AI Image Classifier Application

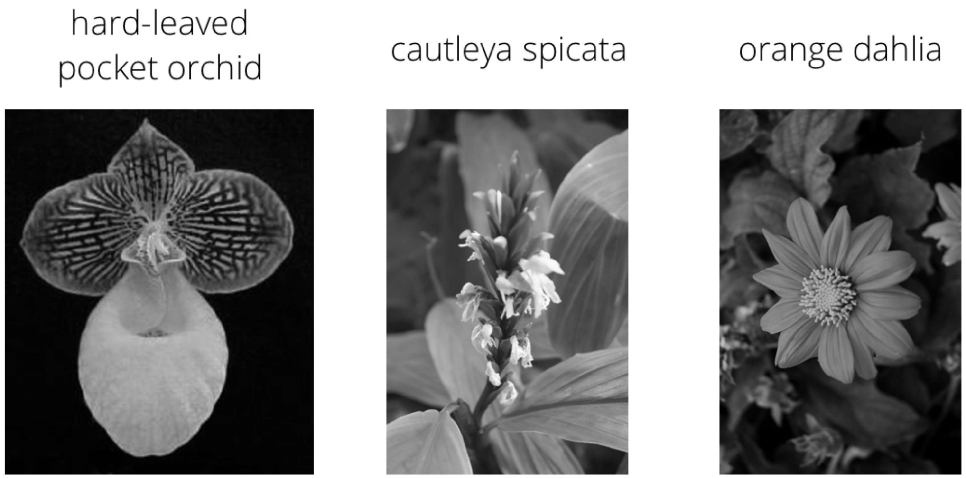
Table of contents

Introduction

Going forward, AI algorithms will be incorporated into more and more everyday applications. For example, you might want to include an image classifier in a smart phone app. To do this, you'd use a deep learning model trained on hundreds of thousands of images as

part of the overall application architecture. A large part of software development in the future will be using these types of models as common parts of applications.

In this project, we'll train an image classifier to recognize different species of flowers. One can imagine using something like this in a phone app that able to recgonize the name of the flower in front of the camera. We'll be using this dataset from Oxford of 102 flower categories, you can see a few examples below.



This project is broken down into multiple steps:

- Load the image dataset and create a pipeline.
- Build and Train an image classifier on this dataset.
- Use the trained model to perform inference on flower images.

Import Resources

```
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import json
import tensorflow_hub as hub
import time
```

```
import numpy as np
```

Control the notebook

```
train = True
load = not(train)
Savedname = 'BestClassifier.h5'
EPOCHS = 100
```

Preparing the data

Loading the Dataset

Here we'll use `tensorflow_datasets` to load the Oxford Flowers 102 dataset. This dataset has 3 splits: `'train'`, `'test'`, and `'validation'`. We'll also need to make sure the training data is normalized and resized to 224x224 pixels as required by the pre-trained networks.

```
# Load the dataset with TensorFlow Datasets.
dataset, dataset_info = tfds.load('oxford_flowers102', as_supervised = True, with_info = True)
```

```
training_set, validation_set, test_set = dataset['train'], dataset['validation'], dataset['test']
```

Exploring the Dataset

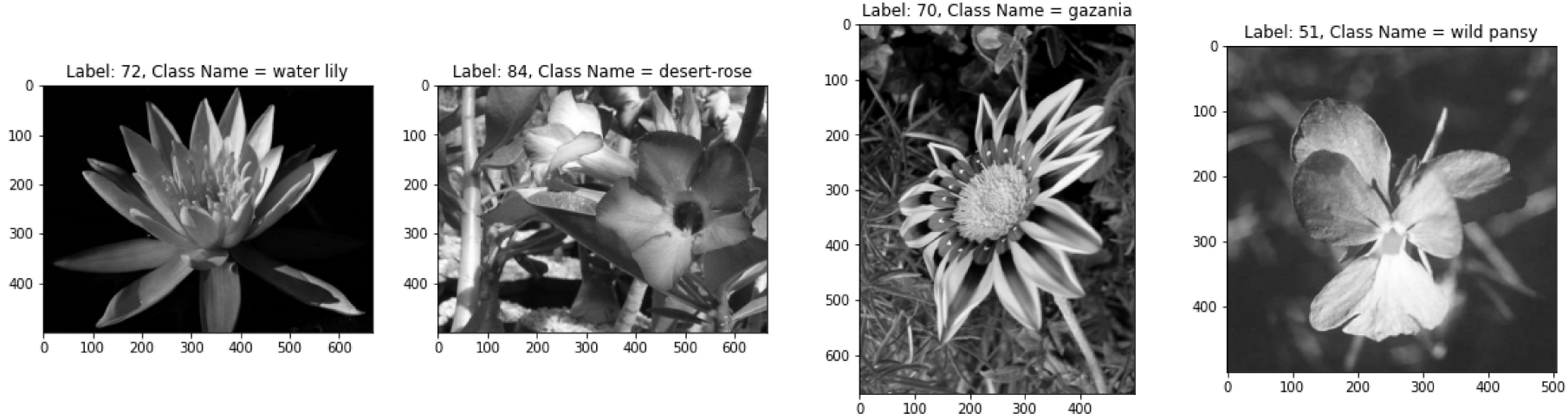
```
# Get the number of classes in the dataset from the dataset info.
num_classes = dataset_info.features['label'].num_classes
# Print important info
print('\u2022 There are {:,} images in the training set'.format(len(training_set)))
```

```
print('\u2022 There are {:,} images in the validation set'.format(len(validation_set)))
print('\u2022 There are {:,} images in the test set'.format(len(test_set)))
print('\u2022 There are {:,} different classes in the dataset'.format(num_classes))
```

- There are 1,020 images in the training set
- There are 1,020 images in the validation set
- There are 6,149 images in the test set
- There are 102 different classes in the dataset

```
# Print some images from the training set.
Names = dataset_info.features['label'].names
num_images = 4
i = 0
fig,ax = plt.subplots(1,num_images, figsize=(5*num_images, 5))

for image, label in training_set.take(num_images):
    image = image.numpy()
    label = label.numpy()
    ax[i].imshow(image)
    ax[i].set_title('Label: {}, Class Name = {}'.format(label,Names[label]))
    i+=1
```



Label Mapping

We need to load in a mapping from label to category name which can be found in the file `label_map.json`. It's a JSON object which gives a dictionary mapping the integer coded labels to the actual names of the flowers.

```
with open('/content/label_map.json', 'r') as f:
    class_names = json.load(f)
```

Create Pipeline

```
def normalize(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    image = tf.image.resize(image, size = (image_size, image_size))
    return image, label

def augment_data(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    image = tf.image.random_contrast(image, 0.4, 1.0)
    image = tf.image.rot90(image)
    return image, label

image_size = 224
batch_size = 32
num_training_examples = len(training_set)
training_batches = training_set.cache().shuffle(num_training_examples//4).padded_batch(batch_size).map(normalize).map(augmer
validation_batches = validation_set.cache().padded_batch(batch_size).map(normalize).prefetch(1)
testing_batches = test_set.cache().padded_batch(batch_size).map(normalize).prefetch(1)
```

Build and Train the Classifier

Now that the data is ready, it's time to build and train the classifier. We would use the MobileNet pre-trained model from TensorFlow Hub to get the image features. Then, we build and train a new feed-forward classifier using those features.

To accomplish this goal, we do the following:

- Load the MobileNet pre-trained network from TensorFlow Hub.
- Define a new, untrained feed-forward network as a classifier.
- Train the classifier.
- Plot the loss and accuracy values achieved during training for the training and validation set.
- Save the trained model as a Keras model.

Building the model

```
if train:
  # Load MobileNet pre-trained model from TensorFlow Hub.
  print('Building the model...')
  URL = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4"
  feature_extractor = hub.KerasLayer(URL, input_shape=(image_size, image_size,3))
  # Preserve trained parameters
  feature_extractor.trainable = False
  # Build the model
  model = tf.keras.Sequential([
          feature_extractor,
          tf.keras.layers.Dense(num_classes, activation = 'softmax')
  ])

  model.summary()
else:
  print('Training is disabled.')
```

```
Building the model...
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
keras_layer_3 (KerasLayer)   (None, 1280)              2257984
_________________________________________________________________
dense_3 (Dense)              (None, 102)               130662
=================================================================
Total params: 2,388,646
Trainable params: 130,662
Non-trainable params: 2,257,984
_________________________________________________________________
```

Training the model

```
if train:
    model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
    model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
                                filepath=Savedname,
                                monitor='val_accuracy',
                                mode='max',
                                save_best_only=True)
    history = model.fit(training_batches,
                        epochs = EPOCHS,
                        validation_data=validation_batches,
                        callbacks = [model_checkpoint_callback, early_stopping])
    history = history.history
    json.dump(history, open('trainHistoryDict.json', 'w'))
else:
  print('Training is disabled.')
```

```
32/32 [==============================] - 15s 466ms/step - loss: 0.1201 - accuracy: 0.9992 - val_loss: 1.1586 - val_a
Epoch 22/100
32/32 [==============================] - 15s 466ms/step - loss: 0.1076 - accuracy: 0.9999 - val_loss: 1.1474 - val_a
Epoch 23/100
32/32 [==============================] - 15s 468ms/step - loss: 0.1119 - accuracy: 0.9991 - val_loss: 1.1404 - val_a
Epoch 24/100
32/32 [==============================] - 15s 463ms/step - loss: 0.0859 - accuracy: 1.0000 - val_loss: 1.1233 - val_a
Epoch 25/100
32/32 [==============================] - 15s 468ms/step - loss: 0.0885 - accuracy: 1.0000 - val_loss: 1.1124 - val_a
Epoch 26/100
```

```
32/32 [==============================] - 15s 466ms/step - loss: 0.0903 - accuracy: 0.9985 - val_loss: 1.1397 - val_a

Epoch 27/100
32/32 [==============================] - 15s 469ms/step - loss: 0.0866 - accuracy: 0.9949 - val_loss: 1.1054 - val_a
Epoch 28/100
32/32 [==============================] - 15s 475ms/step - loss: 0.0673 - accuracy: 1.0000 - val_loss: 1.1113 - val_a
Epoch 29/100
32/32 [==============================] - 15s 468ms/step - loss: 0.0707 - accuracy: 0.9995 - val_loss: 1.1134 - val_a
Epoch 30/100
32/32 [==============================] - 15s 467ms/step - loss: 0.0651 - accuracy: 0.9998 - val_loss: 1.1006 - val_a
Epoch 31/100
32/32 [==============================] - 15s 468ms/step - loss: 0.0651 - accuracy: 0.9967 - val_loss: 1.0884 - val_a
Epoch 32/100
32/32 [==============================] - 15s 465ms/step - loss: 0.0674 - accuracy: 1.0000 - val_loss: 1.0851 - val_a
Epoch 33/100
32/32 [==============================] - 15s 467ms/step - loss: 0.0563 - accuracy: 1.0000 - val_loss: 1.0955 - val_a
Epoch 34/100
32/32 [==============================] - 15s 466ms/step - loss: 0.0549 - accuracy: 1.0000 - val_loss: 1.0708 - val_a
Epoch 35/100
32/32 [==============================] - 15s 467ms/step - loss: 0.0541 - accuracy: 0.9999 - val_loss: 1.0638 - val_a
Epoch 36/100
32/32 [==============================] - 15s 465ms/step - loss: 0.0557 - accuracy: 0.9998 - val_loss: 1.0779 - val_a
Epoch 37/100
32/32 [==============================] - 15s 468ms/step - loss: 0.0476 - accuracy: 1.0000 - val_loss: 1.0700 - val_a
Epoch 38/100
32/32 [==============================] - 15s 466ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 1.0627 - val_a
Epoch 39/100
32/32 [==============================] - 15s 469ms/step - loss: 0.0430 - accuracy: 1.0000 - val_loss: 1.0751 - val_a
Epoch 40/100
32/32 [==============================] - 15s 463ms/step - loss: 0.0421 - accuracy: 0.9998 - val_loss: 1.0647 - val_a
Epoch 41/100
32/32 [==============================] - 15s 466ms/step - loss: 0.0450 - accuracy: 1.0000 - val_loss: 1.0619 - val_a
Epoch 42/100
32/32 [==============================] - 15s 463ms/step - loss: 0.0339 - accuracy: 1.0000 - val_loss: 1.0488 - val_a
Epoch 43/100
32/32 [==============================] - 15s 465ms/step - loss: 0.0342 - accuracy: 0.9997 - val_loss: 1.0536 - val_a
Epoch 44/100
32/32 [==============================] - 15s 467ms/step - loss: 0.0389 - accuracy: 0.9971 - val_loss: 1.0478 - val_a
Epoch 45/100
32/32 [==============================] - 15s 464ms/step - loss: 0.0344 - accuracy: 0.9996 - val_loss: 1.0575 - val_a
Epoch 46/100
32/32 [==============================] - 15s 466ms/step - loss: 0.0378 - accuracy: 1.0000 - val_loss: 1.0668 - val_a
Epoch 47/100
32/32 [==============================] - 15s 472ms/step - loss: 0.0368 - accuracy: 0.9983 - val_loss: 1.0563 - val_a
```

```
Epoch 48/100
32/32 [==============================] - 15s 464ms/step - loss: 0.0396 - accuracy: 0.9964 - val_loss: 1.0562 - val_a
Epoch 49/100
32/32 [==============================] - 15s 464ms/step - loss: 0.0318 - accuracy: 1.0000 - val_loss: 1.0583 - val_a
```

Loading the model

In case we want only need to load the saved model, we can load it using:

```
# Load the Keras model
if load:
    model = tf.keras.models.load_model(Savedname, custom_objects={'KerasLayer':hub.KerasLayer})
    model.summary()
    history = json.load(open('trainHistoryDict.json', 'r'))
```

Loss curve

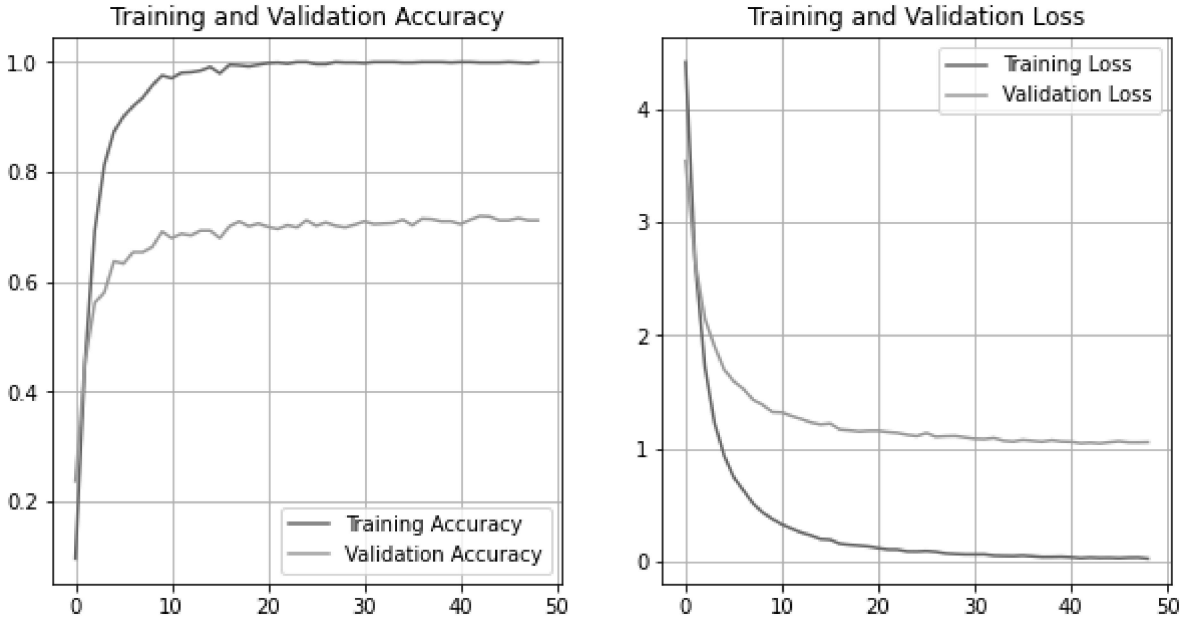
```
training_accuracy = history['accuracy']
validation_accuracy = history['val_accuracy']

training_loss = history['loss']
validation_loss = history['val_loss']

epochs_range=range(len(history['loss']))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, training_accuracy, label='Training Accuracy')
plt.plot(epochs_range, validation_accuracy, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.subplot(1, 2, 2)
```

```
plt.subplot(1, 2, 2)
plt.plot(epochs_range, training_loss, label='Training Loss')
plt.plot(epochs_range, validation_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.grid()
plt.show()
```



Testing the Network

It's a good practice to test the trained network on the test data, images the network has never seen either in training or validation. This will give us a good estimate for the model's performance on completely new images.

```
# Print the loss and accuracy values achieved on the entire test set.
loss, accuracy = model.evaluate(testing_batches)
```

```
193/193 [==============================] - 43s 219ms/step - loss: 1.2286 - accuracy: 0.6954
```

Inference for Classification

Now we'll write a function that uses the trained network for inference which takes an image, a model, and then returns the top K most likely class labels along with the probabilities.

Image Pre-processing

The `process_image` function should take in an image (in the form of a NumPy array) and return an image in the form of a NumPy array with shape `(224, 224, 3)`.

```
# Create the process_image function
def process_image(image):
    imten = tf.convert_to_tensor(image)
    imten = tf.image.resize(imten, (image_size,image_size))
    imten = tf.cast(imten, tf.float32)
    imten /= 255
    return imten.numpy()
```

To check the `process_image` function we have provided 4 images in the `./test_images/` folder:

- cautleya_spicata.jpg
- hard-leaved_pocket_orchid.jpg
- orange_dahlia.jpg
- wild_pansy.jpg

The code below loads one of the above images using `PIL` and plots the original image alongside the image produced by the `process_image` function.

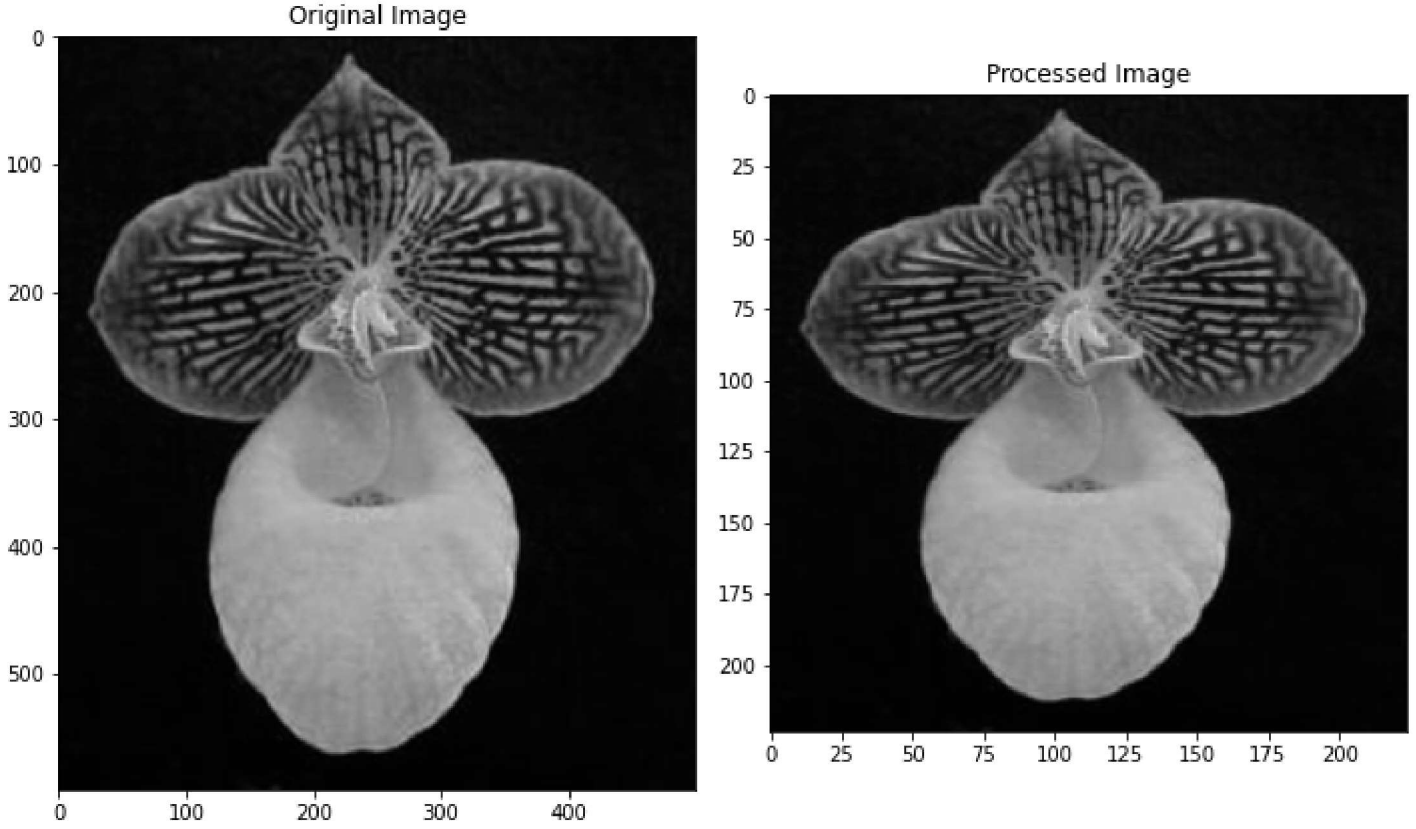
```
from PIL import Image

image_path = '/content/hard-leaved_pocket_orchid.jpg'
```

```
im = Image.open(image_path)
test_image = np.asarray(im)

processed_test_image = process_image(test_image)

fig, (ax1, ax2) = plt.subplots(figsize=(10,10), ncols=2)
ax1.imshow(test_image)
ax1.set_title('Original Image')
ax2.imshow(processed_test_image)
ax2.set_title('Processed Image')
plt.tight_layout()
plt.show()
```



Inference

Once we can get the images in the correct format, it's time to write the `predict` function for making inference with your model.

```
def predict(image_path , model , top_k):
    im = Image.open(image_path)
    test_image = np.asarray(im)
    processed_test_image = process_image(test_image)
    all_probs =  model.predict(np.expand_dims(processed_test_image, axis = 0))
    all_classes = np.argsort(all_probs)
    probs = all_probs[:,all_classes[:,-top_k:]]
    classes = all_classes[:,-top_k:]
    return probs[0][0], classes[0]
```

Sanity Check

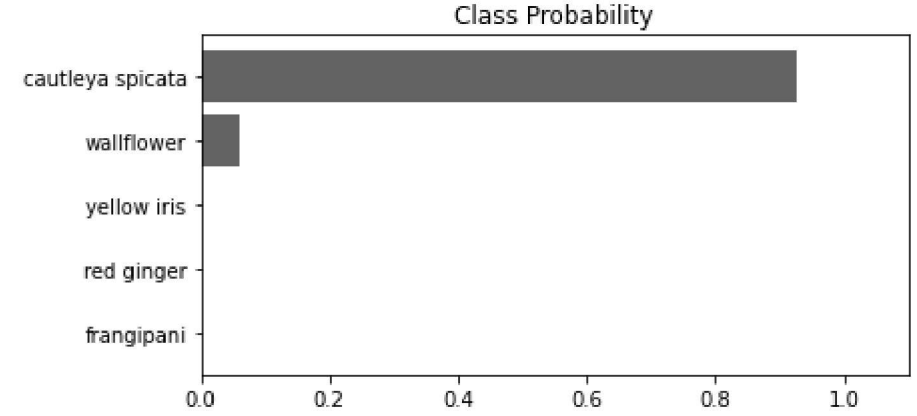
It's always good to check the predictions made by the model to make sure they are correct. To check our predictions we have provided 4 images in the `./test_images/` folder:

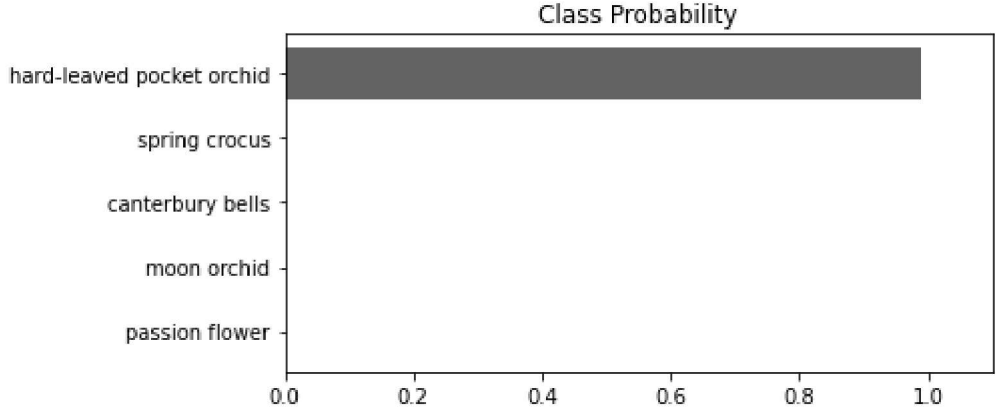
- cautleya_spicata.jpg
- hard-leaved_pocket_orchid.jpg
- orange_dahlia.jpg
- wild_pansy.jpg

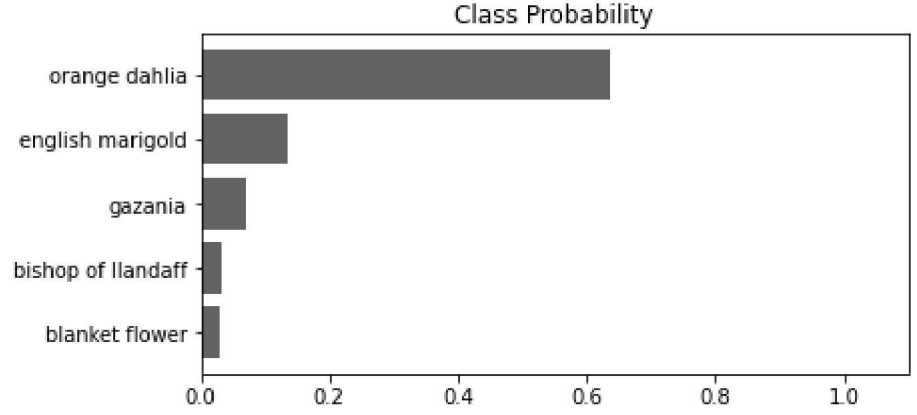
```
probs, classes = predict(image_path, model, 5 )
```

```
test_images = ['cautleya_spicata.jpg', 'hard-leaved_pocket_orchid.jpg', 'orange_dahlia.jpg','wild_pansy.jpg']
fig, ax = plt.subplots(figsize=(15,12), nrows = len(test_images), ncols=2)
i = 0
for imagepath in test_images:
    probs, classes = predict('/content/' + imagepath, model, 5 )
    ax[i][0].imshow(Image.open('/content/' + imagepath), cmap = plt.cm.binary)
    ax[i][0].axis('off')
    ax[i][1].barh([class_names[str(i+1)] for i in classes], probs)
    ax[i][1].set_aspect(0.1)
```

```
    ax[i][1].set_yticks(np.arange(5))
    ax[i][1].set_title('Class Probability')
    ax[i][1].set_xlim(0, 1.1)
    plt.tight_layout()
    i+= 1
```


Class Probability
cautleya spicata
wallflower
yellow iris
red ginger
frangipani
0.0
0.2
0.4
0.6
0.8
1.0


Class Probability
hard-leaved pocket orchid
spring crocus
canterbury bells
moon orchid
passion flower
0.0
0.2
0.4
0.6
0.8
1.0


Class Probability
orange dahlia
english marigold
gazania
bishop of llandaff
blanket flower
0.0
0.2
0.4
0.6
0.8
1.0