Identify Flowers using Transfer Learning

What is Transfer Learning?

Transfer learning is a technique that shortcuts much of this by taking a piece of a model that has already been trained on a related task and reusing it in a new model.

Here I am using google colab which gives ram and GPU integration in the browser :)

Step 1: Installation

```python
import tensorflow as tf
assert tf.__version__.startswith('2')

import os
import numpy as np
```

Step 2: Setup Input Pipeline

Downloading the flower dataset…

```python
_URL = "https://storage.googleapis.com/download.tensor

zip_file = tf.keras.utils.get_file(origin=_URL,
                                   fname="flower_photo
                                   extract=True)
```

Use ImageDataGenerator to rescale the images.

Create the trained generator and specify where the train dataset directory, image size, batch size.

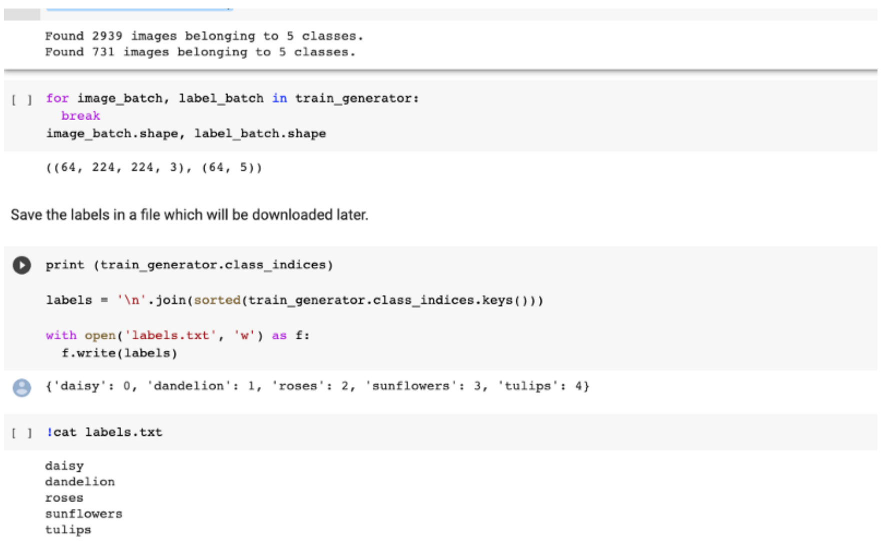
Create the validation generator with a similar approach as the trained generator with the flow_from_directory() method.

```python
IMAGE_SIZE = 224
BATCH_SIZE = 64

datagen = tf.keras.preprocessing.image.ImageDataGener
    rescale=1./255,
    validation_split=0.2)

train_generator = datagen.flow_from_directory(
    base_dir,
    target_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    subset='training')
```

After running the above code, you will see 2939 images in training and 731 for validation dataset as shown in the below screenshot:-



```
Found 2939 images belonging to 5 classes.
Found 731 images belonging to 5 classes.
```

```
[ ] for image_batch, label_batch in train_generator:
      break
    image_batch.shape, label_batch.shape
```

```
((64, 224, 224, 3), (64, 5))
```

Save the labels in a file which will be downloaded later.

```
print (train_generator.class_indices)

labels = '\n'.join(sorted(train_generator.class_indices.keys()))

with open('labels.txt', 'w') as f:
  f.write(labels)
```

```
{'daisy': 0, 'dandelion': 1, 'roses': 2, 'sunflowers': 3, 'tulips': 4}
```

```
[ ] !cat labels.txt
```

```
daisy
dandelion
roses
sunflowers
tulips
```

Create the base model from the pre-trained convnets(Convolutional Network). Create the base model from the MobileNet V2 model developed at Google, and pre-trained on the ImageNet dataset, a large dataset of 1.4M images and 1000 classes of web images.

First, pick which intermediate layer of MobileNet V2 will be used for feature extraction. A common practice is to use the output of the very last layer before the flatten operation, the so-called "bottleneck layer". The reasoning here is that the following fully-connected layers will be too specialized to the task the network was trained on, and thus the features learned by these layers won't be very useful for a new task. The bottleneck features, however, retain many generalities.

