Final_projet_cousera

July 7, 2021

0.1 Final project rated by pairs

0.1.1 Made by Freedel ZINSOU PLY

0.1.2 Data description and main objective

The objective of this project is to evaluate the knowledge acquired during these courses on deep learning and reinforcement learning. My subject is on image recognition mainly on flower image recognition. The data are images of flowers. The objective is to implement a deep learning algorithm allowing to classify each flower image in the category which corresponds to it. The data contains 4323 flowers'image for five catgeories are : Daysly flowers, Tulip flowers, sun flowers, dandedlion flowers and rose flowers. My research in this area shows that the most suitable and most used model for this kind of data is the Convolutional Neuronal Network. So the main objectif of analysis is to implement several CNN model and keep which is best.

The data is available on kaggle https://www.kaggle.com/alxmamaev/flowers-recognition/data

[3]:
```
import keras
from keras.datasets import reuters
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import matplotlib.pyplot as plt
from keras import backend as K
from keras.models import Sequential
from keras.optimizers import Adam,SGD,Adagrad,Adadelta,RMSprop
from keras.utils import to_categorical
import tensorflow as tf
from pathlib import Path
import pandas as pd
```

[4]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import␣
↪accuracy_score,precision_score,recall_score,confusion_matrix,roc_curve,roc_auc_score
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelEncoder
```

[5]:
```
from tqdm import tqdm
import cv2
import numpy as np
import random as rn
import seaborn as sns
```

[6]:
```
import os
print(os.listdir((r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers')))
```

```
['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']
```

[7]:
```
X=[]
labels=[]
IMG_SIZE=150
daisy_flowers=r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers\daisy'
sun_flowers =r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers\sunflower'
tulip_flowers=r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers\tulip'
dandelion_flowers=r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers\dandelion'
rose_flowers=r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers\rose'
```

0.1.3 Data visualization and description

[8]:
```
input_path = Path('C:/Users/33758/Desktop/Freedel ZINSOU PLY/Coursera/flowers/')
folder_dir = input_path / 'flowers'
flower_types=os.listdir(r'C:\Users\33758\Desktop\Freedel ZINSOU␣
 ↪PLY\Coursera\flowers\flowers')
flowers = []
for species in flower_types:
    # Get all the file names
    all_flowers = os.listdir(folder_dir / species)
    # Add them to the list
    for flower in all_flowers:
        flowers.append((species, str(folder_dir /species) + '/' + flower))

# Build a dataframe
flowers = pd.DataFrame(data=flowers, columns=['category', 'image'], index=None)
flowers.head()
```

[8]:

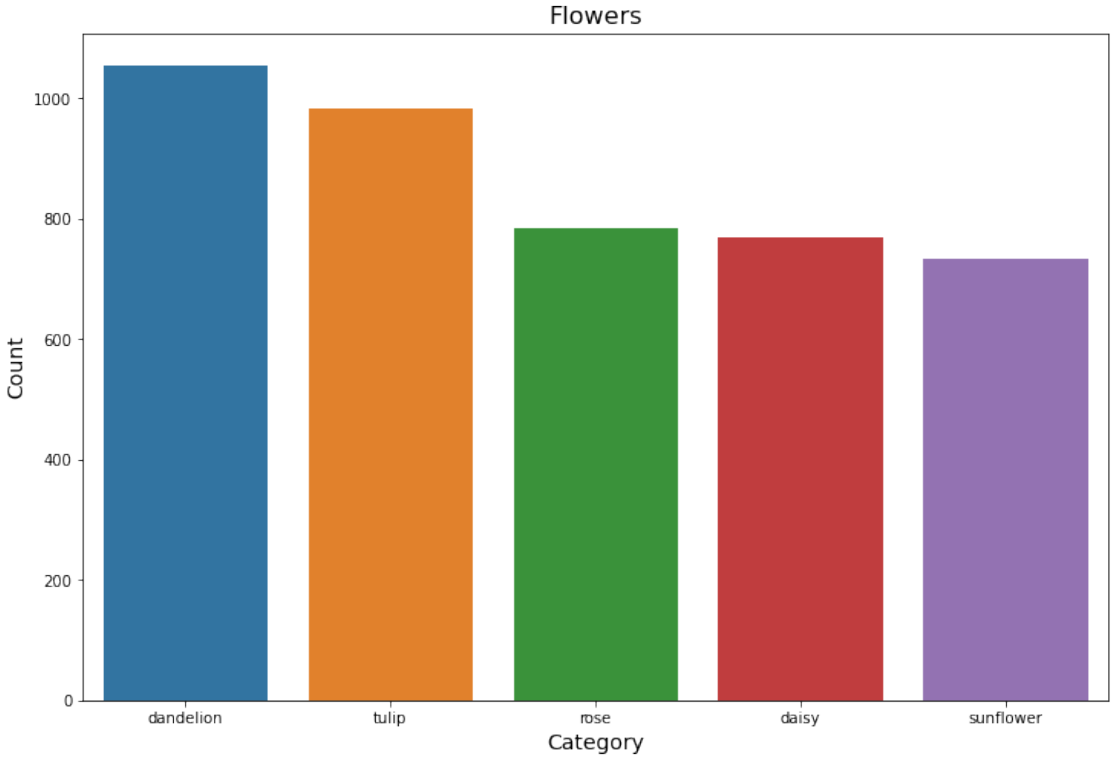
	category	image
0	daisy	C:\Users\33758\Desktop\Freedel ZINSOU PLY\Cour…
1	daisy	C:\Users\33758\Desktop\Freedel ZINSOU PLY\Cour…

```
2    daisy  C:\Users\33758\Desktop\Freedel ZINSOU PLY\Cour…
3    daisy  C:\Users\33758\Desktop\Freedel ZINSOU PLY\Cour…
4    daisy  C:\Users\33758\Desktop\Freedel ZINSOU PLY\Cour…
```

[9]:
```
print("Total number of flowers in the dataset: ", len(flowers))
fl_count = flowers['category'].value_counts()
print("Flowers in each category: ")
print(fl_count)
```

```
Total number of flowers in the dataset:  4326
Flowers in each category:
dandelion    1055
tulip         984
rose          784
daisy         769
sunflower     734
Name: category, dtype: int64
```

[30]:
```
# Let's do some visualization too
plt.figure(figsize=(12,8))
sns.barplot(x=fl_count.index, y=fl_count.values)
plt.title("Flowers", fontsize=16)
plt.xlabel("Category", fontsize=14)
plt.ylabel("Count", fontsize=14)
plt.show()
```



[11]:
```
# Build two functions to assign flowers image's categories with pictures
def assign_label(img,flower_type):
    return flower_type
```

[12]:
```
def train_data(flower_type,DIR):
    for img in tqdm(os.listdir(DIR)):
        label=assign_label(img,flower_type)
        path = os.path.join(DIR,img)
        img = cv2.imread(path,cv2.IMREAD_COLOR)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))

        X.append(np.array(img))
        labels.append(str(label))
```

[13]:
```
train_data('Daisy',daisy_flowers)
print(len(X))
```

```
100%|
  | 769/769 [00:06<00:00, 113.38it/s]
```

```
769
```

[14]:
```
train_data('Sun',sun_flowers)
print(len(X))
```

```
100%|
  | 734/734 [00:02<00:00, 350.18it/s]
```

```
1503
```

[15]:
```
train_data('Tulip',tulip_flowers)
print(len(X))
```

```
100%|
  | 984/984 [00:02<00:00, 399.20it/s]
```

```
2487
```

[16]:
```
train_data('Rose',rose_flowers)
print(len(X))
```

```
100%|
  | 784/784 [00:01<00:00, 424.77it/s]
```

```
3271
```

[17]:
```
train_data('Dandelon',dandelion_flowers)
print(len(X))
```

```
100%|
 | 1052/1055 [00:07<00:00, 139.26it/s]
```

```
---------------------------------------------------------------------------
error                                     Traceback (most recent call last)
<ipython-input-17-16715d1e34c0> in <module>
----> 1 train_data('Dandelon',dandelion_flowers)
      2 print(len(X))

