Chapter 05 – Deep Computer Vision Using CNNs

First, let us import a few common modules,

- ensure MatplotLib plots figures inline
- We also check that Python 3.5 or later is installed (although Python 2.x may work, Python 3 is strongly recommended),
- Scikit-Learn ≥0.20 and
- TensorFlow ≥2.0.

In [1]:

```
# Python ≥3.5 is required
import sys
import sklearn
import tensorflow as tf
from tensorflow import keras
import numpy as np
import os
import tensorflow_datasets as tfds

# to make this notebook's output reproducable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Pooling layer

Pretrained Models for Transfer Learning

If we want to build an image classifier but we do not have enough training data, then it is often a good idea to reuse the lower layers of a pretrained model.

- For example, let’s train a model to classify pictures of flowers, reusing a pretrained Xception model.
- First, let’s load the dataset using TensorFlow Datasets:

In [2]:

```
np.random.seed(42)
tf.random.set_seed(42)
```

In [3]:
```
import tensorflow_datasets as tfds #If "import tensorflow_datasets as tfds" d
```

In [4]:
```
# If the code below does not work, run: conda install -c conda-forge ipywidge
dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
```

```
Downloading and preparing dataset 218.21 MiB (download: 218.21 MiB, generat
ed: 221.83 MiB, total: 440.05 MiB) to ~/tensorflow_datasets/tf_flowers/3.0.
1...

Dl Completed...:   0%|          | 0/5 [00:00<?, ? file/s]

Dataset tf_flowers downloaded and prepared to ~/tensorflow_datasets/tf_flow
ers/3.0.1. Subsequent calls will reuse this data.
```

In [5]:
```
info.splits
```

Out[5]:
```
{'train': <SplitInfo num_examples=3670, num_shards=2>}
```

In [6]:
```
info.splits["train"]
```

Out[6]:
```
<SplitInfo num_examples=3670, num_shards=2>
```

In [7]:
```
class_names = info.features["label"].names
class_names
```

Out[7]:
```
['dandelion', 'daisy', 'tulips', 'sunflowers', 'roses']
```

In [8]:
```
n_classes = info.features["label"].num_classes
n_classes
```

Out[8]:
```
5
```

In [9]:
```
dataset_size = info.splits["train"].num_examples
dataset_size
```

Out[9]:
```
3670
```

Note that we can get information about the dataset by setting with_info=True. Here, we get the dataset size and the names of the classes.

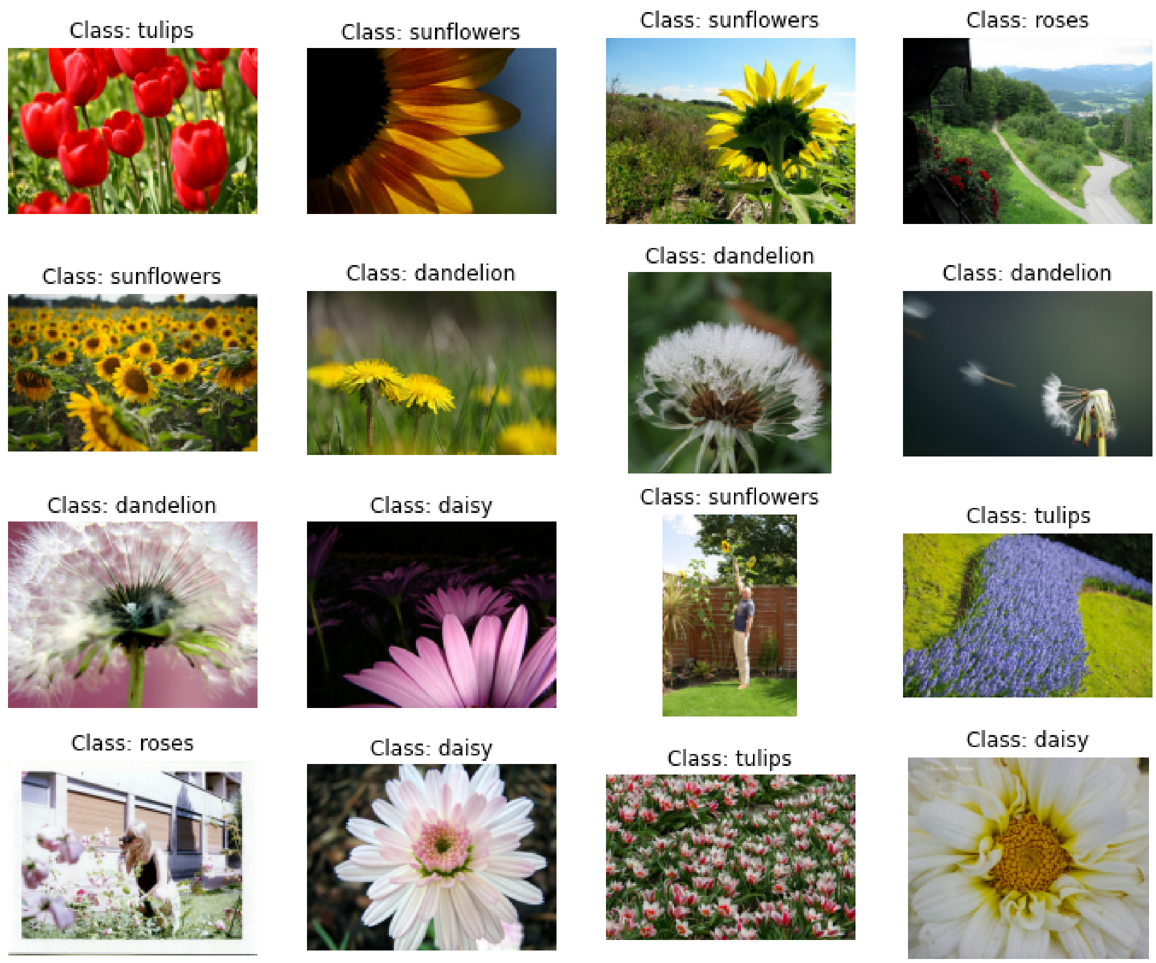
- Unfortunately, there is only a "train" dataset, no test set or validation set, so we need to split the training set.
- The TF Datasets project provides an API for this.
  - For example, let’s take the first 10% of the dataset for testing, the next 10% for validation, and the remaining 80% for training:

In [10]:
```
train_split,valid_split,test_split = tfds.Split.TRAIN.Split=["train[:80%]","t