Let's instantiate a MobileNet V2 model pre-loaded with weights trained on ImageNet. By specifying the include_top=False argument, we load a network that doesn't include the classification layers at the top, which is ideal for feature extraction.

```
IMG_SHAPE = (IMAGE_SIZE, IMAGE_SIZE, 3)

# Create the base model from the pre-trained model Mob
base_model = tf.keras.applications.MobileNetV2(input_s
                                               include_
```

Feature extraction You will freeze the convolutional base created from the previous step and use that as a feature extractor, add a classifier on top of it and train the top-level classifier.

base_model.trainable = False

Add a classification head

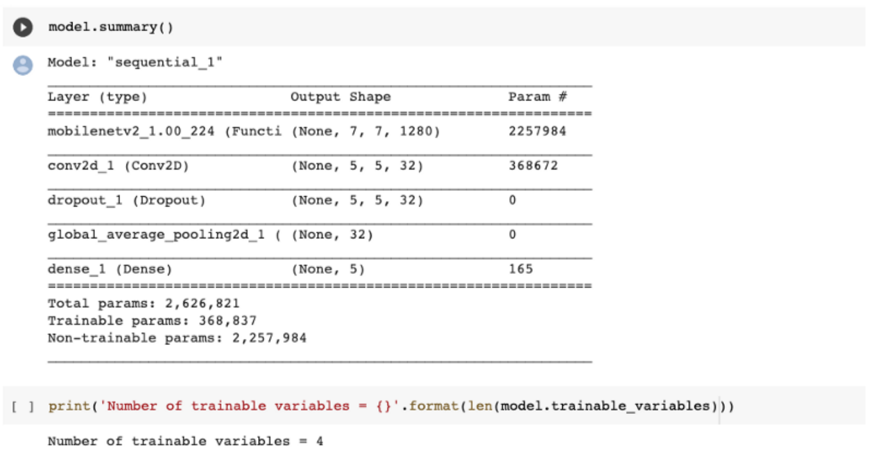
```
model = tf.keras.Sequential([
  base_model,
  tf.keras.layers.Conv2D(32, 3, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(5, activation='softmax')
```

Compile the model You must compile the model before training it. Since there are multiple classes, use a categorical cross-entropy loss

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss='categorical_crossentropy',

metrics=['accuracy'])

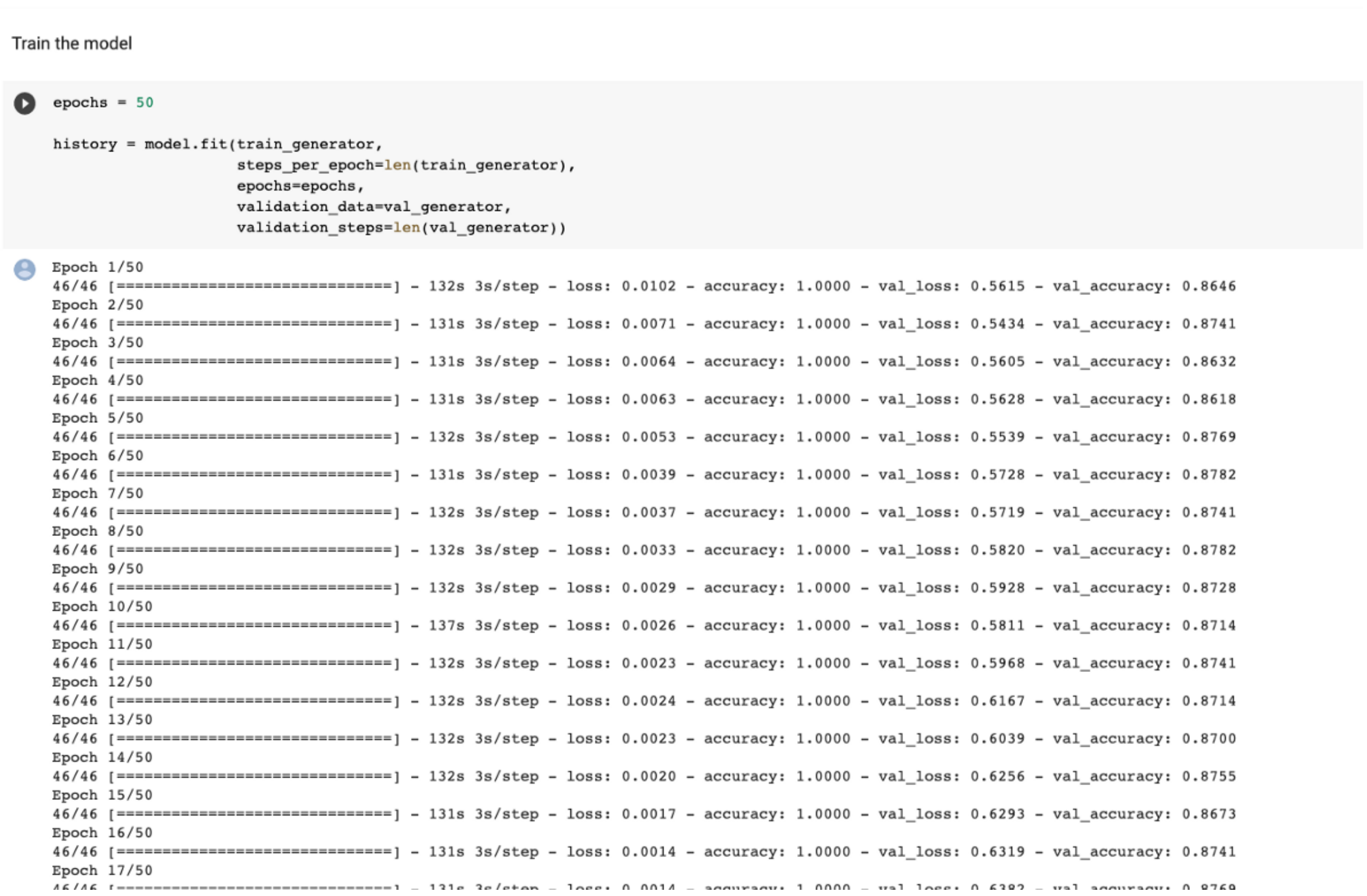


```
model.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 7, 7, 1280)        2257984
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 5, 5, 32)          368672
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 32)          0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 165
=================================================================
Total params: 2,626,821
Trainable params: 368,837
Non-trainable params: 2,257,984
_________________________________________________________________

[ ] print('Number of trainable variables = {}'.format(len(model.trainable_variables)))

Number of trainable variables = 4
```



