Machine Learning

Transfer Learning

We used **Keras** to create models that classify the image into the appropriate category.

For this assignment, we used the **repository** - https://www.robots.ox.ac.uk/~vgg/data/flowers/102/ - this is a dataset that contains flowers's images of 102 categories.

We worked with **Kaggle Kernels** in order to work with the many images and neural network.

The algorithm is written in Python.

1.1. Preprocessing and split

From the site above we get images that are numbered.

Also downloaded labels corresponding to the images.

We created the Pandas's DataFrame containing image paths and image labels and shuffled it.

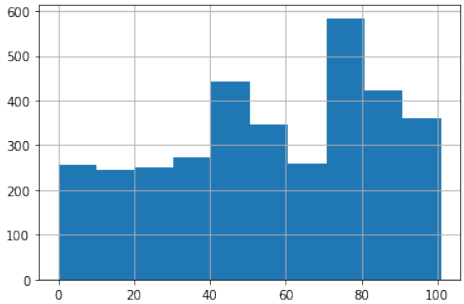
Next, we divided the DataFrame into three parts: train set, validation set and test set, in the following propor tions:

Number of example in Train : 3439
Size=41.995%

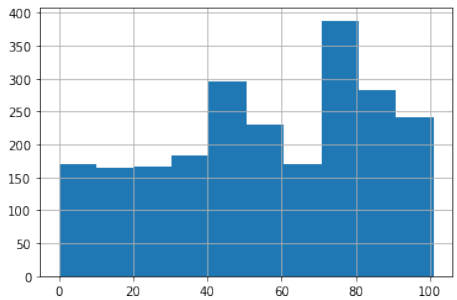
Number of example in Validation : 2293
Size=28.001%

Number of example in Test : 2457
Size=30.004%

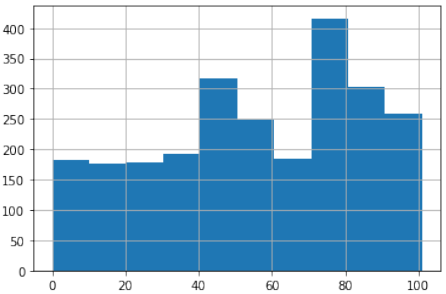
We made sure that all parts of the split have the same target distribution:



Distribution in Train



Distribution in Validation



Distribution in Test

We wrote a function that create folders and sort pictures into folders for each split part according to categories (3 folders of 102 folders (102 categories)).

To increase the size of the training set and diversify it, we used **augmentation** for the train (**ImageDataGenerator**).

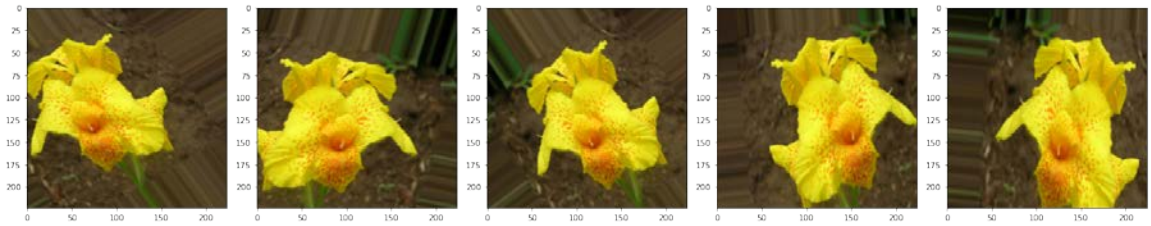
```
Found 3439 images belonging to 102 classes.
Found 2293 images belonging to 102 classes.
Found 2457 images belonging to 102 classes.
```

The objective of ImageDataGenerator (keras.preprocessing) is to import data with labels easily into the model. This class alters the data on the go while passing it to the model. We created an ImageDataGenerator object for training data, for validation data, and for testing data.

To increase the size of the training set and diversify it, we applied the following methods to the images of the training set that were initially available (we did not use an additional repository):

- rescale
- rotation_range
- width_shift_range
- height_shift_range
- shear_range
- zoom_range
- horizontal_flip

Example of received new samples:

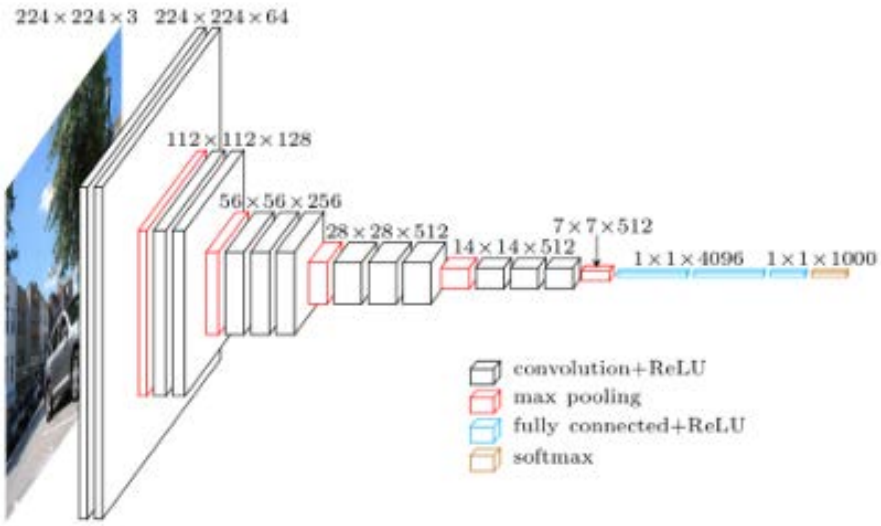


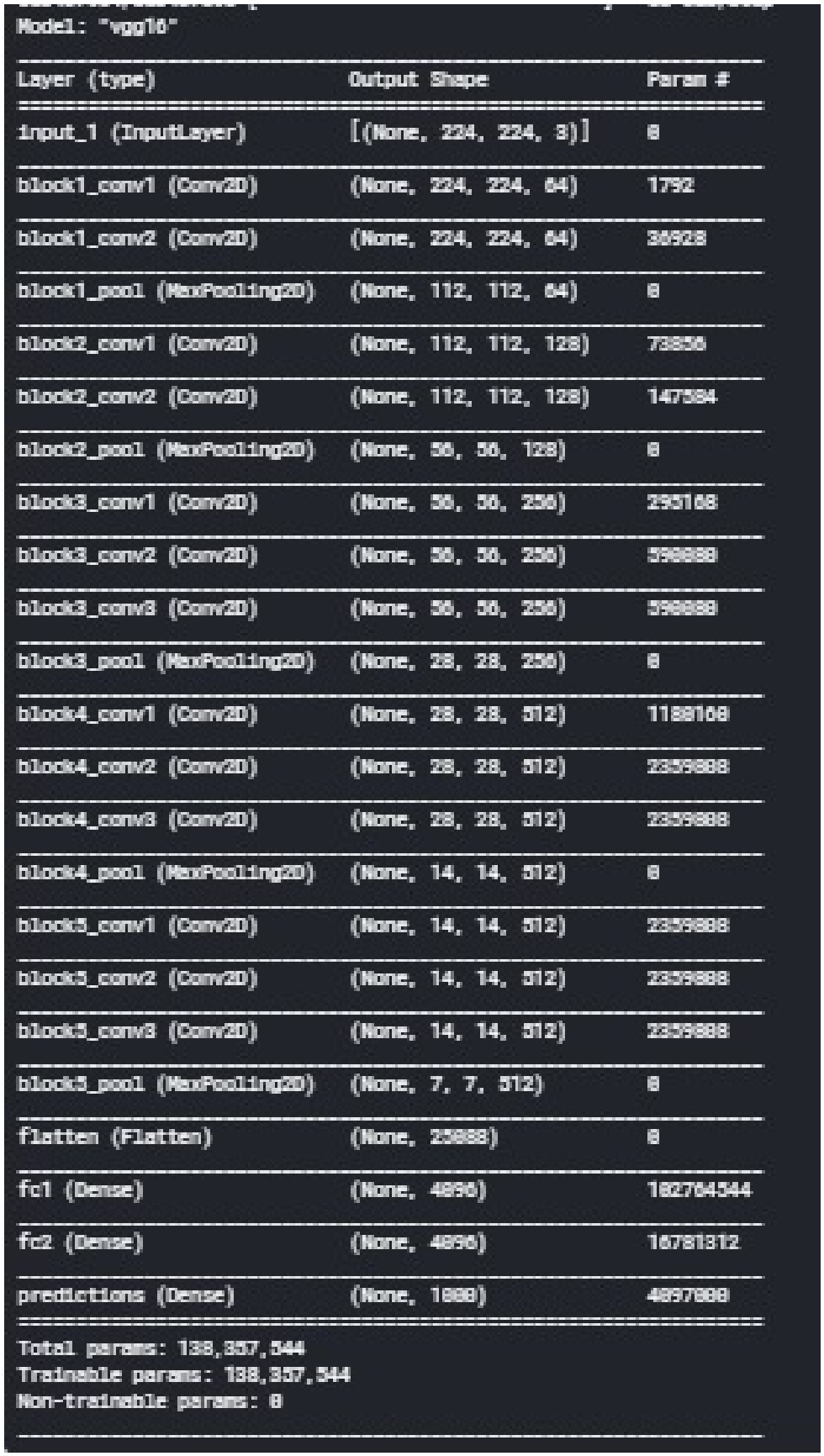
We also rescaled the validation data.

