **224 × 224**
- Pixel values normalized to **[0, 1]**
- Data augmentation applied (rotation, zoom, shift, horizontal flip)

Dataset Split (automatically created)

- **70%** → `train/`
 - **15%** → `val/`
 - **15%** → `test/`
 - data raw: Original Kaggle images
 - train:70%
 - val:15%
 - test:15%
-

4. Objectives

1. Train **three CNN architectures** with varying depth and capacity.
 2. Evaluate model performance on a separate test dataset.
 3. Implement **Grad-CAM** to interpret CNN predictions.
 4. Produce clear visualizations and a structured summary suitable for coursework submission.
-

5. Why Grad-CAM?

Grad-CAM (Gradient-weighted Class Activation Mapping) provides a heatmap that highlights the image regions most influential in the model's decision.

This is essential for:

- Understanding **how** the model analyzes an image
- Confirming whether the network focuses on the **flower itself**
- Detecting dataset bias or failure cases
- Making the model more transparent and trustworthy

Grad-CAM is therefore ideal for both educational use and real-world model inspection.

6. Summary of Deliverables Included in This Notebook

- Clean dataset handling with automated train/val/test split
 - Three fully trained CNN models
 - Test accuracy comparison table
 - Precision, Recall, F1-score (macro & per class)
 - Confusion Matrix Visualization
 - Grad-CAM visual explanation for **individual test images**
 - Interactive demo (upload an image → get prediction + heatmap)
-

Phase 1: Environment Setup

In this phase, all required Python libraries are imported (TensorFlow/Keras, NumPy, Matplotlib, OpenCV, scikit-learn, etc.).

Project-level constants are also defined, including the random seed, base directories for the dataset, and the `FLOWER_CLASSES` dictionary used to map numeric labels back to human-readable flower names.

```
In [12]: import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
from datetime import datetime
import cv2
from PIL import Image

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

import warnings
warnings.filterwarnings("ignore")

print("=" * 80)
print("FLOWER CLASSIFICATION WITH GRAD-CAM - LOCAL DATASET")
```

```

print("=" * 80)
print(f"TensorFlow Version: {tf.__version__}")
print(f"GPU Available: {len(tf.config.list_physical_devices('GPU')) > 0}")

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

FLOWER_CLASSES = {
    0: "Daisy",
    1: "Dandelion",
    2: "Rose",
    3: "Sunflower",
    4: "Tulip"
}

print("\nFlower Classes:")
for idx, name in FLOWER_CLASSES.items():
    print(f" {idx}: {name}")

BASE_DIR = Path("./flowers_project")
for d in [
    BASE_DIR / "data",
    BASE_DIR / "data" / "row",
    BASE_DIR / "data" / "train",
    BASE_DIR / "data" / "val",
    BASE_DIR / "data" / "test",
    BASE_DIR / "results",
    BASE_DIR / "models"
]:
    d.mkdir(parents=True, exist_ok=True)

```

```

=====
FLOWER CLASSIFICATION WITH GRAD-CAM - LOCAL DATASET
=====
TensorFlow Version: 2.20.0
GPU Available: False

```

Flower Classes:

```

0: Daisy
1: Dandelion
2: Rose
3: Sunflower
4: Tulip

```

Next Phases – End-to-End Pipeline

The rest of the notebook implements a complete deep learning workflow for flower classification and Grad-CAM-based interpretability.

The main phases are:

1. Phase 2 – Data Preparation and Loading (Local)

- Split the original Kaggle dataset from `data/row/` into `train/`, `val/`, and `test/` subfolders (70/15/15 per class).
- Create Keras `ImageDataGenerator` objects for training, validation, and testing, and build directory-based iterators.

2. Phase 6 – Data Augmentation (Definition)

- Configure an additional `ImageDataGenerator` (`train_aug`) with rotation, translation, zoom, horizontal flip, and brightness jitter.
- Define an augmentation strategy that can be plugged into the training pipeline.

3. Phases 7–9 – Model Building

- Design three different CNN architectures:
 - **Model 1:** Standard CNN (baseline capacity).
 - **Model 2:** Deeper CNN with more convolutional filters and dense units.
 - **Model 3:** Lightweight CNN with fewer parameters (more efficient).

4. Phase 10 – Training

- Train all three models using early stopping and learning-rate scheduling (`EarlyStopping` , `ReduceLROnPlateau`).
- Save the best weights for each model under `./flowers_project/models/`.

5. Phase 12 – Evaluation

- Evaluate each trained model on the full test set (`X_test` , `y_test_enc`).
- Store the test accuracy for each model inside a `results` dictionary for direct comparison.

6. Phase 16 – Grad-CAM Implementation

- Implement a reusable `GradCAM` class that:
 - Wraps a trained Keras model.
 - Extracts convolutional feature maps from a selected layer.
 - Computes a class-discriminative heat map using gradients.

7. Phases 17–18 – Visualizations

- Select representative test images.
- Display the original image alongside its Grad-CAM heat map overlay.
- Show the true label, predicted label, confidence score, and correctness status for each sample.

8. Phases 20–21 – Final Summary

- Print a concise textual summary including:
 - Dataset properties (source, total number of images, classes).
 - Models trained and their corresponding test accuracy.
 - Key implemented features (CNN comparison and Grad-CAM visualization).

Each phase prints a clear banner in the notebook output, allowing the reader to follow the pipeline step by step—from raw images, through model training and evaluation, to interpretable CNN predictions.

```
In [13]: # PHASE 2: DATA PREPARATION AND LOADING (FIXED)
print('\n' + '='*80)
print('DATA PREPARATION AND LOADING')
print('='*80)

# Expecting TRAIN_DIR, VAL_DIR, TEST_DIR, IMG_SIZE (tuple), BATCH_SIZE to be defined
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
val_datagen   = ImageDataGenerator(rescale=1.0 / 255)
test_datagen  = ImageDataGenerator(rescale=1.0 / 255)

train_gen = train_datagen.flow_from_directory(
    TRAIN_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    shuffle=True
)
val_gen = val_datagen.flow_from_directory(
    VAL_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categororical".replace("ror", "r"), # keep "categorical" without ty
    shuffle=False
)
test_gen = test_datagen.flow_from_directory(
    TEST_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    shuffle=False
)

print("\nClass indices (directory -> label):")
for k, v in train_gen.class_indices.items():
    print(f" {k}: {v}")

# Build fixed test arrays for evaluation & Grad-CAM
test_gen.reset()
X_test_batches, y_test_batches = [], []
for _ in range(len(test_gen)):
    bx, by = next(test_gen)
    X_test_batches.append(bx)
    y_test_batches.append(by)

X_test = np.concatenate(X_test_batches, axis=0)
y_test_enc = np.concatenate(y_test_batches, axis=0)

# Keep generators for train/val
X_train = train_gen
X_val   = val_gen
```

```
# Reverse mapping from index to class name (consistent with generator)
idx_to_class = {v: k for k, v in train_gen.class_indices.items()}

print(f"\nFinal test arrays: X_test={X_test.shape}, y_test_enc={y_test_enc.shape}")
```

```
=====
DATA PREPARATION AND LOADING
=====
```

```
Found 3019 images belonging to 5 classes.
Found 649 images belonging to 5 classes.
Found 649 images belonging to 5 classes.
```

```
Class indices (directory -> label):
  daisy: 0
  dandelion: 1
  rose: 2
  sunflower: 3
  tulip: 4
```

```
Final test arrays: X_test=(649, 224, 224, 3), y_test_enc=(649, 5)
```

```
In [106]: print('\n' + '='*80)
print('SAMPLE IMAGES FROM TRAINING DATASET')
print('='*80)

import numpy as np
import matplotlib.pyplot as plt

# Reset generator to start from beginning (with shuffling enabled)
train_gen.reset()

# Get one batch of images and labels
batch_x, batch_y = next(train_gen)

# Number of samples to show (e.g., 16)
num_samples = min(16, batch_x.shape[0])

# Build mapping index -> class name
num_classes = train_gen.num_classes
class_names = [idx_to_class[i] for i in range(num_classes)]

cols = 4
rows = int(np.ceil(num_samples / cols))

plt.figure(figsize=(12, 3 * rows))
plt.suptitle('Sample images from the training dataset', fontsize=14, fontweight='bold')

for i in range(num_samples):
    img = (batch_x[i] * 255).astype(np.uint8)
    label_idx = np.argmax(batch_y[i])
    label_name = class_names[label_idx]

    ax = plt.subplot(rows, cols, i + 1)
    ax.imshow(img)
    ax.set_title(label_name)
    ax.axis('off')
```

