Alumno:

- **Jorge Javier Sosa Briseño**
  Matrícula: A01749489

Módulo:

- **Deep Learning**

Convolutional Neural Networks y Transfer Learning

En este código se entrena un modelo de red neuronal convolucional (CNN) utilizando Transfer Learning (TL) con la arquitectura VGG16.

El objetivo de este documento es mostrar el uso de la técnica de Transfer Learning mediante la implementación de modelos pre-entrenados para posteriormente entrenarlos con nuevos datos y ser usados para clasficiar nuevos objetivos. En este caso su objetivo es clasificar imágenes de flores en diferentes categorías.

El código utiliza TensorFlow y Keras para construir y entrenar el modelo. Además, se realiza una división del conjunto de datos en conjuntos de entrenamiento y validación y se aplica 'data augmentation' para mejorar el rendimiento del modelo. Se realiza un seguimiento del mejor modelo utilizando un punto de control, lo que garantiza que se guarda el modelo con el mejor rendimiento en la validación. Por último, se muestra una visualización de las métricas de entrenamiento y validación.

Importar dependencias

```
# @title **Importar dependencias**
import numpy as np
import matplotlib.pyplot as plt
import os
import PIL
import tensorflow as tf
from tensorflow import keras
from keras import layers
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.models import Sequential, load_model
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint
```

Se importan librerías como Numpy, Matplotlib, Tensorflow y Keras, Os, PIL, junto con varios módulos específicos de Keras para construir y entrenar el modelo.

Cargar base de datos

```
# @title **Cargar base de datos**

import pathlib
import random
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)
data_dir = pathlib.Path(data_dir)
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[random.randint(0,100)]))
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_imag
228813984/228813984 [==============================] - 1s 0us/step
```

Descargamos y descomprimimos el conjunto de datos de imágenes de flores desde una URL proporcionada. Utilizamos TensorFlow Datasets para cargar y explorar el conjunto de datos. Después se verifica que se haya cargado correctamente.

```
from keras.preprocessing.image import ImageDataGenerator

img_size = (256, 256)
batch_size = 128

# Data augmentation
train_datagen = ImageDataGenerator(
        validation_split=0.2,
        shear_range=0.4,
        zoom_range=0.4,
        rotation_range=60,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(
        validation_split=0.2)

# Data split Train and Validation
train_ds = train_datagen.flow_from_directory(
        data_dir,
        subset='training',
        target_size=img_size,
        batch_size=batch_size,
```

```
        class_mode='categorical',
        seed=42)

valid_ds = test_datagen.flow_from_directory(
        data_dir,
        subset='validation',
        target_size=img_size,
        batch_size=batch_size,
        class_mode='categorical',
        seed=42)

class_names = list(train_ds.class_indices.keys())
print('Class names:', class_names)
```

