智慧化企業整合

Intelligent Integration of Enterprise

Project 2

花卉辨識系統

Group 7

109034546 吳欣晏

109034547 馮品叡

109034548 吳仲人

109034549 溫芳苓

指導教授： 邱銘傳 博士

目錄

一、 研究背景 2

二、 資料前處理過程 3

三、 模型建構 7

四、參數優化 13

五、結論與為來展望.............................17

一、 研究背景

1.情境描述：

在 2010 年的台北國際花卉博覽會吸引 896 萬 3,666 參觀人次，淨效益約為 294.77 億元，由此可知花卉所帶來的經濟效益極高，而花卉除了觀賞用途之外亦有食用、製作香氛、研究等其他用途。舉例來說，向日葵除了可以用來觀賞，其果實為葵花籽可直接作為點心食用，或是將其提煉壓榨為葵花籽油，而玫瑰因香氣特殊深受許多人喜愛，因此許多商人也會將其製作精油、香水等。然而，世界上花的種類舉不勝舉，各種花都有著不同的用途，甚至同一種類的花有許多不同的品種，時常令人混淆造成困擾。為解決上述問題，本小組欲透過深度學習建模，當人們購買花卉或是商人販售花卉有種類上之疑慮時，可將其花卉拍照透過本小組之技術，自動辨識其花卉的種類，並幫助使用者將照片分類歸檔，如此一來不僅可減少人工辨識的錯誤，也可減少其為了分辨花的種類找尋資料所浪費之時間。

2.問題描述(5W1H)：

針對此研究主題，5W1H 分析說明如下表：

項目	內容
What	購買花卉或是商人販售花卉時，需耗費人力自行辨識，不僅耗時也可能辨識錯誤造成財物損失。
Where	花店、欲利用花卉作為其他用途(提煉食用油、製作精油)之工廠。
Who	欲購買花卉之消費者、欲利用花卉作為其他用途(提煉食用油、製作精油)之廠商。
When	當人們購買花卉或是商人販售花卉有種類上之疑慮時
Why	透過深度學習的模型，建構花卉品種辨識並自動分類歸檔之技術，節省人們為了分辨花卉種類找尋資料所浪費之時間,降低人工辨識的錯誤率。
How	各類花卉照片的資料前處理及 CNN 模型訓練

二、 資料前處理過程

1. 資料集簡介：

由 Kaggle 公開數據集中取得花朵的圖像資料集。其中包含：

（1） 雛菊（daisy）: 769 筆圖像資料

（2） 蒲公英（dandelion）: 1052 筆圖像資料

（3） 玫瑰 （rose）: 784 筆圖像資料

（4） 向日葵（sunflower） : 734 筆圖像資料

（5） 鬱金香（tulip） : 984 筆圖像資料

2. Flower Recognition CNN Keras：

（1） 於 colab 導入資料集

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
from zipfile import ZipFile
file_name = 'drive/MyDrive/flowers.zip'
with ZipFile(file_name, 'r') as zip:
    zip.extractall()
    print('Done')
```

```
Done
```

(2) 載入會使用到的套件 Importing Various Modules

```
# data visualisation and manipulation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns

#configure
# sets matplotlib to inline and displays graphs below the corressponding cell.
%matplotlib inline
style.use('fivethirtyeight')
sns.set(style='whitegrid',color_codes=True)

#model selection
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

#preprocess.
from keras.preprocessing.image import ImageDataGenerator

#dl libraraies
from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam,SGD,Adagrad,Adadelta,RMSprop
from keras.utils import to_categorical

# specifically for cnn
from keras.layers import Dropout, Flatten,Activation
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization

import tensorflow as tf
import random as rn

# specifically for manipulating zipped images and getting numpy arrays of pixel values of images.
import cv2
import numpy as np
from tqdm import tqdm
import os
from random import shuffle
from zipfile import ZipFile
from PIL import Image
```

(3) 定義 X、Y 資料集，定義目錄以便導入資料

```
X=[]
Z=[]
IMG_SIZE=150
FLOWER_DAISY_DIR='../content/flowers/daisy'
FLOWER_SUNFLOWER_DIR='../content/flowers/sunflower'
FLOWER_TULIP_DIR='../content/flowers/tulip'
FLOWER_DANDI_DIR='../content/flowers/dandelion'
FLOWER_ROSE_DIR='../content/flowers/rose'
```

(4) 定義 make_train_data 函式：讀入彩色之圖像，並調整圖像的大小（尺寸 150×150）。