Since we used a pre-trained model, we needed to resize the input data (images) to the same format the network was originally trained on, in our case the dimension is 244x244.

1.2. VGG-16

As the first model, we used the architecture of convolutional neural networks (CNN) - VGG-16.





```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
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
_________________________________________________________________
```

Architecture of VGG-16

We froze the weights and biases of the pre-trained model and used the pre-trained layers of VGG, up to layer with the name block5_pool.

After creating the convolution, we passed the data to the dense layer so for that we **flatten** the vector which comes out of the convolutions and add layers:

- 1 x Dense layer of 4096 units
- 1 x DropOut layer with 0.1 rate
- 1 x Dense layer of 4096 units
- 1 x DropOut layer with 0.1 rate
- 1 x Dense Softmax layer of 102 units

We added 2 dense layers with 4096 nodes with the **ReLU** activation function.

We used **drop out** to regularize our model. After each of these two layers, we added a drop out layer with rate 0.1. The dropout layers make the model more generalize and prevents over-fitting. Dropout is easily implemented by randomly selecting nodes to be dropped-out with a given probability, we set 10%.

We use a dense layer of 102 nodes in the end with **softmax** activation, as we have 102 classes to predict from in the end which are different flowers.

After creating the softmax layer, the model is finally prepared. We used a call back and saved the weight of the best model. And later on the test data used this model.

Our results:

- **Accuracy**

```
loss_test_vgg, acc_test_vgg = custom_vgg_model.evaluate_generator(test_data_gen, verbose=1)

acc_val=history_vgg.history['val_accuracy'][history_vgg.history['val_accuracy'].index(max(history_vgg.history['val_accuracy']))]
acc_tr=history_vgg.history['accuracy'][history_vgg.history['val_accuracy'].index(max(history_vgg.history['val_accuracy']))]
print('Accuracy on test set:',acc_test_vgg)
print('Accuracy on validation set:',acc_val)
print('Accuracy on train set:',acc_tr)
print('Loss function on test set:',loss_test_vgg)
```

```
39/39 [==============================] - 34s 878ms/step - loss: 0.8488 - accuracy: 0.7725
Accuracy on test set: 0.7724867463111877
Accuracy on validation set: 0.7823811769485474
Accuracy on train set: 0.7952893376350403
Loss function on test set: 0.848770022392273
```

-

Training: 80%, **Test:** 77%, **Validation:** 78%,

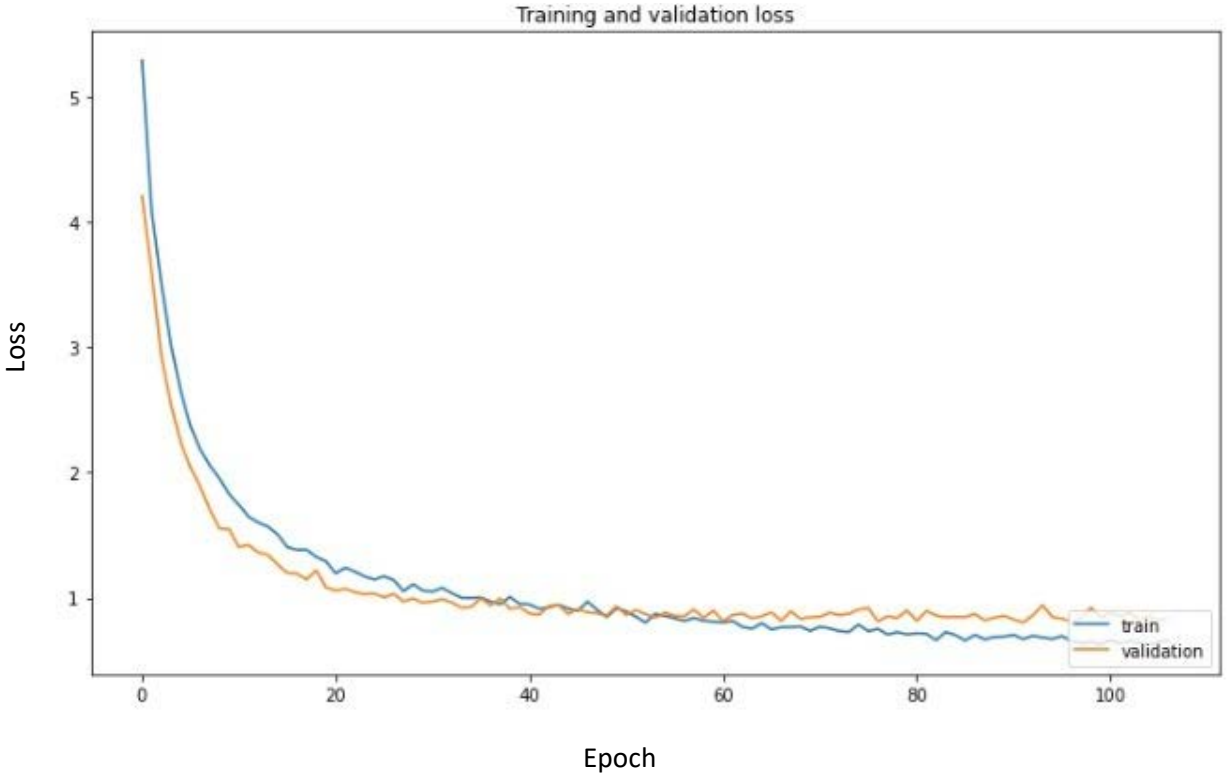
when split the data set by:

42% Train, 30% Test, 28% Validation.

❖

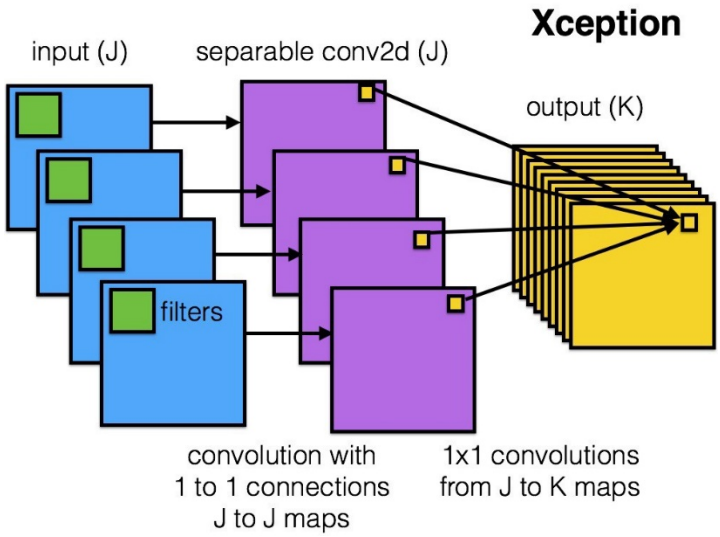

Training and validation accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100
train
validation

