# Project 2

# Flower Classification


| Name | ID | DEPARTMENT |
|---|---|---|
| Ahmed Mostafa | 20221372883 | AI |
| Mazen Gaber | 20221372110 | AI |


---

## 1 Problem Statement

We are given the petals to metals dataset, which is a dataset for flower classification. We're classifying 104 types of flowers based on their images drawn from five different public datasets.

Some classes are very narrow, containing only a particular sub-type of flower (e.g. pink primroses) while other classes contain many sub-types (e.g. wild roses).

## 2 Data

First, We downloaded the dataset from the class's Microsoft Teams

Then we Specified the indices of the classes to visualize and Set 3 images to display per class

We Read the classes from the Classes.txt file and extracted the class names to display them

For Example,Here is a sample of output:

'monkshood'

'purple coneflower'

Then, we counted the number of samples in each class and displayed them ten by ten with their names

For Example,Here is a sample of output:

```
Class:  00 - 09
           'snapdragon' - Number of Samples: 263
Class:         - Number of Samples: 227
Class: colt's foot - Number of Samples: 21
Class:                'king protea' - Number of Samples: 55
Class:       'spear thistle' - Number of Samples: 50
Class:  'yellow iris' - Number of Samples: 90
Class:        'globe-flower' - Number of Samples: 26
Class:          'purple coneflower' - Number of Samples: 19
Class:         'peruvian lily' - Number of Samples: 96
Class:     'balloon flower' - Number of Samples: 48
Class:    'giant white arum lily' - Number of Samples: 19
Class:  10 - 19
           'fire lily' - Number of Samples: 83
Class:         'pincushion flower' - Number of Samples: 21
Class:          'fritillary' - Number of Samples: 34
Class:        'red ginger' - Number of Samples: 119
Class:     'grape hyacinth' - Number of Samples: 109
Class:     'corn poppy' - Number of Samples: 105
Class:            'prince of wales feathers' - Number of Samples: 24
Class:  'stemless gentian' - Number of Samples: 23
Class:  'artichoke' - Number of Samples: 20
Class:         'sweet william' - Number of Samples: 18
```

Then, we made some transformations to data

- The transformations include resizing the images to (192, 192), applying random horizontal flip, random rotation, and color with a certain probability
- The transformed images are converted to tensors and normalized using mean and standard deviation values

After that we Split the training dataset into 90% for training and 10% for validation

We made transformations for test data which include resizing the images to (192, 192), converting them to tensors, and normalizing them using mean and standard deviation values.

# 3 Build CNN Models

## -Simple model:

*The first model we will make is a Simple CNN Model.*

- We will first create it by passing the number of classes as an argument and initialize the model with parameters

- We will send the model to GPU because if we send it to GPU the computations will be faster

- Now we define the loss function and the optimizer, we will chose the loss function to be cross entropy, the optimzer to be stochastic gradient descent with learning rate 0.001.

- We will now train the model. We will select a total of 15 epochs

-For each batch of inputs and labels in the training data, we will do these steps:

--Send the inputs and labels to the device

--Zero the gradients to reset the gradients from the previous batch.

--Perform a forward pass to compute the outputs

--Compute the predicted labels by selecting the class with the highest probability.

--Calculate the loss between the predicted labels and the true labels.

--Perform backpropagation to compute the gradients of the loss with respect to the model parameters.

--Update the model parameters by taking an optimizer step, adjusting them based on the gradients and learning rate.

- Then we will calculate the epoch loss and accuracy based on the running loss and correct predictions.

- The validation loss and accuracy are calculated by iterating over the validation data without calculating gradients.

- We will print the results and then save the final model after running through all the epochs

- For each epoch, the epoch number, training loss, training accuracy, validation loss, and validation accuracy are printed.