<ipython-input-12-b4c73d818451> in train_data(flower_type, DIR)
      4         path = os.path.join(DIR,img)
      5         img = cv2.imread(path,cv2.IMREAD_COLOR)
----> 6         img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
      7
      8         X.append(np.array(img))

error: OpenCV(4.5.1) C:
 ↪\Users\appveyor\AppData\Local\Temp\1\pip-req-build-oduouqig\opencv\modules\im ;proc\src\resi
 ↪cpp:4051: error: (-215:Assertion failed) !ssize.empty() in function 'cv::
 ↪resize'
```

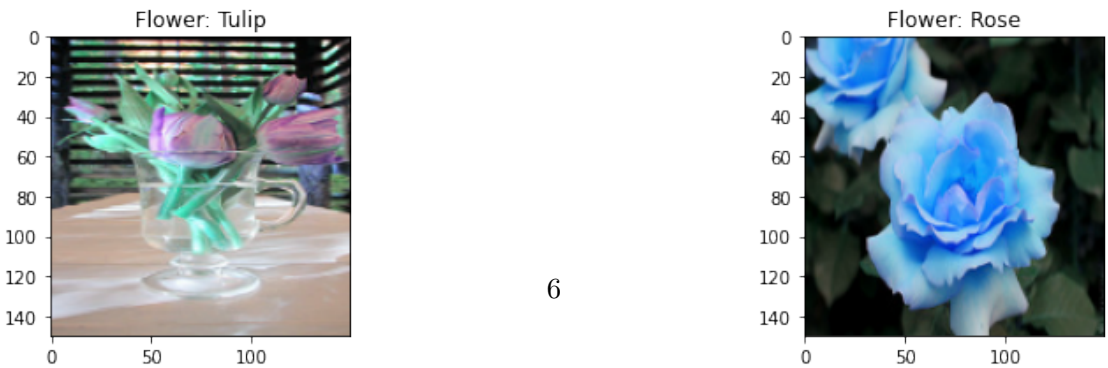
[18]:
```
#Random visualisation of flowers
fig,ax=plt.subplots(5,2)
fig.set_size_inches(15,15)
for i in range(5):
    for j in range (2):
        l=rn.randint(0,len(labels))
        ax[i,j].imshow(X[l])
        ax[i,j].set_title('Flower: '+labels[l])

plt.tight_layout()
```


Flower: Rose
Flower: Daisy


Flower: Tulip
Flower: Rose


Flower: Sun
Flower: Tulip


Flower: Dandelon
Flower: Dandelon


Flower: Tulip
Flower: Rose

0.2 CNN model

[19]:
```
#Generate training set and validation sets
le=LabelEncoder()
Y=le.fit_transform(labels)
Y=to_categorical(Y,5)
X=np.array(X)
X=X/255
len(X)
```

[19]: 4323

[20]:
```
x_train,x_test,y_train,y_test=train_test_split(X,Y,test_size=0.
↪25,random_state=42)
```

First model

[21]:
```
# Let's build a CNN using Keras' Sequential capabilities
model = Sequential()
# 5x5 convolution  and 32 filters
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',activation␣
↪='relu', input_shape = (150,150,3)))
model.add(MaxPooling2D(pool_size=(2,2)))

# 3x3 convolution  and 64 filters
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation␣
↪='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

#3x3 convolution, 2x2 strides  and 64 filters
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation␣
↪='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

#3x3 convolution, 2x2 strides  and 96 filters
model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation␣
↪='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

#Another 3x3 convolution, 2x2 strides  and 96 filters
model.add(Conv2D(filters = 96, kernel_size = (3,3),padding = 'Same',activation␣
↪='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

model.add(Flatten())
```

```
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(5, activation = "softmax"))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 32)      2432
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 75, 75, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 75, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 37, 37, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 37, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 18, 18, 96)        55392
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 9, 9, 96)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 9, 9, 96)          83040
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 96)          0
_________________________________________________________________
flatten (Flatten)            (None, 1536)              0
_________________________________________________________________
dense (Dense)                (None, 512)               786944
_________________________________________________________________
activation (Activation)      (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 2565
=================================================================
Total params: 985,797
Trainable params: 985,797
Non-trainable params: 0
_________________________________________________________________
```

Annealer In order to make the optimizer converge faster and closest to the global minimum of the loss function, i used an annealing method of the learning rate (LR).

[22]:
```
#Lets's train the model
#the loss function is categorical_crossentropy
```

```
#The optimizer function is Adam
batch_size=128
epochs=50
from keras.callbacks import ReduceLROnPlateau
red_lr= ReduceLROnPlateau(monitor='val_acc',patience=3,verbose=1,factor=0.1)
```

[24]:
```
datagen = ImageDataGenerator(
        rotation_range=20,
        zoom_range = 0.20,
        width_shift_range=0.3,
        height_shift_range=0.3,
        horizontal_flip=True,
        vertical_flip=True)

datagen.fit(x_train)
```

[25]:
```
model.compile(optimizer=Adam(lr=0.
↪001),loss='categorical_crossentropy',metrics=['accuracy'])
```

[26]:
```
Historys= model.fit_generator(datagen.flow(x_train,y_train,␣
↪batch_size=batch_size),
                              epochs = epochs, validation_data =␣
↪(x_test,y_test),
                              verbose = 1, steps_per_epoch=x_train.shape[0] //␣
↪batch_size)
```

```
C:\Users\33758\anaconda3\envs\R\lib\site-
packages\tensorflow\python\keras\engine\training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '

Epoch 1/50
25/25 [==============================] - 78s 3s/step - loss: 1.5239 - accuracy:
0.2806 - val_loss: 1.2316 - val_accuracy: 0.4459
Epoch 2/50
25/25 [==============================] - 74s 3s/step - loss: 1.2730 - accuracy:
0.4550 - val_loss: 1.1250 - val_accuracy: 0.5449
Epoch 3/50
25/25 [==============================] - 72s 3s/step - loss: 1.1526 - accuracy:
0.5193 - val_loss: 1.0623 - val_accuracy: 0.5837
Epoch 4/50
25/25 [==============================] - 73s 3s/step - loss: 1.0734 - accuracy:
0.5558 - val_loss: 1.0273 - val_accuracy: 0.5800
Epoch 5/50
25/25 [==============================] - 74s 3s/step - loss: 0.9948 - accuracy:
0.6055 - val_loss: 0.9577 - val_accuracy: 0.6253
Epoch 6/50
```

