Image classification

```
import matplotlib.pyplot as plt
import numpy as np
#The Python Imaging Library adds image processing capabilities to your
Python interpreter.
import PIL
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

Download and explore the dataset

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains five sub-directories, one per class:

```
flower_photo/
  daisy/
  dandelion/
  roses/
  sunflowers/
  tulips/
```

```
import pathlib
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)
```

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

```
data_dir
```

```
PosixPath('/root/.keras/datasets/flower_photos')
```

After downloading, you should now have a copy of the dataset available. There are 3,670 total images:

```
image_count = len(list(data_dir.glob('*/*.jpg')))
```

```
print(image_count)
```

```
3670
```

Here are some roses:

```
roses =list(data_dir.glob('roses/*'))
```

```
PIL.Image.open(str(roses[0]))
```

```
PIL.Image.open(str(roses[1]))
```

And some tulips:

```
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
```

```
PIL.Image.open(str(tulips[1]))
```

Load data using a Keras utility

Next, load these images off disk using the helpful `tf.keras.utils.image_dataset_from_directory` utility. This will take you from a directory of images on disk to a `tf.data.Dataset` in just a couple lines of code. If you like, you can also write your own data loading code from scratch by visiting the Load and preprocess images tutorial.

Create a dataset

Define some parameters for the loader:

```
batch_size = 32
img_height = 180
img_width = 180
```

```
train1_ds = tf.keras.utils.image_dataset_from_directory(data_dir,

validation_split=0.2,

subset="training",
                                                         seed=123,

image_size=(img_height,

img_width),

batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

```
train1_ds.class_names
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

It's good practice to use a validation split when developing your model. Use 80% of the images for training and 20% for validation.

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

```
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

You can find the class names in the `class_names` attribute on these datasets. These correspond to the directory names in alphabetical order.

```
class_names = train_ds.class_names
print(class_names)
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Visualize the data

Here are the first nine images from the training dataset:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```

You will pass these datasets to the Keras `Model.fit` method for training later in this tutorial. If you like, you can also manually iterate over the dataset and retrieve batches of images:

```
for image_batch, labels_batch in train_ds:
  print(image_batch.shape)
  print(labels_batch.shape)
  break
```

```
(32, 180, 180, 3)
(32,)
```

The `image_batch` is a tensor of the shape `(32, 180, 180, 3)`. This is a batch of 32 images of shape `180x180x3` (the last dimension refers to color channels RGB). The `label_batch` is a tensor of the shape `(32,)`, these are corresponding labels to the 32 images.

You can call `.numpy()` on the `image_batch` and `labels_batch` tensors to convert them to a `numpy.ndarray`.

Configure the dataset for performance

Make sure to use buffered prefetching, so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data:

- `Dataset.cache` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.
- `Dataset.prefetch` overlaps data preprocessing and model execution while training.

Interested readers can learn more about both methods, as well as how to cache data to disk in the *Prefetching* section of the Better performance with the tf.data API guide.

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds =
train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Standardize the data

The RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small.

Here, you will standardize values to be in the `[0, 1]` range by using `tf.keras.layers.Rescaling`:

```
normalization_layer = layers.Rescaling(1./255)
```

There are two ways to use this layer. You can apply it to the dataset by calling `Dataset.map`:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

```
0.0 1.0
```

Or, you can include the layer inside your model definition, which can simplify deployment. Use the second approach here.

Note: You previously resized images using the `image_size` argument of `tf.keras.utils.image_dataset_from_directory`. If you want to include the resizing logic in your model as well, you can use the `tf.keras.layers.Resizing` layer.

A basic Keras model

Create the model

The Keras Sequential model consists of three convolution blocks (`tf.keras.layers.Conv2D`) with a max pooling layer (`tf.keras.layers.MaxPooling2D`) in each of them. There's a fully-connected layer (`tf.keras.layers.Dense`) with 128 units on top of it that is activated by a ReLU activation function (`'relu'`). This model has not been tuned for high accuracy; the goal of this tutorial is to show a standard approach.