- Now we need to save this model after running through the iterations, so we will save it a file named 'simple_cnn.pth'

```
Epoch: 1/15 ==> Train Loss: 3.7390, Acc: 0.1235 Validation Loss: 3.3424, Acc: 0.1865
Epoch: 2/15 ==> Train Loss: 3.1505, Acc: 0.2119 Validation Loss: 3.0912, Acc: 0.2053
Epoch: 3/15 ==> Train Loss: 2.8695, Acc: 0.2613 Validation Loss: 2.8925, Acc: 0.2555
Epoch: 4/15 ==> Train Loss: 2.6514, Acc: 0.3102 Validation Loss: 2.8213, Acc: 0.2680
Epoch: 5/15 ==> Train Loss: 2.4595, Acc: 0.3501 Validation Loss: 2.8105, Acc: 0.2751
Epoch: 6/15 ==> Train Loss: 2.3096, Acc: 0.3854 Validation Loss: 2.6730, Acc: 0.3025
Epoch: 7/15 ==> Train Loss: 2.1094, Acc: 0.4314 Validation Loss: 2.6373, Acc: 0.3229
Epoch: 8/15 ==> Train Loss: 1.6908, Acc: 0.5487 Validation Loss: 2.4901, Acc: 0.3589
Epoch: 9/15 ==> Train Loss: 1.5849, Acc: 0.5809 Validation Loss: 2.4847, Acc: 0.3652
Epoch: 10/15 ==> Train Loss: 1.5499, Acc: 0.5874 Validation Loss: 2.5057, Acc: 0.3621
Epoch: 11/15 ==> Train Loss: 1.4932, Acc: 0.6055 Validation Loss: 2.5245, Acc: 0.3652
Epoch: 12/15 ==> Train Loss: 1.4612, Acc: 0.6090 Validation Loss: 2.5043, Acc: 0.3754
Epoch: 13/15 ==> Train Loss: 1.4235, Acc: 0.6202 Validation Loss: 2.5119, Acc: 0.3793
Epoch: 14/15 ==> Train Loss: 1.3780, Acc: 0.6372 Validation Loss: 2.5569, Acc: 0.3621
Epoch: 15/15 ==> Train Loss: 1.3179, Acc: 0.6562 Validation Loss: 2.5179, Acc: 0.3762
```

As we can see from the results, the training and validation loss decreased while the accuracy increased which means the model is performing well

## - Famous CNN Architectures:

### (1) VGG16 Model.

- We will use the torchvision library to load a pretrained VGG16 Model.

- We now will match the number of classes of the flower dataset to the model

- Now we define the loss function and the optimizer, we will chose the loss function to be cross entropy, the optimzer to be stochastic gradient descent with learning rate 0.001

- Now we train the model. We will choose the same number of epochs we chose for the previous model which is 15

-For each batch of inputs and labels in the training data, we will do this:

--Send the inputs and labels to the device

--Zero the gradients to prepare for the new batch

--Perform a forward pass to compute the outputs

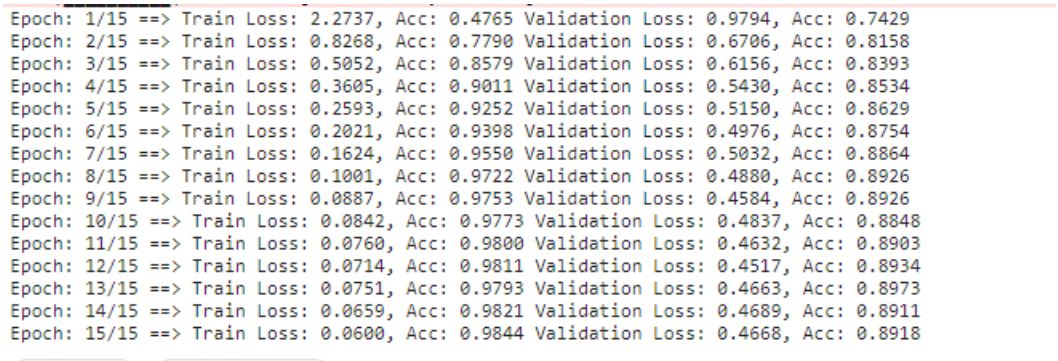
--Compute the predicted labels by selecting the class with the highest probability