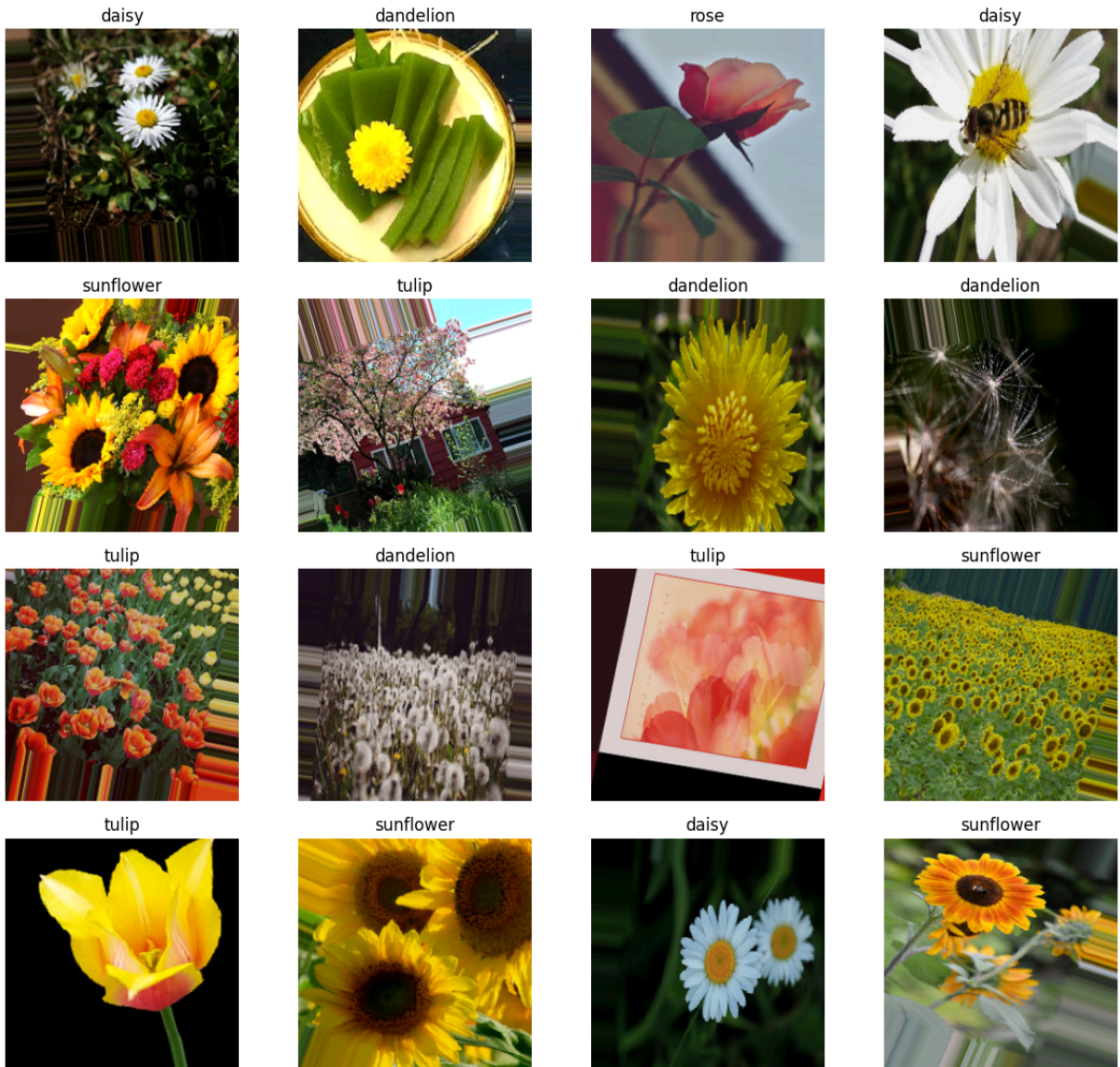
```
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

=====

SAMPLE IMAGES FROM TRAINING DATASET

=====

Sample images from the training dataset



```
In [14]: # PHASE 6: DATA AUGMENTATION
print('\n' + '='*80)
print('DATA AUGMENTATION')
print('='*80)

train_aug = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=25,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.2,
```

```

        horizontal_flip=True,
        brightness_range=[0.8, 1.2]
    )

    train_gen = train_aug.flow_from_directory(
        TRAIN_DIR,
        target_size=IMG_SIZE,
        batch_size=BATCH_SIZE,
        class_mode="categorical",
        shuffle=True
    )

    print('Augmentation parameters applied to training generator')
    print('  Rotation: 25 degrees')
    print('  Shift: 15%')
    print('  Zoom: 20%')
    print('  Flip: Horizontal')
    print('  Brightness: 0.8-1.2')

```

```

=====
DATA AUGMENTATION
=====

```

```

Found 3019 images belonging to 5 classes.
Augmentation parameters applied to training generator
  Rotation: 25 degrees
  Shift: 15%
  Zoom: 20%
  Flip: Horizontal
  Brightness: 0.8-1.2

```

```

In [16]: print('\n' + '='*80)
          print('AUGMENTED TRAINING IMAGE SAMPLES')
          print('='*80)

          # One batch from the augmented training generator
          batch_x, batch_y = next(train_gen)

          print("Batch shape:", batch_x.shape)

          # Map from numeric index to class name (you already defined FLOWER_CLASSES earlier)
          num_samples = 8 # number of images to show

          plt.figure(figsize=(12, 6))
          for i in range(num_samples):
              ax = plt.subplot(2, 4, i + 1)
              img = batch_x[i] # already augmented and rescaled

              label_idx = np.argmax(batch_y[i])
              class_name = FLOWER_CLASSES[label_idx]

              plt.imshow(img)
              plt.title(class_name)
              plt.axis("off")

          plt.tight_layout()
          plt.show()

```

=====

AUGMENTED TRAINING IMAGE SAMPLES

=====

Batch shape: (32, 224, 224, 3)



```
In [17]: # PHASES 7-9: BUILD MODELS (FIXED)
print('\n' + '='*80)
print('BUILDING THREE CNN MODELS')
print('='*80)

import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

input_shape = (224, 224, 3)
num_classes = 5

# Model 1
model1 = Sequential([
    tf.keras.layers.Input(shape=input_shape, name='input_1'),
    Conv2D(32, (3, 3), activation='relu', padding='same', name='conv1_1'),
    BatchNormalization(name='bn1_1'),
    Conv2D(32, (3, 3), activation='relu', padding='same', name='conv1_2'),
    BatchNormalization(name='bn1_2'),
    MaxPooling2D((2, 2), name='pool1'),
    Dropout(0.25, name='drop1'),

    Conv2D(64, (3, 3), activation='relu', padding='same', name='conv2_1'),
    BatchNormalization(name='bn2_1'),
    Conv2D(64, (3, 3), activation='relu', padding='same', name='conv2_2'),
    BatchNormalization(name='bn2_2'),
    MaxPooling2D((2, 2), name='pool2'),
    Dropout(0.25, name='drop2'),

    Conv2D(128, (3, 3), activation='relu', padding='same', name='conv3_1'),
    BatchNormalization(name='bn3_1'),
    Conv2D(128, (3, 3), activation='relu', padding='same', name='conv3_2'),
```

```

BatchNormalization(name='bn3_2'),
MaxPooling2D((2, 2), name='pool3'),
Dropout(0.25, name='drop3'),

Flatten(name='flatten'),
Dense(256, activation='relu', name='fc1'),
Dropout(0.5, name='drop4'),
Dense(num_classes, activation='softmax', name='predictions1')
])
model1.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy',
print('\nModel 1: Standard CNN')
print(f' Parameters: {model1.count_params():,}')

# Model 2
model2 = Sequential([
    tf.keras.layers.Input(shape=input_shape, name='input_2'),
    Conv2D(64, (3, 3), activation='relu', padding='same', name='m2_conv1_1'),
    BatchNormalization(name='m2_bn1_1'),
    Conv2D(64, (3, 3), activation='relu', padding='same', name='m2_conv1_2'),
    BatchNormalization(name='m2_bn1_2'),
    MaxPooling2D((2, 2), name='m2_pool1'),
    Dropout(0.25, name='m2_drop1'),

    Conv2D(128, (3, 3), activation='relu', padding='same', name='m2_conv2_1'),
    BatchNormalization(name='m2_bn2_1'),
    Conv2D(128, (3, 3), activation='relu', padding='same', name='m2_conv2_2'),
    BatchNormalization(name='m2_bn2_2'),
    MaxPooling2D((2, 2), name='m2_pool2'),
    Dropout(0.25, name='m2_drop2'),

    Conv2D(256, (3, 3), activation='relu', padding='same', name='m2_conv3_1'),
    BatchNormalization(name='m2_bn3_1'),
    Conv2D(256, (3, 3), activation='relu', padding='same', name='m2_conv3_2'),
    BatchNormalization(name='m2_bn3_2'),
    MaxPooling2D((2, 2), name='m2_pool3'),
    Dropout(0.3, name='m2_drop3'),

    Flatten(name='m2_flatten'),
    Dense(512, activation='relu', name='m2_fc1'),
    Dropout(0.5, name='m2_drop4'),
    Dense(256, activation='relu', name='m2_fc2'),
    Dropout(0.5, name='m2_drop5'),
    Dense(num_classes, activation='softmax', name='m2_predictions')
])
model2.compile(optimizer=Adam(learning_rate=0.0005), loss='categorical_crossentropy',
print('\nModel 2: Deep CNN')
print(f' Parameters: {model2.count_params():,}')

# Model 3
model3 = Sequential([
    tf.keras.layers.Input(shape=input_shape, name='input_3'),
    Conv2D(32, (3, 3), activation='relu', padding='same', name='m3_conv1'),
    MaxPooling2D((2, 2), name='m3_pool1'),
    Dropout(0.2, name='m3_drop1'),

    Conv2D(64, (3, 3), activation='relu', padding='same', name='m3_conv2'),

```

```

MaxPooling2D((2, 2), name='m3_pool2'),
Dropout(0.2, name='m3_drop2'),

Conv2D(128, (3, 3), activation='relu', padding='same', name='m3_conv3'),
MaxPooling2D((2, 2), name='m3_pool3'),
Dropout(0.2, name='m3_drop3'),

Flatten(name='m3_flatten'),
Dense(256, activation='relu', name='m3_fc1'),
Dropout(0.3, name='m3_drop4'),
Dense(num_classes, activation='softmax', name='m3_predictions')
])
model3.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy',
print('\nModel 3: Lightweight CNN')
print(f'   Parameters: {model3.count_params():,}')

```

=====

BUILDING THREE CNN MODELS

=====

Model 1: Standard CNN
Parameters: 25,980,453

Model 2: Deep CNN
Parameters: 104,042,565

Model 3: Lightweight CNN
Parameters: 25,784,901