Train the model

```
epochs = 50

history = model.fit(train_generator,
                    steps_per_epoch=len(train_generator),
                    epochs=epochs,
                    validation_data=val_generator,
                    validation_steps=len(val_generator))
```

```
Epoch 1/50
46/46 [==============================] - 132s 3s/step - loss: 0.0102 - accuracy: 1.0000 - val_loss: 0.5615 - val_accuracy: 0.8646
Epoch 2/50
46/46 [==============================] - 131s 3s/step - loss: 0.0071 - accuracy: 1.0000 - val_loss: 0.5434 - val_accuracy: 0.8741
Epoch 3/50
46/46 [==============================] - 131s 3s/step - loss: 0.0064 - accuracy: 1.0000 - val_loss: 0.5605 - val_accuracy: 0.8632
Epoch 4/50
46/46 [==============================] - 131s 3s/step - loss: 0.0063 - accuracy: 1.0000 - val_loss: 0.5628 - val_accuracy: 0.8618
Epoch 5/50
46/46 [==============================] - 132s 3s/step - loss: 0.0053 - accuracy: 1.0000 - val_loss: 0.5539 - val_accuracy: 0.8769
Epoch 6/50
46/46 [==============================] - 131s 3s/step - loss: 0.0039 - accuracy: 1.0000 - val_loss: 0.5728 - val_accuracy: 0.8782
Epoch 7/50
46/46 [==============================] - 132s 3s/step - loss: 0.0037 - accuracy: 1.0000 - val_loss: 0.5719 - val_accuracy: 0.8741
Epoch 8/50
46/46 [==============================] - 132s 3s/step - loss: 0.0033 - accuracy: 1.0000 - val_loss: 0.5820 - val_accuracy: 0.8782
Epoch 9/50
46/46 [==============================] - 132s 3s/step - loss: 0.0029 - accuracy: 1.0000 - val_loss: 0.5928 - val_accuracy: 0.8728
Epoch 10/50
46/46 [==============================] - 137s 3s/step - loss: 0.0026 - accuracy: 1.0000 - val_loss: 0.5811 - val_accuracy: 0.8714
Epoch 11/50
46/46 [==============================] - 132s 3s/step - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.5968 - val_accuracy: 0.8741
Epoch 12/50
46/46 [==============================] - 132s 3s/step - loss: 0.0024 - accuracy: 1.0000 - val_loss: 0.6167 - val_accuracy: 0.8714
Epoch 13/50
46/46 [==============================] - 132s 3s/step - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.6039 - val_accuracy: 0.8700
Epoch 14/50
46/46 [==============================] - 132s 3s/step - loss: 0.0020 - accuracy: 1.0000 - val_loss: 0.6256 - val_accuracy: 0.8755
Epoch 15/50
46/46 [==============================] - 131s 3s/step - loss: 0.0017 - accuracy: 1.0000 - val_loss: 0.6293 - val_accuracy: 0.8673
Epoch 16/50
46/46 [==============================] - 131s 3s/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.6319 - val_accuracy: 0.8741
Epoch 17/50
46/46 [==============================] - 131s 3s/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.6382 - val_accuracy: 0.8769
```