--Calculate the loss between the predicted labels and the true labels

--Perform backpropagation to compute the gradients of the loss with respect to the model parameters

--Update the model parameters by taking an optimizer step, adjusting them based on the computed gradients and learning rate

- Then we will calculate the epoch loss and accuracy based on the running loss and correct predictions.
- The validation loss and accuracy are calculated by iterating over the validation data without calculating gradients.
- We will print the results and then save the final model after running through all the epochs
- For each epoch, the epoch number, training loss, training accuracy, validation loss, and validation accuracy are printed.
- Now we need to save this model after running through the iterations, so we will save it a file named 'fine_tuned_vgg16.pth'




As we see from results both train and validation loss decreases and accuracy increases which means the model is performing well

## (2) GoogLeNet:

- We will use the torchvision library to load a pretrained googlenet model.

- Now we match the number of classes of the flower dataset to the model

- We define the loss function to be cross entropy, the optimizer to be stochastic gradient desent with learning rate 0.001

- Now we train the model. We will choose the same number of epochs which is 15

-For each batch of inputs and labels in the training data, we will do this:

--Send the inputs and labels to the device

--Zero the gradients to prepare for the new batch

--Perform a forward pass to compute the outputs

--Compute the predicted labels by selecting the class with the highest probability

--Calculate the loss between the predicted labels and the true labels

--Perform backpropagation to compute the gradients of the loss with respect to the model parameters

--Update the model parameters by taking an optimizer step, adjusting them based on the computed gradients and learning rate

•Then we will calculate the epoch loss and accuracy based on the running loss and correct predictions

•The validation loss and accuracy are calculated by iterating over the validation data without calculating gradients

•We will print the results and then save the final model after running through all the epochs

•For each epoch, the epoch number, training loss, training accuracy, validation loss, and validation accuracy are printed

•Now we need to save this model after running through the iterations, so we will save it a file named 'fine_tuned_googlenet.pth'

```
Epoch: 1/15 ==> Train Loss: 3.5590, Acc: 0.2472 Validation Loss: 2.8515, Acc: 0.3918
Epoch: 2/15 ==> Train Loss: 2.4486, Acc: 0.4546 Validation Loss: 2.1001, Acc: 0.5353
Epoch: 3/15 ==> Train Loss: 1.8256, Acc: 0.5984 Validation Loss: 1.6131, Acc: 0.6387
Epoch: 4/15 ==> Train Loss: 1.4325, Acc: 0.6819 Validation Loss: 1.3128, Acc: 0.6951
Epoch: 5/15 ==> Train Loss: 1.1594, Acc: 0.7353 Validation Loss: 1.1036, Acc: 0.7187
Epoch: 6/15 ==> Train Loss: 0.9644, Acc: 0.7749 Validation Loss: 0.9739, Acc: 0.7516
Epoch: 7/15 ==> Train Loss: 0.8197, Acc: 0.8085 Validation Loss: 0.8746, Acc: 0.7782
Epoch: 8/15 ==> Train Loss: 0.7201, Acc: 0.8348 Validation Loss: 0.8469, Acc: 0.7868
Epoch: 9/15 ==> Train Loss: 0.6981, Acc: 0.8396 Validation Loss: 0.8628, Acc: 0.7868
Epoch: 10/15 ==> Train Loss: 0.6900, Acc: 0.8415 Validation Loss: 0.8445, Acc: 0.7861
Epoch: 11/15 ==> Train Loss: 0.6742, Acc: 0.8466 Validation Loss: 0.8408, Acc: 0.7829
Epoch: 12/15 ==> Train Loss: 0.6672, Acc: 0.8468 Validation Loss: 0.8218, Acc: 0.7868
Epoch: 13/15 ==> Train Loss: 0.6602, Acc: 0.8475 Validation Loss: 0.8317, Acc: 0.7884
Epoch: 14/15 ==> Train Loss: 0.6441, Acc: 0.8514 Validation Loss: 0.8221, Acc: 0.7908
Epoch: 15/15 ==> Train Loss: 0.6339, Acc: 0.8600 Validation Loss: 0.7974, Acc: 0.8017
```