```
25/25 [==============================] - 72s 3s/step - loss: 0.9821 - accuracy:
0.6076 - val_loss: 0.9296 - val_accuracy: 0.6355
Epoch 7/50
25/25 [==============================] - 73s 3s/step - loss: 0.9615 - accuracy:
0.6282 - val_loss: 0.9250 - val_accuracy: 0.6096
Epoch 8/50
25/25 [==============================] - 73s 3s/step - loss: 0.9205 - accuracy:
0.6313 - val_loss: 1.0469 - val_accuracy: 0.6004
Epoch 9/50
25/25 [==============================] - 73s 3s/step - loss: 0.9531 - accuracy:
0.6316 - val_loss: 0.8692 - val_accuracy: 0.6549
Epoch 10/50
25/25 [==============================] - 73s 3s/step - loss: 0.8557 - accuracy:
0.6610 - val_loss: 0.8487 - val_accuracy: 0.6735
Epoch 11/50
25/25 [==============================] - 73s 3s/step - loss: 0.8432 - accuracy:
0.6634 - val_loss: 0.8391 - val_accuracy: 0.6679
Epoch 12/50
25/25 [==============================] - 73s 3s/step - loss: 0.8370 - accuracy:
0.6690 - val_loss: 0.8939 - val_accuracy: 0.6660
Epoch 13/50
25/25 [==============================] - 72s 3s/step - loss: 0.8304 - accuracy:
0.6807 - val_loss: 0.8240 - val_accuracy: 0.6827
Epoch 14/50
25/25 [==============================] - 73s 3s/step - loss: 0.7947 - accuracy:
0.6869 - val_loss: 0.8203 - val_accuracy: 0.6929
Epoch 15/50
25/25 [==============================] - 72s 3s/step - loss: 0.8041 - accuracy:
0.6943 - val_loss: 0.7776 - val_accuracy: 0.7225
Epoch 16/50
25/25 [==============================] - 73s 3s/step - loss: 0.7865 - accuracy:
0.7017 - val_loss: 0.9790 - val_accuracy: 0.6050
Epoch 17/50
25/25 [==============================] - 74s 3s/step - loss: 0.8282 - accuracy:
0.6676 - val_loss: 0.7961 - val_accuracy: 0.7197
Epoch 18/50
25/25 [==============================] - 72s 3s/step - loss: 0.7828 - accuracy:
0.7066 - val_loss: 0.8350 - val_accuracy: 0.6938
Epoch 19/50
25/25 [==============================] - 73s 3s/step - loss: 0.7357 - accuracy:
0.7334 - val_loss: 0.8258 - val_accuracy: 0.7040
Epoch 20/50
25/25 [==============================] - 79s 3s/step - loss: 0.7554 - accuracy:
0.7204 - val_loss: 0.7456 - val_accuracy: 0.7262
Epoch 21/50
25/25 [==============================] - 74s 3s/step - loss: 0.7493 - accuracy:
0.7091 - val_loss: 0.7341 - val_accuracy: 0.7225
Epoch 22/50
```

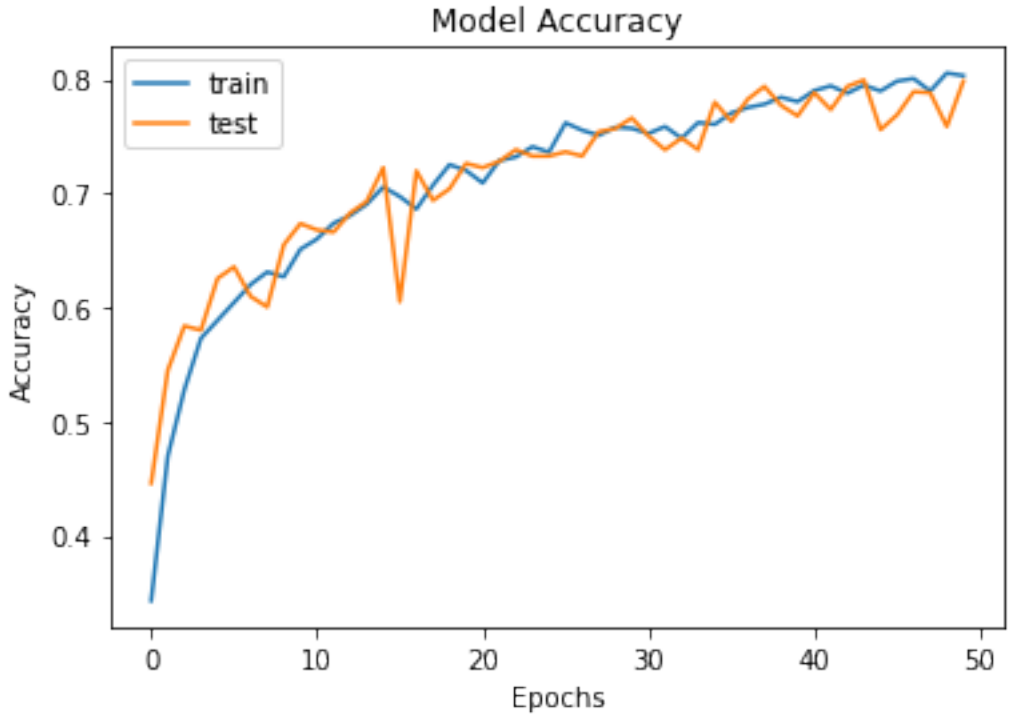
```
25/25 [==============================] - 73s 3s/step - loss: 0.7213 - accuracy:
0.7318 - val_loss: 0.7569 - val_accuracy: 0.7280
Epoch 23/50
25/25 [==============================] - 73s 3s/step - loss: 0.7049 - accuracy:
0.7351 - val_loss: 0.7224 - val_accuracy: 0.7382
Epoch 24/50
25/25 [==============================] - 73s 3s/step - loss: 0.7293 - accuracy:
0.7361 - val_loss: 0.7307 - val_accuracy: 0.7327
Epoch 25/50
25/25 [==============================] - 79s 3s/step - loss: 0.6947 - accuracy:
0.7371 - val_loss: 0.7193 - val_accuracy: 0.7327
Epoch 26/50
25/25 [==============================] - 74s 3s/step - loss: 0.6286 - accuracy:
0.7697 - val_loss: 0.6903 - val_accuracy: 0.7364
Epoch 27/50
25/25 [==============================] - 86s 3s/step - loss: 0.6502 - accuracy:
0.7579 - val_loss: 0.6717 - val_accuracy: 0.7327
Epoch 28/50
25/25 [==============================] - 86s 3s/step - loss: 0.6707 - accuracy:
0.7496 - val_loss: 0.6714 - val_accuracy: 0.7539
Epoch 29/50
25/25 [==============================] - 82s 3s/step - loss: 0.6548 - accuracy:
0.7558 - val_loss: 0.6604 - val_accuracy: 0.7567
Epoch 30/50
25/25 [==============================] - 74s 3s/step - loss: 0.6663 - accuracy:
0.7403 - val_loss: 0.6326 - val_accuracy: 0.7660
Epoch 31/50
25/25 [==============================] - 73s 3s/step - loss: 0.6575 - accuracy:
0.7520 - val_loss: 0.6645 - val_accuracy: 0.7502
Epoch 32/50
25/25 [==============================] - 72s 3s/step - loss: 0.6197 - accuracy:
0.7556 - val_loss: 0.7099 - val_accuracy: 0.7382
Epoch 33/50
25/25 [==============================] - 73s 3s/step - loss: 0.6224 - accuracy:
0.7644 - val_loss: 0.6747 - val_accuracy: 0.7484
Epoch 34/50
25/25 [==============================] - 77s 3s/step - loss: 0.6476 - accuracy:
0.7669 - val_loss: 0.7452 - val_accuracy: 0.7382
Epoch 35/50
25/25 [==============================] - 72s 3s/step - loss: 0.6271 - accuracy:
0.7641 - val_loss: 0.6115 - val_accuracy: 0.7798
Epoch 36/50
25/25 [==============================] - 73s 3s/step - loss: 0.6151 - accuracy:
0.7636 - val_loss: 0.6421 - val_accuracy: 0.7632
Epoch 37/50
25/25 [==============================] - 73s 3s/step - loss: 0.6067 - accuracy:
0.7720 - val_loss: 0.5908 - val_accuracy: 0.7826
Epoch 38/50
```