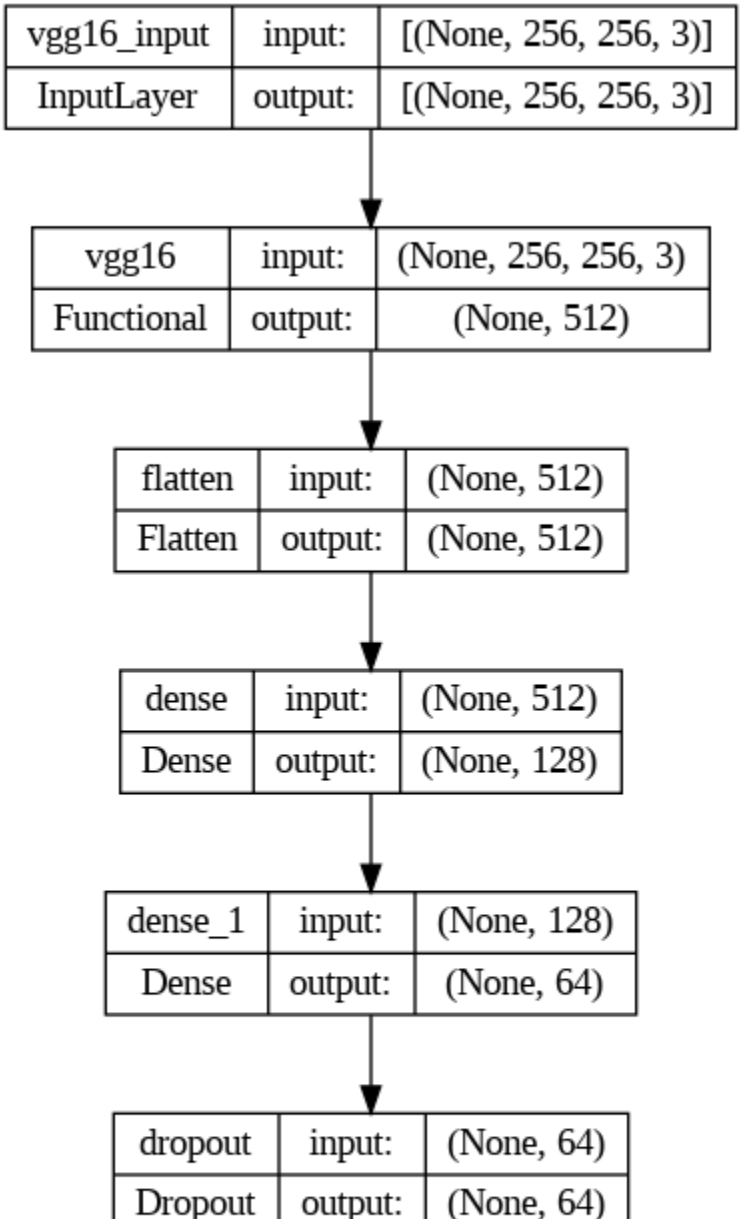
```
Found 2939 images belonging to 5 classes.
Found 731 images belonging to 5 classes.
Class names: ['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Se dividen los datos en entrenamiento y validación. Sin embargo, a los datos de entrenamiento se le implementa la técnica de data augmentation para mejorar la generalización del modelo. Entre otras características, el batch size usado es de 128 y el tamaño de imagen que se usará como entrada es de 200 x 200.

Importar modelo VGG16

```
# @title **Importar modelo VGG16**

pretrained_model = keras.applications.VGG16(
  include_top=False,
  input_shape=img_size+(3,),  # Corrected input shape to match ResNet-50
  pooling='avg',
  classes=len(class_names),
  weights='imagenet')

# Transfer Learning
for layer in pretrained_model.layers:
  layer.trainable = False

resnet_model = Sequential()
resnet_model.add(pretrained_model)
resnet_model.add(Flatten())
resnet_model.add(Dense(128, activation='relu'))
resnet_model.add(Dense(64, activation='relu'))
resnet_model.add(Dropout(0.15))
resnet_model.add(Dense(5, activation='softmax'))
resnet_model.build(input_shape=img_size+(3,))
resnet_model.summary()

from keras.utils import plot_model
from keras.models import load_model  # Import your model