As we see from results both train and validation loss decreases and accuracy increases which means the model is performing well

## - Ensemble step

*For the ensemble step, we will use 2 models, the VGG16 model and the GoogLeNet model*

- We will set both the VGG and GoogLeNet models to evaluation mode

- Then now we create an empty list to store the ensemble predictions

- Now iterate over the data loader to process batches of inputs and labels

- Then send the inputs to the device

- Then make predictions for both models by passing the inputs to each model

- Now we calculate the average from the 2 models (adding them both and dividing by 2)

- Now we need to add the average predictions to the empty list that we created

- Now we will concatenate the ensemble predictions into one tensor, evaluate the ensemble model on the validation set, and reshaping the labels to match the shape of the ensemble predictions

- Then we will calculate the loss between the ensemble predictions and the labels

- We now want to find the predicted labels, to do this we will select the class with the highest probability from the ensemble predictions

- After that we calculate the accuracy of the ensemble predictions by comparing them to the true labels

- The last thing is to print the training loss and accuracy, and the validation loss and accuracy of the ensemble model

```
Train Loss: 0.6339, Acc: 0.8600          Validation Loss: 0.4177, Acc: 0.8918
```

After ensembling both models, we achieved a lower validation loss and a higher validation accuracy, which means the ensembled model achieved the best performance

# 4 Big Picture

*In this step, we will compare between the performance of the models.*

**First, we will do the following :**

Evaluate the performance of the three models (SimpleCNN, VGG16, and GoogLeNet) on the test

dataset. We will load the saved weights for each model, evaluate their predictions using the test data, and compute accuracy and macro F1 score as performance metrics. Then we print the result

These are the steps we did to do that:

- First, we will create a function to evaluate the models
- Now we set the models and load the saved weights
- Then send models to the device
- Now we start to evaluate each model using the function we created. For each model, the function will return the true labels and predicted labels for the testing data. After that we can calculate the Accuracy and F1 score

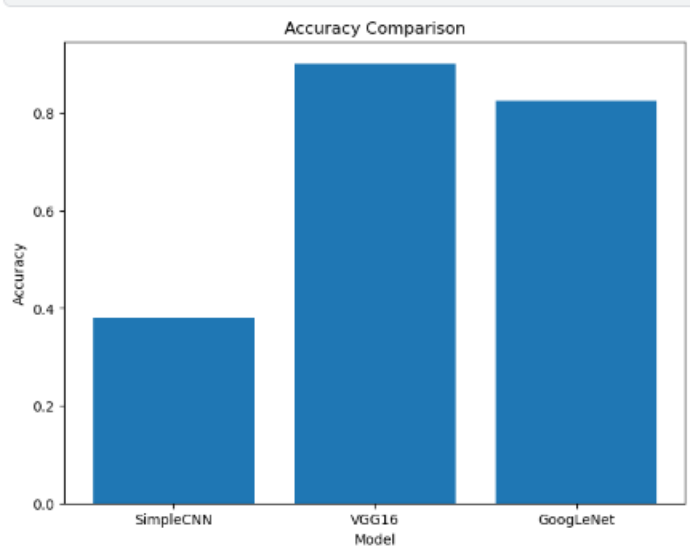
- And the last step is to print the results for each model

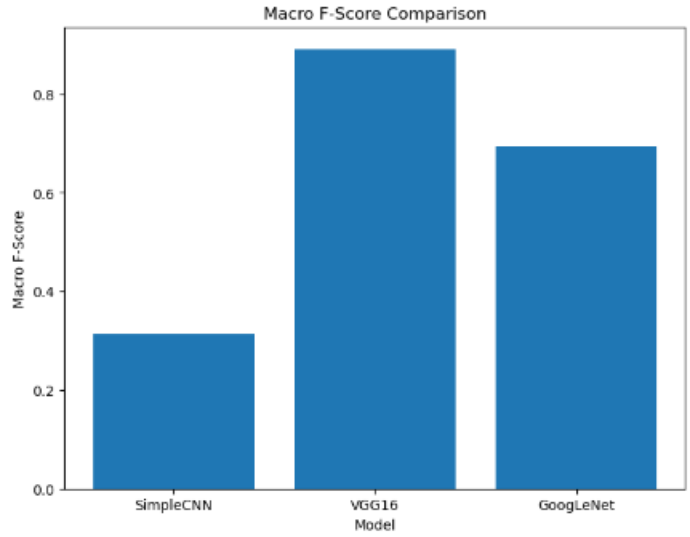
```
SimpleCNN:
Accuracy: 0.3809267241379310 5
Macro F-Score: 0.3125059830175533 3

VGG16:
Accuracy: 0.9011314655172413
Macro F-Score: 0.8909282823731235

GoogLeNet:
Accuracy: 0.8240840517241379
Macro F-Score: 0.6941068550388896
```

# Plots for the performance results obtained in the evaluation part:




# Lets explain and analyze these results:

**Accuracy**: Between the three models, VGG16 achieved the highest accuracy of 0.901131465517241 3, and then the GoogLeNet with an accuracy of 0.824084051724137 9, then the SimpleCNN had the lowest accuracy of 0.380926724137931 05

**Macro F-Score**: Again, VGG16 had the highest Macro F-Score of 0.890928282373123 5, then the GoogLeNet with a Macro F-Score of 0.694106855038889 6, then the SimpleCNN had the lowest Macro F-Score of 0.312505983017553 33

*These results tell us that the VGG16 had the best performance in both accuracy and Macro F-Score, while the SimpleCNN had the lowest performance in both accuracy and Macro F-Score.*

**Second, we will plot the confusion matrices and find the most confusing class:**

- First, define a function to plot a confusion matrix and determine the most confusing class

- The confusion matrix is the counts of true labels versus predicted labels

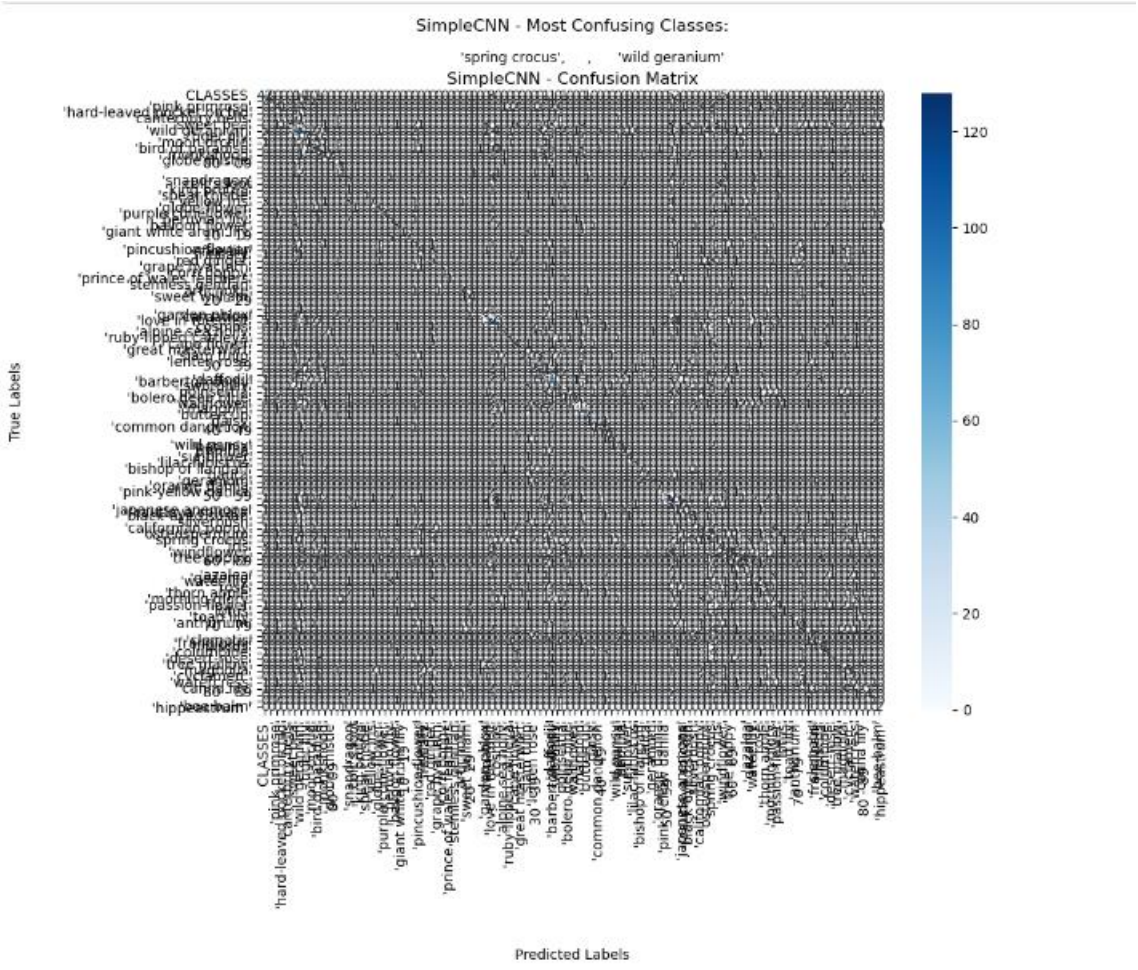
- We will calculate the misclassified counts by subtracting the diagonals (which are the correctly classified) from the confusion matrix and adding the remaining values along the rows