```
25/25 [==============================] - 73s 3s/step - loss: 0.5641 - accuracy:
0.7880 - val_loss: 0.5843 - val_accuracy: 0.7937
Epoch 39/50
25/25 [==============================] - 73s 3s/step - loss: 0.5579 - accuracy:
0.7894 - val_loss: 0.6196 - val_accuracy: 0.7771
Epoch 40/50
25/25 [==============================] - 72s 3s/step - loss: 0.5758 - accuracy:
0.7771 - val_loss: 0.6173 - val_accuracy: 0.7678
Epoch 41/50
25/25 [==============================] - 73s 3s/step - loss: 0.5515 - accuracy:
0.7954 - val_loss: 0.6040 - val_accuracy: 0.7882
Epoch 42/50
25/25 [==============================] - 72s 3s/step - loss: 0.5658 - accuracy:
0.7913 - val_loss: 0.6465 - val_accuracy: 0.7734
Epoch 43/50
25/25 [==============================] - 72s 3s/step - loss: 0.5455 - accuracy:
0.7942 - val_loss: 0.5720 - val_accuracy: 0.7937
Epoch 44/50
25/25 [==============================] - 72s 3s/step - loss: 0.5474 - accuracy:
0.7945 - val_loss: 0.5511 - val_accuracy: 0.7993
Epoch 45/50
25/25 [==============================] - 72s 3s/step - loss: 0.5569 - accuracy:
0.7918 - val_loss: 0.6691 - val_accuracy: 0.7558
Epoch 46/50
25/25 [==============================] - 72s 3s/step - loss: 0.5335 - accuracy:
0.8007 - val_loss: 0.6419 - val_accuracy: 0.7687
Epoch 47/50
25/25 [==============================] - 73s 3s/step - loss: 0.5208 - accuracy:
0.8093 - val_loss: 0.5886 - val_accuracy: 0.7891
Epoch 48/50
25/25 [==============================] - 73s 3s/step - loss: 0.5532 - accuracy:
0.7836 - val_loss: 0.5992 - val_accuracy: 0.7882
Epoch 49/50
25/25 [==============================] - 72s 3s/step - loss: 0.5043 - accuracy:
0.8106 - val_loss: 0.6982 - val_accuracy: 0.7586
Epoch 50/50
25/25 [==============================] - 72s 3s/step - loss: 0.5306 - accuracy:
0.8052 - val_loss: 0.5862 - val_accuracy: 0.7983
```

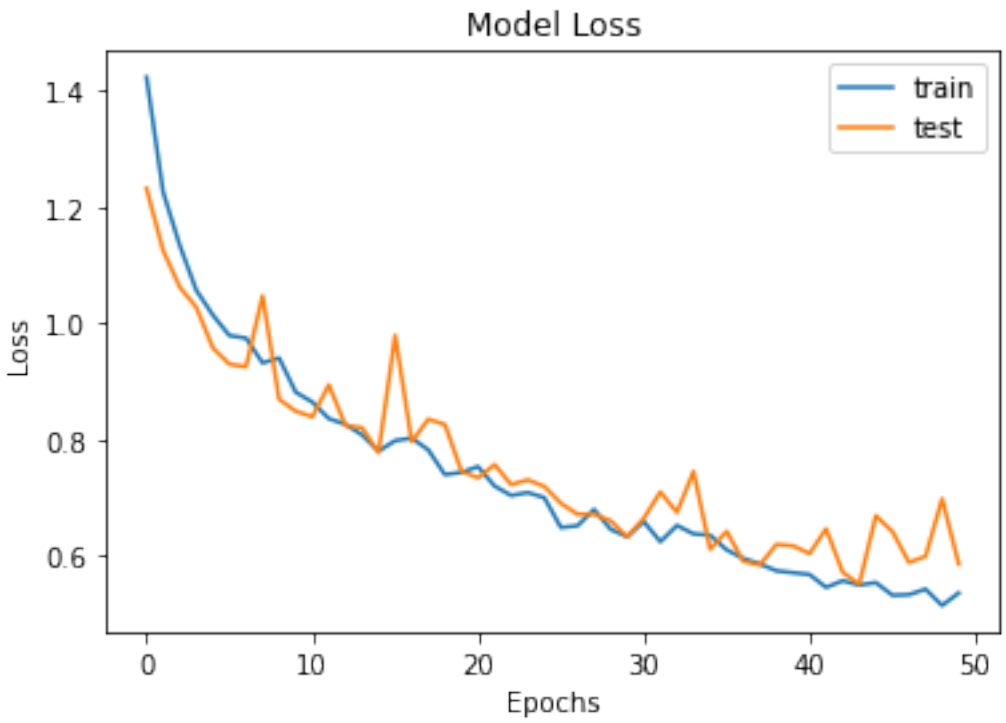
0.2.1 Performance of model

[28]:
```
#Let's make a graph that show the acuraccy  for both dataset
plt.plot(Historys.history['accuracy'])
plt.plot(Historys.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
```

```
plt.legend(['train', 'test'])
plt.show()
```



[29]:
```
#Let's make a graph that show the loss function for both dataset
plt.plot(Historys.history['loss'])
plt.plot(Historys.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['train', 'test'])
plt.show()
```



0.2.2 VGG model

[51]:
```
from keras.applications import VGG16
```

[52]:
```
Base_model = VGG16(include_top= False,␣
 ↪weights='imagenet',input_shape=(150,150,3), pooling='avg')
Base_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 150, 150, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
```

```
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
_________________________________________________________________
global_average_pooling2d_2 ( (None, 512)               0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

[53]:
```
model_VGG = Sequential()
model_VGG.add(Base_model)
model_VGG.add(Dense(256,activation='relu'))
# adding prediction(softmax) layer
model_VGG.add(Dense(5,activation="softmax"))
```

[54]:
```
Base_model.trainable = False
```

[55]:
```
red_lr=ReduceLROnPlateau(monitor='val_acc', factor=0.1, epsilon=0.0001,␣
↪patience=2, verbose=1)
```

```
WARNING:tensorflow:`epsilon` argument is deprecated and will be removed, use
`min_delta` instead.
```

[56]:
```
datagen = ImageDataGenerator(
        rotation_range=20,
        zoom_range = 0.20,
        width_shift_range=0.3,
        height_shift_range=0.3,
        horizontal_flip=True,
        vertical_flip=True)

datagen.fit(x_train)
```

[57]:
```
model_VGG.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 512)               14714688
_________________________________________________________________
dense_4 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_5 (Dense)              (None, 5)                 1285
=================================================================
Total params: 14,847,301
Trainable params: 132,613
Non-trainable params: 14,714,688
_________________________________________________________________
```

[58]:
```
model_VGG.compile(optimizer=Adam(lr = 1e-4), loss= 'categorical_crossentropy',␣
↪metrics=['accuracy'])
```

[59]:
```
batch_size=64
History = model_VGG.fit_generator(datagen.flow(x_train,y_train,␣
↪batch_size=batch_size),
                                  epochs = 50, validation_data = (x_test,y_test),
                                  verbose = 1, steps_per_epoch=x_train.shape[0] //␣
↪batch_size)
```

```
C:\Users\33758\anaconda3\envs\R\lib\site-
packages\tensorflow\python\keras\engine\training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '

Epoch 1/50
50/50 [==============================] - 319s 6s/step - loss: 1.6015 - accuracy:
0.2607 - val_loss: 1.4608 - val_accuracy: 0.4098
Epoch 2/50
50/50 [==============================] - 339s 7s/step - loss: 1.4061 - accuracy:
0.4645 - val_loss: 1.3183 - val_accuracy: 0.5338
```