Learning curves

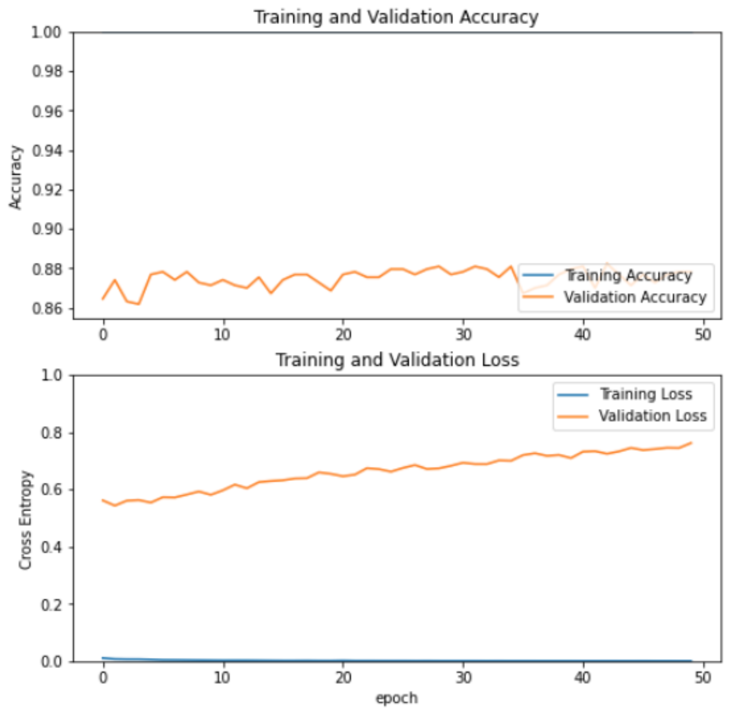
Let’s take a look at the learning curves of the training and validation accuracy/loss when using the MobileNet V2 base model as a fixed feature extractor.

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

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

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

plt.subplot(2, 1, 2)
```



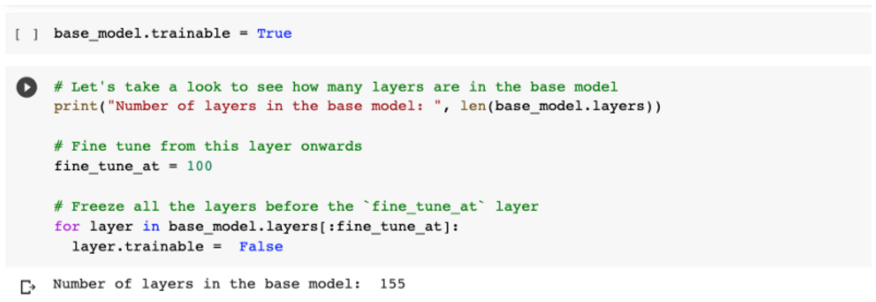
Fine-tuning

In our feature extraction experiment, you were only training a few layers on top of a MobileNet V2 base model. The weights of the pre-trained network were not updated during training.

One way to increase performance even further is to train (or “fine-tune”) the weights of the top layers of the pre-trained model alongside the training of the classifier you added. The training process will force the weights to be tuned from generic features maps to features associated specifically with our dataset.

Un-freeze the top layers of the model

All you need to do is unfreeze the base_model and set the bottom layers be un-trainable. Then, recompile the model (necessary for these changes to take effect), and resume training.



```
[ ] base_model.trainable = True
```

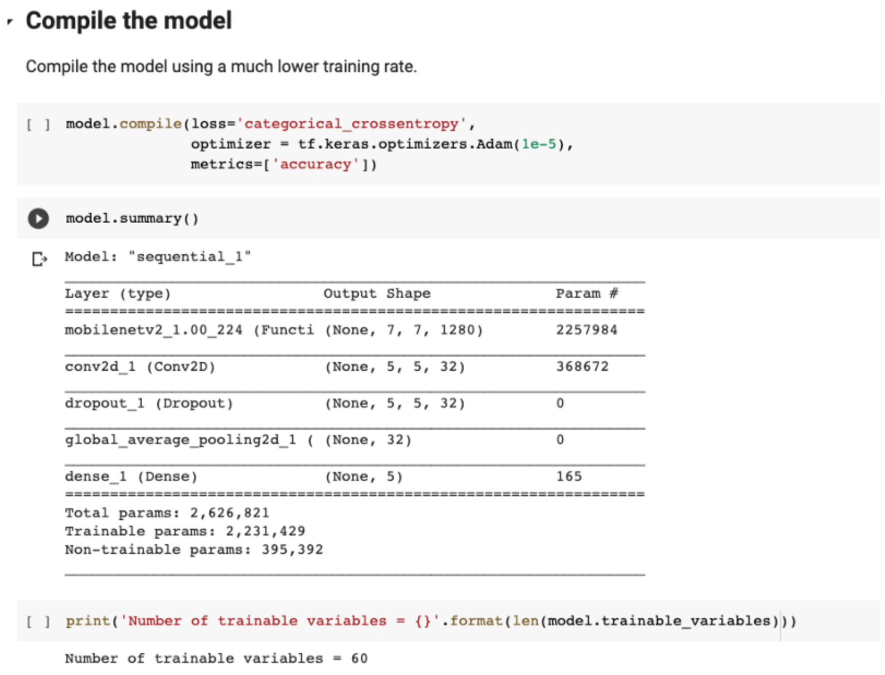
```
# Let's take a look to see how many layers are in the base model
print("Number of layers in the base model: ", len(base_model.layers))

# Fine tune from this layer onwards
fine_tune_at = 100

# Freeze all the layers before the `fine_tune_at` layer
for layer in base_model.layers[:fine_tune_at]:
  layer.trainable =  False
```

```
Number of layers in the base model:  155
```

Compilation of the Model



Compile the model

Compile the model using a much lower training rate.

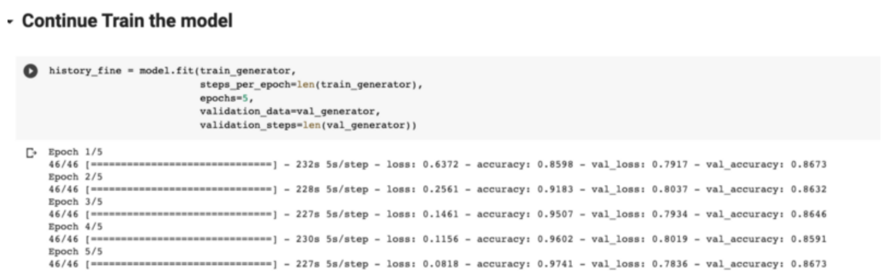
```
[ ] model.compile(loss='categorical_crossentropy',
              optimizer = tf.keras.optimizers.Adam(1e-5),
              metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 7, 7, 1280)        2257984
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 5, 5, 32)          368672
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 32)          0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 165
=================================================================
Total params: 2,626,821
Trainable params: 2,231,429
Non-trainable params: 395,392
_________________________________________________________________
```

```
[ ] print('Number of trainable variables = {}'.format(len(model.trainable_variables)))
```

```
Number of trainable variables = 60
```



Continue Train the model

```
history_fine = model.fit(train_generator,
                         steps_per_epoch=len(train_generator),
                         epochs=5,
                         validation_data=val_generator,
                         validation_steps=len(val_generator))
```