test_set_raw = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set_raw = tfds.load("tf_flowers", split=valid_split, as_supervised=True
train_set_raw = tfds.load("tf_flowers", split=train_split, as_supervised=True
```

In [ ]:
```
plt.figure(figsize=(12, 10))
index = 0
for image, label in train_set_raw.take(16):
    index += 1
    plt.subplot(4, 4, index)
    plt.imshow(image)
    plt.title("Class: {}".format(class_names[label]))
    plt.axis("off")

plt.show()
```



Basic preprocessing: Next we must preprocess the images.

- The CNN expects 224 × 224 images, so we need to resize them.
- We also need to run the images through Xception's `preprocess_input()` function:

In [11]:
```python
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Slightly fancier preprocessing (but you could add much more data augmentation):

In [12]:
```python
def central_crop(image):
    shape = tf.shape(image)
    min_dim = tf.reduce_min([shape[0], shape[1]])
    top_crop = (shape[0] - min_dim) // 4
    bottom_crop = shape[0] - top_crop
    left_crop = (shape[1] - min_dim) // 4
    right_crop = shape[1] - left_crop
    return image[top_crop:bottom_crop, left_crop:right_crop]

def random_crop(image):
    shape = tf.shape(image)
    min_dim = tf.reduce_min([shape[0], shape[1]]) * 90 // 100
    return tf.image.random_crop(image, [min_dim, min_dim, 3])

def preprocess(image, label, randomize=False):
    if randomize:
        cropped_image = random_crop(image)
        cropped_image = tf.image.random_flip_left_right(cropped_image)
    else:
        cropped_image = central_crop(image)
    resized_image = tf.image.resize(cropped_image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Let us apply this preprocessing function to all three datasets, shuffle the training set, and add batching and prefetching to all the datasets:

In [13]:
```python
batch_size = 32
train_set = train_set_raw.shuffle(1000).repeat()
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
#train_set = train_set.map(partial(preprocess, randomize=True)).batch(batch_s
valid_set = valid_set_raw.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set_raw.map(preprocess).batch(batch_size).prefetch(1)
```

Next let us load an Xception model, pretrained on ImageNet.

- We exclude the top of the network by setting `include_top=False` :
  - this excludes the global average pooling layer and the dense output layer.
  - We then add our own global average pooling layer, based on the output of the base model, followed by a dense output layer with one unit per class, using the softmax activation function.

Finally, we create the Keras Model:

Model 1

In [14]:
```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.models.Model(inputs=base_model.input, outputs=output)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/xception/xception_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://storage.googleapis.com/tensorflow/keras-applications/xception/xception_weights_tf_dim_ordering_tf_kernels_notop.h5)
83683744/83683744 [==============================] - 4s 0us/step
```

In [15]:
```
for index, layer in enumerate(base_model.layers):
    print(index, layer.name)
```