```
num_classes = len(class_names)

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

Compile the model

For this tutorial, choose the `tf.keras.optimizers.Adam` optimizer and `tf.keras.losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument to `Model.compile`.

```
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Model summary

View all the layers of the network using the Keras `Model.summary` method:

```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_1 (Rescaling)     (None, 180, 180, 3)       0

 conv2d (Conv2D)             (None, 180, 180, 16)      448
```

```
 max_pooling2d (MaxPooling2D  (None, 90, 90, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 90, 90, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 45, 45, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 45, 45, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 flatten (Flatten)           (None, 30976)             0

 dense (Dense)               (None, 128)               3965056

 dense_1 (Dense)             (None, 5)                 645

=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
_________________________________________________________________
```

Train the model

Train the model for 10 epochs with the Keras `Model.fit` method:

```
epochs=10
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

```
Epoch 1/10
92/92 [==============================] - 12s 38ms/step - loss: 1.4404
- accuracy: 0.3924 - val_loss: 1.1169 - val_accuracy: 0.5354
Epoch 2/10
92/92 [==============================] - 2s 24ms/step - loss: 1.0709 -
accuracy: 0.5695 - val_loss: 1.0114 - val_accuracy: 0.5926
Epoch 3/10
92/92 [==============================] - 2s 23ms/step - loss: 0.9107 -
accuracy: 0.6403 - val_loss: 0.9360 - val_accuracy: 0.6281
Epoch 4/10
92/92 [==============================] - 2s 23ms/step - loss: 0.7201 -
accuracy: 0.7268 - val_loss: 0.9047 - val_accuracy: 0.6512
Epoch 5/10
92/92 [==============================] - 2s 23ms/step - loss: 0.5211 -
```

```
accuracy: 0.8086 - val_loss: 1.0094 - val_accuracy: 0.6362
Epoch 6/10
92/92 [==============================] - 2s 23ms/step - loss: 0.3106 -
accuracy: 0.8927 - val_loss: 1.1783 - val_accuracy: 0.6213
Epoch 7/10
92/92 [==============================] - 2s 23ms/step - loss: 0.1993 -
accuracy: 0.9349 - val_loss: 1.4062 - val_accuracy: 0.6281
Epoch 8/10
92/92 [==============================] - 2s 23ms/step - loss: 0.1617 -
accuracy: 0.9506 - val_loss: 1.4861 - val_accuracy: 0.6335
Epoch 9/10
92/92 [==============================] - 2s 23ms/step - loss: 0.0988 -
accuracy: 0.9700 - val_loss: 1.7732 - val_accuracy: 0.6226
Epoch 10/10
92/92 [==============================] - 2s 23ms/step - loss: 0.0534 -
accuracy: 0.9847 - val_loss: 1.8406 - val_accuracy: 0.6049
```

Visualize training results

Create plots of the loss and accuracy on the training and validation sets:

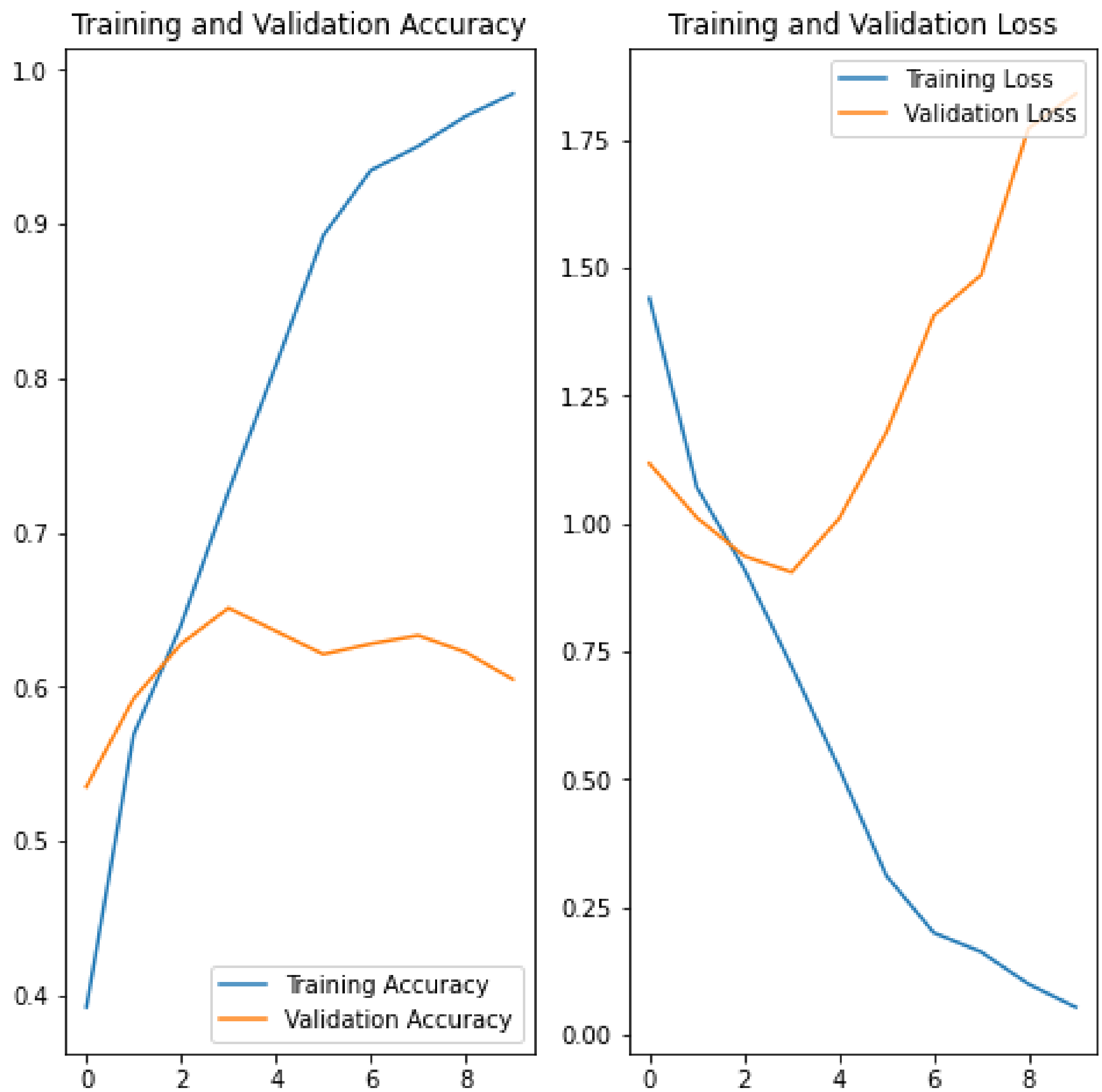
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

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



The plots show that training accuracy and validation accuracy are off by large margins, and the model has achieved only around 60% accuracy on the validation set.

The following tutorial sections show how to inspect what went wrong and try to increase the overall performance of the model.

Overfitting

In the plots above, the training accuracy is increasing linearly over time, whereas validation accuracy stalls around 60% in the training process. Also, the difference in accuracy between training and validation accuracy is noticeable—a sign of overfitting.

When there are a small number of training examples, the model sometimes learns from noises or unwanted details from training examples—to an extent that it negatively impacts the performance of the model on new examples. This phenomenon is known as overfitting. It means that the model will have a difficult time generalizing on a new dataset.

There are multiple ways to fight overfitting in the training process. In this tutorial, you'll use *data augmentation* and add *dropout* to your model.