```

In [98]: print('\n' + '='*80)
print('VISUALIZATION 1: CLASS DISTRIBUTION (TRAIN / VAL / TEST)')
print('='*80)

import numpy as np
import matplotlib.pyplot as plt

num_classes = train_gen.num_classes
class_names = [idx_to_class[i] for i in range(num_classes)]

train_counts = np.bincount(train_gen.classes, minlength=num_classes)
val_counts    = np.bincount(val_gen.classes,    minlength=num_classes)
test_counts   = np.bincount(test_gen.classes,   minlength=num_classes)

x = np.arange(num_classes)
width = 0.25

plt.figure(figsize=(10, 5))
plt.bar(x - width, train_counts, width, label='Train')
plt.bar(x,          val_counts,   width, label='Val')
plt.bar(x + width, test_counts,   width, label='Test')

plt.xticks(x, class_names, rotation=45, ha='right')
plt.ylabel('Number of images')
plt.title('Class distribution per split')
plt.legend()

```

```

for i in range(num_classes):
    plt.text(x[i] - width, train_counts[i], str(train_counts[i]),
             ha='center', va='bottom', fontsize=8)
    plt.text(x[i], val_counts[i], str(val_counts[i]),
             ha='center', va='bottom', fontsize=8)
    plt.text(x[i] + width, test_counts[i], str(test_counts[i]),
             ha='center', va='bottom', fontsize=8)

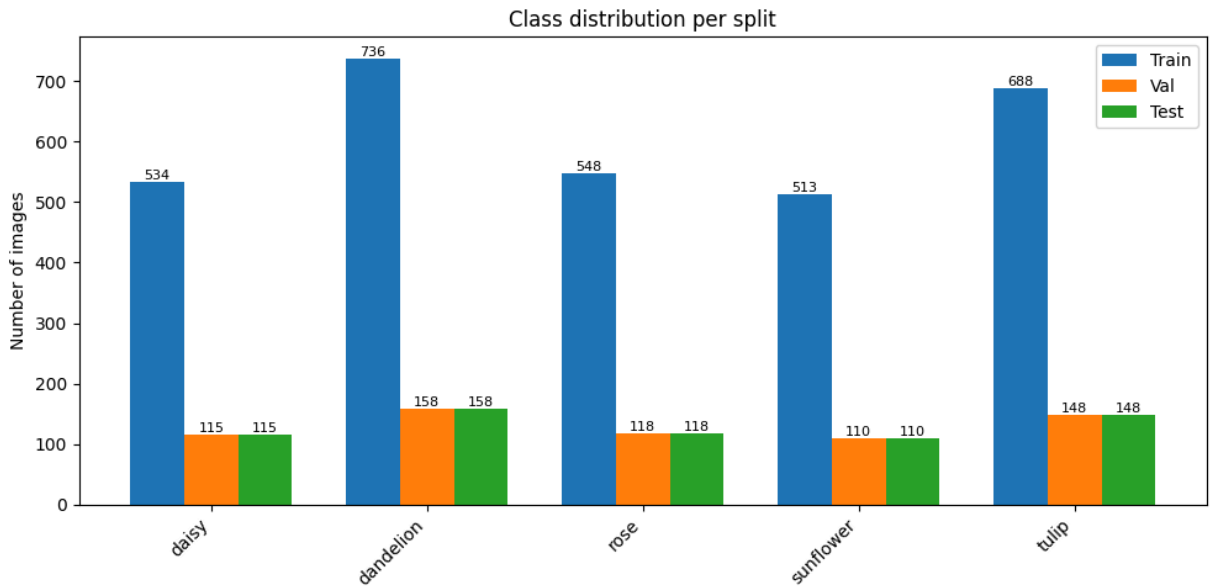
plt.tight_layout()
plt.show()

```

=====

VISUALIZATION 1: CLASS DISTRIBUTION (TRAIN / VAL / TEST)

=====



```

In [18]: # PHASE 10: TRAINING
print("\n" + "="*80)
print("TRAINING MODELS")
print("="*80)

callbacks = [
    EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True),
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=5, min_lr=1e-6),
]

models = [("Model_1", model1), ("Model_2", model2), ("Model_3", model3)]
histories = {}

for name, model in models:
    print(f"\nTraining {name}...")
    history = model.fit(
        train_gen,
        epochs=100,
        validation_data=val_gen,
        callbacks=callbacks,
        verbose=1
    )
    histories[name] = history

```