```
Epoch 1/5
46/46 [==============================] - 232s 5s/step - loss: 0.6372 - accuracy: 0.8598 - val_loss: 0.7917 - val_accuracy: 0.8673
Epoch 2/5
46/46 [==============================] - 228s 5s/step - loss: 0.2561 - accuracy: 0.9183 - val_loss: 0.8037 - val_accuracy: 0.8632
Epoch 3/5
46/46 [==============================] - 227s 5s/step - loss: 0.1461 - accuracy: 0.9507 - val_loss: 0.7934 - val_accuracy: 0.8646
Epoch 4/5
46/46 [==============================] - 230s 5s/step - loss: 0.1156 - accuracy: 0.9602 - val_loss: 0.8019 - val_accuracy: 0.8591
Epoch 5/5
46/46 [==============================] - 227s 5s/step - loss: 0.0818 - accuracy: 0.9741 - val_loss: 0.7836 - val_accuracy: 0.8673
```

Convert to TFLite

Saved the model using tf.saved_model.save and then convert the saved model to a tf lite compatible format.

Download the converted model and labels

saved_model_dir = ‘save/fine_tuning’

tf.saved_model.save(model, saved_model_dir)

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

tflite_model = converter.convert()

with open(‘model.tflite’, ‘wb’) as f:

f.write(tflite_model)

from google.colab import files

files.download(‘model.tflite’)

files.download(‘labels.txt’)

Let’s take a look at the learning curves of the training and validation accuracy/loss, when fine-tuning the last few layers of the MobileNet V2 base model and training the classifier on top of it. The validation loss is much higher than the training loss, so you may get some overfitting.

You may also get some overfitting as the new training set is relatively small and similar to the original MobileNet V2 datasets.

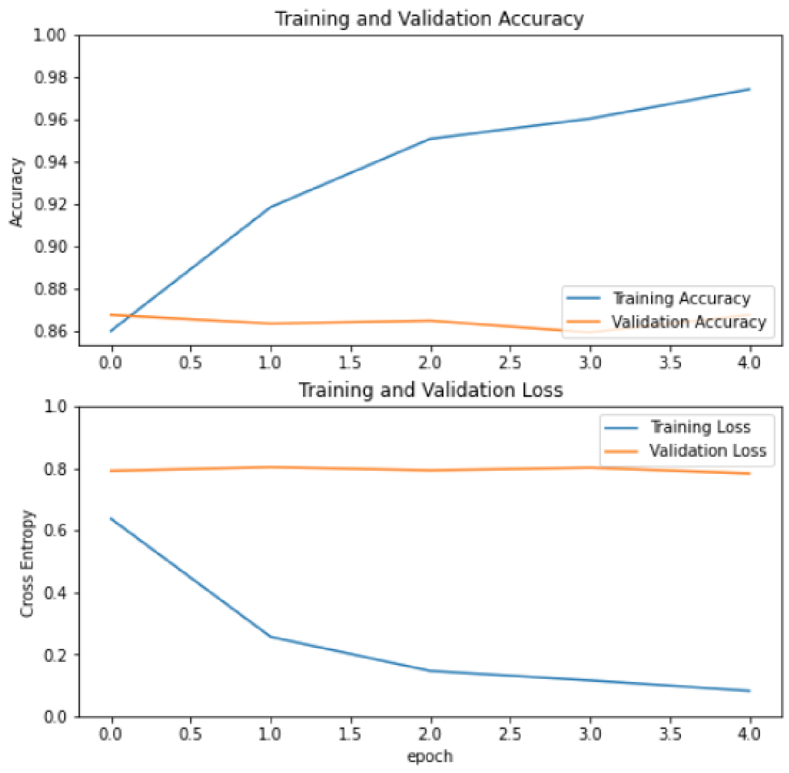
```python
acc = history_fine.history['accuracy']
val_acc = history_fine.history['val_accuracy']

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

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

plt.subplot(2, 1, 2)
```

Output graph of the above code :-



Summary:-

Using a pre-trained model for feature extraction: When working with a small dataset, it is common to take advantage of features learned by a model trained on a larger dataset in the same domain.

This is done by instantiating the pre-trained model and adding a fully-connected classifier on top. The pre-trained model is “frozen” and only the weights of the classifier get updated during training.

In this case, the convolutional base extracted all the features associated with each image and you just trained a classifier that determines the image class given that set of extracted features.

Fine-tuning a pre-trained model: To further improve performance, one might want to repurpose the top-level layers of the pre-trained models to the new dataset via fine-tuning. In this case, you tuned your weights such that your model learned high-level features specific to the dataset.

This technique is usually recommended when the training dataset is large and very similar to the original dataset that the pre-trained model was trained on.

Complete Github Code:-

HarmanBhutani/ML_projects

Permalink GitHub is home to over 50 million developers working together to host and review …

github.com