plot_model(resnet_model, show_shapes=True, show_layer_names=True)
```

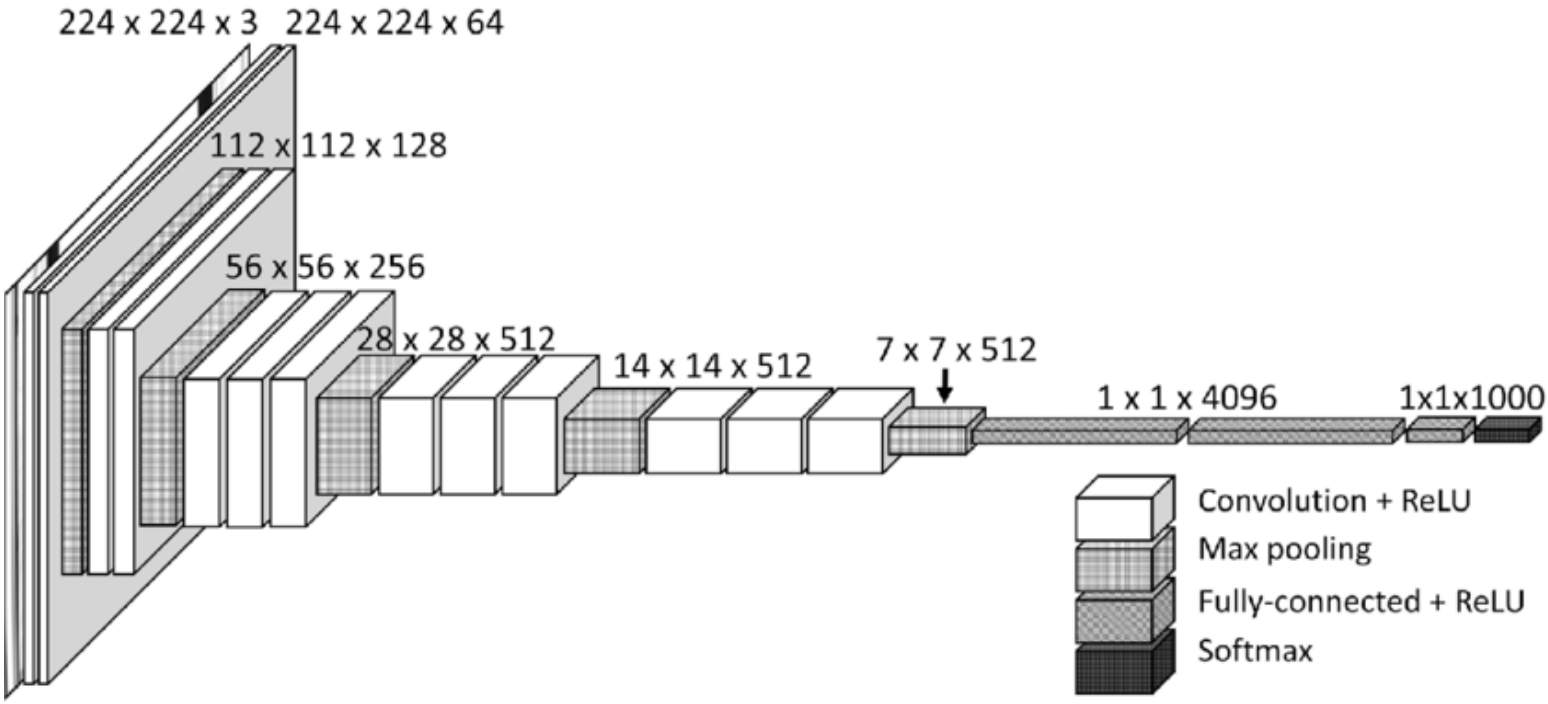
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/
58889256/58889256 [==============================] - 0s 0us/step
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 512)               14714688

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 128)               65664

 dense_1 (Dense)             (None, 64)                8256

 dropout (Dropout)           (None, 64)                0

 dense_2 (Dense)             (None, 5)                 325

=================================================================
Total params: 14788933 (56.42 MB)
Trainable params: 74245 (290.02 KB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```



dense_2	input:	(None, 64)
Dense	output:	(None, 5)

Importamos la arquitectura VGG16 pre-entrenada y configuramos la parte superior para que coincida con nuestras necesidades de clasificación de flores. Aplicamos Transfer Learning congelando las capas pre-entrenadas. Posterior al modelo VGG16, se añade una capa 'Flatten' y dos capas de redes neuronales (de 512 y 5 neurones que serán para la salida) con un 'Dropout' del 10% para regularización.

La arquitectura del modelo VGG16 es la que se muestra en la figura siguiente, sin embargo cabe considerar que la entrada de nuestro modelo fue modificada a 256 x 256 y la salida a 512 neuronas para posteriormente ser adaptada a nuestro nuevo arreglo neuronal.




```
resnet_model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

Se dividen los datos en entrenamiento y validación. Sin embargo, a los datos de entrenamiento se le implementa la técnica de data augmentation para mejorar la generalización del modelo. Entre otras características, el batch size usado es de 128 y el tamaño de imagen que se usará como entrada es de 200 x 200.