- To find the most confusing class, we need to find the index of the maximum misclassified count and get the class name from the list of the class names
- Then we now visualize the confusion matrix

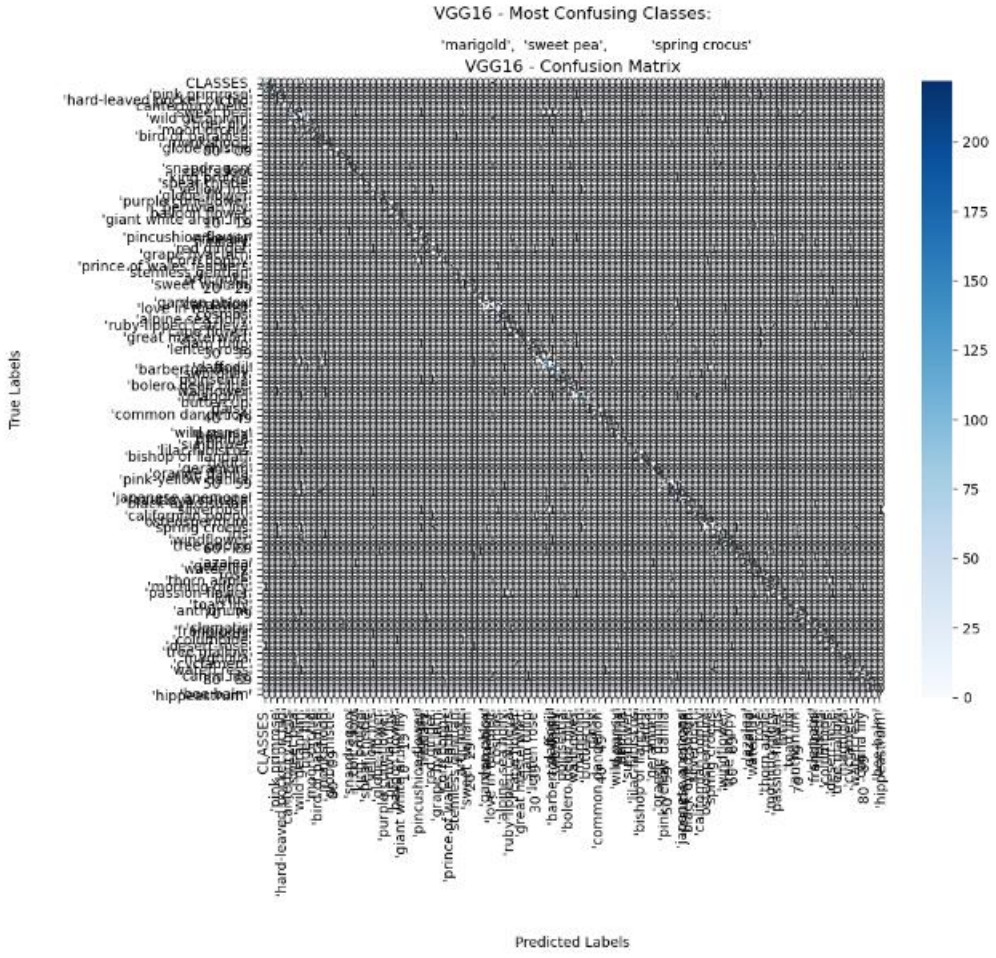
- The x-axis is the predicted labels, the y-axis is the true labels (we will assign the class names to both

the x-axis and y-axis labels to see the classes in the confusion matrix)

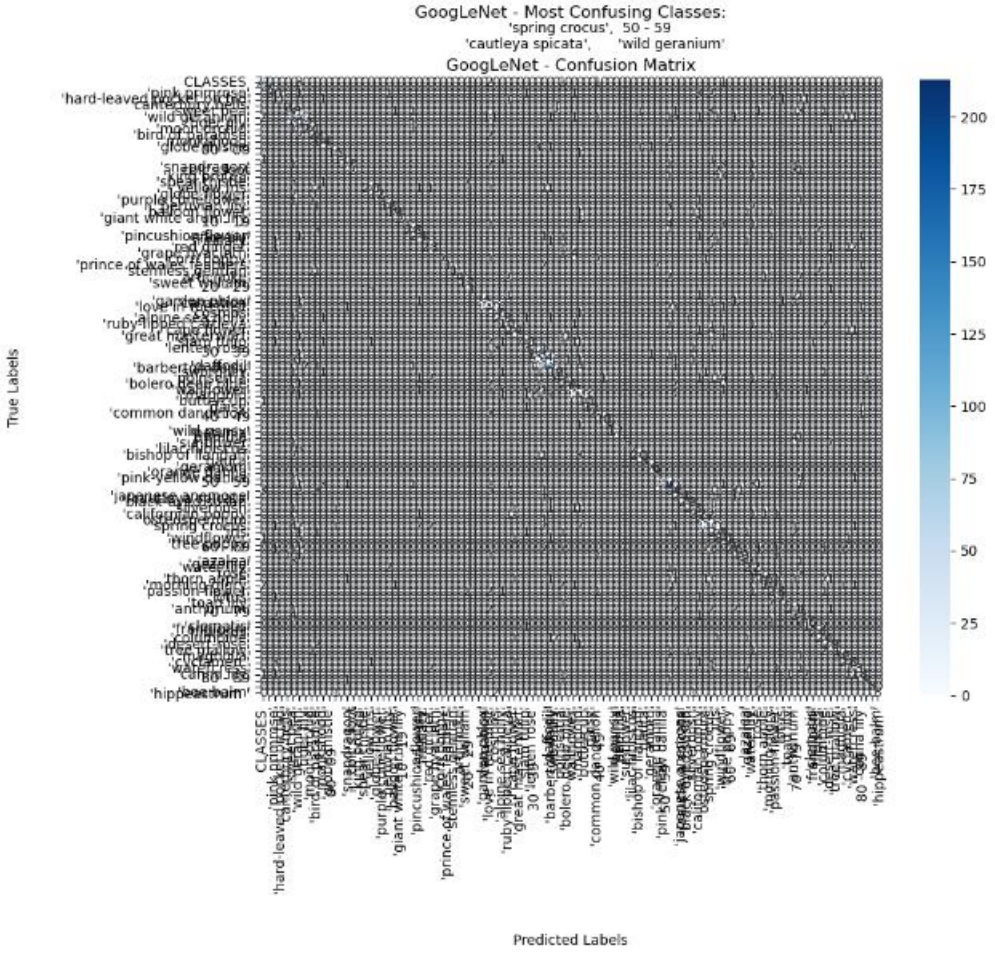
- Now we set the title of the plot to show the model name and the most confusing class
- After this the plots will be shown for each model



We can see the confusion matrix for the SimpleCNN
The most confusing classes were spring crocus and wild geranium




For the VGG16, the most confusing classes were marigold, sweet pea, and spring crocus




For the GoogLeNet, the most confusing were spring crocus, cautleya spicata, and wild geranium

The most confusing classes in each model means which classes are misclassified a lot

## Now we want to see some success cases and some failure cases, in order to do that, we will do some steps:

- We will make a function that takes the model and data loader as inputs. The function will evaluate the model on the data loader then return the true labels and predicted labels, and the images for each of the labels

- Set the model weight's to the ones that we saved in the path for each model, then send it to the device

- We will evaluate each model by passing it and the test data loader in the function we created

After doing these steps for each model, the weights of each model will be loaded then will be evaluated on the

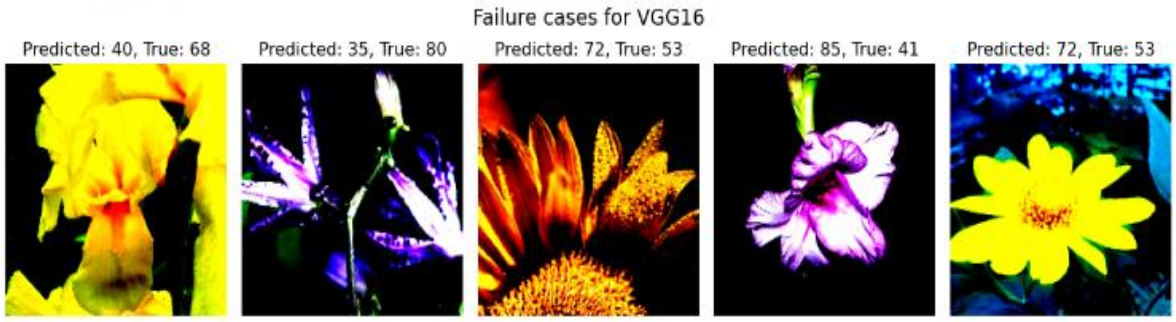
test data, then we will get the true and predicted labels for the images

**Here are some success cases and some failure cases for the SimpleCNN:**

For the VGG:

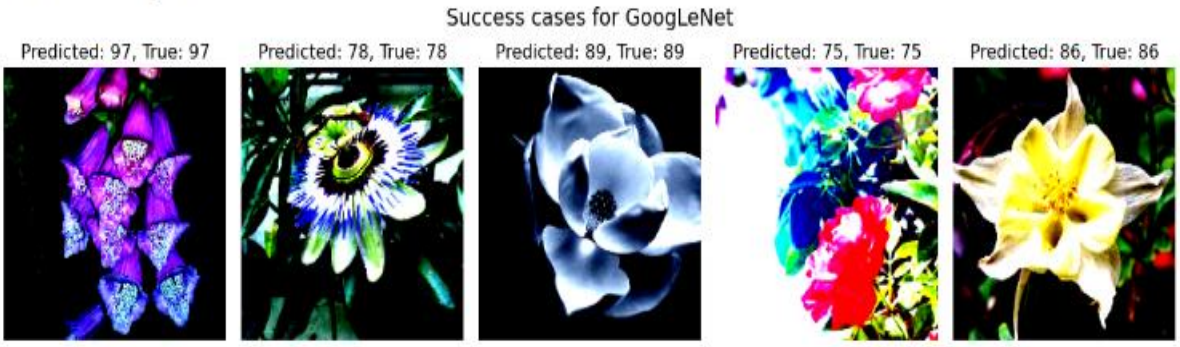
Success cases for VGG16:


Failure cases for VGG16:



# For the GoogLeNet:

Success cases for GoogLeNet:




Failure cases for GoogLeNet:


==========