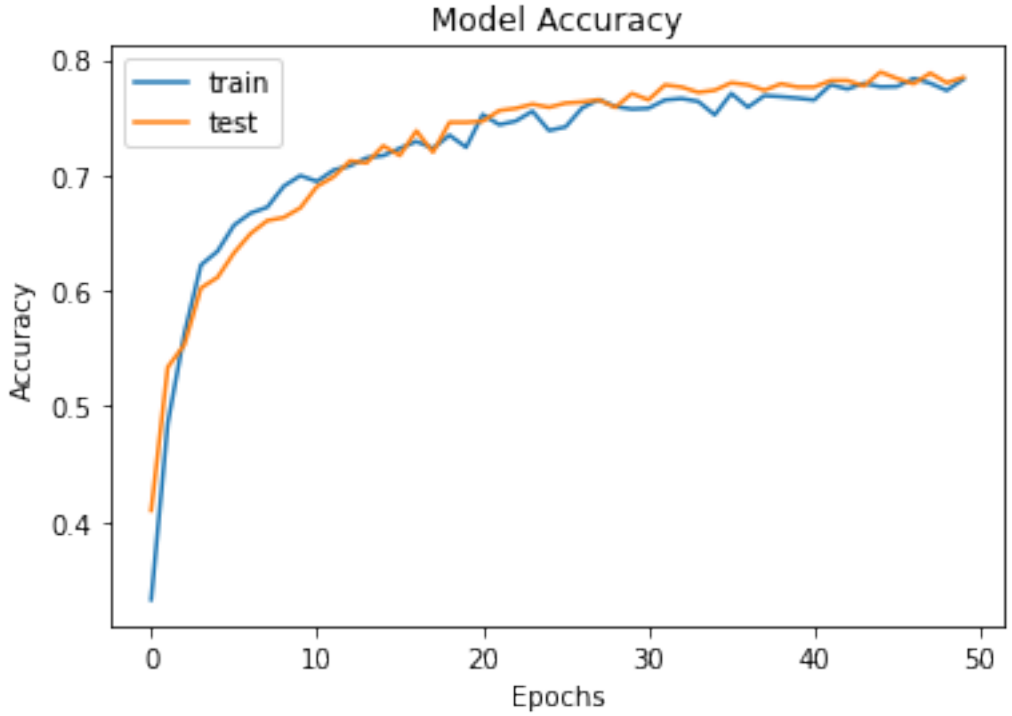
```
Epoch 3/50
50/50 [==============================] - 329s 7s/step - loss: 1.2799 - accuracy:
0.5560 - val_loss: 1.2171 - val_accuracy: 0.5532
Epoch 4/50
50/50 [==============================] - 327s 7s/step - loss: 1.1748 - accuracy:
0.6142 - val_loss: 1.1102 - val_accuracy: 0.6022
Epoch 5/50
50/50 [==============================] - 329s 7s/step - loss: 1.0858 - accuracy:
0.6319 - val_loss: 1.0437 - val_accuracy: 0.6115
Epoch 6/50
50/50 [==============================] - 315s 6s/step - loss: 1.0177 - accuracy:
0.6531 - val_loss: 0.9877 - val_accuracy: 0.6327
Epoch 7/50
50/50 [==============================] - 308s 6s/step - loss: 0.9685 - accuracy:
0.6662 - val_loss: 0.9415 - val_accuracy: 0.6494
Epoch 8/50
50/50 [==============================] - 306s 6s/step - loss: 0.9394 - accuracy:
0.6842 - val_loss: 0.9000 - val_accuracy: 0.6605
Epoch 9/50
50/50 [==============================] - 313s 6s/step - loss: 0.8944 - accuracy:
0.6914 - val_loss: 0.8804 - val_accuracy: 0.6633
Epoch 10/50
50/50 [==============================] - 306s 6s/step - loss: 0.8849 - accuracy:
0.6861 - val_loss: 0.8611 - val_accuracy: 0.6716
Epoch 11/50
50/50 [==============================] - 305s 6s/step - loss: 0.8542 - accuracy:
0.7002 - val_loss: 0.8314 - val_accuracy: 0.6901
Epoch 12/50
50/50 [==============================] - 325s 7s/step - loss: 0.8305 - accuracy:
0.6964 - val_loss: 0.8125 - val_accuracy: 0.6984
Epoch 13/50
50/50 [==============================] - 326s 7s/step - loss: 0.7864 - accuracy:
0.7083 - val_loss: 0.7874 - val_accuracy: 0.7123
Epoch 14/50
50/50 [==============================] - 348s 7s/step - loss: 0.8125 - accuracy:
0.7052 - val_loss: 0.7810 - val_accuracy: 0.7105
Epoch 15/50
50/50 [==============================] - 325s 7s/step - loss: 0.7955 - accuracy:
0.7074 - val_loss: 0.7553 - val_accuracy: 0.7253
Epoch 16/50
50/50 [==============================] - 316s 6s/step - loss: 0.7606 - accuracy:
0.7226 - val_loss: 0.7566 - val_accuracy: 0.7169
Epoch 17/50
50/50 [==============================] - 317s 6s/step - loss: 0.7501 - accuracy:
0.7354 - val_loss: 0.7337 - val_accuracy: 0.7382
Epoch 18/50
50/50 [==============================] - 323s 6s/step - loss: 0.7446 - accuracy:
0.7331 - val_loss: 0.7436 - val_accuracy: 0.7197
```

```
Epoch 19/50
50/50 [==============================] - 320s 6s/step - loss: 0.7216 - accuracy:
0.7420 - val_loss: 0.7116 - val_accuracy: 0.7456
Epoch 20/50
50/50 [==============================] - 312s 6s/step - loss: 0.7692 - accuracy:
0.7072 - val_loss: 0.7067 - val_accuracy: 0.7456
Epoch 21/50
50/50 [==============================] - 314s 6s/step - loss: 0.7160 - accuracy:
0.7616 - val_loss: 0.6993 - val_accuracy: 0.7465
Epoch 22/50
50/50 [==============================] - 337s 7s/step - loss: 0.6929 - accuracy:
0.7480 - val_loss: 0.6894 - val_accuracy: 0.7558
Epoch 23/50
50/50 [==============================] - 314s 6s/step - loss: 0.7275 - accuracy:
0.7365 - val_loss: 0.6834 - val_accuracy: 0.7576
Epoch 24/50
50/50 [==============================] - 339s 7s/step - loss: 0.6870 - accuracy:
0.7519 - val_loss: 0.6737 - val_accuracy: 0.7613
Epoch 25/50
50/50 [==============================] - 313s 6s/step - loss: 0.6981 - accuracy:
0.7364 - val_loss: 0.6681 - val_accuracy: 0.7586
Epoch 26/50
50/50 [==============================] - 315s 6s/step - loss: 0.7092 - accuracy:
0.7336 - val_loss: 0.6692 - val_accuracy: 0.7623
Epoch 27/50
50/50 [==============================] - 330s 7s/step - loss: 0.6961 - accuracy:
0.7475 - val_loss: 0.6613 - val_accuracy: 0.7632
Epoch 28/50
50/50 [==============================] - 313s 6s/step - loss: 0.6667 - accuracy:
0.7610 - val_loss: 0.6598 - val_accuracy: 0.7650
Epoch 29/50
50/50 [==============================] - 608s 12s/step - loss: 0.6560 -
accuracy: 0.7658 - val_loss: 0.6558 - val_accuracy: 0.7586
Epoch 30/50
50/50 [==============================] - 319s 6s/step - loss: 0.6762 - accuracy:
0.7528 - val_loss: 0.6458 - val_accuracy: 0.7706
Epoch 31/50
50/50 [==============================] - 336s 7s/step - loss: 0.6518 - accuracy:
0.7649 - val_loss: 0.6594 - val_accuracy: 0.7650
Epoch 32/50
50/50 [==============================] - 317s 6s/step - loss: 0.6443 - accuracy:
0.7718 - val_loss: 0.6341 - val_accuracy: 0.7780
Epoch 33/50
50/50 [==============================] - 315s 6s/step - loss: 0.6353 - accuracy:
0.7698 - val_loss: 0.6311 - val_accuracy: 0.7761
Epoch 34/50
50/50 [==============================] - 750s 15s/step - loss: 0.6503 -
accuracy: 0.7580 - val_loss: 0.6414 - val_accuracy: 0.7715
```