Entrenamiento del modelo

```
# @title **Entrenamiento del modelo**

epochs = 15
checkpoint = ModelCheckpoint(
    "best_model.h5",
    monitor="val_accuracy",
    save_best_only=True,
    mode="max")

# Train the model with the callback
history = resnet_model.fit(
    train_ds,
    validation_data=valid_ds,
    epochs=epochs,
    callbacks=[checkpoint])

# Load the best model weights
resnet_model.load_weights("best_model.h5")
```

```
Epoch 1/15
23/23 [==============================] - 114s 4s/step - loss: 2.0123 - accuracy: 0.4532 -
Epoch 2/15
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWarning: Y
  saving_api.save_model(
23/23 [==============================] - 58s 3s/step - loss: 0.8221 - accuracy: 0.6904 -
Epoch 3/15
23/23 [==============================] - 58s 3s/step - loss: 0.6858 - accuracy: 0.7462 -
Epoch 4/15
23/23 [==============================] - 58s 3s/step - loss: 0.5834 - accuracy: 0.7741 -
Epoch 5/15
23/23 [==============================] - 58s 3s/step - loss: 0.5289 - accuracy: 0.7999 -
Epoch 6/15
23/23 [==============================] - 58s 3s/step - loss: 0.5015 - accuracy: 0.8044 -
Epoch 7/15
23/23 [==============================] - 58s 2s/step - loss: 0.4690 - accuracy: 0.8231 -
Epoch 8/15
23/23 [==============================] - 57s 2s/step - loss: 0.4507 - accuracy: 0.8295 -
Epoch 9/15
23/23 [==============================] - 58s 3s/step - loss: 0.4235 - accuracy: 0.8523 -
Epoch 10/15
23/23 [==============================] - 57s 2s/step - loss: 0.4174 - accuracy: 0.8414 -
Epoch 11/15
23/23 [==============================] - 57s 2s/step - loss: 0.3791 - accuracy: 0.8551 -
Epoch 12/15
23/23 [==============================] - 57s 2s/step - loss: 0.3539 - accuracy: 0.8690 -
Epoch 13/15
23/23 [==============================] - 58s 3s/step - loss: 0.3326 - accuracy: 0.8775 -
Epoch 14/15
23/23 [==============================] - 59s 3s/step - loss: 0.3310 - accuracy: 0.8761 -
Epoch 15/15
23/23 [==============================] - 58s 3s/step - loss: 0.3315 - accuracy: 0.8751 -
```

Se entrena el modelo durante 20 épocas utilizando el conjunto de datos de entrenamiento y validación. También implementamos un punto de control ('checkpoint') para guardar el modelo con la mejor precisión en el conjunto de validación.

Visualización de resultados

```
# @title **Visualización de resultados**

fig1 = plt.figure(figsize=(8, 5))
plt.subplot(2,1,1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['train', 'validation'])
plt.axis(xmin=0,xmax=epochs-1)
plt.grid()
plt.show()

fig2 = plt.figure(figsize=(8, 5))
plt.subplot(2,1,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['train', 'validation'])
plt.axis(xmin=0,xmax=epochs-1)
plt.grid()
plt.show()
```

Finalmente, se visualizan los resultados de presición en los conjuntos de entrenamiento y validación, incluyendo la pérdida en función de las épocas. Esto nos permite evaluar el rendimiento del modelo y ver su desempeño a lo largo de las épocas.

En la precisión del modelo podemos observar una tendencia logarítmica a la alza del aprendizaje en el conjunto de datos de entrenamiento, el cual llega hasta el 87.61% de precisión. Sin embargo, en el conjunto de datos de validación, la mejora en la precisión es lineal y considerablemente más lenta, y llega hasta el 87.00% de precisión.