```
def  assign_label(img,flower_type):
        return  flower_type
```

```
def  make_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)  #讀入彩色照片
                img  =  cv2.resize(img,  (IMG_SIZE,IMG_SIZE))  #調整照片尺寸

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

(5) 執行此函式

```
make_train_data('Daisy',FLOWER_DAISY_DIR)
print(len(X))
```

```
100%|████████████| 769/769 [00:01<00:00, 489.09it/s]769
```

```
make_train_data('Sunflower',FLOWER_SUNFLOWER_DIR)
print(len(X))
```

```
100%|████████████| 734/734 [00:01<00:00, 369.73it/s]1503
```

```
make_train_data('Tulip',FLOWER_TULIP_DIR)
print(len(X))
```

```
100%|████████████| 984/984 [00:02<00:00, 418.69it/s]3471
```

```
make_train_data('Dandelion',FLOWER_DANDI_DIR)
print(len(X))
```

```
100%|████████████| 1052/1052 [00:02<00:00, 428.24it/s]452
```

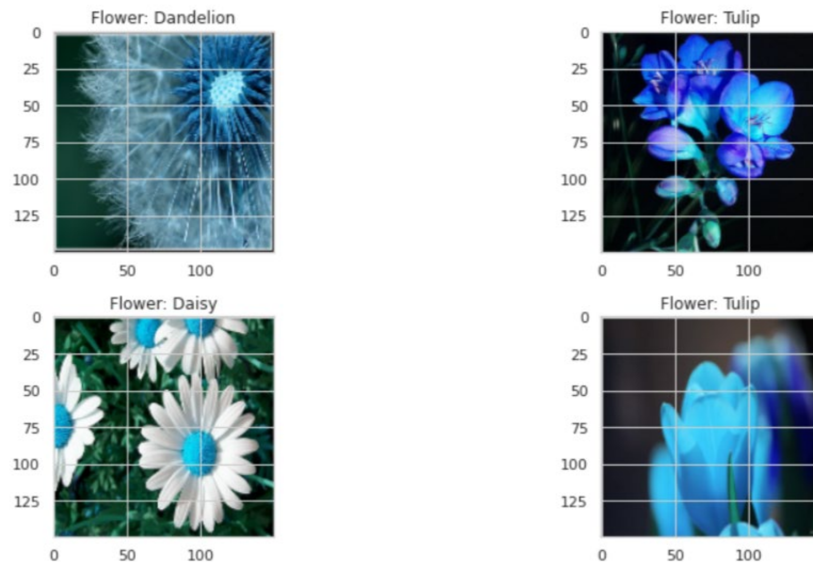
```
make_train_data('Rose',FLOWER_ROSE_DIR)
print(len(X))
```

```
100%|████████████| 784/784 [00:01<00:00, 501.24it/s]1553
```

(6) 以視覺化的方式呈現

```
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(Z))
                ax[i,j].imshow(X[l])
                ax[i,j].set_title('Flower:  '+Z[l])

plt.tight_layout()
```

結果如下所示



(7) 將 labels 進行 one-hot encoding

(i.e. Daisy -> 0, Rose -> 1, Tulip -> 2, etc)

```
le=LabelEncoder()
Y=le.fit_transform(Z)
Y=to_categorical(Y,5)  #將  labels  進行  one-hot  encoding
X=np.array(X)
X=X/255  #將每張圖片像素值皆除以255，圖像做歸一化，使其像素值從 [0-255] 縮放到 [0-1]。
```

(8) 將資料分成訓練集和測試集：使用 0.75/0.25 分割以確保有足夠多的圖像測試模型，並避免過度擬合的問題。

```
x_train,x_test,y_train,y_test=train_test_split(X,Y,test_size=0.25,random_state=42)
```

三、 模型建構

1. 發展模型：使用之模型為 Sequential；第一層 Layer 的 filters 為 32；第二層 Layer 的 filters 為 64；第三層 Layer 的 filters 為 96；第四層 Layer 的 filters 為 96。而 activation 皆設為 relu。

```
# # modelling starts using a CNN.

model = Sequential()
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)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


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(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"))
```