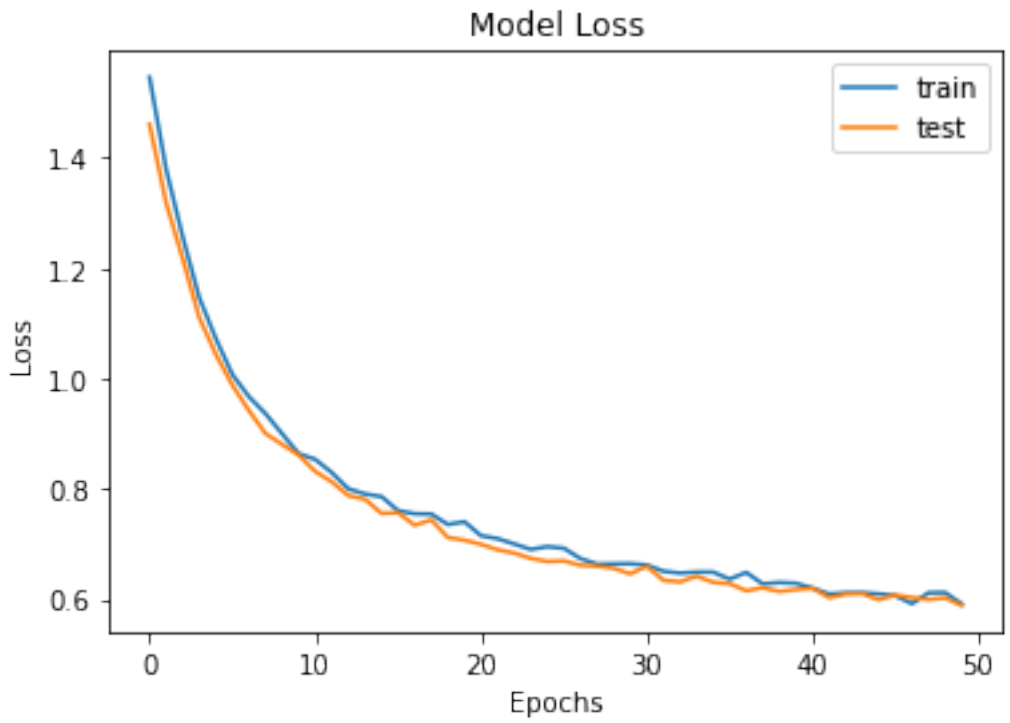
```
Epoch 35/50
50/50 [==============================] - 322s 6s/step - loss: 0.6681 - accuracy:
0.7481 - val_loss: 0.6305 - val_accuracy: 0.7734
Epoch 36/50
50/50 [==============================] - 324s 7s/step - loss: 0.6439 - accuracy:
0.7642 - val_loss: 0.6279 - val_accuracy: 0.7798
Epoch 37/50
50/50 [==============================] - 371s 7s/step - loss: 0.6331 - accuracy:
0.7678 - val_loss: 0.6151 - val_accuracy: 0.7780
Epoch 38/50
50/50 [==============================] - 351s 7s/step - loss: 0.6376 - accuracy:
0.7644 - val_loss: 0.6206 - val_accuracy: 0.7734
Epoch 39/50
50/50 [==============================] - 324s 6s/step - loss: 0.6595 - accuracy:
0.7507 - val_loss: 0.6137 - val_accuracy: 0.7789
Epoch 40/50
50/50 [==============================] - 320s 6s/step - loss: 0.6342 - accuracy:
0.7672 - val_loss: 0.6172 - val_accuracy: 0.7761
Epoch 41/50
50/50 [==============================] - 369s 7s/step - loss: 0.6081 - accuracy:
0.7751 - val_loss: 0.6196 - val_accuracy: 0.7761
Epoch 42/50
50/50 [==============================] - 319s 6s/step - loss: 0.5934 - accuracy:
0.7838 - val_loss: 0.6028 - val_accuracy: 0.7817
Epoch 43/50
50/50 [==============================] - 317s 6s/step - loss: 0.6061 - accuracy:
0.7761 - val_loss: 0.6088 - val_accuracy: 0.7817
Epoch 44/50
50/50 [==============================] - 317s 6s/step - loss: 0.6178 - accuracy:
0.7659 - val_loss: 0.6097 - val_accuracy: 0.7771
Epoch 45/50
50/50 [==============================] - 318s 6s/step - loss: 0.6050 - accuracy:
0.7721 - val_loss: 0.5990 - val_accuracy: 0.7891
Epoch 46/50
50/50 [==============================] - 318s 6s/step - loss: 0.5817 - accuracy:
0.7927 - val_loss: 0.6072 - val_accuracy: 0.7835
Epoch 47/50
50/50 [==============================] - 317s 6s/step - loss: 0.5968 - accuracy:
0.7783 - val_loss: 0.6026 - val_accuracy: 0.7789
Epoch 48/50
50/50 [==============================] - 315s 6s/step - loss: 0.6267 - accuracy:
0.7718 - val_loss: 0.5992 - val_accuracy: 0.7882
Epoch 49/50
50/50 [==============================] - 320s 6s/step - loss: 0.5950 - accuracy:
0.7778 - val_loss: 0.6023 - val_accuracy: 0.7798
Epoch 50/50
50/50 [==============================] - 314s 6s/step - loss: 0.5794 - accuracy:
0.7867 - val_loss: 0.5884 - val_accuracy: 0.7845
```

0.3 Performance of model VGG

[60]:
```
#Let's make a graph that show the acuraccy  for both dataset
plt.plot(History.history['accuracy'])
plt.plot(History.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['train', 'test'])
plt.show()
```



[61]:
```
#Let's make a graph that show the loss function for both dataset
plt.plot(History.history['loss'])
plt.plot(History.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['train', 'test'])
plt.show()
```



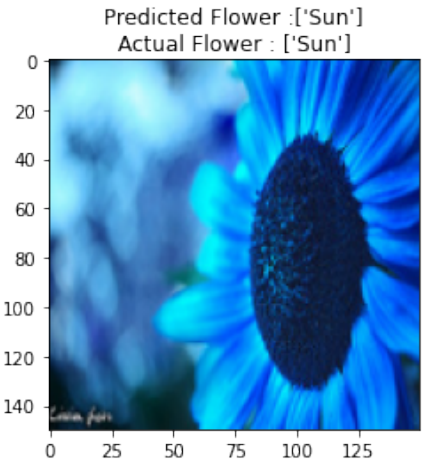
By comparing the two models we realize the result are almost the same but the best one is the simple model CNN (79 %)

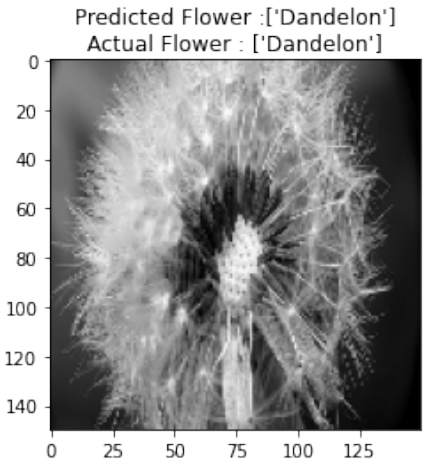
[70]:
```
pred=model.predict(x_test)
pred_digits=np.argmax(pred,axis=1)
```

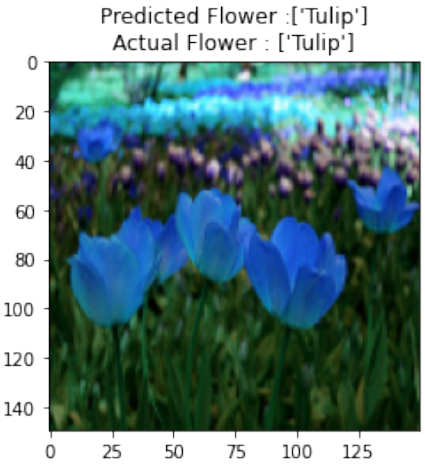
[71]:
```
good_pred=[]
bad_pred=[]

for i in range(len(y_test)):
    if(np.argmax(y_test[i])==pred_digits[i]):
        good_pred.append(i)
    if(len(good_pred)==10):
        break

i=0
for i in range(len(y_test)):
    if(not np.argmax(y_test[i])==pred_digits[i]):
        bad_pred.append(i)
    if(len(bad_pred)==10):
        break
```

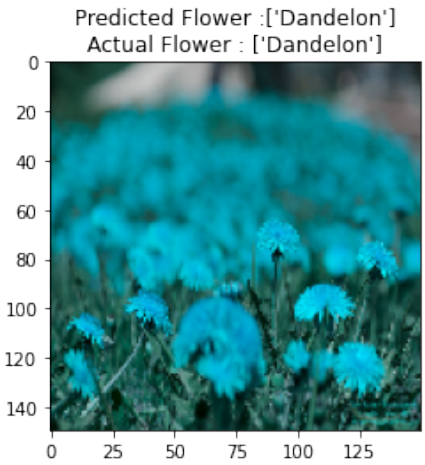
0.3.1 Visualization of good prediction

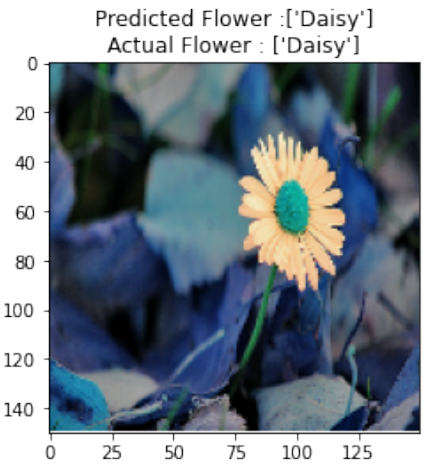
[72]:
```
import warnings
warnings.filterwarnings('always')
warnings.filterwarnings('ignore')

count=0
fig,ax=plt.subplots(4,2)
fig.set_size_inches(15,15)
for i in range (4):
    for j in range (2):
        ax[i,j].imshow(x_test[good_pred[count]])
        ax[i,j].set_title("Predicted Flower :"+str(le.
 ↪inverse_transform([pred_digits[good_pred[count]]]))+"\n"+"Actual Flower :␣
 ↪"+str(le.inverse_transform([np.argmax(y_test[good_pred[count]])])))
        plt.tight_layout()
        count+=1
```