Por el otro lado, la pérdida del modelo a lo largo de las épocas sigue el mismo comportamiento de la precisión, pero a la baja. La pérdida en el conjunto de datos de entrenamiento baja linealmente pero más que la pérdida en el conjunto de datos de prueba. Otra observación es que, en general, la pérdida del modelo en datos de validación es mínima en 0.4160, y 0.3231 en los datos de entrenamiento.


```
from sklearn.metrics import classification_report
import numpy as np

model = load_model("best_model.h5")
y_pred = []
y_true = []

batch_size = 32
random_indices = np.random.choice(len(valid_ds), size=batch_size)

for i in random_indices:
    X_batch, y_batch = valid_ds[i]
    batch_pred = model.predict(X_batch)
    y_pred.extend(np.argmax(batch_pred, axis=1))
    y_true.extend(np.argmax(y_batch, axis=1))

print(classification_report(y_true, y_pred, target_names=class_names))
```

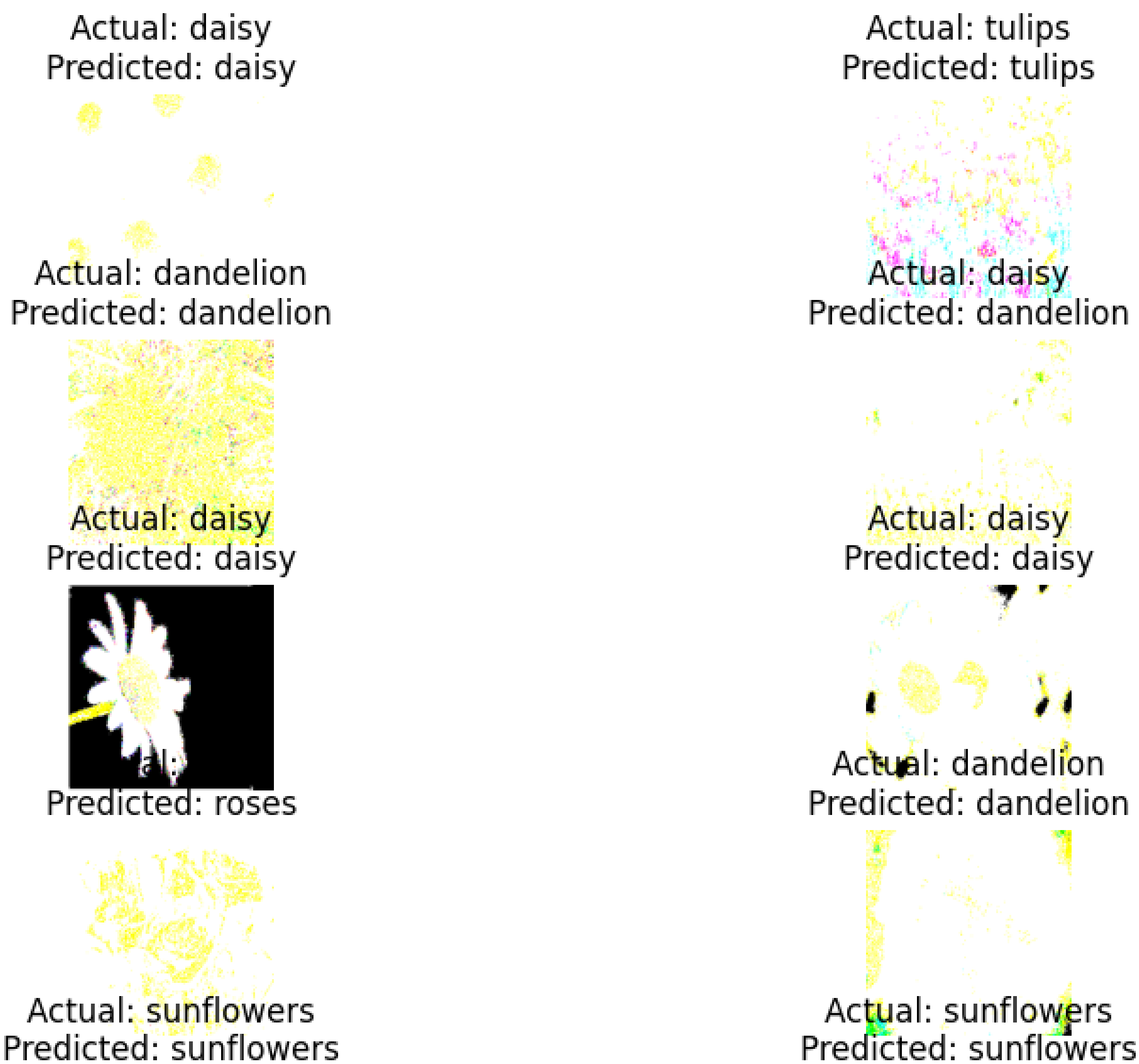
```
4/4 [==============================] - 6s 112ms/step
4/4 [==============================] - 1s 169ms/step
4/4 [==============================] - 1s 156ms/step
4/4 [==============================] - 1s 157ms/step
3/3 [==============================] - 6s 3s/step
3/3 [==============================] - 0s 176ms/step
4/4 [==============================] - 1s 161ms/step
4/4 [==============================] - 1s 159ms/step
4/4 [==============================] - 1s 162ms/step
3/3 [==============================] - 0s 159ms/step
4/4 [==============================] - 1s 158ms/step
4/4 [==============================] - 1s 158ms/step
4/4 [==============================] - 1s 158ms/step
4/4 [==============================] - 1s 160ms/step
4/4 [==============================] - 1s 158ms/step
4/4 [==============================] - 1s 159ms/step
4/4 [==============================] - 1s 159ms/step
3/3 [==============================] - 0s 156ms/step
4/4 [==============================] - 1s 156ms/step
4/4 [==============================] - 1s 157ms/step
```

```
4/4 [==============================] - 1s 157ms/step
4/4 [==============================] - 1s 155ms/step
4/4 [==============================] - 1s 157ms/step
3/3 [==============================] - 0s 155ms/step
4/4 [==============================] - 1s 158ms/step
4/4 [==============================] - 1s 154ms/step
3/3 [==============================] - 0s 155ms/step
4/4 [==============================] - 1s 156ms/step
4/4 [==============================] - 1s 157ms/step
4/4 [==============================] - 1s 157ms/step
4/4 [==============================] - 1s 155ms/step
3/3 [==============================] - 0s 156ms/step
```

	precision	recall	f1-score	support
daisy	0.92	0.84	0.88	654
dandelion	0.88	0.89	0.89	939
roses	0.86	0.83	0.85	672
sunflowers	0.85	0.83	0.84	735
tulips	0.81	0.91	0.86	837
accuracy			0.86	3837
macro avg	0.87	0.86	0.86	3837
weighted avg	0.87	0.86	0.86	3837

```
import numpy as np
import matplotlib.pyplot as plt

# Obtén un lote (batch) de imágenes y etiquetas verdaderas
sample_test_images, ground_truth_labels = next(iter(valid_ds))

# Usa el modelo para predecir las clases
y_pred = model.predict(sample_test_images)
pred_idx = np.argmax(y_pred, axis=-1)

# Nombres de las clases
class_names = list(valid_ds.class_indices.keys())

# Crea una figura con las imágenes y las etiquetas
fig, axes = plt.subplots(len(sample_test_images) // 10, 2, figsize=(10, 20))
axes = axes.flatten()

for i, (img, lbl, pred_id, ax) in enumerate(zip(sample_test_images, ground_truth_labels, pred_idx, axes)):
    if i > len(sample_test_images) // 10 * 2:
      break
    ax.imshow(img)
    label_num = np.argmax(lbl)
    label_actual = class_names[label_num]
    label_predicted = class_names[pred_id]
    label = f"Actual: {label_actual}\nPredicted: {label_predicted}"
    ax.set_title(label)
    ax.axis('off')

plt.show()
```

```
4/4 [==============================] - 1s 172ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
```



Actual: dandelion
Predicted: dandelion

Actual: roses
Predicted: roses

Actual: sunflowers
Predicted: sunflowers

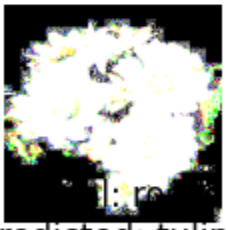
Actual: dandelion
Predicted: dandelion

Actual: daisy
Predicted: daisy

Actual: dandelion
Predicted: dandelion

Actual: daisy
Predicted: daisy

Actual: tulips
Predicted: tulips



Actual: roses
Predicted: tulips

Actual: roses
Predicted: roses

Actual: sunflowers
Predicted: dandelion

Actual: daisy
Predicted: tulips

Actual: dandelion
Predicted: tulips

Actual: dandelion
Predicted: tulips