2. 利用 LR Annealer

```
[16] batch_size=128
     epochs=50

     from keras.callbacks import ReduceLROnPlateau
     red_lr= ReduceLROnPlateau(monitor='val_acc',patience=3,verbose=1,factor=0.1)
```

3. 數據擴充以防止過擬合

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.2,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.2,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images


datagen.fit(x_train)
```

4. 設定模型參數，使用的 optimizer 為 Adam，學習率為 0.2%。

```
model.compile(optimizer=Adam(lr=0.002),loss='categorical_crossentropy',metrics=['accuracy'])
```

5. 模型架構

```
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, 96)        55392
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 96)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 18, 18, 96)        83040
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 9, 9, 96)          0
_________________________________________________________________
flatten (Flatten)            (None, 7776)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3981824
_________________________________________________________________
activation (Activation)      (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 2565
=================================================================
Total params: 4,143,749
Trainable params: 4,143,749
Non-trainable params: 0
```

6. 模型擬合訓練數據並驗證測試數據

```
History = 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)
# model.fit(x_train,y_train,epochs=epochs,batch_size=batch_size,validation_data = (x_test,y_test))
```

```
Epoch 1/50
25/25 [==============================] - 13s 537ms/step - loss: 1.5753 - accuracy: 0.3160 - val_loss: 1.2150 - val_accuracy: 0.4958
Epoch 2/50
25/25 [==============================] - 13s 523ms/step - loss: 1.2060 - accuracy: 0.4888 - val_loss: 1.1646 - val_accuracy: 0.5199
Epoch 3/50
25/25 [==============================] - 13s 523ms/step - loss: 1.1360 - accuracy: 0.5315 - val_loss: 1.1619 - val_accuracy: 0.5254
Epoch 4/50
25/25 [==============================] - 13s 523ms/step - loss: 1.1182 - accuracy: 0.5446 - val_loss: 1.0387 - val_accuracy: 0.5846
Epoch 5/50
25/25 [==============================] - 13s 529ms/step - loss: 1.0413 - accuracy: 0.5784 - val_loss: 1.0379 - val_accuracy: 0.5930
Epoch 6/50
25/25 [==============================] - 13s 526ms/step - loss: 0.9868 - accuracy: 0.6137 - val_loss: 0.9723 - val_accuracy: 0.6309
Epoch 7/50
25/25 [==============================] - 13s 525ms/step - loss: 0.9478 - accuracy: 0.6240 - val_loss: 0.9532 - val_accuracy: 0.6207
Epoch 8/50
25/25 [==============================] - 13s 524ms/step - loss: 0.9053 - accuracy: 0.6355 - val_loss: 0.9864 - val_accuracy: 0.6272
Epoch 9/50
25/25 [==============================] - 13s 532ms/step - loss: 0.8848 - accuracy: 0.6490 - val_loss: 0.9257 - val_accuracy: 0.6549
Epoch 10/50
25/25 [==============================] - 13s 531ms/step - loss: 0.8767 - accuracy: 0.6557 - val_loss: 0.8883 - val_accuracy: 0.6448
Epoch 11/50
25/25 [==============================] - 13s 527ms/step - loss: 0.8373 - accuracy: 0.6779 - val_loss: 0.9264 - val_accuracy: 0.6383
Epoch 12/50
25/25 [==============================] - 13s 538ms/step - loss: 0.8288 - accuracy: 0.6808 - val_loss: 0.8802 - val_accuracy: 0.6735
```