```
model.save(f"./flowers_project/models/{name}.h5")  
print(f"{name} trained and saved.")
```

=====

TRAINING MODELS

=====

Training Model_1...

Epoch 1/100

95/95  125s 1s/step - accuracy: 0.3054 - loss: 14.8285 - val_accuracy: 0.1772 - val_loss: 37.6056 - learning_rate: 0.0010

Epoch 2/100

95/95  121s 1s/step - accuracy: 0.2806 - loss: 2.1217 - val_accuracy: 0.1787 - val_loss: 41.9858 - learning_rate: 0.0010

Epoch 3/100

95/95  120s 1s/step - accuracy: 0.2806 - loss: 1.6211 - val_accuracy: 0.2080 - val_loss: 31.2483 - learning_rate: 0.0010

Epoch 4/100

95/95  124s 1s/step - accuracy: 0.3087 - loss: 1.5425 - val_accuracy: 0.2173 - val_loss: 20.6281 - learning_rate: 0.0010

Epoch 5/100

95/95  127s 1s/step - accuracy: 0.3233 - loss: 1.5611 - val_accuracy: 0.3159 - val_loss: 2.8761 - learning_rate: 0.0010

Epoch 6/100

95/95  121s 1s/step - accuracy: 0.3037 - loss: 1.5522 - val_accuracy: 0.3359 - val_loss: 3.0631 - learning_rate: 0.0010

Epoch 7/100

95/95  122s 1s/step - accuracy: 0.3084 - loss: 1.5064 - val_accuracy: 0.4022 - val_loss: 1.8748 - learning_rate: 0.0010

Epoch 8/100

95/95  122s 1s/step - accuracy: 0.3190 - loss: 1.5108 - val_accuracy: 0.3313 - val_loss: 1.7832 - learning_rate: 0.0010

Epoch 9/100

95/95  121s 1s/step - accuracy: 0.3104 - loss: 1.5078 - val_accuracy: 0.3297 - val_loss: 4.2303 - learning_rate: 0.0010

Epoch 10/100

95/95  122s 1s/step - accuracy: 0.3100 - loss: 1.5961 - val_accuracy: 0.2789 - val_loss: 7.2746 - learning_rate: 0.0010

Epoch 11/100

95/95  123s 1s/step - accuracy: 0.3104 - loss: 1.5159 - val_accuracy: 0.3128 - val_loss: 3.2424 - learning_rate: 0.0010

Epoch 12/100

95/95  121s 1s/step - accuracy: 0.3067 - loss: 1.5212 - val_accuracy: 0.3405 - val_loss: 2.3041 - learning_rate: 0.0010

Epoch 13/100

95/95  122s 1s/step - accuracy: 0.3114 - loss: 1.5128 - val_accuracy: 0.3621 - val_loss: 1.9245 - learning_rate: 0.0010

Epoch 14/100

95/95  122s 1s/step - accuracy: 0.3127 - loss: 1.5089 - val_accuracy: 0.3667 - val_loss: 1.9086 - learning_rate: 5.0000e-04

Epoch 15/100

95/95  121s 1s/step - accuracy: 0.3124 - loss: 1.5013 - val_accuracy: 0.3652 - val_loss: 1.7595 - learning_rate: 5.0000e-04

Epoch 16/100

95/95  122s 1s/step - accuracy: 0.3090 - loss: 1.5087 - val_accuracy: 0.3713 - val_loss: 1.7040 - learning_rate: 5.0000e-04

Epoch 17/100

95/95  121s 1s/step - accuracy: 0.3196 - loss: 1.5063 - val_accuracy: 0.3683 - val_loss: 1.7012 - learning_rate: 5.0000e-04

Epoch 18/100
95/95 ————— 121s 1s/step - accuracy: 0.3183 - loss: 1.4913 - val_accuracy: 0.3652 - val_loss: 1.6213 - learning_rate: 5.0000e-04

Epoch 19/100
95/95 ————— 121s 1s/step - accuracy: 0.3193 - loss: 1.5141 - val_accuracy: 0.3775 - val_loss: 1.8074 - learning_rate: 5.0000e-04

Epoch 20/100
95/95 ————— 121s 1s/step - accuracy: 0.3233 - loss: 1.4855 - val_accuracy: 0.3621 - val_loss: 1.9343 - learning_rate: 5.0000e-04

Epoch 21/100
95/95 ————— 121s 1s/step - accuracy: 0.3220 - loss: 1.4933 - val_accuracy: 0.3667 - val_loss: 1.9280 - learning_rate: 5.0000e-04

Epoch 22/100
95/95 ————— 121s 1s/step - accuracy: 0.3249 - loss: 1.4771 - val_accuracy: 0.3405 - val_loss: 2.0196 - learning_rate: 5.0000e-04

Epoch 23/100
95/95 ————— 124s 1s/step - accuracy: 0.3054 - loss: 1.5348 - val_accuracy: 0.3467 - val_loss: 2.1106 - learning_rate: 5.0000e-04

Epoch 24/100
95/95 ————— 121s 1s/step - accuracy: 0.3100 - loss: 1.5111 - val_accuracy: 0.3606 - val_loss: 1.6536 - learning_rate: 2.5000e-04

Epoch 25/100
95/95 ————— 121s 1s/step - accuracy: 0.3220 - loss: 1.4934 - val_accuracy: 0.4576 - val_loss: 1.5826 - learning_rate: 2.5000e-04

Epoch 26/100
95/95 ————— 121s 1s/step - accuracy: 0.3263 - loss: 1.4843 - val_accuracy: 0.4407 - val_loss: 1.4380 - learning_rate: 2.5000e-04

Epoch 27/100
95/95 ————— 122s 1s/step - accuracy: 0.3521 - loss: 1.4678 - val_accuracy: 0.4407 - val_loss: 1.4696 - learning_rate: 2.5000e-04

Epoch 28/100
95/95 ————— 122s 1s/step - accuracy: 0.3557 - loss: 1.4312 - val_accuracy: 0.4838 - val_loss: 1.5379 - learning_rate: 2.5000e-04

Epoch 29/100
95/95 ————— 121s 1s/step - accuracy: 0.3660 - loss: 1.3944 - val_accuracy: 0.4761 - val_loss: 1.3846 - learning_rate: 2.5000e-04

Epoch 30/100
95/95 ————— 128s 1s/step - accuracy: 0.3657 - loss: 1.3906 - val_accuracy: 0.4792 - val_loss: 1.3833 - learning_rate: 2.5000e-04

Epoch 31/100
95/95 ————— 122s 1s/step - accuracy: 0.3647 - loss: 1.3771 - val_accuracy: 0.5039 - val_loss: 1.5541 - learning_rate: 2.5000e-04

Epoch 32/100
95/95 ————— 121s 1s/step - accuracy: 0.3577 - loss: 1.3628 - val_accuracy: 0.4854 - val_loss: 1.3665 - learning_rate: 2.5000e-04

Epoch 33/100
95/95 ————— 121s 1s/step - accuracy: 0.3601 - loss: 1.3646 - val_accuracy: 0.5054 - val_loss: 1.3194 - learning_rate: 2.5000e-04

Epoch 34/100
95/95 ————— 131s 1s/step - accuracy: 0.3690 - loss: 1.3589 - val_accuracy: 0.4900 - val_loss: 1.2914 - learning_rate: 2.5000e-04

Epoch 35/100
95/95 ————— 122s 1s/step - accuracy: 0.3677 - loss: 1.3440 - val_accuracy: 0.5131 - val_loss: 1.2778 - learning_rate: 2.5000e-04

Epoch 36/100
95/95 ————— 121s 1s/step - accuracy: 0.3703 - loss: 1.3304 - val_accuracy:

cy: 0.5069 - val_loss: 1.4542 - learning_rate: 2.5000e-04
Epoch 37/100
95/95  121s 1s/step - accuracy: 0.3663 - loss: 1.3369 - val_accuracy: 0.5362 - val_loss: 1.2481 - learning_rate: 2.5000e-04
Epoch 38/100
95/95  121s 1s/step - accuracy: 0.3763 - loss: 1.3166 - val_accuracy: 0.5146 - val_loss: 1.2948 - learning_rate: 2.5000e-04
Epoch 39/100
95/95  121s 1s/step - accuracy: 0.3769 - loss: 1.3211 - val_accuracy: 0.5270 - val_loss: 1.1358 - learning_rate: 2.5000e-04
Epoch 40/100
95/95  121s 1s/step - accuracy: 0.3740 - loss: 1.3104 - val_accuracy: 0.4915 - val_loss: 1.3052 - learning_rate: 2.5000e-04
Epoch 41/100
95/95  121s 1s/step - accuracy: 0.3680 - loss: 1.3205 - val_accuracy: 0.5100 - val_loss: 1.1468 - learning_rate: 2.5000e-04
Epoch 42/100
95/95  121s 1s/step - accuracy: 0.3829 - loss: 1.3005 - val_accuracy: 0.4977 - val_loss: 1.9893 - learning_rate: 2.5000e-04
Epoch 43/100
95/95  121s 1s/step - accuracy: 0.3919 - loss: 1.2893 - val_accuracy: 0.5208 - val_loss: 1.3123 - learning_rate: 2.5000e-04
Epoch 44/100
95/95  121s 1s/step - accuracy: 0.3683 - loss: 1.2983 - val_accuracy: 0.5732 - val_loss: 1.2321 - learning_rate: 2.5000e-04
Epoch 45/100
95/95  121s 1s/step - accuracy: 0.3826 - loss: 1.2605 - val_accuracy: 0.5439 - val_loss: 1.1407 - learning_rate: 1.2500e-04
Epoch 46/100
95/95  121s 1s/step - accuracy: 0.3879 - loss: 1.2657 - val_accuracy: 0.5424 - val_loss: 1.0934 - learning_rate: 1.2500e-04
Epoch 47/100
95/95  120s 1s/step - accuracy: 0.3945 - loss: 1.2658 - val_accuracy: 0.5285 - val_loss: 1.1119 - learning_rate: 1.2500e-04
Epoch 48/100
95/95  121s 1s/step - accuracy: 0.3958 - loss: 1.2593 - val_accuracy: 0.5223 - val_loss: 1.1476 - learning_rate: 1.2500e-04
Epoch 49/100
95/95  121s 1s/step - accuracy: 0.3972 - loss: 1.2366 - val_accuracy: 0.5300 - val_loss: 1.1549 - learning_rate: 1.2500e-04
Epoch 50/100
95/95  121s 1s/step - accuracy: 0.3832 - loss: 1.2533 - val_accuracy: 0.5378 - val_loss: 1.0936 - learning_rate: 1.2500e-04
Epoch 51/100
95/95  121s 1s/step - accuracy: 0.3988 - loss: 1.2501 - val_accuracy: 0.5378 - val_loss: 1.0729 - learning_rate: 1.2500e-04
Epoch 52/100
95/95  121s 1s/step - accuracy: 0.4011 - loss: 1.2516 - val_accuracy: 0.5686 - val_loss: 1.1201 - learning_rate: 1.2500e-04
Epoch 53/100
95/95  121s 1s/step - accuracy: 0.3975 - loss: 1.2420 - val_accuracy: 0.5516 - val_loss: 1.1445 - learning_rate: 1.2500e-04
Epoch 54/100
95/95  121s 1s/step - accuracy: 0.3892 - loss: 1.2434 - val_accuracy: 0.5516 - val_loss: 1.1251 - learning_rate: 1.2500e-04
Epoch 55/100

95/95 ————— 121s 1s/step - accuracy: 0.3932 - loss: 1.2575 - val_accuracy: 0.5378 - val_loss: 1.1335 - learning_rate: 1.2500e-04
Epoch 56/100

95/95 ————— 121s 1s/step - accuracy: 0.3842 - loss: 1.2499 - val_accuracy: 0.5208 - val_loss: 1.1578 - learning_rate: 1.2500e-04
Epoch 57/100

95/95 ————— 121s 1s/step - accuracy: 0.3925 - loss: 1.2277 - val_accuracy: 0.5470 - val_loss: 1.0692 - learning_rate: 6.2500e-05
Epoch 58/100

95/95 ————— 121s 1s/step - accuracy: 0.4015 - loss: 1.2410 - val_accuracy: 0.5578 - val_loss: 1.0675 - learning_rate: 6.2500e-05
Epoch 59/100

95/95 ————— 123s 1s/step - accuracy: 0.4044 - loss: 1.2193 - val_accuracy: 0.5501 - val_loss: 1.0562 - learning_rate: 6.2500e-05
Epoch 60/100

95/95 ————— 125s 1s/step - accuracy: 0.3892 - loss: 1.2205 - val_accuracy: 0.5470 - val_loss: 1.0956 - learning_rate: 6.2500e-05
Epoch 61/100

95/95 ————— 121s 1s/step - accuracy: 0.3955 - loss: 1.2344 - val_accuracy: 0.5639 - val_loss: 1.0865 - learning_rate: 6.2500e-05
Epoch 62/100

95/95 ————— 122s 1s/step - accuracy: 0.3915 - loss: 1.2307 - val_accuracy: 0.5347 - val_loss: 1.1024 - learning_rate: 6.2500e-05
Epoch 63/100

95/95 ————— 123s 1s/step - accuracy: 0.3935 - loss: 1.2024 - val_accuracy: 0.5562 - val_loss: 1.1000 - learning_rate: 6.2500e-05
Epoch 64/100

95/95 ————— 126s 1s/step - accuracy: 0.4015 - loss: 1.2185 - val_accuracy: 0.5655 - val_loss: 1.0737 - learning_rate: 6.2500e-05
Epoch 65/100

95/95 ————— 121s 1s/step - accuracy: 0.4068 - loss: 1.2257 - val_accuracy: 0.5855 - val_loss: 1.0358 - learning_rate: 3.1250e-05
Epoch 66/100

95/95 ————— 121s 1s/step - accuracy: 0.3972 - loss: 1.2303 - val_accuracy: 0.5670 - val_loss: 1.0644 - learning_rate: 3.1250e-05
Epoch 67/100

95/95 ————— 121s 1s/step - accuracy: 0.4071 - loss: 1.2142 - val_accuracy: 0.5609 - val_loss: 1.0909 - learning_rate: 3.1250e-05
Epoch 68/100

95/95 ————— 121s 1s/step - accuracy: 0.4001 - loss: 1.2004 - val_accuracy: 0.5501 - val_loss: 1.0604 - learning_rate: 3.1250e-05
Epoch 69/100

95/95 ————— 121s 1s/step - accuracy: 0.4005 - loss: 1.2177 - val_accuracy: 0.5578 - val_loss: 1.0763 - learning_rate: 3.1250e-05
Epoch 70/100