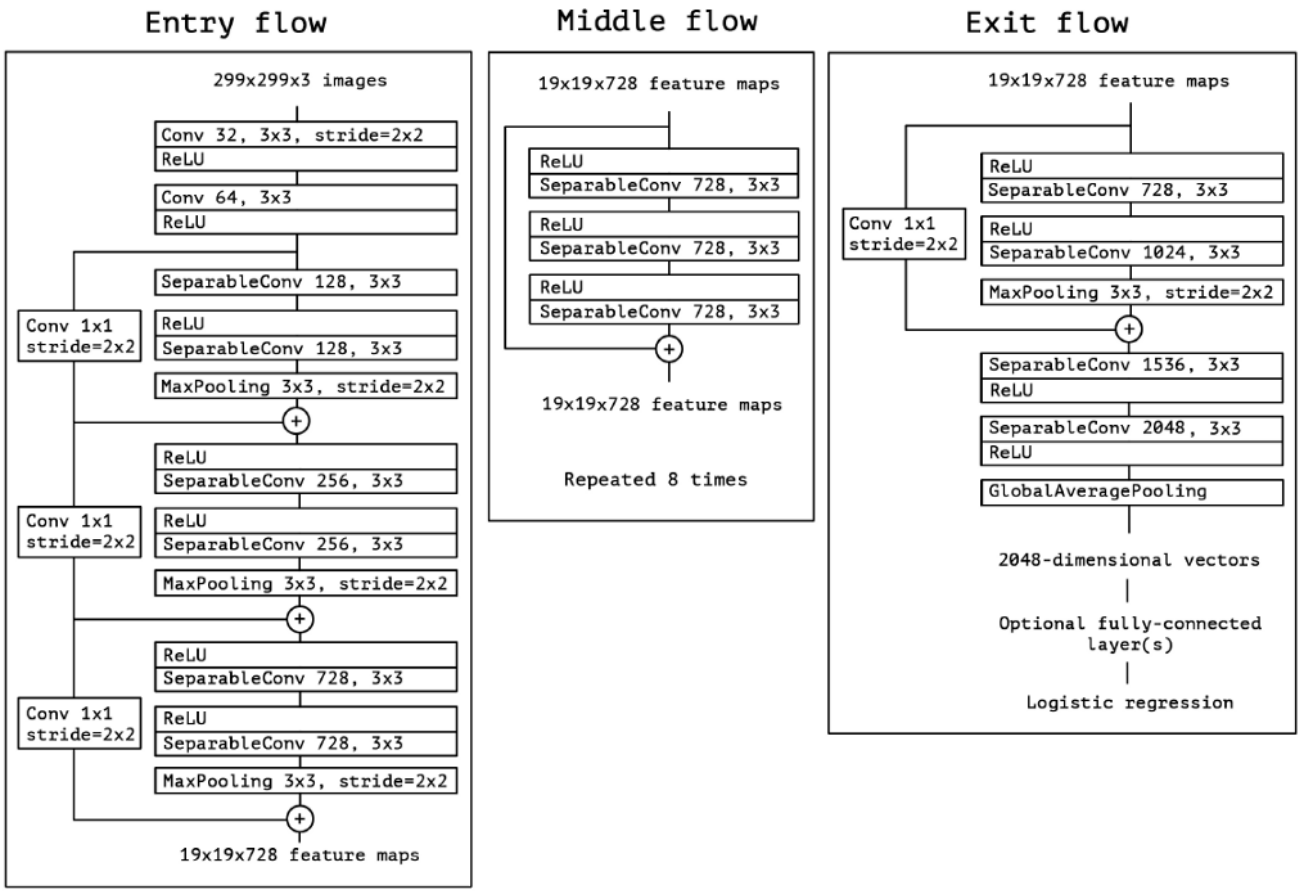
❖


Training and validation loss
Loss
5
4
3
2
1
0
20
40
60
80
100
train
validation
Epoch

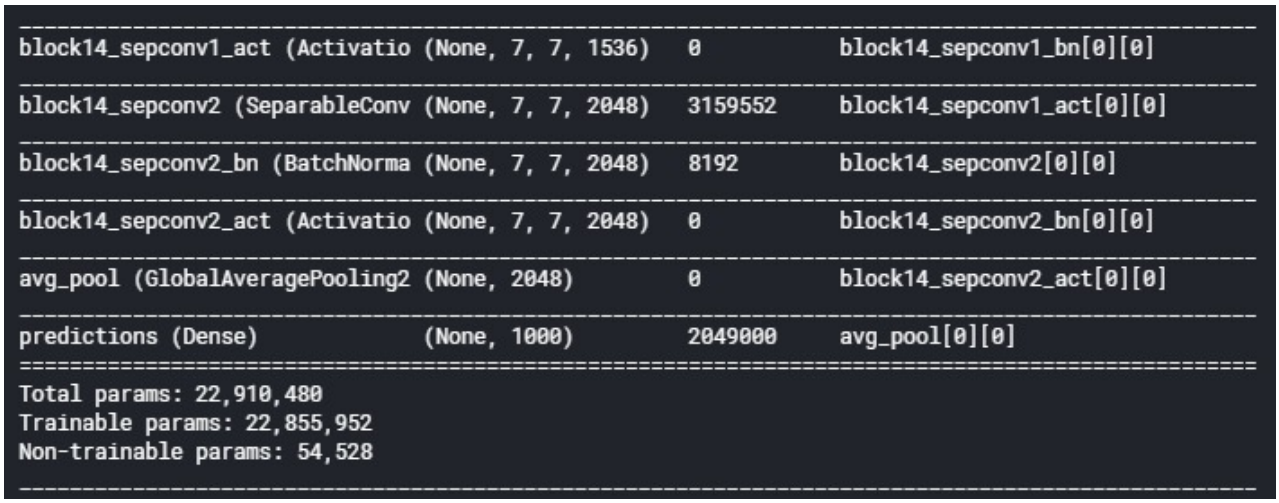
1.2. Xception

As the second model, we used the convolutional neural network (CNN) architecture - Xception.





Architecture of Xception



Architecture of Xception

We froze the weights and biases of the pre-trained model and used the pre-trained layers of Xception, up to layer with the name block14_sepconv2_act.

We passed the data to the dense layer so for that we **flatten** the vector which comes out of the convolutions and add layers:

- 1 x Dense layer of 4096 units
- 1 x DropOut layer with 0.2 rate
- 1 x Dense layer of 4096 units
- 1 x DropOut layer with 0.2 rate
- 1 x Dense Softmax layer of 102 units

We added 2 dense layers with 4096 nodes with the **ReLU** activation function.

We used **drop out** to regularize our model. After each of these two layers, we added a drop out layer with rate 0.2. The dropout layers make the model more generalize and prevents over-fitting. Dropout is easily implemented by randomly selecting nodes to be dropped-out with a given probability, we set 20%.