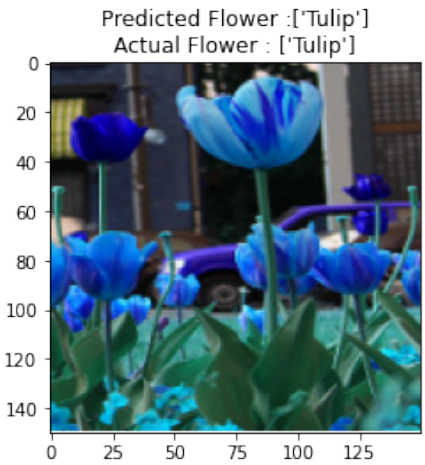

Predicted Flower :['Sun']
Actual Flower : ['Sun']

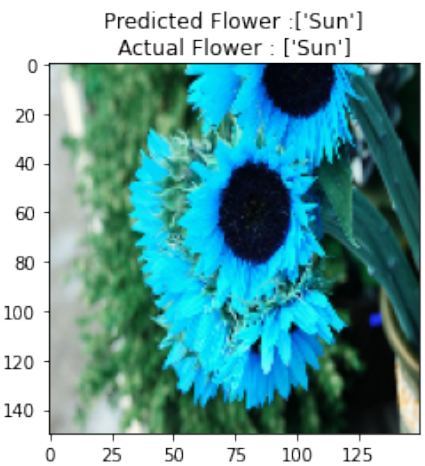

Predicted Flower :['Dandelon']
Actual Flower : ['Dandelon']

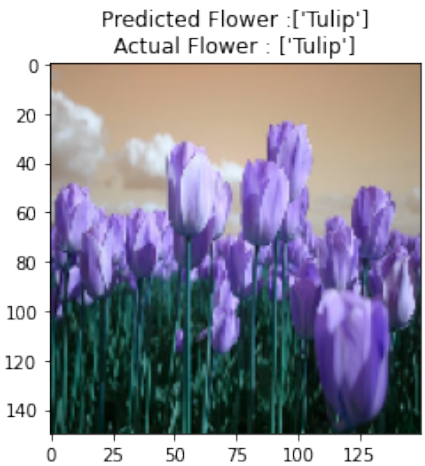

Predicted Flower :['Tulip']
Actual Flower : ['Tulip']


Predicted Flower :['Dandelon']
Actual Flower : ['Dandelon']


Predicted Flower :['Daisy']
Actual Flower : ['Daisy']

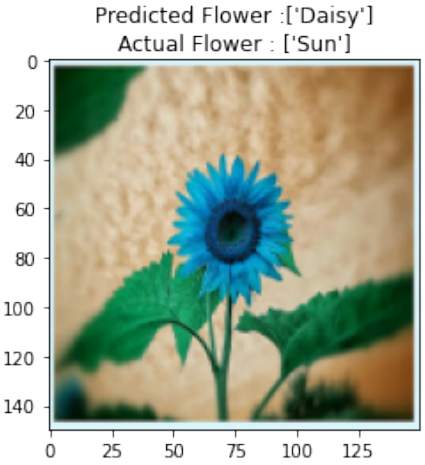

Predicted Flower :['Tulip']
Actual Flower : ['Tulip']

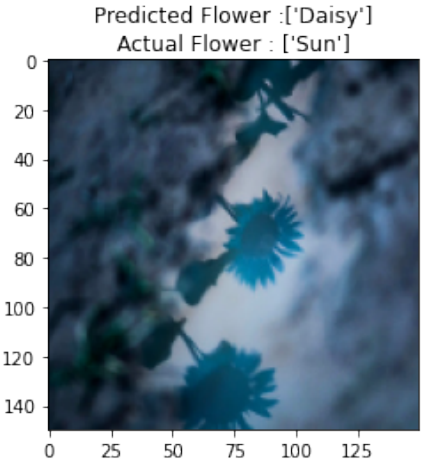

Predicted Flower :['Sun']
Actual Flower : ['Sun']

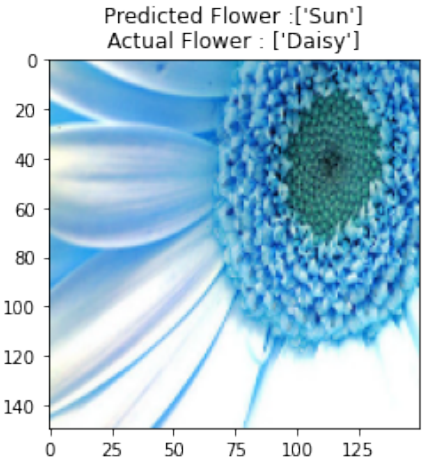

Predicted Flower :['Tulip']
Actual Flower : ['Tulip']

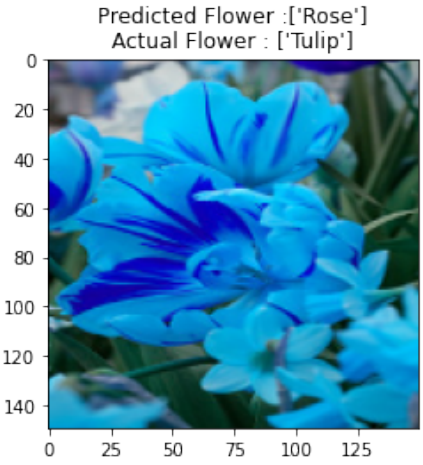
0.3.2 Visualization of bad prediction

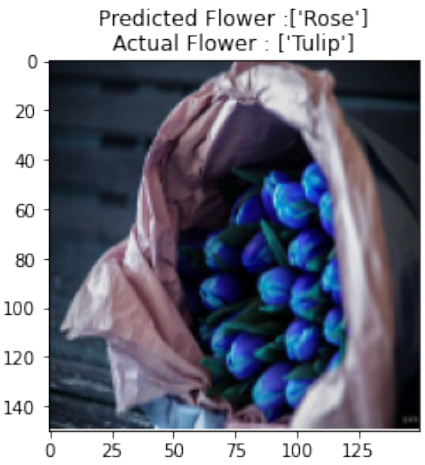
[73]:
```
warnings.filterwarnings('always')
warnings.filterwarnings('ignore')

count=0
fig,ax=plt.subplots(4,2)
fig.set_size_inches(15,15)
for i in range (4):
    for j in range (2):
        ax[i,j].imshow(x_test[bad_pred[count]])
        ax[i,j].set_title("Predicted Flower :"+str(le.
 ↪inverse_transform([pred_digits[bad_pred[count]]]))+"\n"+"Actual Flower :␣
 ↪"+str(le.inverse_transform([np.argmax(y_test[bad_pred[count]])])))
        plt.tight_layout()
        count+=1
```

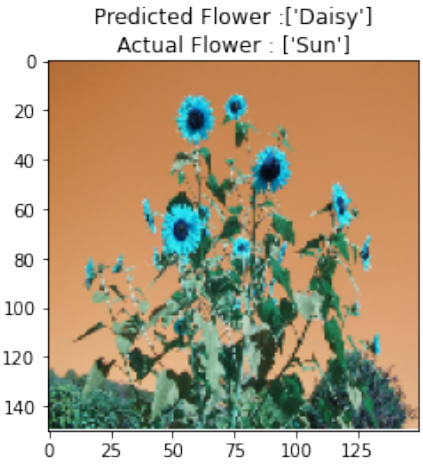

Predicted Flower :['Daisy']
Actual Flower : ['Sun']

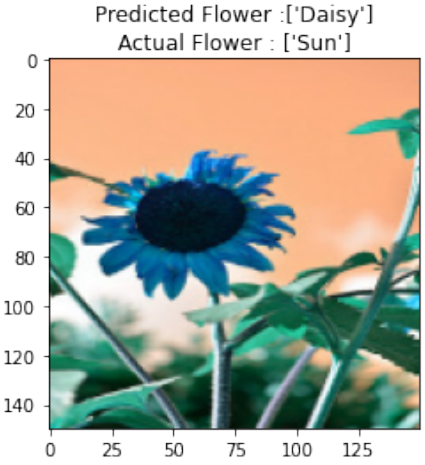

Predicted Flower :['Daisy']
Actual Flower : ['Sun']