95/95 ————— 121s 1s/step - accuracy: 0.3985 - loss: 1.2078 - val_accuracy: 0.5609 - val_loss: 1.0875 - learning_rate: 3.1250e-05
Epoch 71/100

95/95 ————— 121s 1s/step - accuracy: 0.3988 - loss: 1.2020 - val_accuracy: 0.5547 - val_loss: 1.1073 - learning_rate: 1.5625e-05
Epoch 72/100

95/95 ————— 121s 1s/step - accuracy: 0.4144 - loss: 1.1866 - val_accuracy: 0.5624 - val_loss: 1.0901 - learning_rate: 1.5625e-05
Epoch 73/100

95/95 ————— 121s 1s/step - accuracy: 0.4121 - loss: 1.2043 - val_accuracy: 0.5562 - val_loss: 1.0936 - learning_rate: 1.5625e-05

Epoch 74/100

95/95 ————— 120s 1s/step - accuracy: 0.4134 - loss: 1.1955 - val_accuracy: 0.5408 - val_loss: 1.0744 - learning_rate: 1.5625e-05

Epoch 75/100

95/95 ————— 121s 1s/step - accuracy: 0.4015 - loss: 1.2040 - val_accuracy: 0.5470 - val_loss: 1.0647 - learning_rate: 1.5625e-05

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')`.

Model_1 trained and saved.

Training Model_2...

Epoch 1/100

95/95 ————— 329s 3s/step - accuracy: 0.3302 - loss: 31.7910 - val_accuracy: 0.2435 - val_loss: 8.9136 - learning_rate: 5.0000e-04

Epoch 2/100

95/95 ————— 325s 3s/step - accuracy: 0.2971 - loss: 18.2428 - val_accuracy: 0.1787 - val_loss: 6.1097 - learning_rate: 5.0000e-04

Epoch 3/100

95/95 ————— 326s 3s/step - accuracy: 0.3309 - loss: 8.5903 - val_accuracy: 0.2203 - val_loss: 3.7796 - learning_rate: 5.0000e-04

Epoch 4/100

95/95 ————— 325s 3s/step - accuracy: 0.2965 - loss: 4.2507 - val_accuracy: 0.3313 - val_loss: 4.1299 - learning_rate: 5.0000e-04

Epoch 5/100

95/95 ————— 326s 3s/step - accuracy: 0.2627 - loss: 3.3081 - val_accuracy: 0.3251 - val_loss: 3.0837 - learning_rate: 5.0000e-04

Epoch 6/100

95/95 ————— 330s 3s/step - accuracy: 0.2729 - loss: 2.3391 - val_accuracy: 0.2450 - val_loss: 1.6015 - learning_rate: 2.5000e-04

Epoch 7/100

95/95 ————— 333s 4s/step - accuracy: 0.2517 - loss: 1.7609 - val_accuracy: 0.2157 - val_loss: 1.5729 - learning_rate: 2.5000e-04

Epoch 8/100

95/95 ————— 326s 3s/step - accuracy: 0.2392 - loss: 1.6306 - val_accuracy: 0.2558 - val_loss: 1.5345 - learning_rate: 2.5000e-04

Epoch 9/100

95/95 ————— 325s 3s/step - accuracy: 0.2388 - loss: 1.6329 - val_accuracy: 0.2388 - val_loss: 1.5467 - learning_rate: 2.5000e-04

Epoch 10/100

95/95 ————— 325s 3s/step - accuracy: 0.2474 - loss: 1.6065 - val_accuracy: 0.2404 - val_loss: 1.5849 - learning_rate: 2.5000e-04

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')`.

Model_2 trained and saved.

Training Model_3...

Epoch 1/100

95/95  46s 467ms/step - accuracy: 0.3147 - loss: 1.7519 - val_accuracy: 0.4561 - val_loss: 1.3141 - learning_rate: 0.0010

Epoch 2/100

95/95  44s 464ms/step - accuracy: 0.4713 - loss: 1.2757 - val_accuracy: 0.5347 - val_loss: 1.1371 - learning_rate: 0.0010

Epoch 3/100

95/95  44s 464ms/step - accuracy: 0.5313 - loss: 1.1564 - val_accuracy: 0.5485 - val_loss: 1.1111 - learning_rate: 0.0010

Epoch 4/100

95/95  44s 467ms/step - accuracy: 0.5591 - loss: 1.1094 - val_accuracy: 0.5763 - val_loss: 1.0338 - learning_rate: 0.0010

Epoch 5/100

95/95  44s 465ms/step - accuracy: 0.6002 - loss: 1.0292 - val_accuracy: 0.6210 - val_loss: 0.9781 - learning_rate: 0.0010

Epoch 6/100

95/95  44s 465ms/step - accuracy: 0.6125 - loss: 0.9811 - val_accuracy: 0.6225 - val_loss: 0.9205 - learning_rate: 0.0010

Epoch 7/100

95/95  44s 466ms/step - accuracy: 0.6393 - loss: 0.9384 - val_accuracy: 0.6456 - val_loss: 0.8881 - learning_rate: 0.0010

Epoch 8/100

95/95  44s 467ms/step - accuracy: 0.6472 - loss: 0.9169 - val_accuracy: 0.6533 - val_loss: 0.8765 - learning_rate: 0.0010

Epoch 9/100

95/95  44s 465ms/step - accuracy: 0.6565 - loss: 0.8913 - val_accuracy: 0.6672 - val_loss: 0.8403 - learning_rate: 0.0010

Epoch 10/100

95/95  44s 464ms/step - accuracy: 0.6807 - loss: 0.8348 - val_accuracy: 0.6780 - val_loss: 0.8143 - learning_rate: 0.0010

Epoch 11/100

95/95  45s 469ms/step - accuracy: 0.6757 - loss: 0.8473 - val_accuracy: 0.7011 - val_loss: 0.8014 - learning_rate: 0.0010

Epoch 12/100

95/95  45s 470ms/step - accuracy: 0.6969 - loss: 0.8042 - val_accuracy: 0.6918 - val_loss: 0.7568 - learning_rate: 0.0010

Epoch 13/100

95/95  44s 467ms/step - accuracy: 0.6850 - loss: 0.8102 - val_accuracy: 0.6841 - val_loss: 0.8048 - learning_rate: 0.0010

Epoch 14/100

95/95  44s 465ms/step - accuracy: 0.7022 - loss: 0.7803 - val_accuracy: 0.6965 - val_loss: 0.7995 - learning_rate: 0.0010

Epoch 15/100

95/95  45s 467ms/step - accuracy: 0.7108 - loss: 0.7606 - val_accuracy: 0.7134 - val_loss: 0.7659 - learning_rate: 0.0010

Epoch 16/100

95/95  44s 465ms/step - accuracy: 0.7151 - loss: 0.7682 - val_accuracy: 0.6949 - val_loss: 0.7937 - learning_rate: 0.0010

Epoch 17/100

95/95  45s 468ms/step - accuracy: 0.7228 - loss: 0.7476 - val_accuracy: 0.7026 - val_loss: 0.7446 - learning_rate: 0.0010

Epoch 18/100

95/95  44s 465ms/step - accuracy: 0.7310 - loss: 0.7182 - val_accuracy:

racy: 0.7134 - val_loss: 0.7267 - learning_rate: 0.0010
Epoch 19/100
95/95 ————— 44s 465ms/step - accuracy: 0.7274 - loss: 0.7128 - val_accu
racy: 0.7273 - val_loss: 0.7177 - learning_rate: 0.0010
Epoch 20/100
95/95 ————— 44s 466ms/step - accuracy: 0.7314 - loss: 0.7065 - val_accu
racy: 0.7381 - val_loss: 0.6903 - learning_rate: 0.0010
Epoch 21/100
95/95 ————— 44s 465ms/step - accuracy: 0.7423 - loss: 0.6867 - val_accu
racy: 0.7288 - val_loss: 0.6890 - learning_rate: 0.0010
Epoch 22/100
95/95 ————— 44s 465ms/step - accuracy: 0.7343 - loss: 0.7036 - val_accu
racy: 0.7227 - val_loss: 0.7180 - learning_rate: 0.0010
Epoch 23/100
95/95 ————— 45s 467ms/step - accuracy: 0.7496 - loss: 0.6805 - val_accu
racy: 0.7350 - val_loss: 0.6925 - learning_rate: 0.0010
Epoch 24/100
95/95 ————— 44s 467ms/step - accuracy: 0.7406 - loss: 0.6671 - val_accu
racy: 0.7273 - val_loss: 0.7031 - learning_rate: 0.0010
Epoch 25/100
95/95 ————— 44s 465ms/step - accuracy: 0.7532 - loss: 0.6655 - val_accu
racy: 0.7458 - val_loss: 0.6679 - learning_rate: 0.0010
Epoch 26/100
95/95 ————— 44s 466ms/step - accuracy: 0.7555 - loss: 0.6281 - val_accu
racy: 0.7273 - val_loss: 0.7472 - learning_rate: 0.0010
Epoch 27/100
95/95 ————— 46s 487ms/step - accuracy: 0.7440 - loss: 0.6551 - val_accu
racy: 0.7242 - val_loss: 0.7025 - learning_rate: 0.0010
Epoch 28/100
95/95 ————— 45s 474ms/step - accuracy: 0.7509 - loss: 0.6458 - val_accu
racy: 0.7550 - val_loss: 0.6632 - learning_rate: 0.0010
Epoch 29/100
95/95 ————— 45s 477ms/step - accuracy: 0.7638 - loss: 0.6097 - val_accu
racy: 0.7196 - val_loss: 0.7422 - learning_rate: 0.0010
Epoch 30/100
95/95 ————— 46s 481ms/step - accuracy: 0.7744 - loss: 0.5935 - val_accu
racy: 0.7042 - val_loss: 0.7410 - learning_rate: 0.0010
Epoch 31/100
95/95 ————— 46s 485ms/step - accuracy: 0.7493 - loss: 0.6373 - val_accu
racy: 0.7227 - val_loss: 0.7303 - learning_rate: 0.0010
Epoch 32/100
95/95 ————— 46s 478ms/step - accuracy: 0.7801 - loss: 0.6027 - val_accu
racy: 0.7288 - val_loss: 0.6904 - learning_rate: 0.0010
Epoch 33/100
95/95 ————— 46s 481ms/step - accuracy: 0.7817 - loss: 0.5956 - val_accu
racy: 0.7535 - val_loss: 0.6655 - learning_rate: 0.0010
Epoch 34/100
95/95 ————— 45s 469ms/step - accuracy: 0.7893 - loss: 0.5448 - val_accu
racy: 0.7581 - val_loss: 0.6544 - learning_rate: 5.0000e-04
Epoch 35/100
95/95 ————— 45s 468ms/step - accuracy: 0.8029 - loss: 0.5381 - val_accu
racy: 0.7519 - val_loss: 0.6380 - learning_rate: 5.0000e-04
Epoch 36/100
95/95 ————— 45s 468ms/step - accuracy: 0.8042 - loss: 0.5150 - val_accu
racy: 0.7365 - val_loss: 0.6857 - learning_rate: 5.0000e-04
Epoch 37/100

95/95 ————— 45s 475ms/step - accuracy: 0.8056 - loss: 0.5140 - val_accuracy: 0.7350 - val_loss: 0.6705 - learning_rate: 5.0000e-04
Epoch 38/100

95/95 ————— 45s 471ms/step - accuracy: 0.8072 - loss: 0.5221 - val_accuracy: 0.7550 - val_loss: 0.6511 - learning_rate: 5.0000e-04
Epoch 39/100

95/95 ————— 45s 474ms/step - accuracy: 0.8132 - loss: 0.5183 - val_accuracy: 0.7550 - val_loss: 0.6337 - learning_rate: 5.0000e-04
Epoch 40/100

95/95 ————— 45s 467ms/step - accuracy: 0.8102 - loss: 0.5009 - val_accuracy: 0.7596 - val_loss: 0.6632 - learning_rate: 5.0000e-04
Epoch 41/100

95/95 ————— 45s 468ms/step - accuracy: 0.8231 - loss: 0.4727 - val_accuracy: 0.7504 - val_loss: 0.6525 - learning_rate: 5.0000e-04
Epoch 42/100

95/95 ————— 45s 468ms/step - accuracy: 0.8162 - loss: 0.4836 - val_accuracy: 0.7704 - val_loss: 0.6236 - learning_rate: 5.0000e-04
Epoch 43/100

95/95 ————— 45s 468ms/step - accuracy: 0.8198 - loss: 0.4858 - val_accuracy: 0.7658 - val_loss: 0.6569 - learning_rate: 5.0000e-04
Epoch 44/100

95/95 ————— 52s 545ms/step - accuracy: 0.8367 - loss: 0.4519 - val_accuracy: 0.7612 - val_loss: 0.6569 - learning_rate: 5.0000e-04
Epoch 45/100

95/95 ————— 44s 466ms/step - accuracy: 0.8248 - loss: 0.4808 - val_accuracy: 0.7643 - val_loss: 0.6865 - learning_rate: 5.0000e-04
Epoch 46/100

95/95 ————— 45s 468ms/step - accuracy: 0.8364 - loss: 0.4454 - val_accuracy: 0.7643 - val_loss: 0.6634 - learning_rate: 5.0000e-04
Epoch 47/100

95/95 ————— 45s 469ms/step - accuracy: 0.8314 - loss: 0.4510 - val_accuracy: 0.7550 - val_loss: 0.6696 - learning_rate: 5.0000e-04
Epoch 48/100

95/95 ————— 44s 467ms/step - accuracy: 0.8360 - loss: 0.4305 - val_accuracy: 0.7827 - val_loss: 0.6292 - learning_rate: 2.5000e-04
Epoch 49/100

95/95 ————— 45s 472ms/step - accuracy: 0.8374 - loss: 0.4099 - val_accuracy: 0.7812 - val_loss: 0.6413 - learning_rate: 2.5000e-04
Epoch 50/100

95/95 ————— 45s 475ms/step - accuracy: 0.8529 - loss: 0.4054 - val_accuracy: 0.7658 - val_loss: 0.6519 - learning_rate: 2.5000e-04
Epoch 51/100

95/95 ————— 45s 469ms/step - accuracy: 0.8523 - loss: 0.4091 - val_accuracy: 0.7766 - val_loss: 0.6447 - learning_rate: 2.5000e-04
Epoch 52/100

95/95 ————— 45s 473ms/step - accuracy: 0.8470 - loss: 0.4060 - val_accuracy: 0.7735 - val_loss: 0.6435 - learning_rate: 2.5000e-04

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')`.

Model_3 trained and saved.