We use a dense layer of 102 nodes in the end with **softmax** activation, as we have 102 classes to predict from in the end which are different flowers.

After creating the softmax layer, the model is finally prepared. We used a call back and saved the weight of the best model. And later on the test data used this model.

Our results:

- **Accuracy**

```
loss_test_vgg, acc_test_vgg = custom_xception_model.evaluate_generator(test_data_gen, verbose=1)
acc_val=history_xception.history['val_accuracy'][history_xception.history['val_accuracy'].index(max(history_xception.history['val_acc
acc_tr=history_xception.history['accuracy'][history_xception.history['val_accuracy'].index(max(history_xception.history['val_accuracy

print('Accuracy on test set:',acc_test_vgg)
print('Accuracy on validation set:',acc_val)
print('Accuracy on train set:',acc_tr)
print('Loss function on test set:',loss_test_vgg)
```

```
39/39 [==============================] - 38s 978ms/step - loss: 0.6387 - accuracy: 0.8185
Accuracy on test set: 0.8184778094291687
Accuracy on validation set: 0.8168338537216187
Accuracy on train set: 0.8333817720413208
Loss function on test set: 0.6386771202087402
```

-

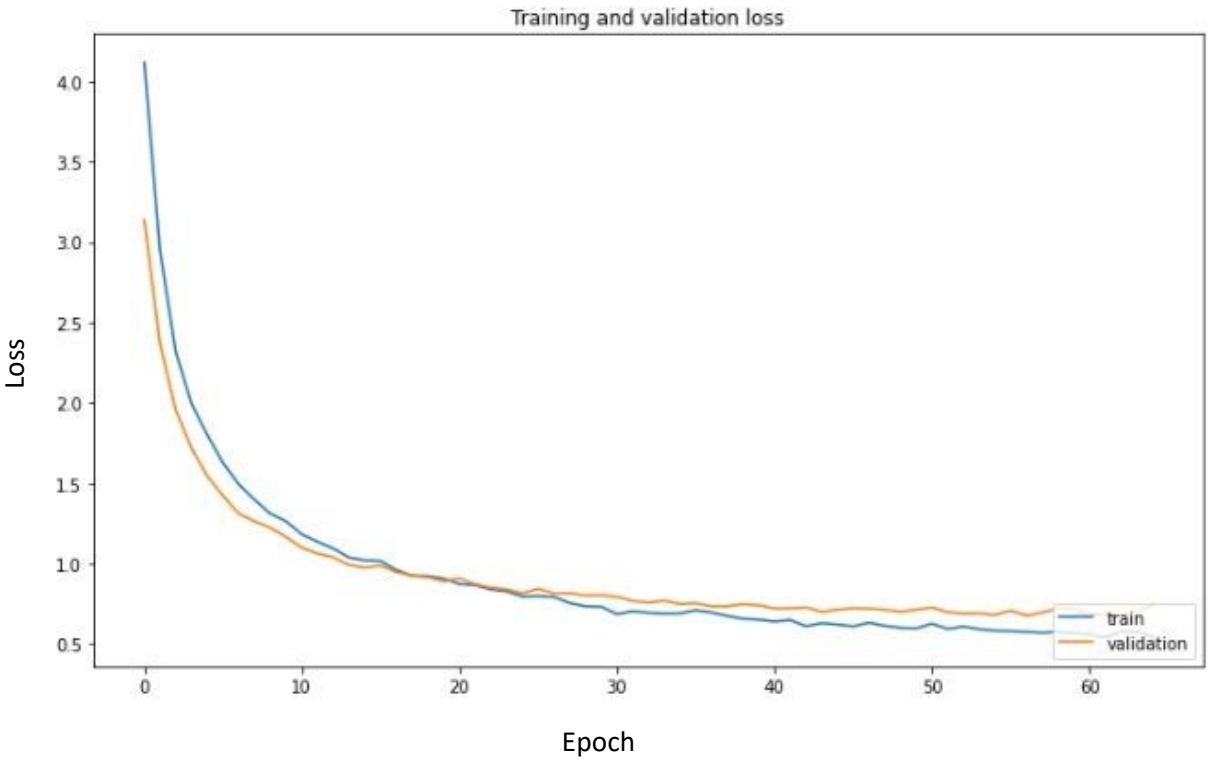
Training: 83%, **Test:** 82%, **Validation:** 82%,

when split the data set by:

42% Train, 30% Test, 28% Validation.