Data augmentation

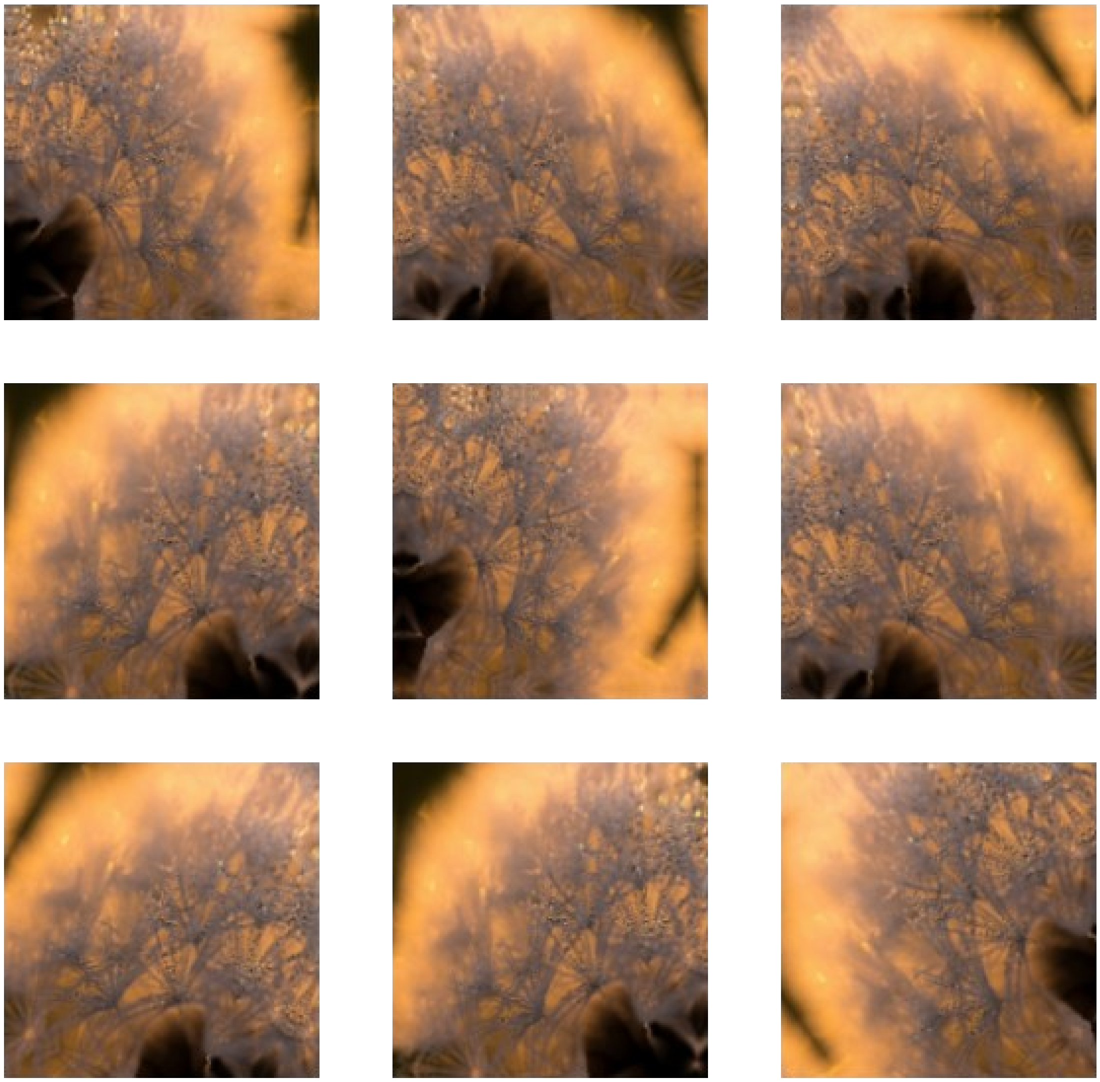
Overfitting generally occurs when there are a small number of training examples. Data augmentation takes the approach of generating additional training data from your existing examples by augmenting them using random transformations that yield believable-looking images. This helps expose the model to more aspects of the data and generalize better.