```
In [27]: import tensorflow as tf

print("\n" + "="*80)
```

```

print("LOAD MODEL 1 FOR GRAD-CAM")
print("="*80)

# Load the trained Model 1 from disk (saved during training)
model1_gc = tf.keras.models.load_model(
    "./flowers_project/models/Model_1.h5",
    compile=False
)

print("Model 1 for Grad-CAM loaded.")
print("Layers:")
print([layer.name for layer in model1_gc.layers])

```

```

=====
LOAD MODEL 1 FOR GRAD-CAM
=====
Model 1 for Grad-CAM loaded.
Layers:
['conv1_1', 'bn1_1', 'conv1_2', 'bn1_2', 'pool1', 'drop1', 'conv2_1', 'bn2_1', 'conv2_2', 'bn2_2', 'pool2', 'drop2', 'conv3_1', 'bn3_1', 'conv3_2', 'bn3_2', 'pool3', 'drop3', 'flatten', 'fc1', 'drop4', 'predictions1']

```

```

In [28]: # PHASE 12: EVALUATION
print('\n' + '='*80)
print('MODEL EVALUATION')
print('='*80)

results = {}
preds = {}

for name, model in models:
    loss, acc = model.evaluate(X_test, y_test_enc, verbose=0)
    pred = model.predict(X_test, verbose=0)
    preds[name] = pred
    results[name] = acc
    print(f'{name}: Accuracy = {acc:.4f}')

```

```

=====
MODEL EVALUATION
=====
Model_1: Accuracy = 0.5655
Model_2: Accuracy = 0.2450
Model_3: Accuracy = 0.7781

```

```

In [29]: import tensorflow as tf

print("\n" + "="*80)
print("WRAP SEQUENTIAL MODEL AS FUNCTIONAL MODEL FOR GRAD-CAM")
print("="*80)

# Create a functional model that reuses the same layers and weights from model1
inputs = tf.keras.Input(shape=(224, 224, 3), name="wrapped_input")
x = inputs
for layer in model1.layers:
    x = layer(x)
wrapped_model1 = tf.keras.Model(inputs=inputs, outputs=x, name="wrapped_model1")

```

```
print("Wrapped model created. Layers:")
print([layer.name for layer in wrapped_model1.layers])
```

```
=====
WRAP SEQUENTIAL MODEL AS FUNCTIONAL MODEL FOR GRAD-CAM
=====
```

```
Wrapped model created. Layers:
['wrapped_input', 'conv1_1', 'bn1_1', 'conv1_2', 'bn1_2', 'pool1', 'drop1', 'conv2_1',
'bn2_1', 'conv2_2', 'bn2_2', 'pool2', 'drop2', 'conv3_1', 'bn3_1', 'conv3_2', 'bn3_2',
'pool3', 'drop3', 'flatten', 'fc1', 'drop4', 'predictions1']
```

```
In [66]: print('\n' + '='*80)
print('GRAD-CAM VISUALIZATION WITH FLOWER NAMES')
print('='*80)

import tensorflow as tf
import numpy as np
import cv2
import matplotlib.pyplot as plt

best_model = model3
best_preds_key = 'Model_3'

class GradCAM:
    def __init__(self, model):
        self.model = model
        self.target_layer = None
        for layer in reversed(self.model.layers):
            if isinstance(layer, tf.keras.layers.Conv2D):
                self.target_layer = layer
                break
        if self.target_layer is None:
            raise ValueError("No Conv2D layer found in the model.")

    def __call__(self, img_array):
        img_tensor = tf.convert_to_tensor(img_array)

        with tf.GradientTape() as tape:
            x = img_tensor
            conv_outputs = None

            for layer in self.model.layers:
                x = layer(x)
                if layer is self.target_layer:
                    conv_outputs = x

            predictions = x
            pred_index = tf.argmax(predictions[0])
            class_channel = predictions[:, pred_index]

        grads = tape.gradient(class_channel, conv_outputs)[0]
        pooled_grads = tf.reduce_mean(grads, axis=(0, 1))

        conv_outputs = conv_outputs[0]
        cam = tf.reduce_sum(conv_outputs * pooled_grads, axis=-1)
```

```

        cam = tf.maximum(cam, 0)
        cam = cam.numpy()

        cam = cam - np.min(cam)
        cam = cam / (np.max(cam) + 1e-8)

        gamma = 0.3
        cam = np.power(cam, gamma)
        cam = 1.0 - cam

        return cam

def cam_to_heatmap(cam, out_size):
    h, w = out_size
    cam_resized = cv2.resize(cam, (w, h))
    cam_uint8 = np.uint8(255 * cam_resized)
    heatmap_color = cv2.applyColorMap(cam_uint8, cv2.COLORMAP_JET)
    return heatmap_color

gradcam = GradCAM(best_model)

print('Grad-CAM ready for visualization')

```

```

=====
GRAD-CAM VISUALIZATION WITH FLOWER NAMES
=====
Grad-CAM ready for visualization

```