-

❖



2.1. New split

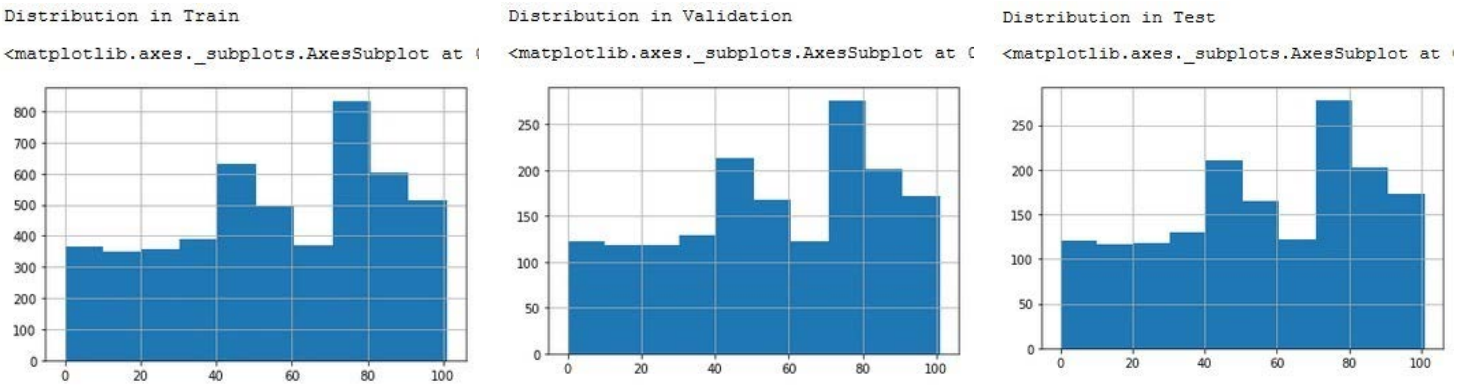
In the next two steps, we used the same models, but changed the proportions of the split of the data frame:

Number of example in Train : 4913
Size=59.995%

Number of example in Validation : 1638
Size=20.002%

Number of example in Test : 1638
Size=20.002%

We made sure that all parts of the split have the same target distribution:



On the new splits of the data frame, as well as above, we used **ImageDataGenerator**:

```
Found 4913 images belonging to 102 classes.
Found 1638 images belonging to 102 classes.
Found 1638 images belonging to 102 classes.
```

2.2. VGG-16

- **Accuracy**

```
loss_test_vgg, acc_test_vgg = custom_vgg_model.evaluate_generator(test_data_gen, verbose=1)

acc_val=history_vgg.history['val_accuracy'][history_vgg.history['val_accuracy'].index(max(history_vgg.history['val_accuracy']
acc_tr=history_vgg.history['accuracy'][history_vgg.history['val_accuracy'].index(max(history_vgg.history['val_accuracy']))]
print('Accuracy on test set:',acc_test_vgg)
print('Accuracy on validation set:',acc_val)
print('Accuracy on train set:',acc_tr)
print('Loss function on test set:',loss_test_vgg)
```

```
26/26 [==============================] - 22s 835ms/step - loss: 0.8120 - accuracy: 0.7747
Accuracy on test set: 0.7747252583503723
Accuracy on validation set: 0.7875458002090454
Accuracy on train set: 0.7447587847709656
Loss function on test set: 0.8120282888412476
```

-

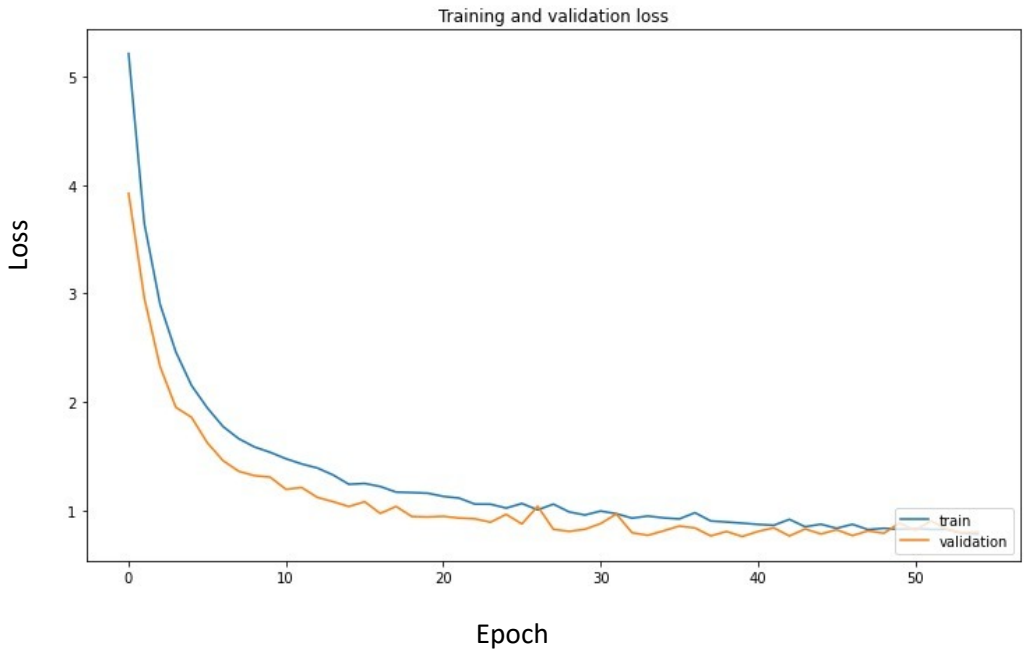
Training: 74%, **Test:** 77%, **Validation:** 79%,

when split the data set by:

60% Train, 20% Test, 20% Validation.

-

❖



2.3. Xception

❖ **Accuracy**

```
loss_test_vgg, acc_test_vgg = custom_xception_model.evaluate_generator(test_data_gen, verbose=1)
acc_val=history_xception.history['val_accuracy'][history_xception.history['val_accuracy'].index(max(history_xception.history['val_acc
acc_tr=history_xception.history['accuracy'][history_xception.history['val_accuracy'].index(max(history_xception.history['val_accuracy

print('Accuracy on test set:',acc_test_vgg)
print('Accuracy on validation set:',acc_val)
print('Accuracy on train set:',acc_tr)
print('Loss function on test set:',loss_test_vgg)
```

```
26/26 [==============================] - 25s 957ms/step - loss: 0.6289 - accuracy: 0.8333
Accuracy on test set: 0.8333333134651184
Accuracy on validation set: 0.8479853272438049
Accuracy on train set: 0.8137593865394592
Loss function on test set: 0.6288878917694092
```

❖

Training: 81%, **Test:** 83%, **Validation:** 85%,

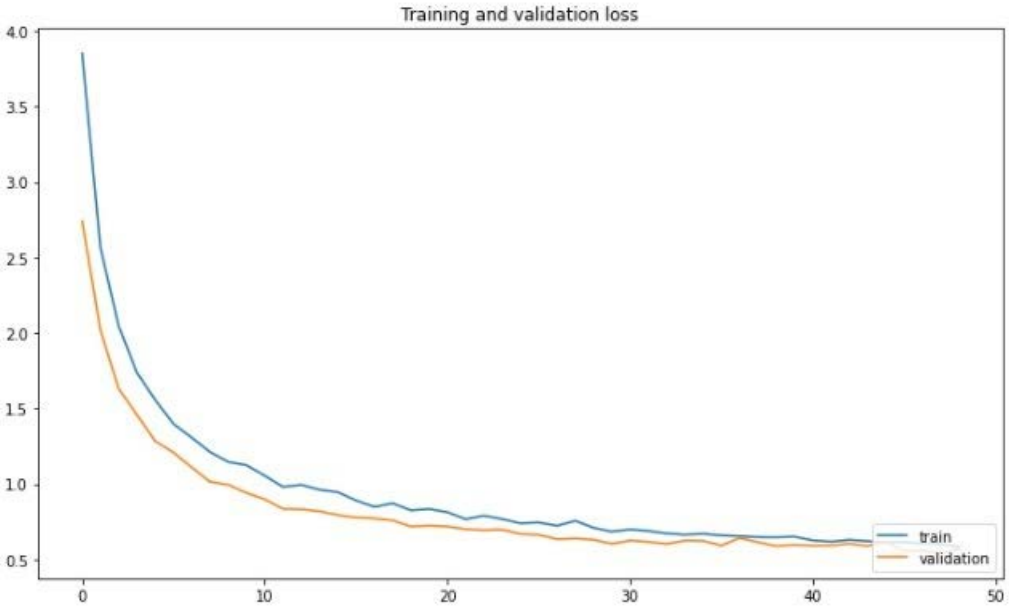
when split the data set by:

60% Train, 20% Test, 20% Validation.

❖


Training and validation accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
10
20
30
40
50
train
validation

❖


Training and validation loss
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
10
20
30
40
50
train
validation

Loss

Epoch

Summary

Split 1: 42% Train, 30% Test, 28% Validation

Model	Accuracy		
	Train	Validation	Test
VGG-16	80%	78%	77%
Xception	83%	82%	82%

Split 2: 60% Train, 20% Test, 20% Validation

Model	Accuracy		
	Train	Validation	Test
VGG-16	74%	79%	77%
Xception	81%	85%	83%

- When comparing pre-trained models, Xception model showed the best results in both options for splitting the data frame.
- At the same time, Xception achieves accuracy in validation of 70% by the tenth epoch, while it takes 20 epochs to achieve the same result from VGG-16 model.