You will implement data augmentation using the following Keras preprocessing layers: `tf.keras.layers.RandomFlip`, `tf.keras.layers.RandomRotation`, and `tf.keras.layers.RandomZoom`. These can be included inside your model like other layers, and run on the GPU.

```
data_augmentation = keras.Sequential(
  [
    layers.RandomFlip("horizontal",
                      input_shape=(img_height,
                                   img_width,
                                   3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
  ]
)
```

Visualize a few augmented examples by applying data augmentation to the same image several times:

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
  for i in range(9):
    augmented_images = data_augmentation(images)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_images[0].numpy().astype("uint8"))
    plt.axis("off")
```



You will add data augmentation to your model before training in the next step.

Dropout

Another technique to reduce overfitting is to introduce dropout{:.external} regularization to the network.

When you apply dropout to a layer, it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

Create a new neural network with `tf.keras.layers.Dropout` before training it using the augmented images:

```
model = Sequential([
  data_augmentation,
```

```
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes, name="outputs")
])
```

Compile and train the model

```
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.summary()

Model: "sequential_2"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_1 (Sequential)   (None, 180, 180, 3)       0

 rescaling_2 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_3 (Conv2D)           (None, 180, 180, 16)      448

 max_pooling2d_3 (MaxPooling  (None, 90, 90, 16)       0
 2D)

 conv2d_4 (Conv2D)           (None, 90, 90, 32)        4640

 max_pooling2d_4 (MaxPooling  (None, 45, 45, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 45, 45, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 dropout (Dropout)           (None, 22, 22, 64)        0

 flatten_1 (Flatten)         (None, 30976)             0

 dense_2 (Dense)             (None, 128)               3965056
```

```
 outputs (Dense)             (None, 5)                 645

=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
_________________________________________________________________
```

```
epochs = 15
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

```
Epoch 1/15
92/92 [==============================] - 6s 51ms/step - loss: 1.2808 - accuracy: 0.4646 - val_loss: 1.1099 - val_accuracy: 0.5504
Epoch 2/15
92/92 [==============================] - 4s 40ms/step - loss: 1.0604 - accuracy: 0.5773 - val_loss: 1.0105 - val_accuracy: 0.6008
Epoch 3/15
92/92 [==============================] - 4s 41ms/step - loss: 0.9531 - accuracy: 0.6325 - val_loss: 0.9612 - val_accuracy: 0.6226
Epoch 4/15
92/92 [==============================] - 4s 40ms/step - loss: 0.8849 - accuracy: 0.6557 - val_loss: 0.8793 - val_accuracy: 0.6499
Epoch 5/15
92/92 [==============================] - 4s 41ms/step - loss: 0.8464 - accuracy: 0.6747 - val_loss: 0.8700 - val_accuracy: 0.6649
Epoch 6/15
92/92 [==============================] - 4s 41ms/step - loss: 0.8042 - accuracy: 0.6887 - val_loss: 0.8351 - val_accuracy: 0.6866
Epoch 7/15
92/92 [==============================] - 4s 41ms/step - loss: 0.7630 - accuracy: 0.7101 - val_loss: 0.7955 - val_accuracy: 0.6785
Epoch 8/15
92/92 [==============================] - 4s 41ms/step - loss: 0.7151 - accuracy: 0.7275 - val_loss: 0.8050 - val_accuracy: 0.6866
Epoch 9/15
92/92 [==============================] - 4s 40ms/step - loss: 0.6901 - accuracy: 0.7360 - val_loss: 0.8583 - val_accuracy: 0.6717
Epoch 10/15
92/92 [==============================] - 4s 39ms/step - loss: 0.6761 - accuracy: 0.7371 - val_loss: 0.7300 - val_accuracy: 0.7003
Epoch 11/15
92/92 [==============================] - 4s 40ms/step - loss: 0.6333 - accuracy: 0.7606 - val_loss: 0.7450 - val_accuracy: 0.7248
Epoch 12/15
92/92 [==============================] - 4s 39ms/step - loss: 0.6078 - accuracy: 0.7691 - val_loss: 0.7622 - val_accuracy: 0.7057
```

```
Epoch 13/15
92/92 [==============================] - 4s 40ms/step - loss: 0.5892 - 
accuracy: 0.7868 - val_loss: 0.7019 - val_accuracy: 0.7330
Epoch 14/15
92/92 [==============================] - 4s 40ms/step - loss: 0.5530 - 
accuracy: 0.7950 - val_loss: 0.7281 - val_accuracy: 0.7275
Epoch 15/15
92/92 [==============================] - 4s 39ms/step - loss: 0.5355 - 
accuracy: 0.7984 - val_loss: 0.7469 - val_accuracy: 0.7221
```