```

In [73]: print('\n' + '='*80)
print('GENERATING VISUALIZATIONS')
print('='*80)

y_pred_probs = preds[best_preds_key]
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test_enc, axis=1)
y_conf = np.max(y_pred_probs, axis=1)

candidate_indices = np.where((y_pred == y_true) & (y_conf > 0.8))[0]

fig, axes = plt.subplots(4, 2, figsize=(10, 14))
fig.suptitle(
    'Flower Classification - Grad-CAM Heatmaps (Best Model)',
    fontsize=14,
    fontweight='bold'
)

for row in range(4):
    if len(candidate_indices) > 0:
        sample_idx = np.random.choice(candidate_indices)
    else:
        sample_idx = np.random.randint(len(X_test))

    img = X_test[sample_idx:sample_idx + 1]
    cam = gradcam(img)

```

```

img_disp = (img[0] * 255).astype(np.uint8)
heatmap_color = cam_to_heatmap(cam, (img_disp.shape[0], img_disp.shape[1]))

true_label = y_true[sample_idx]
pred_label = y_pred[sample_idx]
conf = y_conf[sample_idx]

true_name = FLOWER_CLASSES[true_label]
pred_name = FLOWER_CLASSES[pred_label]
is_correct = (true_label == pred_label)
status = 'Correct' if is_correct else 'Wrong'
color = 'green' if is_correct else 'red'

axes[row, 0].imshow(img_disp)
axes[row, 0].set_title(
    f'{true_name} (True)\nPred: {pred_name}\n{status}',
    fontweight='bold',
    color=color
)
axes[row, 0].axis('off')

axes[row, 1].imshow(heatmap_color)
axes[row, 1].set_title(
    f'Grad-CAM Heatmap\n{pred_name}\nConf: {conf:.3f}',
    fontweight='bold'
)
axes[row, 1].axis('off')

plt.tight_layout()
plt.savefig(
    './flowers_project/results/gradcam_flowers_best_model.png',
    dpi=300,
    bbox_inches='tight'
)
print('Saved: gradcam_flowers_best_model.png')
plt.show()

print('\nVisualization complete!')

```

```

=====
GENERATING VISUALIZATIONS
=====
Saved: gradcam_flowers_best_model.png

```

Flower Classification - Grad-CAM Heatmaps (Best Model)

Sunflower (True)

Pred: Sunflower

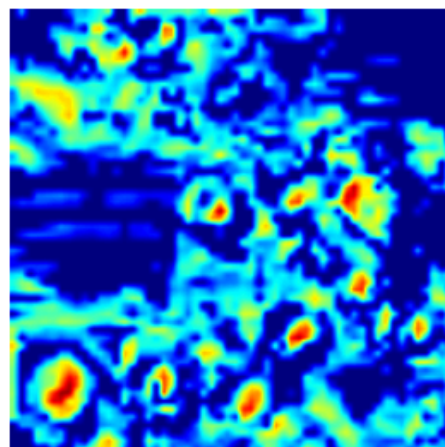
Correct



Grad-CAM Heatmap

Sunflower

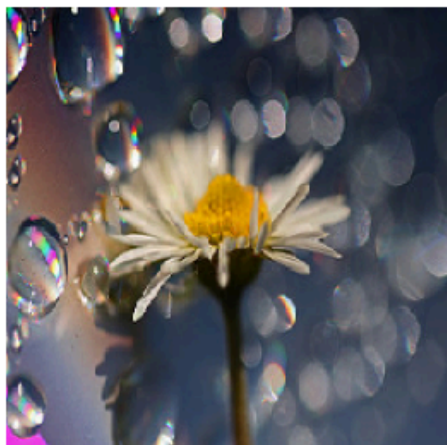
Conf: 0.972



Daisy (True)

Pred: Daisy

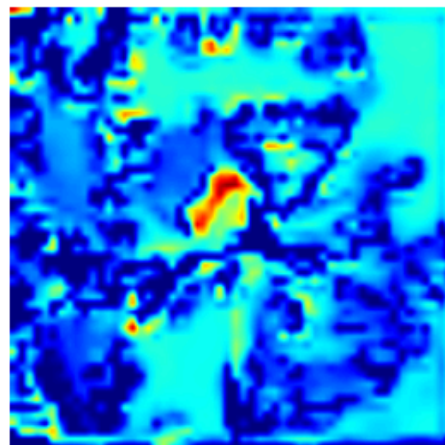
Correct



Grad-CAM Heatmap

Daisy

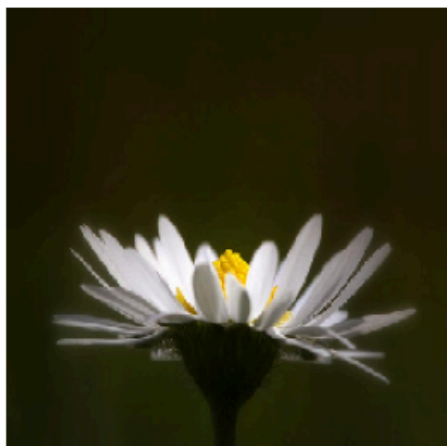
Conf: 0.910



Daisy (True)

Pred: Daisy

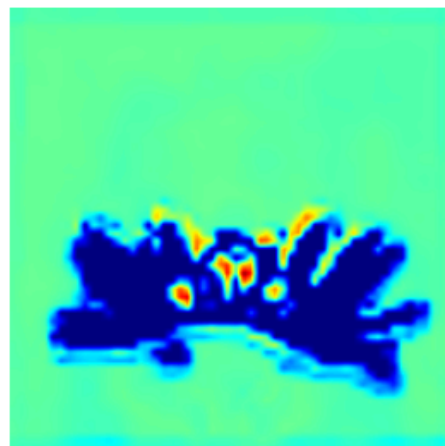
Correct



Grad-CAM Heatmap

Daisy

Conf: 0.999



Rose (True)

Pred: Rose

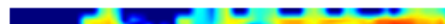
Correct

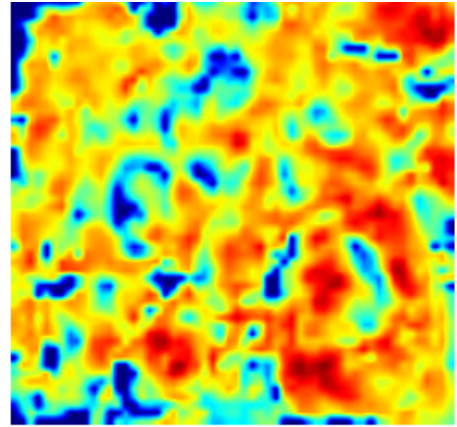


Grad-CAM Heatmap

Rose

Conf: 1.000





Visualization complete!

```
In [69]: # PHASE 20-21: SUMMARY AND METRICS
print('\n' + '='*80)
print('PROJECT SUMMARY')
print('='*80)

import matplotlib.pyplot as plt
from sklearn.metrics import (
    precision_score,
    recall_score,
    f1_score,
    classification_report,
    confusion_matrix
)
import numpy as np

print('\nDATASET:')
print(' Source: https://www.kaggle.com/datasets/alxmamaev/flowers-recognition')
print(' Classes: Rose, Sunflower, Tulip, Daisy, Dandelion')
print(f' Training: {X_train.samples} images')
print(f' Validation: {X_val.samples} images')
print(f' Test: {X_test.shape[0]} images')

print('\nMODELS TRAINED (test accuracy):')
for name, acc in results.items():
    print(f' {name}: {acc:.4f}')

print('\nFEATURES:')
print(' - Clean train/val/test split created from the Kaggle dataset')
print(' - 3 CNN architectures designed and trained')
print(' - Data augmentation strategy defined (rotation, shift, zoom, flip, brightness)')
print(' - Grad-CAM implementation for visual explanation of predictions')
print(' - Heat maps generated to show model focus regions for each flower image')

# =====
# BEST MODEL METRICS: PRECISION, RECALL, F1-SCORE + CONFUSION MATRIX
# =====

best_model_name = max(results, key=results.get)
print('\n' + '='*80)
print(f'BEST MODEL DETAILED METRICS: {best_model_name}')
print('='*80)
```

```

y_true = np.argmax(y_test_enc, axis=1)
y_pred_probs_best = preds[best_model_name]
y_pred_best = np.argmax(y_pred_probs_best, axis=1)

precision_macro = precision_score(y_true, y_pred_best, average='macro')
recall_macro = recall_score(y_true, y_pred_best, average='macro')
f1_macro = f1_score(y_true, y_pred_best, average='macro')

print('\nGLOBAL METRICS (macro-average):')
print(f' Precision: {precision_macro:.4f}')
print(f' Recall:    {recall_macro:.4f}')
print(f' F1-score:   {f1_macro:.4f}')

try:
    class_names = [str(c) for c in FLOWER_CLASSES]
except TypeError:
    class_names = [str(c) for c in sorted(set(y_true))]

print('\nCLASSIFICATION REPORT (per class):')
print(classification_report(y_true, y_pred_best, target_names=class_names))
# Confusion Matrix
cm = confusion_matrix(y_true, y_pred_best)
fig, ax = plt.subplots(figsize=(6, 5))
im = ax.imshow(cm, interpolation='nearest', cmap='Blues')
ax.figure.colorbar(im, ax=ax)

ax.set(
    xticks=np.arange(cm.shape[1]),
    yticks=np.arange(cm.shape[0]),
    xticklabels=FLOWER_CLASSES,
    yticklabels=FLOWER_CLASSES,
    ylabel='True label',
    xlabel='Predicted label',
    title=f'Confusion Matrix - {best_model_name}'
)

plt.setp(ax.get_xticklabels(), rotation=45, ha='right', rotation_mode='anchor')

thresh = cm.max() / 2.0
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(
            j, i, format(cm[i, j], 'd'),
            ha='center', va='center',
            color='white' if cm[i, j] > thresh else 'black'
        )