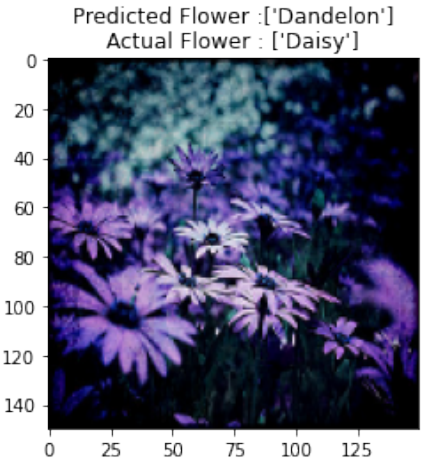

Predicted Flower :['Sun']
Actual Flower : ['Daisy']


Predicted Flower :['Rose']
Actual Flower : ['Tulip']


Predicted Flower :['Rose']
Actual Flower : ['Tulip']

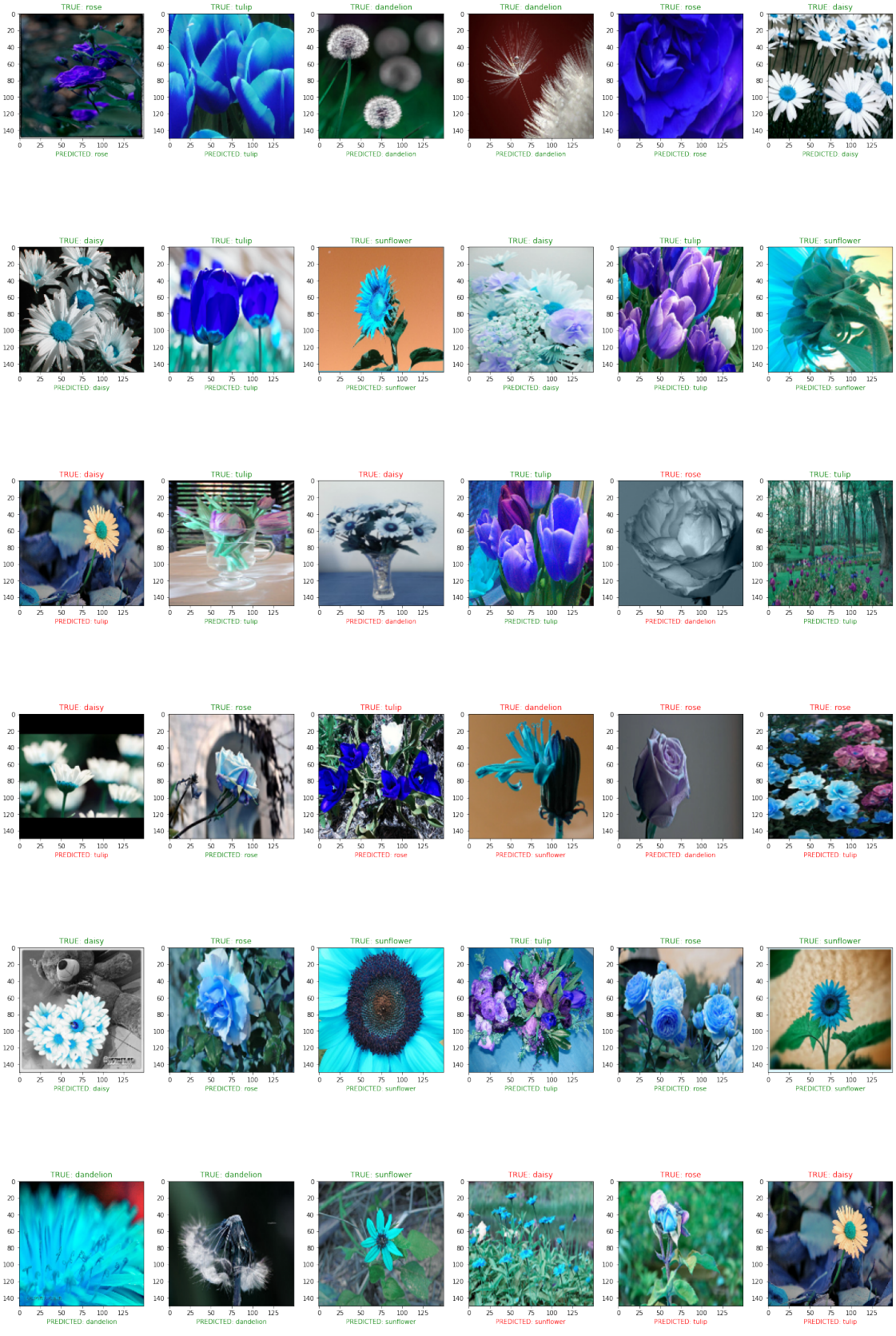

Predicted Flower :['Daisy']
Actual Flower : ['Sun']


Predicted Flower :['Daisy']
Actual Flower : ['Sun']


Predicted Flower :['Dandelon']
Actual Flower : ['Daisy']

0.4 General overview of prediction

[27]:
```
SIZE=150
folder_dir=r'C:/Users/33758/Desktop/Freedel ZINSOU PLY/Coursera/flowers/flowers'
categories = np.sort(os.listdir(folder_dir))
fig, ax = plt.subplots(6,6, figsize=(25, 40))

for i in range(6):
    for j in range(6):
        k = int(np.random.random_sample() * len(x_test))
        if(categories[np.argmax(y_test[k])] == categories[np.argmax(model.
↪predict(x_test)[k])]):
            ax[i,j].set_title("TRUE: " + categories[np.argmax(y_test[k])],␣
↪color='green')
            ax[i,j].set_xlabel("PREDICTED: " + categories[np.argmax(model.
↪predict(x_test)[k])], color='green')
            ax[i,j].imshow(np.array(x_test)[k].reshape(SIZE, SIZE, 3),␣
↪cmap='gray')
        else:
            ax[i,j].set_title("TRUE: " + categories[np.argmax(y_test[k])],␣
↪color='red')
            ax[i,j].set_xlabel("PREDICTED: " + categories[np.argmax(model.
↪predict(x_test)[k])], color='red')
            ax[i,j].imshow(np.array(x_test)[k].reshape(SIZE, SIZE, 3),␣
↪cmap='gray')
```


TRUE: rose
PREDICTED: rose
TRUE: tulip
PREDICTED: tulip
TRUE: dandelion
PREDICTED: dandelion
TRUE: dandelion
PREDICTED: dandelion
TRUE: rose
PREDICTED: rose
TRUE: daisy
PREDICTED: daisy
TRUE: daisy
PREDICTED: daisy
TRUE: tulip
PREDICTED: tulip
TRUE: sunflower
PREDICTED: sunflower
TRUE: daisy
PREDICTED: daisy
TRUE: tulip
PREDICTED: tulip
TRUE: sunflower
PREDICTED: sunflower
TRUE: daisy
PREDICTED: tulip
TRUE: tulip
PREDICTED: tulip
TRUE: daisy
PREDICTED: dandelion
TRUE: tulip
PREDICTED: tulip
TRUE: rose
PREDICTED: dandelion
TRUE: tulip
PREDICTED: tulip
TRUE: daisy
PREDICTED: tulip
TRUE: rose
PREDICTED: rose
TRUE: tulip
PREDICTED: rose
TRUE: dandelion
PREDICTED: sunflower
TRUE: rose
PREDICTED: dandelion
TRUE: rose
PREDICTED: tulip
TRUE: daisy
PREDICTED: daisy
TRUE: rose
PREDICTED: rose
TRUE: sunflower
PREDICTED: sunflower
TRUE: tulip
PREDICTED: tulip
TRUE: rose
PREDICTED: rose
TRUE: sunflower
PREDICTED: sunflower
TRUE: dandelion
PREDICTED: dandelion
TRUE: dandelion
PREDICTED: dandelion
TRUE: sunflower
PREDICTED: sunflower
TRUE: daisy
PREDICTED: sunflower
TRUE: rose
PREDICTED: tulip
TRUE: daisy
PREDICTED: tulip

0.4.1 Conclusion

The objective of this model was to classify and predict the name of each flower associate with his image, and the results show that this objective was achieved.We use two model, a simple model of CNN and a model with a VGG16 achitecture model. Finally we choose the simple model because , it's train better than VGG16 and the result on test dataset is better too. The model predicts well with both training and validation models. The loss function used minimises the errors as much as possible, which facilitates learning and thus increases the accuracy of the model 80 % for training dataset and 79% for validation dataset.