```
Epoch 13/50
25/25 [==============================] - 13s 527ms/step - loss: 0.8018 - accuracy: 0.6866 - val_loss: 0.8425 - val_accuracy: 0.6846
Epoch 14/50
25/25 [==============================] - 13s 530ms/step - loss: 0.7817 - accuracy: 0.6917 - val_loss: 0.8397 - val_accuracy: 0.6892
Epoch 15/50
25/25 [==============================] - 13s 527ms/step - loss: 0.7559 - accuracy: 0.7065 - val_loss: 0.8592 - val_accuracy: 0.6892
Epoch 16/50
25/25 [==============================] - 13s 530ms/step - loss: 0.7352 - accuracy: 0.7174 - val_loss: 0.8324 - val_accuracy: 0.6772
Epoch 17/50
25/25 [==============================] - 13s 525ms/step - loss: 0.7267 - accuracy: 0.7171 - val_loss: 0.8332 - val_accuracy: 0.6966
Epoch 18/50
25/25 [==============================] - 13s 526ms/step - loss: 0.7132 - accuracy: 0.7216 - val_loss: 0.7934 - val_accuracy: 0.7169
Epoch 19/50
25/25 [==============================] - 13s 524ms/step - loss: 0.6951 - accuracy: 0.7225 - val_loss: 0.7694 - val_accuracy: 0.7169
Epoch 20/50
25/25 [==============================] - 13s 540ms/step - loss: 0.6754 - accuracy: 0.7335 - val_loss: 0.7708 - val_accuracy: 0.7299
Epoch 21/50
25/25 [==============================] - 13s 524ms/step - loss: 0.6611 - accuracy: 0.7463 - val_loss: 0.8292 - val_accuracy: 0.7095
Epoch 22/50
25/25 [==============================] - 13s 529ms/step - loss: 0.6422 - accuracy: 0.7463 - val_loss: 0.7961 - val_accuracy: 0.7058
Epoch 23/50
25/25 [==============================] - 13s 526ms/step - loss: 0.6509 - accuracy: 0.7514 - val_loss: 0.8928 - val_accuracy: 0.6966
Epoch 24/50
25/25 [==============================] - 13s 524ms/step - loss: 0.6330 - accuracy: 0.7588 - val_loss: 0.7678 - val_accuracy: 0.7428
Epoch 25/50
25/25 [==============================] - 13s 524ms/step - loss: 0.6240 - accuracy: 0.7543 - val_loss: 0.7483 - val_accuracy: 0.7299
```

```
Epoch 26/50
25/25 [==============================] - 13s 524ms/step - loss: 0.6031 - accuracy: 0.7598 - val_loss: 0.8074 - val_accuracy: 0.7132
Epoch 27/50
25/25 [==============================] - 13s 524ms/step - loss: 0.5811 - accuracy: 0.7771 - val_loss: 0.7518 - val_accuracy: 0.7456
Epoch 28/50
25/25 [==============================] - 13s 528ms/step - loss: 0.5565 - accuracy: 0.7829 - val_loss: 0.7402 - val_accuracy: 0.7475
Epoch 29/50
25/25 [==============================] - 13s 525ms/step - loss: 0.5754 - accuracy: 0.7787 - val_loss: 0.7471 - val_accuracy: 0.7521
Epoch 30/50
25/25 [==============================] - 13s 526ms/step - loss: 0.5625 - accuracy: 0.7855 - val_loss: 0.7439 - val_accuracy: 0.7576
Epoch 31/50
25/25 [==============================] - 13s 530ms/step - loss: 0.5402 - accuracy: 0.7900 - val_loss: 0.9242 - val_accuracy: 0.7095
Epoch 32/50
25/25 [==============================] - 13s 527ms/step - loss: 0.5634 - accuracy: 0.7861 - val_loss: 0.7779 - val_accuracy: 0.7290
Epoch 33/50
25/25 [==============================] - 13s 526ms/step - loss: 0.5029 - accuracy: 0.8154 - val_loss: 0.8046 - val_accuracy: 0.7299
Epoch 34/50
25/25 [==============================] - 13s 526ms/step - loss: 0.5103 - accuracy: 0.8025 - val_loss: 0.7424 - val_accuracy: 0.7558
Epoch 35/50
25/25 [==============================] - 13s 524ms/step - loss: 0.5007 - accuracy: 0.8057 - val_loss: 0.7240 - val_accuracy: 0.7428
Epoch 36/50
25/25 [==============================] - 13s 525ms/step - loss: 0.5217 - accuracy: 0.8003 - val_loss: 0.8539 - val_accuracy: 0.7280
Epoch 37/50
25/25 [==============================] - 13s 526ms/step - loss: 0.5127 - accuracy: 0.8006 - val_loss: 0.7201 - val_accuracy: 0.7586
Epoch 38/50
25/25 [==============================] - 13s 523ms/step - loss: 0.5048 - accuracy: 0.8028 - val_loss: 0.7425 - val_accuracy: 0.7345
```

```
Epoch 39/50
25/25 [==============================] - 13s 529ms/step - loss: 0.5238 - accuracy: 0.7954 - val_loss: 0.7918 - val_accuracy: 0.7447
Epoch 40/50
25/25 [==============================] - 13s 527ms/step - loss: 0.4999 - accuracy: 0.8064 - val_loss: 0.7199 - val_accuracy: 0.7660
Epoch 41/50
25/25 [==============================] - 13s 540ms/step - loss: 0.4892 - accuracy: 0.8141 - val_loss: 0.8682 - val_accuracy: 0.7373
Epoch 42/50
25/25 [==============================] - 13s 526ms/step - loss: 0.4328 - accuracy: 0.8314 - val_loss: 0.7507 - val_accuracy: 0.7475
Epoch 43/50
25/25 [==============================] - 13s 526ms/step - loss: 0.4175 - accuracy: 0.8365 - val_loss: 0.7842 - val_accuracy: 0.7391
Epoch 44/50
25/25 [==============================] - 13s 527ms/step - loss: 0.4457 - accuracy: 0.8292 - val_loss: 0.7749 - val_accuracy: 0.7364
Epoch 45/50
25/25 [==============================] - 13s 529ms/step - loss: 0.4338 - accuracy: 0.8353 - val_loss: 0.7163 - val_accuracy: 0.7650
Epoch 46/50
25/25 [==============================] - 13s 526ms/step - loss: 0.4389 - accuracy: 0.8282 - val_loss: 0.7916 - val_accuracy: 0.7428
Epoch 47/50
25/25 [==============================] - 13s 524ms/step - loss: 0.4144 - accuracy: 0.8465 - val_loss: 0.7579 - val_accuracy: 0.7502
Epoch 48/50
25/25 [==============================] - 13s 525ms/step - loss: 0.4268 - accuracy: 0.8423 - val_loss: 0.7392 - val_accuracy: 0.7595
Epoch 49/50
25/25 [==============================] - 13s 526ms/step - loss: 0.4120 - accuracy: 0.8404 - val_loss: 0.7170 - val_accuracy: 0.7586
Epoch 50/50
25/25 [==============================] - 13s 526ms/step - loss: 0.4082 - accuracy: 0.8475 - val_loss: 0.7017 - val_accuracy: 0.7798
```

7. 模型校度分析

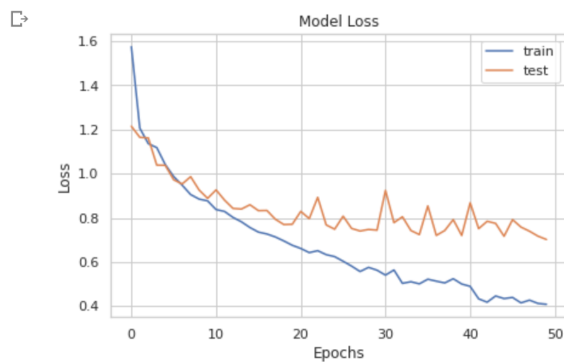
（1） 取得測試集準確率，test accuracy 為 78%，如下圖所示：

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

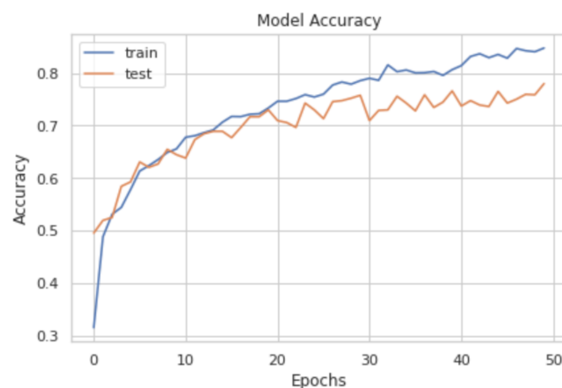
```
26/26 [==============================] - 1s 25ms/step - loss: 0.2692 - accuracy: 0.8924
9/9 [==============================] - 0s 21ms/step - loss: 0.7017 - accuracy: 0.7798
Train accuracy: 0.8923503756523132
Test accuracy: 0.7798334956169128
```

（2）分別繪製 Loss、Accuracy 與 Epochs 之關係圖

```
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()
```




```
[23] 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()
```



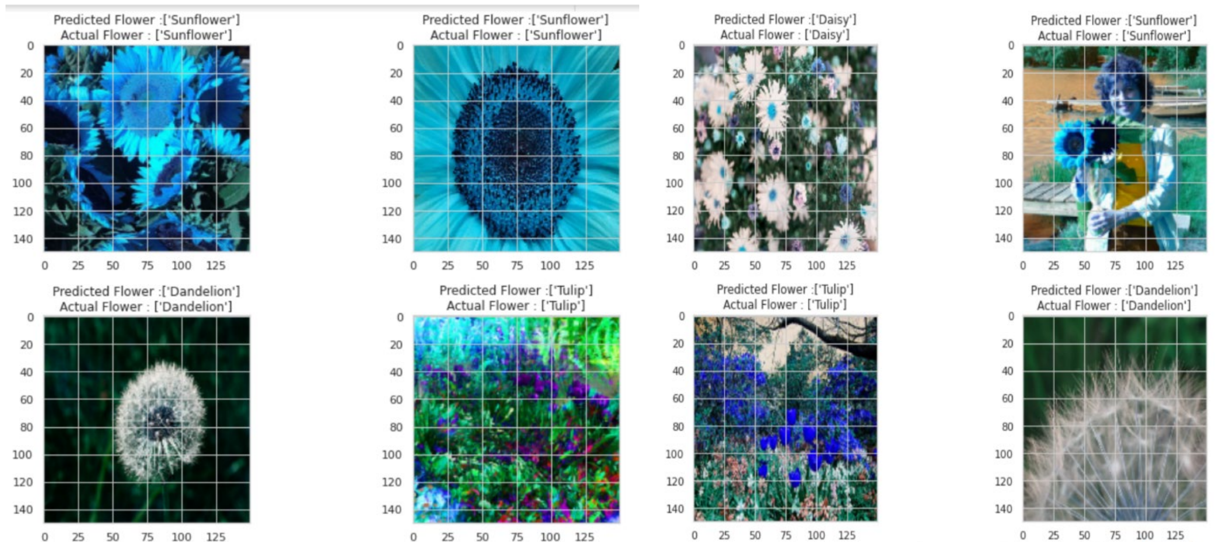
8. 視覺化預測結果與實際結果

```
[24] # getting predictions on val set.
     pred=model.predict(x_test)
     pred_digits=np.argmax(pred,axis=1)
```

```
[25] # now storing some properly as well as misclassified indexes'.
     i=0
     prop_class=[]
     mis_class=[]

     for i in range(len(y_test)):
         if(np.argmax(y_test[i])==pred_digits[i]):
             prop_class.append(i)
         if(len(prop_class)==8):
             break

     i=0
     for i in range(len(y_test)):
         if(not np.argmax(y_test[i])==pred_digits[i]):
             mis_class.append(i)
         if(len(mis_class)==8):
             break
```

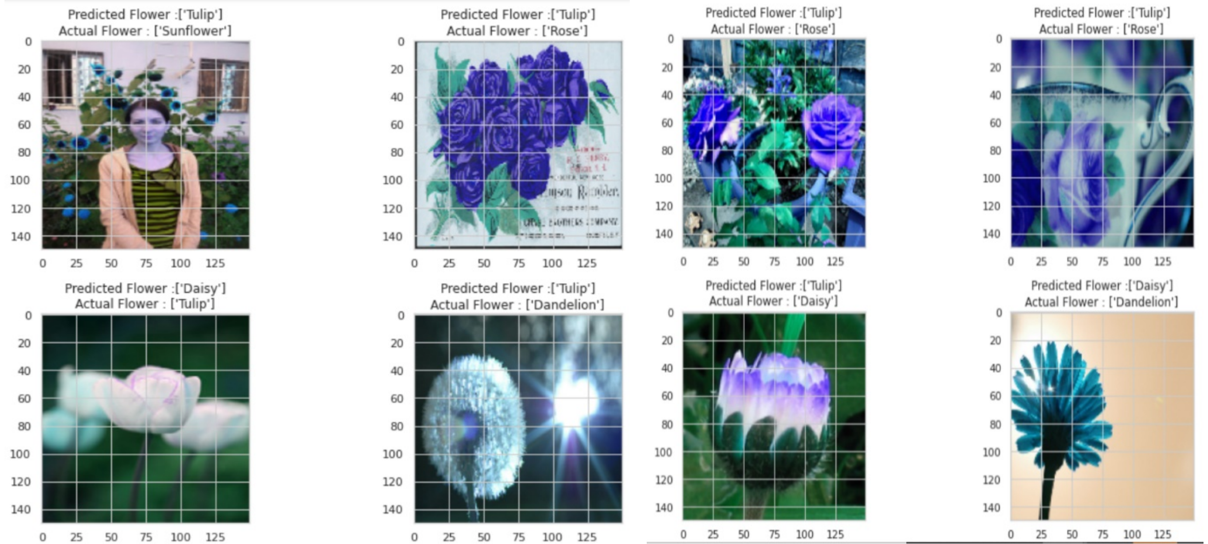
(1) 預測結果與實際結果相同

```
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[prop_class[count]])
        ax[i,j].set_title("Predicted Flower :"+str(le.inverse_transform([pred_digits[prop_class[count]]]))
                          +"\n"+"Actual Flower : "+str(le.inverse_transform([np.argmax(y_test[prop_class[count]])])))
        plt.tight_layout()
        count+=1
```



（2） 預測結果與實際結果不同

```
[27] 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[mis_class[count]])
                     ax[i,j].set_title("Predicted  Flower  :"+str(le.inverse_transform([pred_digits[mis_class[count]]]))
                                                         +"\n"+"Actual  Flower  :  "+str(le.inverse_transform([np.argmax(y_test[mis_class[count]])])))
                     plt.tight_layout()
                     count+=1
```



四、 參數優化

為了增加準確率，本組嘗試調整 Activation、Filters、Learning Rate，並分別採用 test accuracy 較高之參數。

1. 調整 Activation 參數，並分別取得 Sigmoid、Relu、Tanh 之 test accuracy。

   (1) 將 Activation 設為 Sigmoid，並取得 Sigmoid 之 test accuracy 約為 21%。

```
# # modelling starts using a CNN.

model = Sequential()
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',activation ='sigmoid', input_shape = (150,150,3)))
model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='sigmoid'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='sigmoid'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

model.add(Conv2D(filters = 96, kernel_size = (3,3),padding = 'Same',activation ='sigmoid'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

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

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 28ms/step - loss: 1.5981 - accuracy: 0.2535
9/9 [==============================] - 0s 33ms/step - loss: 1.6151 - accuracy: 0.2128
Train accuracy: 0.25354719161987305
Test accuracy: 0.21276596188545227
```

   (2) 將 Activation 設為 Relu，並取得 Relu 之 test accuracy 約為 78%。

```
# # modelling starts using a CNN.

model = Sequential()
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)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


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(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"))
```

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 25ms/step - loss: 0.2692 - accuracy: 0.8924
9/9 [==============================] - 0s 21ms/step - loss: 0.7017 - accuracy: 0.7798
Train accuracy: 0.8923503756523132
Test accuracy: 0.7798334956169128
```

(3) 將 Activation 設為 Tanh，並取得 Tanh 之 test accuracy 約為 21%。

```
[36] # # modelling starts using a CNN.

model = Sequential()
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',activation ='tanh', input_shape = (150,150,3)))
model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='tanh'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='tanh'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

model.add(Conv2D(filters = 96, kernel_size = (3,3),padding = 'Same',activation ='tanh'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

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

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 30ms/step - loss: 1.7168 - accuracy: 0.2535
9/9 [==============================] - 0s 26ms/step - loss: 1.7725 - accuracy: 0.2128
Train accuracy: 0.25354719161987305
Test accuracy: 0.21276596188545227
```

2. 調整 Filters 參數，並分別取得各 Filters 之 test accuracy。

(1) 將 Filters 設為 8，並取得其 test accuracy 約為 75%。

```
# # modelling starts using a CNN.

model = Sequential()
model.add(Conv2D(filters = 8, kernel_size = (5,5),padding = 'Same',activation ='relu', input_shape = (150,150,3)))
model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


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(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"))
```

```
[21] train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
     test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
     print('Train accuracy:', train_acc)
     print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 22ms/step - loss: 0.2884 - accuracy: 0.8942
9/9 [==============================] - 0s 25ms/step - loss: 0.7857 - accuracy: 0.7549
Train accuracy: 0.8942010998725891
Test accuracy: 0.7548565864562988
```

(2) 將 Filters 設為 16，並取得其 test accuracy 約為 76%。

```
# # modelling starts using a CNN.

model = Sequential()
model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'Same',activation ='relu', input_shape = (150,150,3)))
model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


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(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"))
```

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 23ms/step - loss: 0.2531 - accuracy: 0.9078
9/9 [==============================] - 0s 25ms/step - loss: 0.8014 - accuracy: 0.7567
Train accuracy: 0.907772958278656
Test accuracy: 0.7567067742347717
```

(3) 將 Filters 設為 32，並取得其 test accuracy 約為 78%。

```
# # modelling starts using a CNN.

model = Sequential()
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)))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


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(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"))
```

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 25ms/step - loss: 0.2692 - accuracy: 0.8924
9/9 [==============================] - 0s 21ms/step - loss: 0.7017 - accuracy: 0.7798
Train accuracy: 0.8923503756523132
Test accuracy: 0.7798334956169128
```

3. 調整 Learning Rate 參數，並分別取得各 Learning Rate 之 test accuracy。

(1) 將 Learning Rate 設為 0.5%，並取得其 test accuracy 約為 64%。

```
model.compile(optimizer=Adam(lr=0.005),loss='categorical_crossentropy',metrics=['accuracy'])
```

```
[21] train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
     test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
     print('Train accuracy:', train_acc)
     print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 25ms/step - loss: 0.7723 - accuracy: 0.7051
9/9 [==============================] - 0s 31ms/step - loss: 0.9539 - accuracy: 0.6420
Train accuracy: 0.705120325088501
Test accuracy: 0.6419981718063354
```

(2) 將 Learning Rate 設為 0.2%，並取得其 test accuracy 約為 78%。

```
model.compile(optimizer=Adam(lr=0.002),loss='categorical_crossentropy',metrics=['accuracy'])
```

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 25ms/step - loss: 0.2692 - accuracy: 0.8924
9/9 [==============================] - 0s 21ms/step - loss: 0.7017 - accuracy: 0.7798
Train accuracy: 0.8923503756523132
Test accuracy: 0.7798334956169128
```

(3) 將 Learning Rate 設為 0.05%，並取得其 test accuracy 約為 75%。

```
model.compile(optimizer=Adam(lr=0.0005),loss='categorical_crossentropy',metrics=['accuracy'])
```

```
train_loss, train_acc = model.evaluate(x_train, y_train, batch_size=batch_size)
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Train accuracy:', train_acc)
print('Test accuracy:', test_acc)
```

```
26/26 [==============================] - 1s 25ms/step - loss: 0.2774 - accuracy: 0.8967
9/9 [==============================] - 0s 22ms/step - loss: 0.7834 - accuracy: 0.7530
Train accuracy: 0.8966687321662903
Test accuracy: 0.7530064582824707
```

本組將各參數調整彙整為下表，由下表可知，Activation 若採用 Relu 其 test accuracy 較高，Filters 若採用 32 其 test accuracy 較高，Learning Rate 若採用 0.2%其 test accuracy 較高；本組將採用以上 test accuracy 較高之參數。

Activation		Filters		Learning Rate	
參數	**Test accuracy**	**參數**	**Test accuracy**	**參數**	**Test accuracy**
Sigmoid	21%	8	75%	0.5%	64%
Relu	78%	16	76%	0.2%	78%
Tanh	21%	32	78%	0.05%	75%

五、結論與未來展望

建構花卉品種辨識並自動分類歸檔之技術，節省所浪費之時間以及降低人工辨識的錯誤率，本小組透過多次超參數的調整可使準確率的提升，未來也可以使用 Generative Adversarial Network(GA)等方法提升準確率；AI 與生技醫療結合也是目前的主流；RPA 與 AI 結合更能大幅降低成本與時間，上述方法能提升準確率及降低成本

參考文獻：

https://medium.com/@kartikeya.lakhanpal/implementing-cnn-for-image-classification-problems-on-google-colab-a70344b5bd10