```
0 input_1
1 block1_conv1
2 block1_conv1_bn
3 block1_conv1_act
4 block1_conv2
5 block1_conv2_bn
6 block1_conv2_act
7 block2_sepconv1
8 block2_sepconv1_bn
9 block2_sepconv2_act
10 block2_sepconv2
11 block2_sepconv2_bn
12 conv2d
13 block2_pool
14 batch_normalization
15 add
16 block3_sepconv1_act
17 block3_sepconv1
18 block3_sepconv1_bn
```

It is usually a good idea to freeze the weights of the pretrained layers, at least at the beginning of training:

In [16]:
```
for layer in base_model.layers:
    layer.trainable = False
```

Since our model uses the base model's layers directly, rather than the base_model object itself, setting `base_model.trainable=False` would have no effect.

Finally, we can compile the model and start training:

In [17]:
```
optimizer = keras.optimizers.SGD(learning_rate=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.80 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.10 * dataset_size / batch_size),
                    epochs=5)
```

```
Epoch 1/5
91/91 [==============================] - 24s 143ms/step - loss: 1.3638 - ac
curacy: 0.7957 - val_loss: 0.7792 - val_accuracy: 0.8750
Epoch 2/5
91/91 [==============================] - 12s 137ms/step - loss: 0.5643 - ac
curacy: 0.9011 - val_loss: 0.5810 - val_accuracy: 0.8835
Epoch 3/5
91/91 [==============================] - 13s 139ms/step - loss: 0.2466 - ac
curacy: 0.9378 - val_loss: 0.4684 - val_accuracy: 0.8835
Epoch 4/5
91/91 [==============================] - 14s 152ms/step - loss: 0.1237 - ac
curacy: 0.9605 - val_loss: 0.3940 - val_accuracy: 0.9062
Epoch 5/5
91/91 [==============================] - 13s 138ms/step - loss: 0.0870 - ac
curacy: 0.9742 - val_loss: 0.4342 - val_accuracy: 0.8977
```

After training the model for a few epochs, its validation accuracy should reach about 75–80% and stop making much progress.

- This means that the top layers are now pretty well trained, so we are ready to unfreeze all the layers (or we could try unfreezing just the top ones) and continue training (don’t forget to compile the model when you freeze or unfreeze layers).
- This time we use a much lower learning rate to avoid damaging the pretrained weights:

In [18]:
```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                                 nesterov=True, decay=0.001)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.80 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.10 * dataset_size / batch_size),
                    epochs=40)
```

```
Epoch 1/40
91/91 [==============================] - 60s 589ms/step - loss: 0.3362 -
accuracy: 0.8843 - val_loss: 0.5749 - val_accuracy: 0.8835
Epoch 2/40
91/91 [==============================] - 56s 620ms/step - loss: 0.0698 -
accuracy: 0.9791 - val_loss: 0.3809 - val_accuracy: 0.9091
Epoch 3/40
91/91 [==============================] - 58s 634ms/step - loss: 0.0252 -
accuracy: 0.9924 - val_loss: 0.2876 - val_accuracy: 0.9290
Epoch 4/40
91/91 [==============================] - 58s 632ms/step - loss: 0.0208 -
accuracy: 0.9938 - val_loss: 0.2657 - val_accuracy: 0.9318
Epoch 5/40
91/91 [==============================] - 57s 631ms/step - loss: 0.0072 -
accuracy: 0.9983 - val_loss: 0.2622 - val_accuracy: 0.9347
Epoch 6/40
91/91 [==============================] - 58s 637ms/step - loss: 0.0047 -
accuracy: 0.9986 - val_loss: 0.2580 - val_accuracy: 0.9432
Epoch 7/40
```

In [19]:
```
x1=history.history['val_accuracy']
from numpy import argmax
arr = np.array(x1)
max1=arr.argsort()[-3:][::-1]
max1=arr[max1]
max1
```

Out[19]:
```
array([0.95454544, 0.94886363, 0.94886363])
```

Some Data Augmentation function that could have been used;

Horizontal Flip Augmentation; Reversing the entire rows and columns of an image pixels in horizontally is called horizontal flip augmentation.

Vertical Flip Augmentation;Reversing the entire rows and columns of an image pixels in vertically is called Vertical flip augmentation.

Random Brightness Augmentation;In Random brightness the image brightness can be augmented to bright or dark based on the given brightness range. The brightness range which has less than 1.0 % darkens the image.

In [20]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

Model 2

In [21]:
```
base_model2 = keras.applications.MobileNet(weights="imagenet",
                                           include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model2.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model2 = keras.models.Model(inputs=base_model2.input, outputs=output)
```

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows` is n
ot in [128, 160, 192, 224]. Weights for input shape (224, 224) will be load
ed as the default.

Downloading data from https://storage.googleapis.com/tensorflow/keras-appli
cations/mobilenet/mobilenet_1_0_224_tf_no_top.h5 (https://storage.googleapi
s.com/tensorflow/keras-applications/mobilenet/mobilenet_1_0_224_tf_no_top.h
5)
17225924/17225924 [==============================] - 2s 0us/step
```

In [22]:
```
for index, layer in enumerate(base_model2.layers):
    print(index, layer.name)
```

```
0 input_2
1 conv1
2 conv1_bn
3 conv1_relu
4 conv_dw_1
5 conv_dw_1_bn
6 conv_dw_1_relu
7 conv_pw_1
8 conv_pw_1_bn
9 conv_pw_1_relu
10 conv_pad_2
11 conv_dw_2
12 conv_dw_2_bn
13 conv_dw_2_relu
14 conv_pw_2
15 conv_pw_2_bn
16 conv_pw_2_relu
17 conv_dw_3
18 conv_dw_3_bn
19 conv_dw_3_relu
20 conv_pw_3
21 conv_pw_3_bn
22 conv_pw_3_relu
23 conv_pad_4
24 conv_dw_4
25 conv_dw_4_bn
26 conv_dw_4_relu
27 conv_pw_4
28 conv_pw_4_bn
29 conv_pw_4_relu
30 conv_dw_5
31 conv_dw_5_bn
32 conv_dw_5_relu
33 conv_pw_5
34 conv_pw_5_bn
35 conv_pw_5_relu
36 conv_pad_6
37 conv_dw_6
38 conv_dw_6_bn
39 conv_dw_6_relu
40 conv_pw_6
41 conv_pw_6_bn
42 conv_pw_6_relu
43 conv_dw_7
44 conv_dw_7_bn
45 conv_dw_7_relu
46 conv_pw_7
47 conv_pw_7_bn
48 conv_pw_7_relu
49 conv_dw_8
50 conv_dw_8_bn
51 conv_dw_8_relu
52 conv_pw_8
```

```
53 conv_pw_8_bn
54 conv_pw_8_relu
55 conv_dw_9
56 conv_dw_9_bn
57 conv_dw_9_relu
58 conv_pw_9
59 conv_pw_9_bn
60 conv_pw_9_relu
61 conv_dw_10
62 conv_dw_10_bn
63 conv_dw_10_relu
64 conv_pw_10
65 conv_pw_10_bn
66 conv_pw_10_relu
67 conv_dw_11
68 conv_dw_11_bn
69 conv_dw_11_relu
70 conv_pw_11
71 conv_pw_11_bn
72 conv_pw_11_relu
73 conv_pad_12
74 conv_dw_12
75 conv_dw_12_bn
76 conv_dw_12_relu
77 conv_pw_12
78 conv_pw_12_bn
79 conv_pw_12_relu
80 conv_dw_13
81 conv_dw_13_bn
82 conv_dw_13_relu
83 conv_pw_13
84 conv_pw_13_bn
85 conv_pw_13_relu
```

In [23]:

```
for layer in base_model2.layers:
    layer.trainable = False
```

In [24]:

```
optimizer = keras.optimizers.SGD(learning_rate=0.2, momentum=0.9, decay=0.01)
model2.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history2 = model2.fit(train_set,
                      steps_per_epoch=int(0.80 * dataset_size / batch_size),
                      validation_data=valid_set,
                      validation_steps=int(0.10 * dataset_size / batch_size),
                      epochs=5)
```

```
Epoch 1/5
91/91 [==============================] - 5s 42ms/step - loss: 6.9283 - accu
racy: 0.7933 - val_loss: 3.7726 - val_accuracy: 0.8750
Epoch 2/5
91/91 [==============================] - 5s 57ms/step - loss: 1.7214 - accu
racy: 0.9245 - val_loss: 2.4706 - val_accuracy: 0.9034
Epoch 3/5
91/91 [==============================] - 3s 37ms/step - loss: 0.7917 - accu
racy: 0.9471 - val_loss: 2.3456 - val_accuracy: 0.9176
Epoch 4/5
91/91 [==============================] - 3s 37ms/step - loss: 0.5054 - accu
racy: 0.9578 - val_loss: 2.4356 - val_accuracy: 0.9119
Epoch 5/5
91/91 [==============================] - 3s 37ms/step - loss: 0.2994 - accu
racy: 0.9698 - val_loss: 2.3095 - val_accuracy: 0.9176
```

In [25]:
```
for layer in base_model2.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                                 nesterov=True, decay=0.001)
model2.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])

history2 = model2.fit(train_set,
                    steps_per_epoch=int(0.80 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.10 * dataset_size / batch_size),
                    epochs=40)
```

```
Epoch 1/40
91/91 [==============================] - 20s 196ms/step - loss: 8.5484 -
accuracy: 0.4107 - val_loss: 16.3212 - val_accuracy: 0.2415
Epoch 2/40
91/91 [==============================] - 18s 199ms/step - loss: 1.2861 -
accuracy: 0.5206 - val_loss: 11.3729 - val_accuracy: 0.1847
Epoch 3/40
91/91 [==============================] - 18s 195ms/step - loss: 1.0787 -
accuracy: 0.5852 - val_loss: 1.4921 - val_accuracy: 0.5426
Epoch 4/40
91/91 [==============================] - 18s 192ms/step - loss: 0.9631 -
accuracy: 0.6198 - val_loss: 1.1846 - val_accuracy: 0.6136
Epoch 5/40
91/91 [==============================] - 18s 193ms/step - loss: 0.8796 -
accuracy: 0.6693 - val_loss: 0.9556 - val_accuracy: 0.6165
Epoch 6/40
91/91 [==============================] - 18s 194ms/step - loss: 0.7602 -
accuracy: 0.7184 - val_loss: 0.9336 - val_accuracy: 0.6534
Epoch 7/40
```

In [26]:
```
x2=history2.history['val_accuracy']
arr2 = np.array(x2)
max2=arr2.argsort()[-3:][::-1]
max2=arr2[max2]
max2
```

Out[26]:
```
array([0.77556819, 0.77556819, 0.76420456])
```

In [ ]:

Model 3

In [27]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

In [28]:

```
base_model3 = keras.applications.NASNetMobile(weights="imagenet",
                                              include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model3.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model3 = keras.models.Model(inputs=base_model3.input, outputs=output)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-appli
cations/nasnet/NASNet-mobile-no-top.h5 (https://storage.googleapis.com/tens
orflow/keras-applications/nasnet/NASNet-mobile-no-top.h5)
19993432/19993432 [==============================] - 2s 0us/step
```

In [29]:

```
for index, layer in enumerate(base_model3.layers):
    print(index, layer.name)
```

```
0 input_3
1 stem_conv1
2 stem_bn1
3 activation
4 reduction_conv_1_stem_1
5 reduction_bn_1_stem_1
6 activation_1
7 activation_3
8 separable_conv_1_pad_reduction_left1_stem_1
9 separable_conv_1_pad_reduction_right1_stem_1
10 separable_conv_1_reduction_left1_stem_1
11 separable_conv_1_reduction_right1_stem_1
12 separable_conv_1_bn_reduction_left1_stem_1
13 separable_conv_1_bn_reduction_right1_stem_1
14 activation_2
15 activation_4
16 separable_conv_2_reduction_left1_stem_1
17 separable_conv_2_reduction_right1_stem_1
18 activation_5
```

In [30]:

```
for layer in base_model3.layers:
    layer.trainable = False
```

In [31]:

```
optimizer = keras.optimizers.SGD(learning_rate=0.2, momentum=0.9, decay=0.01)
model3.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history3 = model3.fit(train_set,
                    steps_per_epoch=int(0.80 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.10 * dataset_size / batch_size),
                    epochs=5)
```

```
Epoch 1/5
91/91 [==============================] - 22s 118ms/step - loss: 3.2047 - ac
curacy: 0.7479 - val_loss: 1.2936 - val_accuracy: 0.8778
Epoch 2/5
91/91 [==============================] - 7s 76ms/step - loss: 1.0210 - accu
racy: 0.8805 - val_loss: 0.8600 - val_accuracy: 0.8977
Epoch 3/5
91/91 [==============================] - 7s 82ms/step - loss: 0.5290 - accu
racy: 0.9172 - val_loss: 0.7015 - val_accuracy: 0.8977
Epoch 4/5
91/91 [==============================] - 7s 76ms/step - loss: 0.3270 - accu
racy: 0.9351 - val_loss: 0.7702 - val_accuracy: 0.8864
Epoch 5/5
91/91 [==============================] - 7s 77ms/step - loss: 0.2569 - accu
racy: 0.9475 - val_loss: 0.6365 - val_accuracy: 0.8807
```

In [32]:
```
for layer in base_model3.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                                 nesterov=True, decay=0.001)
model3.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])

history3 = model3.fit(train_set,
                    steps_per_epoch=int(0.80 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.10 * dataset_size / batch_size),
                    epochs=40)
```

```
Epoch 1/40
91/91 [==============================] - 59s 420ms/step - loss: 1.3368 -
accuracy: 0.5714 - val_loss: 13710745715867648.0000 - val_accuracy: 0.258
5
Epoch 2/40
91/91 [==============================] - 35s 382ms/step - loss: 1.0822 -
accuracy: 0.5611 - val_loss: 7688042315776.0000 - val_accuracy: 0.2756
Epoch 3/40
91/91 [==============================] - 34s 375ms/step - loss: 0.8938 -
accuracy: 0.6418 - val_loss: 80824270848.0000 - val_accuracy: 0.1392
Epoch 4/40
91/91 [==============================] - 34s 377ms/step - loss: 0.7498 -
accuracy: 0.7188 - val_loss: 3133452288.0000 - val_accuracy: 0.2358
Epoch 5/40
91/91 [==============================] - 35s 385ms/step - loss: 0.6373 -
accuracy: 0.7634 - val_loss: 387271776.0000 - val_accuracy: 0.2386
Epoch 6/40
91/91 [==============================] - 34s 379ms/step - loss: 0.5299 -
accuracy: 0.8036 - val_loss: 19313564.0000 - val_accuracy: 0.2415
```

In [33]:
```
x3=history3.history['val_accuracy']
arr3 = np.array(x3)
max3=arr3.argsort()[-3:][::-1]
max3=arr3[max3]
max3
```

Out[33]:
```
array([0.28125   , 0.27556819, 0.27556819])
```

Three best validation accuracies of the three models

In [39]:

```
import pandas as pd
best_model_max_val=[max1,max2,max3]
columns = ["MAX1", "MAX2", "MAX3"]
rows = ["Xception", "MobileNet", "NASNetMobile"]
data = best_model_max_val
df = pd.DataFrame(data=data, index=rows, columns=columns)
df
```

Out[39]:

	MAX1	MAX2	MAX3
Xception	0.954545	0.948864	0.948864
MobileNet	0.775568	0.775568	0.764205
NASNetMobile	0.281250	0.275568	0.275568

END