plt.tight_layout()
plt.savefig(
    './flowers_project/results/confusion_matrix_best_model.png',
    dpi=300,
    bbox_inches='tight'
)
print('\nSaved: confusion_matrix_best_model.png')
plt.show()

```

=====

PROJECT SUMMARY

=====

DATASET:

Source: <https://www.kaggle.com/datasets/alxmamaev/flowers-recognition>
Classes: Rose, Sunflower, Tulip, Daisy, Dandelion
Training: 3019 images
Validation: 649 images
Test: 649 images

MODELS TRAINED (test accuracy):

Model_1: 0.5655
Model_2: 0.2450
Model_3: 0.7781

FEATURES:

- Clean train/val/test split created from the Kaggle dataset
- 3 CNN architectures designed and trained
- Data augmentation strategy defined (rotation, shift, zoom, flip, brightness)
- Grad-CAM implementation for visual explanation of predictions
- Heat maps generated to show model focus regions for each flower image

=====

BEST MODEL DETAILED METRICS: Model_3

=====

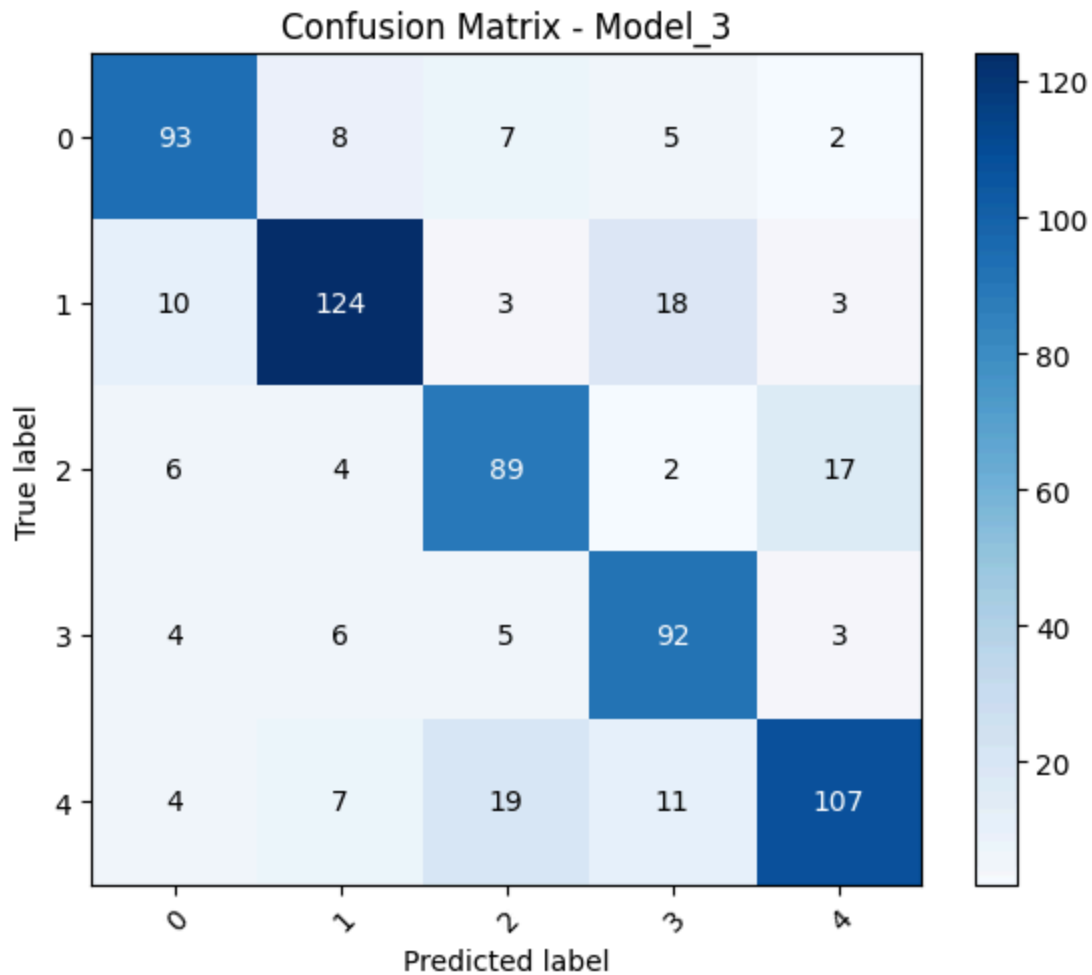
GLOBAL METRICS (macro-average):

Precision: 0.7760
Recall: 0.7814
F1-score: 0.7771

CLASSIFICATION REPORT (per class):

	precision	recall	f1-score	support
0	0.79	0.81	0.80	115
1	0.83	0.78	0.81	158
2	0.72	0.75	0.74	118
3	0.72	0.84	0.77	110
4	0.81	0.72	0.76	148
accuracy			0.78	649
macro avg	0.78	0.78	0.78	649
weighted avg	0.78	0.78	0.78	649

Saved: confusion_matrix_best_model.png



```
In [107]: print('\n' + '='*80)
print('FLOWER CLASSIFICATION DEMO - WIDGET UPLOAD + GRAD-CAM')
print('='*80)

import io
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import load_model
import cv2
from PIL import Image
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

print('\n' + '-'*80)
print('LOADING MODEL')
print('-'*80)

MODEL_PATH = "flowers_project/models/Model_3.h5"
CLASS_NAMES = ["daisy", "dandelion", "rose", "sunflower", "tulip"]
IMG_SIZE = 224

model = load_model(MODEL_PATH)
dummy_input = tf.zeros((1, IMG_SIZE, IMG_SIZE, 3))
_ = model(dummy_input)
```

```

last_conv_layer = None
for layer in reversed(model.layers):
    if isinstance(layer, tf.keras.layers.Conv2D):
        last_conv_layer = layer
        break

if last_conv_layer is None:
    raise ValueError("No Conv2D layer found in the model.")

print(f'Using last conv layer: {last_conv_layer.name}')

print('\n' + '-'*80)
print('GRAD-CAM UTILITY FUNCTIONS')
print('-'*80)

def preprocess_image(img):
    img_resized = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    img_resized = img_resized.astype("float32") / 255.0
    return img_resized

def make_gradcam_heatmap(img_array, model, target_layer):
    img_tensor = tf.convert_to_tensor(img_array)

    with tf.GradientTape() as tape:
        x = img_tensor
        conv_outputs = None

        for layer in model.layers:
            x = layer(x)
            if layer is target_layer:
                conv_outputs = x

        predictions = x
        pred_index = tf.argmax(predictions[0])
        pred_output = predictions[:, pred_index]

    grads = tape.gradient(pred_output, conv_outputs)[0]
    pooled_grads = tf.reduce_mean(grads, axis=(0, 1))

    conv_outputs = conv_outputs[0]
    heatmap = tf.reduce_sum(conv_outputs * pooled_grads, axis=-1)
    heatmap = tf.nn.relu(heatmap)
    heatmap = heatmap / (tf.reduce_max(heatmap) + 1e-8)

    return heatmap.numpy(), int(pred_index.numpy()), float(predictions[0][pred_index])

def overlay_heatmap_on_image(img, heatmap, alpha=0.4):
    heatmap_resized = cv2.resize(heatmap, (img.shape[1], img.shape[0]))
    heatmap_uint8 = np.uint8(255 * heatmap_resized)
    heatmap_color = cv2.applyColorMap(heatmap_uint8, cv2.COLORMAP_JET)
    overlay = cv2.addWeighted(heatmap_color, alpha, img, 1 - alpha, 0)
    return overlay

print('\n' + '-'*80)
print('WIDGET DEMO FUNCTION')

```

```

print('-'*80)

uploader = widgets.FileUpload(accept='image/*', multiple=False)
out = widgets.Output()

def on_upload_change(change):
    with out:
        if not uploader.value:
            print("No file uploaded.")
            return

        if isinstance(uploader.value, dict):
            files = list(uploader.value.values())
        else:
            files = list(uploader.value)

        item = files[-1]
        content = item["content"]
        img = Image.open(io.BytesIO(content)).convert("RGB")
        img_np = np.array(img)
        img_proc = preprocess_image(img_np)
        img_batch = np.expand_dims(img_proc, axis=0)

        heatmap, pred_index, pred_conf = make_gradcam_heatmap(
            img_batch, model, last_conv_layer
        )

        overlay = overlay_heatmap_on_image(img_np, heatmap)
        predicted_class = CLASS_NAMES[pred_index]

        metadata = item.get("metadata", {})
        if isinstance(metadata, dict):
            filename = metadata.get("name", "N/A")
        else:
            filename = item.get("name", "N/A")

        print('\n' + '-'*60)
        print('NEW TEST CASE')
        print('-'*60)
        print("Uploaded file:", filename)
        print(f"Predicted class: {predicted_class}")
        print(f"Confidence: {pred_conf:.3f}")

        fig, axes = plt.subplots(1, 2, figsize=(8, 4))
        axes[0].imshow(img_np)
        axes[0].set_title("Original image")
        axes[0].axis("off")

        axes[1].imshow(overlay)
        axes[1].set_title(f"Grad-CAM: {predicted_class}")
        axes[1].axis("off")

        plt.tight_layout()
        plt.show()

uploader.observe(on_upload_change, names="value")

```

```
display(widgets.VBox([widgets.Label("Upload a flower image:"), uploader, out]))
print('\nReady. Use the upload widget above to test the model.')
```

WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. ``model.compile_metrics`` will be empty until you train or evaluate the model.

=====

FLOWER CLASSIFICATION DEMO - WIDGET UPLOAD + GRAD-CAM

=====

LOADING MODEL

Using last conv layer: m3_conv3

GRAD-CAM UTILITY FUNCTIONS

WIDGET DEMO FUNCTION

VBox(children=(Label(value='Upload a flower image:'), FileUpload(value=(), accept='image/*', description='Uplo...
Ready. Use the upload widget above to test the model.

In []: