Flower Power

30-September-2019 | Version 1.0

Dai Yirui, Dong Meirong, Lim Chong Seng Hermann, Yam Gui Peng David

Prepared for NUS-ISS Master of Technology in Intelligent Systems Pattern Recognition CA2

Table of Contents

1. Abstract

The team has chosen to work on the problem of differentiating between flowers and non-flowers type images, with the goal of training a model that can classify at a high accuracy on whether an input image is a flower or non-flower type. In order to better measure the performance of each model, we created a baseline model that was able to get 85% accuracy rate. After evaluating the baseline model we looked at the State-of-the-Art (SOTA) models for similar problem sets (such as CIFAR 10) and attempted to understand and subsequently implement the architectures suggested. We conducted a series of experiments following popular architectures - Deep CNN, VGG, AlexNet, ResNet, Inception V4, Inception-ResNet V2 and SE Inception-ResNet V2. All images (Data_V1 and Data_V2) used were scraped from the Internet, and were subsequently resized and stored in .npz format for different models to load. After rounds of parameter tuning, Deep CNN, VGG and AlexNet were able to reach around 90-92% accuracy using Data_V1. In comparison, ResNet, Inception V4, Inception-ResNet V2 and SE Inception-ResNet V2 reached around 94-95% using Data_V2. In the end, we put all the results together and managed to get a further optimal model using a voting approach.

2. Environment Setup

All training and testing introduced in VGG16/AlexNet approaches were conducted using Google Colab (https://colab.research.google.com). Programs run on Python 3 with GPU enabled and server equipped with 12.72G RAM/358.27 GB disk.

All training and testing introduced in ResNet, Inception V4, Inception-ResNet V2 and SE Inception-ResNet V2 were conducted on local computer using a NVIDIA GeForce 1050 GTX GPU card/ 16G RAM. Shown below are the codes required to setup the given environment:

```
$ conda create --name tf-gpu
$ conda activate tf-gpu
$ conda install -c aaronzs tensorflow-gpu
$ conda install -c anaconda cudatoolkit==9.0
$ conda install -c anaconda cudnn==7.1.4
```

3. Data Set

The dataset is built using the images scraped from the Internet.

The first version data set (Data_V1) contains 3061 flower pictures, 854 fungus pictures, 997 rock pictures and 419 umbrella pictures. This makes a 3061 flower vs. 2270 non-flower set. The images were scraped from ImageNet and Pexel using custom created scripts (Download-ImageNet.ipynb and Download-Pexels.ipynb).

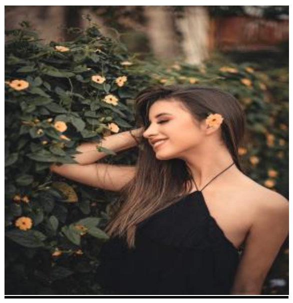
The second version data set (Data_V2) was created to extend the amount of flower and non-flower examples that were viewed by the model. It contains 11250 flower pictures, 2638 fungus pictures, 1007 rock pictures and 3777 umbrella pictures. This makes a 11250 flower vs. 7422 non-flower set. Additional images were either scrapped with an external helper script OIDv4_ToolKit from Google OpenImage or downloaded manually from 102flowersdataset.zip found in Kaggle.

For both Data_V1 and DataV2, the images were saved in either .jpg and .jpeg formats in their original form. The images were then resized to either 64x64, 96x96 or 224x224 pixel sizes and saved in .npz format for easy usage. A 80-20 train-test dataset was created from the resized .npz files and separated into another .npz file for subsequent usage.

Whilst it is possible to redownload the images and resize the images, it is suggested to use the .npz files created which serve as save points to ensure the dataset is exactly the same. This is important because it was noticed that the web servers hosting the images might unexpectedly remove the respective images, resulting in mismatches between various train-test sets. Hence both the resized images (64x64, 96x96, 224x224) and the train-test datasets were saved and provided.

Note: After the first few runs of training, some images which were misclassified (tagged as “flowers” but classified as “non-flowers” or the reverse) were evaluated and it was noticed that some images were wrongly tagged or had the “flowers” in very indistinct locations. These images were subsequently removed when generating the resized .npz files.

Shown below are a few examples of images erroneously tagged as “flowers”:

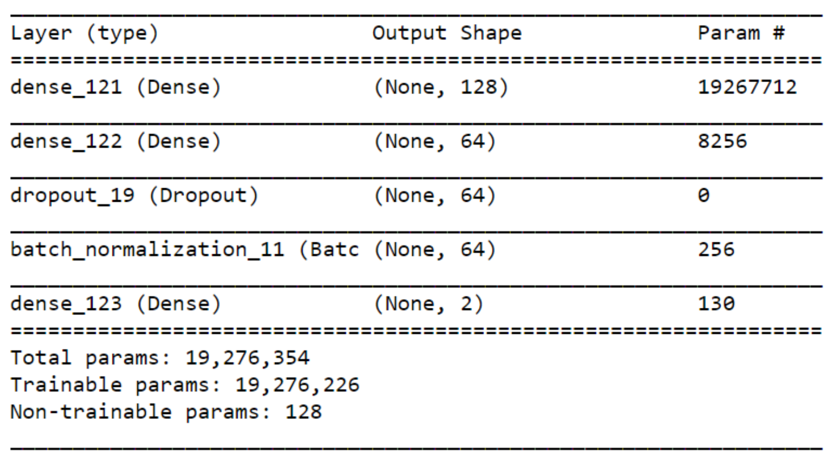
4. Baseline Model

Data Set:

The first version data set (Data_V1) was used for training and testing.

Experiment 0 - Neural Network

The neural network baseline model is designed with a stack of 3 dense layers, otherwise known as fully-connected layers without a convolutional layer.

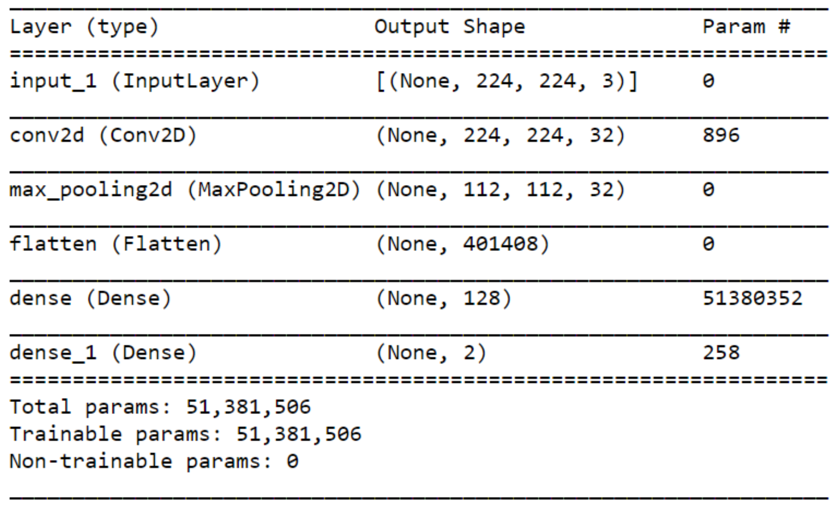


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_121 (Dense)            (None, 128)               19267712
_________________________________________________________________
dense_122 (Dense)            (None, 64)                8256
_________________________________________________________________
dropout_19 (Dropout)         (None, 64)                0
_________________________________________________________________
batch_normalization_11 (Batc (None, 64)                256
_________________________________________________________________
dense_123 (Dense)            (None, 2)                 130
=================================================================
Total params: 19,276,354
Trainable params: 19,276,226
Non-trainable params: 128
_________________________________________________________________
```

The best accuracy achieved (average of multiple runs) after hyperparameter tuning is about **68%**.

Experiment 1 - Simple Convolutional Neural Network

The accuracy improves significantly after a simple convolutional neural network is used.



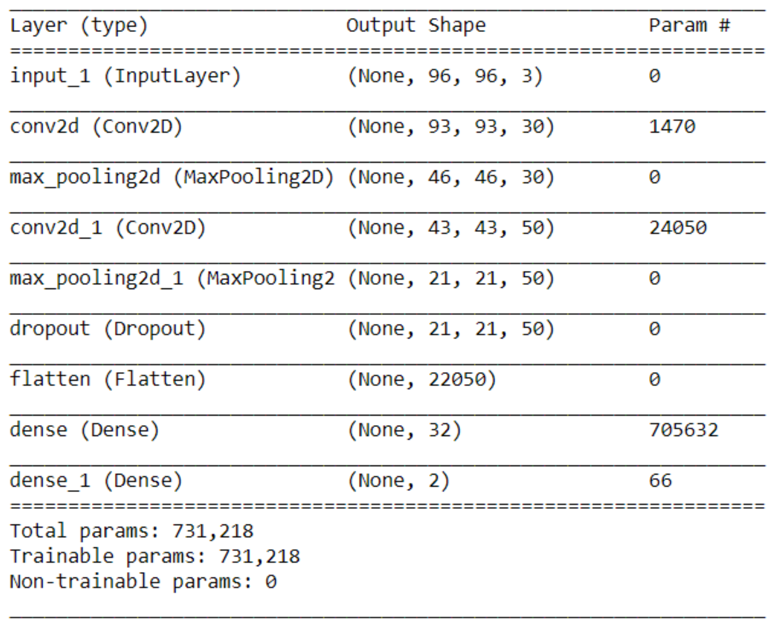
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0         
_________________________________________________________________
conv2d (Conv2D)              (None, 224, 224, 32)      896       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 32)      0         
_________________________________________________________________
flatten (Flatten)            (None, 401408)            0         
_________________________________________________________________
dense (Dense)                (None, 128)               51380352  
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 258       
=================================================================
Total params: 51,381,506
Trainable params: 51,381,506
Non-trainable params: 0
_________________________________________________________________
```

The accuracy (average of multiple runs) immediately improves to about **85%**. This is because in a regular neural network, the entire image is used to train the network. Therefore, neural network works well for simple centered images (for example a centered handwritten digit image) but fails to recognize images with more complex variation (a person holding a flower). While convolutional neural network learns the patterns in the images: edges, corners, arcs, followed by more complex figures and works better for image classification.

Experiment 2 - Deep Convolutional Neural Network

We use a more complicated basic architecture of Convolution, Pooling, Convolution, Pooling, Dropout, Dense:

```
def createBaselineModel():
  inputs = Input(shape=(imgrows, imgclms, channel))
  x = Conv2D(30, (4, 4), activation='relu')(inputs)
  x = MaxPooling2D(pool_size=(2, 2))(x)
  x = Conv2D(50, (4, 4), activation='relu')(x)
  x = MaxPooling2D(pool_size=(2, 2))(x)
  x = Dropout(0.3)(x)
  x = Flatten()(x)
  x = Dense(32, activation='relu')(x)
  x = Dense(num_classes, activation='softmax')(x)

  model = Model(inputs=[inputs],outputs=x)

  model.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])
  return model
```



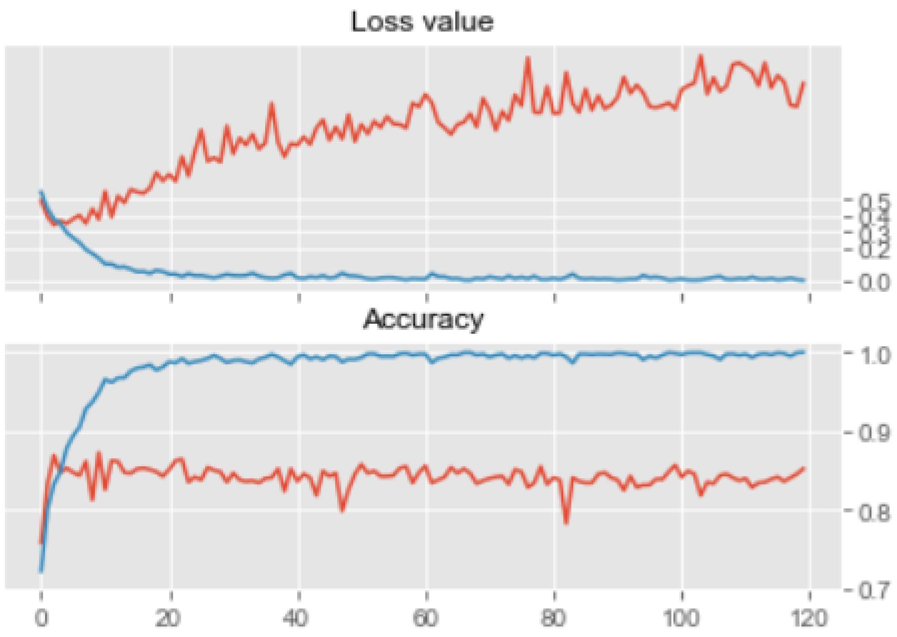
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 96, 96, 3)         0         
_________________________________________________________________
conv2d (Conv2D)              (None, 93, 93, 30)        1470      
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 46, 46, 30)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 43, 43, 50)        24050     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 21, 21, 50)        0         
_________________________________________________________________
dropout (Dropout)            (None, 21, 21, 50)        0         
_________________________________________________________________
flatten (Flatten)            (None, 22050)             0         
_________________________________________________________________
dense (Dense)                (None, 32)                705632    
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 66        
=================================================================
Total params: 731,218
Trainable params: 731,218
Non-trainable params: 0
_________________________________________________________________
```

It was trained for 120 epochs without any form of data augmentation:

```
Epoch 118/120
4264/4264 [==============================] - 4s 891us/step - loss: 0.0138 - acc: 0.9951 - val_loss: 1.0789 - val_acc: 0.8407
Epoch 119/120
4264/4264 [==============================] - 4s 881us/step - loss: 0.0046 - acc: 0.9993 - val_loss: 1.0697 - val_acc: 0.8454
Epoch 120/120
4264/4264 [==============================] - 4s 1ms/step - loss: 6.6477e-04 - acc: 0.9998 - val_loss: 1.2111 - val_acc: 0.8519
```

It achieved a respectable accuracy of **87.25%**:

```
Best accuracy (on testing dataset): 87.25%
              precision    recall  f1-score   support

      flower     0.8581    0.8392    0.8486       454
  non-flower     0.8828    0.8972    0.8900       613

    accuracy                         0.8725      1067
   macro avg     0.8705    0.8682    0.8693      1067
weighted avg     0.8723    0.8725    0.8723      1067

[[381  73]
 [ 63 550]]
```

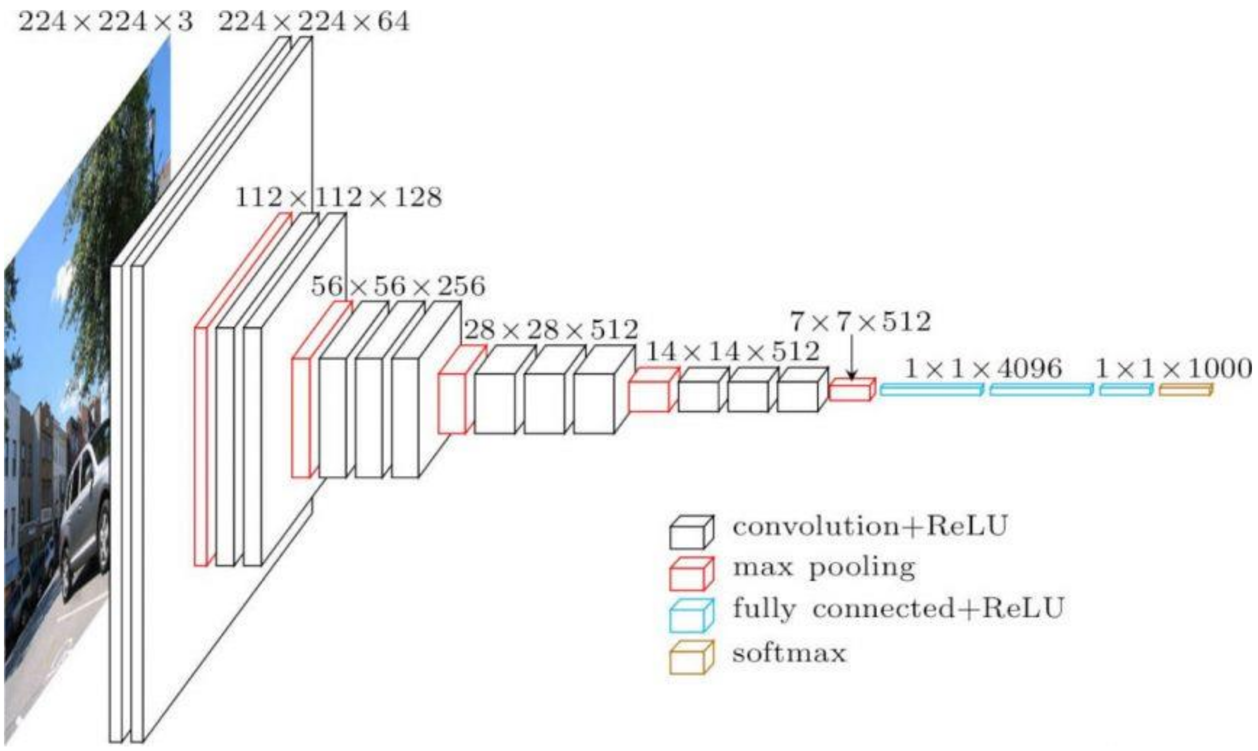


5. Approach Towards VGG16 Structure

Data Set:
Data (images) used for this approach are scraped from the Internet. The first version data set (Data_V1) was used for training and testing with VGG16. Although there were more data collected (Data_V2) along the process, only Data_V1 was used for modeling and experiments.

Images within Data_V1 are all in .jpg or .JPEG format. We firstly resize images to a standard 224*224 pixels shape to align with VGG standard (Simonyan & Zisserman, 2015) and stored in numpy array (.npz). Then we split the data into 80% training and 20% testing sets.

VGG16 Architecture:
One of the most commonly known VGG model - VGG16 was proposed by Simonyan and Zisserman. VGG16 significantly outperformed models in previous generations in the ILSVRC (Hassan, 2018).

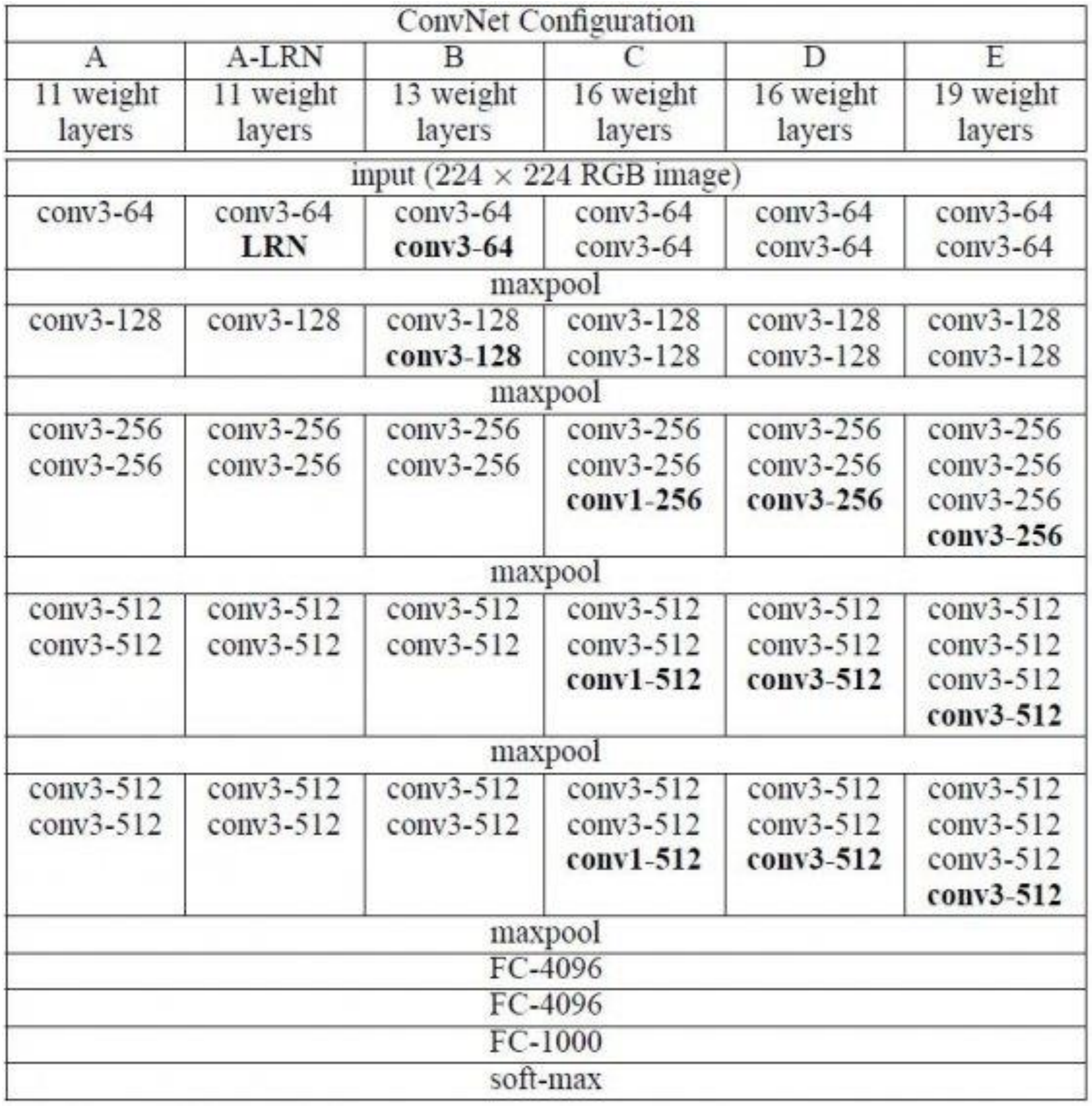


(source: https://neurohive.io/en/popular-networks/vgg16/)

In the VGG16 model, there are 5 Conv2d + MaxPooling blocks followed by 3 fully connected layers. The filter is a 3*3 kernel. The model used a softmax activation at its last layer and is able to classify 1000 types.

The target of original VGG16 model is to classify input images into 1000 classes. To achieve that, researchers built a dataset containing more than 1 million images: training (1.3M images), validation (50K images), and testing (100K images with held-out class labels) (Simonyan & Zisserman, 2015).

Notably, VGG16 wasn't the only model built during the original research. There were other models with similar configuration but different weight layers been built. The researchers mentioned that some initializations may lead to stall the learning due to the instability of gradient in deep nets (Simonyan & Zisserman, 2015). Thus they trained shallower net - Configuration A first and used net A to initialize the deeper net. We will discuss this technique in the coming experiment sections.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

(D: the VGG16 model configuration. source: https://neurohive.io/en/popular-networks/vgg16/)

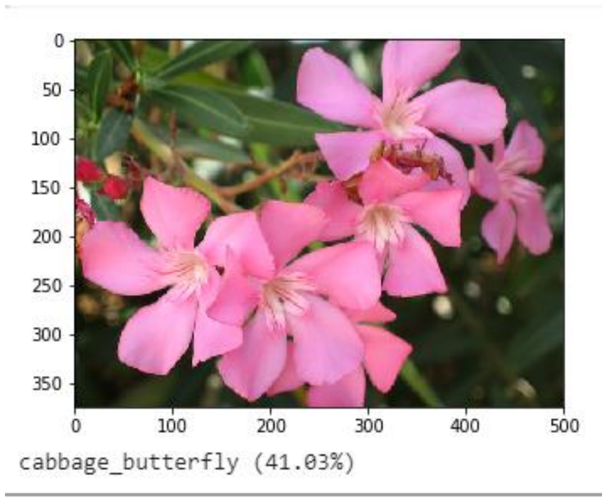
Experiment 0 - The Pre-trained Keras VGG16:

We notice that there is already an available package on Keras which we can directly import VGG16 model with pre-trained dataset sets to 'imagenet' as shown below:

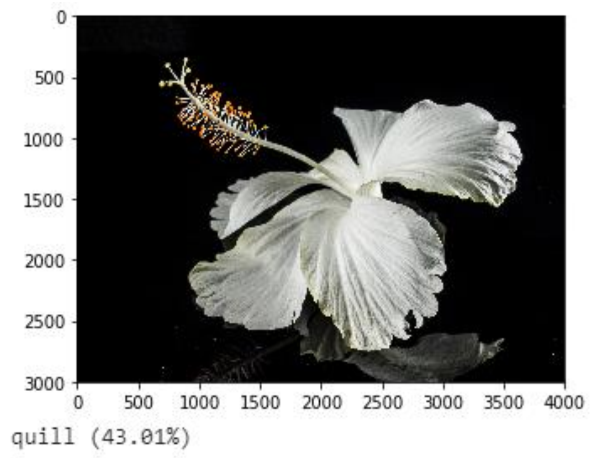
```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

(source: https://keras.io/applications/)

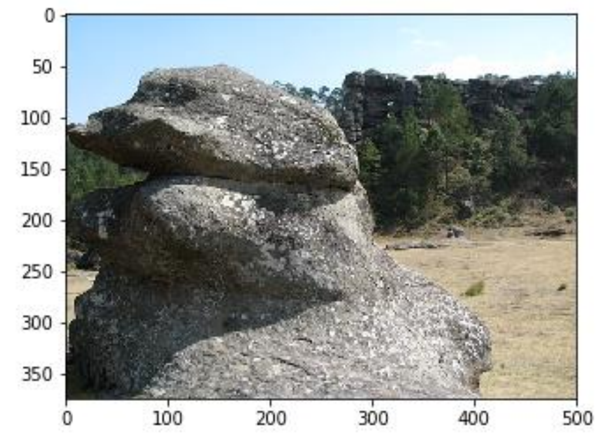
We constructed a similar implementation but found that the pretrained VGG16 doesn't perform well in our case. It can detect rock, fungus and umbrella pretty well but not flower. This preliminary test could probably indicate that the 'imagenet' setup did not perform heavy training on different types of flowers. But still, using Keras VGG16 package with 'imagenet' setup could become a lightweight solution in other cases.



```
cabbage_butterfly (41.03%)
```



```
quill (43.01%)
```

 megalith (99.29%)	thatch (71.90%)

With the failure of this preliminary test, we will now try to implement our customized VGG model and train with our own dataset.

Experiment 1 - The VGG16 with 2 output channels:
As mentioned above, the original VGG16 model was built in such a way that it can differentiate among 1000 classes. But in our scenario, our goal is to differentiate between only 2 simple cases: flower vs. non-flower.

In the first round of experiments, we implemented the exact structure of VGG16 except the last layer, which now changed to num_classes = 2. The implementation together with training and testing are showing in - VGG_Model_Train_Test_baseline.ipynb (VGG_BL_Model).

Original VGG16 - Last 3 FC layers and Params	Customized VGG16 - Last 3 FC layers and Params

`fc1 (Dense) (None, 4096) 102764544`
`________________________________________________________________`
`fc2 (Dense) (None, 4096) 16781312`
`________________________________________________________________`
`predictions (Dense) (None, 1000) 4097000`
`================================================================`
`Total params: 138,357,544`
`Trainable params: 138,357,544`
`Non-trainable params: 0`
	dense_22 (Dense) (None, 4096) 102764544
	____
	dense_23 (Dense) (None, 4096) 16781312
	____
	dense_24 (Dense) (None, 2) 8194
	==
	Total params: 134,268,738
	Trainable params: 134,268,738

	Non-trainable params: 0

Soon after we implemented the VGG_BL_Model, we encountered a problem that no matter how we tuned the model parameters, the test accuracy would never change and always stayed at 56.10% (Shown in VGG_BL_Model). We noticed that in the testing set there are 538 flowers and 421 non-flowers, which means the amount of flowers pictures is around 56.10% of total testing dataset. After we printed out some testing samples and compared those with the model predictions, we found that the VGG_BL_Model classified all testing image as flower. This indicated that VGG_BL_Model did not 'learn' any meaningful features from the training dataset.

At first we doubted it could be the data problem. We then verified this hypothesis by feeding the same data to our very initial baseline model, and it turned out that the baseline model was able to achieve a reasonable accuracy, which was higher than 80%. Thus, we turned to suspect that it was the VGG_BL_Model structure that caused failure of learning.

Experiment 2 - The VGG Model with 9 Weight Layer:

One of the possible root causes could be the vanishing gradient problem. We noticed that with baseline setup, we are more likely (100% till now) to get a well trained model with reasonable accuracy. In addition, the researchers built VGG16 also started to train shallower net - Configuration A (VGG11) first to avoid the stall-learning problem (Simonyan & Zisserman, 2015). Thus, we decided to take out some hidden layers for further exploration.

At first, we took out the last 2 Conv2d+MaxPooling blocks plus 1 256-Conv2d layer. This made the VGG_BL_Model left with only 9 weight layers: 2 64-Conv2d layers, 2 128-Conv2d layers, 2 256-Conv2d layers and 3 Dense layers . While testing the new model, we realized that by continue applying 4096 output shape at dense layer can easily max-up RAM usage. We then lowered the output shape to 1024. This model is reflected in VGG_Model_Train_Test_Mod1.ipynb (VGG_Mod1).

Structure of VGG_Mod1:

```
Layer (type)               Output Shape            Param #
=================================================================
input_3 (InputLayer)       [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d_12 (Conv2D)         (None, 224, 224, 64)      1792
_________________________________________________________________
conv2d_13 (Conv2D)         (None, 224, 224, 64)      36928
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 112, 112, 64)      0
_________________________________________________________________
```

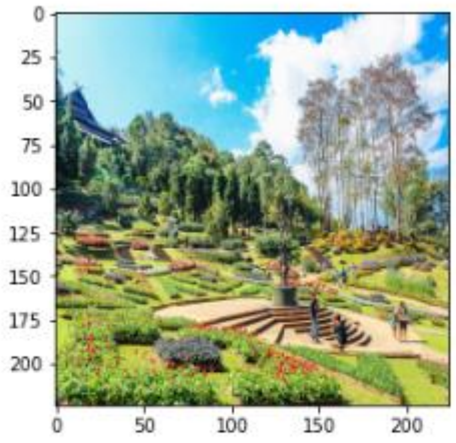
```
conv2d_14 (Conv2D)         (None, 112, 112, 128)    73856
_________________________________________________________________
conv2d_15 (Conv2D)         (None, 112, 112, 128)    147584
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 56, 56, 128)      0
_________________________________________________________________
conv2d_16 (Conv2D)         (None, 56, 56, 256)      295168
_________________________________________________________________
conv2d_17 (Conv2D)         (None, 56, 56, 256)      590080
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 28, 28, 256)      0
_________________________________________________________________
flatten_2 (Flatten)        (None, 200704)           0
_________________________________________________________________
dense_6 (Dense)            (None, 1024)             205521920
_________________________________________________________________
dropout_4 (Dropout)        (None, 1024)             0
_________________________________________________________________
dense_7 (Dense)            (None, 1024)             1049600
_________________________________________________________________
dropout_5 (Dropout)        (None, 1024)             0
_________________________________________________________________
dense_8 (Dense)            (None, 2)                2050
```

The first few rounds with VGG_Mod1 setup met our expectations. The model was able to achieve testing accuracy higher than 92% and the last validation test score was 91.45% (Shown in VGG_Mod1). This showed around 5%-6% improvement from our baseline model. Moreover, while we printed out those mis-classified pictures, we found that some images were in fact quite confusing also. Thus, we believe by further fine pre-processing the input data, the model can achieve even a higher accuracy.

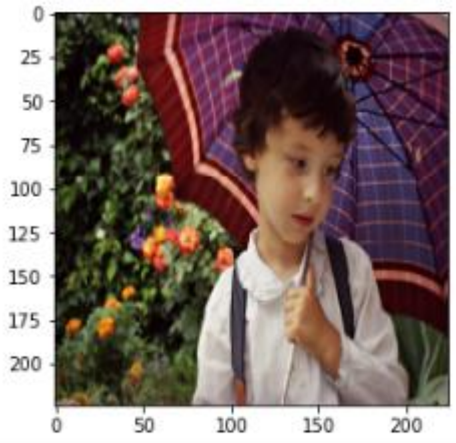
Mis-classified samples:

Flowers are too small to detect	There are flowers in the background

Showing 374 index image
Predicted as 0 but is actually 1



Showing 622 index image
Predicted as 1 but is actually 0



While fine tuning parameters, we again faced the same problem in the later runs. This could mean that by cutting down the layers , the VGG_Mod1 could still be too deep to be trained efficiently. Since we had obtained a structure: VGG_Mod1 that could generate a satisfiable result, we now turned to explore possible solutions to overcome this problem.

Experiment 3 - The Progressive Learning Approach:

The first solution we tried to implement was inspired by Simonyan and Zisserman in their essay - *VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION.* We started by training a much shallower net (Base_VGG) and stored its weight for later use. Then we loaded its weight to VGG_Mod1 by matching the names of layers in previous model and initialized the rest using 'he_normal'. The approach is shown in VGG_Model_Train_Test_Mod2.ipynb.

Structure of Base_VGG:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d_1_1 (Conv2D)          (None, 224, 224, 64)      1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 64)      0
_________________________________________________________________
conv2d_2_1 (Conv2D)          (None, 112, 112, 128)     73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_3_1 (Conv2D)          (None, 56, 56, 256)       295168
```

```
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
flatten (Flatten)            (None, 200704)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              205521920
_________________________________________________________________
dropout (Dropout)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 2)                 2050
```

In order to match and load layer weight to VGG_Mod1, specific names were assigned to Con2d layers: conv2d_1_1, conv2d_2_1 and conv2d_3_1. We applied the same training strategy but only with 20 epoch, which was enough for Base_VGG to generate appropriate weights. After training was completed, the testing accuracy of Base_VGG managed to reach stable 0.8571 (Shown in VGG_Model_Train_Test_Mod2.ipynb).

We then loaded the Base_VGG weights into VGG_Mod1 and started training. In the first round of epoch, testing accuracy was able to hit 0.7351, comparing to 0.5474 which was trained from scratch. In the end, the progressively trained VGG_Mod1 was able to reach a prediction accuracy of 90.62%.

However, the stall-learning problem was not solved by this method. In later rounds of parameter tuning, we encountered the same problem showing testing accuracy was stunted at 56.10%. But still, the progressive approach has a remarkable advantage that is it shortens the time taken to obtain suitable weights for later initialization.

Experiment 4 - The Batch Normalization Approach:

Lastly, we tried to resolve this problem by applying the classic solution - Batch Normalization. The implementation is straight forward. Before activating the Conv2d layers, we firstly did a batch normalization on the input values. This is shown in VGG_Model_Train_Test_Mod3.ipynb (VGG_Mod3).

In the first batch of runnings, VGG_Mod3 managed to reach an accuracy of 88.43% which is considerably lower than VGG_Mod1. Moreover, the training time was significantly longer and testing accuracy fluctuated a lot along the training process. But most importantly, the stall-learning problem did not show up again. This made VGG_Mod3 a solid model for us to further tune on.

In the following rounds, we increased the number of epochs from 120 to 150 so that the model could be trained fully. In comparison, training rate in later stage with 150 epoch could reach 94-95%, whereas with 120 epoch it could only hit around 91%. Accordingly, we postponed the learning rate decay timeline:

	1st Decay	2nd Decay	3rd Decay
Original	71 epoch	90 epoch	100 epoch
Latest	91 epoch	120 epoch	140 epoch

Throughout the experiments, we noticed that by giving a higher starting learning rate (say 0.01 or 0.005) could cause the stall-learning problem. Thus, we fixed our starting rate at 1e-3 and decay at the above mentioned rate.

While we were training with dataset - Data_V1, more images were added to the data pool and formed another dataset - Data_V2. Our team tried to feed Data_V2 to VGG_Mod3 but then realized that training was very difficult to continue with Data_V2. The significant increase in input size made each epoch to take 8 minutes to complete. To shorten the training time, we tried to increase batch size from 32 to 48/64, but RAM (doubled to 25G) on colab server was not able to handle. On the other hand, by cutting down the number of epoch was also not effective in terms of getting a good accuracy.

In the end, VGG_Mod3 (with Data_V1) was able to reach **92.18%** accuracy. Although the training time of VGG_Mod3 was significantly longer than previous versions, it reached the highest prediction accuracy. Most importantly, it overcame the stall-learning problem faced by previous experiments.

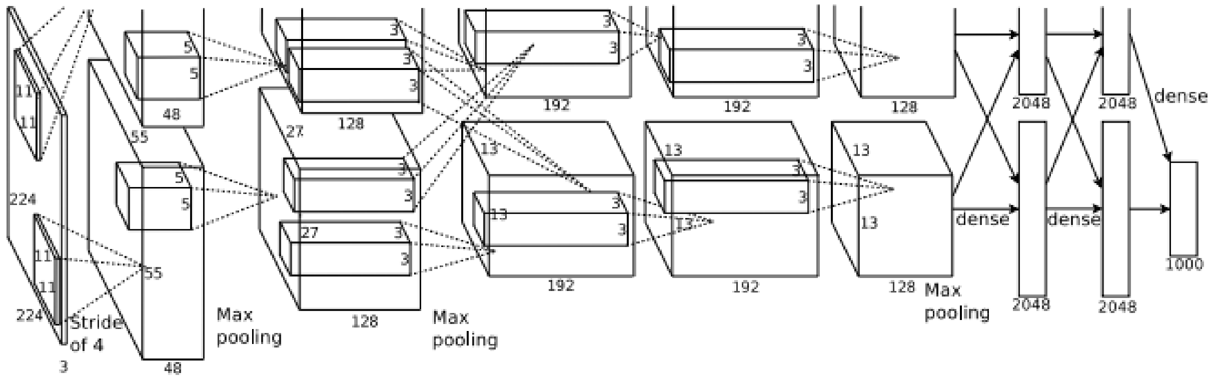
VGG Performance Review:

Param\Model	VGG_BL_Model	VGG_Mod1	Progressive	VGG_Mod3
Accuracy	56.10%	91.45%	90.62%	**92.18%**
Time/Epoch	~64s	50-67s	**38-54s**	~120s
Stall-Learning	Always	Very Likely	Very Likely	**None**

6. Approach Towards AlexNet Structure

Data Set:
Data (images) used for this approach are the same as what we used in VGG16 approach. All input images are resized to a standard 224*224 pixels shape. We also did a 80-20 split for training and testing respectively.

AlexNet Architecture:



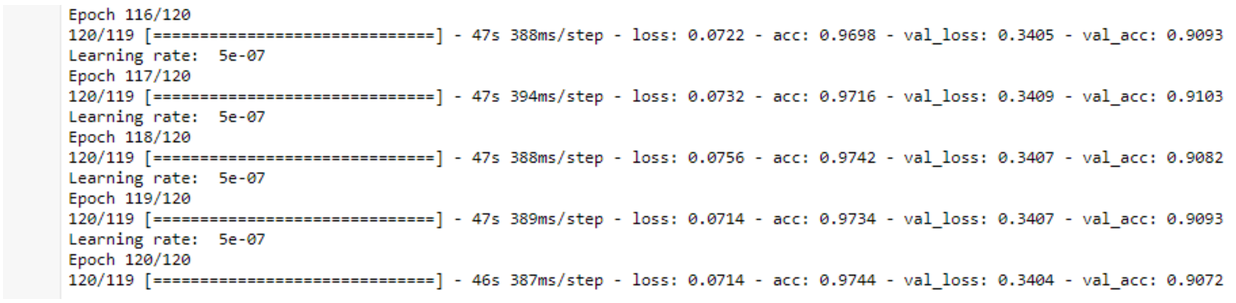
(source: https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5)

The AlexNet came to our attention that although it was introduced in earlier years (2012) than VGG (2015), it has less layers than its subsequent CNN models. In 2012, AlexNet was constructed and trained using 2 GPUs, and thus gave its parallel structure as shown above. The AlexNet has total 8 weight layers, of which the first 5 are the Conv2d layers and later 3 are Dense layers. Although introduced in 2012, the model had applied some of the commonly used techniques such as data augmentation (cropping 256*256 to 224*224), ReLU activation and dropouts. Besides, “Overlapping-pooling” (MaxPooling with kernel size = (3,3) and strides = (2,2)) was used to reduce the “top-1 and top-5 error rates by 0.4% and 0.3%, respectively” (Krizhevsky, Sutskever & Hinton, 2012).

Experiment 1 - The AlexNet:
We have implemented the exact AlexNet structure based on our understanding. On top of that we patched batch normalization layers (to avoid stall-learning alike problem) followed by activation. This is shown in AlexNet_Model_Train_Test.ipynb (AlexNet_Mod)

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 56, 56, 96)        34944
_________________________________________________________________
batch_normalization (BatchNo (None, 56, 56, 96)        384
_________________________________________________________________
activation (Activation)      (None, 56, 56, 96)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 27, 27, 96)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 27, 27, 256)       614656
_________________________________________________________________
batch_normalization_1 (Batch (None, 27, 27, 256)       1024
_________________________________________________________________
activation_1 (Activation)    (None, 27, 27, 256)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 13, 384)       885120
_________________________________________________________________
batch_normalization_2 (Batch (None, 13, 13, 384)       1536
_________________________________________________________________
activation_2 (Activation)    (None, 13, 13, 384)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 13, 384)       1327488
_________________________________________________________________
batch_normalization_3 (Batch (None, 13, 13, 384)       1536
_________________________________________________________________
activation_3 (Activation)    (None, 13, 13, 384)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 256)       884992
_________________________________________________________________
batch_normalization_4 (Batch (None, 13, 13, 256)       1024
_________________________________________________________________
activation_4 (Activation)    (None, 13, 13, 256)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 4096)              37752832
_________________________________________________________________
dropout (Dropout)            (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 8194
```

In the first few rounds of experiments, the AlexNet model was able to achieve a 89.68% accuracy and we did not encounter the stall-learning problem along the progress. Then, we tried to extend epoch runs from 120 to 150 but the accuracy did not increase significantly. Although the training accuracy could hike as high as 97%, the testing accuracy could only reach around 90-91%. This could indicate that model was not able to generalize a more comprehensive set of features using existing training data. This is shown in AlexNet_Model_Train_Test.ipynb.



```
Epoch 116/120
120/119 [==============================] - 47s 388ms/step - loss: 0.0722 - acc: 0.9698 - val_loss: 0.3405 - val_acc: 0.9093
Learning rate:  5e-07
Epoch 117/120
120/119 [==============================] - 47s 394ms/step - loss: 0.0732 - acc: 0.9716 - val_loss: 0.3409 - val_acc: 0.9103
Learning rate:  5e-07
Epoch 118/120
120/119 [==============================] - 47s 388ms/step - loss: 0.0756 - acc: 0.9742 - val_loss: 0.3407 - val_acc: 0.9082
Learning rate:  5e-07
Epoch 119/120
120/119 [==============================] - 47s 389ms/step - loss: 0.0714 - acc: 0.9734 - val_loss: 0.3407 - val_acc: 0.9093
Learning rate:  5e-07
Epoch 120/120
120/119 [==============================] - 46s 387ms/step - loss: 0.0714 - acc: 0.9744 - val_loss: 0.3404 - val_acc: 0.9072
```

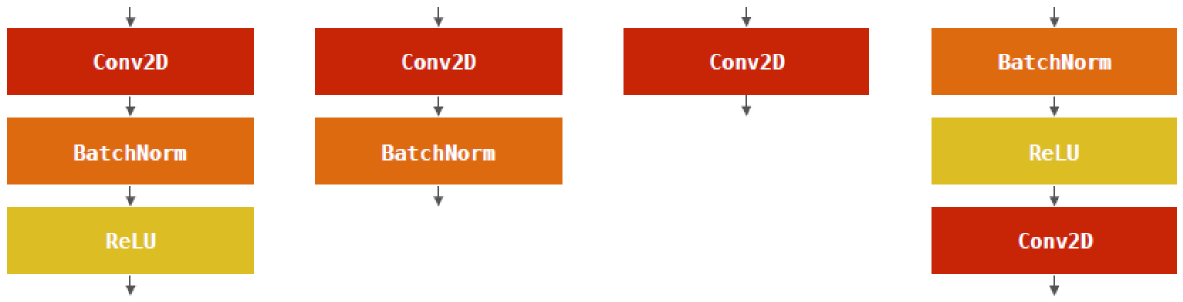
With this idea in mind, we tried to feed a more refined and extended data set on top of Data_V1. As shown in the result, the testing accuracy reached **91%** this time. This could mean that AlexNet has the potential to reach a higher accuracy given more fine-processed and more comprehensive set of training data. The result is shown in AlexNet_Model_Train_Test_2.ipynb.

```
Best accuracy (on testing dataset): 91.00%
              precision    recall  f1-score   support

      flower     0.8943    0.8943    0.8943       454
  non-flower     0.9217    0.9217    0.9217       613

    accuracy                         0.9100      1067
   macro avg     0.9080    0.9080    0.9080      1067
weighted avg     0.9100    0.9100    0.9100      1067

[[406  48]
 [ 48 565]]
```

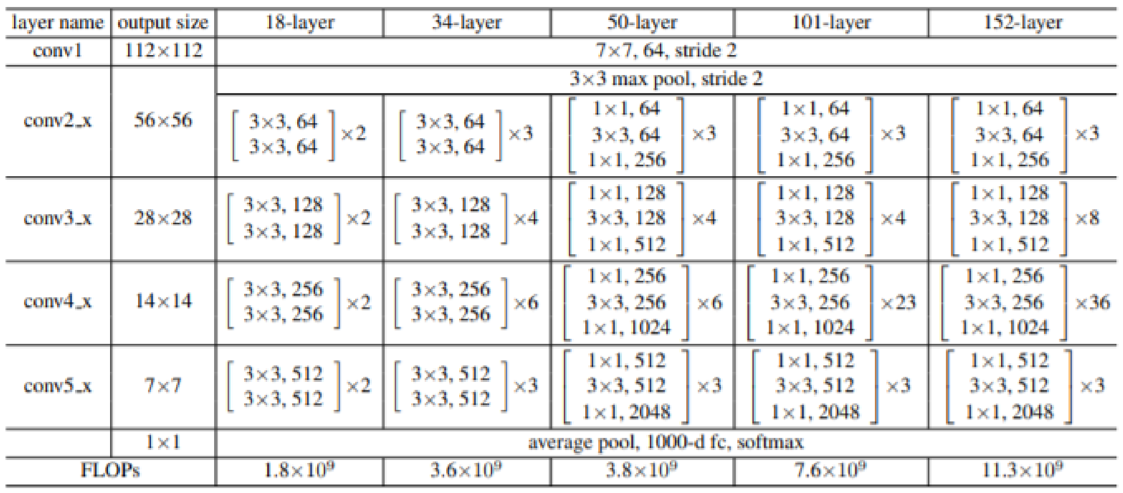
7. Approach Towards ResNet Structure

Data Set:
The first version data set (Data_V1) was used for training and testing initially, however the second version data set (Data_V2) was used in the end.

ResNet Architecture:
The Residual Network architecture as implemented in the class was utilized and consists of the following building blocks:



The full ResNet architecture is as follows:

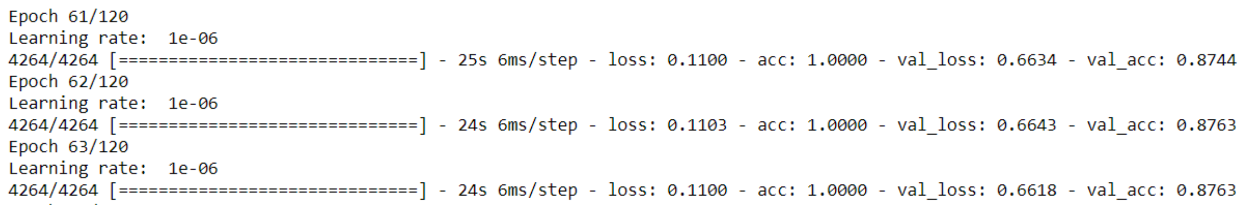


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\left[\begin{array}{c}3\times3, 64\\3\times3, 64\end{array}\right]\times2$	$\left[\begin{array}{c}3\times3, 64\\3\times3, 64\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 64\\3\times3, 64\\1\times1, 256\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 64\\3\times3, 64\\1\times1, 256\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 64\\3\times3, 64\\1\times1, 256\end{array}\right]\times3$
conv3_x	28×28	$\left[\begin{array}{c}3\times3, 128\\3\times3, 128\end{array}\right]\times2$	$\left[\begin{array}{c}3\times3, 128\\3\times3, 128\end{array}\right]\times4$	$\left[\begin{array}{c}1\times1, 128\\3\times3, 128\\1\times1, 512\end{array}\right]\times4$	$\left[\begin{array}{c}1\times1, 128\\3\times3, 128\\1\times1, 512\end{array}\right]\times4$	$\left[\begin{array}{c}1\times1, 128\\3\times3, 128\\1\times1, 512\end{array}\right]\times8$
conv4_x	14×14	$\left[\begin{array}{c}3\times3, 256\\3\times3, 256\end{array}\right]\times2$	$\left[\begin{array}{c}3\times3, 256\\3\times3, 256\end{array}\right]\times6$	$\left[\begin{array}{c}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{array}\right]\times6$	$\left[\begin{array}{c}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{array}\right]\times23$	$\left[\begin{array}{c}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{array}\right]\times36$
conv5_x	7×7	$\left[\begin{array}{c}3\times3, 512\\3\times3, 512\end{array}\right]\times2$	$\left[\begin{array}{c}3\times3, 512\\3\times3, 512\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{array}\right]\times3$	$\left[\begin{array}{c}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{array}\right]\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Experiment 1 - in-class ResNet Architecture:
We used the in-class implementation of the ResNet architecture because our classification problem (2 classes) is significantly simpler than the classification problem of the original ResNet (ImageNet dataset of >20,000 classes). This implementation had 3 layers of residual block (v1) with 16, 32 & 64

filters respectively with downsampling on the second and third layers, followed by an average pooling with pool size 8.

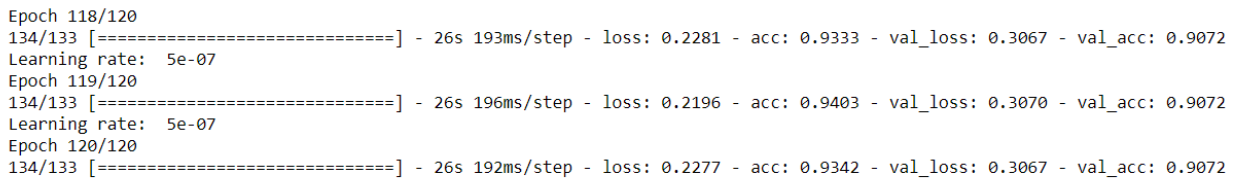
However, it was found that the in-class implementation of ResNet was already overfitting our classification problem (shown by **100%** training accuracy and **87.63%** test accuracy):



```
Epoch 61/120
Learning rate:  1e-06
4264/4264 [==============================] - 25s 6ms/step - loss: 0.1100 - acc: 1.0000 - val_loss: 0.6634 - val_acc: 0.8744
Epoch 62/120
Learning rate:  1e-06
4264/4264 [==============================] - 24s 6ms/step - loss: 0.1103 - acc: 1.0000 - val_loss: 0.6643 - val_acc: 0.8763
Epoch 63/120
Learning rate:  1e-06
4264/4264 [==============================] - 24s 6ms/step - loss: 0.1100 - acc: 1.0000 - val_loss: 0.6618 - val_acc: 0.8763
```

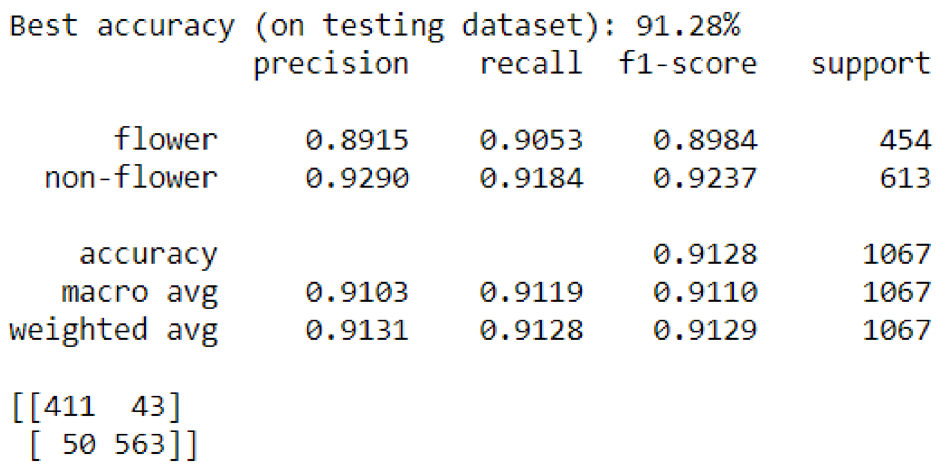
Hence, the training was cut short as it was inadequate.

Experiment 2 - in-class ResNet Architecture with Dropout, Data Augmentation:
To counter the overfitting, dropout layers of various values (0.20, 0.25 & 0.30) was added in between the residual block (v1) layers and average pooling layers. Also the data augmentation techniques taught in class were utilized (horizontal/vertical shift, horizontal flip, rotation). These additions allowed the Neural Network model to generalize more effectively on the test dataset:

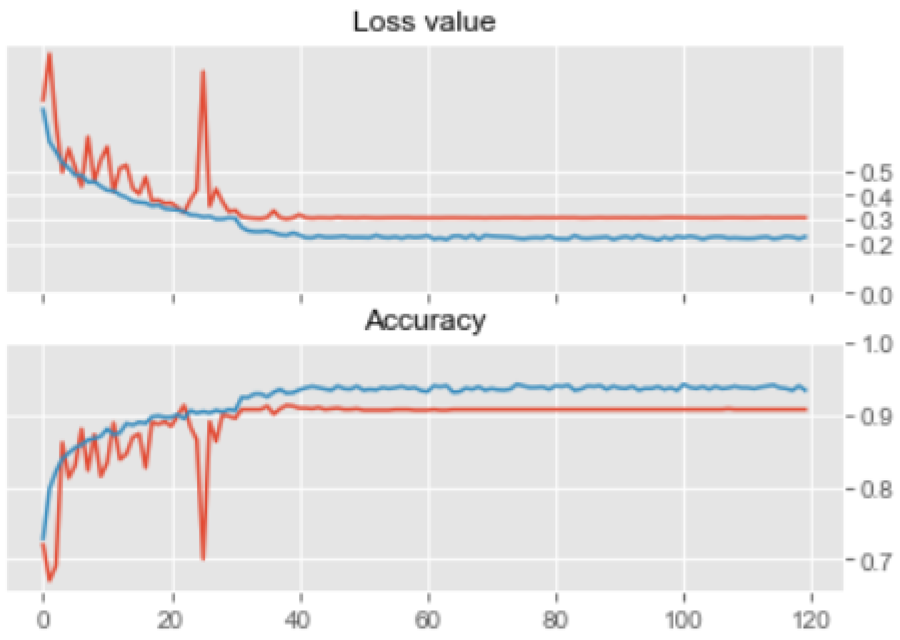


```
Epoch 118/120
134/133 [==============================] - 26s 193ms/step - loss: 0.2281 - acc: 0.9333 - val_loss: 0.3067 - val_acc: 0.9072
Learning rate:  5e-07
Epoch 119/120
134/133 [==============================] - 26s 196ms/step - loss: 0.2196 - acc: 0.9403 - val_loss: 0.3070 - val_acc: 0.9072
Learning rate:  5e-07
Epoch 120/120
134/133 [==============================] - 26s 192ms/step - loss: 0.2277 - acc: 0.9342 - val_loss: 0.3067 - val_acc: 0.9072
```

Training was allowed to complete and the best accuracy obtained was **91.28%**:



```
Best accuracy (on testing dataset): 91.28%
              precision    recall  f1-score   support

      flower     0.8915    0.9053    0.8984       454
  non-flower     0.9290    0.9184    0.9237       613

    accuracy                         0.9128      1067
   macro avg     0.9103    0.9119    0.9110      1067
weighted avg     0.9131    0.9128    0.9129      1067

[[411  43]
 [ 50 563]]
```



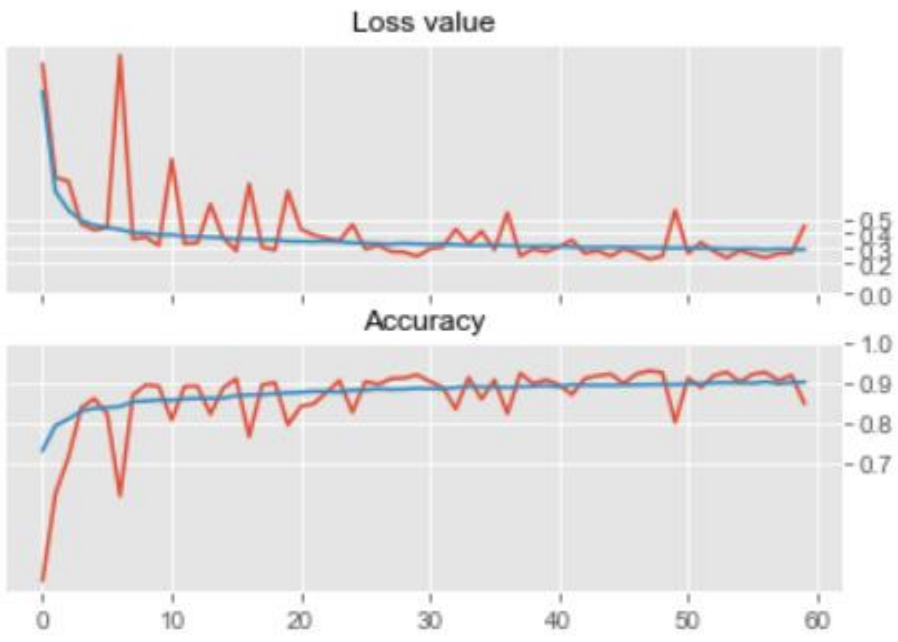
Experiment 3 - in-class ResNet Architecture with Dropout, Data Augmentation & Added Layers:
As the training accuracy still had room to improve, layers were added to increase the complexity of the model and hopefully increase it's prediction power. However, the model was unable to reach an accuracy above **91.28%**.

Experiment 4 - in-class ResNet Architecture with Dropout, Data Augmentation, Added Layers & RMSProp:
The next experiment was to change the optimization algorithm from ADAM to RMSProp. This enabled a marginal improvement of accuracy from **91.28%** to **92.41%**. This is likely because in this particular case RMSProp converges slightly more quickly to the minima than ADAM, hence in 120 epochs it is able to get slightly higher accuracy.

Experiment 5 - in-class ResNet Architecture with Dropout, Data Augmentation, Added Layers, RMSProp & Added Data:
The next method explored was to increase the amount observed by the Neural Network model in the training dataset, resulting in an additional 13341 images obtained (Data_V2). The 2 fold increase in the dataset enabled the model to achieve a higher accuracy even with a reduced number of epochs (60 epochs vs 120 epochs).

Training was allowed to complete and the best accuracy obtained was **92.97%**:

```
Best accuracy (on testing dataset): 92.97%
              precision    recall  f1-score   support

  non-flower     0.8891    0.9371    0.9125      1446
      flower     0.9581    0.9249    0.9412      2250

    accuracy                         0.9297      3696
   macro avg     0.9236    0.9310    0.9268      3696
weighted avg     0.9311    0.9297    0.9300      3696

[[1355   91]
 [ 169 2081]]
```



It is likely that this is the most that the in-class ResNet can be pushed to, hence more advanced Neural Network architectures were explored subsequently (Inception V4, Inception-ResNet V2 & SE Inception-ResNet V2).

8. Approach Towards Inception V4 Structure

Data Set:
The first version data set (Data_V2) was used for training and testing as it was deemed as more comprehensive than the first version data set (Data_V1).

Inception V4 Architecture:
Inception V4 has a more simplified block structure than its predecessors and in the original form has more layers. It consists of the following original components -

Stem block:

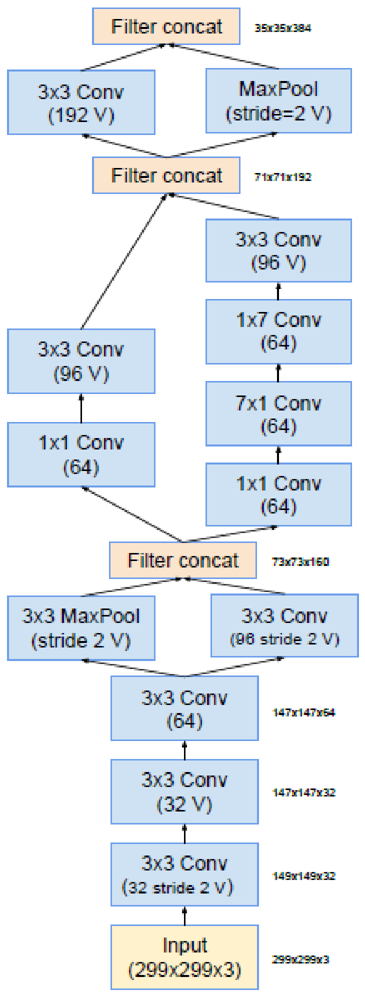


Figure 3. The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks. Cf. Figures 9 and 15

Inception A block:

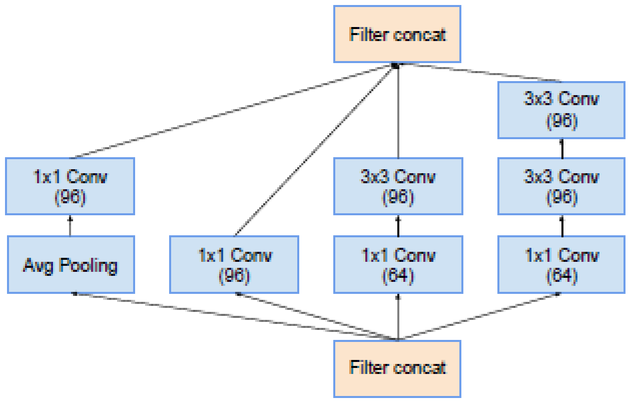


Figure 4. The schema for 35 × 35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

Inception B block:

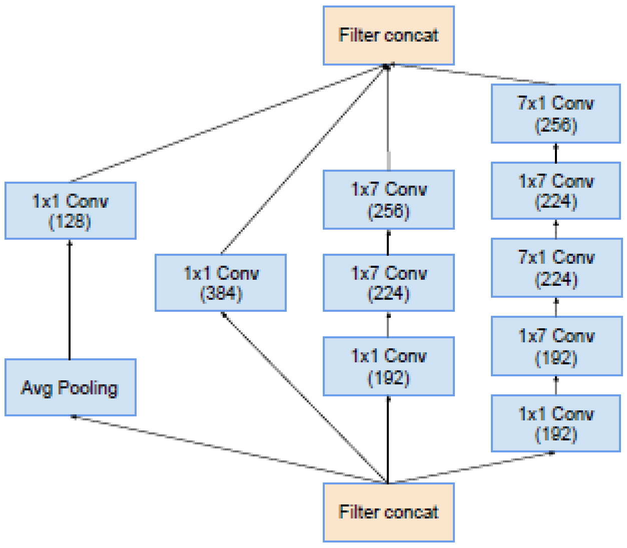


Figure 5. The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

Inception C block:

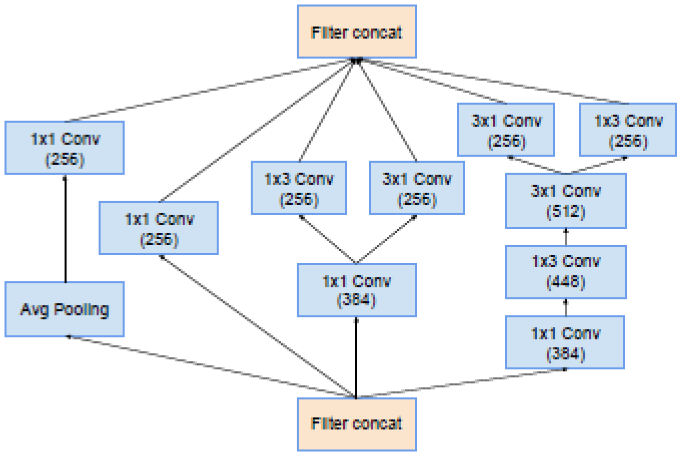


Figure 6. The schema for 8 × 8 grid modules of the pure Inception-v4 network. This is the Inception-C block of Figure 9.

Inception Reduction A block:

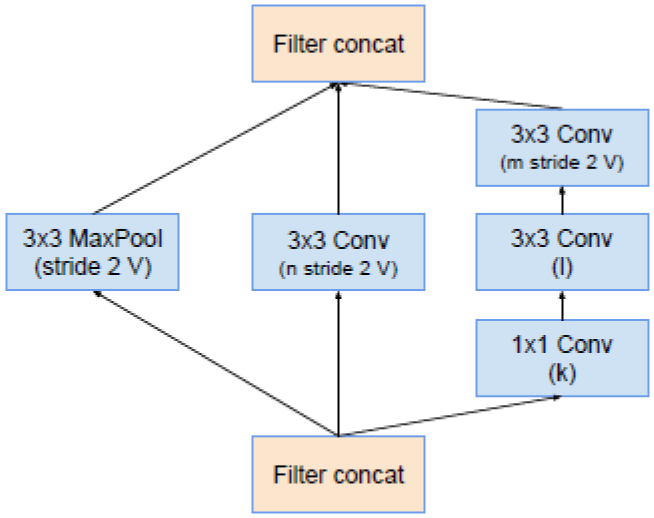


Figure 7. The schema for 35 × 35 to 17 × 17 reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1.

Network	k	l	m	n
Inception-v4	192	224	256	384
Inception-ResNet-v1	192	192	256	384
Inception-ResNet-v2	256	256	384	384

Table 1. The number of filters of the Reduction-A module for the three Inception variants presented in this paper. The four numbers in the colums of the paper parametrize the four convolutions of Figure 7

Inception Reduction B block:

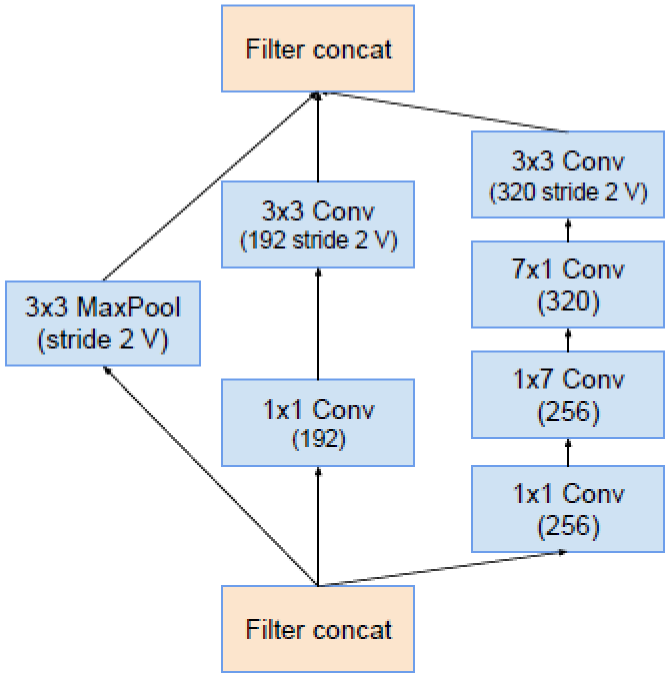


Figure 8. The schema for 17×17 to 8×8 grid-reduction module. This is the reduction module used by the pure Inception-v4 network in Figure 9.

Original Inception V4 architecture:

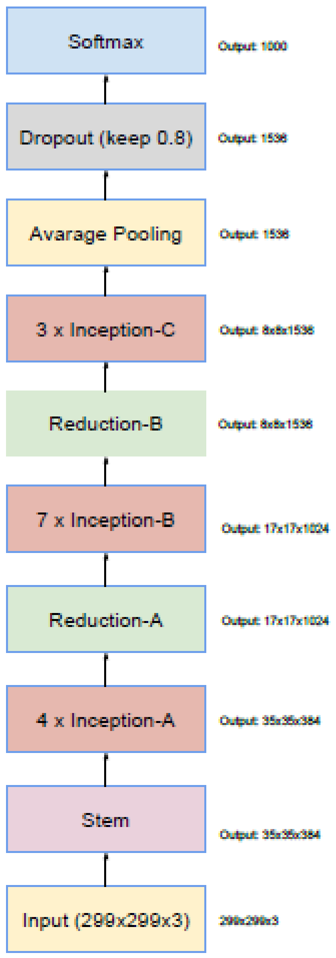


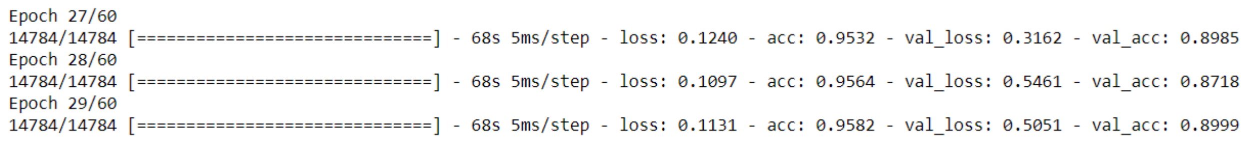
Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

Experiment 1 - Original Inception V4:

We have implemented the exact Inception V4 structure of the paper based on our understanding. However, a zero padding of (0,1), (0,1) was added and average padding was modified to (1,1) due to a size mismatch between the original dataset (299, 299, 3) and our dataset (96, 96, 3).

In the experiment, the number of layers were too extensive for the classification problem and we were faced with the vanishing gradient problem (gradients go to 0 or infinity). This resulted in accuracy values of 60.88% and 39.12% which corresponds to predicting all as flowers or all as non-flowers respectively.

Experiment 2 - Inception V4 with Batch Normalization & halving kernel filters/ pool sizes:
To advert the vanishing gradient problem 2 methods were explored, 1) BatchNormalization layers were added in between each block (A, B, C) and 2) the kernel filters/ pool sizes were halved (e.g. 128 to 64). This enabled us to avoid the vanishing gradient problem, however the model was still too complicated and overfitted the dataset:



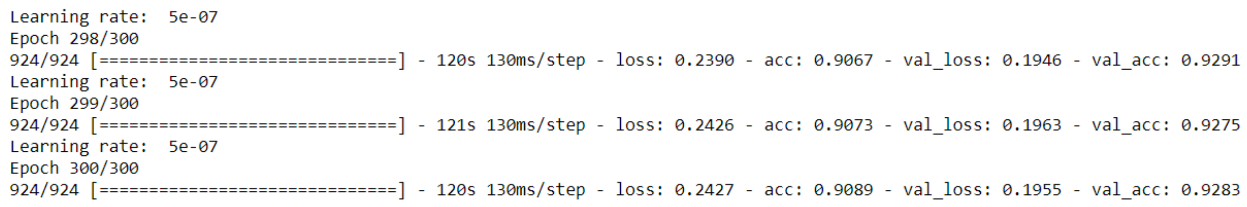
```
Epoch 27/60
14784/14784 [==============================] - 68s 5ms/step - loss: 0.1240 - acc: 0.9532 - val_loss: 0.3162 - val_acc: 0.8985
Epoch 28/60
14784/14784 [==============================] - 68s 5ms/step - loss: 0.1097 - acc: 0.9564 - val_loss: 0.5461 - val_acc: 0.8718
Epoch 29/60
14784/14784 [==============================] - 68s 5ms/step - loss: 0.1131 - acc: 0.9582 - val_loss: 0.5051 - val_acc: 0.8999
```

Hence, the training was cut short as it was inadequate.

Experiment 3 - Inception V4 with Batch Normalization, halving kernel filters/ pool sizes, Data Augmentation, RMSProp:
With data augmentation (horizontal/vertical shift, horizontal flip, rotation) the model generalized better but did not manage to get accuracy above **92.41%**. This was even when modifying the optimizer to RMSProp

Experiment 4 - Inception V4 with Batch Normalization, halving kernel filters/ pool sizes, Data Augmentation, RMSProp, modifying learning rate, reduced steps/epoch and added epochs:
To improve test accuracy, more epochs were added and the batch size was reduced to 16. This would mean that training would be more cumbersome, requiring long training times but it would potentially improve the test accuracy. We also modified the learning rate to reduce more slowly as there were now 300 epochs instead of 60 or 120.

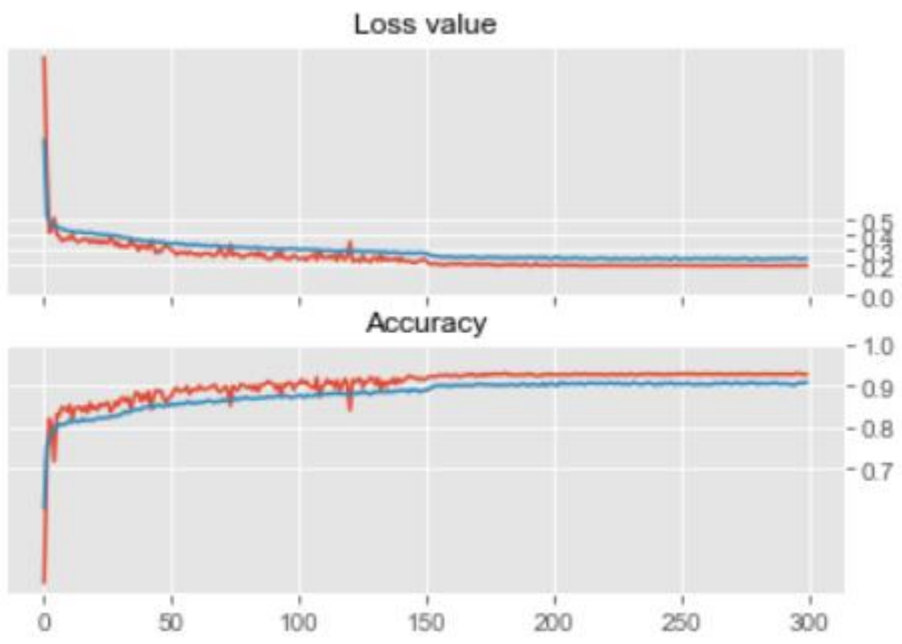
The experiment was a success as the test accuracy improved and the model was allowed to train to completion:



```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 120s 130ms/step - loss: 0.2390 - acc: 0.9067 - val_loss: 0.1946 - val_acc: 0.9291
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 121s 130ms/step - loss: 0.2426 - acc: 0.9073 - val_loss: 0.1963 - val_acc: 0.9275
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 120s 130ms/step - loss: 0.2427 - acc: 0.9089 - val_loss: 0.1955 - val_acc: 0.9283
```

The best accuracy obtained was **93.07%**:

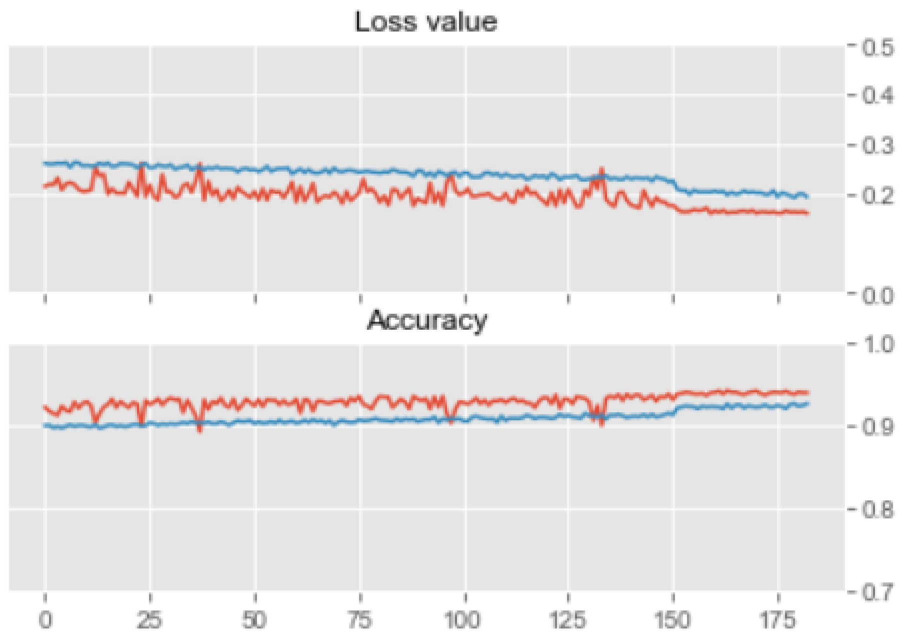
```
Best accuracy (on testing dataset): 93.07%
              precision    recall  f1-score   support

  non-flower     0.8946    0.9329    0.9133      1446
      flower     0.9557    0.9293    0.9423      2250

    accuracy                         0.9307      3696
   macro avg     0.9251    0.9311    0.9278      3696
weighted avg     0.9318    0.9307    0.9310      3696

[[1349   97]
 [ 159 2091]]
```

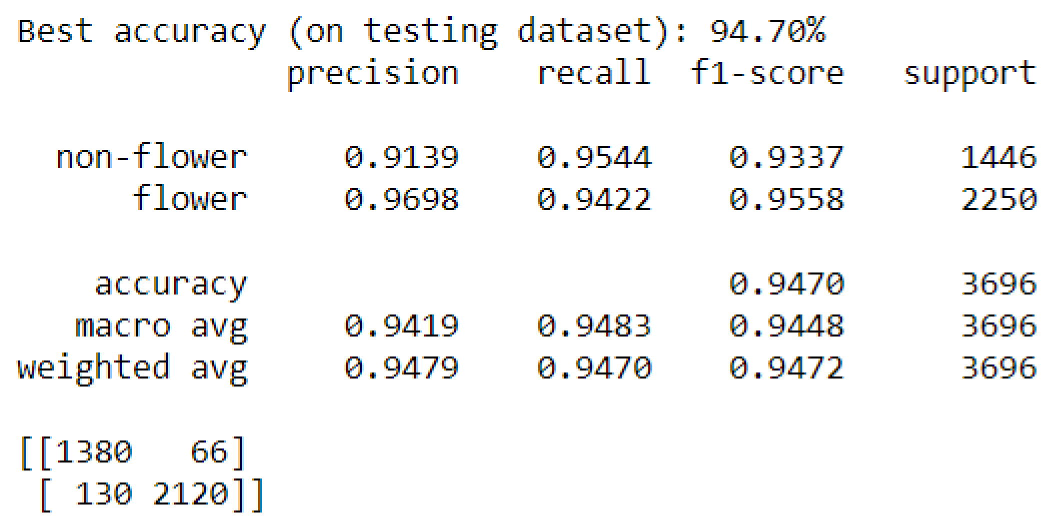


Experiment 4 - latest Inception V4 settings, pre-saved weights for another 300x2 epochs:
As it was noticed that the training accuracy had not peaked close to ~99%, it was deemed that the model could possibly learn more by having increased epochs. Hence the saved weights for the previous best model (by accuracy, **93.07%**) was used as a starting point for another 300x2 epochs of training.

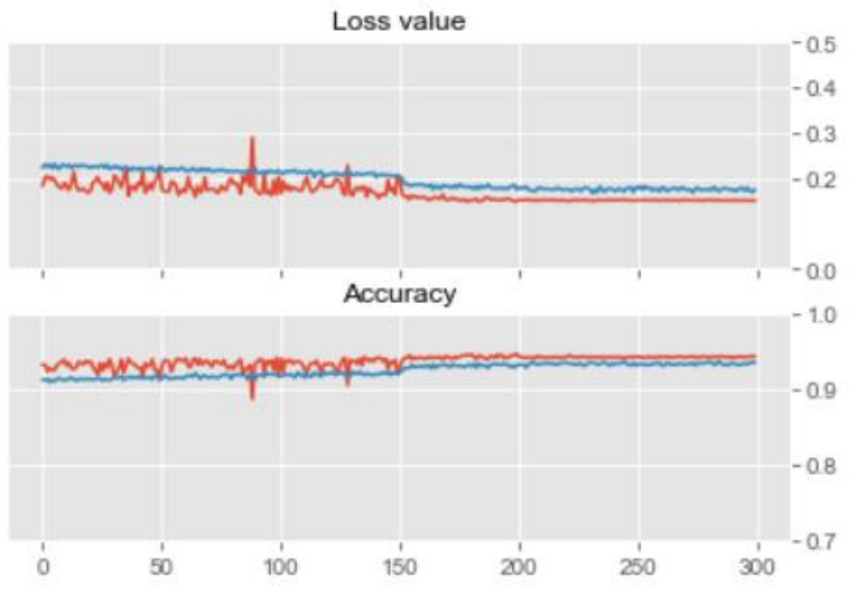
This enabled us to increase the test accuracy to **94.18%** and subsequently **94.70%**:

```
Best accuracy (on testing dataset): 94.18%
              precision    recall  f1-score   support

  non-flower     0.9047    0.9516    0.9275      1446
      flower     0.9678    0.9356    0.9514      2250

    accuracy                         0.9418      3696
   macro avg     0.9362    0.9436    0.9395      3696
weighted avg     0.9431    0.9418    0.9421      3696

[[1376   70]
 [ 145 2105]]
```





```
Best accuracy (on testing dataset): 94.70%
              precision    recall  f1-score   support

  non-flower     0.9139    0.9544    0.9337      1446
      flower     0.9698    0.9422    0.9558      2250

    accuracy                         0.9470      3696
   macro avg     0.9419    0.9483    0.9448      3696
weighted avg     0.9479    0.9470    0.9472      3696

[[1380   66]
 [ 130 2120]]
```



Although the training accuracy still had not peaked close to ~99%, due to a lack of time, we did not continue this method of training for more epochs:

```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 117s 127ms/step - loss: 0.1711 - acc: 0.9342 - val_loss: 0.1500 - val_acc: 0.9435
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 120s 129ms/step - loss: 0.1689 - acc: 0.9364 - val_loss: 0.1498 - val_acc: 0.9435
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 120s 130ms/step - loss: 0.1734 - acc: 0.9348 - val_loss: 0.1500 - val_acc: 0.9435
```

Hence as of its current iteration, our implementation of the Inception-V4 model has **94.70%** test accuracy.

9. Approach Towards Inception-ResNet V2 Structure

Data Set:

The first version data set (Data_V2) was used for training and testing as it was deemed as more comprehensive than the first version data set (Data_V1).

Inception-ResNet V2 Architecture:

Inception-ResNet architecture was created to combine the multiple deep layers of Inception with the Residual layers of ResNet useful for training deep layers. It was mentioned by the authors that both Inception V4 and Inception-ResNet perform similarly well, with Inception-ResNet being slightly more efficient. It consists of the following original components -

Stem block: (same as Inception V4)

Inception-ResNet V2 A block:

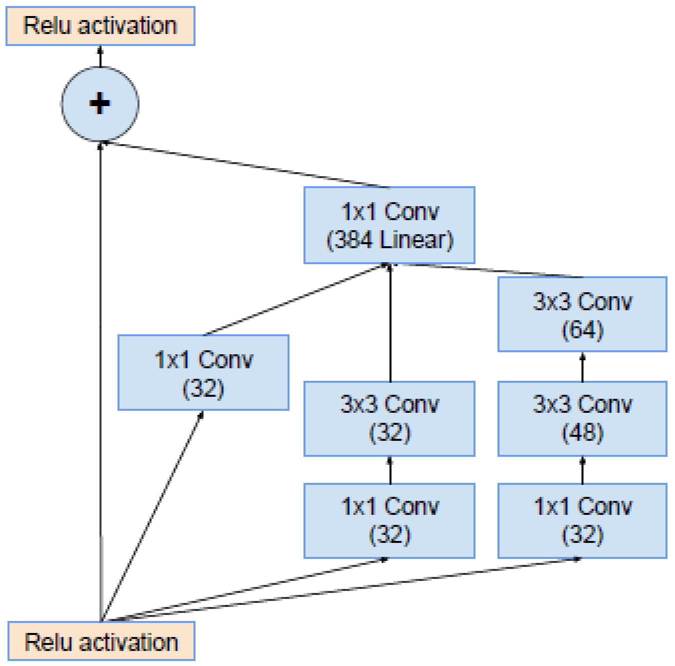


Figure 16. The schema for 35 × 35 grid (Inception-ResNet-A) module of the Inception-ResNet-v2 network.

Inception-ResNet V2 B block:

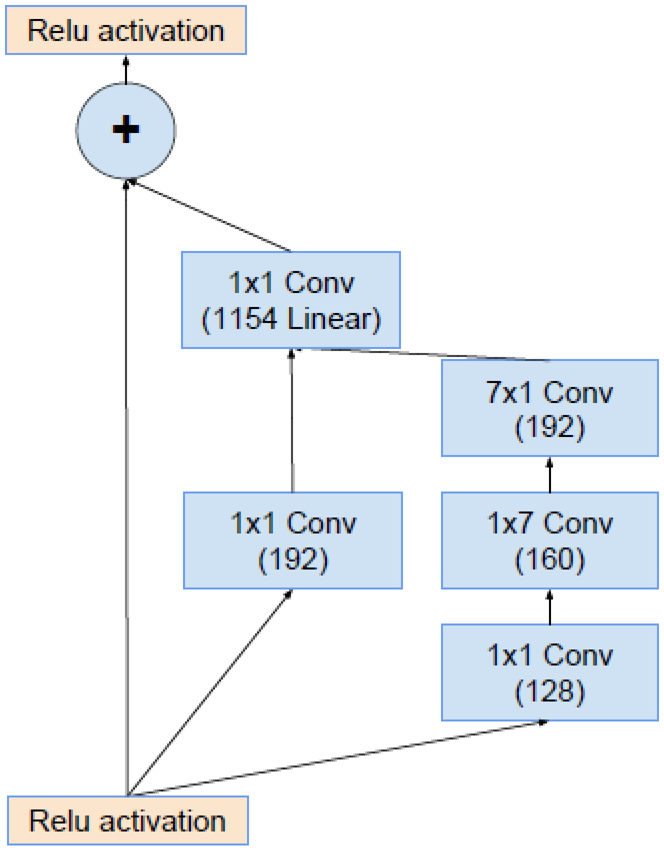


Figure 17. The schema for 17 × 17 grid (Inception-ResNet-B) module of the Inception-ResNet-v2 network.

Inception-ResNet V2 C block:

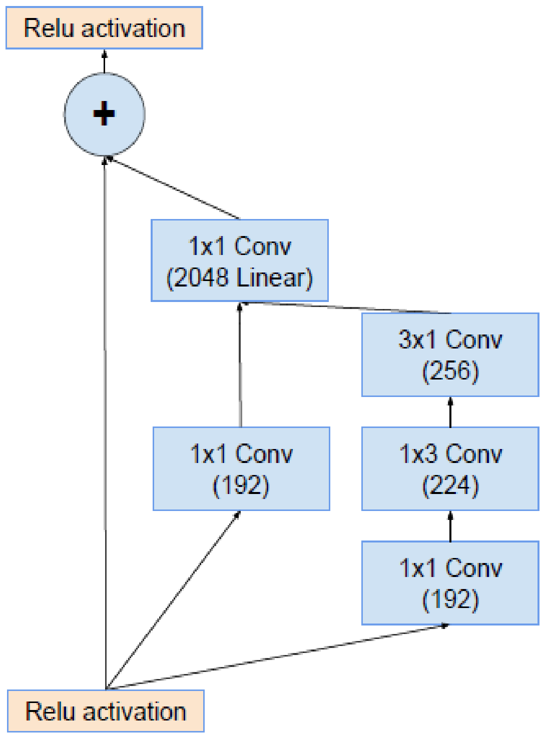


Figure 19. The schema for 8×8 grid (Inception-ResNet-C) module of the Inception-ResNet-v2 network.

Reduction A block: (same structure as Inception V4, albeit with different k, l, m, n values)

Inception-ResNet V2 Reduction B block:
(Note: a typo in the description, it should be "Inception-ResNet-v2" instead of Inception-ResNet-v1(

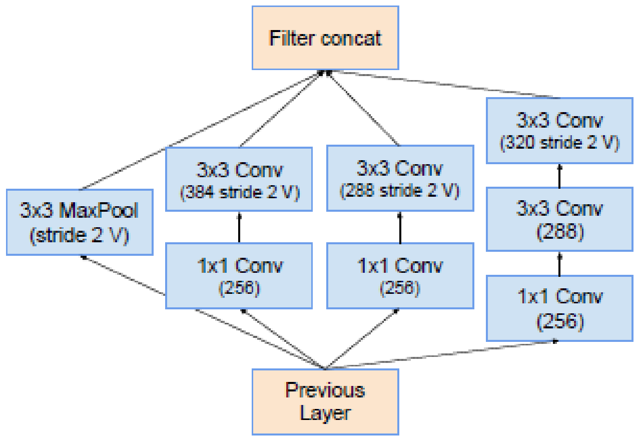


Figure 18. The schema for 17 × 17 to 8 × 8 grid-reduction module. Reduction-B module used by the wider Inception-ResNet-v1 network in Figure 15.

Original Inception-ResNet V2 architecture:

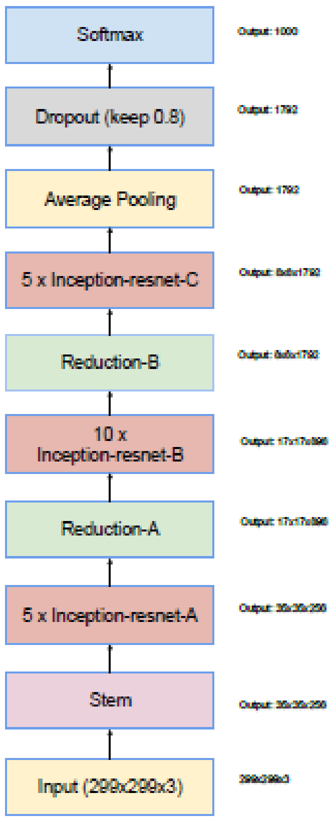


Figure 15. Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ. Inception-ResNet-v1 uses the blocks as described in Figures 14, 10, 7, 11, 12 and 13. Inception-ResNet-v2 uses the blocks as described in Figures 3, 16, 7,17, 18 and 19. The output sizes in the diagram refer to the activation vector tensor shapes of Inception-ResNet-v1.

Experiment 1 - Original Inception-ResNet V2:

We have implemented the exact Inception-ResNet V2 structure of the paper based on our understanding. However, a zero padding of (0,1), (0,1) was added and average padding was modified to (1,1) due to a size mismatch between the original dataset (299, 299, 3) and our dataset (96, 96, 3).

In the experiment, (similar to the original Inception V4) the number of layers were also too extensive for the classification problem and we were faced with the vanishing gradient problem (gradients go to 0 or infinity). This resulted in accuracy values of 60.88% and 39.12% which corresponds to predicting all as flowers or all as non-flowers respectively.

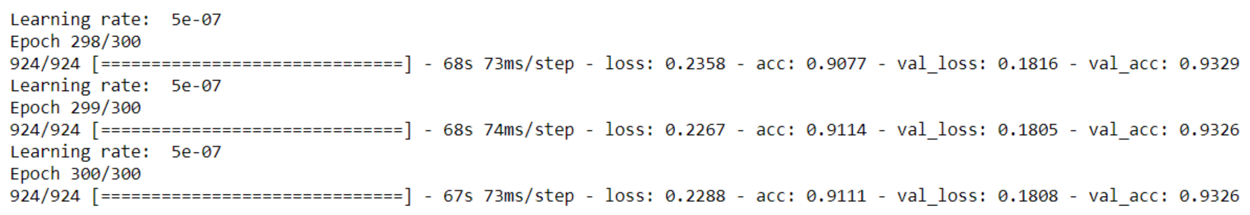
Experiment 2 - Inception-ResNet V2 with Batch Normalization, halving kernel filters/ pool sizes, Data Augmentation, RMSProp, modifying learning rate, reduced steps/epoch, added epochs and reducing blocks:

Learning from the experiments in Inception V4, we utilized various techniques:

1) Adding Batch Normalization between each Inception-ResNet block to avoid the vanishing gradient problem.
2) Using Data Augmentation, halving kernel filters/ pool sizes to reduce overfitting.
3) Using RMSProb, reducing learning rate, reducing steps/epoch, adding epochs to increase test accuracy.

In addition we also reduced the number of Inception-ResNet V2 A, B, C blocks from the original 5, 10, 5 to 1, 3, 1 to reduce the model complexity and hopefully reduce overfitting.

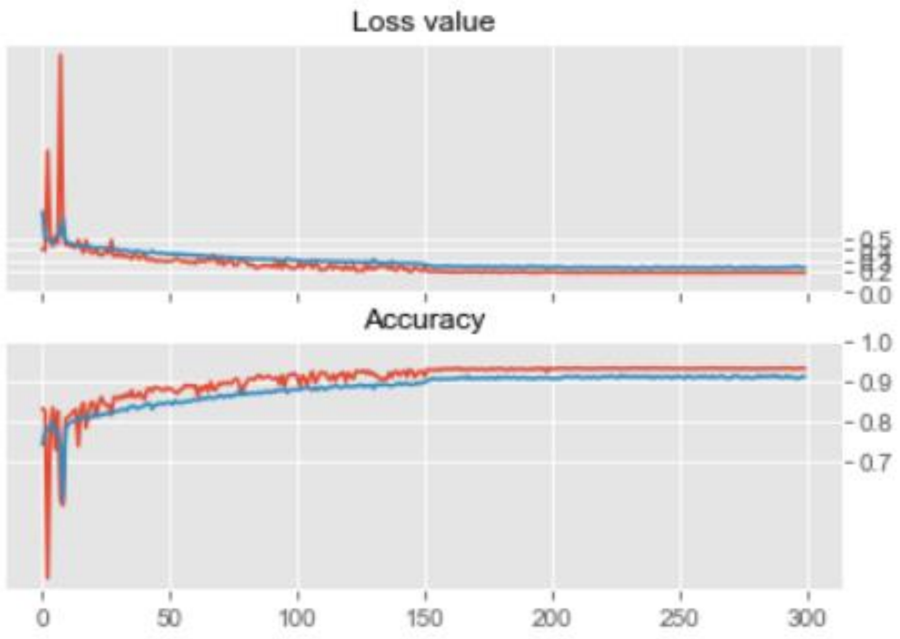
The experiment was a success as the test accuracy improved and the model was allowed to train to completion:



```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 68s 73ms/step - loss: 0.2358 - acc: 0.9077 - val_loss: 0.1816 - val_acc: 0.9329
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 68s 74ms/step - loss: 0.2267 - acc: 0.9114 - val_loss: 0.1805 - val_acc: 0.9326
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 67s 73ms/step - loss: 0.2288 - acc: 0.9111 - val_loss: 0.1808 - val_acc: 0.9326
```

The best accuracy obtained was **93.37%**:

```
Best accuracy (on testing dataset): 93.37%
                precision    recall  f1-score   support

  non-flower       0.8852    0.9544    0.9185      1446
      flower       0.9691    0.9204    0.9442      2250

    accuracy                           0.9337      3696
   macro avg       0.9271    0.9374    0.9313      3696
weighted avg       0.9363    0.9337    0.9341      3696

[[1380   66]
 [ 179 2071]]
```

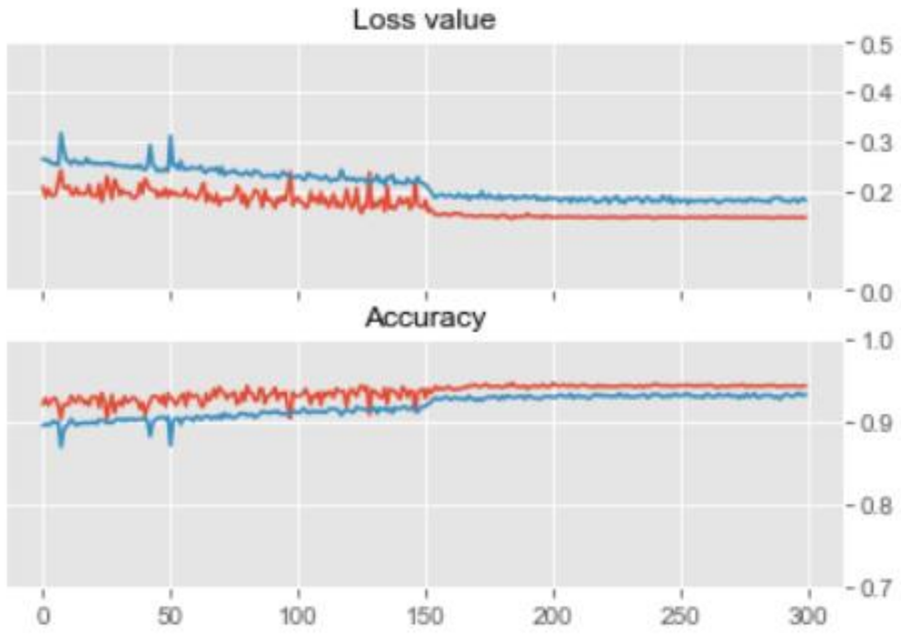


Experiment 3 - latest Inception-ResNet V2 settings, pre-saved weights for another 300x2 epochs:

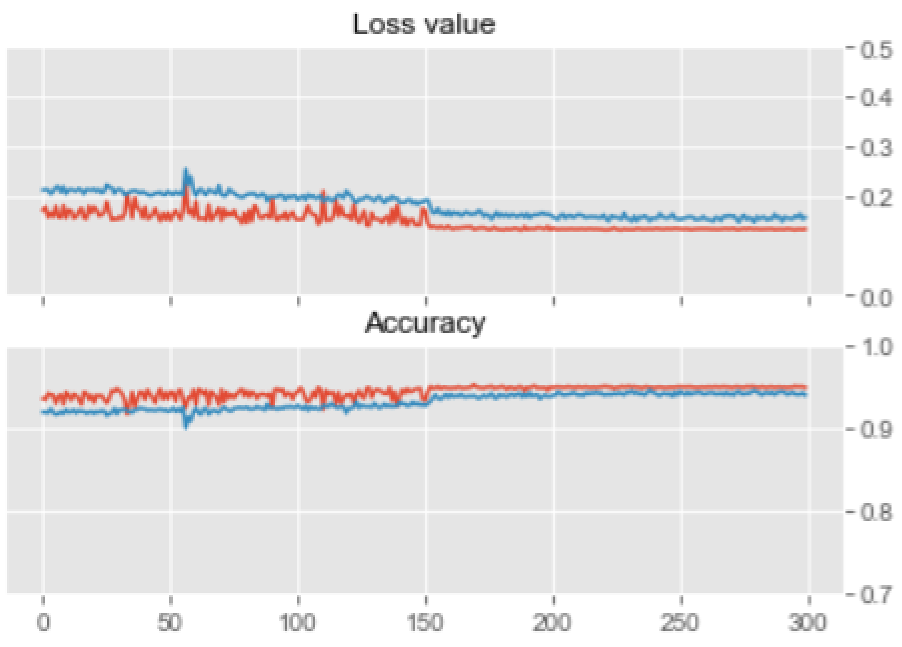
As it was noticed that the training accuracy had not peaked close to ~99%, it was deemed that the model could possibly learn more by having increased epochs. Hence the saved weights for the previous best model (by accuracy, **93.37%**) was used as a starting point for another 300x2 epochs of training.

This enabled us to increase the test accuracy to **94.67%** and subsequently **95.21%**:

```
Best accuracy (on testing dataset): 94.67%
              precision    recall  f1-score   support

  non-flower     0.9111    0.9571    0.9336      1446
      flower     0.9715    0.9400    0.9555      2250

    accuracy                         0.9467      3696
   macro avg     0.9413    0.9486    0.9445      3696
weighted avg     0.9479    0.9467    0.9469      3696

[[1384   62]
 [ 135 2115]]
```



```
Best accuracy (on testing dataset): 95.21%
              precision    recall  f1-score   support

  non-flower     0.9216    0.9592    0.9400      1446
      flower     0.9731    0.9476    0.9601      2250

    accuracy                         0.9521      3696
   macro avg     0.9473    0.9534    0.9501      3696
weighted avg     0.9529    0.9521    0.9523      3696

[[1387   59]
 [ 118 2132]]
```

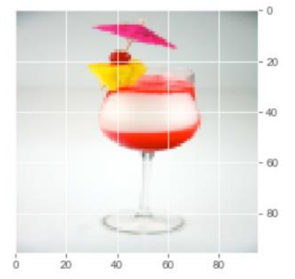
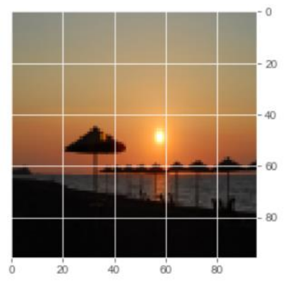
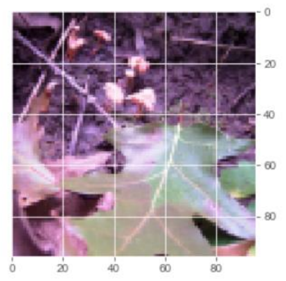
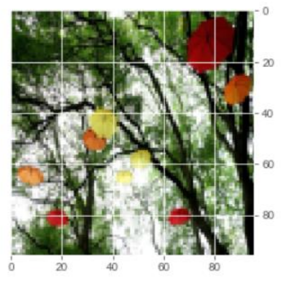


Although the training accuracy still had not peaked close to ~99%, due to a lack of time, we did not continue this method of training for more epochs:

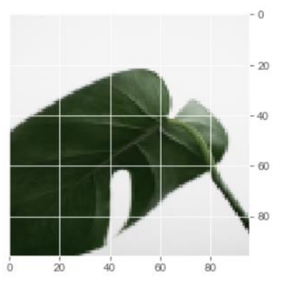
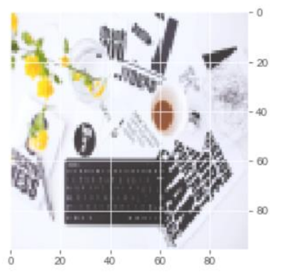
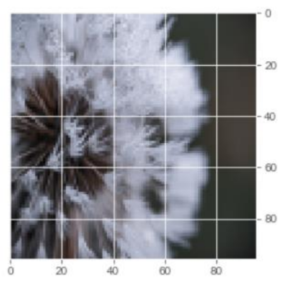
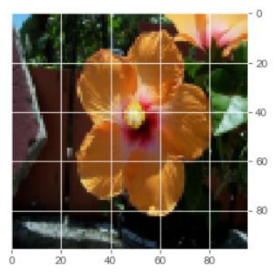
```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 67s 73ms/step - loss: 0.1628 - acc: 0.9405 - val_loss: 0.1321 - val_acc: 0.9499
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 69s 75ms/step - loss: 0.1530 - acc: 0.9422 - val_loss: 0.1312 - val_acc: 0.9499
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 73s 79ms/step - loss: 0.1560 - acc: 0.9397 - val_loss: 0.1325 - val_acc: 0.9489
```

Hence as of its current iteration, our implementation of the Inception-ResNet-V2 model has **95.21%** test accuracy.

Shown below are a few examples of images erroneously tagged as “flowers”:

Shown below are a few examples of images erroneously tagged as “non-flowers”:

As it can be seen, some of the images tagging as “flowers” or “non-flowers” are quite debatable, for these cases a further cleaning of the data is required. However there are also cases where it is obviously a “flower”, in these cases it should be further studied as to why the model predicts otherwise, to determine whether it is a borderline case (0.5) or is it something else.

10. Approach Towards Squeeze Excitation Inception-ResNet V2 Structure

Data Set:

The first version data set (Data_V2) was used for training and testing as it was deemed as more comprehensive than the first version data set (Data_V1).

Squeeze Excitation (SE) Inception-ResNet V2 Architecture:

Similar to the idea of attention in recurrent neural networks and the gates in LSTM, the SE block helps to emphasise important features and suppress the rest: "We introduce a new architectural unit, which we term the Squeeze-and-Excitation (SE) block, with the goal of improving the quality of representations produced by a network by explicitly modelling the interdependencies between the channels of its convolutional features. To this end, we propose a mechanism that allows the network to perform feature recalibration, through which it can learn to use global information to selectively emphasise informative features and suppress less useful ones."

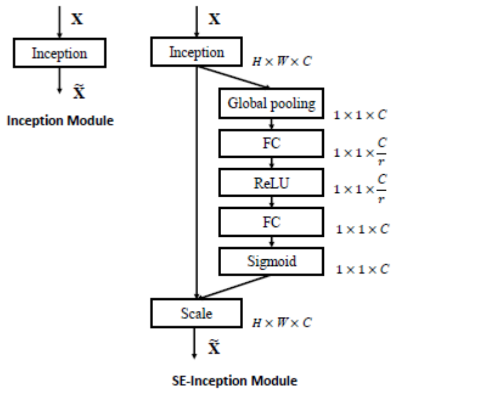


Fig. 2. The schema of the original Inception module (left) and the SE-Inception module (right).

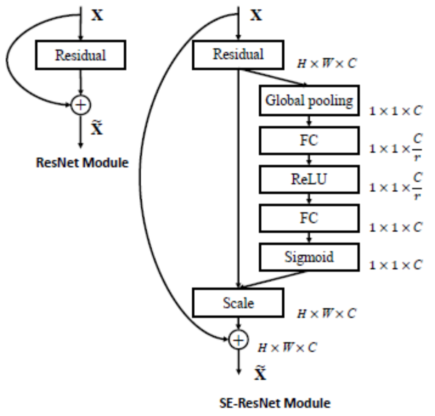
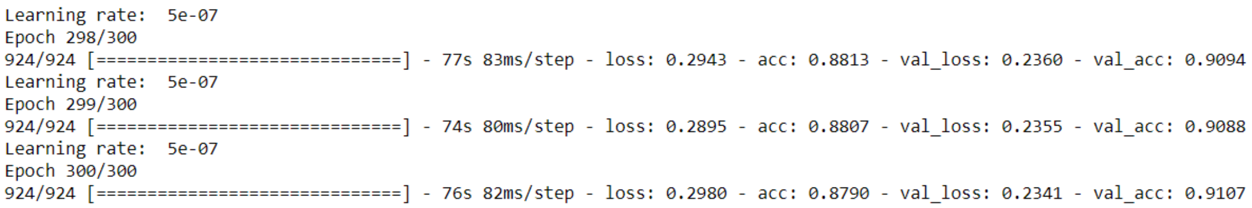


Fig. 3. The schema of the original Residual module (left) and the SE-ResNet module (right).

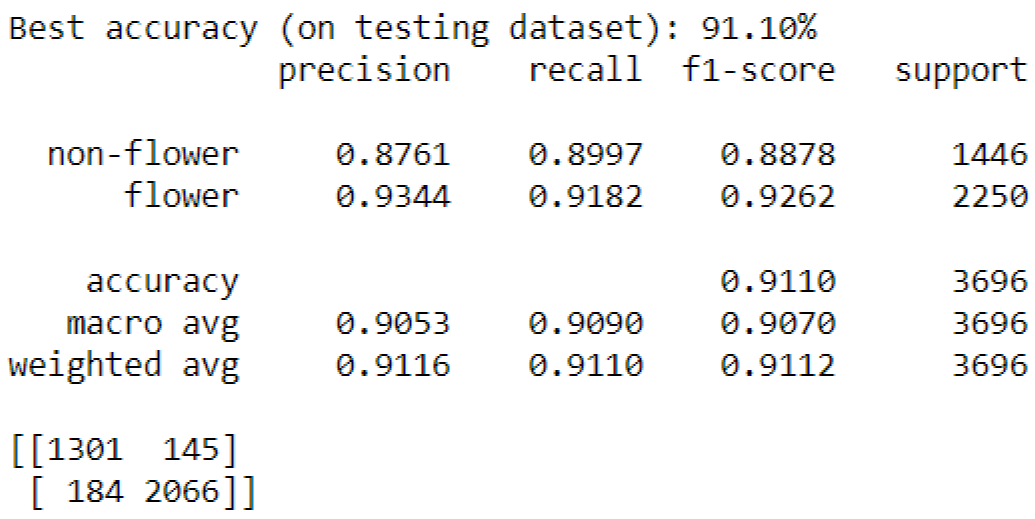
In our implementation, we only managed to combine the Squeeze Excitation with the Inception-ResNet V2 due to time constraints.

Experiment 1 - latest Inception-ResNet V2 settings with Squeeze Excitation (SE) blocks:
We decided to continue from the latest Inception-ResNet V2 settings instead of starting from the original Inception-ResNet V2 because it would almost certainly face the same issues and problems. The SE layer was added after every A, & B block of the Inception-ResNet V2.

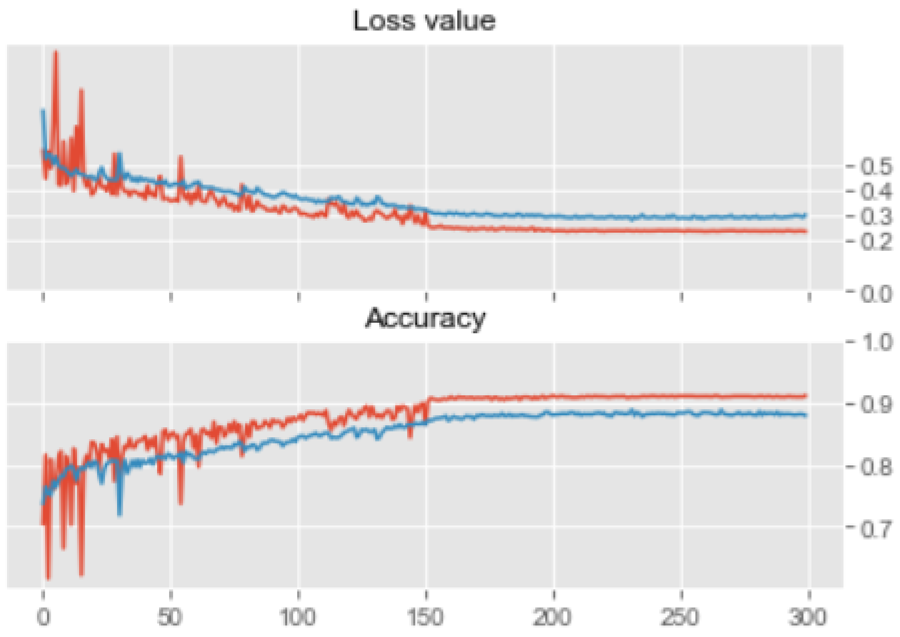
The model was not susceptible to the vanishing gradient problem/ overfitting and was allowed to train to completion:



```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 77s 83ms/step - loss: 0.2943 - acc: 0.8813 - val_loss: 0.2360 - val_acc: 0.9094
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 74s 80ms/step - loss: 0.2895 - acc: 0.8807 - val_loss: 0.2355 - val_acc: 0.9088
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 76s 82ms/step - loss: 0.2980 - acc: 0.8790 - val_loss: 0.2341 - val_acc: 0.9107
```



```
Best accuracy (on testing dataset): 91.10%
              precision    recall  f1-score   support

  non-flower     0.8761    0.8997    0.8878      1446
      flower     0.9344    0.9182    0.9262      2250

    accuracy                         0.9110      3696
   macro avg     0.9053    0.9090    0.9070      3696
weighted avg     0.9116    0.9110    0.9112      3696

[[1301  145]
 [ 184 2066]]
```



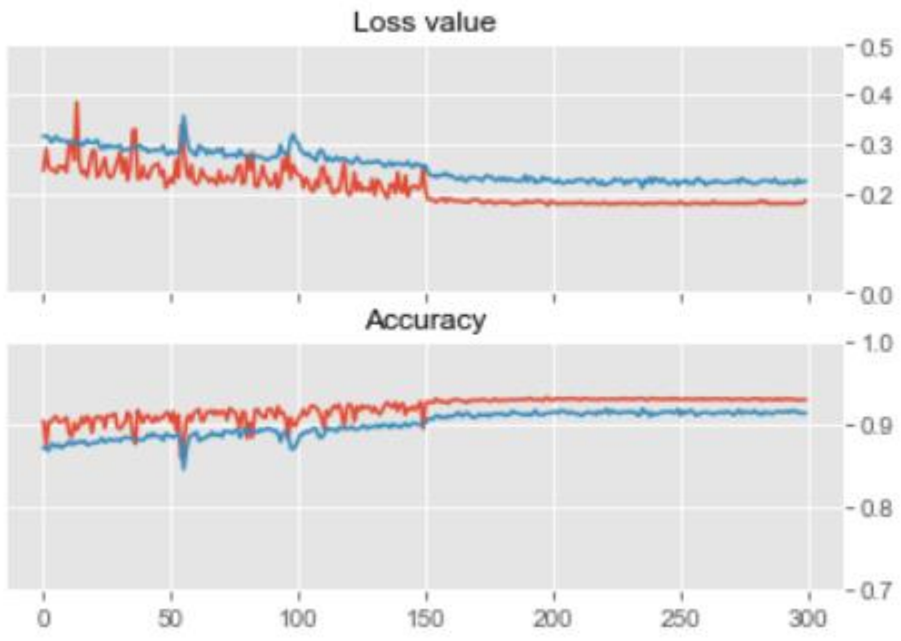
It would seem like the addition of the SE block only hindered the model as the accuracy dropped from **94.70%** to **91.10%**. However it should be noted that the training accuracy was much lower at ~88% as compared to the final Inception-ResNet V2 model.

Experiment 2 - latest SE Inception-ResNet V2 settings, pre-saved weights for another 300x3 epochs:

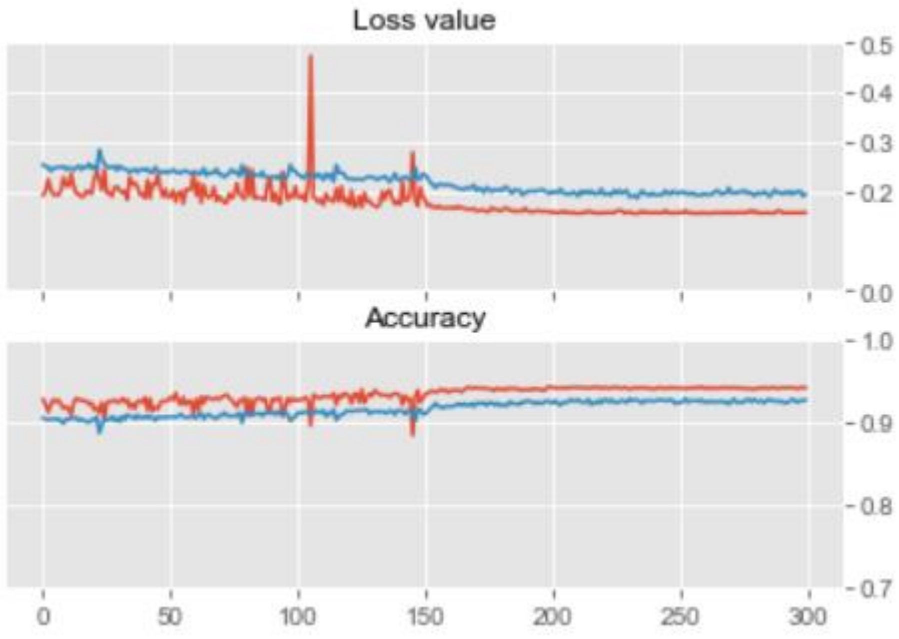
As it was noticed that the training accuracy had not peaked close to ~99%, it was deemed that the model could possibly learn more by having increased epochs. Hence the saved weights for the previous best model (by accuracy, **91.10%**) was used as a starting point for another 300x3 epochs of training.

This enabled us to increase the test accuracy to **93.21%, 94.29%** and subsequently **94.89%**:

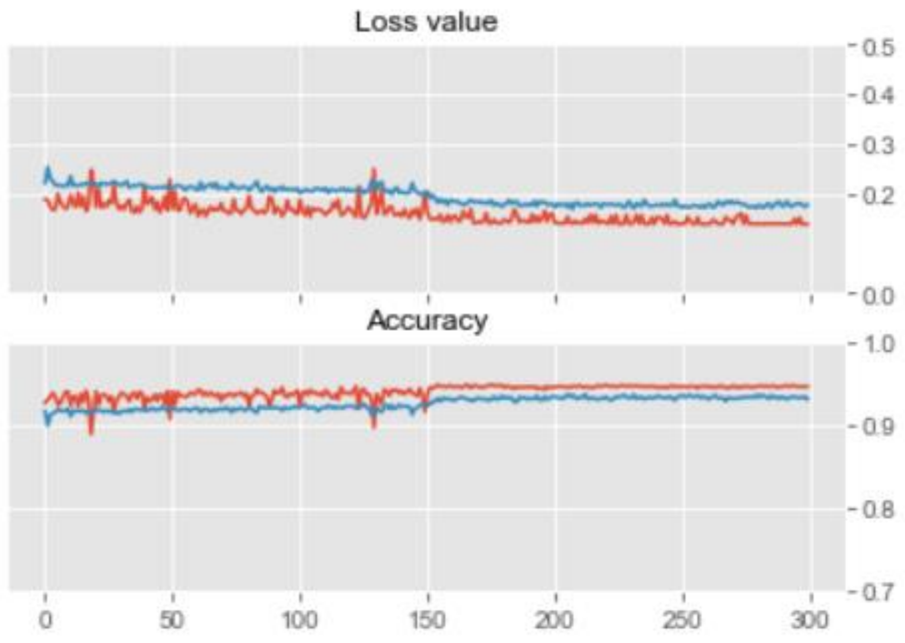
```
Best accuracy (on testing dataset): 93.21%
              precision    recall  f1-score   support

  non-flower     0.8960    0.9350    0.9151      1446
      flower     0.9570    0.9302    0.9434      2250

    accuracy                         0.9321      3696
   macro avg     0.9265    0.9326    0.9292      3696
weighted avg     0.9331    0.9321    0.9323      3696

[[1352   94]
 [ 157 2093]]
```



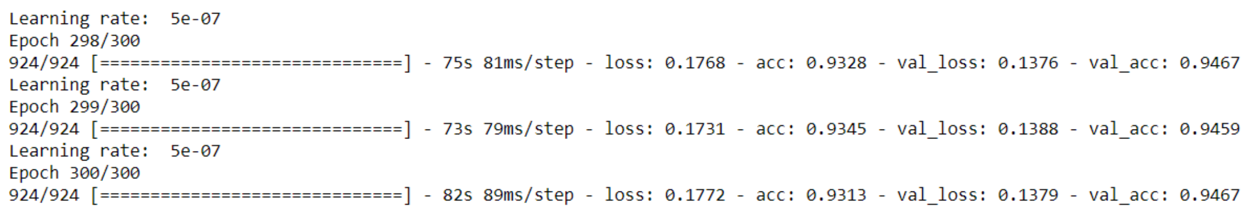
```
Best accuracy (on testing dataset): 94.29%
              precision    recall  f1-score   support

  non-flower     0.9087    0.9495    0.9286      1446
      flower     0.9666    0.9387    0.9524      2250

    accuracy                         0.9429      3696
   macro avg     0.9376    0.9441    0.9405      3696
weighted avg     0.9439    0.9429    0.9431      3696

[[1373   73]
 [ 138 2112]]
```



```
Best accuracy (on testing dataset): 94.89%
              precision    recall  f1-score   support

  non-flower     0.9296    0.9405    0.9350      1446
      flower     0.9615    0.9542    0.9578      2250

    accuracy                         0.9489      3696
   macro avg     0.9455    0.9474    0.9464      3696
weighted avg     0.9490    0.9489    0.9489      3696

[[1360   86]
 [ 103 2147]]
```



Although the training accuracy still had not peaked close to ~99%, due to a lack of time, we did not continue this method of training for more epochs:



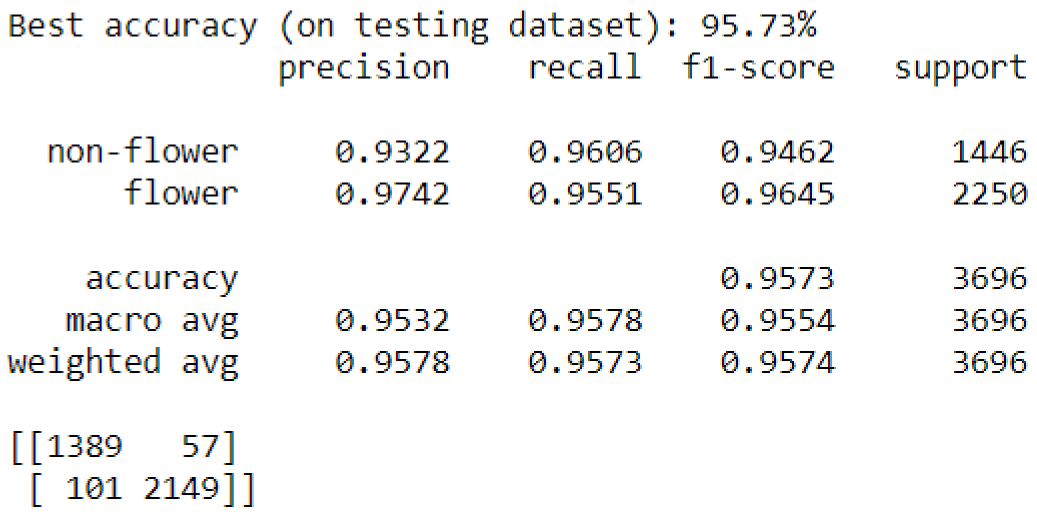
```
Learning rate:  5e-07
Epoch 298/300
924/924 [==============================] - 75s 81ms/step - loss: 0.1768 - acc: 0.9328 - val_loss: 0.1376 - val_acc: 0.9467
Learning rate:  5e-07
Epoch 299/300
924/924 [==============================] - 73s 79ms/step - loss: 0.1731 - acc: 0.9345 - val_loss: 0.1388 - val_acc: 0.9459
Learning rate:  5e-07
Epoch 300/300
924/924 [==============================] - 82s 89ms/step - loss: 0.1772 - acc: 0.9313 - val_loss: 0.1379 - val_acc: 0.9467
```

Hence as of its current iteration, our implementation of the SE Inception-ResNet-V2 model has **94.89%** test accuracy.

11. Combining various Neural Network Architectures

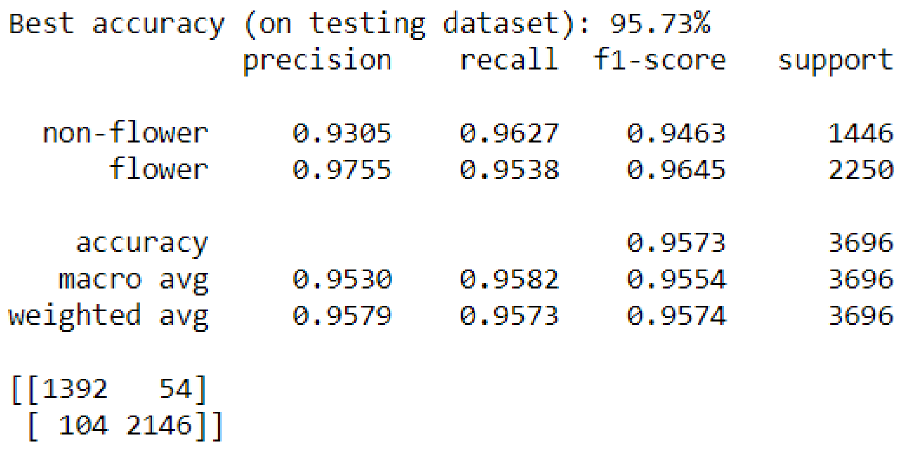
As all 3 models (Inception V4, Inception-ResNet V2, SE Inception-ResNet V2) have similar test accuracy values (**94.70%, 95.21%, 94.89%**), we also explored having the 3 models collectively vote for the best score. The prediction scores would be weighted and re-calculated.

Experiment 1 - Voting with equal weights:
We placed equal weightage on all 3 models (0.33, 0.33, 0.33) to calculate the final predicted score. The results are as follows:



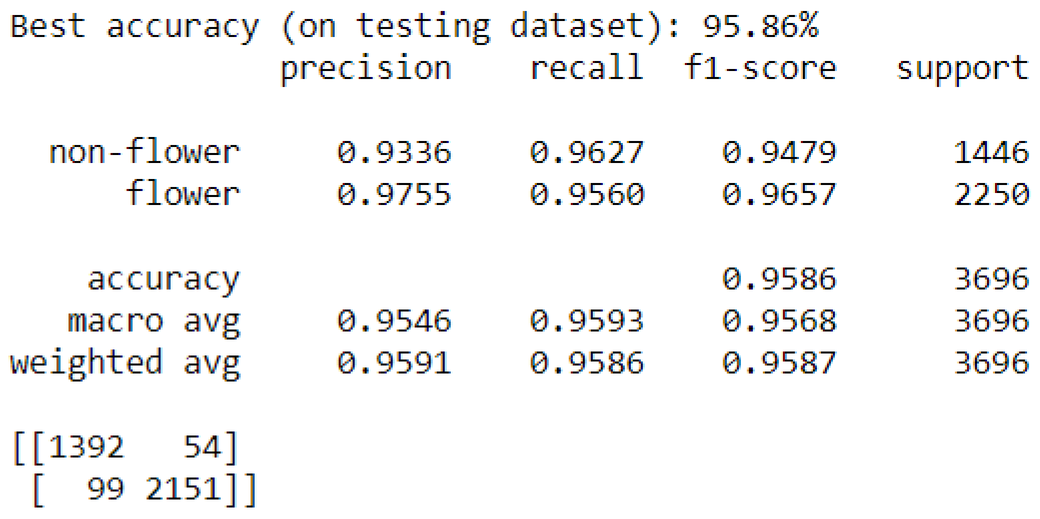
```
Best accuracy (on testing dataset): 95.73%
              precision    recall  f1-score   support

  non-flower     0.9322    0.9606    0.9462      1446
      flower     0.9742    0.9551    0.9645      2250

    accuracy                         0.9573      3696
   macro avg     0.9532    0.9578    0.9554      3696
weighted avg     0.9578    0.9573    0.9574      3696

[[1389   57]
 [ 101 2149]]
```

Experiment 2 - Voting with unequal weights:
We placed higher weightage on the Inception-ResNet V2 model (0.30, 0.40, 0.30) to calculate the final predicted score.
The results are as follows:



```
Best accuracy (on testing dataset): 95.73%
              precision    recall  f1-score   support

  non-flower     0.9305    0.9627    0.9463      1446
      flower     0.9755    0.9538    0.9645      2250

    accuracy                         0.9573      3696
   macro avg     0.9530    0.9582    0.9554      3696
weighted avg     0.9579    0.9573    0.9574      3696

[[1392   54]
 [ 104 2146]]
```

Experiment 3 - Voting with optimal weights:
We ran possible combinations of the weightage for the voting model to calculate the final predicted score. (The weights for each varied by 0.025.) The results are as follows:



```
Best accuracy (on testing dataset): 95.86%
              precision    recall  f1-score   support

  non-flower     0.9336    0.9627    0.9479      1446
      flower     0.9755    0.9560    0.9657      2250

    accuracy                         0.9586      3696
   macro avg     0.9546    0.9593    0.9568      3696
weighted avg     0.9591    0.9586    0.9587      3696

[[1392   54]
 [  99 2151]]
```

The optimal weight obtained was (0.35, 0.375, 0.275). It is strange that the SE Inception-ResNet V2 model which has a test accuracy of **94.89%** is prioritized less than the Inception V4 model which has a test accuracy of **94.70%**. One explanation could be that the Inception V4 model is able to correctly identify different images from the Inception-ResNet V2 model whilst the SE Inception-ResNet V2 model overlaps largely in the images it correctly identifies.

The voting classifier has **95.86%** test accuracy.

12. Conclusion

In conclusion, the team has experimented on several popular SOTA CNN architectures, namely: Deep CNN, VGG, AlexNet, ResNet, Inception V4, Inception-ResNet V2 and SE Inception-ResNet V2. Throughout the process to further improve model performance, the team has applied techniques including data augmentation, dropout, normalization, progressive learning, loading pre-trained model weight and voting. In the end, we managed to obtain a model that can predict with reasonable accuracy of **95.86%**.

References

ResNet
Kaiming He, Xiangyu, Zhang Shaoqing and Ren Jian Sun, Deep Residual Learning for Image Recognition, 2015. https://arxiv.org/pdf/1512.03385.pdf

Inception V4 & Inception-ResNet V2
Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke and Alex Alemi,
Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning
https://arxiv.org/pdf/1602.07261.pdf

SE Inception-ResNet V2
Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu
Squeeze-and-Excitation Networks
https://arxiv.org/pdf/1709.01507.pdf

VGG
Karen Simonyan and Andrew Zisserman,
VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION, 2015.
https://arxiv.org/pdf/1409.1556.pdf.

Muneeb ul Hassan,
VGG16 - Convolutional Network for Classification and Detection, 2018, https://neurohive.io/en/popular-networks/vgg16

AlexNet
Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton,
ImageNet Classification with Deep Convolutional Neural Networks, 2012,
https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Citation
ImageNet:http://www.image-net.org/
Pexels :https://www.pexels.com/
OIDv4_Tookit(https://github.com/EscVM/OIDv4_ToolKit)
https://storage.googleapis.com/openimages/web/index.html
https://www.kaggle.com/wassimseifeddine/102flowersdataset/downloads/102flowersdataset.zip/1.

(https://paperswithcode.com/sota/image-classification-on-cifar-10)