Visualize training results

After applying data augmentation and `tf.keras.layers.Dropout`, there is less overfitting than before, and training and validation accuracy are closer aligned:

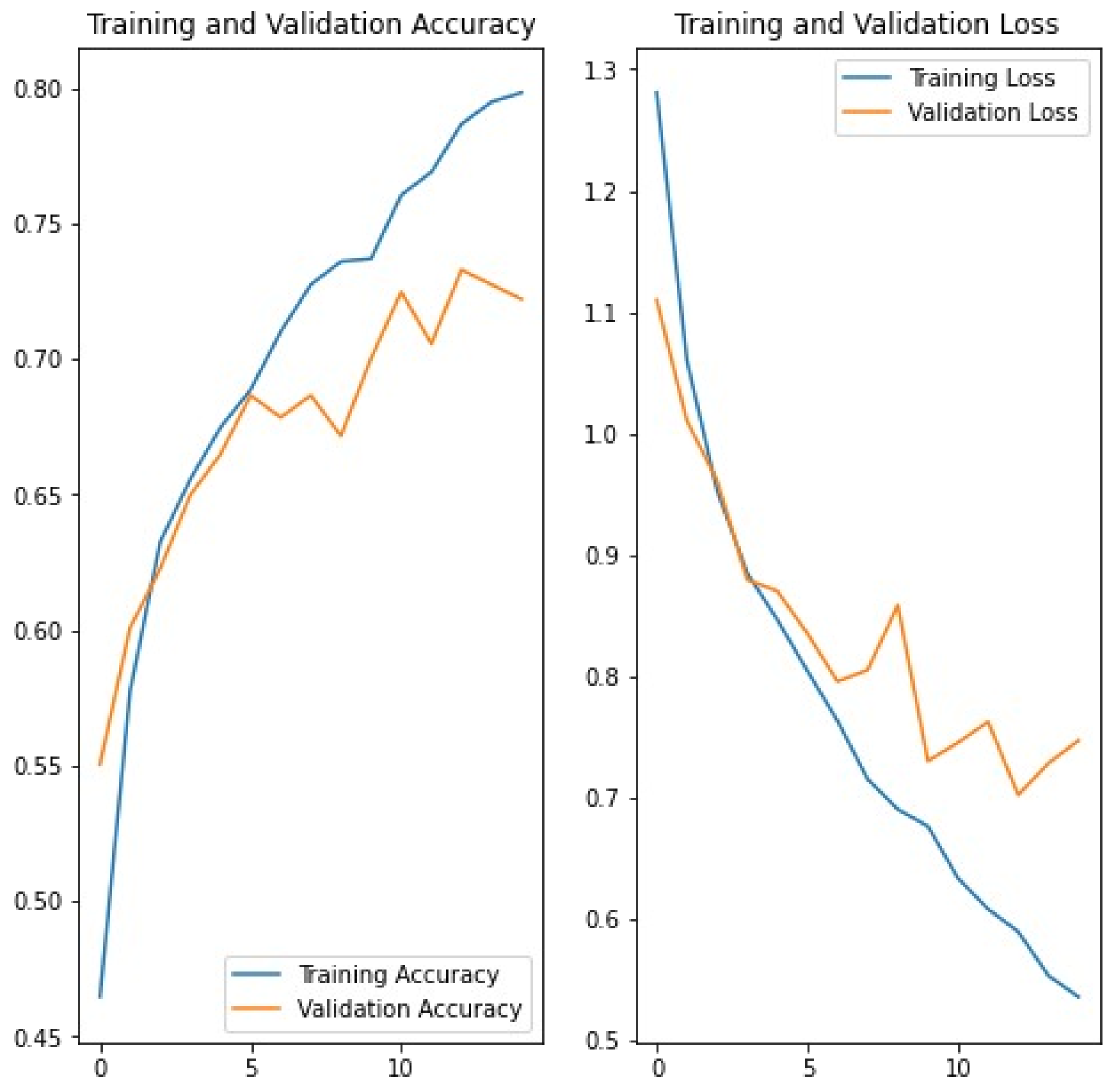
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

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



Predict on new data

Use your model to classify an image that wasn't included in the training or validation sets.

Note: Data augmentation and dropout layers are inactive at inference time.

```
sunflower_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images
/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower',
origin=sunflower_url)
```

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/example_images/
592px-Red_sunflower.jpg
117948/117948 [==============================] - 0s 0us/step
```

```
sunflower_path
```

```
{"type":"string"}
```

```
sunflower_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images
/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower',
origin=sunflower_url)


img = tf.keras.utils.load_img(
    sunflower_path, target_size=(img_height, img_width)
)

img_array = tf.keras.utils.img_to_array(img)

img_array.shape
```

```
(180, 180, 3)
```

```
img_array = tf.expand_dims(img_array, 0) # Create a batch


img_array.shape
```

```
TensorShape([1, 180, 180, 3])
```

```
predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent
confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

```
1/1 [==============================] - 0s 168ms/step
This image most likely belongs to sunflowers with a 99.19 percent
confidence.
```

```
predictions
```

```
array([[-3.4378102, -2.0066147,  1.3114015,  7.5418034,  2.4469857]],
      dtype=float32)
```

Use TensorFlow Lite

TensorFlow Lite is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices.

Convert the Keras Sequential model to a TensorFlow Lite model

To use the trained model with on-device applications, first convert it to a smaller and more efficient model format called a TensorFlow Lite model.

In this example, take the trained Keras Sequential model and use `tf.lite.TFLiteConverter.from_keras_model` to generate a TensorFlow Lite model:

```
# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

The TensorFlow Lite model you saved in the previous step can contain several function signatures. The Keras model converter API uses the default signature automatically. Learn more about TensorFlow Lite signatures.

Run the TensorFlow Lite model

You can access the TensorFlow Lite saved model signatures in Python via the `tf.lite.Interpreter` class.

Load the model with the `Interpreter`:

```
TF_MODEL_FILE_PATH = 'model.tflite' # The default path to the saved TensorFlow Lite model

interpreter = tf.lite.Interpreter(model_path=TF_MODEL_FILE_PATH)
```

Print the signatures from the converted model to obtain the names of the inputs (and outputs):

```
interpreter.get_signature_list()
```

In this example, you have one default signature called `serving_default`. In addition, the name of the `'inputs'` is `'sequential_1_input'`, while the `'outputs'` are called `'outputs'`. You can look up these first and last Keras layer names when running `Model.summary`, as demonstrated earlier in this tutorial.

Now you can test the loaded TensorFlow Model by performing inference on a sample image with `tf.lite.Interpreter.get_signature_runner` by passing the signature name as follows:

```
classify_lite = interpreter.get_signature_runner('serving_default')
classify_lite
```

Similar to what you did earlier in the tutorial, you can use the TensorFlow Lite model to classify images that weren't included in the training or validation sets.

You have already tensorized that image and saved it as `img_array`. Now, pass it to the first argument (the name of the `'inputs'`) of the loaded TensorFlow Lite model (`predictions_lite`), compute softmax activations, and then print the prediction for the class with the highest computed probability.

```
predictions_lite = classify_lite(sequential_1_input=img_array)
['outputs']
score_lite = tf.nn.softmax(predictions_lite)
```

```
print(
    "This image most likely belongs to {} with a {:.2f} percent
confidence."
    .format(class_names[np.argmax(score_lite)], 100 *
np.max(score_lite))
)
```

The prediction generated by the lite model should be almost identical to the predictions generated by the original model:

```
print(np.max(np.abs(predictions - predictions_lite)))
```

Of the five classes—`'daisy'`, `'dandelion'`, `'roses'`, `'sunflowers'`, and `'tulips'`—the model should predict the image belongs to sunflowers, which is the same result as before the TensorFlow Lite conversion.

Next steps

This tutorial showed how to train a model for image classification, test it, convert it to the TensorFlow Lite format for on-device applications (such as an image classification app), and perform inference with the TensorFlow Lite model with the Python API.

You can learn more about TensorFlow Lite through tutorials and guides.